

O'REILLY®

Second
Edition

Infrastructure as Code

Dynamic Systems for the Cloud Age

Early
Release
RAW &
UNEDITED



Kief Morris

1. Preface

- a. How I Learned to Stop Worrying and to Love the Cloud
 - i. The Sorcerer’s Apprentice
 - ii. Cloud from Scratch
 - iii. Legacy cloud infrastructure
- b. Why I Wrote This Book
- c. Why A Second Edition
- d. Who This Book Is For
- e. What Tools Are Covered
- f. Principles, Practices, and Patterns
- g. The FoodSpin examples
- h. Conventions Used in This Book
- i. O’Reilly Online Learning
- j. How to Contact Us

2. I. Foundations

3. 1. What is Infrastructure as Code?

- a. From the Iron Age to the Cloud Age
- b. Infrastructure as Code
 - i. Benefits of Infrastructure as Code
- c. Use Infrastructure as Code to optimize for change
 - i. Objection: “We don’t make changes often enough to justify automating

them”

ii. Objection: “We should build first and automate later”

iii. Objection: “We must choose between speed and quality”

d. Three core practices for Infrastructure as Code

i. Core practice: Define everything as code

ii. Core practice: Continuously validate all your work in progress

iii. Core practice: Build small, simple pieces that you can change independently

e. The parts of an infrastructure system

f. Conclusion

4. 2. Principles of Cloud Age Infrastructure

a. Principle: Assume systems are unreliable

b. Principle: Make everything reproducible

c. Pitfall: Snowflake systems

d. Principle: Create disposable things

e. Principle: Minimize variation

i. Configuration Drift

f. Principle: Ensure that you can repeat any process

g. Conclusion

5. 3. Infrastructure Platforms

a. What is a dynamic infrastructure platform?

b. Infrastructure Resources

c. Compute Resources

- i. Virtual machines
- ii. Physical servers
- iii. Containers
- iv. Server clusters
- v. Serverless code execution (FaaS)

d. Storage Resources

- i. Block storage (virtual disk volumes)
- ii. Object storage
- iii. Networked filesystems (shared network volumes)
- iv. Structured data storage
- v. Secrets management

e. Network Resources

- i. Network address blocks
- ii. Traffic management and routing
- iii. Network access rules
- iv. Caches
- v. Service meshes

f. Conclusion

6. 4. Core Practice: Define everything as code

- a. Why you should define your infrastructure as code
- b. What you can define as code

i. Choose tools that are configured with code

ii. Manage your code in a version control system

iii. Secrets and source code

c. Infrastructure coding languages

i. Scripting your infrastructure

ii. Building infrastructure with declarative code

iii. DSLs for infrastructure

iv. The return of general-purpose languages for infrastructure

d. Implementation Principles for defining infrastructure as code

i. Implementation Principle: Avoid mixing different types of code

ii. Implementation Principle: Separate infrastructure code concerns

iii. Implementation Principle: Treat infrastructure code like real code

e. Conclusion

7. II. Working With Infrastructure Stacks

8. 5. Building Infrastructure Stacks as Code

a. What is an infrastructure stack?

i. Stack code

ii. Stack instance

iii. Configuring servers in a stack

b. Patterns and antipatterns for structuring stacks

- i. Antipattern: Monolithic Stack
- ii. Pattern: Application Group Stack
- iii. Pattern: Service Stack
- iv. Pattern: Micro Stack

c. Conclusion

9. 6. Using Modules to Share Stack Code

- a. Examples of using modules
- b. Patterns and antipatterns for infrastructure modules
 - i. Pattern: Facade Module
 - ii. Antipattern: Anemic Module
 - iii. Pattern: Domain Entity Module
 - iv. Antipattern: Spaghetti Module
 - v. Antipattern: Obfuscation Layer
 - vi. Antipattern: One-shot Module

c. Conclusion

10. 7. Building Environments With Stacks

- a. What environments are all about
 - i. Release delivery environments
 - ii. Multiple production environments
 - iii. Environments, consistency, and configuration
- b. Patterns for building environments

- i. Antipattern: Multiple-Environment Stack
- ii. Antipattern: Copy-Paste Environments
- iii. Pattern: Reusable Stack
- c. Building environments with multiple stacks
- d. Conclusion

11. 8. Configuring Stacks

- a. Using stack parameters to create unique identifiers
- b. Example stack parameters
 - i. Handling secrets as parameters
- c. Patterns for configuring stacks
 - i. Antipattern: Manual Stack Parameters
 - ii. Pattern: Stack Environment Variables
 - iii. Pattern: Scripted Parameters
 - iv. Pattern: Stack Configuration Files
 - v. Pattern: Wrapper Stack
 - vi. Pattern: Pipeline Stack Parameters
 - vii. Pattern: Stack Parameter Registry
- d. Configuration Registry
 - i. Implementing a Configuration Registry
 - ii. Single or multiple configuration registries
 - iii. Configuration Management Database (CMDB)

e. Conclusion

12. 9. Core Practice: Continuously validate all work in progress

a. Why continuously validate infrastructure code?

- i. What continuous validation means
- ii. What should we validate with infrastructure?

b. Challenges with testing infrastructure code

- i. Challenge: Tests for declarative code often have low value
- ii. Challenge: Testing infrastructure code is slow

c. Progressive validation

- i. Validation stages
- ii. Testing in production

d. Progressive validation models

- i. Test pyramid
- ii. Swiss cheese testing model

e. Pipelines for validation

- i. Pipeline stages
- ii. Delivery pipeline software and services

f. Conclusion

13. 10. Testing Infrastructure Stacks

a. Example infrastructure

- i. The example stack

ii. Pipeline for the example stack

b. Offline validation stages for stacks

i. Syntax checking

ii. Offline static code analysis

iii. Static code analysis with API

iv. Testing with mock API

c. Online validation stages for stacks

i. Preview: Seeing what changes will be made

ii. Verification: Making assertions about infrastructure resources

iii. Outcomes: Proving infrastructure works correctly

d. Using test fixtures to handle dependencies

i. Test doubles for upstream dependencies

ii. Test fixtures for downstream dependencies

e. Lifecycle patterns for test instances of stacks

i. Pattern: Persistent test stack

ii. Pattern: Ephemeral test stack

iii. Antipattern: Dual Persistent and Ephemeral Stack Stages

iv. Pattern: Periodic stack rebuild

v. Pattern: Continuous stack reset

f. Test orchestration

i. Support local testing

ii. Avoid tight coupling with pipeline tools

iii. Test orchestration tools

g. Conclusion

Infrastructure as Code

SECOND EDITION

Evolving systems in the Cloud

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Kief Morris

Infrastructure as Code

by Kief Morris

Copyright © 2020 Kief Morris. 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>). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

Editors: John Devins and Virginia Wilson

Production Editor: Christopher Faucher

Cover Designer: Karen Montgomery

June 2016: First Edition

July 2020: Second Edition

Revision History for the Early Release

- 2019-12-03: First Release
- 2020-01-22: Second Release

See <http://oreilly.com/catalog/errata.csp?isbn=9781098114671> for release details.

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

While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author 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.

978-1-098-11460-2

Preface

Modern organizations are increasingly using dynamic cloud platforms, whether public or private, to exploit digital technology to deliver services and products to their users. Automation is essential to managing continuously changing and evolving systems, and tools that define systems “as code” have become dominant for this.

Although cloud technology and infrastructure coding tools are becoming pervasive, most teams are still learning the best ways to put them to use.

This book is my attempt to share what I’ve learned from various people, teams, and organizations. I’m not giving you specific instructions on how to implement a specific tool or language. Instead, I’ve assembled principles, practices, and patterns that you can use to shape how you approach the design, implementation, and evolution of your infrastructure.

These draw heavily on agile engineering principles and practices. I believe that given cloud-based infrastructure is essentially just another software system, we can benefit from lessons learned in other software domains.

Infrastructure as code has grown along with the DevOps movement. Andrew Clay-Shafer and Patrick Debois triggered the DevOps movement with a talk at the Agile 2008 conference. The first uses I’ve found for the term “Infrastructure as Code” are from

a talk on [Agile Infrastructure](#) that Clay-Shafer gave at the Velocity conference in 2009, and [an article](#) John Willis wrote summarizing the talk¹. My own Infrastructure as Code journey began a decade or so before this.

How I Learned to Stop Worrying and to Love the Cloud

I set up my first server, a dialup BBS,² in 1992. This led to Unix system administration and then to building and running hosted applications for various companies, from startups to enterprises. In 2001 I discovered the [Infrastructures.org](#) website, which taught me how to build servers in a highly consistent way using CFEngine³

By 2007, my team had accumulated around 20 1U and 2U physical servers in our office's server racks. These overflowed with test instances of our company's software, along with a variety of miscellaneous applications and services - a couple of wikis, bug trackers, DNS servers, mail servers, databases, and so on.

Whenever someone found something else to run, we crammed it onto an existing server. When the product folks wanted a new environment, it took several weeks to order and assemble the hardware.

Then we learned that virtualization was a thing. We started with a pair of beefy HP rack servers and VMWare ESX Server licenses, and before long, everything was a VM. Every application could run on a dedicated VM, and it took minutes to create a new environment.

Virtualization made it easy to create new servers, which solved one set of problems. But it led to a whole new set of problems.

The Sorcerer's Apprentice

A year later, we were running well over 100 VMs and counting. We were well underway with virtualizing our production servers and experimenting with Amazon's new cloud hosting service. The benefits virtualization had brought to the business people meant we had money for more ESX servers and for shiny SAN devices to feed our infrastructure's shocking appetite for storage.

We created virtual servers, then more, then even more. They overwhelmed us. When something broke, we tracked down the VM and fixed whatever was wrong with it, but we couldn't keep track of what changes we'd made where. We felt like Mickey Mouse in "The Sorcerer's Apprentice" from *Fantasia* (an adaption of the von Goethe poem).

*Well, a perfect hit!
See how he is split!
Now there's hope for me,
and I can breathe free!*

*Woe is me! Both pieces
come to life anew,
now, to do my bidding
I have servants two!
Help me, O great powers!
Please, I'm begging you!*

—Excerpted from Brigitte Dubiel's translation of
“Der Zauberlehrling” (“The Sorcerer’s
Apprentice”) by Johann Wolfgang von Goethe

We faced a never-ending stream of updates to the operating systems, web servers, application servers, database servers, JVMs, and other software packages we used. We struggled to keep up with them. We might apply them successfully to some VMs, but on others, the upgrades broke things. We didn't have time to stomp out every incompatibility, and over time had many combinations of versions of things strewn across hundreds of VMs.

We were using configuration automation software even before we virtualized, which should have helped with these issues. I had used CFEngine in previous companies, and when I started this team, we tried a new tool called Puppet. Later, my colleague introduced us to Chef when he spiked out ideas for an AWS infrastructure. All of these tools were useful, but particularly in the early days, they didn't get us out of the quagmire of wildly different servers.

In theory, we should have configured our automation tools to run continuously, applying and reapplying the same configuration to all of our servers every hour or so. But we didn't trust them enough to let them run unattended. We had too much variation across our servers, and it was too easy for something to break without us noticing. We would write a Puppet manifest to configure and manage a particular application server. But when we ran it against a different server, we found that it had a different version of Java, application server software, or OS packages. The Puppet run would fail, or worse, corrupt the application server.

So we ended up using Puppet ad hoc. We could safely run it against new VMs, although we might need to make some tweaks to make it work. We would write a new manifest to carry out a

specific upgrade, and then run it against our servers one at a time, carefully checking the result and making fixes as needed.

Configuration automation was a useful aid, better than shell scripts, but the way we used it didn't save us from our sprawl of inconsistent servers.

Cloud from Scratch

Things changed when we began moving things onto the cloud. The technology itself wasn't what improved things; we could have done the same thing with our own VMware servers. But because we were starting fresh, we adopted new ways of managing servers based on what we had learned with our virtualized farm. We also followed what people were doing at companies like Flickr, Etsy, and Netflix. We baked these new ideas into the way we managed services as we migrated them onto the cloud.

The key idea of our new approach was that we could rebuild every server automatically from scratch. Our configuration tooling would run continuously, not ad hoc. Every server added into our new infrastructure would fall under this approach. If automation broke on some edge case, we would either change the automation to include it or else fix the design of the service so that it was no longer an edge case.

The new regime wasn't painless. We had to learn new habits, and we had to find ways of coping with the challenges of a highly automated infrastructure. As the members of the team moved on to other organizations and got involved with communities such as DevOpsDays, we learned and grew. Over time, we reached the point where we were habitually working with automated

infrastructures with hundreds of servers, with much less effort and headache than we had been in our “Sorcerer’s Apprentice” days.

Joining ThoughtWorks was an eye-opener for me. The development teams I worked with were passionate about using XP engineering practices like test-driven development (TDD), continuous integration (CI) and continuous delivery (CD). Because I had already learned to manage infrastructure scripts and configuration files in source control systems, it was natural to apply these rigorous engineering practices to them.

Working with ThoughtWorks has also brought me into contact with many IT operations teams. Over the years, I’ve seen organizations experimenting with and then adopting virtualization, cloud, containers, and automation tooling. Working with them to share and learn new ideas and techniques has been a fantastic experience.

Legacy cloud infrastructure

In the past few years, I’ve seen more teams who have moved beyond early adoption of cloud-based infrastructures using “as-code” tools. My ThoughtWorks colleagues and I have noticed some common issues with more mature infrastructure.

Many teams find that their infrastructure codebases have grown difficult to manage. Setting up a new environment for a new customer may take a month or more. Rolling out patches to a container orchestration system is a painful, disruptive process. New members of the team take too long to learn the complicated and messy set of scripts they use to run their infrastructure tools.

The interesting thing about these situations is that they look oddly familiar. For years, clients have asked us to help them with messy, monolithic software architectures, and error-prone application deployment processes. And now we're seeing the same issues appearing with infrastructure.

Turns out, infrastructure as code isn't just a metaphor. Code really is code!

So we've doubled down on the idea that software engineering practices can be useful with infrastructure codebases. TDD, CI, CD, microservices⁴, and Evolutionary Architectures[<http://shop.oreilly.com/product/0636920080237.do>] are coming into their own for cloud infrastructure.

Why I Wrote This Book

I've met and worked with many teams who are in the same place I was a few years ago: people who were using cloud, virtualization, and automation tools but hadn't got it all running as smoothly as they know they could. I hope that this book provides a practical vision and specific techniques for managing complex IT infrastructure in the cloud.

Why A Second Edition

I started working on the first edition of this book in 2014. The landscape of cloud and infrastructure was shifting as I worked, with innovations like Docker popping up, forcing me to add and revise what I was writing. I felt like I was in a race, with people in

the industry creating things for me to write about faster than I could write it.

Things haven't slowed down since we published the book in June 2016. Docker went from a curiosity to the mainstream, and Kubernetes emerged as the dominant way to orchestrate Docker containers. Existing distributed orchestration and PaaS (Platform as a Service) products either fell by the wayside or rebuilt themselves around Docker and Kubernetes. Serverless, service meshes, and observability are all emerging as core technologies for modern cloud-based systems.

Even as I write the second edition, a new generation of infrastructure tools is emerging, led (so far) by Pulumi and the AWS CDK (Cloud Development Kit). These are challenging the incumbent tools, using general-purpose procedural languages rather than dedicated declarative languages.

Nevertheless, it feels useful to update the book, bringing it up to the current state of the industry. Some books explain how to use a specific tool or language. The concepts in this book are relevant across tools, and even across new versions and types of tools. The core practices, principles, and most of the implementation patterns I describe are valid even if specific technologies and tools become obsolete.

I have several goals with this new edition:

Include newer technologies

Although the concepts apply regardless of the specific tools you use, it's useful to describe them in current contexts. People

often ask me how infrastructure as code applies with serverless, containers, and service meshes. This second edition should make this more clear.

Improve relevance to mature cloud systems

When I wrote the first edition, very few organizations were making full use of cloud and infrastructure as code. Since then, even conservative organizations like banks and governments have begun adopting public cloud. And many other teams have built up large infrastructure codebases, often accumulating technical debt and even legacy systems and code. I've updated the content in this book based on lessons I've learned working with these teams.

Evolve and expand

One of the benefits that I hadn't anticipated from writing a book was the amount of exposure I gained to different people building systems on the cloud. I've learned so much from giving talks, running workshops, visiting organizations at various phases of adoption, and working with clients. I've learned about pitfalls, good practices, and challenges from many people and teams. I've also learned how different ways of talking about this stuff resonates with people, so I've been able to hone my messaging. The result is that this edition of the book has more content and is presented in what I believe is a stronger structure.

Who This Book Is For

This book is for people who work with dynamic IT infrastructure. You may be a system administrator, infrastructure engineer, team lead, architect, or a manager with technical interest. You might also be a software developer who wants to build and use infrastructure.

I'm assuming you have some exposure to cloud infrastructure, so you know how to provision and configure systems. You've probably at least played with tools like Ansible, Chef, CloudFormation, Puppet, or Terraform.

While this book may introduce some readers to infrastructure as code, I hope people who work this way already will also find it interesting. I see it as a way to share ideas and start conversations about how to do it even better.

What Tools Are Covered

This book doesn't offer instructions in using specific scripting languages or tools. I generally use pseudo-code for examples and try to avoid making examples specific to a particular cloud platform. This book should be helpful to you regardless of whether you use CloudFormation and Ansible on AWS, Terraform, and Puppet on Azure, Pulumi and Chef on Google Cloud, or a completely different stack.

The concepts I explain are relevant across different platforms and toolchains. When I introduce a concept, I often give examples from specific tools to illustrate what I mean. The tools I do

mention are ones that I am most familiar with. But there are many other tools out there-just because I don't mention your favorite one doesn't mean I'm dismissing it, it only means I don't know enough about it.

Principles, Practices, and Patterns

I use the terms *principles*, *practices*, and *patterns* (and *antipatterns*) to describe essential concepts. Here are the ways I use each of these terms:

Principle

A principle is a rule that helps you to choose between potential solutions.

Practice

A practice is a way of implementing something. A given practice is not always the only way to do something, and may not even be the best way for a particular situation. You should use principles to guide you in choosing the most appropriate practice for a given situation.

Pattern

A pattern is a potential solution to a problem. It's very similar to a practice, in that different patterns may be more effective in different contexts. Each pattern is described in a format that should help you to evaluate how relevant it is for your problem.

Antipattern

An antipattern is a potential solution that I am recommending you avoid in most situations. Usually, it's either something that seems like a good idea or else it's something that you fall into doing without realizing it.

WHY I DON'T USE THE TERM "BEST PRACTICE"

Folks in our industry love to talk about "best practices". The problem with this term is that it often leads people to think there is only one solution to a problem, no matter what the context.

I prefer to describe practices and patterns, and note when they are useful and what their limitations are. I do describe some of these as more effective or more appropriate, but I try to be open to alternatives. For practices that I believe are less effective, I hope I explain why I think this.

Here are how I'm using these terms to drive the structure of this book:

Principles of Cloud Age Infrastructure

These are rules driven by the dynamic nature of cloud systems⁵ that help you decide how to approach building and running your stuff. They are a contrast to legacy approaches from the Iron Age of infrastructure, which assume resources are static and expensive to change. These principles lead us to Infrastructure as Code as an approach.

Core Practices for Infrastructure as Code

There are many practices for implementing infrastructure as code. The three I've highlighted as "core" are the ones that I believe are fundamental for successfully building infrastructure in the Cloud Age. In other words, the Principles of Cloud Age Infrastructure drive the use of these Core Practices.

Implementation Principles

Given each of the core practices for infrastructure as code, these principles are rules for implementing that practice. They help you to decide which patterns and approaches are most useful. I tend to list these in a chapter for each core practice.

Patterns (and antipatterns)

Potential solutions to designing, building, and running your systems. The decisions of which ones are appropriate are driven by the implementation principles.

The FoodSpin examples

I use the fictional company Foodspin to illustrate concepts throughout this book. Foodspin provides an online menu service for fast food restaurants. I doubt this would be a viable business, but it works as an example. The company has clients in different countries, with different cuisines, including Curry Hut in the UK, The Fish King in Australia, and Bomber Burrito and Burger Barn in North America.

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.

TIP

This element signifies a tip or suggestion.

NOTE

This element signifies a general note.

WARNING

This element indicates a warning or caution.

O'Reilly Online Learning

NOTE

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

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

How to Contact Us

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

O'Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

800-998-9938 (in the United States or Canada)

707-829-0515 (international or local)

707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at

[*http://bit.ly/infrastructureAsCode_1e*](http://bit.ly/infrastructureAsCode_1e).

To comment or ask technical questions about this book, send email to [*bookquestions@oreilly.com*](mailto:bookquestions@oreilly.com).

For more information about our books, courses, conferences, and news, see our website at [*http://www.oreilly.com*](http://www.oreilly.com).

Find us on Facebook: [*http://facebook.com/oreilly*](http://facebook.com/oreilly)

Follow us on Twitter: [*http://twitter.com/oreillymedia*](http://twitter.com/oreillymedia)

Watch us on YouTube: [*http://www.youtube.com/oreillymedia*](http://www.youtube.com/oreillymedia)

¹ Adam Jacob and Luke Kanies were also using the phrase around this time. Mark Burgess pioneered the practice of Infrastructure as Code with CFEngine before anyone even used the phrase.

2 Bulletin Board System.

3 Sadly, Infrastructures.org hasn't been updated since 2007, but the content was still there the last time I looked.

4 <http://shop.oreilly.com/product/0636920033158.do>

5 As I'll explain in Chapter 3, "cloud" isn't exclusive to public, shared cloud platforms like AWS, Azure, and GCP. The "Cloud Age" reaches into on-premise systems and private data centers as well.

Part I. Foundations

Chapter 1. What is Infrastructure as Code?

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

These technologies are especially relevant as organizations become digital. “Digital” is how people in business attire say that software systems are essential to what the organization does.¹ The move to digital increases the pressure on you to do more and to do it faster. You need to add and support more services. More business activities. More employees. More customers, suppliers, and other stakeholders.

Cloud and automation tools help by making it far easier to add and change infrastructure. But many teams struggle to find enough time to keep up with the infrastructure they already have. Making it easier to create even more stuff to manage is unhelpful. As one of my clients told me, “Using cloud knocked down the walls that kept our tire fire² contained.”

Many people respond to the threat of unbounded chaos by tightening their change management processes. They hope they can prevent chaos by limiting and controlling changes. So they wrap the cloud in chains.

There are two problems with this. One is that it removes the benefits of using cloud technology; the other is that users *want* the benefits of cloud technology. So users bypass the people who are trying to limit the chaos. In the worst cases, people completely ignore risk management, deciding it's not relevant in the brave new world of cloud. They embrace cowboy IT³, which adds different problems.

The premise of this book is that you can exploit cloud and automation technology to make changes easily, safely, quickly, and responsibly. These benefits don't come out of the box with automation tools or cloud platforms. They depend on the way you use this technology.

DEVOPS AND INFRASTRUCTURE AS CODE

DevOps is a movement to reduce barriers and friction between organizational silos - development, operations, and other stakeholders involved in planning, building, and running software. Although technology is the most visible, and in some ways simplest face of DevOps, it's culture, people, and processes which have the most impact on flow and effectiveness. Technology and engineering practices like infrastructure as code should be used to support efforts to bridge gaps and improve collaboration.

In this chapter, I explain that modern, dynamic infrastructure requires a “Cloud Age” mindset. This mindset is fundamentally different from the traditional, “Iron Age” approach we used with static pre-cloud systems. I define three Core Practices for implementing infrastructure as code: define everything as code, continuously validate everything as you work, and build your system from small, loosely-coupled pieces.

Also in this chapter, I describe the reasoning behind the Cloud Age approach to infrastructure. This approach discards the false dichotomy of trading speed off against quality. Instead, we use speed as a way to improve quality, and we use quality to enable delivery at speed.

I'll explain why and how this dynamic works:

- The differences between Iron Age systems and Cloud Age systems, in terms of both technology and ways of working,
- How this leads to Infrastructure as Code,
- Some common objections to going “all-in” on Infrastructure as Code,
- The parts of a dynamic infrastructure,
- The three core practices of Infrastructure as Code.

From the Iron Age to the Cloud Age

Cloud Age technologies make it faster to provision and change infrastructure than traditional, Iron Age technologies (Table 1-1).

Table 1-1. Technology changes in the Cloud Age

Iron Age	Cloud Age
Physical hardware	Virtualized resources
Provisioning takes weeks	Provisioning takes minutes
Manual processes	Automated processes

However, these technologies don't necessarily make it easier to manage and grow your systems. Moving a system with technical debt onto unbounded cloud infrastructure accelerates the chaos.

Maybe you could use well-proven, traditional governance models to control the speed and chaos that newer technologies unleash. Thorough up-front design, rigorous change review, and strictly segregated responsibilities will impose order!

Unfortunately, these models optimize for the Iron Age, where changes are slow and expensive. They add extra work up-front, hoping to reduce the time spent making changes later. This arguably makes sense when making changes later is slow and expensive. But cloud makes changes cheap and fast. You should exploit this speed to learn and improve your system continuously. Iron Age ways of working are a massive tax on learning and improvement.

Rather than using slow-moving Iron Age processes with fast-moving Cloud Age technology, adopt a new mindset. Exploit faster-paced technology to reduce risk and improve quality. Doing this requires a fundamental change of approach and new ways of thinking about change and risk (Table 1-2).

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

Iron Age	Cloud Age
Cost of change is high	Cost of change is low
Changes represent failure (changes must be “managed”, “controlled”)	Changes represent learning and improvement
Reduce opportunities to fail	Maximize speed of improvement
Deliver in large batches, test at the end	Deliver small changes, test continuously
Long release cycles	Short release cycles
Monolithic architectures (fewer, larger moving parts)	Microservices architectures (more, smaller parts)
GUI-driven or physical configuration	Configuration as Code

Infrastructure as Code is a Cloud Age approach to managing systems that embraces continuous change for high reliability and quality.

THE CLOUD AGE

Cloud Age technologies make it possible to rapidly provision and change infrastructure resources. However, you need to adopt Cloud Age ways of working to exploit the technology to deliver better reliability, security, and quality.

Cloud Age approaches are not only relevant when using shared public clouds, but are essential for modern on-premise and data center infrastructures as well.

Infrastructure as Code

Infrastructure as Code is an approach to infrastructure automation based on practices from software development. It emphasizes consistent, repeatable routines for provisioning and changing systems and their configuration. You make changes to code, then use automation to test and apply those changes to your systems.

Throughout this book, I explain how to use agile engineering practices such as Test Driven Development (TDD), Continuous Integration (CI), and Continuous Delivery (CD) to make changing infrastructure fast and safe. I also describe how modern software design can create resilient, well-maintained infrastructure. These practices and design approaches reinforce each other. Well-designed infrastructure is easier to test and deliver. Automated testing and delivery drive simpler and cleaner design.

TIP

Infrastructure as Code applies software engineering practices to the management of infrastructure for better reliability, security, and quality.

Benefits of Infrastructure as Code

To summarize, organizations adopting infrastructure as code hope to achieve benefits including:

- Using IT infrastructure as an enabler for rapid delivery of value,
- Reducing the effort and risk of making changes to infrastructure,
- Enabling users of infrastructure to get the resources they need, when they need it,

- Providing common tooling across development, operations, and other stakeholders,
- Creating systems that are reliable, secure, and cost-effective,
- Make governance, security, and compliance controls visible,
- Improving the speed to troubleshoot and resolve failures.

Use Infrastructure as Code to optimize for change

Given that:

- Changes are the biggest risk to a production system⁴,
- Continuous change is inevitable, and
- Making changes is the only way to improve a system,

Then it makes sense to optimize your capability to make changes both rapidly *and* reliably. Research from the *Accelerate State of the DevOps Report* backs this up. Making changes frequently and reliably is correlated to organizational success⁵.

There are several objections I hear when I recommend a team implement automation to optimize for change. I believe these come from misunderstandings of how you can use automation.

Objection: “We don’t make changes often enough to justify automating them”

We want to think that we build a system, and then it’s “done.” In this view, we don’t make many changes, so automating changes is a waste of time.

In reality, very few systems stop changing, at least before they are retired. Some people assume that their current level of change is temporary. Others create heavyweight change request processes to discourage people from asking for changes. These people are in denial. Most teams who are supporting actively used systems handle a continuous stream of changes.

Consider these common examples of infrastructure changes:

- An essential new application feature requires adding a new database,
- A new application feature requires an upgrade to the application server,
- Usage levels grow faster than expected. You need more servers, new clusters, and expanded network and storage capacity,
- Performance profiling shows that the current application deployment architecture is limiting performance. You need to redeploy the applications across different application servers. Doing this requires changes to the clustering and network architecture,
- There is a newly announced security vulnerability in system packages for your OS. You need to patch dozens of production servers,
- You need to update servers running a deprecated version of the OS and critical packages,
- Your web servers experience intermittent failures. You need to make a series of configuration changes to diagnose the problem. Then you need to update a module to resolve the issue,
- You find a configuration change that improves the performance of your database.

A fundamental truth of the Cloud Age is: *Stability comes from making changes.*

Unpatched systems are not stable; they are vulnerable. If you can't fix issues as soon as you discover them, your system is not stable. If you can't recover from failure quickly, your system is not stable. If the changes you do make involve considerable downtime, your system is not stable. If changes frequently fail, your system is not stable.

Objection: “We should build first and automate later”

Getting started with infrastructure as code is a steep curve. Setting up the tools, services, and working practices to automate infrastructure delivery is loads of work, especially if you're also adopting a new infrastructure platform. The value of this work is hard to demonstrate before you start building and deploying services with it. Even then, the value may not be apparent to people who don't work directly with the infrastructure.

So stakeholders often pressure infrastructure teams to build new cloud-hosted systems quickly, by hand, and worry about automating it later.

There are two reasons why automating afterward is a bad idea:

- Automation should enable faster delivery, even for new things. Implementing automation after most of the work has been done sacrifices much of the benefits.
- Automation makes it easier to write automated tests for what you build. And it makes it easier to quickly fix and

rebuild when you find problems. Doing this as a part of the build process helps you to build better infrastructure.

- Automating an existing system is very hard. Automation is a part of a system's design and implementation. To add automation to a system built without it, you need to change the design and implementation of that system significantly. This is also true for automated testing and deployment.

Cloud infrastructure built without automation becomes a write-off sooner than you expect. The cost of manually maintaining and fixing the system can escalate quickly. If the service it runs is successful, stakeholders will pressure you to expand and add features rather than stopping to rebuild.

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

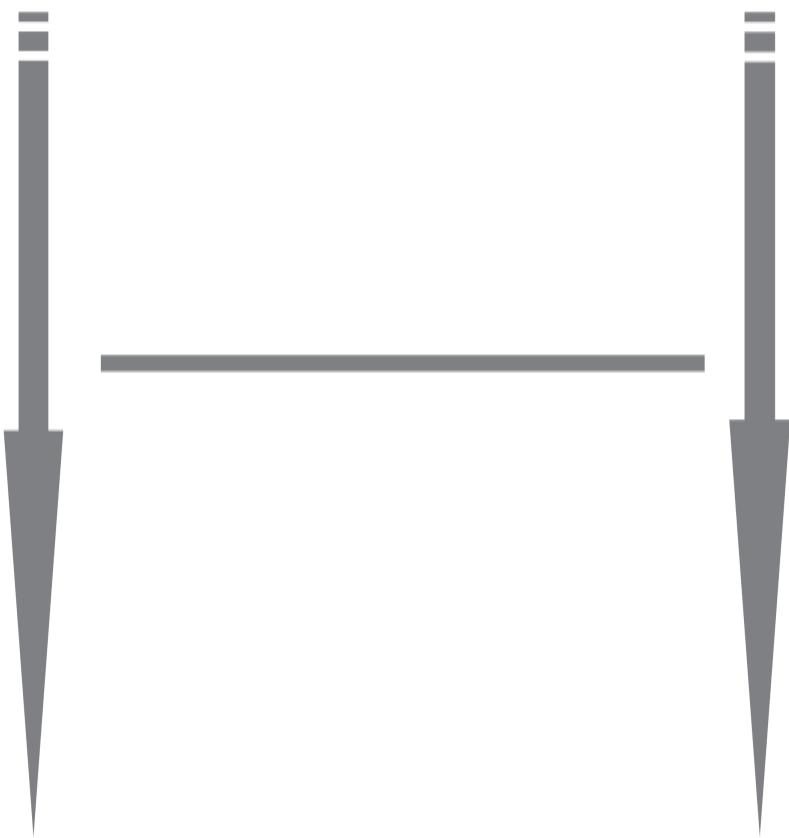
The solution is to build your system incrementally, automating as you go. Ensure you deliver a steady stream of value, while also building the capability to do so continuously.

Objection: “We must choose between speed and quality”

It's natural to think that you can only move fast by skimping on quality; and that you can only get quality by moving slowly. You might see this as a continuum, as shown in Figure 1-1.

Careful

Fast



Slow

Careless

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

However, the *Accelerate* research I mentioned earlier (“Use Infrastructure as Code to optimize for change”) shows otherwise.

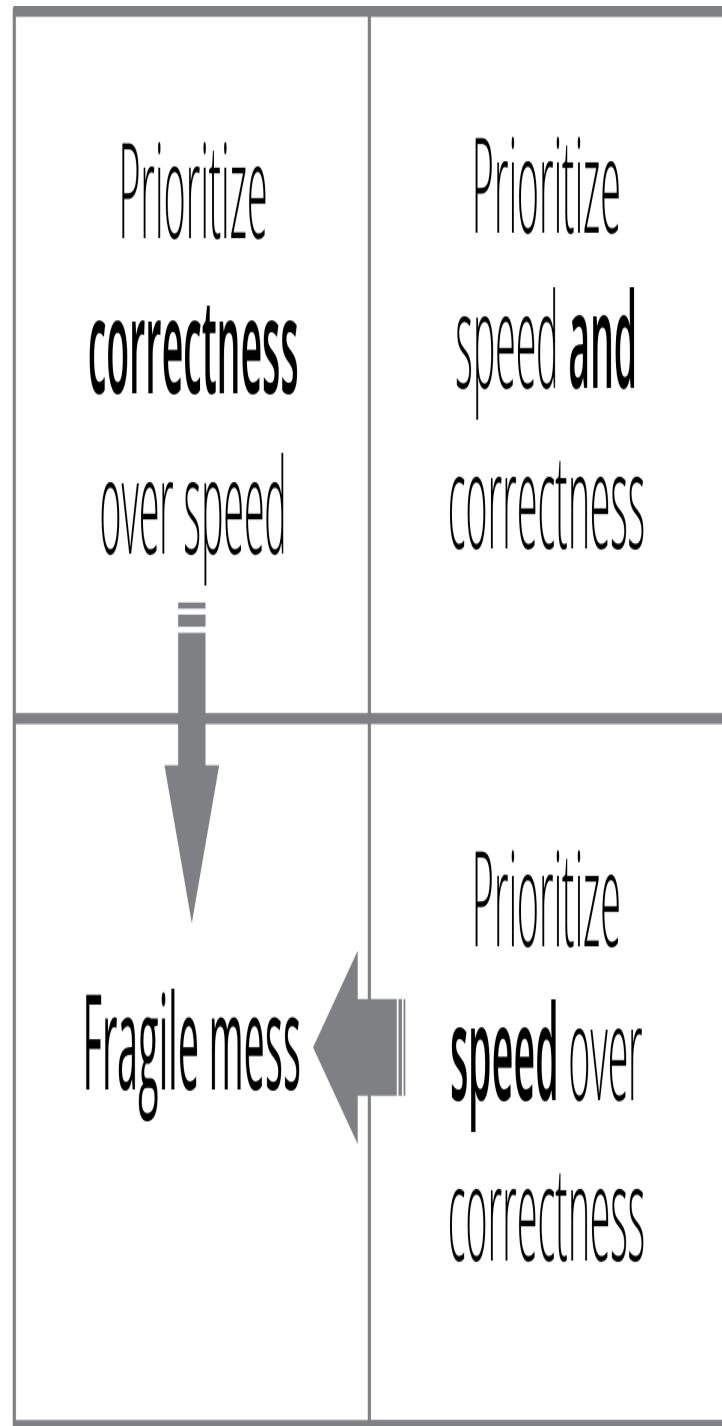
These results demonstrate that there is no tradeoff between improving performance and achieving higher levels of stability and quality. Rather, high performers do better at all of these measures. This is precisely what the Agile and Lean movements predict, but much dogma in our industry still rests on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

—Forsgren PhD, Nicole. 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.

I prefer to see quality and speed as a quadrant⁶ rather than a continuum, as shown in Figure 1-2.

Careful
Careless



Slow Fast

Figure 1-2. Change and stability is a quadrant

This quadrant model shows why trying to choose between speed and quality leads to being mediocre at both.

Lower-right quadrant

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. Many startups who have been working this way for a while complain about losing their “mojo.” Simple changes that they would have whipped out quickly in the old days now take days or weeks because the system is a tangled mess.

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 make the system hard to fix and improve. 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* it’s too hard to improve them. They add more processes in response to failures. These processes make it even harder to make improvements and increases fragility and risk. This leads to more failures and more process. Many people working in organizations that work this way assume this is normal⁷, especially those who work in risk-sensitive industries⁸.

The upper-right quadrant is the goal of modern approaches like lean, agile, and DevOps. Being able to move quickly and maintain 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 you find “high performers.”

WHY YOU SHOULD AUTOMATE FROM THE START

- You make changes more often than you may think,
- It's far easier to automate as you build something than to add it afterward,
- Automating the provisioning and configuration of environments ensures consistency from the start,
- Automation helps you make changes rapidly and reliably. Speed enables quality, and quality enables speed (Figure 1-3).

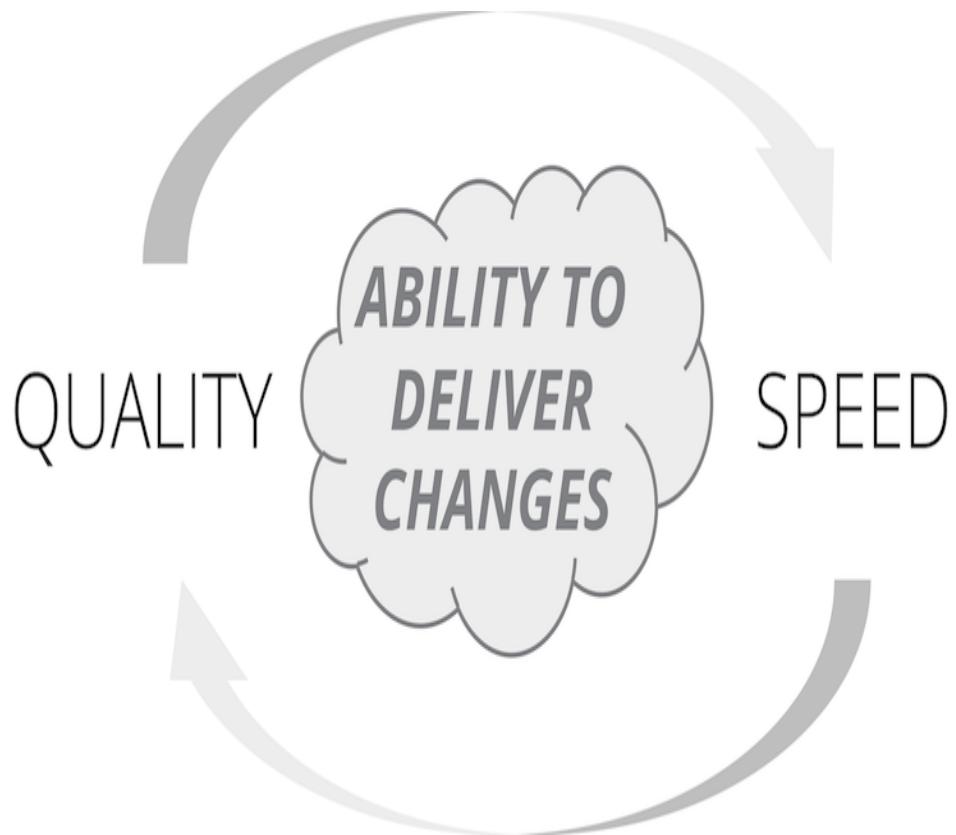


Figure 1-3. Speed enables quality, and quality enables speed

Three core practices for Infrastructure as Code

The Cloud Age concept exploits the dynamic nature of modern infrastructure and application platforms to make changes

frequently and reliably. Infrastructure as Code is an approach to building infrastructure that embraces continuous change for high reliability and quality. So how can your team do this?

There are three core practices for implementing Infrastructure as Code:

- Define everything as code
- Continuously validate all work in progress
- Build small, simple pieces that you can change independently

I'll summarize each of these now, to set the context for further discussion. Later, I'll devote a chapter to the principles for implementing each of these practices.

Core practice: Define everything as code

Defining all your stuff “as code” is a core practice for making changes rapidly and reliably. There are a few reasons why this helps:

Reusability

If you define a thing as code, you can create many instances of it. You can repair and rebuild your things quickly. 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. This makes testing more reliable. This enables continuous validation mentioned in “Core practice: Continuously validate all your work in progress”.

Transparency

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

I'll expand on concepts and implementation principles for defining things as code in [Chapter 4](#).

Core practice: Continuously validate all your work in progress

Effective infrastructure teams are rigorous about testing. They use automation to deploy and test each component of their system. They 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 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 only integrate when they finish. According to the *Accelerate* research, however, teams get better results when everyone integrates their work at least daily. Continuous Integration (CI) involves merging and testing everyone's code throughout development. Continuous Delivery (CD) takes this further, keeping the merged code always production-ready.

I'll go into more detail on how to continuously validate infrastructure code in [Chapter 9](#).

Core practice: Build small, simple pieces that you 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.

When you look at the codebase of a high performing team, you see the difference. Their system is composed of small, simple pieces. Each piece is easy to understand and has clearly defined interfaces. They can easily change each component on its own. And, they can deploy and test each component in isolation.

I dig more deeply into implementation principles for this core practice in [Link to Come].

RECAP: THE CORE PRACTICES OF INFRASTRUCTURE AS CODE

- Define everything as code
- Continuously validate all work in progress
- Build small, simple pieces that can be changed independently

The parts of an infrastructure system

There are many different parts, and types of parts, in a modern cloud infrastructure. I find it helpful to group these parts into platform layers. I use these as rough architectural boundaries for systems and dedicate a chapter of this book to two of these layers.

Infrastructure Platform

The Infrastructure Platform is the set of infrastructure resources and the tools and services that manage them. Cloud and virtualization platforms provide infrastructure resources, including compute, storage, and networking primitives. People also call this Infrastructure as a Service (IaaS). I'll elaborate on these in [Chapter 3](#). This layer also involves tools to define and manage infrastructure resources. An *infrastructure stack* is a collection of infrastructure resources managed as a unit. I'll talk about infrastructure stacks and tools that manage them, such as Terraform and CloudFormation, in [chapter Chapter 5](#).

Application Runtime Platform

Application Runtime Platforms build on infrastructure platforms to provide environments for running applications. Examples of services and constructs in an application runtime platform include container clusters, serverless environments, application servers, OS processes, and databases. People often refer to this as Platform as a Service (PaaS). The idea of “cloud native” systems has become popular in recent years, and fits in with this layer. I'll talk about these more in [\[Link to Come\]](#).

Applications

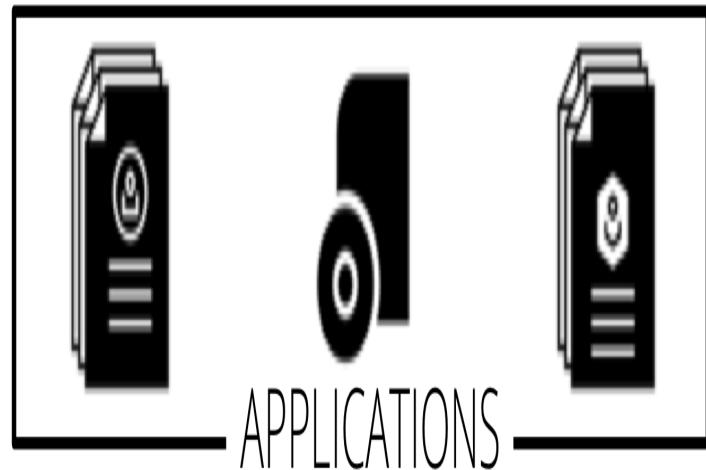
Applications and services run on your infrastructure, providing services to your organization and its users.

[Figure 1-4](#) shows these layers:

Application Packages

Container Instances

Serverless Code



Servers

Container Clusters

Database Clusters



Compute Resources

Network Structures

Storage

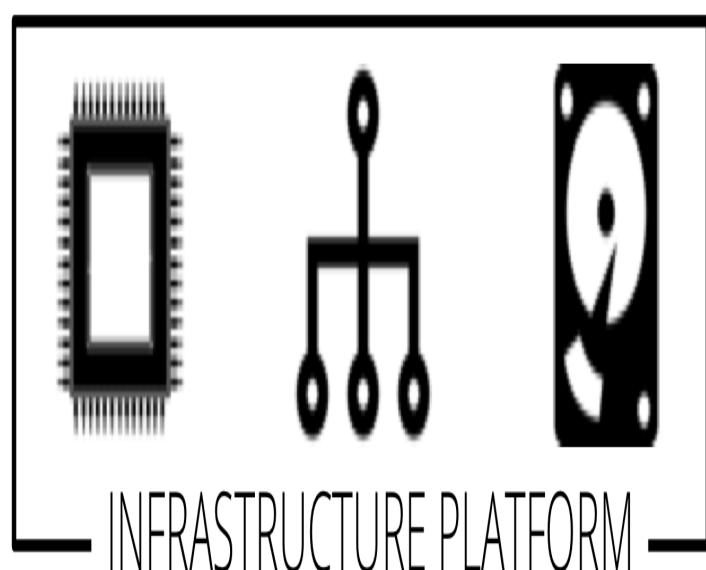


Figure 1-4. Layers of system elements

Conclusion

To get the value of cloud and infrastructure automation, you need a Cloud Age mindset. This means exploiting speed to improve quality, and building quality in to gain speed. Automating your infrastructure takes work, especially when you’re learning how to do it. But doing it helps you to make changes, including building the system in the first place.

I’ve described the parts of a typical infrastructure system, as these provide the foundations for chapters explaining how to implement Infrastructure as Code.

Finally, I defined three core practices for Infrastructure as Code:

- Build small, simple pieces that you can change independently
- Define everything as code
- Continuously validate all work in progress

In the next chapter, I’ll describe the principles that drive Cloud Age approaches for managing infrastructure.

-
- 1 This is as opposed to what they usually said a few years ago, which is that software is “not part of our core business.” Since then, those people realized their organizations were being overtaken by ones run by people who see better software as a way to compete, rather than as a cost to reduce.
 - 2 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.”

- 3 By “cowboy IT,” I mean people building IT systems without any particular method or consideration for future consequences. Often, people who have never supported production systems take the quickest path to get things working without considering security, maintenance, performance, and other operability concerns.
- 4 According to the [Visible Ops Handbook](#), changes cause 80% of unplanned outages.
- 5 Reports from the *Accelerate* research are available in the annual [State of the DevOps Report](#), and in the book *Accelerate* by Dr. Nicole Forsgren, Jez Humble, Gene Kim.
- 6 Yes, I do work at a consultancy, why do you ask?
- 7 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](#)
- 8 It’s ironic (and scary) that so many people in industries like finance, government, and healthcare consider fragile IT systems, and processes that obstruct improving them, to be normal, and even desirable.

Chapter 2. Principles of Cloud Age Infrastructure

Computing resources in the Iron Age of IT were tightly coupled to physical hardware. We assembled CPUs, memory, and hard drives in a case, mounted it into a rack, and cabled it to switches and routers. We installed and configured an operating system and application software. We could describe where an application server was in the data center: which floor, which row, which rack, which slot.

The Cloud Age decouples the computing resources from the physical hardware they run on. The hardware still exists, of course, but servers, hard drives, and routers float across it. These are no longer physical things, having transformed into virtual constructs that we create, duplicate, change, and destroy at will.

This transformation has forced us to change how we think about, design, and use computing resources. We can't rely on the physical attributes of our application server to be constant. We must be able to add and remove instances of our systems without ceremony. And we need to be able to easily maintain the consistency and quality of our systems even as we rapidly expand their scale.

CLOUD NATIVE APPLICATION ARCHITECTURES

This book focuses on how to design and build infrastructure in the context of Cloud Age platforms. The dynamic nature of these platforms also affects the design of application software. Cloud native applications are decoupled from specific infrastructure resources like servers, hard-drives, and network locations. Developers assume their applications will be automatically scaled up and down, and that workloads will be shifted between instances. [Link to Come] and the rest of [Link to Come] describe how to build infrastructure that supports cloud native software.

There are several principles for designing and implementing infrastructure on cloud platforms. These principles articulate the reasoning for using the three core practices (define everything as code, continuously validate, build small pieces). I also list several common pitfalls teams make with dynamic infrastructure.

Together, these principles and pitfalls drive more specific implementation principles for the core infrastructure as code practices and help choose between different implementation patterns. They set the context for the advice throughout this book.

Principle: Assume systems are unreliable

In the Iron Age, we assumed our systems were running on reliable hardware. In the Cloud Age, you need to assume your system runs on unreliable hardware¹.

Cloud scale infrastructure involves hundreds of thousands of devices, if not more. At this scale, failures happen even when

using reliable hardware. And most cloud vendors use cheap, less reliable hardware, detecting and replacing it when it breaks.

You'll need to take parts of your system offline for reasons other than unplanned failures. You'll need to patch and upgrade systems. You'll resize. Redistribute load. Troubleshoot problems.

With static infrastructure, doing these things means taking systems offline. But in many modern organizations, taking systems offline means taking the business offline.

So you can't treat the infrastructure your system runs on as a stable foundation. Instead, you must design for uninterrupted service when underlying resources change².

² See the section on "Design for Change" in the book "The Phoenix Project" by Gene Kim, Kevin Behr, and George Spafford.

THE TRADEOFFS OF “LIFT AND SHIFT” MIGRATION TO THE CLOUD

Many organizations are migrating their systems from Iron Age infrastructure to cloud platforms. A common approach is “lift and shift”, which aims to emulate existing on-premise infrastructure, so that applications can be moved with few or no changes to code and configuration. Teams replicate physical network structures and static addresses in the cloud, and even use virtual appliance versions of network devices such as firewalls and routers.

The reasons people use lift and shift include:

- To use familiar, trusted technology, vendors, and topologies, rather than having to retrain,
- To avoid the cost of modifying or reconfiguring applications,
- To simplify the migration process, avoiding big hairy rewrites.

The problems with lift and shift include:

- Replicating physical infrastructure configuration in the cloud adds complexity and cost, not only for the migration process but also for ongoing maintenance.
- Many products, topologies, and patterns that are appropriate for data centers are not appropriate for the cloud. Transplanting them adds cost, and may not add any benefit. They can even make performance, security, and maintainability worse.
- Resources provided by the cloud platform provide benefits for reliability, security, and scalability. Reproducing data center configurations loses those benefits.

Lift and shift aims to reduce the work of moving an application to the cloud by adding implementation work to its infrastructure. This may succeed if the infrastructure work can realistically be done more easily than changing the application. More often, applications can be tweaked to make them cope easier in their new environment. In any case, you should have a concrete strategy for adapting, rewriting, or managing applications after they have been shifted to the cloud.

Principle: Make everything reproducible

One way to make a system recoverable is to make sure you can rebuild its parts effortlessly and reliably.

Effortlessly means that there is no need to make any decisions about how to build things. You should define things such as

configuration settings, software versions, and dependencies as code. Rebuilding is then a simple “yes/no” decision.

MEASURING REPRODUCIBILITY

You and your team should track the time it takes you to build a new instance of an existing system. This best way to know this, with confidence, is to rebuild frequently, as a part of routine operations.

Some teams build fresh infrastructure every time they deploy a new software release. This process is usually part of a zero-downtime release process (see [Link to Come]). Other teams rebuild infrastructure in test environments as part of a pipeline ([Link to Come]).

Either approach regularly proves the speed that you can rebuild. This in turn means you can be confident you can do it in an emergency situation.

Not only does reproducibility make it easy to recover a failed system, but it also helps you to:

- Make testing environments consistent with production,
- Replicate systems across regions for availability,
- Add instances on demand to cope with high load,
- Replicate systems to give each customer a dedicated instance.

Of course, the system generates data, content, and logs, which you can’t define ahead of time. You need to identify these and find ways to keep them as a part of your replication strategy. Doing this might be as simple as copying or streaming data to a backup and then restoring it when rebuilding. I’ll describe options for doing this in [Link to Come].

The ability to effortlessly build and rebuild any part of the infrastructure is powerful. It removes risk and fear of making changes. You can handle failures with confidence. You can rapidly provision new services and environments.

Pitfall: Snowflake systems

A snowflake is an instance of a system or part of a system that is difficult to rebuild. It may also be an environment that should be similar to other environments, such as a staging environment, but is different in ways that its team doesn't fully understand.

People don't set out to build snowflake systems. They are a natural occurrence. The first time you build something with new tool, you learn lessons along the way, which involves making mistakes. But if people are relying on the thing you've built, you may not have time to go back and rebuild or improve it using what you learned. Improving what you've built is especially hard if you don't have the mechanisms and practices that make it easy to and safe to change.

Another cause of snowflakes is when people make changes to one instance of a system that they don't make to others. They may be under pressure to fix a problem that only appears in one system. Or they may start a major upgrade in a test environment, but run out of time to roll it out to others.

You know a system is a snowflake when you're not confident you can safely change or upgrade it. Worse, if the system does break, it's hard to fix it. So people avoid making changes to the system, leaving it out of date, unpatched, and maybe even partly broken.

Snowflake systems create risk and waste the time of the teams that manage them. It is almost always worth the effort to replace them with reproducible systems. If a snowflake system isn't worth improving, then it may not be worth keeping at all.

The best way to replace a snowflake system is to write code that can replicate the system, running the new system in parallel until it's ready. Use automated tests and pipeline to prove that it is correct, reproducible, and that you can change it easily.

Principle: Create disposable things

Building a system that can cope with dynamic infrastructure is one level. The next level is building a system that is itself dynamic. You should be able to gracefully add, remove, start, stop, change, and move the parts of your system. Doing this creates operational flexibility, availability, and scalability. It also simplifies and de-risks changes.

Making the pieces of your system malleable is the main idea of **cloud-native software**. Cloud abstracts infrastructure resources (compute, networking, and storage) from physical hardware. Cloud-native software completely decouples application functionality from the infrastructure it runs on. See [Link to Come] for more on this.

CATTLE, NOT PETS

“Treat your servers like cattle, not pets,” is a popular expression about disposability³. I miss giving fun names to each new server I create. But I don’t miss having to tweak and coddle every server in our estate by hand.

If your systems are dynamic, then the tools you use to manage them need to cope with this. For example, your monitoring should not raise an alert every time you rebuild part of your system. However, it should raise a warning if something gets into a loop rebuilding itself.

THE CASE OF THE DISAPPEARING FILE SERVER

People can take a while to get used to ephemeral infrastructure. One team I worked with set up automated infrastructure with VMWare and Chef. They deleted and replaced virtual machines as needed.

A new developer on the team needed a web server to host files to share with teammates. So he manually installed an HTTP server on a development server and put the files there. A few days later, I rebuilt the VM, and his web server disappeared.

After some confusion, the developer understood why this had happened. He added his web server to the Chef code, and persisted his files to the SAN. The team now had a reliable file sharing service.

Principle: Minimize variation

As a system grows, it becomes harder to understand, harder to change, and harder to fix. The work involved grows with the number of pieces, and also with the number of different *types* of pieces. So a useful way to keep a system manageable is to have fewer types of pieces—to keep variation low. It's easier to manage 100 identical servers than 5 completely different servers.

The reproducibility principle (“[Principle: Make everything reproducible](#)”) complements this idea. If you define a simple component and create many identical instances of it, then you can easily understand, change, and fix it.

To make this work, you must apply any change you make to all instances of the component. Otherwise, you create configuration-drift.

Here are some types of variation you may have in your system:

- Multiple operating systems, application runtimes, databases, and other technologies. Each one of these needs people in your team to keep up skills and knowledge.
- Multiple *versions* of software such as an operating system or database. Even if you only use one operating system, each version may need different configurations and tooling.
- Different versions of a package. When some servers have a newer version of a package, utility, or library than others, you have risk. Commands may not run consistently across them. Older versions may have vulnerabilities or bugs.

Organizations have tension between allowing each team to choose technologies and solutions that are appropriate to their needs, versus keeping the amount of variation in the organization to a manageable level.

LIGHTWEIGHT GOVERNANCE

Modern, digital organizations are learning the value of *Lightweight Governance* in IT to balance autonomy and centralized control. This is a key element of the EDGE model for agile organizations. For more on this, see the book, [EDGE: Value-Driven Digital Transformation](#), or Jonny LeRoy's talk, [The Goldilocks zone of lightweight architectural governance](#).

Configuration Drift

Configuration drift is variation that happens over time across systems that were once identical. [Figure 2-1](#) shows this. Manually making changes can cause configuration drift. It can also happen if you use automation tools to make ad-hoc changes to only some of the instances.

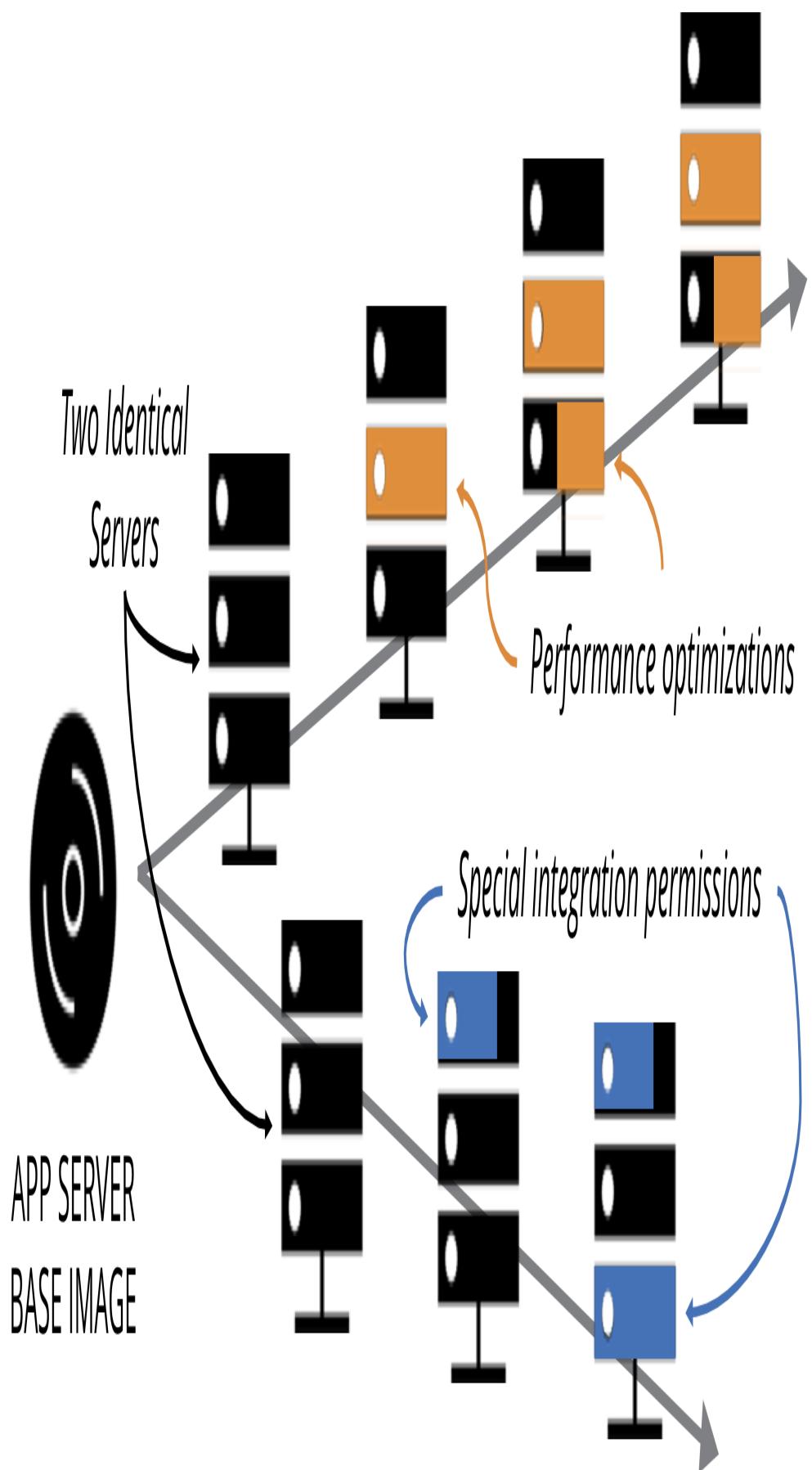


Figure 2-1. Configuration drift is when instances of the same thing become different over time.

Configuration drift makes it harder to maintain consistent automation.

FOODSPIN EXAMPLE: DIVERGING INFRASTRUCTURE

Foodspin is a fictional company that provides an online menu service for fast food restaurants. I'll be using it throughout the book to demonstrate how concepts may play out in practice. First, let's look at Foodspin as an example of configuration drift.

Foodspin runs a separate instance of their application for each restaurant, each instance configured to use custom branding and menu content. In the early days, the Foodspin team ran scripts to create a new application server for each new restaurant. They managed the infrastructure manually or by writing scripts and tweaking them each time they needed to make a change.

One of the restaurants, Curry Hut, has far more traffic to its menu application than the others, so the team tweaked the configuration for the Curry Hut server. They didn't make the changes to the other customers, because the team was busy and it didn't seem necessary.

Later, the Foodspin team adopted Ansible to automate their application server configuration. They first tested it with the server for The Fish King, a lower-volume customer, and then rolled it out to their other customers. Unfortunately, their code didn't include the performance optimizations for Curry Hut, so it stripped those improvements. The Curry Hut server slowed to a crawl until the team caught and fixed their mistake.

The Foodspin team overcame this by parameterizing their Ansible code. It can now set resource levels different for each customer. This way, they still apply the same code across every customer, while still optimizing it for each.

We frequently need to support specific variations between instances of otherwise identical infrastructure. I'll describe some patterns and antipatterns for dealing with this in Chapter 8.

THE AUTOMATION FEAR SPIRAL

The automation fear spiral describes how many teams fall into configuration drift and technical debt.

At an [Open Space](#) session on configuration automation at a [DevOpsDays](#) conference, I asked the group how many of them were using automation tools like Ansible, Chef, or Puppet. The majority of hands went up. I asked how many were running these tools unattended, on an automatic schedule. Most of the hands went down.

Many people have the same problem I had in my early days of using automation tools. I used automation selectively—for example, to help build new servers, or to make a specific configuration change. I tweaked the configuration each time I ran it, to suit the particular task I was doing.

I was afraid to turn my back on my automation tools because I lacked confidence in what they would do.

I lacked confidence in my automation because my servers were not consistent.

My servers were not consistent because I wasn't running automation frequently and consistently.

This is the automation fear spiral, as shown in [Figure 2-2](#). Infrastructure teams must break this spiral to use automation successfully. The most effective way to break the spiral is to face your fears. Start with one set of servers. Make sure you can apply, and then re-apply your infrastructure code to these servers. Then schedule an hourly process that continuously applies the code to those servers. Then pick another set of servers and repeat the process. Do this until every server is continuously updated.



Figure 2-2. The automation fear spiral

Good monitoring and automated testing builds the confidence to continuously synchronize your code. This exposes configuration drift as it happens, so you can fix it immediately.

Principle: Ensure that you can repeat any process

Building on the reproducibility principle, you should be able to repeat anything you do to your infrastructure. It's easier to repeat actions using scripts and configuration management tools than to

do it by hand. But automation can be a lot of work, especially if you're not used to it.

For example, let's say I have to partition a hard drive as a one-off task. Writing and testing a script is much more work than just logging in and running the `fdisk` command. So I do it by hand.

The problem comes later on, when someone else on my team, Priya, needs to partition another disk. She comes to the same conclusion I did, and does the work by hand rather than writing a script. However, she makes slightly different decisions about how to partition the disk. I made an 80 GB `/var` ext3 partition on my server, but Priya made `/var` a 100 GB xfs partition on hers. We're creating configuration drift, which will erode our ability to automate with confidence.

Effective infrastructure teams have a strong scripting culture. If you can script a task, script it⁴. If it's hard to script it, dig deeper. Maybe there's a technique or tool that can help, or maybe you can simplify the task or handle it differently. Breaking work down into scriptable tasks usually makes it simpler, cleaner, and more reliable.

PHOENIX SERVERS

A Phoenix Server⁵ is frequently rebuilt, in order to ensure that the provisioning process is repeatable. This can be done with other infrastructure constructs, including infrastructure stacks.

Conclusion

The Principles of Cloud Age Infrastructure embody the differences between traditional, static infrastructure, and modern, dynamic

infrastructure:

- Assume systems are unreliable,
- Make everything reproducible,
- Create disposable things,
- Minimize variation,
- Ensure that you can repeat any action.

These principles are the key to exploiting the nature of cloud platforms. Rather than resisting the ability to make changes with minimal effort, exploit that ability to gain quality and reliability.

The next chapter focuses on the types of infrastructure resources provided by cloud platforms. It sets the context for what we define as code, validate continuously, in small pieces, which are the three core practices for implementing infrastructure as code.

¹ I learned this idea from Sam Johnson’s article, “[Simplifying Cloud: Reliability](#)”.

² The principle of assuming systems are unreliable drives [Chaos Engineering](#), which injects failures in controlled circumstances to test and improve the reliability of your services. I talk about this more in [Link to Come]

³ I first heard this expression in Gavin McCance’s presentation “[CERN Data Centre Evolution](#)”. Randy Bias credits Bill Baker’s presentation “[Architectures for Open and Scalable Clouds](#)”. Both of these presentations are an excellent introduction to these principles.

⁴ My colleague Florian Sellmayr says, “If it’s worth documenting, it’s worth automating.”

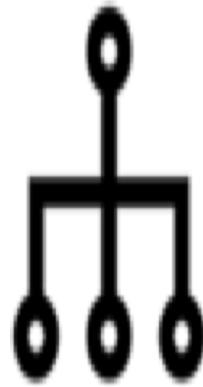
⁵ My colleague [Kornelis Sietsma](#) coined the term Phoenix Server, which Martin Fowler later documented on [his site](#).

Chapter 3. Infrastructure Platforms

The first thing you need for infrastructure as code is infrastructure. In Chapter 1 I shared a layered platform model (“The parts of an infrastructure system”). The infrastructure is the foundational layer of this model.

APPLICATIONS

APPLICATION RUNTIME PLATFORM



INFRASTRUCTURE PLATFORM

Figure 3-1. The infrastructure platform is the foundation layer of the platform model from chapter 1

Infrastructure as code requires a *dynamic* infrastructure platform, something that you can use to provision and change resources on demand with an API. This is the essential definition of a cloud¹. When I talk about an “infrastructure platform” in this book, you can assume I mean a dynamic, Infrastructure as a Service (IaaS)² type of platform.

In the old days-the Iron Age of computing-infrastructure was hardware. Virtualization³ decoupled systems from the hardware they ran on. Cloud added APIs to manage those virtualized resources. Thus began the Cloud Age.

There are different types of infrastructure platforms, from full-blown public clouds to private clouds; from commercial vendors to open source platforms. In this chapter, I outline these variations and then describe the different types of infrastructure resources they provide.

At the basic level, infrastructure platforms provide compute, storage, and networking resources. The platforms provide these resources in various formats. For instance, you may run compute as virtual servers, container runtimes, and serverless code execution.

Different vendors may package and offer the same resources in different ways, or at least with different names. For example, AWS object storage, Azure blob storage, and GCP cloud storage are all pretty much the same thing.

I describe the common types of resources that most cloud platforms provide in this chapter. I want the implementation patterns and recommendations throughout this book to be useful no matter which cloud you use. So I try to use names that are relevant across clouds. Rather than “VPC” and “Subnet,” I talk about “Network address blocks.”

This chapter should be a useful reference if you are wondering how to apply infrastructure as code practices and patterns to a particular aspect of your system, for example, container clusters.

Not everyone draws the line between infrastructure resources and application runtime services like container clusters and databases in the same place. I don’t think it particular matters where you draw the boundaries in your system. I have included some things in this chapter and others in [Link to Come], based on my personal preferences.

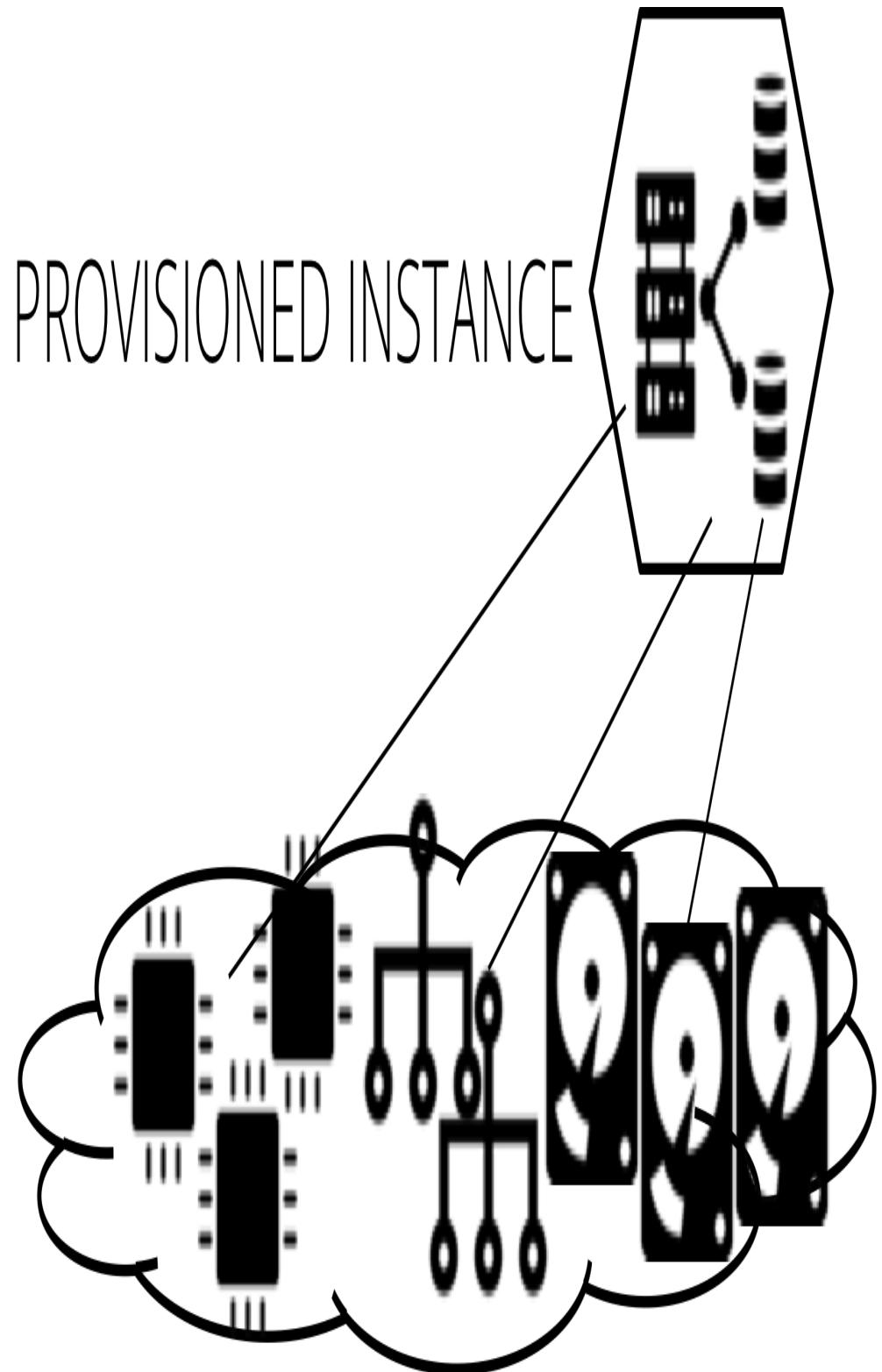
What is a dynamic infrastructure platform?

Public clouds are the first thing most of us think of as platforms for infrastructure as code. But there are other options for cloud infrastructure.

The key characteristics of a dynamic infrastructure platform are:

- It provides a pool of compute, networking, and storage resources,
- It allows you to provision and change these resources *on demand*,

- You can provision, configure, and change resources programmatically.



PROVISIONED INSTANCE

RAW INFRASTRUCTURE RESOURCES

Figure 3-2. An infrastructure platform provides a pool of resources that you can provision on demand.

A public IaaS cloud clearly has these characteristics. However, many private cloud platforms also qualify. A private cloud may run on a pool of hardware owned by the organization. However, its users use an API to provision infrastructure from that hardware pool, so it still counts as a cloud.

When performance or other requirements make it useful to run applications directly on server hardware, you can implement a bare-metal cloud. A bare-metal cloud provides an API to provision operating systems directly onto physical servers on demand.

Table 3-1 lists examples of vendors, products, and tools for each type of cloud infrastructure platform.

Table 3-1. Examples of dynamic infrastructure platforms

Type of Platform	Providers or Products
Public IaaS cloud services	AWS, Azure, Digital Ocean, GCE, Linode, and Oracle Cloud
Private IaaS cloud products	CloudStack, OpenStack, and VMware vCloud
Bare-metal cloud tools	Cobbler, FAI, and Foreman

Infrastructure Resources

There are three essential resources provided by an infrastructure platform: compute, storage, and networking. Different platforms combine and package these resources in different ways. For example, you may be able to provision virtual machines and

container instances on your platform. You may also be able to provision a database instance, which combines compute, storage, and networking.

I call the more elemental infrastructure resources primitives. Each of the compute, networking, and storage resources described in the sections later in this chapter is a primitive. Cloud platforms combine infrastructure primitives into composite resources, such as:

- Database as a Service (DBaaS)
- Load balancing
- DNS
- Identity management
- Secrets management

The line between a primitive and a composite resource is arbitrary, as is the line between a composite infrastructure resource and an application runtime service. Even a basic storage service like object storage (think AWS S3 buckets) involves compute and networking resources to read and write data. But it's a useful distinction, which I'll use now to list common forms of infrastructure primitives. These fall under three basic resource types: compute, storage, and networking.

Compute Resources

Compute resources execute code. At its most elemental, compute is execution time on a physical server CPU core. But platforms provide compute in more useful ways. The most common compute resource primitives are:

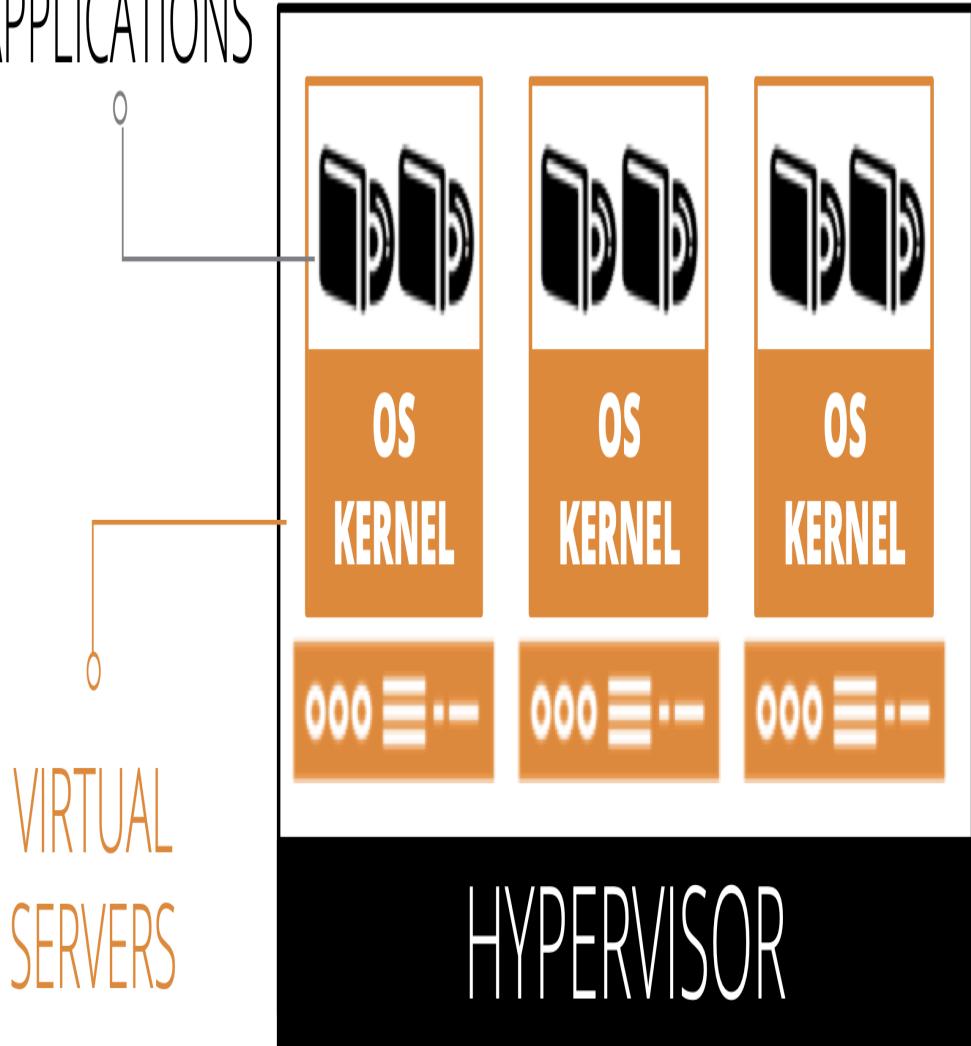
- Virtual machines
- Physical servers
- Containers
- Server clusters
- Serverless code execution (FaaS)

I'll describe each of these in more detail.

Virtual machines

Virtual Machines (VMs), sometimes called virtual servers, are the workhorse of cloud infrastructure. The infrastructure platform manages a pool of physical host servers, and runs virtual machine instances on hypervisors across these hosts:

APPLICATIONS



HYPERVERISOR

PHYSICAL
SERVER

Figure 3-3. Each virtual machine has its own operating system kernel

The difference between ordinary server virtualization and an IaaS cloud platform is provisioning. To provision a non-cloud VM, you specify which host server it should run on. When you provision a cloud VM, the cloud platform decides where to run it, and you shouldn't need to know or care where.

Some clouds automatically migrate live server instances between hosts, transparently to processes running on them. This ability allows platform operators to balance workloads more efficiently. They can also carry out maintenance such as patching without downtime. It's useful to understand how your cloud provider handles this and how it may impact you.

Some platform operators commit to keeping VMs running and available, restarting them if they fail, or moving them to a new host if their host fails. Others put the responsibility for this on the user. Check your vendor documentation to understand how it works so that you can be prepared. [Link to Come] discusses continuity of service.

[Link to Come] goes into much more detail about how servers are provisioned, configured, and managed.

Physical servers

As I mentioned earlier, although we tend to think of virtual servers or containers when we think of cloud, it's entirely possible to provision physical servers on demand. There are several reasons why you might need this:

- To avoid the performance overhead of virtualization. Applications such as real-time trading which demand low latency for processing or data access can benefit from this.
- To allow direct access to physical resources, such as special devices, that might not be accessible from within a virtual machine or container.
- To ensure applications are not impacted by other processes running on the same host as a virtual machine.
- To more strongly enforce segregation of processes and data for security, governance, or contractual requirements.
- If you are the one providing a virtualized or containerized platform to your users, you need to provision and manage the physical host servers your platform runs on.

IMPLEMENTING A BARE-METAL CLOUD

A bare-metal cloud platform lets you write code to install and reinstall OS images onto physical servers automatically. For example, a central process receives a request to provision a server. It:

1. Selects an unused physical server from its inventory,
2. Triggers the server to boot in a “network install” mode supported by the server firmware (e.g., PXE boot),
3. Provides an OS image that the server’s firmware downloads from the network and copies it to the primary hard drive,
4. Reboots the server so that the OS boots,
5. The OS image includes a server configuration agent that runs on startup, retrieves configuration from a server, and applies it.

Some tools that you can use for this process include [Cobbler](#), [FAI - Fully Automatic Installation](#), [Foreman](#), and [Crowbar](#). These tools can take advantage of the PXE specification [Preboot Execution Environment](#) to boot a basic OS image downloaded from the network, which then runs an installer to download and boot an OS installer image. The installer image runs a script, perhaps with something like [Kickstart](#), to configure the OS.

Often, triggering a server to use PXE to boot a network image requires pressing a function key while the server starts. Doing this can be tricky to do unattended. However, many hardware vendors have lights-out management (LOM) functionality that makes it possible to do this remotely.

Containers

Containers are a popular way to package and deploy applications onto cloud platforms. They simplify the packaging and deployment process by abstracting the details of the system that runs the application. They are also much smaller than a virtual machine, which makes them more efficient and faster to start.

Most cloud platforms offer Containers as a Service (CaaS) to deploy and run container instances. Often, you build a container image in a standard format (e.g., Docker), and the platform uses this to run instances.

WHAT IS THE DIFFERENCE BETWEEN A VIRTUAL MACHINE AND A CONTAINER?

A Virtual Machine (VM) contains a complete operating system, while a container shares the operating system kernel with its host system.

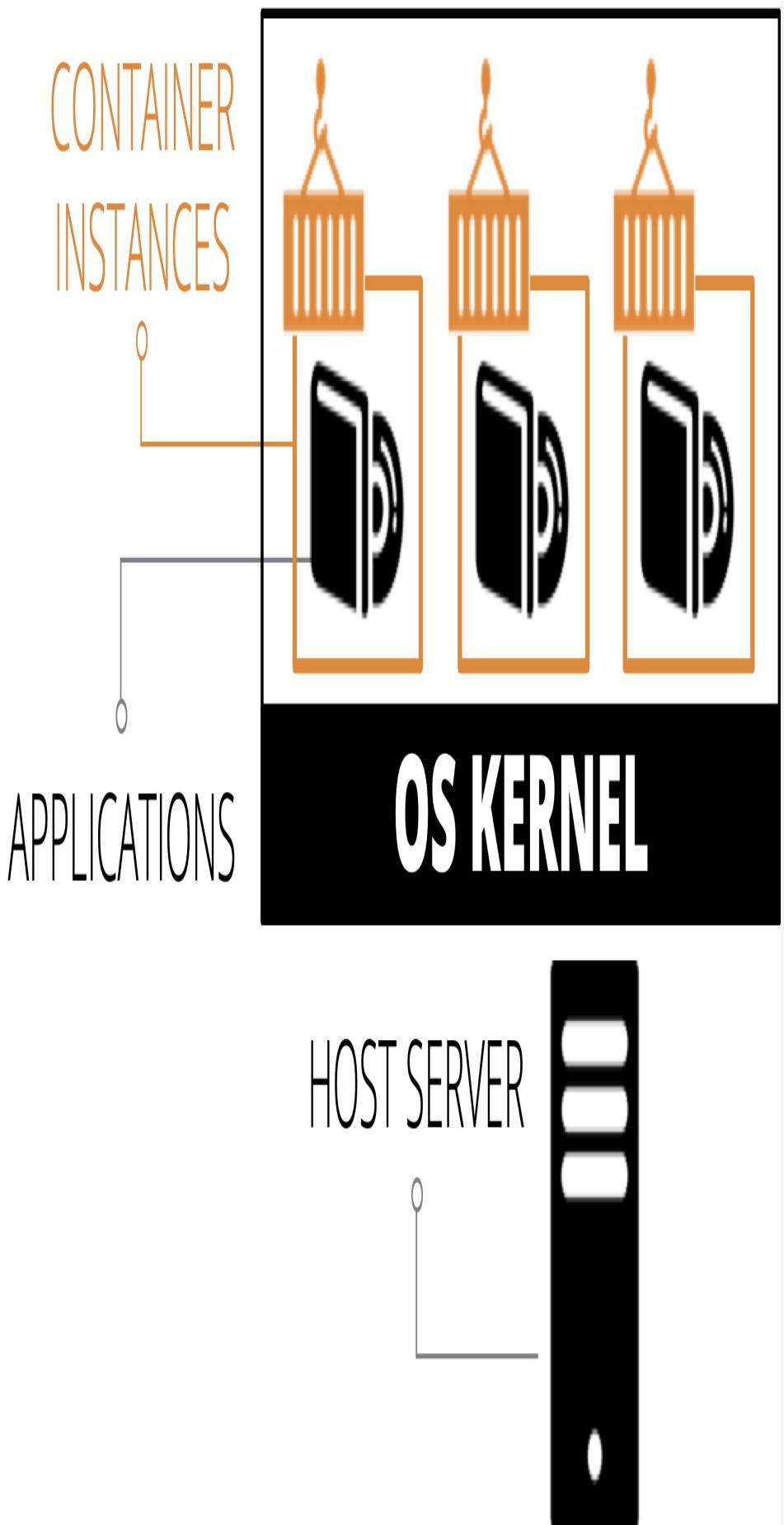


Figure 3-4. The container instances on a host share the same OS kernel

Including the entire OS makes a VM heavier than a container, but each VM running on a host can use a different OS than its host. A container is lighter than a VM because it doesn't need to package or instantiate an OS kernel. However, a container is coupled to the host OS kernel version.

People tend to use containers and VMs in different ways. A single VM often runs multiple applications, while a container instance typically runs only one application.

Conceptually, containers are an application packaging format. They are a "fat" package, meaning they include all of the runtime dependencies for the application.

Many application runtime platforms are built around containerization. [Link to Come] discusses this in more detail.

Containers require host servers to run on. Some platforms provide these hosts transparently, but many require you to define a cluster and its hosts yourself.

Server clusters

A server cluster is a pool of server instances-either virtual machines or physical servers-which the infrastructure platform provisions and manages as a group. You can configure how the platform should manage the cluster. Typical configuration properties include a minimum and a maximum number of servers to run at a time, algorithms and strategies for adding and removing servers, and specification of what type of servers to use and how to provision them.

There are two common types of server clusters, a *basic server cluster*, and an *application hosting cluster*.

BASIC SERVER CLUSTERS

A basic server cluster simply runs the appropriate number of servers. You need to implement the deployment of applications onto each node of the cluster. Typically, every server in a basic cluster runs an identical set of applications.

EXAMPLES OF BASIC SERVER CLUSTERS PROVIDED BY THE MAJOR CLOUD PLATFORMS

- AWS Auto Scaling Group (ASG)
- Azure virtual machine scale set
- Google Managed Instance Group (MIGs)

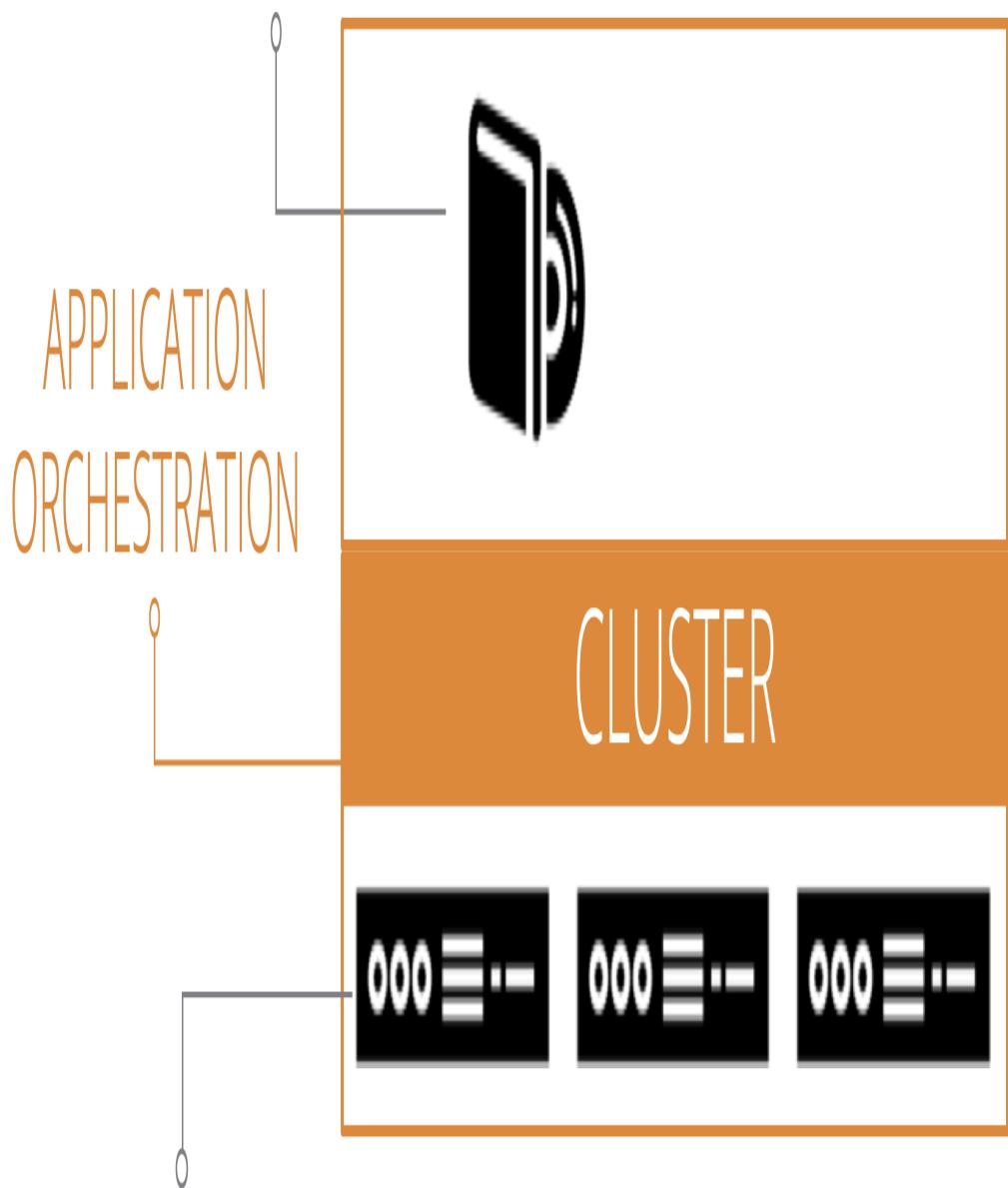
You use a basic server cluster to automatically add compute power when demand increases and to remove it when demand drops. You can also use clusters for availability because they can automatically rebuild failed instances.

APPLICATION HOSTING CLUSTER

Like a basic server cluster, an application hosting cluster is a pool of servers, either virtual machines or physical servers. However, the platform adds services that deploy and run applications across the servers in the cluster.

You deploy an application to the cluster, defining some properties for the deployment, and the platform works out where to run the application.

DEPLOYED APPLICATION



SERVERS

Figure 3-5. An application hosting cluster is a group of servers which acts as a single deployment target

Unlike a basic cluster, you typically deploy multiple applications to an application hosting cluster. The properties you specify for each application tell the platform how many instances of the application to deploy, with rules for adding and removing instances. I go into more detail on application hosting clusters in [Link to Come].

EXAMPLES OF APPLICATION HOSTING CLUSTERS ON PUBLIC CLOUD PLATFORMS

- Amazon Elastic Container Services (ECS)
- Amazon Elastic Container Service for Kubernetes (EKS)
- Azure Kubernetes Service (AKS)
- Google Kubernetes Engine (GKE)

Serverless code execution (FaaS)

An infrastructure platform executes serverless code on-demand, in response to an event or schedule, and then terminates it after it has completed its action. Unlike applications deployed on servers or in containers, serverless code does not run continuously, so the platform only allocates resources when they are used.

Serverless code is useful for well-defined, short-lived actions where the code starts quickly. Typical examples are handling HTTP requests or responding to error events in a message queue. The platform launches multiple instances of the code in parallel when needed, for example, to handle multiple events coming in simultaneously.

Serverless can be very efficient for workloads where the demand varies greatly, scaling up when there are peaks, and not running at all when not needed.

“Serverless” isn’t the most accurate term for this, because of course, the code does run on a server. It’s just that the server is effectively invisible to you as a developer. The same is true with containers, so what is distinctive about so-called serverless isn’t the level of abstraction from servers. The real distinction with serverless is that it is a short-lived process rather than a long-running process.

For this reason, many people prefer the term Function as a Service (FaaS) rather than serverless.⁴

SERVERLESS BEYOND CODE

The serverless concept goes beyond running code. AWS Aurora is essentially a serverless database: the servers involved in hosting the database are transparent to you, unlike more traditional DBaaS offerings like AWS RDS. For that matter, you could consider S3 buckets to be storage as a service.

Some cloud platforms offer specialized on-demand application execution environments. For example, you can use AWS SageMaker, Azure ML Services, or Google ML Engine to deploy and run machine learning models.

I describe approaches to packaging and deploying serverless code in [Link to Come].

FOODSPIN EXAMPLE: COMPUTE RESOURCES

Let's look at Foodspin's use of compute resources. Their tech stack includes a cluster of containerized Nginx web servers, which is shared across all of their customers. Each customer's ordering website runs as its own Java process deployed onto a Linux virtual machine. They also have several FaaS applications that carry out reporting activities.

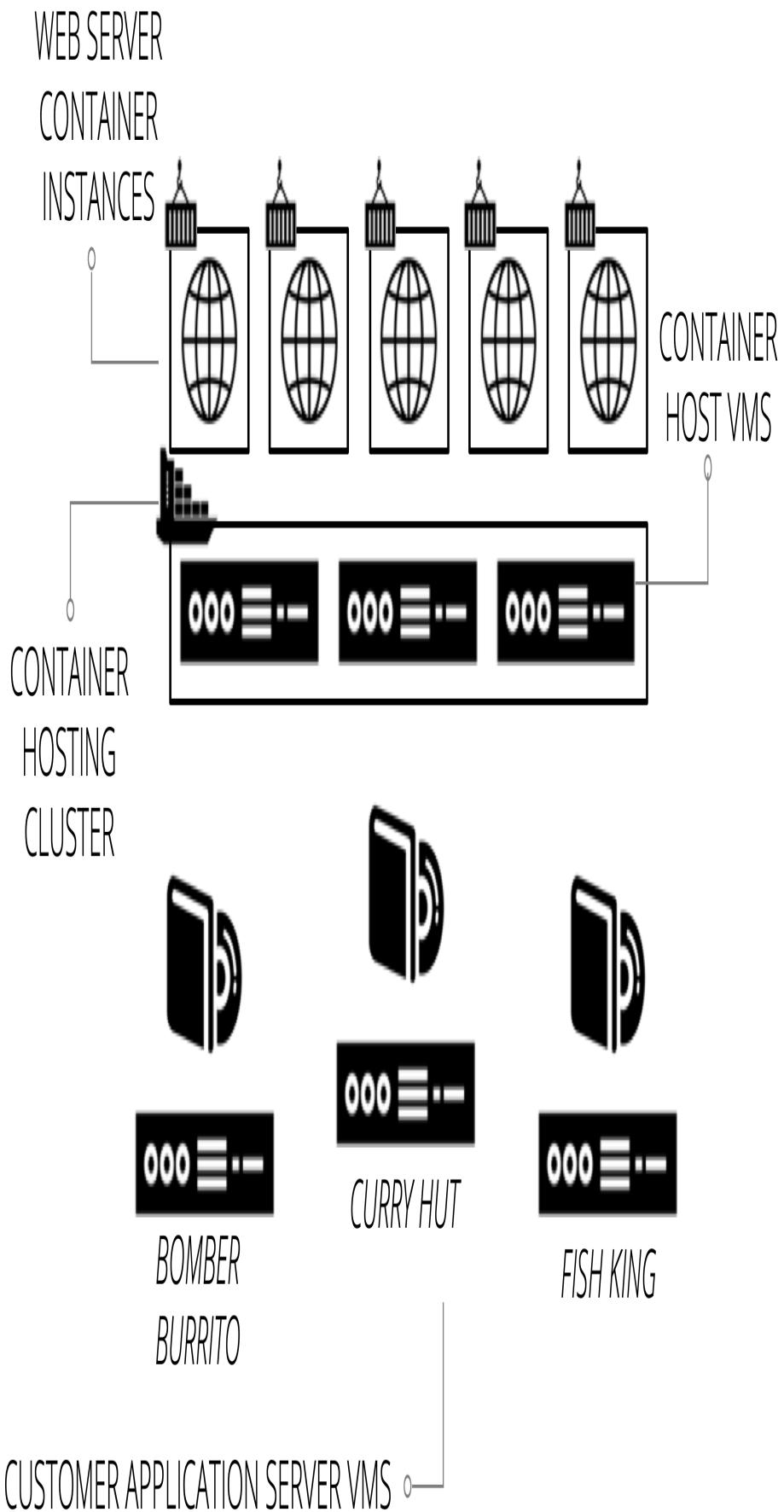


Figure 3-6. Foodspin compute resources

This example demonstrates how a single system can use multiple types of compute resources. Later examples in this chapter add storage and networking resources to Foodspin's infrastructure. In later chapters, I use these examples to illustrate how to code, test, and evolve a system.

Storage Resources

Many dynamic systems need storage, such as disk volumes, databases, or central repositories for files. Even if your application doesn't use storage directly, many of the services it does use will need it, if only for storing compute images (e.g., virtual machine snapshots and container images).

A genuinely dynamic platform manages and provides storage to applications transparently. This feature differs from classic virtualization systems, where you need to explicitly specify which physical storage to allocate and attach to each instance of compute.

Common storage resources include:

- Block storage
- Object storage
- Networked volumes
- Structured storage
- Secrets management

I'll describe each of these in more detail.

Block storage (virtual disk volumes)

You can mount a block storage volume on a server or container instance as if it was a local disk. Examples of block storage services provided by cloud platforms include AWS EBS, Azure Page Blobs, OpenStack Cinder, and GCE Persistent Disk.



Figure 3-7. A block storage volume mounted on a virtual machine

Some volumes are ephemeral, automatically created and destroyed by the platform along with the compute instances that use them. But you can also declare persistent volumes, which you can then create and attach, detach, and re-attach to compute instances. These volumes can be useful to keep data when you rebuild a server.

It can be useful to replicate volumes for availability scenarios (as discussed in [Link to Come]) and even scalability. For example, with some distributed databases, you can add a node to a cluster by duplicating a disk volume and mounting it on a newly provisioned

server. Doing this cuts the time it takes for data to synchronize to the new node since it only copies recent changes.

A block volume looks like a local disk drive to an application running on a VM, but depending on the platform implementation, it could be hosted over a network. The latency for reads and writes over the network may cause performance issues.

If performance is a problem, you should research how your platform implements block volumes. Run tests to emulate your use cases, and then tune your configuration and use of storage to get the performance you need. Your platform may provide different storage options for higher performance, such as solid-state drives or faster I/O.

Object storage

Most infrastructure platforms provide an object storage service, which you can use to store and access files over the network. Amazon's S3, Azure Block Blobs, Google Cloud Storage, and OpenStack Swift are all examples.

While only one compute instance can mount a block storage volume at a time, any instance can access object storage. So object storage is useful for creating and sharing files between instances.

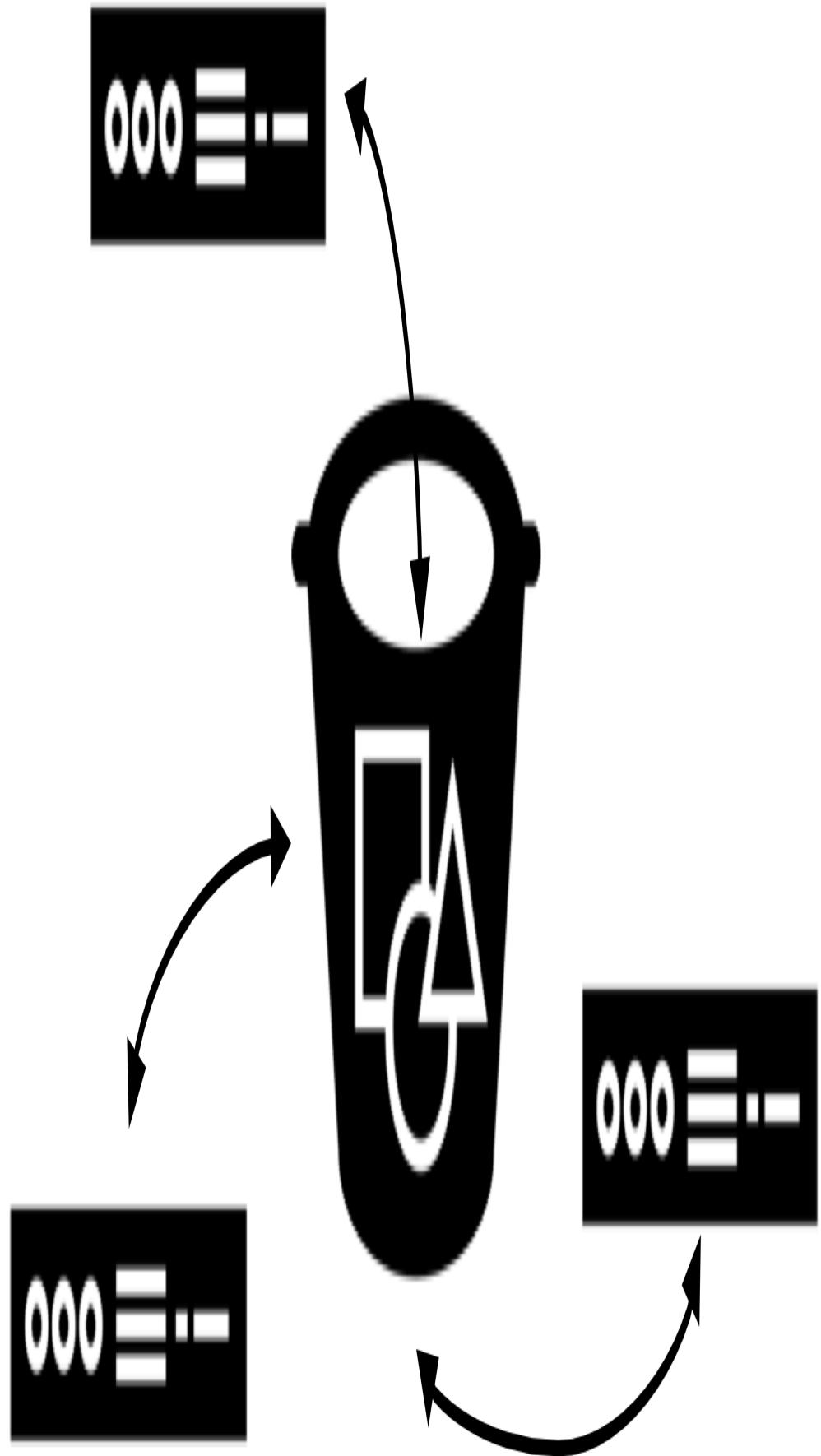


Figure 3-8. Multiple compute instances can access object storage

Object storage is usually cheaper and more reliable than block storage, but with higher latency.

Networked filesystems (shared network volumes)

There are many standard protocols for multiple servers to mount shared storage volumes locally. Examples include NFS, AFS, and SMB/CIFS⁵ (Figure 3-9). Like object storage, and unlike block storage, a networked storage volume is available to multiple compute instances at the same time. Like block storage, compute instances treat a shared network volume as if it was a local hard drive.

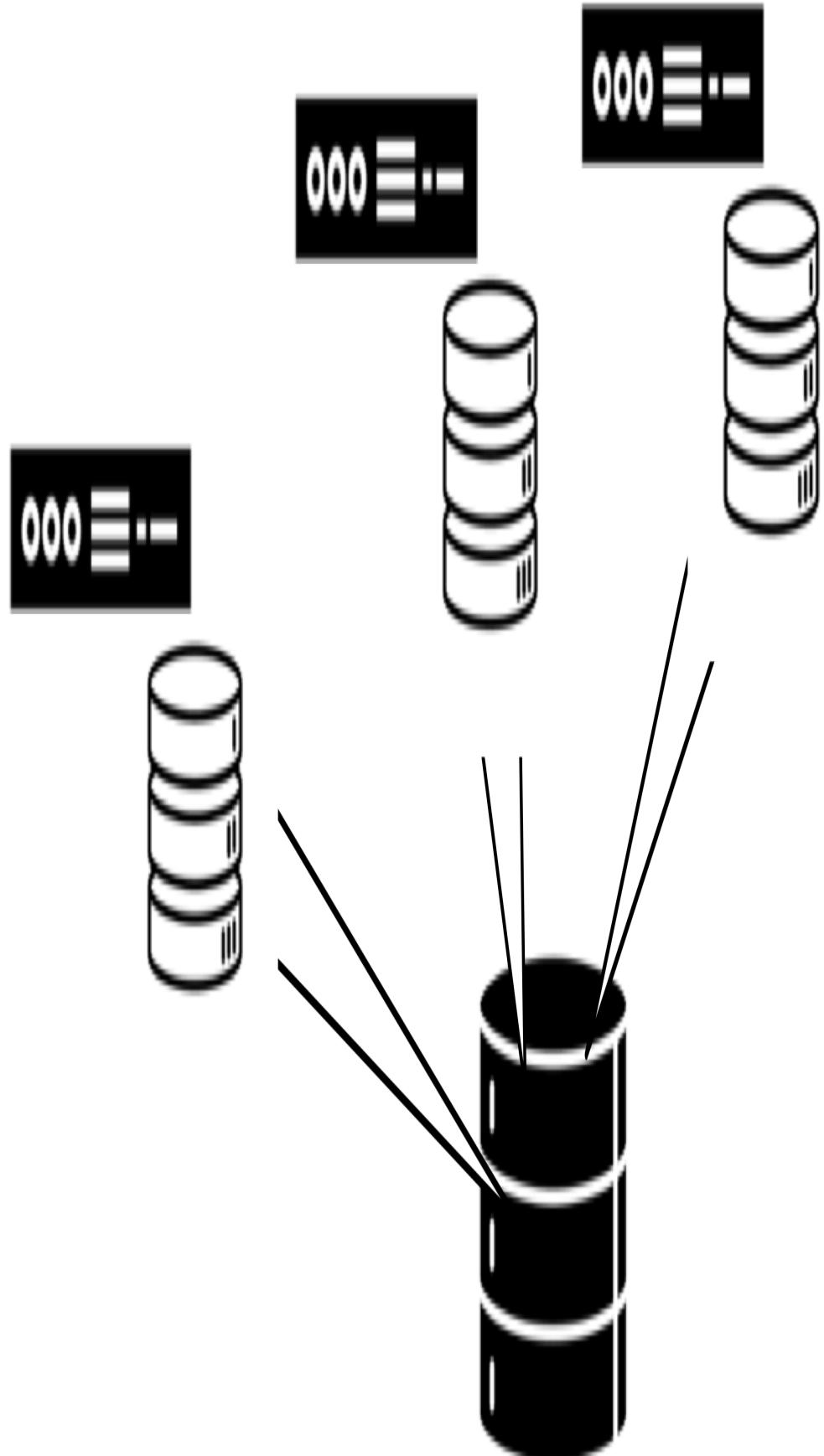


Figure 3-9. Multiple compute instances can mount the same networked file volume

Some platforms offer networked file systems as a service using standard protocols, for example, AWS EFS and Azure File Storage. Alternatively, you can run a file server on compute instances you build and manage yourself. Some people do this as part of a migration strategy, copying legacy systems from data centers into the cloud.

However, a cloud platform's managed storage services are optimized for its physical networking. Optimizing networked file sharing is challenging even when you manage your own data centers and physical networking. When using someone else's virtualized infrastructure, you'll probably get better performance and reliability by using their optimized file sharing solutions than you would by building your own.

Of course, it's always best to test and measure to be sure you're getting the results you need from your system in terms of performance and availability.

ENCRYPTION AT REST

All of these storage resource primitives should have encryption capability built in. You can easily set an option in your infrastructure code to encrypt all data stored on the resource “at rest” (as opposed to encrypting “in transit”, when it is passing over the network).

You should fully understand your threat model and how encryption does and does not address your threats. You especially need to understand how the platform manages the keys to encrypt and decrypt your data. Typically, these keys are available to the cloud platform itself. Anyone with privileged access to your platform (such as employees of your hosting vendor) may be able to gain access to these keys. Anyone with access to manage your infrastructure (such as your infrastructure team members) can use the keys to decrypt the data.

For example, even if someone is unable to gain direct access to the keys, if they have permission to use an API to read data from the encrypted storage, then the encryption does not give any protection from that person doing intentional or unintentional harm.

Know how your platform manages encryption keys, and what options you have to control access to them. For example, you may be able to create your own keys and upload them to the platform. Most important, understand your actual threat models, and make sure you understand how to use encryption to provide real value.

Structured data storage

Most hosted infrastructure platforms offer services for storing structured data. These are typically Database as a Service (DBaaS) systems managed by the provider. A DBaaS may handle things like:

- The server operating system
- Database software
- Storage devices

- Scaling clusters
- Geographical distribution
- Replication
- Backups

Ideally, it gives you a certain amount of control over configuration and optimization.

In addition to the vendor managing the database for you, a DBaaS typically integrates with the platform's other services. For example, your database can use the platform's authentication and authorization, encryption, and monitoring services.

Many DBaaS services offer managed instances of standard commercial or open source database applications, such as MySQL, Postgres, and SQL Server.

Some platforms also offer custom structured data storage services. Examples include key-value stores and formatted document stores, e.g., for storing and searching JSON or XML content. These services may be optimized to make better use of the underlying infrastructure platform to deliver high performance, geographical distribution, and availability.

The major cloud vendors offer data storage and processing services for managing, transforming, and analyzing large amounts of data. These include services for batch data processing, map-reduce, streaming, indexing and searching, and more. Understand the tradeoffs of each of these services, and test them for your workloads and use cases, to get the most value from them.

Secrets management

Any storage resource can be encrypted so you can store passwords, keys, and other information which attackers might exploit to gain privileged access to systems and resources. A secrets management service adds functionality specifically designed to help manage these kinds of resources. Features of a secrets management service may include:

- Automatically generating and rotating secrets
- Giving access to secrets to automated services so they can carry out controlled actions
- Tracking and auditing usage of secrets
- Generating short-term, single-use secrets

FOODSPIN EXAMPLE: STORAGE RESOURCES

Let's return to the Foodspin team to see how they added storage resources to their system.

The Nginx web server container images include configuration files and static content, mainly for Foodspin branding, so they don't need any persistent local storage. They do send web server logs to object storage. The team can use other tools and services to download the logs from there to analyze user traffic.

Each customer has a virtual server instance running a Java application. Each instance has a block storage volume where the application stores local files. Whenever they rebuild a server, they unmount the volume from the old server and attach it to the new one so that the application can resume service.

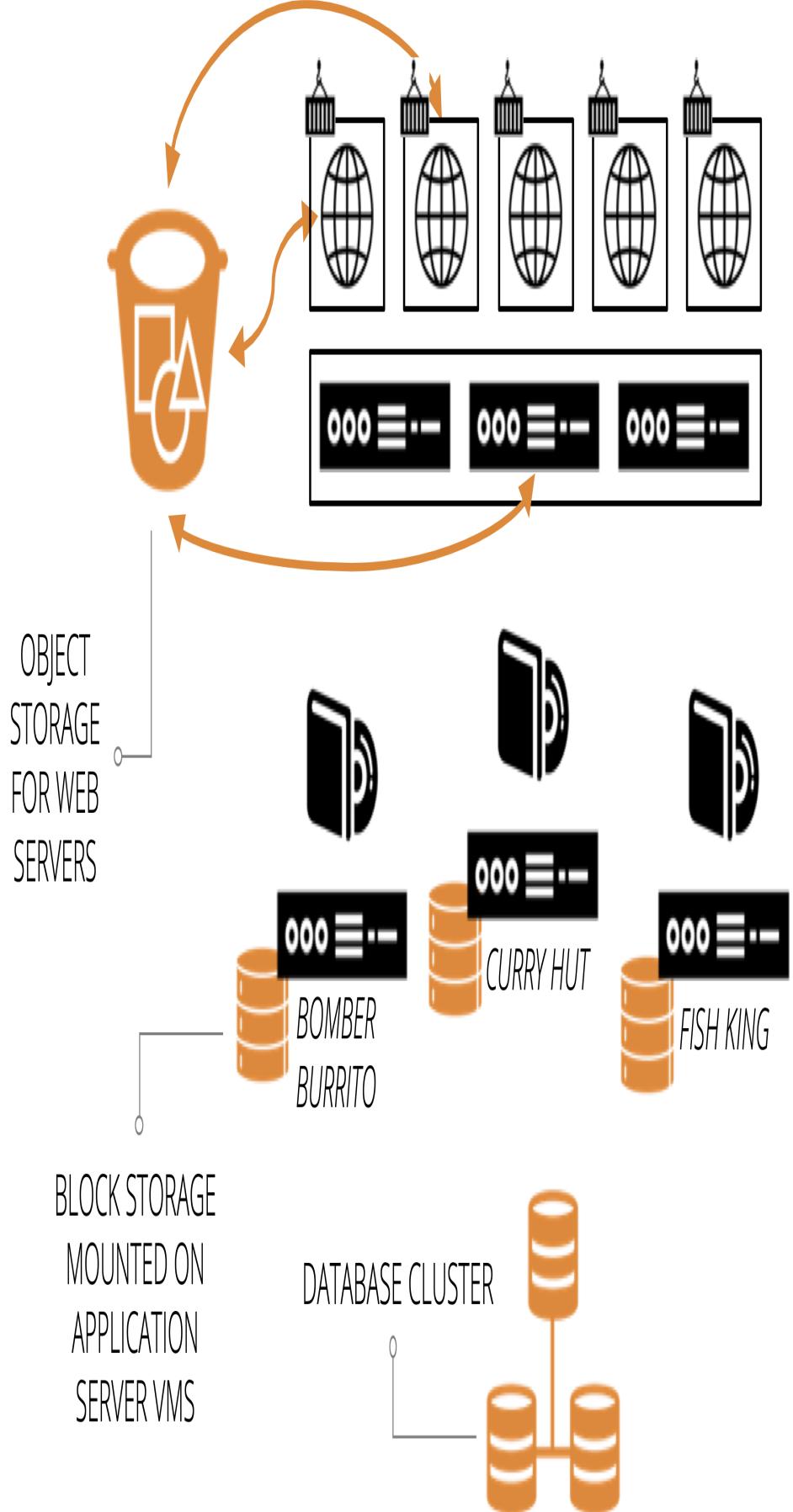


Figure 3-10. Foodspin storage resources

Each customer application also uses its own PostgreSQL database. Foodspin creates a single Postgress server cluster, using their cloud provider's DBaaS service. When the team provisions a new customer, they create a new database in this cluster. In the future, they may need more than one cluster to scale the number of customers they can support, but for the moment, this is enough.

Network Resources

As with the other types of infrastructure resources, the capability of dynamic platforms to provision and change networking on demand, from code, creates great opportunities. These opportunities go beyond changing networking more quickly; they also include much safer use of networking.

Part of the safety comes from the ability to quickly and accurately test a networking configuration change before applying it to a critical environment. Beyond this, Software Defined Networking (or SDN) makes it possible to create finer-grained network security constructs than you can do manually. This is especially true with systems where you create and destroy elements dynamically.

ZERO-TRUST SECURITY MODEL WITH SDN

A zero-trust security model⁶ secures every service, application, and other resource in a system at the lowest level. This is different from a traditional perimeter-based security model, which assumes that every device inside a secure network can be trusted.

It's only feasible to implement a zero-trust model for a non-trivial system by defining the system as code. The manual work to manage controls for each process in the system would be overwhelming otherwise.

For a zero-trust system, each new application is annotated to indicate which applications and services it needs to access. The platform uses this to automatically enable the required access and ensure everything else is blocked.

The benefits of this type of approach include:

- Each application and service has only the privileges and access it explicitly requires, which follows the principle of least privilege,
- Zero-trust, or **perimeterless** security involves putting smaller-scoped barriers and controls onto the specific resources that need to be protected. This approach avoids the need to allow broad-scoped trust zones, for example, granting trust to every device attached to a physical network. Google documents their approach to this with its [BeyondCorp security model](#).
- The security relationships between elements of the system are visible, which enables validation, auditing, and reporting.

Some typical types of networking constructs and services an infrastructure platform may support include:

- Network address blocks
- Traffic management and routing
- Network access rules
- Caches
- Service meshes

Network address blocks

Network address blocks are a fundamental structure for grouping entities involved in network communication. The top-level block in AWS is a VPC (Virtual Private Cloud). In Azure, GCP, and

others, it's a Virtual Network. The top-level block is often divided into smaller blocks, such as subnets, as shown in Figure 3-11.

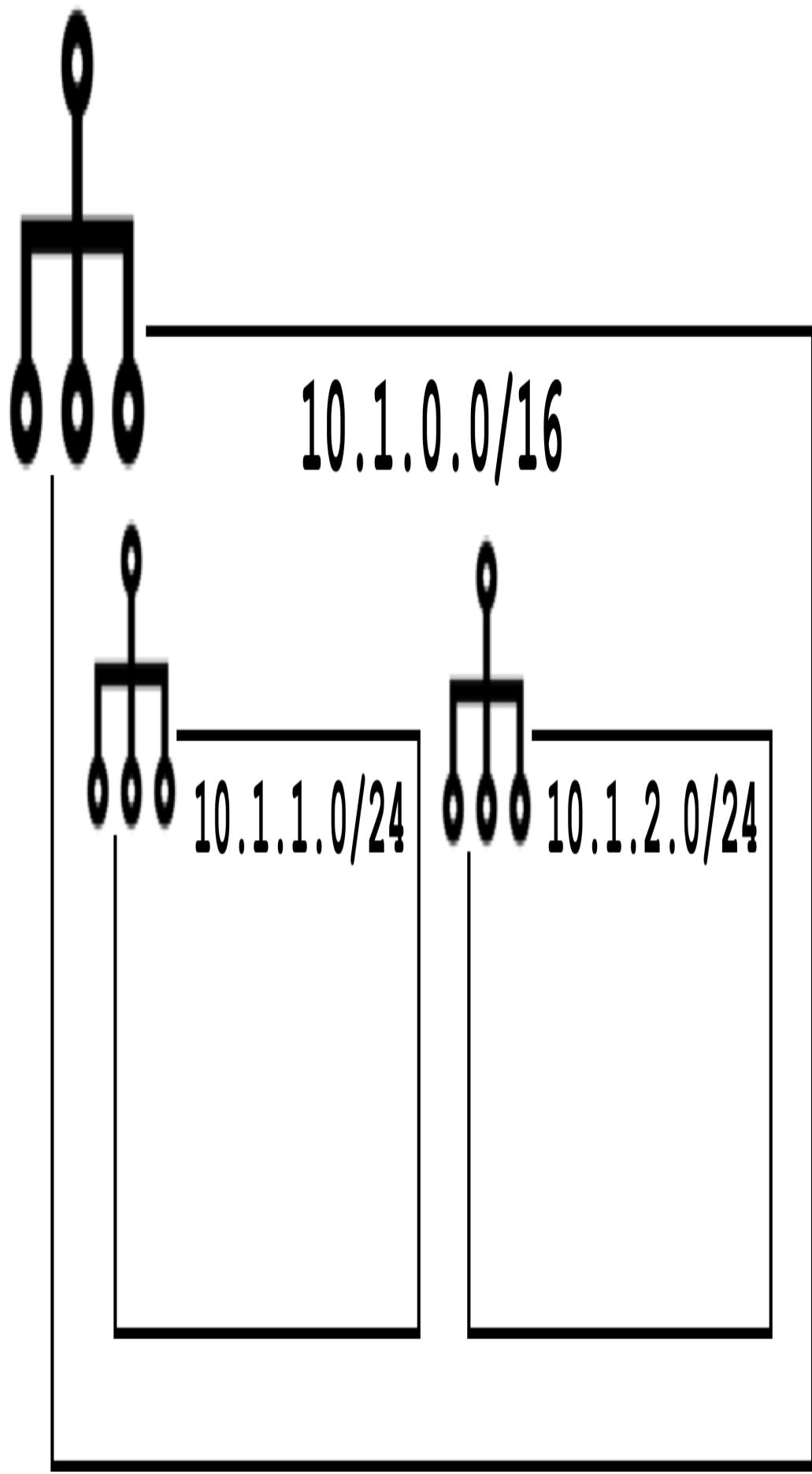


Figure 3-11. Network address blocks

The primary purpose of address blocks is for routing network traffic. Given a compute instance, you assign it an interface and address in an address block. The instance can then communicate with other entities in the same block. Traffic routing constructs (described shortly) enable communication across blocks.

You can use blocks to segregate entities by defining rules for what connections are allowed in and out of a block. It's common to define blocks in vertical tiers. For example, a "public" tier block allows connections in from the Internet; a "private" tier block only accepts connections from the public block. You can improve security with network access rules (see below), and by securing individual entities in a zero-trust model.

Address blocks may or may not correspond to physical networking. You can often assume that entities attached to the same block (e.g., in the same subnet) have lower latency when communicating with each other than if they were in separate blocks. If you require low network latency, you should research how your platform implements networking and carry out tests for your workload to find the best configuration.

AWS also uses its networking blocks (subnets) to indicate the physical location, in terms of geography and data centers, of resources attached to them. Each subnet corresponds to a particular data center, with a designated availability zone (AZ). This scheme has implications for availability - if AWS has a data center outage, it impacts all of the entities in that subnet. Assigning redundant resources (e.g., compute instances) to different subnets with

different availability zones spreads the risk across more than one location.

FOODSPIN EXAMPLE: ADDRESS BLOCKS

Our friends at Foodspin use a single high level address block (a VPC) for everything running in a single geographical region. They divide this into smaller blocks (subnets).

They use three subnets for their web server containers. They have configured the cluster to provision a host node in each subnet. Because their cloud provider uses a different physical data center for each subnet, this arrangement spreads web server workload across different locations, which improves resilience. Although a data center failure will take down some number of customer application servers, as I'll describe in a moment, they can still serve a branded error page.

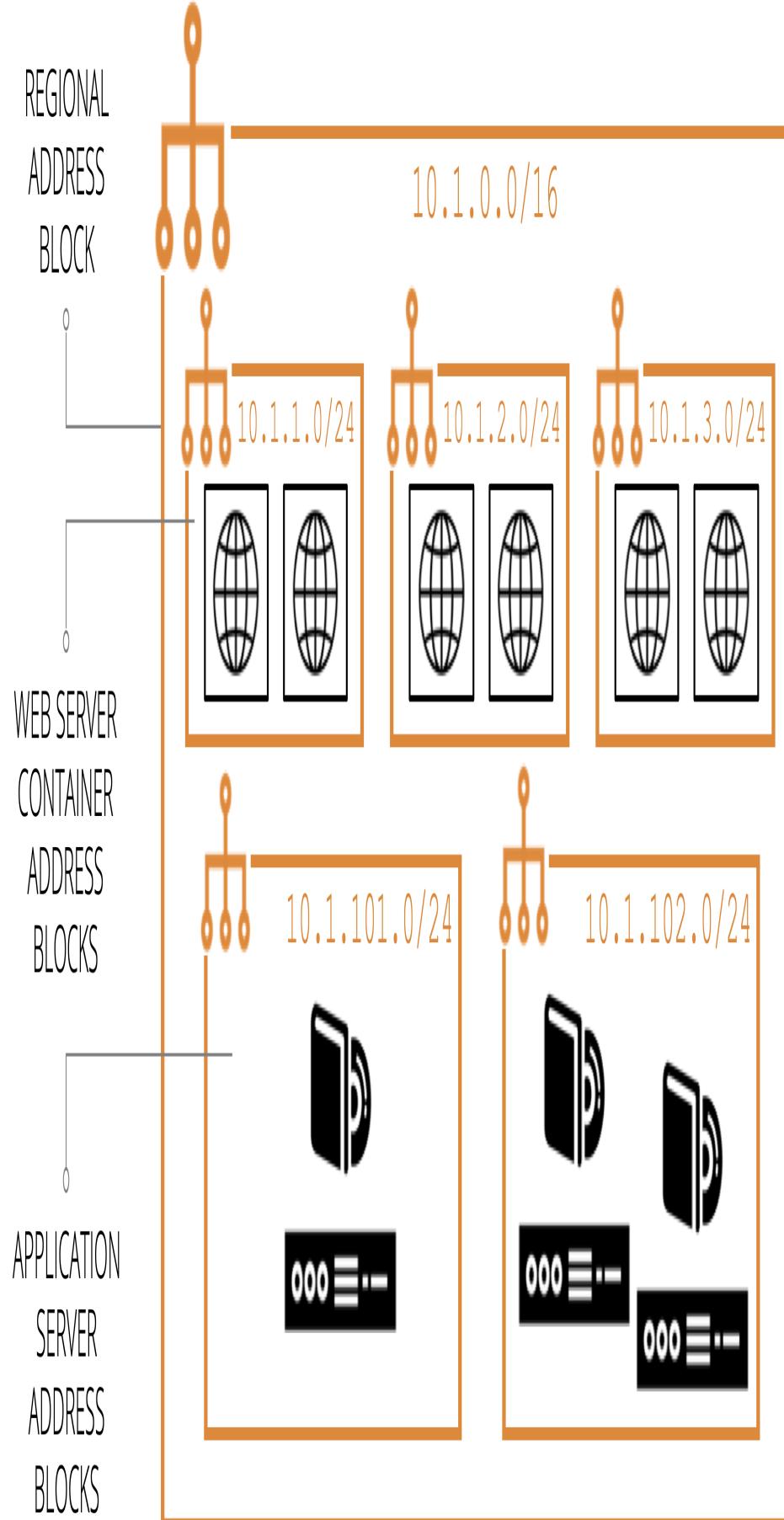


Figure 3-12. Foodspin address blocks

Foodspin creates two more subnets for the application servers. They provision each new server in one of these subnets, and the team tries to keep them spread reasonably evenly across them.

If one of their cloud vendor's data centers fails, the Foodspin team can create a new application server for each affected customer. Because these servers require persistent data, this only works if they either have access to the data volumes or recent backups. [Link to Come] discusses ways they can approach this, as well as better solutions for availability.

Traffic management and routing

There are many ways to route and manage network traffic. These include:

- Names, such as DNS entries, which are mapped to lower-level network addresses, usually IP addresses,
- Routes, which configure what traffic is allowed between and within address blocks,
- Gateways and NAT (Network Address Translation) gateways, which may be needed to direct traffic in and out of blocks, depending on the platform,
- Load balancers, which forward each connection coming into a single address to one of a pool of resources,
- Proxies, which accept connections, and may carry out transformations or routing based on different aspects of the connection,
- API gateways are proxies, usually for HTTP/S connections, which can carry out activities to handle non-core aspects of APIs, such as authentication and rate throttling. See also “Service meshes”,
- VPNs (Virtual Private Networks), which allow different address blocks to be connected across locations so that they appear to be part of a single network,

- Direct connections, a dedicated connection set up between a cloud network and another location, typically a data center or office network.

One of the issues with traffic routing in a dynamic system is that the target locations may change. For example, if you run a cluster of web servers as described in “Foodspin Example: Address blocks”, there are times when you need to rebuild or even move the cluster, which can change its IP address.

DESIGNING FOR DISPOSABILITY MEANS CHANGES ARE ROUTINE RATHER THAN CATASTROPHIC

This is a concrete example of the disposability principle (“Principle: Create disposable things”). The traditional (Iron Age) approach assumes that a cluster such as the one in this example will not change very often. When it does need to be changed, it’s a disruptive event, which probably involves downtime, people working overnight, and manually fixing problems discovered after the cutover.

The Cloud Age approach recognizes that you need to make changes to a system such as a web server cluster frequently. When a security vulnerability requires applying a patch, when a system upgrade is released, or when you need to make a configuration change to improve performance or reliability, you would like to be able to carry this out quickly and with little work, rather than needing to undertake a major project.

For this reason, you design your systems assuming that any component will go away. This not only makes it safer to carry out planned changes, it also makes it easier to handle unplanned failures.

You can use dynamic routing resources to continue routing traffic to a resource after its IP address changes. A few options that you can use to do this include:

- Define DNS entries as code and integrate this code with the code that defines the cluster. When you change the cluster, you can automatically update the DNS entries by re-applying the code.
- Define a static IP address (e.g., AWS Elastic IP or ENI), and direct external traffic to this address rather than to the cluster's address. You can map the static address to the new cluster when it changes. As with the DNS-based solution, it takes little effort to safely apply this change across the resources when you have defined them as code.
- Create a load balancer, again defining it as code and connecting it to the cluster configuration. Doing this has the added benefit that you may be able to create the new cluster while the old cluster is still running, giving you options for handling the change with little or no downtime (see [Link to Come]).

FOODSPIN EXAMPLE: NETWORK ROUTING

Given the address blocks described earlier (“Foodspin Example: Address blocks”), the team at Foodspin need some routing. This diagram shows how they’ve configured some of this:

DNS ENTRIES

bomber-

burrito.com

→ 192.168.1.52 → GATEWAY

curry-hut.biz

fish-

king.co.uk

LOAD BALANCER

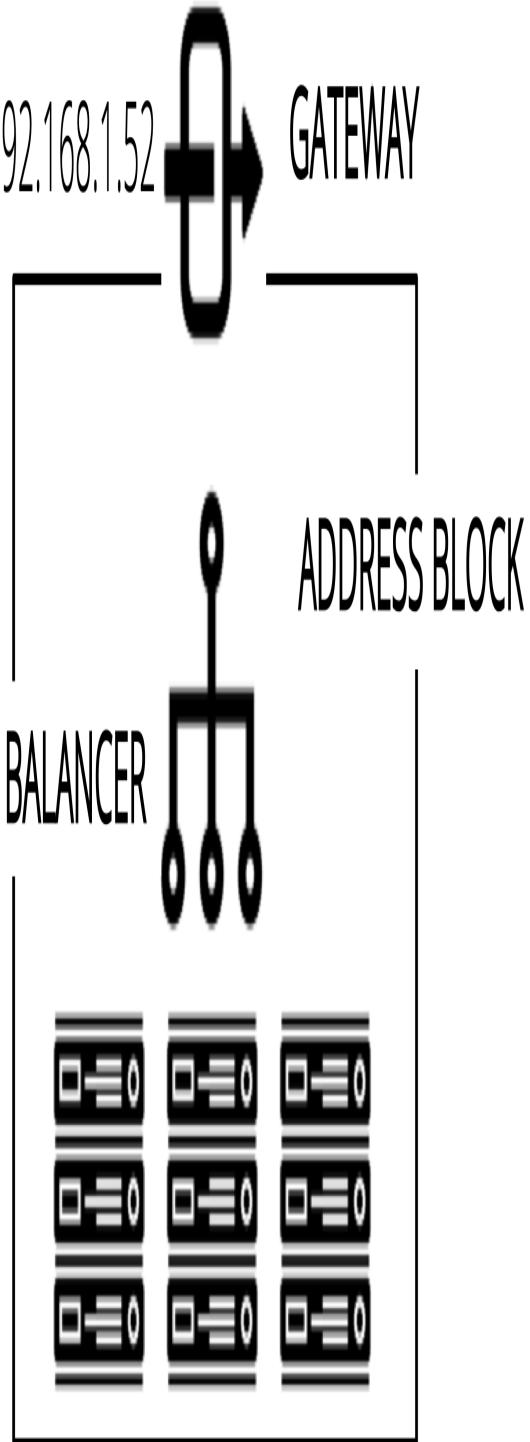


Figure 3-13. Foodspin network routing

They have a DNS name for each of their customers, which maps to a public IP addresses on the internet gateway. Their gateway routes connections, based on which address it's coming into, to a load balancer, which distributes requests among servers in a pool.

The details of networking are outside the scope of this book, so check the documentation for your platform provider, and perhaps a reference such as [Craig Hunt's TCP/IP Networking Adminstration](#).

Network access rules

Infrastructure platforms should provide a way to define access rules as code. These may map directly to physical firewall rules, or the platform may use virtual firewall mechanisms. Some platforms impose these directly on compute instances.

Defining these rules in code makes it easier to secure dynamic resources, and to secure them at a finer level of granularity than with other means. For example, your system may include application servers running in a single network segment. This arrangement permits connections between application servers, which an attacker who compromises one server can exploit to gain access to the others, as shown in [Figure 3-14](#).

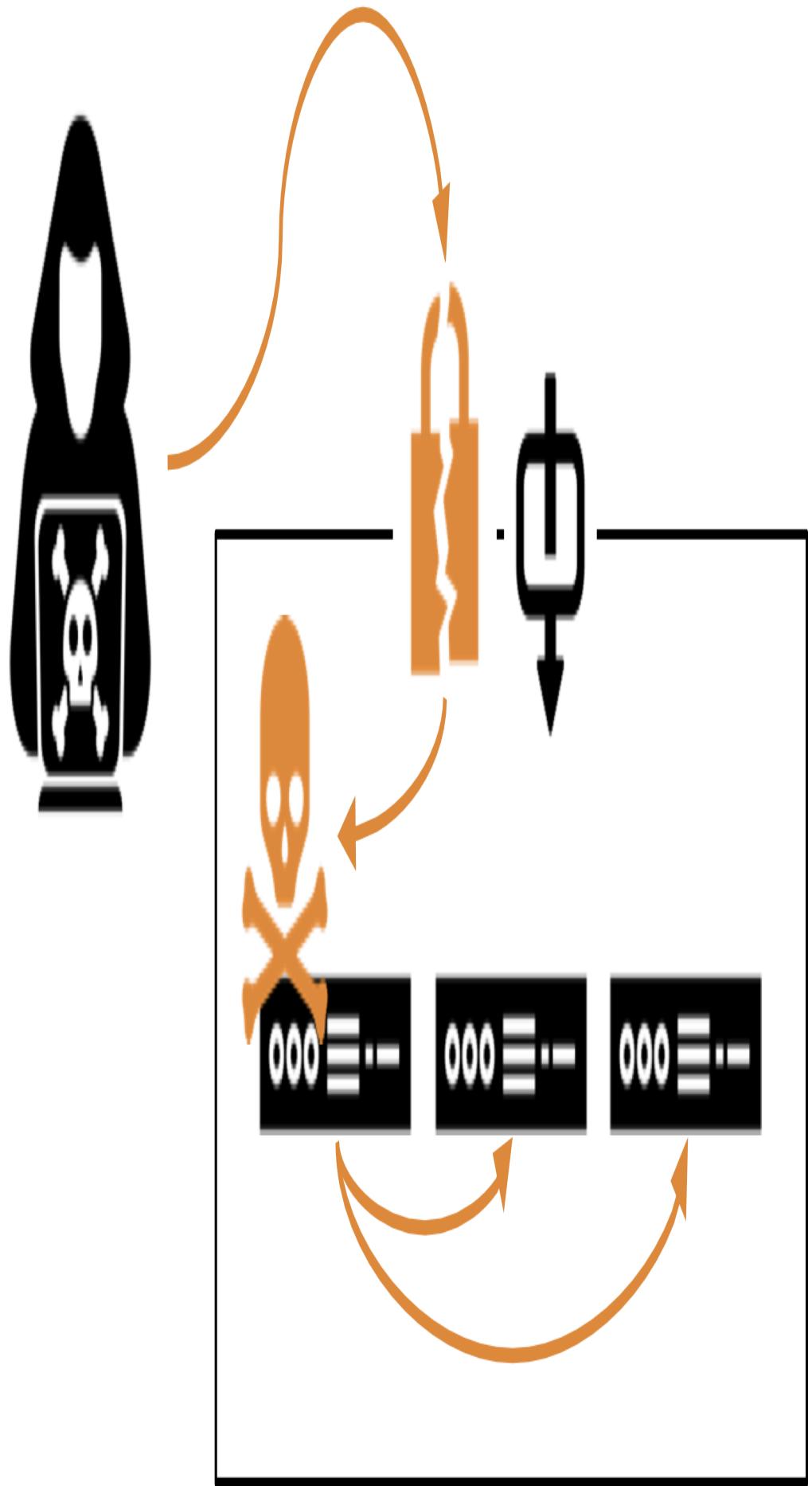


Figure 3-14. When a bad actor gains access, they are trusted by everything

Instead, you can define access rules for each application server to restrict access to between servers, as shown in Figure 3-15.

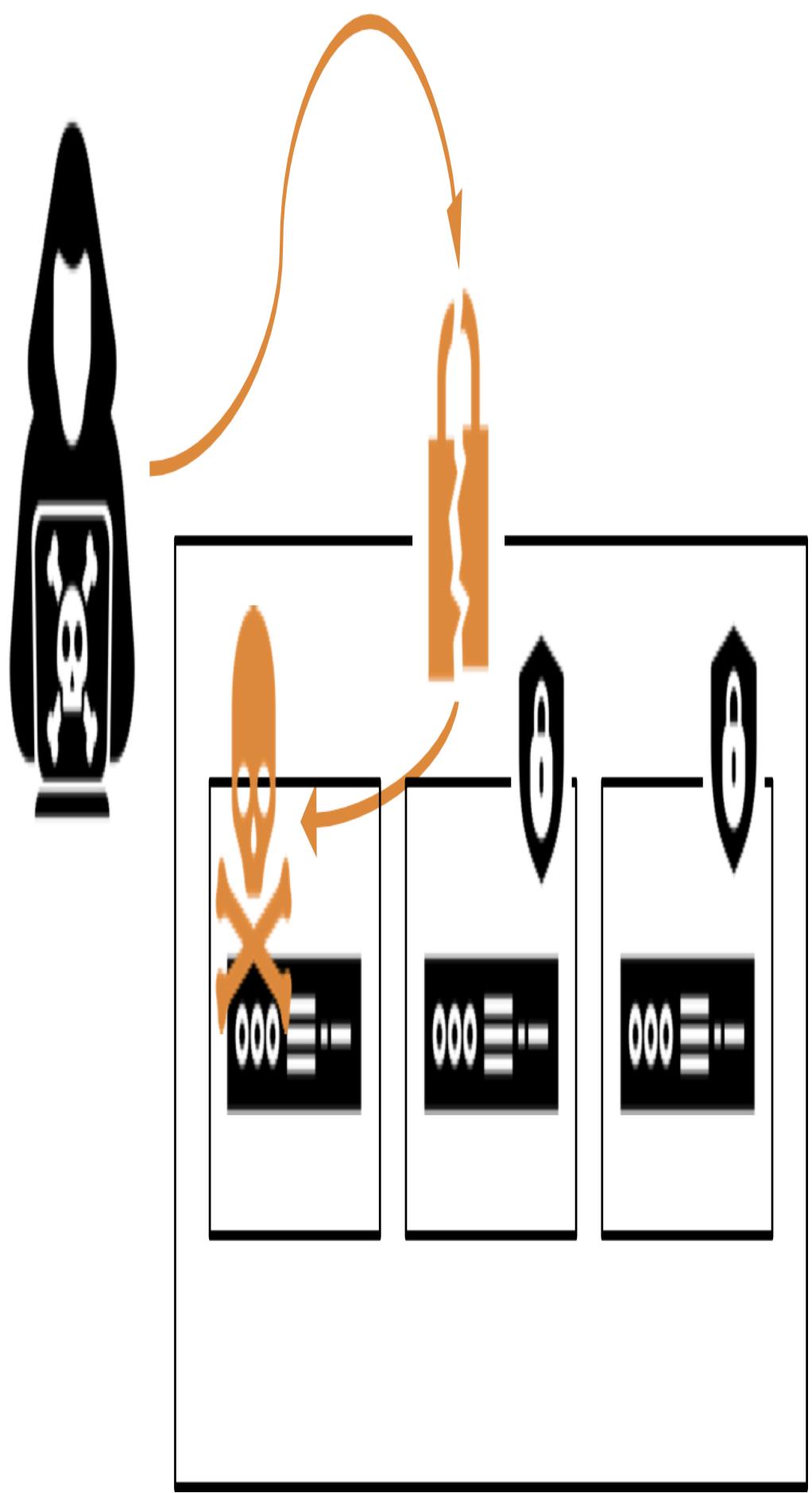


Figure 3-15. A bad actor who gains access to one server cannot access the others

In [Chapter 6](#), I'll explain how to write code that you can reuse to define multiple instances of infrastructure. Reusable code makes it easier to secure each instance at the right level.

Caches

Some services cache content to improve latency. A CDN (Content Distribute Network) is a service that can distribute static content (and in some cases executable code) to multiple locations geographically, usually for content delivered using HTTP/S.

Defining caching and CDN services as infrastructure code can make it easier to integrate other networking services. For example, code that defines a CDN can automatically update the CDN configuration when the content server is rebuilt or moves.

Service meshes

A service mesh is a decentralized network of services that dynamically manages connectivity between parts of a distributed system. It moves networking capabilities from the infrastructure layer to the application runtime layer of the model I described in [“The parts of an infrastructure system”](#). In a typical service mesh implementation, each application instance delegates communication with other instances to a sidecar process.

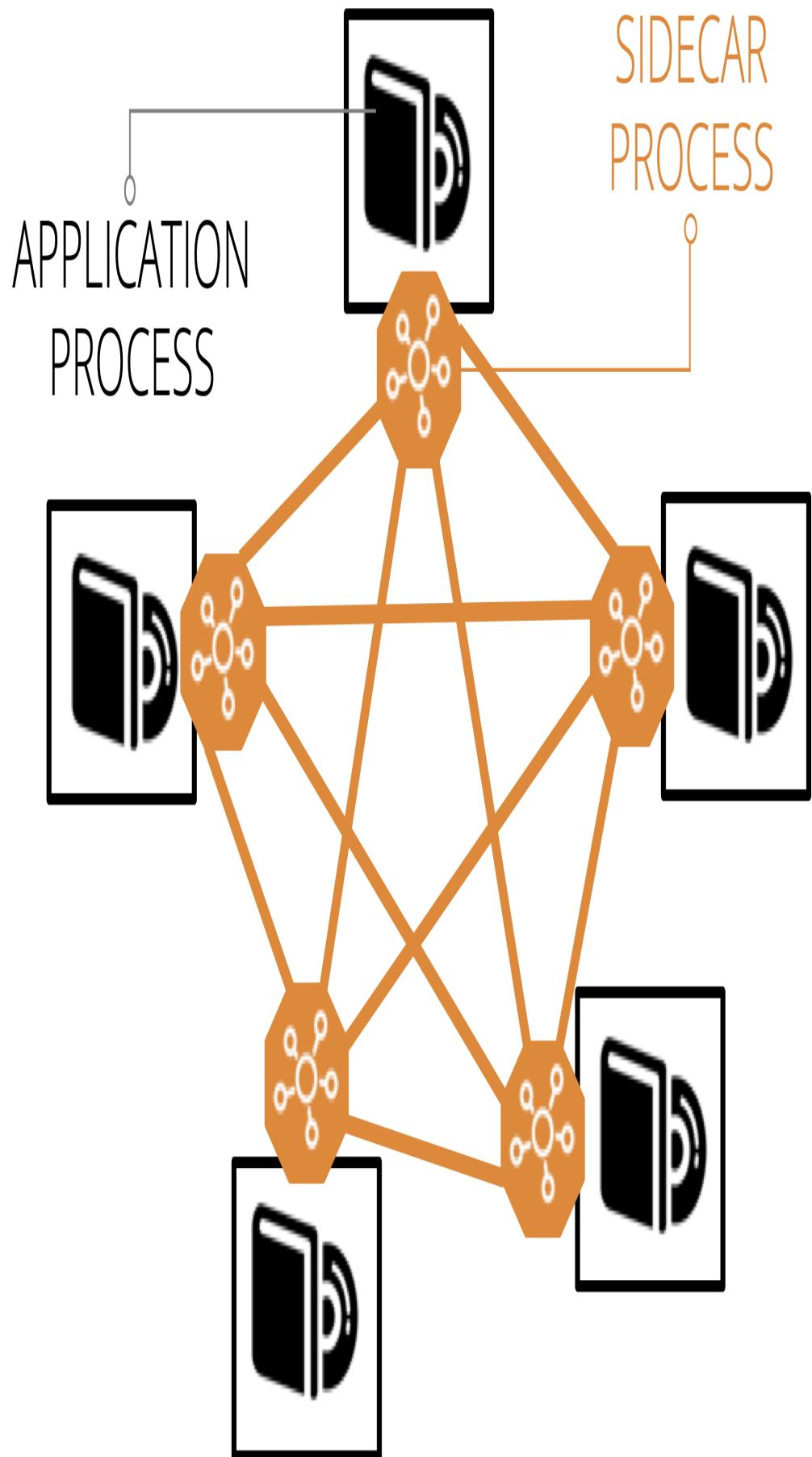


Figure 3-16. Sidecars enable communication with other processes in a service mesh

Some of the services that a service mesh can provide to applications include:

- Routing - Direct traffic to the most appropriate instance of a given application, wherever it is currently running.
Dynamic routing with a service mesh enables advanced deployment scenarios, such as blue-green and canary, as described in [Link to Come].
- Availability - Enforce rules for limiting numbers of requests, for example, circuit breakers.
- Security - handle encryption, including certificates.
- Authentication - enforce rules on which services can connect to which. Manage certificates for peer to peer authentication.
- Observability, monitoring, and troubleshooting - record connections and other events so that people can trace requests through complex distributed systems.

A service mesh works well in combination with application hosting clusters (“Application hosting cluster”). The application cluster decoupled the deployment of applications from the infrastructure layer. The service mesh then decouples application communication from the infrastructure layer. The benefits of this model are:

- Simplify application development, by moving common concerns out of the application and into the sidecar,
- Make it easier to build and improve common concerns across your estate, since you only need to deploy updates to the sidecar, without needing to make code changes to all of your applications and services,

- Handle the dynamic nature of application deployment, since the same orchestration and scheduling system that deploys and configures application instances (e.g., in containers) can deploy and configure the sidecar instances along with them.

Some examples of service meshes include Hashicorp Consul, Envoy, Istio, and Linkerd.

Service meshes are most commonly associated with containerized systems. However, you can implement the model in non-containerized systems, for example, by deploying sidecar processes onto virtual machines.

A service mesh does add complexity. As with cloud-native architectural models like microservices, a service mesh is appealing because it simplifies the development of individual applications. However, the complexity does not disappear; you've only moved it out into the infrastructure. So your organization needs to be prepared to manage this, including being ready for a steep learning process.

It's essential to keep clear boundaries between networking implemented at the infrastructure level, and networking implemented in the service mesh. Without a good design and implementation discipline, you may duplicate and intermingle concerns. Your system is harder to understand, riskier to change, and harder to troubleshoot.

MULTICLOUD, POLYCLOUD, HYBRID CLOUD

Many organizations end up hosting across multiple platforms. A few terms crop up to describe variations of this:

Hybrid Cloud

Hosting applications and services for a system across both private infrastructure and a public cloud service. People often do this because of legacy systems that they can't easily migrate to a public cloud service (such as services running on mainframes). In other cases, organizations have requirements that public cloud vendors can't currently meet, such as legal requirements to host data in a country where the vendor doesn't have a presence.

Cloud Agnostic

Building systems so that they can run on multiple public cloud platforms. People often do this hoping to avoid lock-in to one vendor. In practice, this results in locks-in to software that promises to hide differences between clouds; or involves building and maintaining vast amounts of customized code; or both.

Polycloud

Running different applications, services, and systems on more than one public cloud platform. This is usually to exploit different strengths of different platforms.

Conclusion

The different types of infrastructure resources and services covered in this chapter are the pieces that you arrange into useful systems - the “infrastructure” part of Infrastructure as Code. In the next chapter, I explain the first of the three core practices for infrastructure as code (define everything as code, validate continuously, use small pieces). Then I combine that practice with this chapter’s discussion of cloud platforms to explore patterns for using code to define infrastructure stacks (Chapter 5).

-
- 1 The US National Institute of Standards and Technology (NIST) has an excellent definition of cloud computing.
 - 2 Here’s how NIST defines Infrastructure as a Service: “The capability provided to the consumer is to provision processing, storage, networks, and other fundamental computing resources where the consumer is able to deploy and run arbitrary software, which can include operating systems and applications. The consumer

does not manage or control the underlying cloud infrastructure but has control over operating systems, storage, and deployed applications; and possibly limited control of select networking components (e.g., host firewalls).”

- 3 Virtualization technology has been around since the 1960s, but emerged into the mainstream world of x86 servers in the 2000s.
- 4 I tend to use *serverless*. FaaS is more technically accurate, but when I say FaaS, I usually have to explain what it means. The easiest way to explain what FaaS means is to say, “serverless.” I’m lazy, so I find it simpler to say “serverless” in the first place.
- 5 [Network File System](#), [Andrew File System](#), and [Server Message Block](#), respectively.
- 6 Google documents their approach to zero-trust (also known as perimeterless) with their [BeyondCorp security model](#)

Chapter 4. Core Practice: Define everything as code

In Chapter 1, I identified three core practices that help you to change infrastructure rapidly and reliably:

- Define everything as code
- Continuously validate all work in progress
- Build small, simple pieces that you can change independently

In this chapter, I explore the first of these core practices. Why would you want to define your infrastructure as code? What do you need to do this? Then I explore some issues about the nature of infrastructure coding languages.

This seemingly banal subject is a hot topic in the industry at the time of this writing, with debate raging over the nature of configuration vs. coding, and special-purpose languages vs. standard programming languages.

I'll close this chapter with some implementation principles to guide you in deciding how to organize your infrastructure code.

The goal of this chapter is to lay out the fundamental concepts of coding infrastructure. With these in place, later chapters offer specific patterns and recommendations for implementation. I explain how to organize infrastructure resources into useful units

that I call *stacks* in [Chapter 5](#) and the following chapters. I expand these to servers, clusters, and other application runtime platforms in later chapters.

Why you should define your infrastructure as code

Even given a dynamic cloud platform, there are simpler ways to provision infrastructure than writing code and running a tool. Go to the platform's web-based user interface and poke and click an application server cluster into being. Drop to the prompt, and using your command-line prowess, wield the vendor's CLI (Command-Line Interface) tool to forge an unbreakable network boundary.

But seriously, the previous chapters have explained why it's better to use code to build your systems. As a quick recap of "[Core practice: Define everything as code](#)", some of the benefits of defining things as code are:

Reusability

If you define a thing as code, you can create many instances of it. You can repair and rebuild your things quickly. 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. This makes testing more reliable. This enables continuous validation.

Transparency

Everyone can see how the thing is built by looking at the code, which helps in many ways. People can review the code and

suggest improvements. They can learn things to use in other code. They gain insight to help with troubleshooting. They can review and audit for compliance.

What you can define as code

Every infrastructure tool has a different name for its source code—for example, playbooks, cookbooks, manifests, and templates. I refer to these in a general sense as infrastructure code, or sometimes as an infrastructure definition.

Infrastructure code specifies both the infrastructure elements you want and how you want them configured. You run an infrastructure tool to apply your code to an instance of your infrastructure. The tool either creates new infrastructure, or it modifies existing infrastructure to match what you've defined in your code.

Some of the things you might define as code include:

- An infrastructure stack is a collection of elements provisioned from an infrastructure cloud platform. I write about infrastructure platforms in [Chapter 3](#), and discuss stacks in [Chapter 5](#).
- Elements of a server's configuration, such as packages, files, user accounts, and services. ([\[Link to Come\]](#))
- A server role is a collection of server elements that are applied together to a single server instance ([\[Link to Come\]](#))
- A server image definition generates an image for building multiple server instances. ([\[Link to Come\]](#))

- An application package defines how to build a deployable application artifact, including containers. ([Link to Come])
- Configuration and scripts for delivery services, which include pipelines and deployment. ([Link to Come])
- Configuration for operations services, such as monitoring checks. ([Link to Come])
- Validation rules, which include automated tests and compliance rules. (Chapter 9)

Choose tools that are configured with code

Infrastructure as Code, by definition, involves specifying your infrastructure in text-based files. You manage these files separately from the tools that you use to apply them to your system. You can read, edit, analyze, and manipulate your specifications using any tools you want.

Other infrastructure automation tools store your infrastructure specifications as data that you can't access directly. Instead, you can only use and edit the specifications by using the tool itself. The tool may have some combination of GUI, API, and command-line.

The issue with these black-box tools is that they limit the practices and workflows you can use:

- You can only version your infrastructure specifications if the tool has built-in versioning,
- You can only use Continuous Integration if the tool has a way to trigger a job automatically when you make a change,

- You can only create delivery pipelines if the tool makes it easy to version and promote your infrastructure specifications,
- You can only split monolithic infrastructure into independent pieces if the tool supports it.

LESSONS FROM SOFTWARE SOURCE CODE

The externalized configuration pattern mirrors the way most software source code works. Some development environments keep source code hidden away, such as Visual Basic for Applications. But for non-trivial systems, developers prefer keeping their source code in external files.

It is challenging to use agile engineering practices such as Test-Driven Development, Continuous Integration, and Continuous Delivery with black-box infrastructure management tools.

A tool that uses external code for its specifications doesn't constrain you to use a specific workflow. You can use an industry-standard source control system, text editor, CI server, and automated testing framework. You can build delivery pipelines using the tool that works best for you.

AUTOMATING BLACK BOX CONFIGURATION

Sometimes a black-box tool's configuration can be exported and imported by unattended scripts. This might make it possible to integrate the tool with infrastructure as code, although it will be clumsy. It's certainly not recommended for a core infrastructure tool, but could be useful for a specialized tool that doesn't have a good alternative.

This model has some limitations. It's often difficult to merge changes from different dumps, which means different team members can't work on different parts of the configuration at the same time. People must be careful about making changes to downstream instances of the tool to avoid conflicts with upstream work.

Depending on the format of the configuration dump, it may not be very transparent—it may not be easy to tell at a glance the differences between one version and the next.

Another way to approach black-box configuration is by injecting configuration automatically. Ideally, this would be done using an API provided by the tool. I have also seen this done by automating interaction with the tool's UI—for example, using scripted tools to make HTTP requests and post forms. But UI automation is brittle, breaking whenever a new version changes the UI. A well-supported API keeps backward compatibility, and so is more reliable.

The configuration injection approach allows configuration to be defined in external files. These may simply be scripts, or could involve a configuration format or DSL. If a tool is configured by automated injection from external files, then this should be the only way configuration changes are made. Changes made through the UI are likely to cause conflicts and inconsistencies.

Manage your code in a version control system

If you're defining your stuff as code, then putting that code into a version control system (VCS) is simple and powerful. By doing this, you get:

Traceability

VCS provides a history of changes, who made them, and context about why¹. This history is invaluable when debugging problems.

Rollback

When a change breaks something—and especially when multiple changes break something—it's useful to be able to restore things to exactly how they were before.

Correlation

Keeping scripts, specifications, and configuration in version control helps when tracing and fixing gnarly problems. You can correlate across pieces with tags and version numbers.

Visibility

Everyone can see each change committed to the version control system, giving the team situational awareness.

Someone may notice that a change has missed something important. If an incident happens, people are aware of recent commits that may have triggered it.

Actionability

The VCS can trigger an action automatically for each change committed. Triggers enable CI jobs and CD pipelines.

RECOMMENDATION: AVOID BRANCHING

Branching is version control system feature that allows people to work on code in separate streams. There are many popular workflows based around branches. I'll explain how these workflows can conflict with the core practice of Continuous Integration in [Link to Come].

Secrets and source code

Systems need various secrets. Your stack tool may need a password or key to use your platform's API to create and change infrastructure. You may also need to provision secrets into environments, for example making sure an application has the password for its database.

It's essential to handle these types of secrets in a secure way from the very beginning. Whether you are using a public cloud or a

private cloud a leaked password can have terrible consequences. So even when you are only writing code to learn how to use a new tool or platform, you should never put secrets into code. There are many stories of people who checked a secret into a source repository they thought was private, only to find it had been discovered by hackers who exploited it to run up huge bills.

One solution to this is to encrypt secrets in order to store them in code². Infrastructure developers and unattended systems need to be able to decrypt these secrets without storing a secret. So you still have at least one secret to manage outside of source control!

There are a few approaches for handling secrets needed by infrastructure code without actually putting them into code. These include secretless authorization, runtime secret injection, and disposable secrets.

SECRETLESS AUTHORIZATION

Many services and systems provide ways to authorize actions without using secrets. Most cloud platforms can mark a compute service-such as a virtual machine or container instance-as authorized for privileged actions.

For example, an AWS EC2 instance can be assigned an IAM profile that gives processes on the instance rights to carry out a set of API commands. If you configure a stack management tools to run on one of these instances, you avoid the need to manage a secret that might be exploited by attackers.

In some cases, secretless authorization can be used to avoid the need to provision secrets on infrastructure when it is created. For

example, an application server might need to access a database instance. Rather than a server configuration tool provisioning a password onto the application server, the database server might be configured to authorize connections from the application server, perhaps based on its network address.

Tying privileges to a compute instance or network address only shifts the possible attack vector. Anyone gaining access to that instance can exploit those privileges. You need to put in the work to protect access to privileged instances. On the other hand, someone gaining access to an instance may be able to access secrets stored there, so giving privileges to the instance may not be any worse. And a secret can potentially be exploited from other locations, so removing the use of secrets entirely is generally a good thing.

INJECTING SECRETS AT RUNTIME

When you can't avoid using secrets for stacks or other infrastructure code, you can explore ways to inject secrets at runtime. You'll normally implement it as stack parameters, which is the topic of [Chapter 8](#). I describe the details of handling secrets as parameters with each of the patterns and antipatterns in that chapter.

There are two different runtime situations to consider, local development and unattended agents. People who work on infrastructure code³ will often keep secrets in a local file that isn't stored in version control. The stack tool could read that file directly, especially appropriate if you're using the stack configuration file pattern ([“Pattern: Stack Configuration Files”](#)). Or the file could be a script that sets the secrets in environment

variables, which works well with the stack environment variables pattern (“[Pattern: Stack Environment Variables](#)”).

These approaches also work on unattended agents, such as those used for CI testing or CD delivery pipelines⁴. But you need to store the secrets on the server or container that runs the agent. Alternatively, you can use secrets management features of your agent software to provide secrets to the stack command, as with the pipeline stack parameters pattern (“[Pattern: Pipeline Stack Parameters](#)”). Another option is to pull secrets from a secrets management service (of the type described in “[Secrets management](#)”), which aligns to the stack parameter registry pattern (“[Pattern: Stack Parameter Registry](#)”).

DISPOSABLE SECRETS

A cool thing you can do with dynamic platforms is to create secrets on the fly, and only use them on a “need-to-know” basis. In the database password example, the code that provisions the database automatically generates a password and passes it to the code that provisions the application server. Humans don’t ever need to see the secret, so it’s never stored anywhere else.

You can apply the code to reset the password as needed. If the application server is rebuilt, you can re-run the database server code to generate a new password for it.

Secrets management services, such as Hashicorp Vault, can also generate and set a password in other systems and services on the fly. It can then make the password available either to the stack tool when it provisions the infrastructure, or else directly to the services that uses it, such as the application server.

Infrastructure coding languages

System administrators have been using scripts to automate infrastructure management tasks for decades. General-purpose scripting languages like Bash, Perl, Powershell, Ruby, and Python are still an essential part of an infrastructure team's toolkit.

[CFEngine](#) pioneered the use of declarative, Domain Specific Languages (DSL - see “[DSLs for infrastructure](#)” a bit later) for infrastructure management. [Puppet](#) and then [Chef](#) emerged alongside mainstream server virtualization and IaaS cloud. [Ansible](#), [Saltstack](#), and others followed.

Stack-oriented tools like [Terraform](#) and [CloudFormation](#) arrived a few years later, following the same declarative DSL model.

Recently⁵, there is a trend of new infrastructure tools that use existing general-purpose programming languages to define infrastructure. [Pulumi](#) and the [AWS CDK](#) (Cloud Development Kit) are examples, supporting languages like Typescript, Python, and Java. These newer languages use procedural language structures rather than being declarative.

The principles, practices, and patterns in this book should be relevant regardless of what language you use to implement them. The languages we use should make it easy to write code that is easy to understand, test, maintain, and improve. Let’s review the evolution of infrastructure coding languages and consider how different aspects of language choice affect this goal.

Scripting your infrastructure

Before standard tools appeared for provisioning cloud infrastructure declaratively, we wrote scripts in general-purpose, procedural languages. These used SDK (Software Development Kit) libraries to interact with the cloud provider's API.

Example 4-1 uses pseudo-code, but is similar to scripts that I wrote in Ruby with the AWS SDK. It creates a server named `my_application_server` and then runs the (fictional) `servermaker` tool to configure it.

Example 4-1. Example of procedural code that creates a server

```
import 'cloud-api-library'

network_segment = CloudApi.find_network_segment('private')

app_server = CloudApi.find_server('my_application_server')
if(app_server == null) {
    app_server = CloudApi.create_server(
        name: 'my_application_server',
        image: 'base_linux',
        cpu: 2,
        ram: '2GB',
        network: network_segment
    )
    while(app_server.ready == false) {
        wait 5
    }
    app_server.provision(
        provisioner: servermaker,
        role: tomcat_server
    )
}
```

This script combines *what* and *how*. It specifies attributes of the server, including the CPU and memory resources to provide it, what OS image to start from, and what Ansible role to apply to the server. It also implements logic: it checks whether the server

named `my_application_server` already exists, to avoid creating a duplicate server, and then it waits for the server to become ready before running Ansible on it. The script would need additional logic to handle errors, which I didn't include in the example.

The example code also doesn't handle changes to the server's attributes. What if you need to increase the RAM? You could change the script so that if the server exists, the script will check each attribute and change it if necessary. Or you could write a new script to find and change existing servers.

More realistic scenarios include multiple servers of different types. In addition to our application server, my team had web servers and database servers. We also had multiple environments, which meant multiple instances of each server.

Teams I worked with often turned simplistic scripts like the one in this example into a multi-purpose script. This kind of script would take arguments specifying the type of server and the environment, and use these to create the appropriate server instance. We evolved this into a script that would read configuration files that specify various server attributes.

I was working on a script like this, wondering if it would be worth releasing it as an open-source tool, when Hashicorp released the first version of Terraform.

Building infrastructure with declarative code

Terraform, like most other stack provisioning tools and server configuration tools, uses a declarative language. Rather than a procedural language, which executes a series of statements using

control flow logic like *if* statements and *while* loops, a declarative language is a set of statements that declare the result you want.

Example 4-2 creates the same server as Example 4-1. The code in this example (as with most code examples in this book) is a fictional language⁶.

Example 4-2. Example of declarative code

```
virtual_machine:  
  name: my_application_server  
  source_image: 'base_linux'  
  cpu: 2  
  ram: 2GB  
  network: private_network_segment  
  provision:  
    provisioner: servermaker  
    role: tomcat_server
```

This code doesn't include any logic to check whether the server already exists or to wait for the server to come up before running Ansible. The tool that applies the code implements this logic, along with error-handling. The tool also checks the current attributes of infrastructure against what is declared, and work out what changes to make to bring the infrastructure in line. So to increase the RAM of the application server in this example, you can edit the file and re-run the tool.

Declarative infrastructure tools like Terraform and Chef keep the *what* and *how* separate. The code you write as a user of the tool only declares the attributes of your infrastructure. The tool implements the logic for how to make it happen. As a result, your code is cleaner and more direct.

IDEMPOTENCY

To continuously synchronize system definitions, the tool you use for this must be idempotent. No matter how many times you run it, the outcome is the same. If you run a tool that isn't idempotent multiple times, it might make a mess of things.

Here's an example of a shell script that is not idempotent:

```
echo "spock:*:1010:1010:Spock:/home/spock:/bin/bash" \  
    >> /etc/passwd
```

If you run this script once you get the outcome you want: the user *spock* is added to the */etc/passwd* file. But if you run it ten times, you'll end up with ten identical entries for this same user.

With an idempotent infrastructure tool, you specify how you want things to be:

```
user:  
  name: spock  
  full_name: Spock  
  uid: 1010  
  gid: 1010  
  home: /home/spock  
  shell: /bin/bash
```

No matter how many times you run the tool with this code, it will ensure that only one entry exists in the */etc/passwd* file for the user *spock*. No unpleasant side effects.

DSLs for infrastructure

In addition to being declarative, infrastructure tools often use their own DSL⁷.

The advantage of a DSL for infrastructure code is keeping the code simple and focused.

A general-purpose language needs extra syntax, such as declarations of variables and references to class structures in a DSL:

```
import 'cloud-api-library'  
app_server = CloudApi.find_server('my_application_server')
```

```
if(app_server == null) {  
    app_server = CloudApi.create_server(name: 'my_application_server')
```

A DSL can strip this to the most relevant elements:

```
virtual_machine:  
  name: my_application_server
```

The return of general-purpose languages for infrastructure

Newer tools, such as Pulumi and the AWS CDK, bring general-purpose languages back to infrastructure coding. There are several arguments for doing this, some more convincing than others.

Configuration isn't real code

Some folks take the phrase “Infrastructure as Code” to heart. They argue that declarative languages are just configuration, not a “real” language. Personally, I’m not bothered if someone disparages my code as being mere configuration. I still find it useful to keep “what” and “how” separate, and to avoid writing repetitive, verbose code.

It's useful not to have to learn a new language

Using a popular language like JavaScript means more people can learn to write infrastructure as code since they don’t need to learn a peculiar special-purpose language. I have sympathy for making it easy for people to adopt infrastructure as code. But I don’t think a new language syntax, especially a simple declarative language, is as hard to learn as the domain-specific aspects of infrastructure code like networking constructs.

Infrastructure DSLs are not well-supported by development tools

This is generally true. There are many mature IDE’s⁸ for languages like JavaScript, Python, and Ruby. These have loads of features, like syntax highlighting and code refactoring, that

help developers to be more productive. Rather than discouraging the use of new languages (of any kind), I would love to see vendors improve their support for infrastructure coding languages.

Proper support for libraries helps you to simplify code

Most infrastructure DSLs let you write modules (as I'll discuss in [Chapter 6](#)). But these are not as rich and flexible as libraries in mature, procedural languages like JavaScript, Python, and Ruby. I discuss reusable modules and libraries for stacks in [Chapter 6](#).

Using the same language lets you combine infrastructure and application code

Examples of this show application code that provisions its own infrastructure. People who've been developing web applications for a while recall how nifty it was to be able to embed SQL statements into HTML code. Experience has shown that this does not lead to cleanly designed, maintainable systems. It is often useful to specify actions for a tool to trigger on specific changes to infrastructure, such as provisioning newly created resources. But this can be done without intermingling the different styles of code.

Infrastructure languages are less testable

Also true, although there are several reasons for this. One is that the ecosystem of testing tools for infrastructure is not as mature as that for other types of code. Another reason is that testing declarations requires a different approach to testing logic. A third reason is that testing code that provisions infrastructure has much longer feedback loops. I delve into these in "[Challenges with testing infrastructure code](#)".

The swing back towards general-purpose languages for infrastructure is new⁹. Some of the arguments threaten to regress us to codebases filled with verbose spaghetti code mingling

configuration, application logic, and repetitive utility code. But I expect that this is just one step on a path to languages that support better coding.

For many teams today, the challenges with their codebase do not come from the language they use. Regardless of language, they need ways to keep their code clean, well-organized, and easy to maintain.

Implementation Principles for defining infrastructure as code

To update and evolve your infrastructure systems easily and safely, you need to keep your codebase clean: easy to understand, test, maintain, and improve. Code quality is a familiar theme in software engineering. The following implementation principles are guidelines for designing and organizing your code to support this goal.

Implementation Principle: Avoid mixing different types of code

As discussed earlier, some infrastructure coding languages are procedural, and some are declarative¹⁰. Each of these paradigms has its strengths and weaknesses.

A particular scourge of infrastructure is code that mixes both declarative and procedural code, as in Example 4-3. This code includes declarative code for defining a server as well as procedural code that determines which attributes to assign to the server depending on the server role and environment.

Example 4-3. Example of mingled procedural and declarative code

```
for ${env} in ["test", "staging", "prod"] {
    for ${server_role} in ["app", "web", "db"] {
        server:
            name: ${server_role}-${env}
            image: 'base_linux'
            cpu: if(${env} == "prod" || ${env} == "staging") {
                4
            } else {
                2
            }
            ram: if(${server_role} == "app") {
                "4GB"
            } else if (${server_role} == "db") {
                "2GB"
            } else {
                "1GB"
            }
            provision:
                provisioner: servermaker
                role: ${server_role}
    }
}
```

Mixed declarative and procedural code is a design smell¹¹<https://martinfowler.com/bliki/CodeSmell.html>. A “smell” is some characteristic of a system that you observe that suggests there is an underlying problem. In the example from the text, code that mixes declarative and procedural constructs is a smell. This smell suggests that your code may be trying to do multiple things and that it may be better to pull them apart into different pieces of code, perhaps in different languages.]. It’s not easy to understand this code, which makes it harder to debug, and harder to change without breaking something.

The messiness of this code comes from intermingling two different concerns, which leads to the next implementation principle.

Implementation Principle: Separate infrastructure code concerns

A common reason for messy code is that it is doing multiple things. These things may be related, but teasing them apart can make them easier to distinguish, and improve the readability and maintainability of the code.

Example 4-3 does two things: it specifies the infrastructure resource to create, and it configures that resource differently in different contexts. The example illustrates two of the four most common concerns of infrastructure code:

Specification

Specifications define the shape of your infrastructure. Your server has specific packages, configuration files, and user accounts. Declarative languages work well for this because specifications are *what* you want. Specifications are what most people mean when they talk about infrastructure code.

Configuration

Configuration defines the things that vary when you provision different instances of infrastructure. Different application servers built from a single specification may need different amounts of RAM. You may want to deploy different applications onto otherwise identical servers. Configuration is almost always declarative.

Execution

Execution applies specification and configuration to the actual infrastructure resources. Together, the specification and configuration declare *what* you want. The orchestration is about *how* to make that happen. Procedural or functional code usually works best for orchestration. Ideally, you should use an off the shelf tool for this rather than writing your own code.

Orchestration

Orchestration combines multiple specifications and configurations. For example, you may need to create a server in the cloud platform, and then install packages on the server. Or you may need to create networking structures, and then create multiple servers attached to those structures.

SEPARATING SPECIFICATION AND CONFIGURATION

Let's take a simple example of code that mixes concerns and split it into two cleaner pieces of code. This example specifies an application server, assigning it to a different network depending on which customer uses it:

```
virtual_machine:  
  name: application_server_${CUSTOMER}  
  source_image: 'base_linux'  
  provision:  
    tool: servermaker  
    role: foodspin_application  
  network: $(switch ${CUSTOMER}) {  
    "bomber_burrito": us_network  
    "curry_hut": uk_network  
    "burger_barn": au_network  
  }
```

The code uses the CUSTOMER parameter to choose the network for the server.

This code is hard to test because any test instance needs to specify a customer. Using a real customer couples your test code with that customer's configuration, which makes the tests brittle. You could create a test customer. But then you need to add the configuration for your fake customer to your infrastructure code. Mingling code

and configuration between test and production descends even farther into the depths of poor code.

Let's go in the opposite direction, and pull the specification into its own code:

```
virtual_machine:  
  name: application_server_${CUSTOMER}  
  source_image: 'base_linux'  
  provision:  
    tool: servermaker  
    role: foodspin_application  
  network: ${NETWORK}
```

This code is more straightforward than the previous example. It only includes code for the things which are common to all application server instances. Because this doesn't vary, we don't need logic, so this fits nicely as declarative code.

The only parts of the specification which vary are set using variables, CUSTOMER to give the server a unique name, and NETWORK to assign it to a unique network. There are different patterns for assigning these variables, which are the subject of an entire chapter of this book (Chapter 8).

To illustrate the separation of concerns, let's use a script to work out the configuration for the server. The script works out which network to assign the server to based on the environment, as before:

```
switch (${CUSTOMER}) {  
  case 'bomber_burrito':  
    return 'us_network'  
  case 'curry_hut':  
    return 'uk_network'  
  case 'burger_barn':
```

```
    return 'au_network'  
}
```

Each of these two pieces of code is easier to understand. If you need to add a new customer, you can easily add it to the configuration code, without confusing things. This example doesn't really need procedural code to work out the configuration. But it illustrates that if you find the need for more complicated logic, it's cleaner to separate it from the declarative code.

[Link to Come] gives more detailed advice on how to implement separation of concerns in your codebase.

Implementation Principle: Treat infrastructure code like real code

To keep an infrastructure codebase clean, you need to treat it as a first-class concern. Too often, people don't consider infrastructure code to be "real" code. They don't give it the same level of engineering discipline as application code.

Design and manage your infrastructure code so that it is easy to understand and maintain. Follow code quality practices, such as code reviews, pair programming, and automated testing. Your team should be aware of technical debt and strive to minimize it.

CODE AS DOCUMENTATION

Writing documentation and keeping it up to date can be too much work. For some purposes, the infrastructure code is more useful than written documentation. It's always an accurate and updated record of your system.

- New joiners can browse the code to learn about the system,
- Team members can read the code, and review commits, to see what other people have done,
- Technical reviewers can use the code to assess what to improve,
- Auditors can review code and version history to gain an accurate picture of the system.

Infrastructure code is rarely the only documentation required. High-level documentation is helpful for context and strategy. You may have stakeholders who need to understand aspects of your system but who don't know your tech stack.

You may want to manage these other types of documentation as code. Many teams write [Architecture Decision Records \(ADRs\)](#) in a markup language and keep them in source control.

You can automatically generate useful material like architecture diagrams from code. You can put this in a change management pipeline to update diagrams every time someone makes a change to the code.

Conclusion

In this chapter, I explored the core practice of defining your system as code. This included the key prerequisites for this practice, considerations of different types of infrastructure coding languages, and a few principles for good code design. The next core practice, continuously validating your code, builds on this material.

But before I cover that in [Chapter 9](#), I'll describe specific patterns and antipatterns for implementing this first practice in the context of infrastructure stacks.

[Chapter 5](#) explains how to use infrastructure code to provision resources from your cloud platform into useful groups I call

stacks. Chapter 6 covers the use of modules within stacks. Then, Chapter 7 describes how to structure your stack code to create multiple environments. After that, Chapter 8 offers patterns for managing the configuration of different stack instances across environments.

- 1 Context about why depends on people to write useful commit messages
- 2 `git-crypt`, `blackbox`, `sops`, and `transcrypt` are a few tools that help you to encrypt secrets in a git repository. Some of these tools integrate with cloud platform authorization, so unattended systems can decrypt them.
- 3 I explain how people can work on stack code locally in more detail in [Link to Come].
- 4 I describe how these are used in [Link to Come]
- 5 “Recently” as I write this in late 2019
- 6 I use this pseudo-code language to illustrate the concepts I’m trying to explain, without tying them to any specific tool.
- 7 Martin Fowler and Rebecca Parsons define a DSL as a “small language, focused on a particular aspect of a software system, in their book *Domain-Specific Languages* (Addison-Wesley Professional)
- 8 Integrated Development Environment, a specialized editor for programming languages.
- 9 Again, I’m writing this in late 2019. By the time you read this, things will have moved forward in one direction or another.
- 10 Object-Oriented Programming (OOP) and Functional Programming are two other programming paradigms used in application software development. Although there are a few examples of tools that use each of these (Riemann), neither is common with infrastructure code. There is no reason tools couldn’t use them with the domain. But even with OOP or functional programming, the advice in this section would still apply: don’t write code that mixes language paradigms.
- 11 The term *design smell* derives from [code smell]

Part II. Working With Infrastructure Stacks

Chapter 5. Building Infrastructure Stacks as Code

In [Chapter 3](#), I said that an infrastructure platform is a pool of infrastructure resources that you can provision and change on demand using an API. In [Chapter 4](#), I explained the value of using code to define the infrastructure for your system. This chapter puts these together by describing patterns for implementing code to manage infrastructure.

The concept that I use to talk about doing this is the infrastructure stack.

What is an infrastructure stack?

An Infrastructure Stack is a collection of infrastructure resources that you define, provision, and update as a unit.

You write source code to define the elements of a stack, which are resources and services that your infrastructure platform provides. For example, your stack may include a virtual machine (“[Compute Resources](#)”), disk volume (“[Storage Resources](#)”), and a subnet (“[Network Resources](#)”).

You run a stack management tool, which reads your stack source code and uses the cloud platform’s API to assemble the elements

defined in the code to provision an instance of your stack.

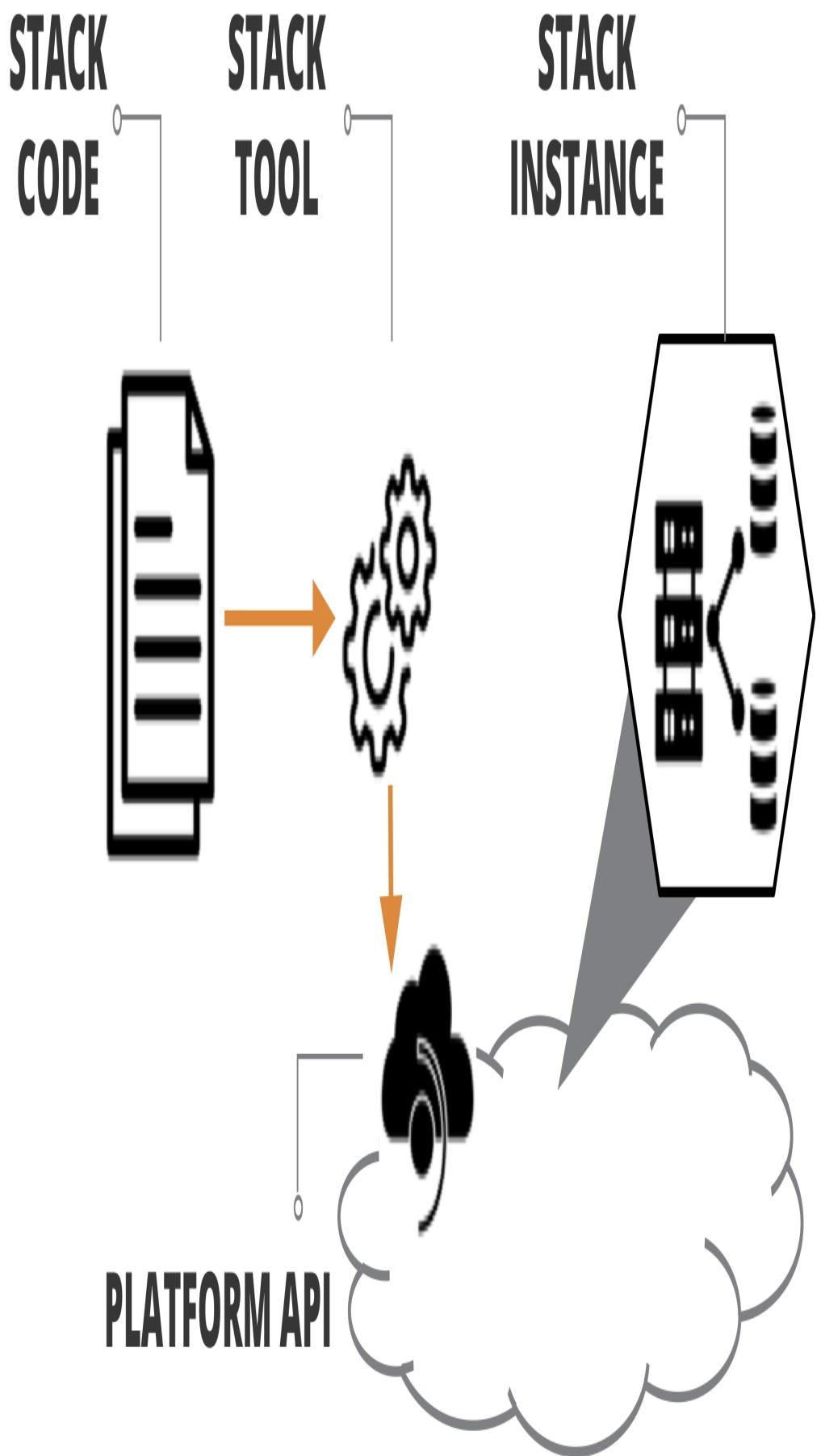


Figure 5-1. An infrastructure stack is a collection of infrastructure elements managed as a group.

Examples of stack management tools include:

- [Hashicorp Terraform](#)
- [AWS CloudFormation](#)
- [Azure Resource Manager](#)
- [Google Cloud Deployment Manager](#)
- [OpenStack Heat](#)
- [Pulumi](#)

Some server configuration tools (which I'll talk about much more in [Link to Come]) have extensions to work with infrastructure stacks. Examples of these are [Ansible Cloud Modules](#), [Chef Provisioning](#) (now end-of-lifed¹), [Puppet Cloud Management](#), and [Salt Cloud](#).

“STACK” AS A TERM

Most stack management tools don't call themselves stack management tools. Each tool has its own terminology to describe the unit of infrastructure that it manages. In this book, I'm describing patterns and practices that should be relevant for any of these tools.

I've chosen to use the word “stack.”

Various people have told me there is a far better term for this concept than “stack.” Each of these people had a completely different word in mind. As of this writing, there is no agreement in the industry as to what to call this thing. So until there is, I'll continue to use the word “stack.”

Stack code

Each stack is defined by source code that declares what infrastructure elements it should include. Terraform code (.tf files) and CloudFormation templates are both examples of infrastructure stack code. A stack project contains the source code that defines the infrastructure for a stack.

Example 5-1 shows the folder structure for a stack source code project. The language and tool for this example are fictitious. I use pseudo-code examples throughout this chapter.

Example 5-1. Project folder structure of a stack project using a fictitious tool

```
stack-project/
  └── src/
    ├── dns.infra
    ├── load_balancers.infra
    ├── networking.infra
    └── webserver.infra
  └── test/
```

Stack instance

You can use a single stack project to provision more than one stack instance. When you run the stack tool for the project, it uses the platform API to ensure the stack instance exists, and to make it match the project code. If the stack instance doesn't exist, the tool creates it. If the stack instance exists but doesn't exactly match the code, then the tool modifies the instance to make it match.

I often describe this process as “applying” the code to an instance.

If you change the code and rerun the tool, it changes the stack instance to match your changes. If you run the tool one more time without making any changes to the code, then it should leave the stack instance as it was.

Configuring servers in a stack

Infrastructure codebases for systems that aren't fully container-based or serverless application architecture tend to include much code to provision and configure servers. Even container-based systems need to build host servers to run containers. The first mainstream infrastructure as code tools, like CFEngine, Puppet, and Chef, were used to configure servers.

You should decouple code that builds servers from code that builds stacks. Doing this makes the code easier to understand, simplifies changes by decoupling them, and supports reusing and testing server code.

Stack code typically specifies what servers to create, and passes information about the environment they will run in, by calling a server configuration tool. Example 5-2 is an example of a stack definition that calls the fictitious `servermaker` tool to configure a server.

Example 5-2. Example of a stack definition calling a server configuration tool

```
virtual_machine:  
  name: appserver-burgerbarn-${environment}  
  source_image: foodspin-base-appserver  
  memory: 4GB  
  provision:  
    tool: servermaker  
    parameters:  
      maker_server: maker.foodspin.io  
      role: appserver  
      environment: ${environment}
```

This stack defines an application server instance, created from a server image called `foodspin-appserver`, with 4 GB of RAM.

The definition includes a clause to trigger a provisioning process that runs servermaker. The code also passes several parameters for the servermaker tool to use. These parameters include the address of a configuration server (`maker_server`), which hosts configuration files, and a role, `appserver`, which servermaker uses to decide which configurations to apply to this particular server. It also passes the name of the environment, which the configurations can use to customize the server.

[Link to Come] describes patterns for managing servers as code.

Patterns and antipatterns for structuring stacks

One challenge with infrastructure design is deciding how to size and structure stacks. You could create a single stack code project to manage your entire system. But this becomes unwieldy as your system grows. In this section, I'll describe patterns and antipatterns for structure infrastructure stacks.

THE DESCRIPTION FORMAT FOR PATTERNS AND ANTIPATTERNS

I use patterns and antipatterns throughout this book to describe potential ways to solve common problems². Unlike principles, which are intended as rules to follow, a given pattern may or may not be appropriate for your situation. One way to think of these is that principles help you to decide which pattern is right in your context.

An antipattern is a solution that is rarely appropriate. The reason for describing an antipattern is to help you to recognize it and to understand why it's problematic. People implement antipatterns either because they don't realize its pitfalls, or unintentionally. For example, someone might implement “Antipattern: One-shot Module” because it seems like a reasonable way to organize their project code. On the other hand, people don't set out to code a “Antipattern: Spaghetti Module”, it just happens over time.

The description of each pattern and antipattern follows a set format, with each of these fields:

Name

The name of the pattern or antipattern.

Also Known As

Other names you may have heard used to describe this.

Motivation

Why people may decide to implement this pattern or antipattern.

Applicability

When this pattern is a good idea, and when it's not.

Consequences

What you should think about if you choose to implement this pattern. What happens if you implement this antipattern.

Implementation

How to implement the pattern.

Related patterns

Other patterns and antipatterns, particularly alternatives.

Not every pattern or antipattern description includes all of these fields.

The following patterns all describe ways of grouping the pieces of a system into one or more stacks. You can view them as a continuum:

- A *monolithic stack* puts an entire system into one stack,
- An *application group stack* groups multiple, related pieces of a system into stacks,
- A *service stack* puts all of the infrastructure for a single application into a single stack,
- A *microstack* breaks the infrastructure for a given application or service into multiple stacks.

Antipattern: Monolithic Stack

A Monolithic Stack is an infrastructure stack that includes too many elements, making it difficult to maintain.

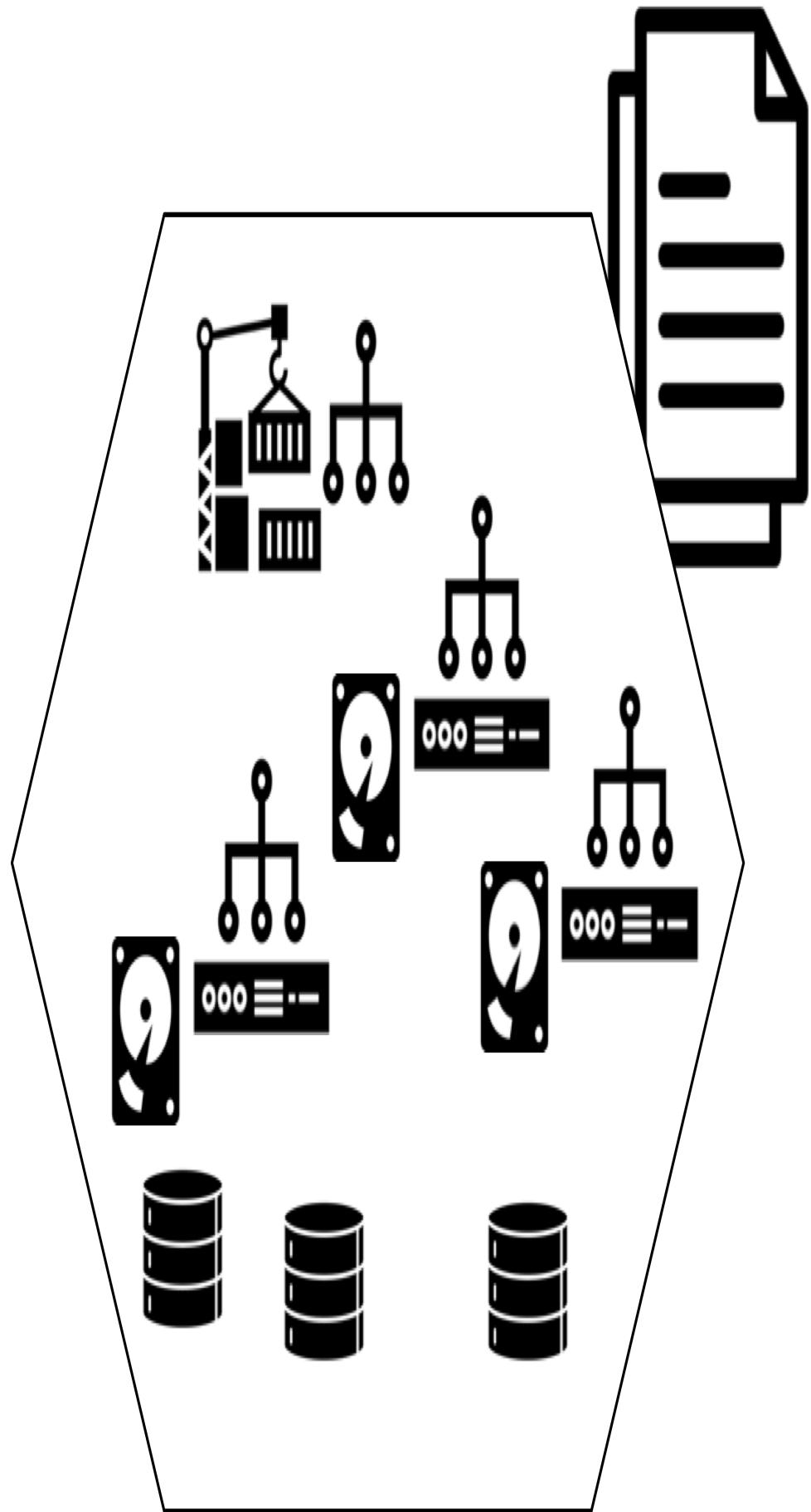


Figure 5-2. A Monolithic Stack is an infrastructure stack that includes too many elements, making it difficult to maintain.

What distinguishes a monolithic stack from other patterns is that the number or relationship of infrastructure elements within the stack is difficult to manage well.

ALSO KNOWN AS

Spaghetti stack, big ball of mud

MOTIVATION

People build monolithic stacks because the simplest way to add a new element to a system is to add it to the existing project. Each new stack adds more moving parts, which may need to be orchestrated, integrated, and tested. A single stack is simpler to manage.

APPLICABILITY

A monolithic stack may be appropriate when your system is small and simple. It's not appropriate when your system grows, taking longer to provision and update.

CONSEQUENCES

Changing a large stack is riskier than changing a smaller stack. More things that can go wrong—it has a larger blast radius. The impact of a failed change may be broader since there are more services and applications within the stack. Larger stacks are also slower to provision and change, which makes them harder to manage.

As a result of the speed and risk of changing a monolithic stack, people tend to make changes less frequently and take longer to do it. This added friction can lead to higher levels of technical debt.

BLAST RADIUS

The term blast radius³ describes the potential damage a given change could make to a system. It's usually based on the elements of the system you're changing, what other elements depend on them, and what elements are shared.

IMPLEMENTATION

You build a monolithic stack by creating an infrastructure stack project and then continuously adding code, rather than splitting it into multiple stacks.

RELATED PATTERNS

The opposite of a monolithic stack is a micro stack ([“Pattern: Micro Stack”](#)), which aims to keep stacks small so that they are easier to maintain and improve. A monolithic stack may be an application group stack [“Pattern: Application Group Stack”](#) that has grown out of control.

IS MY STACK A MONOLITH?

Whether your infrastructure stack is a monolith is a matter of judgment. The symptoms of a monolithic stack include:

- It's difficult to understand how the pieces of the stack fit together (they may be too messy to understand, or perhaps they don't fit well together),
- New people take a while learning the stack's codebase,
- Debugging problems with the stack is hard,
- Changes to the stack frequently cause problems,
- You spend too much time maintaining systems and processes whose purpose is to manage the complexity of the stack.

A key indicator of whether a stack is becoming monolithic is how many people are working on changes to it at any given time. The more common it is for multiple people to work on the stack simultaneously, the more time you spend coordinating changes. Multiple teams making changes to the same stack is even worse. If you frequently have failures and conflicts when deploying changes to a given stack, it may be too large.

Feature branching is a strategy for coping with this, but it can add friction and overhead to delivery. The habitual use of feature branches to work on a stack suggests that the stack has become monolithic.

Continuous Integration (CI) is a more sustainable way to make it safer for multiple people to work on a single stack. However, as a stack grows increasingly monolithic, the CI build takes longer to run, and it becomes harder to maintain good build discipline. If your team's CI is sloppy, it's another sign that your stack is a monolith.

These issues relate to a single team working on an infrastructure stack. Multiple teams working on a shared stack is a clear sign to consider splitting it into more manageable pieces.

Pattern: Application Group Stack

An Application Group Stack includes the infrastructure for multiple related applications or services. The infrastructure for all of these applications is provisioned and modified as a group.

PRODUCT APPLICATION GROUP STACK

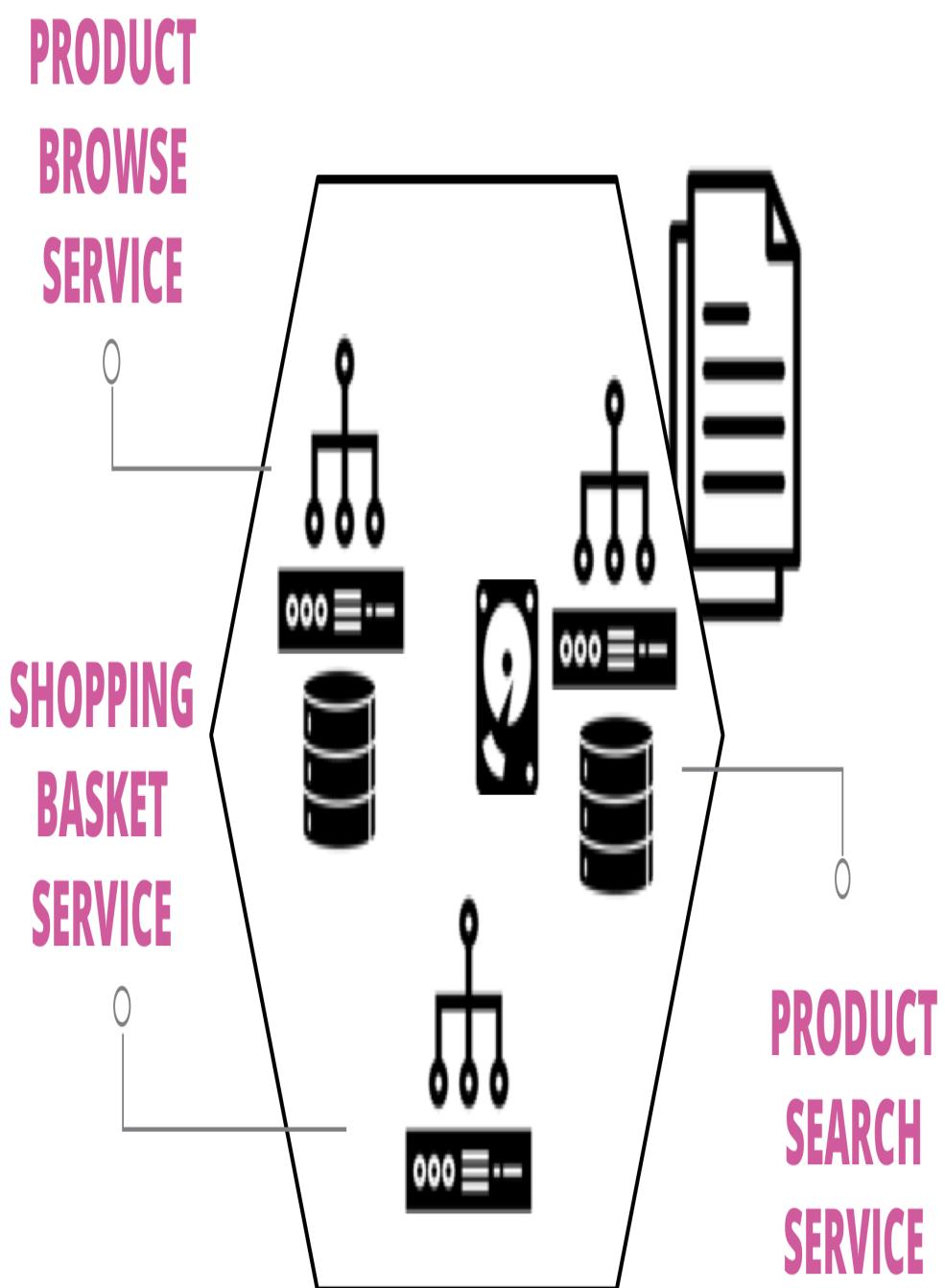


Figure 5-3. An Application Group Stack hosts multiple processes in a single instance of the stack.

For example, an online shopping company's product application group may include separate services for browsing products, searching for products, and managing a shopping basket. An application group stack could provide the infrastructure that runs all three of these services.

ALSO KNOWN AS

Combined stack, service group stack, multi-application stack.

MOTIVATION

Defining the infrastructure for multiple related services together can make it easier to manage the application as a single unit.

APPLICABILITY

This pattern can work well when a single team owns the infrastructure and deployment of all of the pieces of the application. An application group stack can align the boundaries of the stack to the team's responsibilities.

Multi-service stacks are sometimes useful as an incremental step from a monolithic stack to service stacks.

CONSEQUENCES

Grouping the infrastructure for multiple applications together also combines the time, risk, and pace of changes. The team needs to manage the risk to the entire stack for every change, even if only one part is changing. This pattern is inefficient if some parts of the stack change more frequently than others.

The time to provision, change, and test a stack is based on the entire stack. So again, if it's common to change only one part of a stack at a time, having it grouped adds unnecessary overhead and risk.

IMPLEMENTATION

To create an application group stack, you define an infrastructure project that builds all of the infrastructure for a set of services. You can provision and destroy all of the pieces of the application with a single command.

RELATED PATTERNS

This pattern risks growing into a Monolithic Stack (“[Antipattern: Monolithic Stack](#)”). In the other direction, breaking each service in an application group stack into a separate stack creates a service stack (“[Pattern: Service Stack](#)”).

Pattern: Service Stack

A Service Stack manages the infrastructure for each deployable application component in a separate infrastructure stack.

SEPARATE PRODUCT SERVICE STACKS

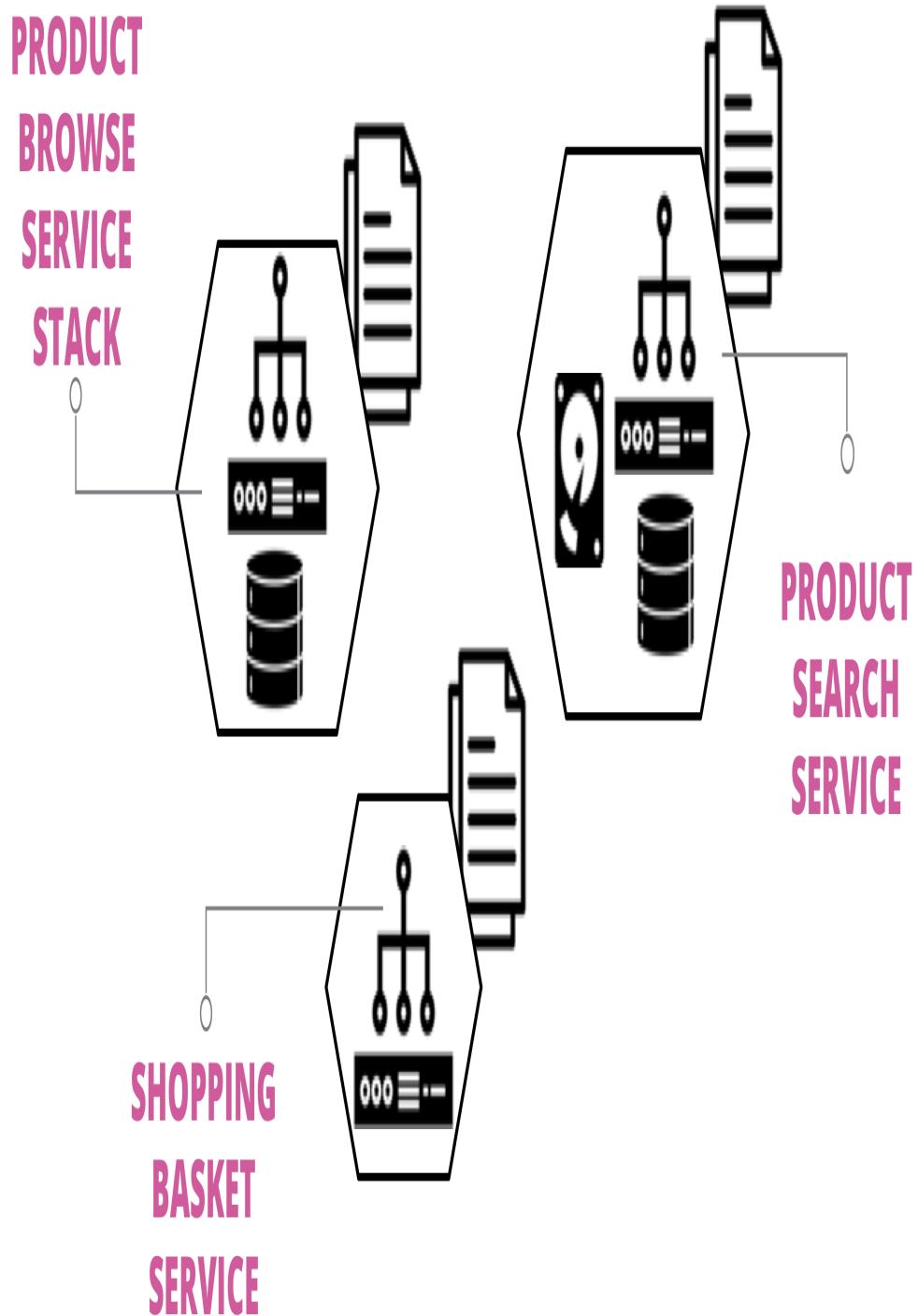


Figure 5-4. A Service Stack manages the infrastructure for each deployable application component in a separate infrastructure stack.

ALSO KNOWN AS

Stack per app, single service stack.

MOTIVATION

Service stacks align the boundaries of infrastructure to the software that runs on it. This alignment limits the blast radius for a change to one service, which simplifies the process for scheduling changes. Service teams can own the infrastructure that relates to their software.

APPLICABILITY

Service stacks can work well with microservice application architectures⁴. They also help organizations with autonomous teams to ensure each team owns its infrastructure⁵.

CONSEQUENCES

If you have multiple applications, each with an infrastructure stack, there could be an unnecessary duplication of code. For example, each stack may include code that specifies how to provision an application server. Duplication can encourage inconsistency, such as using different operating system versions, or different network configurations. You can mitigate this by using modules to share code (as in Chapter 6).

IMPLEMENTATION

Each application or service has a separate infrastructure code project. When creating a new application, a team might copy code

from another application's infrastructure. Or the team could use a reference project, with boilerplate code for creating new stacks.

In some cases, each stack may be complete, not sharing any infrastructure with other application stacks. In other cases, teams may create stacks with infrastructure that supports multiple application stacks. You can learn more about different patterns for this in [Link to Come].

RELATED PATTERNS

The service stack pattern falls between an application group stack (“[Pattern: Application Group Stack](#)”), which has multiple applications in a single stack, and a micro stack (“[Pattern: Micro Stack](#)”), which breaks the infrastructure for a single application across multiple stacks.

Pattern: Micro Stack

The Micro Stack pattern divides the infrastructure for a single service across multiple stacks.

BROWSE SERVICE MICROSTACKS

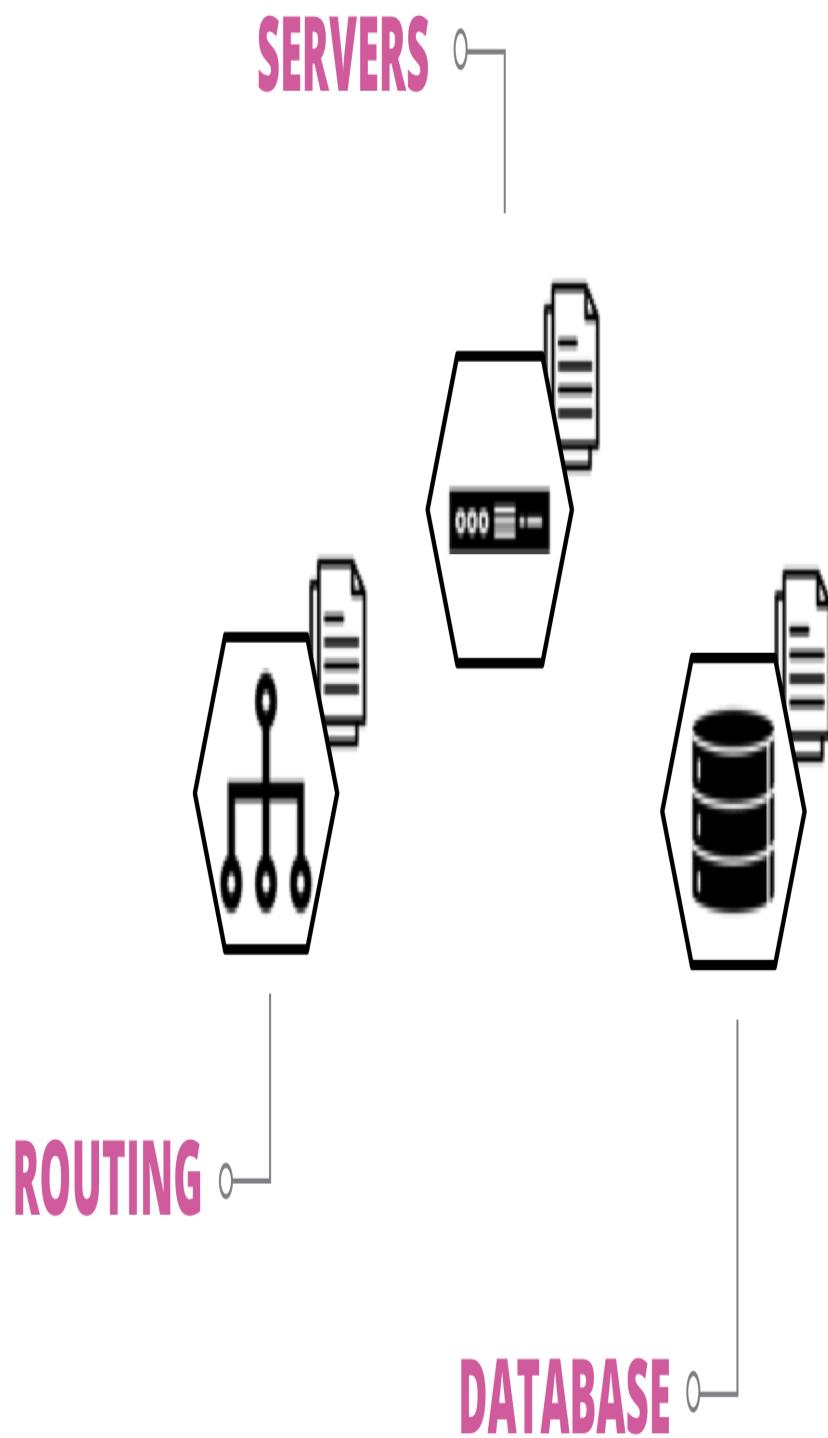


Figure 5-5. Micro stacks divide the infrastructure for a single service across multiple stacks.

For example, you may have a separate stack project each for the networking, servers, and database.

MOTIVATION

Different parts of a service's infrastructure may change at different rates. Or they may have different characteristics which make them easier to manage separately. For instance, some methods for managing server instances involve frequently destroying and rebuilding them⁶. However, some services use persistent data in a database or disk volume. Managing the servers and data in separate stacks means they can have different lifecycles, with the server stack being rebuilt much more often than the data stack.

CONSEQUENCES

Although smaller stacks are themselves simpler, handling the dependencies between them is more complicated.

IMPLEMENTATION

Adding a new microstack involves creating a new stack project. You need to draw boundaries in the right places between stacks to keep them appropriately sized and easy to manage. The related patterns include solutions to this. You may also need to integrate different stacks, which I describe in [Link to Come].

RELATED PATTERNS

Micro stacks are the opposite end of the spectrum from a monolithic stack (“Antipattern: Monolithic Stack”), where a single stack contains all the infrastructure for a system.

Conclusion

Infrastructure stacks are fundamental building blocks for automated infrastructure. The patterns in this chapter are a starting point for thinking about organizing infrastructure into stacks.

Given that a stack is a unit of change, the main principle for deciding where to draw boundaries between stacks in your system is making it easy and safe to make changes. [Link to Come] explores this topic in more detail.

In the next chapter, [Chapter 6](#), I describe patterns and antipatterns for sharing code across stacks using modules.

¹ <https://github.com/chef-boneyard/chef-provisioning>

² The architect Christopher Alexander originated the idea of *design patterns* in *A Pattern Language: Towns, Buildings, Construction* (Alexander, Jacobson, Silverstein, Ishikawa, 1977, Oxford University Press). Kent Beck, Ward Cunningham, and others adapted the concept to *Software design patterns*. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, known as the “Gang of Four” (or “GoF”), cataloged common patterns and antipatterns in *Design Patterns: Elements of Reusable Object-Oriented Software* (1994, Addison Wesley).

³ I don’t know who, if anyone, can be said to have coined the term “blast radius” in the context of software system risks, but [Charity Majors](#) popularized it, probably starting with her post [Terraform, VPC, and why you want a tfstate file per env](#)

⁴ See [Microservices](#) by James Lewis.

⁵ See [The Art of Building Autonomous Teams](#) by John Ferguson Smart, among many other references

⁶ Examples of these are the phoenix server pattern ([Link to Come]) and immutable servers ([Link to Come])

Chapter 6. Using Modules to Share Stack Code

One of the principles of good software design is the DRY principle (“Don’t Repeat Yourself”)¹. As you write code for different infrastructure stacks, you may find yourself writing very similar code multiple times. Rather than maintaining the same code in multiple places, it is often easier to maintain a single copy of the code.

Most stack management tools support modules, or libraries, for this reason. A Stack Code Module is a piece of infrastructure code that you can use across multiple stack projects. You can version, test, and release the module code independently of the stack that uses it.

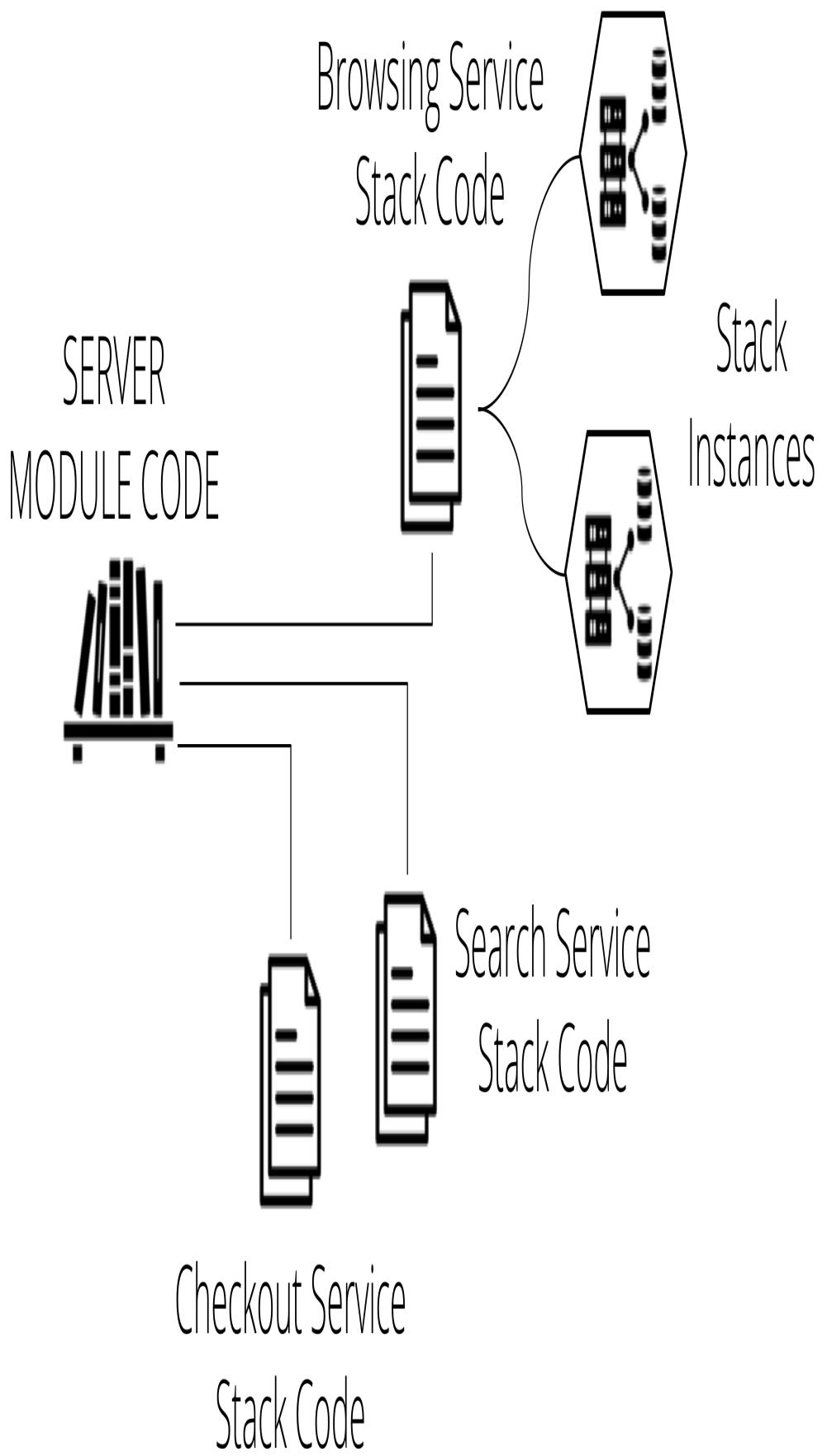


Figure 6-1. A Stack Code Module is a piece of infrastructure code that you can use across multiple stack projects.

Examples of using modules

As an example, the Foodspin team has several servers in their system-container host servers, application servers, and a CD server. The code for each of these is similar:

Example 6-1. Example of stack code with duplication

```
virtual_machine:  
  name: foodspin-clusterhost  
  source_image: hardened-linux-base  
  memory: 16GB  
  provision:  
    tool: servermaker  
    maker_server: maker.foodspin.io  
    role: clusterhost  
  
virtual_machine:  
  name: foodspin-appserver  
  source_image: hardened-linux-base  
  memory: 8GB  
  provision:  
    tool: servermaker  
    maker_server: maker.foodspin.io  
    role: appserver  
  
virtual_machine:  
  name: foodspin-gocd-master  
  source_image: hardened-linux-base  
  memory: 4GB  
  provision:  
    tool: servermaker  
    maker_server: maker.foodspin.io  
    role: appserver
```

The Foodspin team writes a module to declare a server, with placeholders for parameters:

Example 6-2. Example of a stack module for defining a server

```
declare module: foodspin-server
parameters:
  name: STRING
  memory: STRING
  server_role: STRING

virtual_machine:
  name: ${name}
  source_image: hardened-linux-base
  memory: ${memory}
  provision:
    tool: servermaker
    maker_server: maker.foodspin.io
    role: ${server_role}
```

Each stack that needs a server can then use this module:

Example 6-3. Example of a stack module that defines a server

```
use module: foodspin-server
  name: foodspin-clusterhost
  memory: 16GB
  server_role: clusterhost

use module: foodspin-server
  name: foodspin-appserver
  memory: 8GB
  server_role: appserver

use module: foodspin-server
  name: foodspin-gocd-master
  memory: 4GB
  server_role: gocd_master
```

The code for each server is simpler using modules since it only declares the elements that are different rather than duplicating the declarations for the server image and server provisioner. Most usefully, they can make changes to how they provision servers in one location, rather than needing to change code in different codebases.

Patterns and antipatterns for infrastructure modules

Modules can be useful for reusing code. However, they also add complexity. So it's essential to use modules in a way that creates more value than complexity. A useful module simplifies stack code, improving readability and maintainability. A poor module reduces flexibility and is difficult to understand and maintain.

Here are a few patterns and antipatterns that illustrate what works well and what doesn't:

- A Facade Module provides a simplified interface for a resource provided by the stack tool language, or the underlying platform API.
- An Anemic Module wraps the code for an infrastructure element but does not simplify it or add any particular value.
- A Domain Entity Module implements a high-level concept by combining multiple low-level infrastructure resources.
- A Spaghetti Module is configurable to the point where it creates significantly different results depending on the parameters given to it
- An Obfuscation Layer is composed of multiple modules. It is intended to hide or abstract details of the infrastructure from people writing stack code but instead makes the codebase as a whole harder to understand, maintain, and use.
- A One-shot Module is only used once in a codebase, rather than being reused.

Pattern: Facade Module

A Facade Module provides a simplified interface for a resource provided by the stack tool language, or the underlying platform API.

The facade module presents a small number of parameters to code that uses it:

Example 6-4. Example code using a facade module

```
use module: foodspin-server
  name: burgerbarn-appserver
  memory: 8GB
```

The module itself includes a larger amount of code, that the calling code doesn't need to worry about:

Example 6-5. Code for the example facade module

```
declare module: foodspin-server
  virtual_machine:
    name: ${name}
    source_image: hardened-linux-base
    memory: ${memory}
  provision:
    tool: servermaker
    maker_server: maker.foodspin.io
    role: application_server
  network:
    vlan: application_zone_vlan
    gateway: application_zone_gateway
  firewall_inbound:
    port: 22
    from: management_zone
  firewall_inbound:
    port: 443
    from: webserver_zone
```

MOTIVATION

A facade module simplifies and standardizes a common use case for an infrastructure resource. The stack code that uses a facade module should be simpler and easier to read. Improvements to the quality of the module code are rapidly available to all of the stacks which use it.

APPLICABILITY

Facade modules work best for simple use cases, usually involving a low-level infrastructure resource.

CONSEQUENCES

A facade module limits how you can use the underlying infrastructure resource. Doing this can be useful, simplifying options and standardizing around good quality implementations. However, it can also reduce flexibility.

A module is an extra layer of code between the stack code and the code that directly specifies the infrastructure resources. This extra layer adds at least some overhead to maintaining, debugging, and improving code. It can also make it harder to understand the stack code.

IMPLEMENTATION

Implementing a facade module generally involves specifying an infrastructure resource with a number of hard-coded values, and a small number of values that are passed through from the code that uses the module.

RELATED PATTERNS

An anemic module (“[Antipattern: Anemic Module](#)”) is a facade module that doesn’t hide much, so adds complexity without adding much value. A domain entity module (“[Pattern: Domain Entity Module](#)”) is similar to a facade, in that it presents a simplified interface to the code that uses it. But a domain entity combines multiple lower-level elements to present a single higher-level entity to the calling code. In contrast, a facade module is a more direct mapping of a single element.

Antipattern: Anemic Module

An Anemic Module wraps the code for an infrastructure element but does not simplify it or add any particular value. It may be a facade module (“[Pattern: Facade Module](#)”) gone wrong, or it may be part of an obfuscation layer (“[Antipattern: Obfuscation Layer](#)”).

Example 6-6. Example code using an anemic module

```
use module: any_server
  server_name: burgerbarn-appserver
  ram: 8GB
  source_image: base_linux_image
  provisioning_tool: servermaker
  server_role: application_server
  vlan: application_zone_vlan
  gateway: application_zone_gateway
  firewall_rules:
    firewall_inbound:
      port: 22
      from: management_zone
    firewall_inbound:
      port: 443
      from: webserver_zone
```

The module itself passes the parameters directly to the stack management tool’s code:

Example 6-7. Code for the example anemic module

```
declare module: any_server
virtual_machine:
  name: ${server_name}
  source_image: ${origin_server_image}
  memory: ${ram}
  provision:
    tool: ${provisioning_tool}
    role: ${server_role}
  network:
    vlan: ${server_vlan}
    gateway: ${server_gateway}
    firewall_inbound: ${firewall_rules}
```

ALSO KNOWN AS

Value-Free Wrapper, Pass-Through Module, Obfuscator Module.

MOTIVATION

Sometimes people write this kind of module aiming to follow the DRY (Don't Repeat Yourself) principle. They see that code that defines a common infrastructure element, such as a virtual server, load balancer, or security group, is used in multiple places in the codebase. So they create a module that declares that element type once and use that everywhere. But because the elements are being used differently in different parts of the code, they need to expose a large number of parameters in their module. The result is that the code that uses the module is no simpler than directly using the stack tool's code. The codebase still has repeated code, only now it's repeated use of the module.

APPLICABILITY

Nobody intentionally writes an anemic module. You may debate whether a given module is anemic or is a facade, but the debate itself is useful. You should consider whether a module adds real

value and, if not, then refactor it into code that uses the stack language directly.

CONSEQUENCES

Writing, using, and maintaining module code rather than directly using the constructs provided by your stack tool adds overhead. It makes your stack code more difficult to understand, especially for people who know the stack tool but are new to your codebase. It adds more code to maintain, usually with separate versioning and release. You can end up with different versions of a module being used by different stacks, causing people to waste time and energy managing releases, upgrades, and testing.

IMPLEMENTATION

If a module does more than pass parameters, but still presents too many parameters to code that uses it, you might consider splitting it into multiple modules. Doing this makes sense when there are a few common but different use cases for the infrastructure element the module defines. For example, rather than a single module for defining firewall rules, you may want one module to define a firewall rule for public HTTPS traffic, another for internal HTTPS traffic, and a third for SSH traffic. Each of these may need fewer parameters than a single module that handles multiple protocols and scenarios.

RELATED PATTERNS

Some anemic modules are “[Pattern: Facade Module](#)” that grew out of control. Others are part of an “[Antipattern: Obfuscation Layer](#)”. It may also be a “[Pattern: Domain Entity Module](#)” that maps a bit

too directly to a single underlying infrastructure element, rather than to a useful combination of elements.

COUPLING AND COHESION WITH INFRASTRUCTURE MODULES

When organizing code of any kind into different components, you need to consider dependencies. A stack project that uses a module depends on that module². The level of *coupling* describes how much a change to one part of the codebase affects other parts. If a stack is *tightly* coupled to a module, then changes to the module will probably require changing the stack code as well. This is a problem when multiple stacks are tightly coupled to a module, or when there is tight coupling across multiple modules and stacks. Tight coupling creates friction and risk for making changes to code.

You should aim to make your code *loosely* coupled. You can draw on lessons from software architecture to find ways to identify and avoid tight coupling³.

The level of *cohesion* of a component describes how well-focused it is. A module that does too much, like a spaghetti module (“[Antipattern: Spaghetti Module](#)”), has low cohesion. A module has high cohesion when it has a clear focus. A facade module (“[Pattern: Facade Module](#)”) can be highly cohesive around a low-level infrastructure resource, while a domain entity module (“[Pattern: Domain Entity Module](#)”) can be highly cohesive around a clear, logical entity.

Pattern: Domain Entity Module

A Domain Entity Module implements a high-level concept by combining multiple low-level infrastructure resources. An example of a higher level concept is the infrastructure needed to run a specific Java application. This example shows how a module that implements a Java application infrastructure instance might be used from stack project code:

Example 6-8. Example of using a domain entity module

```
use module: java-application-infrastructure
name: "shopping_app_cluster"
application: "shopping_app"
application_version: "4.20"
network_access: "public"
```

The module creates a complete set of infrastructure elements, including a cluster of virtual servers with load balancer, database instance, and firewall rules.

MOTIVATION

Infrastructure stack languages provide language constructs that map directly to resources and services provided by infrastructure platforms (the things I described in [Chapter 3](#)). Teams combine these low-level constructs into higher-level constructs to serve a particular purpose, such as hosting a Java application or running a data pipeline.

It can take a fair bit of low-level infrastructure code to define all of the pieces needed to meet that high-level purpose. As I described above, to create the infrastructure to run a Java application, you may need to write code for a cluster of virtual servers, for a load balancer, for a database instance, and for firewall rules. That's a lot of code. Although you may write modules so you can share code for each low-level resource, if you need multiple different applications with similar infrastructure, it would be useful to share code at this higher level as well.

EXAMPLE: FOODSPIN'S APPLICATION INFRASTRUCTURE ENTITY MODULE

The Foodspin team runs several different clusters for running Java applications. They host a separate application instance for each of their customers. They also run an instance of the Atlassian Jira bug tracker, and the GoCD pipeline server. So they define one stack project for each customer⁴, one for Jira, and one for GoCD.

The team notices that they're spending too much time copying changes between these stacks, so they decide to create a module for the common infrastructure they need. In their case, each instance has a cluster of virtual machines, all built from the same server image. Different instances have different minimum and maximum numbers of servers in the cluster, so the team makes these into parameters for the module. Some applications use a database, and some don't. So they leave this out of their module and instead have an optional parameter to pass connection details to the server provisioning code.

Their module also defines the networking structures, including firewall rules and routing, that the application needs. Here, they use a parameter to indicate whether the application is public-facing (for the customer applications) or internal (for Jira and GoCD), which results in some different networking configurations.

By using this module in each stack that needs an application server, the Foodspin team has simplified each stack's project code, so it is easier to understand. They also have a single place to make improvements to the infrastructure for multiple applications. They write a set of tests for this module (as we'll see in Chapter 9 and later chapters), which gives them the confidence to improve their code continuously.

APPLICABILITY

A domain entity module is useful for things that are fairly complex to implement, and that are used in pretty much the same way in multiple places in your system. Don't create a domain entity module that you only use once. And don't create one of these modules for multiple instances of the same thing, when each instance is significantly different. The first case is a one-shot module (“Antipattern: One-shot Module”), the second risks becoming a spaghetti module (“Antipattern: Spaghetti Module”).

For example, you may run multiple application servers that use the same stack-operating system, application language, and application deployment method. This is a candidate for a domain entity module.

As a counter-example, you may have multiple application servers, but some run Windows, some run Linux. One runs a stateless PHP application, another runs a Java application on Tomcat with a MySQL database, while a third hosts a .Net application that uses message queues to integrate with other applications. Although all three of these are application servers, their implementation is quite different. Any module that can create a suitable infrastructure stack for all of these is unlikely to have a clean design and implementation.

IMPLEMENTATION

On a concrete level, implementing a domain entity model is a matter of writing the code for a related grouping of infrastructure in a single module. But the best way to create a high-quality codebase that is easy for people to learn and maintain is to take a design-led approach.

I recommend drawing from lessons learned in software architecture and design. The domain entity module pattern derives from *Domain Driven Design* (DDD)⁵, which creates a conceptual model for the business domain of a software system, and uses that to drive the design of the system itself. Infrastructure, especially one designed and built as software, can be seen as a domain in its own right. The domain is building, delivering, and running software.

A particularly powerful approach is for an organization to use DDD to design the architecture for the business software, and then extend the domain to include the systems and services used for building and running that software.

The code to implement a Java application server might look like Example 6-9. This example creates and assembles three different resources: a DNS entry, which points to a load balancer, which routes traffic to a server cluster.

Example 6-9. Example domain entity module

```
declare module: java-application-infrastructure

dns_entry:
  id: "${APP_NAME}-hostname"
  record_type: "A"
  hostname: "${APP_NAME}.foodspin.io"
  ip_address: ${load_balancer.ip_address}

server_cluster:
  id: "${APP_NAME}-cluster"
  min_size: 1
  max_size: 3
  each_server_node:
    source_image: base_linux
    memory: 4GB
    provision:
      tool: servermaker
      role: appserver
    parameters:
      app_package: "${APP_NAME}-${APP_VERSION}.war"
      app_repository: "repository.foodspin.io"

load_balancer:
  protocol: https
  target:
    type: server_cluster
    target_id: "${APP_NAME}-cluster"
```

Code to provision a stack that runs a search application using this module might look like this:

```
use module: java-application-infrastructure
app_name: foodspin_search_app
app_version: 1.23
```

RELATED PATTERNS

If you find that some stack projects have very little code other than importing a single module, you might consider putting the code directly into the stack project. Reusing code in the form of an entire stack, rather than a module, is the reusable stack pattern (“Pattern: Reusable Stack”). In some cases, having the entire stack code in a module is deliberate, as in the wrapper stack pattern (“Pattern: Wrapper Stack”).

A facade module (“Pattern: Facade Module”) is like a domain entity module that only creates a single low-level infrastructure element. A spaghetti module (“Antipattern: Spaghetti Module”) is similar to a domain entity module but has too many different options.

Antipattern: Spaghetti Module

A Spaghetti Module is configurable to the point where it creates significantly different results depending on the parameters given to it. The implementation of the module is messy and difficult to understand, because it has too many moving parts:

Example 6-10. Example of a spaghetti module

```
declare module: application-server-infrastructure
variable: network_segment = {
    if ${parameter.network_access} = "public"
        id: public_subnet
    else if ${parameter.network_access} = "customer"
        id: customer_subnet
    else
        id: internal_subnet
    end
}

switch ${parameter.application_type}:
    "java":
```

```

virtual_machine:
  origin_image: base_tomcat
  network_segment: ${variable.network_segment}
  server_configuration:
    if ${parameter.database} != "none"
      database_connection:
        ${database_instance.my_database.connection_string}
    end
    ...
  ".NET":
    virtual_machine:
      origin_image: windows_server
      network_segment: ${variable.network_segment}
      server_configuration:
        if ${parameter.database} != "none"
          database_connection:
            ${database_instance.my_database.connection_string}
        end
        ...
  "php":
    container_group:
      cluster_id: ${parameter.container_cluster}
      container_image: nginx_php_image
      network_segment: ${variable.network_segment}
      server_configuration:
        if ${parameter.database} != "none"
          database_connection:
            ${database_instance.my_database.connection_string}
        end
        ...
    end

  switch ${parameter.database}:
    "mysql":
      database_instance: my_database
      type: mysql
      ...
    ...

```

The example code above assigns the server it creates to one of three different network segments, and optionally creates a database cluster and passes a connection string to the server configuration. In some cases, it creates a group of container instances rather than a virtual server. This module is a bit of a beast.

ALSO KNOWN AS

Swiss Army module.

MOTIVATION

As with other antipatterns, people create a spaghetti module by accident, often over time. You may create a facade module (“Pattern: Facade Module”) for a common infrastructure element, such as a server, that grows so that it can create radically different types of servers. Or you may create a domain entity module (“Pattern: Domain Entity Module”) for an application’s infrastructure. But then you find that the module needs to optionally include a variety of elements, such as databases, message queues, load balancers, and public Internet gateways, depending on the application that runs on it.

CONSEQUENCES

A module that does too many things is less maintainable than one with a tighter scope. The more things a module does, and the more variations there are in the infrastructure that it can create, the harder it is to change it without breaking something. These modules are harder to test. As I explain in a later chapter (Chapter 9), better-designed code is easier to test, so if you’re struggling to write automated tests and build pipelines to test the module in isolation, It’s a sign that you may have a spaghetti module.

IMPLEMENTATION

A spaghetti module’s code often contains conditionals, that apply different specifications in different situations. For example, a

database cluster module might take a parameter to choose which database to provision.

When you realize you have a spaghetti module on your hands, you should refactor it. Often, you can split it into different modules, each with a more focused remit. For example, you might decompose your single application infrastructure module into different modules for different parts of the application's infrastructure. An example of a stack that uses decomposed modules in this way, rather than using the spaghetti module from the earlier example ([Example 6-10](#)) might look like this:

Example 6-11. Example of using decomposed modules rather than a single spaghetti module

```
use module: java-application-servers
  name: burgerbarn_appserver
  application: "shopping_app"
  application_version: "4.20"
  network_segment: customer_subnet
  server_configuration:
    database_connection: ${module.mysql-
database.outputs.connection_string}

use module: mysql-database
  cluster_minimum: 1
  cluster_maximum: 3
  allow_connections_from: customer_subnet
```

Each of the modules is smaller, simpler, and so easier to maintain and tested than the original spaghetti module.

RELATED PATTERNS

A spaghetti module is often a domain entity module ([“Pattern: Domain Entity Module”](#)) gone wrong.

Antipattern: Obfuscation Layer

Unlike the other patterns and antipatterns in this chapter, an Obfuscation Layer is composed of multiple modules. Intended to hide or abstract details of the infrastructure from people writing stack code, it instead makes the codebase as a whole harder to understand, maintain, and use.

ALSO KNOWN AS

Abstraction layer, portability layer, in-house infrastructure model.

MOTIVATION

An obfuscation layer is usually intended to simplify or standardize implementation. A team might create a library of modules as an in-house model for building infrastructure. Sometimes the intention is for people to write stack code without needing to learn the stack tool itself. In other cases, the layer tries to abstract the specifics of the infrastructure platform so that code can be written once and used across multiple platforms.

CONSEQUENCES

If your team uses a heavily customized model for infrastructure code, then it becomes a barrier to new people joining your team. Even someone who knows the stack tools you use has a steep learning curve to understand your special language. An in-house obfuscation layer tends to be inflexible, forcing people to either work around its limitations or spend time making changes to the obfuscation layer itself. Either way, people waste time creating and maintaining extra code.

People, especially managers and hands-off architects⁶, often overestimate the benefit of hiding an underlying platform or tools

from people writing infrastructure code. I have yet to see a cloud abstraction layer that adds noticeable value, or that avoids adding cost and complexity. Most fail at both.

CLOUD ABSTRACTION LAYERS CONSIDERED HARMFUL

Different infrastructure platforms provide common types of resources, as you see when reading or skimming through [Chapter 3](#). So it seems simple to map these to a common set of abstractions. But the resources are implemented differently. The actual code to implement a virtual server on AWS, Azure, and GCE, for example, is not trivial.

And each platform provides hundreds of different services. Even if you limit yourself to using a few dozen services, that's a large amount of code to implement and maintain.

The theory of an abstraction layer is that you can reduce the cost of writing and maintaining a stack across multiple platforms, as visualized here:

One stack code project for all
three cloud platforms

Separate stack code project for
each cloud platform

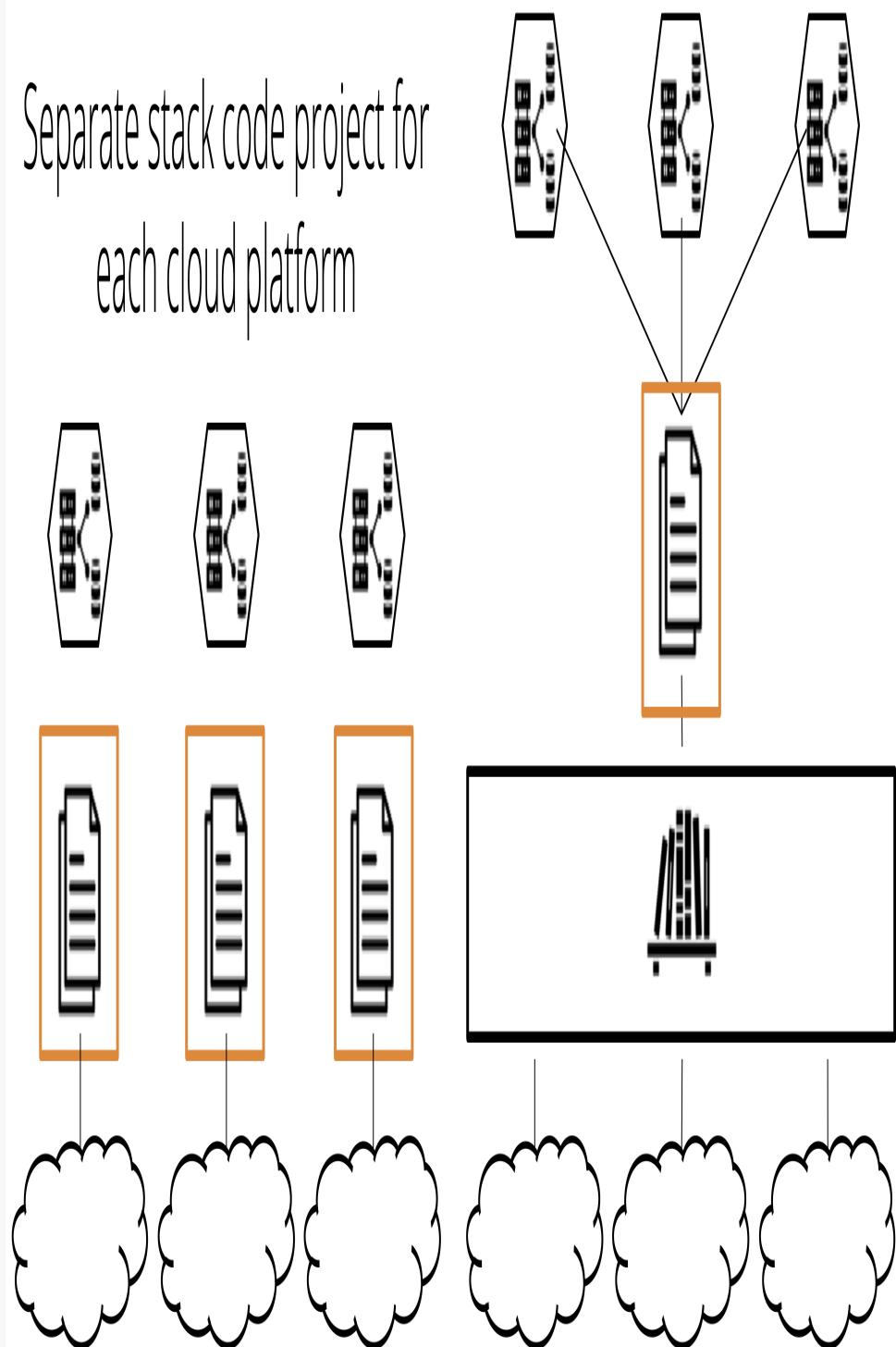


Figure 6-2. The goal of an abstraction layer is to reduce the cost of a stack across multiple platforms

In practice, the cost of writing and maintaining an abstraction layer for multiple platforms is more than the cost of the separate code for each platform, as shown here:

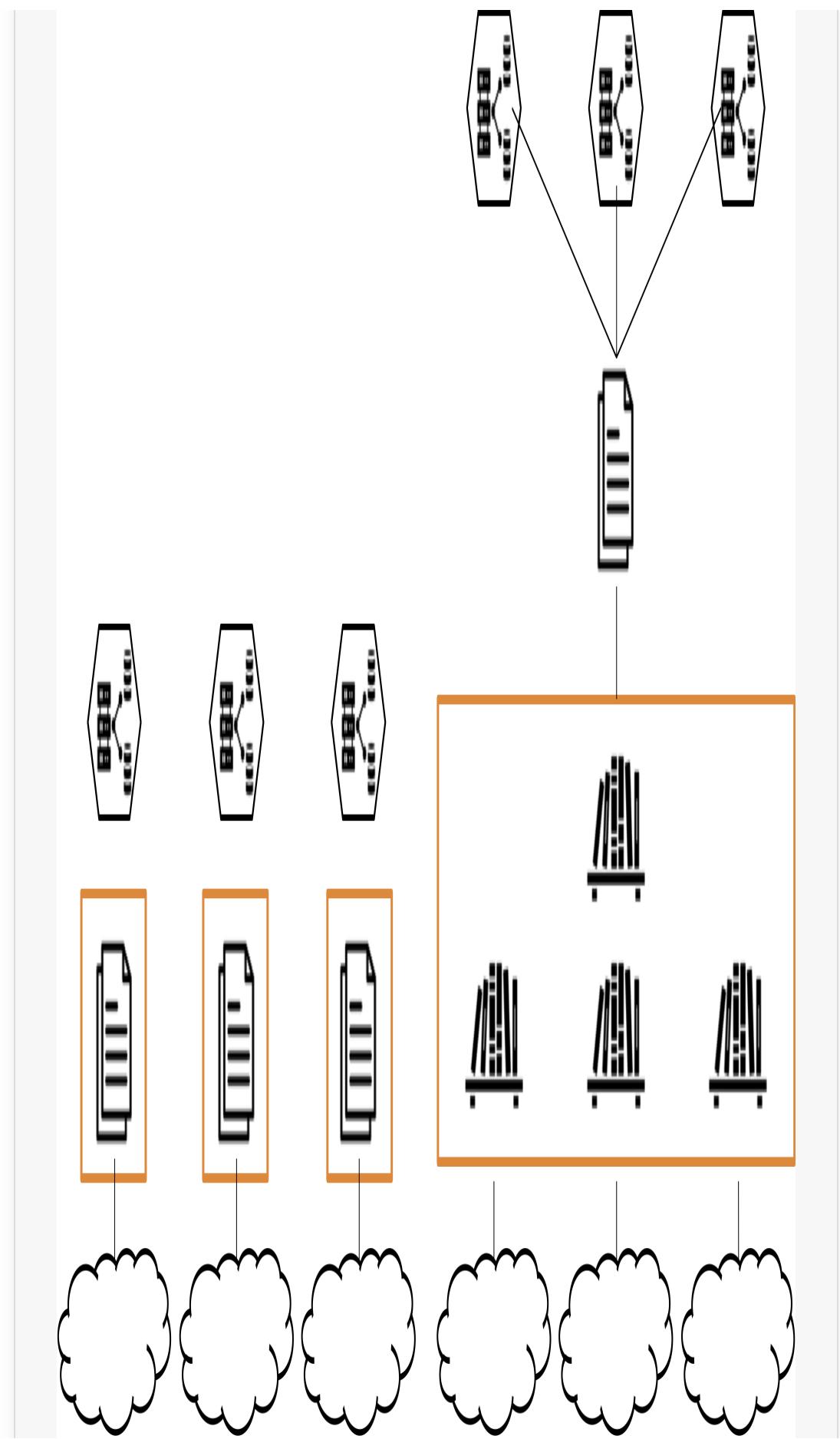


Figure 6-3. The cost of an abstraction layer

The flipside of the high cost of an abstraction layer is reduced value. It's impractical to cover everything the underlying platforms offer. And if you want to ensure you can create a stack on all of the platforms, then you won't be able to leverage any features that are unique to one platform or another. The result is using a least-common-denominator subset of cloud features, at a higher cost than using them directly.

RELATED PATTERNS

Domain entity modules (“[Pattern: Domain Entity Module](#)”) may be used to create an obfuscation layer at a higher level, while facade modules (“[Pattern: Facade Module](#)”) would be used to obfuscate lower-level resources. If each module in the layer handles multiple platforms, they are probably spaghetti modules (“[Antipattern: Spaghetti Module](#)”).

Antipattern: One-shot Module

A one-shot module is only used once in a codebase, rather than being reused.

MOTIVATION

People usually create one-shot modules as a way to organize the code in a project.

APPLICABILITY

If a stack project includes enough code that it becomes difficult to navigate, you have a few options. Splitting the stack into modules is one approach. If the stack is conceptually doing too much, it might be better to divide it into multiple stacks, using an appropriate stack structural pattern (see “[Patterns and antipatterns for structuring stacks](#)”). Otherwise, merely organizing code into different files and, if necessary, different folders, can make it

easier to navigate and understand the codebase without the overhead of the other options.

CONSEQUENCES

Organizing the code into modules adds the overhead of declaring the module and passing parameters. You add even more complexity if you manage the module code separately, with separate versioning. This overhead is worthwhile when you share code across multiple stack projects. The benefits of code reuse make up for the added time and energy of maintaining a separate module. Paying that cost when you're not using the benefit is a waste.

RELATED PATTERNS

A one-shot module may map closely to lower-level infrastructure elements, like a facade module (“[Pattern: Facade Module](#)”), or to a higher-level entity, like a domain entity module (“[Pattern: Domain Entity Module](#)”).

Conclusion

This chapter has explored the use of modules to share code across stacks. Modules are a popular mechanism to manage the growth of a codebase, but as you can see, it's easy to create an unmaintainable mess.

The next chapter ([Chapter-Building-Environments](#)) is devoted to one of the most common situations that require multiple instances of the same or similar infrastructure. This need leads to some

patterns and antipatterns for implementing shared code at the level of the stack.

- **1** The DRY principle, and others, can be found in *The Pragmatic Programmer: From Journeyman to Master* by Andrew Hunt and David Thomas
- 2** In later chapters ([Link to Come]) we'll see that a stack can depend on another stack, as well.
- 3** Two resources to get started include Martin Fowler's paper [Reducing Coupling](#), and Mohamed Sanaulla's post [Cohesion and Coupling: Two OO Design Principles](#).
- 4** Later on ("Example of how Foodspin moves to reusable stacks") we'll see how the Foodspin team evolves this to use a single reusable stack project
- 5** See *Domain-Driven Design: Tackling Complexity in the Heart of Software*, by Eric Evans, 2003 Addison Wesley
- 6** For a thoughtful view on how architecture, and architects, relates to implementation, I recommend the [Architect Elevator](#), by Gregor Hohpe.

Chapter 7. Building Environments With Stacks

In Chapter 5, I described an infrastructure stack as a collection of infrastructure resources that you manage as a single unit. An environment is also a collection of infrastructure resources. So is a stack the same thing as an environment? In this chapter, I explain that maybe it is, but maybe it isn't.

The difference between an environment and a stack is that an environment is conceptual, while a stack is concrete. You define an environment to fulfill a particular purpose, like deploying and testing software. You define a stack with code, and then provision it, change it, and destroy it using a tool. As I'll explain in this chapter, you might implement an environment as a single stack, or you might compose an environment from multiple stacks. You could even create several environments in one stack, although you shouldn't.

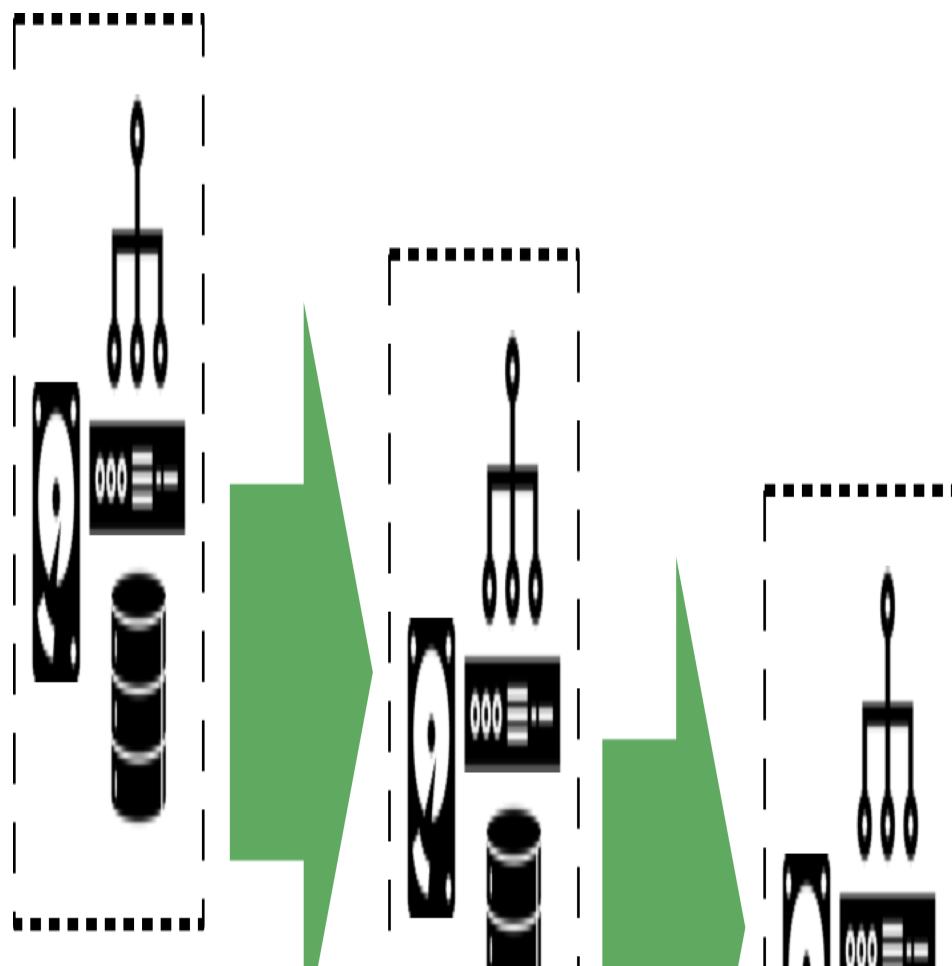
What environments are all about

The concept of an environment is one of those things that we take for granted in IT. But we often mean slightly different things when we use the term in different contexts. In this book, an environment is a collection of operationally related infrastructure resources. That is, the resources in an environment support a particular activity, such as testing or running a system. Most often, multiple environments exist, each running an instance of the same system.

There are two typical use cases for having multiple environments running instances of the same system. One is to support a release delivery process, and the other is to run multiple production instances of the system.

Release delivery environments

The most familiar use case for multiple environments is to support a progressive software release process - sometimes called the *path to production*. A given build of an application is deployed to each environment in turn to support different development and testing activities until it is finally deployed to the production environment.



TEST

STAGING

PRODUCTION

Figure 7-1. Foodspin delivery environments

I'll use this set of environments throughout this chapter to illustrate patterns for defining environments as code.

Multiple production environments

You might also use multiple environments for complete, independent copies of a system in production. Reasons to do this include:

Fault tolerance

If one environment fails, others can continue to provide service. Doing this could involve a failover process to shift load from the failed environment. You can also have fault tolerance within an environment, by having multiple instances of some infrastructure, as with a server cluster. Running an additional environment duplicates all of the infrastructure, creating a higher degree of fault tolerance, although at a higher cost. See [Link to Come] for more on this.

Scalability

You can spread a workload across multiple environments. People often do this geographically, with a separate environment for each region. Multiple environments may be used to achieve both scalability and fault tolerance. If there is a failure in one region, the load is shifted to another region's environment until the failure is corrected.

Segregation

You may run multiple instances of an application or service for different user bases, such as different clients. Running these instances in different environments can strengthen segregation. Stronger segregation may help meet legal or compliance requirements and give greater confidence to customers.

FOODSPIN EXAMPLE: MULTIPLE PRODUCTION ENVIRONMENTS

In the “Foodspin Example: Compute resources” example in Chapter 3 I explained that Foodspin runs a separate application server for each of its restaurant customers. As they expand to support customers in North America, Europe, and South Asia they decide to create a separate environment for each of these regions.

NORTH AMERICA

EUROPE

ASIA-PACIFIC

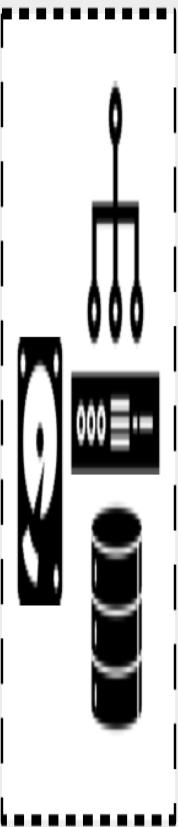
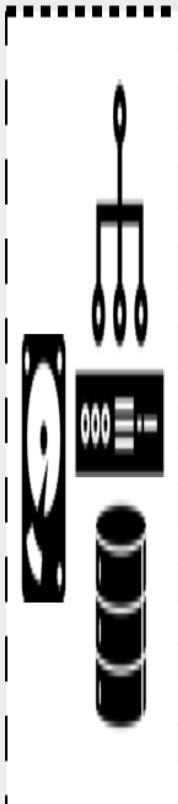
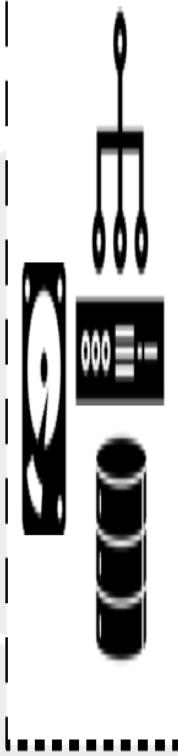


Figure 7-2. Foodspin regional environments

Using fully separated environments, rather than having a single environment spread across the regions, helps them to ensure that they are complying with different regulations about storing customer data in different regions. Also, if they need to make changes that involve downtime, they can do so in each region at a different time. This makes it easier to align downtime to different timezones.

Later, Foodspin lands a contract with a large fast-food chain, Hungry Hungry Horses. Hungry Hungry Horses is worried that having their customer data hosted on the same infrastructure with data from their competitors runs the risk of leaking data. So Foodspin's team offers to run a completely separate environment dedicated to Hungry Hungry Horses, at a higher cost than running in a shared environment.

Environments, consistency, and configuration

Since multiple environments are meant to run instances of the same system, the infrastructure in each environment should be consistent. Consistency across environments is one of the main drivers of Infrastructure as Code.

Differences between environments create the risk of inconsistent behavior. Testing software in one environment might not uncover problems that occur in another. It's even possible that software deploys successfully in some environments but not others.

On the other hand, you typically need some specific differences between environments. Test environments may be smaller than production environments. Different people may have different privileges in different environments. Environments for different customers may have different features and characteristics. At the very least, names and IDs may be different (*appserver-test*, *appserver-stage*, *appserver-prod*). So you need to configure at least some aspects of your environments.

A key consideration for environments is your testing and validation strategy. When the same infrastructure code is applied to every environment, testing it in one environment tends to give confidence that it will work correctly in other environments. You don't get this confidence if the infrastructure varies much across instances, however.

You may be able to improve confidence by testing infrastructure code using different configuration values. However, it may not be practical to test many different values. In these situations, you may need additional validation, such as post-provisioning tests or monitoring production environments. I'll go more in-depth on testing and validation in [Chapter 9](#).

Patterns for building environments

As I mentioned earlier, an environment is a conceptual collection of infrastructure elements, and a stack is a concrete collection of infrastructure elements. A stack project is the source code you use to create one or more stack instances. So how should you use stack projects and instances to implement environments?

I'll describe two antipatterns and one pattern for implementing environments using infrastructure stacks. Each of these patterns describes a way to define multiple environments using infrastructure stacks. Some systems are composed of multiple stacks, as I described in "[Patterns and antipatterns for structuring stacks](#)". I'll explain what this looks like for multiple environments in "[Building environments with multiple stacks](#)".

Antipattern: Multiple-Environment Stack

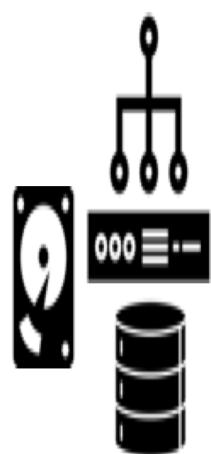
A Multiple-Environment Stack defines and manages the infrastructure for multiple environments as a single stack instance.

For example, if there are three environments for testing and running an application, a single stack project includes the code for all three of the environments.



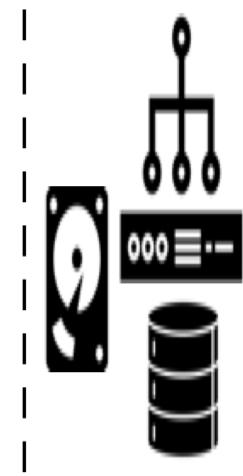
ONE CODE
PROJECT

ONE STACK
INSTANCE



TEST

STAGING



PRODUCTION

ALL THE
ENVIRONMENTS

Figure 7-3. A multiple-environment stack manages the infrastructure for multiple environments as a single stack instance.

ALSO KNOWN AS

- One stack with many environments
- All your environments in one basket

MOTIVATIONS

Many people create this type of structure when they're learning a new stack tool because it seems natural to add new environments into an existing project.

CONSEQUENCES

When running a tool to update a stack instance, the scope of a potential change is everything in the stack. If you have a mistake or conflict in your code, everything in the instance is vulnerable¹.

When your production environment is in the same stack instance as another environment, changing the other environment risks causing a production issue. A coding error, unexpected dependency, or even a bug in your tool can break production when you only meant to change a test environment.

RELATED PATTERNS

You can limit the blast radius of changes by dividing environments into separate stacks. One obvious way to do this is the copy-paste environment (“Antipattern: Copy-Paste Environments”), where each environment is a separate stack project, although this is considered an antipattern.

A better approach is the reusable stack pattern ([“Pattern: Reusable Stack”](#)). A single project is used to define the generic structure for an environment and is then used to manage a separate stack instance for each environment. Although this involves using a single project, the project is only applied to one environment instance at a time. So the blast radius for changes is limited to that one environment.

Antipattern: Copy-Paste Environments

The Copy-Paste Environments antipattern uses a separate stack source code project for each infrastructure stack instance.

In our example of three environments named *test*, *staging*, and *production*, there is a separate infrastructure project for each of these environments. Changes are made by editing the code in one environment and then copying the changes into each of the other environments in turn.

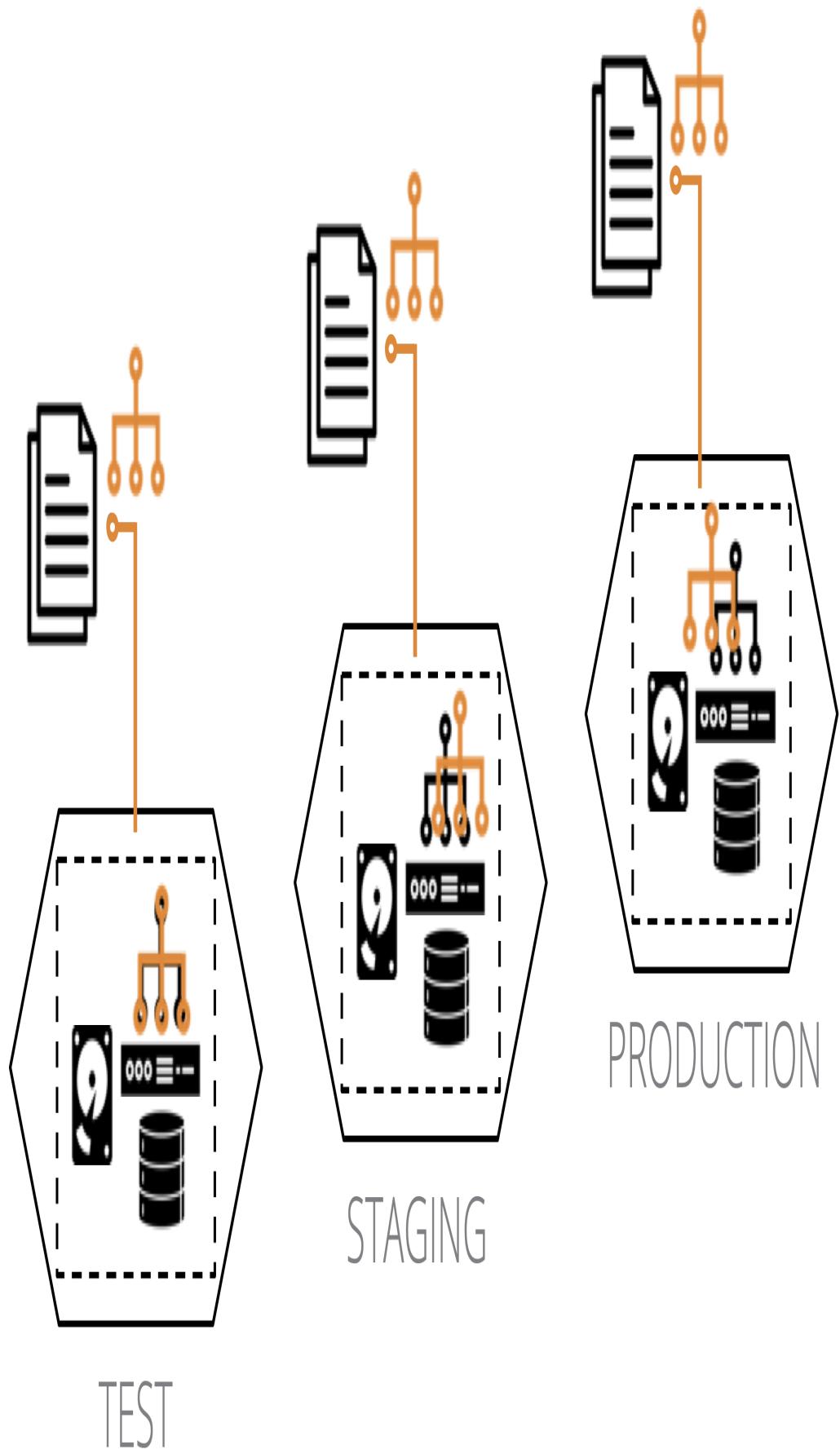


Figure 7-4. A copy-paste environment has a separate copy of the source code project for each instance.

ALSO KNOWN AS

- Singleton stack
- Snowflake stack
- Copy-pasta

MOTIVATION

Copy-paste environments are an intuitive way to maintain multiple environments. They avoid the blast radius problem of the multi-headed stack antipattern. You can also easily customize each stack instance.

APPLICABILITY

Copy-paste environments may be appropriate when you want to maintain and change different instances and aren't worried about code duplication or consistency.

CONSEQUENCES

It can be challenging to maintain multiple copy-paste environments. When you want to make a code change, you need to copy it to every project. You probably need to test each instance separately, as a change may work in one but not another.

Copy-paste environments often suffer from configuration drift (“Configuration Drift”). Using copy-paste environments for delivery environments reduces the reliability of the deployment process and the validity of testing, due to inconsistencies from one environment to the next.

IMPLEMENTATION

You create a copy-paste environment by copying the project code from one stack instance into a new project. You then edit the code to customize it for the new instance. When you make a change to one stack, you need to copy and paste it across all of the other stack projects, while keeping the customizations in each one.

RELATED PATTERNS

In cases where stack instances are meant to represent the same stack, the reusable stack pattern (“[Pattern: Reusable Stack](#)”) is usually more appropriate. The wrapper stack pattern (“[Pattern: Wrapper Stack](#)”) avoids the disadvantages of a copy-paste stack by having all of the logic in a module and only using each stack project for configuration.

Pattern: Reusable Stack

A Reusable Stack is an infrastructure source code project that is used to create multiple instances of a stack.

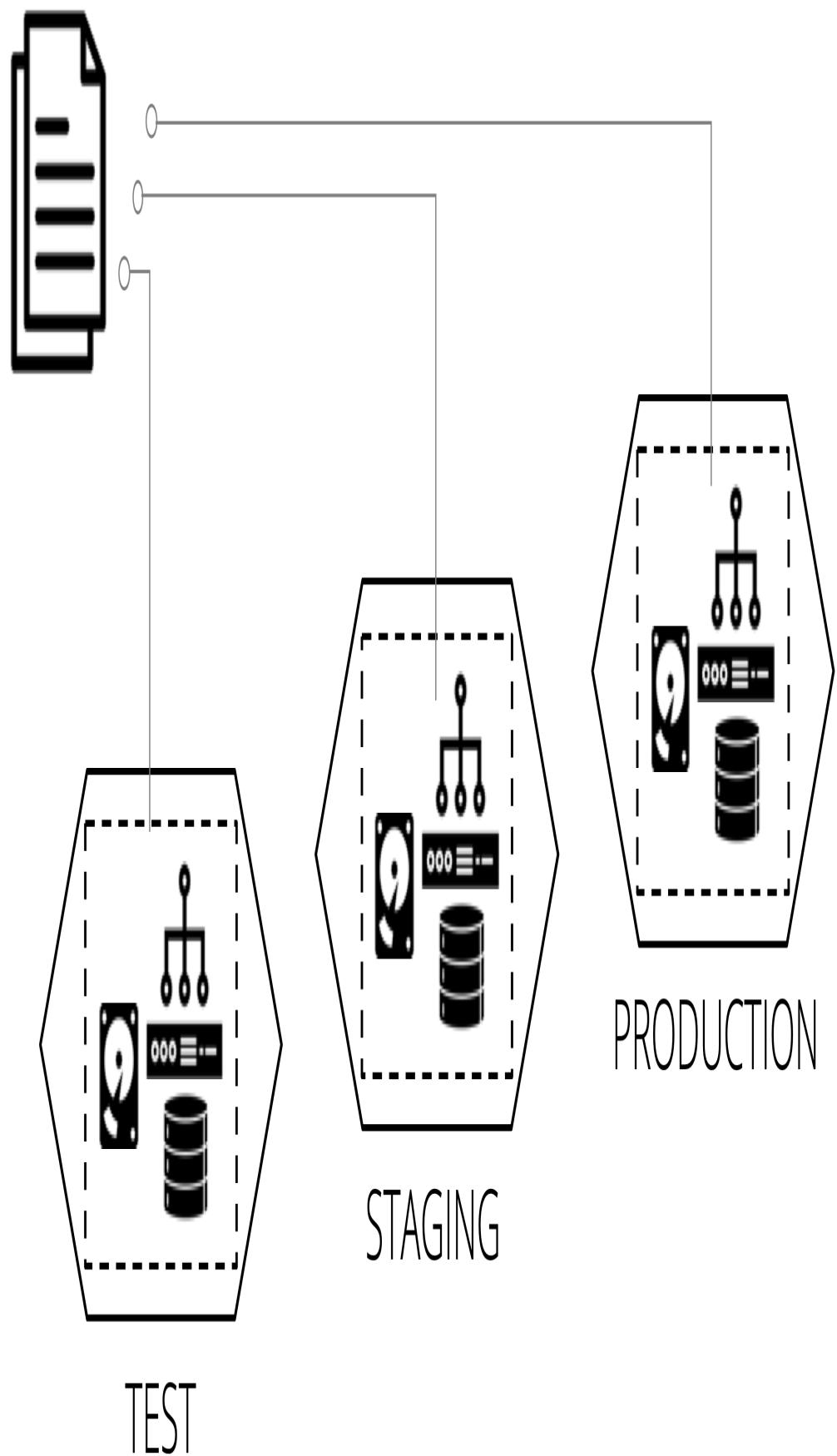


Figure 7-5. A Reusable Stack is an infrastructure source code project that is used to create multiple instances of a stack.

ALSO KNOWN AS

- Cookie Cutter Stack
- Template Stack
- Library Stack

MOTIVATION

You create a reusable stack to maintain multiple consistent instances of infrastructure. When you make changes to the stack code, you can apply and test it in one instance, and then use the same code version to create or update multiple additional instances. You want to provision new instances of the stack with minimal ceremony, maybe even automatically.

EXAMPLE OF HOW FOODSPIN MOVES TO REUSABLE STACKS

In “[Example: Foodspin’s application infrastructure entity module](#)”, the Foodspin team extracted common code from different stack projects that all used an application server. They put the common code into a module used by each of the stack projects. Later, they realized that the stack projects for their customer applications still looked very similar. In addition to using the module to create an application server, each stack had code to create databases and dedicated logging and reporting services for each customer.

Changing and testing changes to this code across multiple customers was becoming a hassle, and they were signing up new customers every month. So the team decided to create a single stack project that defines a customer application stack. This project still uses the shared Java application server module, as do a few other applications (Jira and GoCD). But the project also has the code for setting up the rest of the per-customer infrastructure as well.

Now, when they sign on a new customer, the team uses the common customer stack project to create a new instance. When they fix or improve something in the project codebase, they apply it to test instances to make sure it’s OK, and then they roll it out one by one to the customer instances.

APPLICABILITY

You can use a reusable stack for delivery environments or for multiple production environments. This pattern is useful when you don't need much variation between the environments. It is less applicable when environments need to be heavily customized.

CONSEQUENCES

The ability to provision and update multiple stacks from the same project enhances scalability, reliability, and throughput. You can manage more instances with less effort, make changes with a lower risk of failure, and roll changes out to more systems more rapidly.

You typically need to configure some aspects of the stack differently for different instances, even if it's just what you name things. I'll spend a whole chapter talking about this ([Chapter 8](#)).

You should test your stack project code before you apply changes to business-critical infrastructure. I'll spend multiple chapters on this, including [Chapter 9](#) and [Chapter 10](#).

IMPLEMENTATION

You create a reusable stack as an infrastructure stack project, and then run the stack management tool each time you want to create or update an instance of the stack. Use the syntax of the stack tool command to tell it which instance you want to create or update. With Terraform, for example, you would specify a different state file or workspace for each instance. With CloudFormation, you pass a unique stack ID for each instance.

The following example command provisions two stack instances from a single project using a fictional command called `stack`. The

command takes an argument `env` that identifies unique instances:

```
> stack up env=test --source mystack/src
SUCCESS: stack 'test' created
> stack up env=staging --source mystack/src
SUCCESS: stack 'staging' created
```

As a rule, you should use simple parameters to define differences between stack instances - strings, numbers, or in some cases, lists. Additionally, the infrastructure created by a reusable stack should not vary much across instances.

RELATED PATTERNS

The reusable stack is an improvement on the copy-paste environment antipattern (“[Antipattern: Copy-Paste Environments](#)”), making it easier to keep multiple instances consistent.

Stack code modules ([Chapter 6](#)) allow you to define code once and then share it across multiple stack projects. Unlike a reusable stack, a stack module is not used directly to create infrastructure—instead, a stack project imports modules. So modules are more of a supplement to reusable stacks than a substitute.

The wrapper stack pattern (“[Pattern: Wrapper Stack](#)”) uses modules to implement a combination of copy-paste environments (one project per instance) and a reusable stack (one copy of the code that defines the stack).

The patterns I describe in [Chapter 8](#) offer ways to configure instances of reusable stacks.

Building environments with multiple stacks

The reusable stack pattern describes an approach for implementing multiple environments. In [Chapter 5](#) I described different ways to structuring a system's infrastructure across multiple stacks (“Patterns and antipatterns for structuring stacks”). There are several ways you can implement your stacks to combine these two dimensions of environments and system structure.

The simple case is implementing the complete system as a single stack. When you provision an instance of the stack, you have a complete environment. I depicted this in the diagram for the reusable stack pattern ([Figure 7-5](#)).

But you should split larger systems into multiple stacks. For example, if you follow the service stack pattern (“[Pattern: Service Stack](#)”) you have a separate stack for each service:

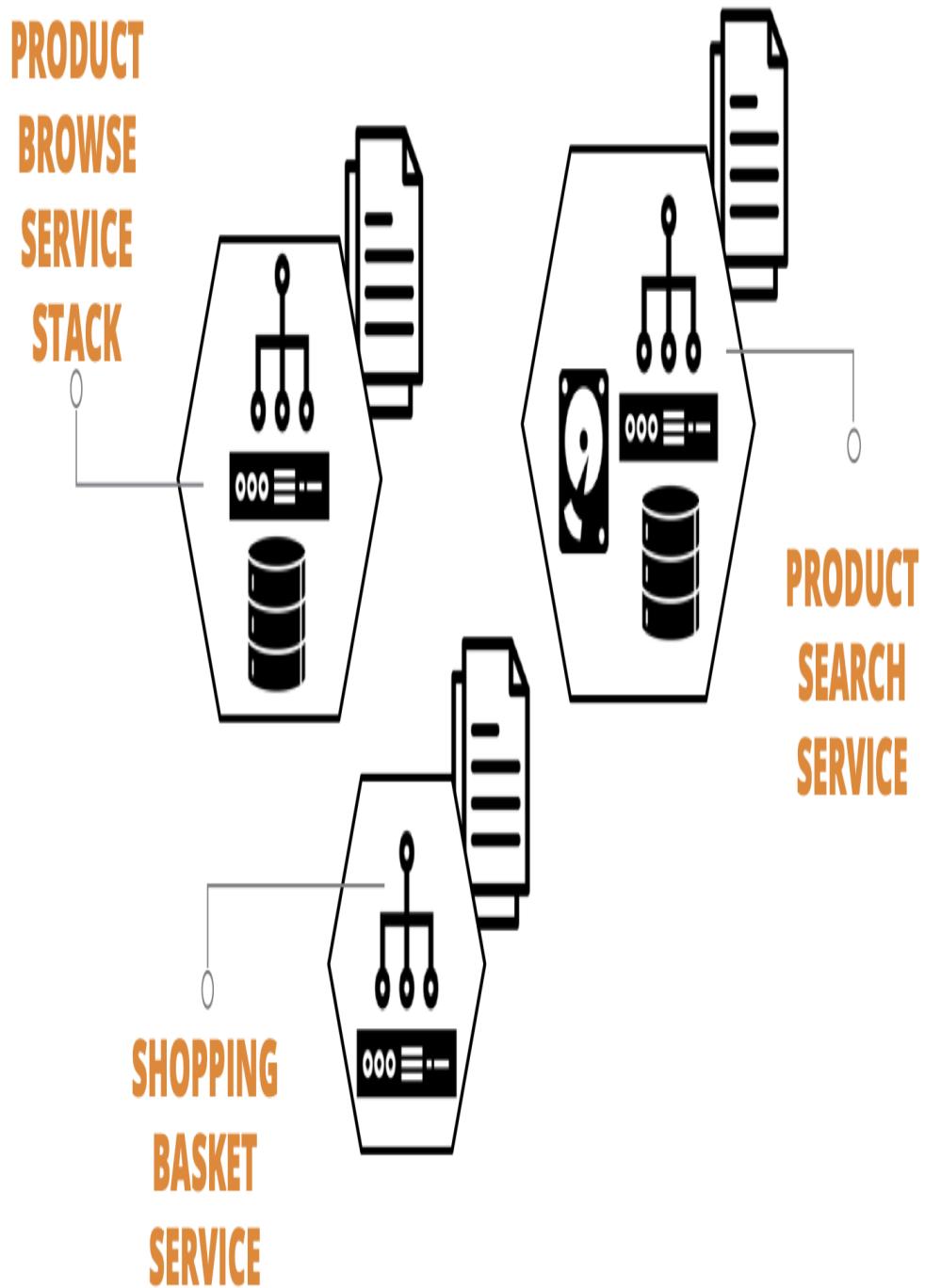


Figure 7-6. Example using a separate infrastructure stack for each service

To create multiple environments, you provision an instance of each service stack for each environment:

REUSABLE STACK PROJECTS

PRODUCT PRODUCT SHOPPING
BROWSE SEARCH BASKET
SERVICE SERVICE SERVICE

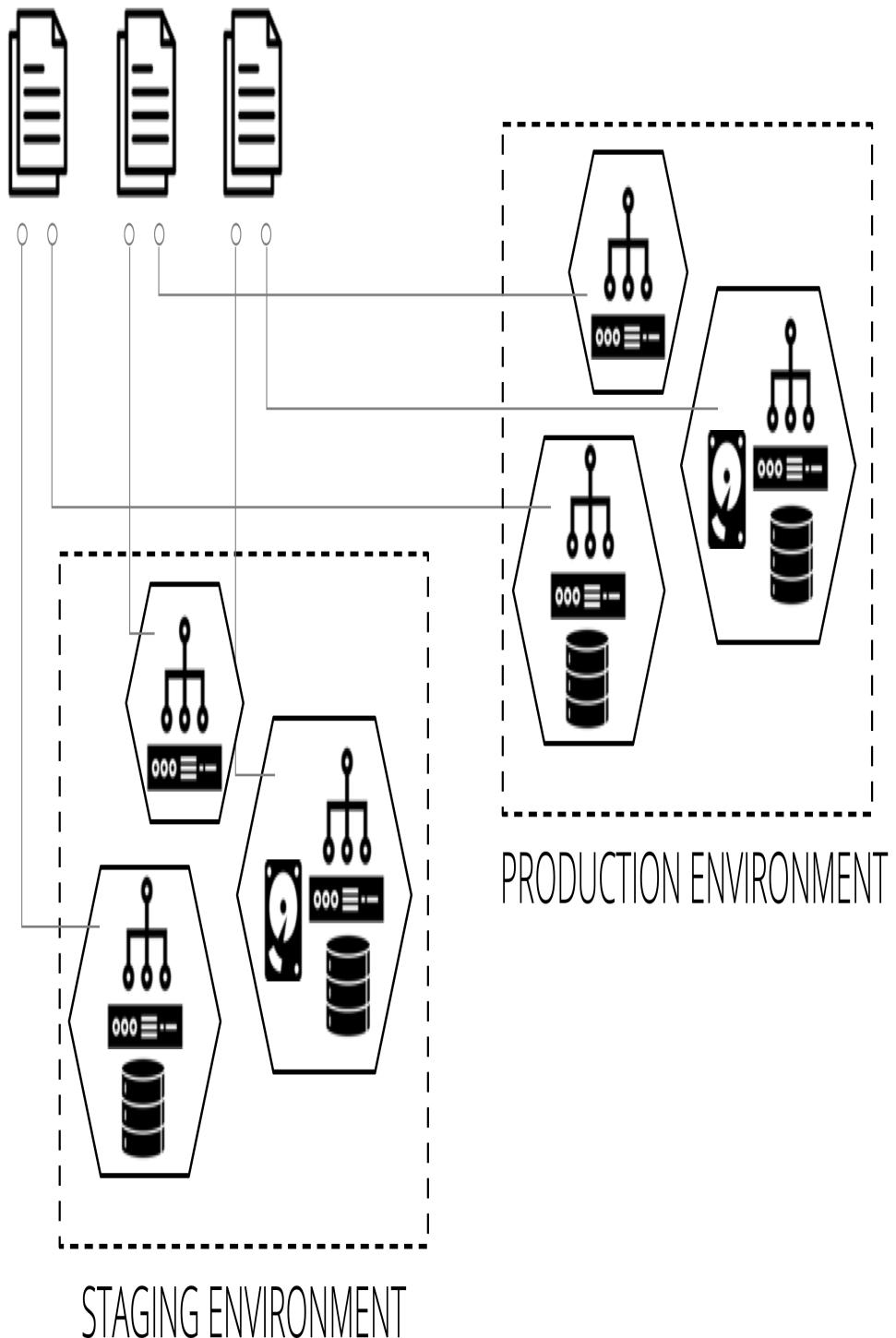


Figure 7-7. Using multiple stacks to build each environment

You would use commands like the following to build a full environment with multiple stacks:

```
> stack up env=staging --source product_browse_stack/src  
SUCCESS: stack 'product_browse-staging' created  
> stack up env=staging --source product_search_stack/src  
SUCCESS: stack 'product_search-staging' created  
> stack up env=staging --source shopping_basket_stack/src  
SUCCESS: stack 'shopping_basket-staging' created
```

In [Link to Come] I describe strategies for splitting systems into multiple stacks, and how to integrate infrastructure across stacks.

Conclusion

Reusable stacks should be the workhorse pattern for most teams who need to manage large infrastructures. This pattern creates challenges and opportunities. One challenge is how to configure each stack instance, which is the topic of [Chapter 8](#), the next chapter.

However, reusable stacks create the opportunity to test infrastructure code before you apply it to environments you care about. In [Chapter 9](#) I'll explain the core practice of continuously testing infrastructure code, and in [Chapter 10](#) I'll describe implementation patterns following this practice with infrastructure stacks.

¹ Charity Majors shared her painful experiences of working with a multiple-environment stack in a [blog post](#).

Chapter 8. Configuring Stacks

Using a single stack code project makes it easier to maintain multiple consistent instances of infrastructure, as I described in “[Pattern: Reusable Stack](#)”. However, you often need to customize stack instances. For example, you might run smaller clusters in development environments than in production. Here is an example of stack code that defines a container hosting cluster with configurable minimum and maximum numbers of servers:

```
container_cluster: web_cluster-${environment}
  min_size: ${cluster_minimum}
  max_size: ${cluster_maximum}
```

You pass different parameter values to this code for each environment, as depicted in Figure 8-1.

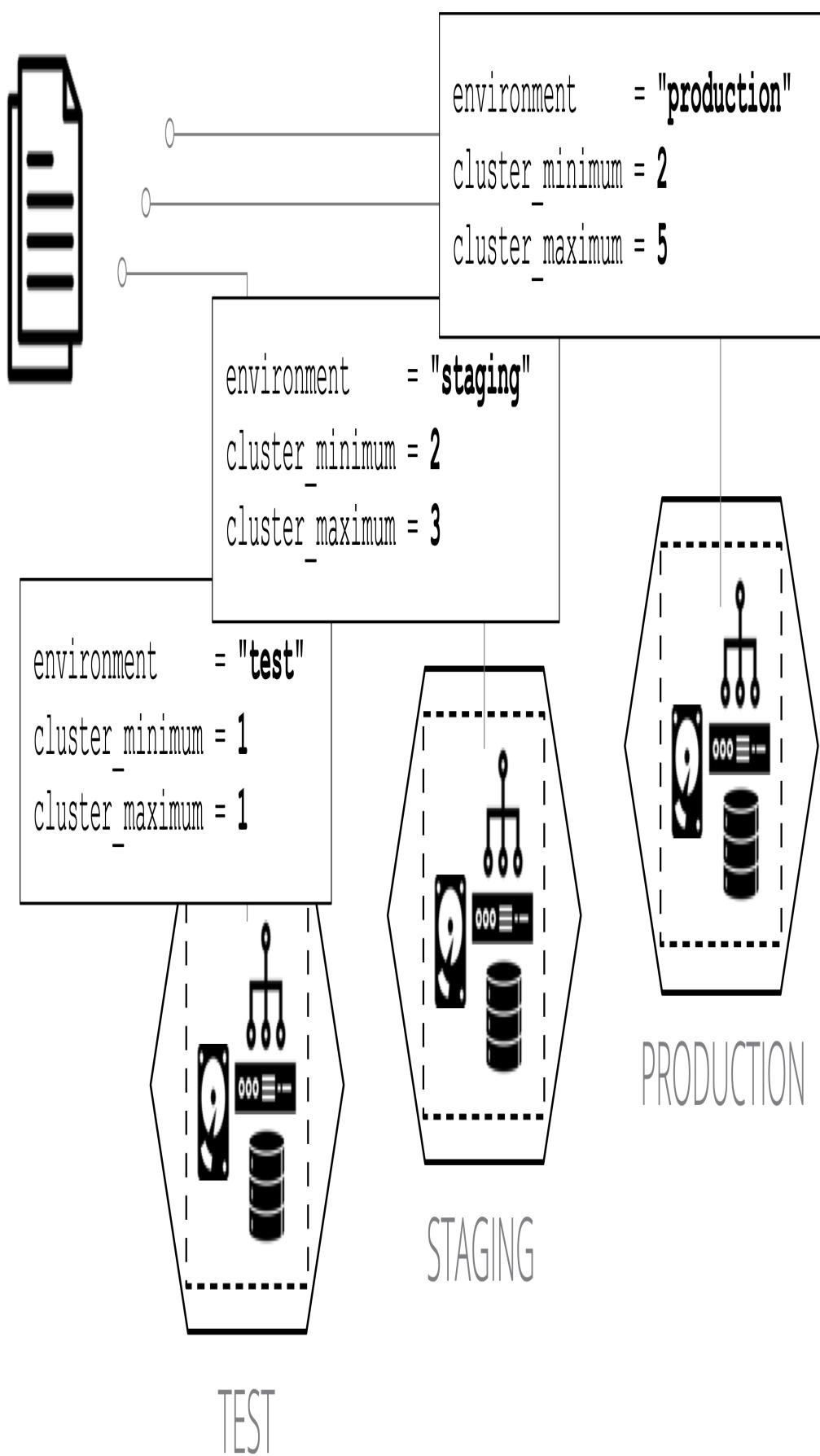


Figure 8-1. Using the same code with different parameter values for each environment

Stack tools such as Terraform and CloudFormation support multiple ways of setting configuration parameter values. These typically include passing values on the command-line, reading them from a file, and having the infrastructure code retrieve them from a key-value store.

Teams managing infrastructure need to decide how to use these features to manage and pass configuration values to their stack tool. It's essential to ensure the values are defined and applied consistently to each environment.

DESIGN PRINCIPLE: KEEP PARAMETERS SIMPLE

A major reason for defining your infrastructure as code is to ensure systems are consistently configured, as described in “[Principle: Minimize variation](#)”). Configurable stack code creates the opportunity for inconsistency. The more configurable a stack project is, the more difficult it is to understand the behavior of different instances, to ensure you're testing your code effectively, and to deliver changes regularly and reliably to all of your instances.

So it's best to keep stack parameters simple and to use them in simple ways.

- Prefer simple parameter types, like strings, numbers, and perhaps lists and key-value maps. Avoid passing more complex data structures.
- Minimize the number of parameters that you can set for a stack. Avoid defining parameters that you “might” need. Only add parameter when you have an immediate need for it. You can always add a parameter later on if you discover you need it.
- Avoid using parameters as conditionals that create significant differences in the resulting infrastructure. For example, a boolean (yes/no) parameter to indicates whether to provision a service within a stack adds complexity.

When it becomes hard to follow this advice, it's probably a sign that you should refactor your stack code, perhaps splitting it into multiple stack projects.

Using stack parameters to create unique identifiers

If you create multiple stack instances from the same project (as per the reusable stack pattern “Pattern: Reusable Stack”), you may see failures from infrastructure resources that require unique identifiers. To see what I mean, look at the following pseudo-code that defines an application server:

```
server:  
  id: appserver  
  subnet_id: appserver-subnet
```

The fictional cloud platform requires the `id` value to be unique, so when I run the `stack` command to create the second stack, it fails:

```
> stack up environment=test --source mystack/src  
SUCCESS: stack 'test' created  
> stack up environment=staging --source mystack/src  
FAILURE: server 'appserver' already exists in another stack
```

I can use parameters in my stack code to avoid these clashes. I change my code to take a parameter called `environment` and use it to assign a unique server ID. I also add the server into a different subnet in each environment:

```
server:  
  id: appserver-${environment}  
  subnet_id: appserver-subnet-${environment}"
```

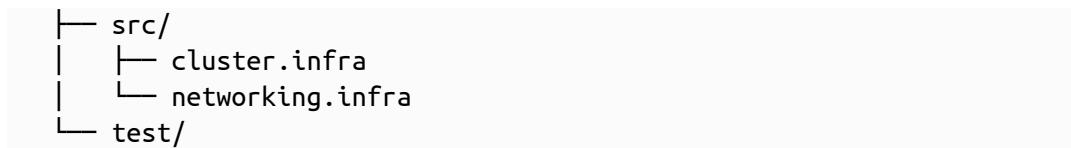
Now I can run my fictional `stack` command to create multiple stack instances without error.

Example stack parameters

I’ll use an example stack to compare and contrast the different stack configuration patterns in this chapter. The example is a template stack project that defines a container cluster, composed of

a dynamic pool of host nodes and some networking constructs. Here is the project structure:

Example 8-1. Example project structure for a template stack that defines a container cluster



The cluster stack uses the parameters listed in [Example 8-2](#) for three different stack instances. `environment` is a unique id for each environment, which can be used to name things and create unique identifiers. `cluster_minimum` and `cluster_maximum` define the range of sizes for the container host cluster. The infrastructure code in the file `cluster.infra` defines the cluster on the cloud platform, which scales the number of host nodes depending on load. Each of the three environments, *test*, *staging*, and *production*, uses a different set of values.

Example 8-2. Example parameter values used for the pattern descriptions

Stack Instance	environment	cluster_minimum	cluster_maximum
cluster_test	test	1	1
cluster_staging	staging	2	3
cluster_production	production	2	6

Handling secrets as parameters

In “[Secrets and source code](#)”, I explained that you should not store unencrypted secrets in source code. Two of the solutions I

described (“Injecting secrets at runtime” and “Disposable secrets”) are based on passing secrets as parameter to stack code. To implement these, you’ll need a way to manage those secrets on the system that runs your stack tool, whether it’s locally to a team member’s workstation or laptop, or on a compute instance that’s running the tool as part of a pipeline or other service.

I’ll include implementation details for secrets for each of the following patterns. You may want to use one pattern for normal, non-secret parameters, and a different pattern for secrets.

My examples also use an example secret, in this case, a passphrase for an SSL certificate passphrase, `ssl_cert_passphrase`. The value for this parameter in all environments is `+correct horse battery staple1`.

Patterns for configuring stacks

We’ve looked at why you need to parameterize stacks and a bit on how the tools implement parameters. Now I’ll describe some patterns and antipatterns for managing parameters and passing them to your tool:

- Manual Stack Parameters: Run the stack tool and type the parameter values on the command line.
- Scripted Parameters: Hard-code parameter values for each instance in a script that runs the stack tool.
- Stack Configuration Files: Declare parameter values for each instance in configuration files kept in the stack code project.

- **Wrapper Stack:** Create a separate infrastructure stack project for each instance, and import a shared module with the stack code.
- **Pipeline Stack Parameters.** Define parameter values in the configuration of a pipeline stage for each instance.
- **Stack Parameter Registry Pattern:** Store parameter values in a central location.

Antipattern: Manual Stack Parameters

The most natural approach to provide values for a stack instance is to type the values on the command-line manually, as in

Example 8-3.

Example 8-3. Example of manually typing command-line parameters

```
> stack up environment=production --source mystck/src
FAILURE: No such directory 'mystck/src'
> stack up environment=production --source mystack/src
SUCCESS: new stack 'production' created
> stack destroy environment=production --source mystack/src
SUCCESS: stack 'production' destroyed
> stack up environment=production --source mystack/src
SUCCESS: existing stack 'production' modified
```

MOTIVATION

It's dirt-simple to type values on the command-line, which is helpful when you're learning how to use a tool.

CONSEQUENCES

It's easy to make a mistake when typing a value on the command-line. It can also be hard to remember which values to type. For infrastructure that people care about, you probably don't want the risk of accidentally breaking something important by mistyping a command when making an improvement or fix. When multiple

people work on an infrastructure stack, as in a team, you can't expect everyone to remember the correct values to type for each instance.

Manual stack parameters aren't suitable for automatically applying infrastructure code to environments, such as with CI or CD.

IMPLEMENTATION

For the example parameters ([Example 8-2](#)), pass the values on the command-line according to the syntax expected by the particular tool. With my fictional `stack` tool the command looks like this:

```
stack up \
  environment=test \
  cluster_minimum=1 \
  cluster_maximum=1 \
  ssl_cert_passphrase="correct horse battery staple"
```

Anyone who runs the command needs to know the secrets, like passwords and keys, to use for a given environment and pass them on the command line. Your team should use a team password management tool² to store and share them between team members securely, and rotate secrets when people leave the team.

RELATED PATTERNS

The scripted parameters pattern ([“Pattern: Scripted Parameters”](#)) takes the command that you would type and puts it into a script.

The pipeline stack parameters pattern ([“Pattern: Pipeline Stack Parameters”](#)) does the same thing but puts them into the pipeline configuration instead of a script.

Pattern: Stack Environment Variables

The Stack Environment Variables pattern involves setting parameter values as environment variables for the stack tool to use. This pattern is often combined with another pattern to set the environment variables.

The environment variables are set beforehand (see the implementation section for more on how):

```
export STACK_ENVIRONMENT=test  
export STACK_CLUSTER_MINIMUM=1  
export STACK_CLUSTER_MAXIMUM=1  
export STACK_SSL_CERT_PASSPHRASE="correct horse battery staple"
```

There are different implementation options, but the most basic one is for the stack code to reference them directly:

```
container_cluster: web_cluster-${ENV("STACK_ENVIRONMENT")}  
min_size: ${ENV("STACK_CLUSTER_MINIMUM")}  
max_size: ${ENV("STACK_CLUSTER_MAXIMUM")}
```

MOTIVATION

Most platforms and tools support environment variables, so it's easy to do.

APPLICABILITY

If you're already using environment variables in your system and have suitable mechanisms to manage them, you might find it convenient to use them for stack parameters.

CONSEQUENCES

You need to use an additional pattern from this chapter to get the values to set. Doing this adds moving parts, making it hard to trace

configuration values for any given stack instance, and more work to change the values.

Using environment variables directly in stack code, as in the earlier example (???), arguably couples stack code too tightly to the runtime environment.

Setting secrets in environment variables may expose them to other processes that run on the same system.

IMPLEMENTATION

Again, you need to set the environment variables to use, which means selecting another pattern from this chapter. For example, if you expect people to set environment variables in their local environment to apply stack code, you are using the manual stack parameters antipattern (“Antipattern: Manual Stack Parameters”). You could set them in a script that runs the stack tool (the scripted parameters pattern “Pattern: Scripted Parameters”), or have the pipeline toolset them (“Pattern: Pipeline Stack Parameters”).

Another approach is to put the values into a script that people or instances import into their local environment. This is a variation of the stack configuration files pattern (“Pattern: Stack Configuration Files”). The script to set the variables would be exactly like the earlier example (???), and any command that runs the stack tool would import it into the environment:

```
source ./environments/staging.env
stack up --source ./src
```

Alternately, you could build the environment values into a compute instance that runs the stack tool. For example, if you

provision a separate CD agent node to run the stack tool to build and update stacks in each environment, the code to build the node could set the appropriate values as environment variables. Those environment variables would be available to any command that runs on the node, including the stack tool.

But to do this, you need to pass the values to the code that builds your agent nodes. So you need to select another pattern from this chapter to do that.

The other side of implementing this pattern is how the stack tool gets the environment values. The earlier example (<<???) shows how stack code can directly read environment variables.

But you could, instead, use a stack orchestration script ([Link to Come]) to read the environment variables and pass them to the stack tool on the command line. The code in the orchestration script would look like this:

```
stack up \
  environment=${STACK_ENVIRONMENT} \
  cluster_minimum=${STACK_CLUSTER_MINIMUM} \
  cluster_maximum=${STACK_CLUSTER_MAXIMUM} \
  ssl_cert_passphrase="${STACK_SSL_CERT_PASSPHRASE}"
```

This approach decouples your stack code from the environment it runs in.

RELATED PATTERNS

Any of the other patterns in this chapter can be combined with this one to set environment values.

Pattern: Scripted Parameters

Scripted Parameters involves hard-coding the parameter values into a script that runs the stack tool. You can write a separate script for each environment or a single script which includes the values for all of your environments:

```
if ${ENV} == "test"
    stack up cluster_maximum=1 env="test"
elseif ${ENV} == "staging"
    stack up cluster_maximum=3 env="staging"
elseif ${ENV} == "production"
    stack up cluster_maximum=5 env="production"
end
```

ALSO KNOWN AS

Environment provisioning script

MOTIVATION

Scripts are a simple way to capture the values for each instance, avoiding the problems with the “[Antipattern: Manual Stack Parameters](#)”. You can be confident that values are used consistently for each environment. By checking the script into version control, you ensure you are tracking any changes to the configuration values.

APPLICABILITY

A stack provisioning script is a useful way to set parameters when you have a fixed set of environments that don’t change very often. It doesn’t require the additional moving parts of some of the other patterns in this chapter.

Because it is wrong to hard-code secrets in scripts (as you know from reading “[Secrets and source code](#)”), this pattern is not suitable for secrets. That doesn’t mean you shouldn’t use this

pattern, only that you'll need to implement a separate pattern for dealing with secrets.

CONSEQUENCES

It's common for the commands used to run the stack tool to become complicated over time. Provisioning scripts can grow into messy beasts. [Link to Come] discusses how these scripts are used and outlines pitfalls and recommendations for keeping them maintainable. You should test provisioning scripts since they can be a source of issues with the systems they provision.

IMPLEMENTATION

There are two common implementations for this pattern. One is a single script that takes the environment as a command-line argument, with hard-coded parameter values for each environment. Example 8-4 is a simple example of this.

Example 8-4. Example of a script that includes the parameters for multiple environments

```
#!/bin/sh

case $1 in
test)
    CLUSTER_MINIMUM=1
    CLUSTER_MAXIMUM=1
    ;;
staging)
    CLUSTER_MINIMUM=2
    CLUSTER_MAXIMUM=3
    ;;
production)
    CLUSTER_MINIMUM=2
    CLUSTER_MAXIMUM=6
    ;;
*)
    echo "Unknown environment $1"
    exit 1
    ;;
esac
```

```
esac

stack up \
    environment=$1 \
    cluster_minimum=${CLUSTER_MINIMUM} \
    cluster_maximum=${CLUSTER_MAXIMUM}
```

Another implementation is a separate script for each stack instance, as in [Example 8-5](#).

Example 8-5. Example project structure with a script for each environment.

```
our-infra-stack/
  └── bin/
      ├── test.sh
      ├── staging.sh
      └── production.sh
  └── src/
      └── test/
```

Each of these scripts is identical but has different parameter values hard-coded in it. The scripts are smaller because they don't need logic to select between different parameter values. However, they need more maintenance. If you need to change the command, you need to make it across all of the scripts. Having a script for each environment also tempts people to customize different environments, creating inconsistency.

Commit your provisioning script or scripts to source control. Putting it in the same project as the stack it provisions ensures that it stays in sync with the stack code. For example, if you add a new parameter, you add it to the infrastructure source code and also to your provisioning script. You always know which version of the script to run for a given version of the stack code.

[Link to Come] discusses the use of scripts to run stack tools in much more detail.

As mentioned earlier, you shouldn't hard-code secrets into scripts, so you'll need to use a different pattern for those. You can use the script to support that pattern. In this example, a command-line tool fetches the secret from a secrets manager, following the parameter registry pattern (“Pattern: Stack Parameter Registry”):

Example 8-6. Fetching a key from a secrets manager in a script

```
...
# (Set environment specific values as in other examples)
...

SSL_CERT_PASSPHRASE=$(some-tool get-secret
id="/ssl_cert_passphrase/${ENV}")

stack up \
environment=${ENV} \
cluster_minimum=${CLUSTER_MINIMUM} \
cluster_maximum=${CLUSTER_MAXIMUM} \
ssl_cert_passphrase="${SSL_CERT_PASSPHRASE}"
```

The `some-tool` command connects to the secrets manager and retrieves the secret for the relevant environment using the ID `/ssl_cert_passphrase/${ENV}`. This example assumes the session is authorized to use the secrets manager. An infrastructure developer may use the tool to start a session before running this script. Or the compute instance that runs the script may be authorized to retrieve secrets using secretless authorization (as I described in “Secretless authorization”).

RELATED PATTERNS

Provisioning scripts run the command-line tool for you, so are a way to move beyond the manual stack parameters antipattern (“Antipattern: Manual Stack Parameters”). The stack configuration files pattern (“Pattern: Stack Configuration Files”) extracts the parameter values from the script into separate files.

Pattern: Stack Configuration Files

Stack Configuration Files manage parameter values for each instance in a separate file, which you manage in version control with your stack code.

```
└── src/
    ├── cluster.infra
    ├── host_servers.infra
    └── networking.infra
    └── environments/
        ├── test.properties
        ├── staging.properties
        └── production.properties
    └── test/
```

ALSO KNOWN AS

Environment configuration files

MOTIVATION

Creating configuration files for a stack's instances is straightforward and easy to understand. Because the file is committed to the source code repository, it is easy:

- To see what values are used for any given environment (“what is the maximum cluster size for production?”),
- to trace the history for debugging (“when did the maximum cluster size change?”),
- and to audit changes (“who changed the maximum cluster size?”).

Stack configuration files enforce the separation of configuration from the stack code.

APPLICABILITY

Stack configuration files are appropriate when the number of environments doesn't change often. They require you to add a file to your project to add a new stack instance. They also require (and help enforce) consistent logic in how different instances are created and updated, since the configuration files can't include logic.

CONSEQUENCES

When you want to create a new stack instance, you need to add a new configuration file to the stack project. Doing this prevents you from automatically creating new environments on the fly. In “[Pattern: Ephemeral test stack](#)”, I describe an approach for managing test environments that relies on creating environments automatically. You could work around this by creating a configuration file for an ephemeral environment on demand.

Parameter files can add friction for changing the configuration of downstream environments in a change delivery pipeline of the kind I describe in [Link to Come]. Every change to the stack project code must progress through each stage of the pipeline before being applied to production. It can take a while for this to complete and doesn't add any value when the configuration change is only applied to production.

Defining parameter values can be a source of considerable complexity in provisioning scripts. I'll talk about this more in [Link to Come], but as a teaser, consider that teams often want to define default values for stack projects, and for environments, and then need logic to combine these into values for a given instance of a given stack in a different environment. Inheritance models for parameter values can get messy and confusing.

Configuration files in source control should not include secrets. So for secrets, you either need to select an additional pattern from this chapter to handle secrets or implement a separate secrets configuration file outside of source control.

IMPLEMENTATION

You define stack parameter values in a separate file for each environment, as shown in the earlier example project structure (???).

The contents of a parameter file could look like this:

```
env = staging
cluster_minimum = 2
cluster_maximum = 3
```

Pass the path to the relevant parameter file when running the stack command:

```
stack up --source ./src --config ./environments/staging.properties
```

If the system is composed of multiple stacks, then it can get messy to manage configuration across all of the environments. There are two common ways of arranging parameter files in these cases. One is to put configuration files for all of the environments with the code for each stack:

```
cluster_stack/
  src/
    cluster.infra
    host_servers.infra
    networking.infra
  environments/
    test.properties
    staging.properties
    production.properties
appserver_stack/
```

```
└── src/
    ├── server.infra
    └── networking.infra
└── environments/
    ├── test.properties
    ├── staging.properties
    └── production.properties
```

The other is to centralize the configuration for all of the stacks in one place:

```
└── cluster_stack/
    ├── cluster.infra
    ├── host_servers.infra
    └── networking.infra
└── appserver_stack/
    ├── server.infra
    └── networking.infra
└── environments/
    ├── test/
    │   ├── cluster.properties
    │   └── appserver.properties
    ├── staging/
    │   ├── cluster.properties
    │   └── appserver.properties
    └── production/
        ├── cluster.properties
        └── appserver.properties
```

Each approach can become messy and confusing in its own way. When you need to make a change to all of the things in an environment, making changes to configuration files across dozens of stack projects is painful. When you need to change the configuration for a single stack across the various environments it's in, trawling through a tree full of configuration for dozens of other stacks is also not fun.

If you want to use configuration files to provide secrets, rather than using a separate pattern for secrets, then you need to manage those files outside of the project code checked into source control.

For local development environments, you can require users to create the file in a set location manually. Pass the file location to the stack command like this:

```
stack up --source ./src \  
  --config ./environments/staging.properties \  
  --config ../.secrets/staging.properties
```

In this example, you provide two `--config` arguments to the stack tool, and it reads parameter values from both. You have a directory named `.secrets` outside the project folder, so it is not in source control.

It can be trickier to do this when running the stack tool automatically, from a compute instance like a CD pipeline agent. You could provision similar secrets property files onto these compute instances, but that can expose secrets to other processes that run on the same agent. You also need to provide the secrets to the process that builds the compute instance for the agent, so you still have a bootstrapping problem.

RELATED PATTERNS

Putting configuration values into files simplifies the provisioning scripts described in “[Pattern: Scripted Parameters](#)”. You can avoid some of the limitations of environment configuration files by using the “[Pattern: Stack Parameter Registry](#)” instead. Doing this moves parameter values out of the stack project code and into a central location, which allows you to use different workflows for code and configuration.

Pattern: Wrapper Stack

A Wrapper Stack uses an infrastructure stack project for each instance as a wrapper to import a stack code module (see [Chapter 6](#)). Each wrapper project defines the parameter values for one instance of the stack. It then imports a module shared by all of the stack instances.

STACK CODE MODULE

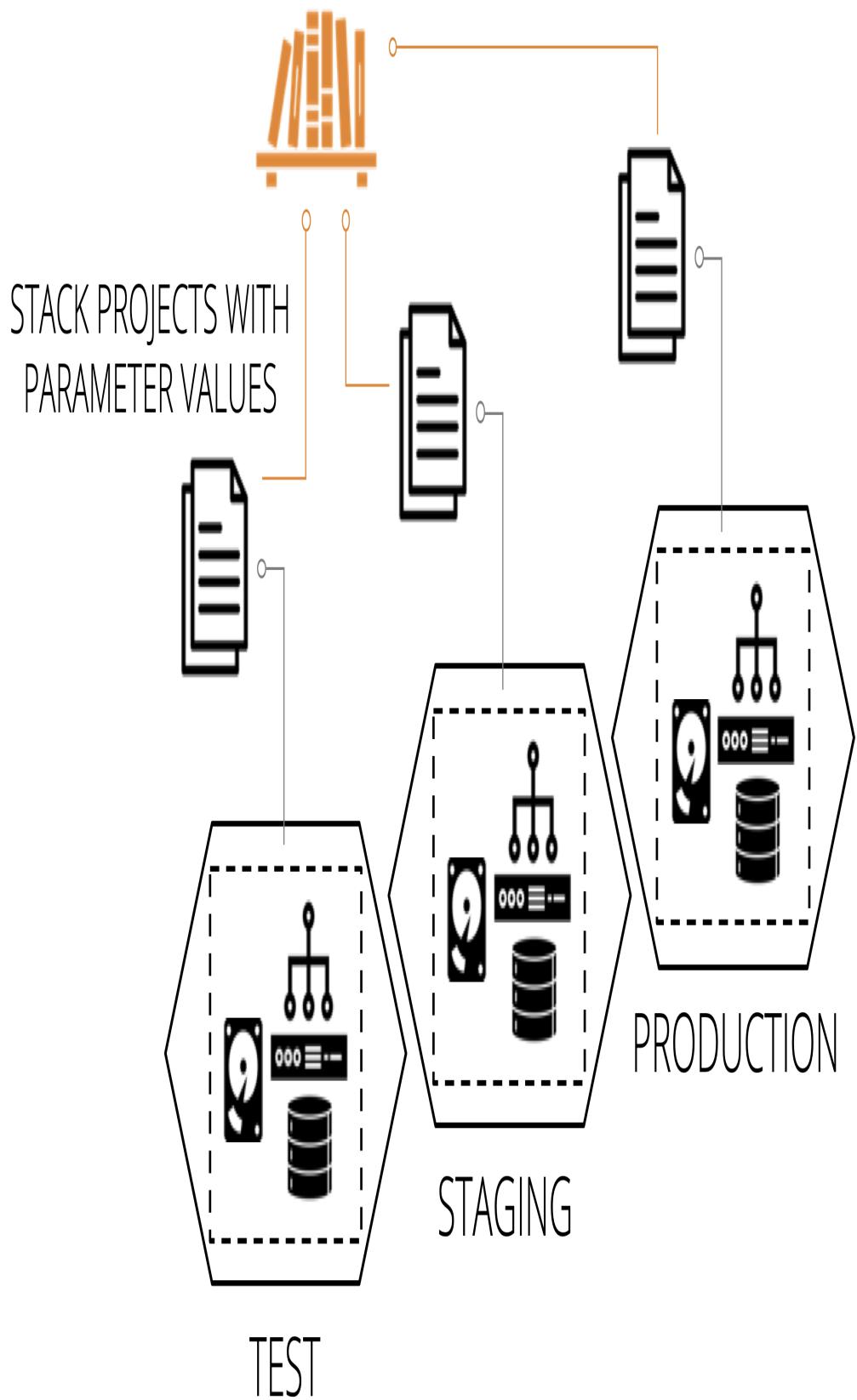


Figure 8-2. A Wrapper Stack uses an infrastructure stack project for each instance as a wrapper to import a stack code module

MOTIVATION

A wrapper stack leverages the stack tool’s module functionality to re-use shared code across stack instances. You can use the tool’s module versioning, dependency management, and artifact repository functionality to implement a change delivery pipeline ([Link to Come]). As of this writing, most infrastructure stack tools don’t have a project packaging format that you can use to implement pipelines for stack code. So you need to create a custom stack packaging process yourself. You can work around this by using a wrapper stack, and versioning and promoting your stack code as a module.

With wrapper stacks, you can write the logic for provisioning and configuring stacks in the same language that you use to define your infrastructure, rather than using a separate language as you would with a provisioning script (“[Pattern: Scripted Parameters](#)”).

CONSEQUENCES

Modules add an extra layer of complexity between your stack and the code contained in the module. You now have two levels: the stack project, which contains the wrapper projects, and the module which contains the code for the stack.

Because you have a separate code project for each stack instance, people may be tempted to add custom logic for each instance. Custom instance code makes your codebase inconsistent and hard to maintain.

Because you define parameter values in wrapper projects managed in source control, you can't use this pattern to manage secrets. So you need to add another pattern from this chapter to provide secrets to stacks.

IMPLEMENTATION

Each stack instance has a separate infrastructure stack project. For example, you would have a separate Terraform project for each environment. You can implement this like a copy-paste environment (“[Antipattern: Copy-Paste Environments](#)”), with each environment in a separate repository.

Alternatively, each environment project could be a folder in a single repository:

```
my_stack/
  └── test/
    └── stack.infra
  └── staging/
    └── stack.infra
  └── production/
    └── stack.infra
```

Define the infrastructure code for the stack as a module, according to your tool's implementation. You could put the module code in the same repository with your wrapper stacks. However, this would prevent you from leveraging module versioning functionality. That is, you wouldn't be able to use different versions of the infrastructure code in different environments, which is crucial for progressively testing your code.

The following example is a wrapper stack that imports a module called `container_cluster_module`, specifying the version of the module, and the configuration parameters to pass to it:

```
module:  
  name: container_cluster_module  
  version: 1.23  
  parameters:  
    env: test  
    cluster_minimum: 1  
    cluster_maximum: 1
```

The wrapper stack code for the *staging* and *production* environments is similar, other than the parameter values, and perhaps the module version they use.

The project structure for the module could look like this:

```
└── container_cluster_module/  
    ├── cluster.infra  
    ├── networking.infra  
    └── test/
```

When you make a change to the module code, you test and upload it to a module repository. How the repository works depends on your particular infrastructure stack tool. You can then update your test stack instance to import the new module version and apply it to the test environment.

Terragrunt is a stack orchestration tool that implements the wrapper stack pattern.

RELATED PATTERNS

A wrapper stack is similar to the scripted parameters pattern. The main differences are that it uses your stack tool's language rather than a separate scripting language and that the infrastructure code is in a separate module.

Pattern: Pipeline Stack Parameters

With the Pipeline Stack Parameters pattern, you define values for each instance in the configuration of a delivery pipeline.

I explain how to use a change delivery pipeline to apply infrastructure stack code to environments in [Link to Come]. You can implement a pipeline using a tool like Jenkins, GoCD, or ConcourseCI (see “[Delivery pipeline software and services](#)” for more on these tools).

STACK PROJECT

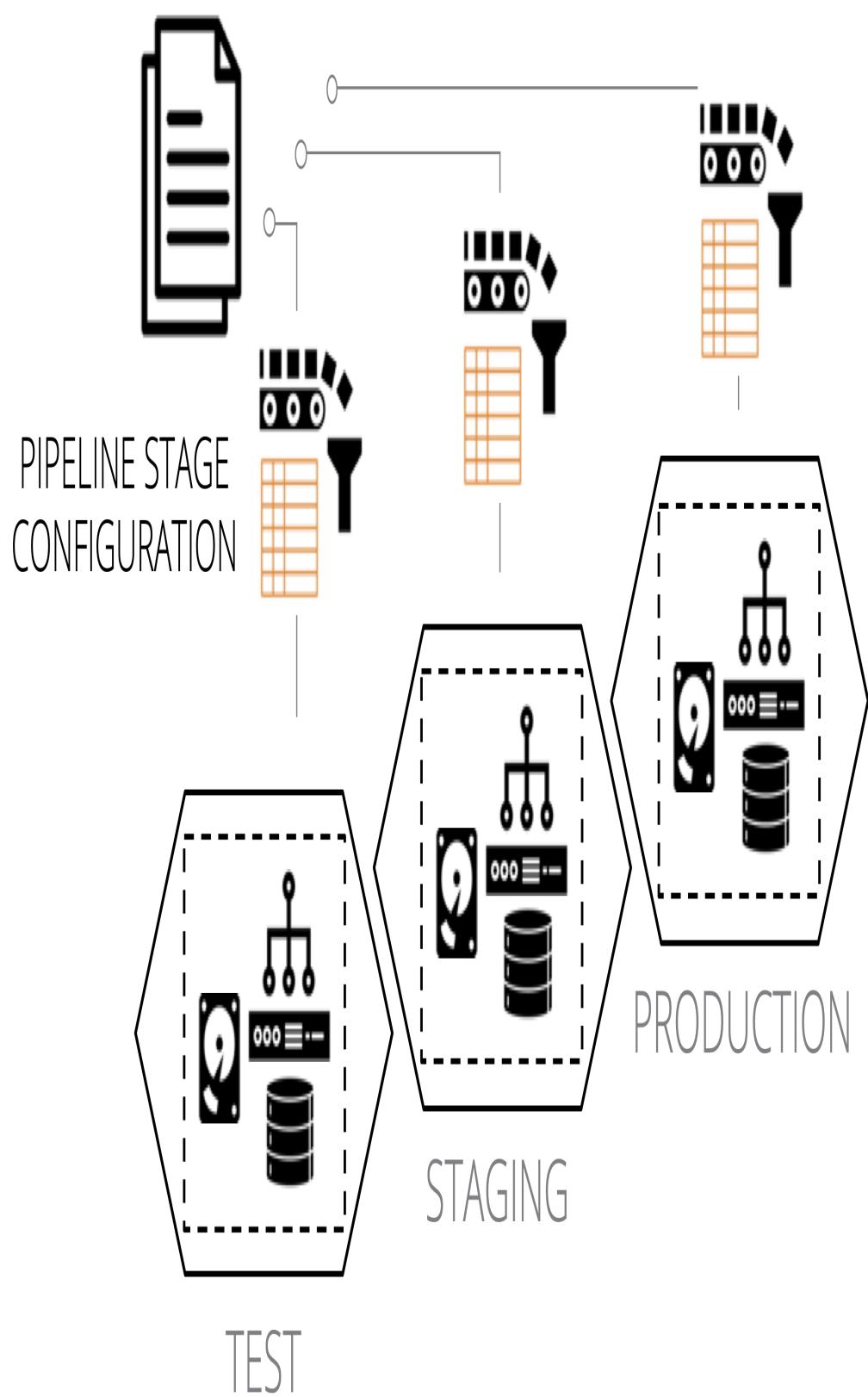


Figure 8-3. Each stage that applies the stack code passes the relevant configuration values for the environment.

MOTIVATION

If you're using a pipeline tool to run your infrastructure stack tool, it provides the mechanism for storing and passing parameter values to the tool out of the box. Assuming your pipeline tool is itself configured by code, then the values are defined as code and stored in version control.

Configuration values are kept separate from the infrastructure code. You can change configuration values for downstream environments and apply them immediately, without needing to progress a new version of the infrastructure code from the start of the pipeline.

APPLICABILITY

Teams who are already using a pipeline to apply infrastructure code to environments can easily leverage this to set stack parameters for each environment. However, if stacks require more than a few parameter values, defining these in the pipeline configuration has serious drawbacks, so you should avoid this.

CONSEQUENCES

By defining stack instance variables in the pipeline configuration, you couple configuration values with your delivery process. There is a risk of the pipeline configuration becoming complicated and hard to maintain.

The more configuration values you define in your pipeline, the harder it is to run the stack tool outside the pipeline. Your pipeline

can become a single point of failure—you may not be able to fix, recover, or rebuild an environment in an emergency until you have recovered your pipeline. And it can be hard for your team to develop and test stack code outside the pipeline.

In general, it's best to keep the pipeline configuration for applying a stack project as small and simple as possible. Most of the logic should live in a script called by the pipeline, rather than in the pipeline configuration.

CI SERVERS, PIPELINES, AND SECRETS

The first thing most attackers look for when they gain access to a corporate network is CI and CD servers. These are well-known treasure troves of passwords and keys that they can exploit to inflict the maximum evil on your users and customers.

Most of the CI and CD tools that I've worked with do not provide a very robust security model. You should assume that anyone who has access to your pipeline tool or who can modify code that the tool executes (i.e., probably every developer in your organization) can access any secret stored by the tool.

This is true even when the tool encrypts the secrets, because the tool can also decrypt the secrets. If you can get the tool to run a command, you can usually get it to decrypt any secret it stores. You should carefully analyze any CI or CD tool you use to assess how well it supports your organization's security requirements.

IMPLEMENTATION

Parameters should be implemented using “as code” configuration of the pipeline tool:

Example 8-7. Example pipeline stage configuration

```
stage: apply-test-stack
  input_artifacts: container_cluster_stack
  commands:
    unpack ${input_artifacts}
    stack up --source ./src environment=test cluster_minimum=1
    cluster_maximum=1
    stack test environment=test
```

This example passes the values on the command line. You may also set them as environment variables that the stack code uses (as described in “[Pattern: Stack Environment Variables](#)”):

Example 8-8. Example pipeline stage configuration using environment variables

```
stage: apply-test-stack
  input_artifacts: container_cluster_stack
  environment_vars:
    STACK_ENVIRONMENT=test
    STACK_CLUSTER_MINIMUM=1
    STACK_CLUSTER_MAXIMUM=1
  commands:
    unpack ${input_artifacts}
    stack up --source ./src
    stack test environment=test
```

In this example, the pipeline toolsets those environment variables before running the **commands**.

Many pipeline tools provide secret management features that you can use to pass secrets to your stack command. You set the secret values in the pipeline tool in some fashion, and can then refer to them in your pipeline job:

Example 8-9. Example pipeline stage with secret

```
stage: apply-test-stack
  input_artifacts: container_cluster_stack
  commands:
    unpack ${input_artifacts}
    stack up --source ./src environment=test \
      cluster_minimum=1 \
      cluster_maximum=1 \
      ssl_cert_passphrase=${STACK_SSL_CERT_PASSPHRASE}
```

RELATED PATTERNS

Defining the commands and parameters to apply stack code for each environment in pipeline configuration is similar to the scripted parameters pattern. The difference is where the scripting lives-in the pipeline configuration versus in script files.

Pattern: Stack Parameter Registry

A Stack Parameter Registry manages the parameter values for stack instances in a central location, rather than with your stack code. The stack tool retrieves the relevant values when it applies the stack code to a given instance.

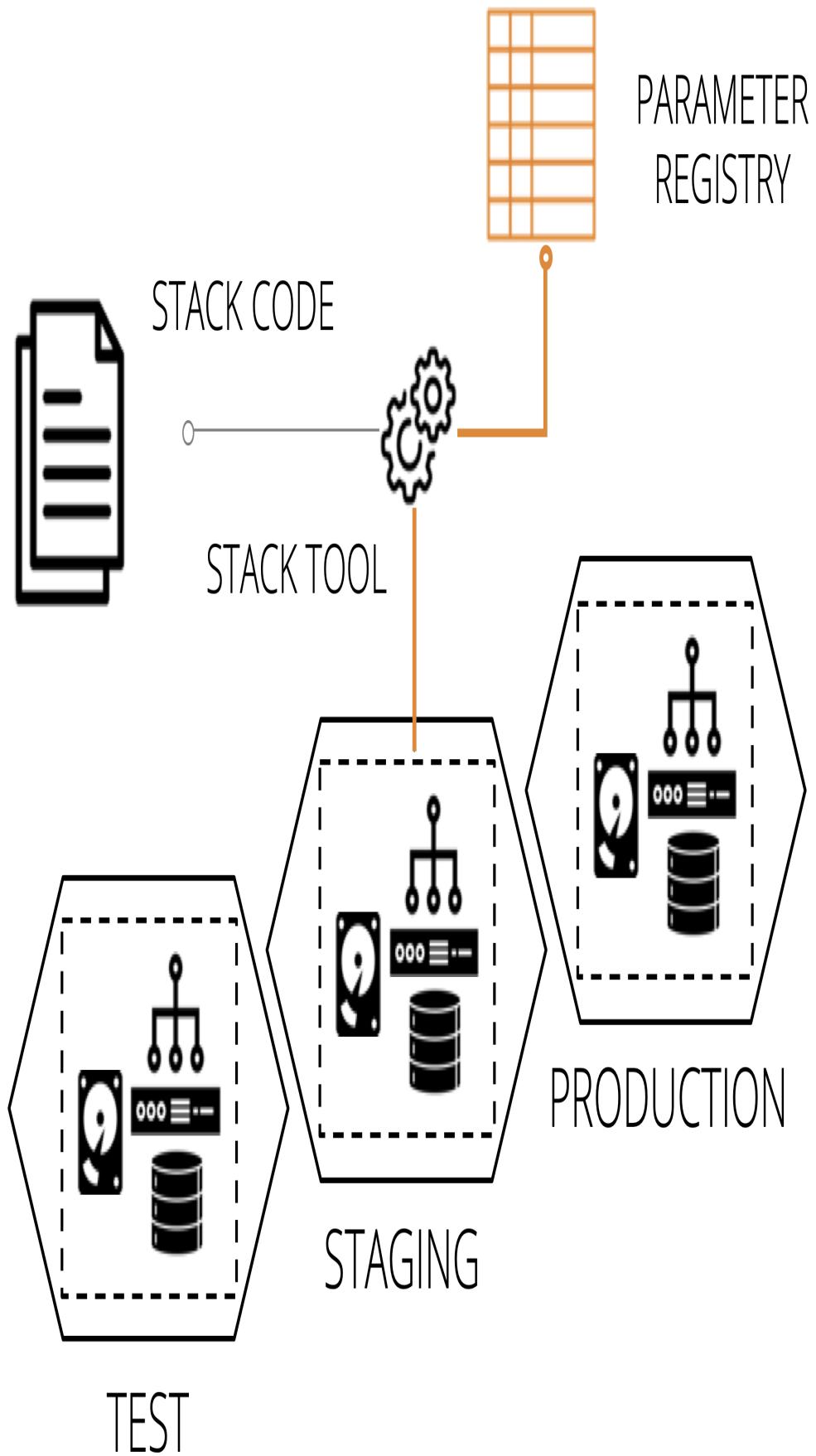


Figure 8-4. Stack instance parameter values stored in a central registry.

CONFIGURATION REGISTRIES AND STACK PARAMETER REGISTRIES

I use the term “configuration registry” to describe a service which stores configuration values that may be used for many purposes, including service discovery, stack integration, or monitoring configuration. I’ll describe this in more detail in [“Configuration Registry”](#).

When talking specifically about storing configuration values for stack instances, I use the term “stack parameter registry”. So a stack parameter registry is a specific use case for a configuration registry.

ALSO KNOWN AS

Configuration registry for stacks, Infrastructure configuration registry

MOTIVATION

Storing parameter values in a registry separates configuration from implementation. Parameters in a registry can be set, used, and viewed by different tools, using different languages and technologies. This flexibility reduces coupling between different parts of the system. You can replace any tool that uses the registry without affecting any other tool that uses it.

Because they are tool-agnostic, stack parameter registries can act as a source of truth for infrastructure and even system configuration, acting as a Configuration Management Database (CMDB - see [“Configuration Management Database \(CMDB\)”](#)). This configuration data can be useful in regulated contexts, making it easy to generate reports for auditing.

APPLICABILITY

If you are using a configuration registry for other purposes, it makes sense to use it as a stack parameter registry, as well. For example, a configuration registry is a useful way to integrate multiple stacks (see [Link to Come]).

CONSEQUENCES

A stack parameter registry requires a configuration registry, which is an extra moving part for your overall system. The registry is a dependency for your stack and a potential point of failure. If the registry becomes unavailable, it may be impossible to re-provision or update the infrastructure stack until you can restore it. This dependency can be painful in disaster recovery scenarios, putting the registry service on the critical path.

Managing parameter values separately from the stack code that uses it has tradeoffs. You can change the configuration of a stack instance without making a change to the stack project. If one team maintains a reusable stack project, other teams can use it to create their own stack instances without needing to add or change configuration files in the stack project itself.

On the other hand, making changes across more than one place-stack project and parameter registry-adds complexity and opportunity for mistakes.

IMPLEMENTATION

I'll discuss ways to implement a parameter registry below (“Configuration Registry”). In short, it may be a service that stores key/value pairs, or it could be a file or directory structure of files

that contain key/value pairs. Either way, parameter values can usually be stored in a hierarchical structure, so you can store and find them based on the environment and the stack, and perhaps other factors like the application, service, team, geography, or customer.

The values for this chapter's example container cluster could look like:

Example 8-10. Example of configuration registration entries

```
└ environments/
  └ test/
    └ container_cluster/
      └ cluster_minimum = 1
      └ cluster_maximum = 1
  └ staging/
    └ container_cluster/
      └ cluster_minimum = 2
      └ cluster_maximum = 3
  └ production/
    └ container_cluster/
      └ cluster_minimum = 2
      └ cluster_maximum = 6
```

When you apply the infrastructure stack code to an instance, the stack tool uses the key to retrieve the relevant value. You will need to pass the `environment` parameter to the stack tool, and the code uses this to refer to the relevant location in the registry:

```
cluster:
  id: container_cluster-${environment}
  minimum:
    ${get_registry_item("/environments/${environment}/container_cluster/c
luster_minimum")}
  maximum:
    ${get_registry_item("/environments/${environment}/container_cluster/cl
uster_maximum")}
```

The `get_registry_item()` function in the stack code looks up the value.

This implementation ties your stack code to the configuration registry. You need the registry to run and test your code, which can be too heavy. You could work around this by fetching the values from the registry in a script. The script then passes them to the stack code as normal parameters. Doing this gives you the flexibility to set parameter values in other ways. For reusable stack code this is particularly useful, giving users of your code more options for how to configure their stack instances.

Secrets management services (“[Secrets management](#)”) are a special type of parameter registry. Used correctly, they ensure that secrets are only available to people and services that require them, without exposing them more widely. Some configuration registry products and services can be used to store both secret and non-secret values. But it’s important to avoid storing secrets in registries which don’t protect them. Doing so makes the registry an easy target for attackers.

RELATED PATTERNS

You probably need to pass at least one parameter to the stack tool to indicate which stack instance’s parameters to use. You can use either the stack provisioning script or pipeline stack parameter pattern for this.

Configuration Registry

Larger organizations with many teams working across larger systems with many moving parts often find a configuration registry useful. It can be useful for configuring stacks instances, as I described in “[Pattern: Stack Parameter Registry](#)”. It can also be useful for managing integration dependencies across different

stack instances, applications, and other services, as I'll explain in [Link to Come].

And a registry can provide a useful source of information about the composition and state of your infrastructure. You can use this to create tools, dashboards, and reports, as well as for monitoring and auditing your systems.

So it's worth digging into how to implement and use a configuration registry.

Implementing a Configuration Registry

There are different ways to build a configuration registry. You can use a registry provided out of the box by your infrastructure automation tool. Or you can run a general-purpose registry product. Most cloud providers also have configuration registry services that you can use. If you are brave, you can hand-roll a practical registry using fairly basic pieces.

INFRASTRUCTURE AUTOMATION TOOL REGISTRIES

Many infrastructure automation toolchains include a configuration registry service. These tend to be part of a centralized service that may also include features such as source code management, monitoring, dashboards, and command orchestration. Examples of these include:

- Chef Infra Server
- PuppetDB
- Ansible Tower

- Salt Mine

You may be able to use these services with tools outside the toolchain that provides them. Most can expose values, so you could write a script that discovers information about the current state of infrastructure managed by the configuration tool. Some infrastructure tool registries are extensible, so you can use them to store the data from other tools.

However, this creates a dependency on whatever toolchain provides the registry service. The service may not fully support integration with third-party tools. They might not offer a contract or API that guarantees future compatibility.

So if you're considering using an infrastructure tool's data store as a general-purpose configuration registry, consider how well it supports this use case, and what kind of lock-in it creates.

GENERAL PURPOSE CONFIGURATION REGISTRY PRODUCTS

There are many dedicated configuration registry and key-value store database products available outside the toolchains of a particular automation tool. Some examples include:

- Zookeeper
- Consul³
- etcd
- doozerd

These are generally compatible with different tools, languages, and systems, so avoid locking you into any particular tool-chain.

However, it can take a fair bit of work to define how data should be stored. Should keys be structured like *environment/service/application*, *service/application/environment*, or something else entirely? You may need to write and maintain custom code to integrate different systems with your registry. And a configuration registry gives your team yet another thing to deploy and run.

PLATFORM REGISTRY SERVICES

Most cloud platforms provide a key-value store service, such as the AWS SSM Parameter Store. These give you most of the advantages of a general-purpose configuration registry product, without forcing you to install and support it yourself. However, it does tie you to that cloud provider. In some cases, you may find yourself using a registry service on one cloud to manage infrastructure running on another!

DIY CONFIGURATION REGISTRIES

Rather than running a configuration registry server, some teams build a custom lightweight configuration registry by storing configuration files in a central location, or by using distributed storage. They typically use an existing file storage service like an object store (e.g., an S3 bucket on AWS), a version control system, networked filesystem, or even a web server.

A variation of this is packaging configuration settings into system packages, such as a *.deb* or *.rpm* file, and pushing them to an internal APT or YUM repository. You can then download configuration files to local servers using the standard package management tool.

Another variation is using a standard relational or document store database server.

All of these approaches leverage existing services, so they can be quick to implement for a simple project rather than needing to install and run a new server. But when you get beyond trivial situations, you may find yourself building and maintaining the functionality that you could get off the shelf.

Single or multiple configuration registries

Combining all configuration values from across all of your systems, services, and tools is an appealing idea. You could keep everything in one place rather than sprawling across many different systems. “One registry to rule them all.” However, this isn’t always practical in larger, more heterogeneous environments.

Many tools, such as monitoring services and server configuration systems, have their own registry. You’ll often find different registry and directory products that are very good at specific tasks, such as license management, service discovery, and user directories. Bending all of these tools to use a single system creates an ongoing flow of work. Every update to every tool needs evaluation, testing, and potentially more work to maintain the integration.

It may be better to pull relevant data from across the services where they are stored. Make sure you know which system is the source of truth for any particular data or configuration item. Design your systems and tools with this understanding.

Some teams use messaging systems to share configuration data as events. Whenever a system changes a configuration value, it sends an event. Other systems can monitor the event queue for changes to configuration items in which they are interested.

Configuration Management Database (CMDB)

A Configuration Management Database (CMDB) is a directory of information about IT assets. Many organizations use these for auditing, control, and governance. They usually include physical assets like servers, laptops, and racks, as well as software, configuration, versions, and licenses.

The cloud age is challenging for traditional approaches to CMDBs. Assumptions about relationships between software, configuration, data, and hardware no longer hold with virtualization and cloud platforms. Top-down processes to approve and make configuration changes struggle to cope with systems that automatically add, remove, and reconfigure resources on the fly.

It's useful to consider the outcomes you need from a CMDB and look at ways to achieve these in dynamic environments. Too many organizations implement CMDBs, assuming there is value in assembling all information in one place. Instead, they should start by considering how they will use the information, and work back from there to find the best solutions.

Two reasons organizations use CMDBs are to provide visibility and to control changes.

CMDB FOR VISIBILITY

Data visibility supports many use cases. These include managing costs, identifying policy conflicts, and surfacing security vulnerabilities. For each of your organization's use cases, be sure you understand the requirement. Identify how information needs to be presented to support it, for example, dashboards, alerts, and reports.

Then, look at ways to implement the presentation of the data, and work backward to solutions for collating it.

It's essential to understand which system is the source of truth, or *system of record*, for a given type of data. You may need to extract data from multiple systems of record for a particular use case. For many purposes, it's best to collect data directly, rather than marshal it into a separate dataset. For example, writing scripts that directly probe your systems is more reliable for auditing and compliance than examining a CMDB, which may or may not be accurate and timely.

CMDB TO CONTROL CHANGES

Some organizations use a CMDB system as a way to configure their systems from a central location. A CMDB product may offer sophisticated permission models and workflows to create tight controls over changes.

You should consider the different elements of your system and how each is defined and changed. For configuration parameters such as stack instance parameters, you might want a system that ensures changes are made by authorized users and are tracked and approved where appropriate.

These solutions may not work as well for code that defines infrastructure stacks, server configuration, deployment processes, and the like. The idea of infrastructure as code is to manage changes to these using source control systems, automated tests, and change delivery pipelines. These provide capabilities for authorization, auditing, and quality enforcement. Adding another tool in addition to these, especially one not designed to support agile engineering practices, usually adds more complexity, friction, and risk than value.

Conclusion

The past few chapters have demonstrated how to apply the core practice of defining systems as code to infrastructure stacks. Each chapter has described patterns and antipatterns for implementing stacks as code.

The next chapter moves on to the next core practice of infrastructure as code, continuously validating code as you work on it. Following that, I'll describe implementation patterns to apply that practice to infrastructure stacks. Afterward, I'll show how to use stacks to build application runtime environments-by which I mean servers, containers, and clusters-following these core practices.

-
- 1 This passphrase comes from Randall Munroe's XKCD comic [Password Strength](#). I use it here so that if you haven't read that comic, you will, because it makes an important point about good passwords. And if you have read the comic, then you and I share the smug glow that comes from recognizing a somewhat obscure reference.
 - 2 Some examples of tools teams can use to securely share passwords include [GPG](#), [KeePass](#), [1Password](#), [Keeper](#), and [LastPass](#)

- 3 Consul is a product of Hashicorp, which also makes Terraform, and of course, these products work well together. But Consul was created and is maintained as an independent tool, and is not required for Terraform to function. This is why I count it as a general-purpose registry product.

Chapter 9. Core Practice: Continuously validate all work in progress

Continuous validation is the second of the three core practices of infrastructure as code¹:

- Define everything as code
- Continuously validate all work in progress
- Build small, simple pieces that you can change independently

Testing is a cornerstone of agile software engineering. Extreme Programming (XP) emphasizes writing tests first (TDD) and frequently integrating code (CI)². Continuous Delivery (CD)³ extends this to validate the full production readiness of code as developers work on it, rather than waiting until they finish working on a release.

If a strong focus on testing creates good results when writing application code, it's reasonable to expect it to be useful for infrastructure code as well. In this chapter, I explore strategies for testing and delivering infrastructure. I draw heavily on agile engineering approaches to quality, including TDD, CI, and CD.

Even teams experienced with application testing struggle to test infrastructure very well. So I'll explain some of the challenges of testing your infrastructure code. I'll also discuss models for

thinking about testing strategy, including the test pyramid and Swiss cheese model, and how they relate to infrastructure. Then I'll explain how to use delivery pipelines to implement testing strategies. In the chapter following this one, I'll describe some specific patterns and techniques for testing infrastructure stacks.

But first, let's consider what continuous validation means for infrastructure code.

Why continuously validate infrastructure code?

Testing changes to your infrastructure is clearly a good idea. But the need to build and maintain a suite of test automation code may not be as clear. We often think of building infrastructure as a one-off activity: build it, test it, then use it. Why spend the effort to create all that test code?

Creating an automated testing suite is hard work, especially when you consider the work needed to implement the delivery and testing tools and services - CI servers, pipelines, test runners, test scaffolding, and various types of scanning and validation tools. When getting started with infrastructure as code, building all of these things may seem like more work than building the systems you'll run on them.

In “Use Infrastructure as Code to optimize for change” I explained the rationale for implementing systems for delivering changes to infrastructure. To recap, you make changes to any non-trivial system far more often than you might expect. You create a new system by making a series of changes. You evolve the design and

implementation as you learn more about the system you're building, and about the technology you're using to build it.

And once a system is live, if it has any traction with its users, you need to continuously improve, update, patch, fix, and grow it, which again involves a continuous series of changes.

The advice in this book aims to help you to continuously validate over as broad a scope of risk as possible.

What continuous validation means

One of the cornerstones of agile engineering is validating as you work-build *quality in*. The earlier you can find out whether each line of code you write is ready for production, the faster you can work, and the sooner you can deliver value. Finding problems more quickly also means spending less time going back to investigate problems and less time fixing and rewriting code. Fixing problems continuously avoids accumulating less technical debt.

Most people get the importance of fast feedback. But what differentiates genuinely high performing teams is how aggressively they pursue truly *continuous* feedback.

Traditional approaches involve testing after the team has implemented the system's complete functionality. Timeboxed methodologies take this further. The team tests periodically during development, such as at the end of a sprint, or perhaps with nightly builds. Teams following Lean or Kanban test each story as they complete it.

Lean and Kanban approaches test each “story⁴” as you complete it.

Truly continuous validation involves testing even more frequently than this. People write and run tests as they code. And they frequently push their code into a centralized, automated validation system-ideally at least once a day⁵.

People need to get feedback as soon as possible when they push their code so that they can respond to it with as little interruption to their flow of work as possible. Tight feedback loops are the essence of Continuous Integration.

IMMEDIATE VALIDATION AND EVENTUAL VALIDATION

Another way to think of this is to classify each of your validation activities as either immediate or eventual. Immediate validation happens when you push your code. Eventual validation happens after some delay, perhaps after a manual review, or maybe on a schedule.

Ideally, validation is truly immediate, happening as you code. These are validation activities that run in your editor, such as syntax highlighting, or running unit tests. Your editor may support this, or you may use a utility like `inotifywait` or `entr` to run checks in a terminal when your code changes.

Another example of immediate validation is [pair programming](#), which is essentially a code review that happens as you work. Pairing provides much faster feedback than code reviews that happen after you’ve finished working on a story or feature, and someone else finds time to review what you’ve done.

The CI build and the CD pipeline should run immediately every time someone pushes a change to the codebase. Running immediately on each change not only gives them feedback faster. It also ensures a small scope of change for each run. If the pipeline only runs periodically, it may include multiple changes from multiple people. If any of the validations fail, it’s harder to work out which change caused the issue, meaning more people need to get involved and spend time to find and fix it.

What should we validate with infrastructure?

The essence of Continuous Integration is to validate every change someone makes as soon as possible. The essence of Continuous Delivery is to maximize the scope of that validation. As Jez

Humble says, “We achieve all this by ensuring our code is always in a deployable state.”⁶

Quality assurance is about managing the risks of applying our code to our systems. Will the code break when we apply it? Does it create the right infrastructure? Does the infrastructure do what we need it to do? Does it meet operational criteria for performance, reliability, and security? Does it comply with regulatory and governance rules?

Continuous Delivery is about broadening the scope of risks that are immediately validated when pushing a change to the codebase, rather than waiting for eventual validation days, weeks, or even months afterwards. So on every push, a pipeline applies the code to realistic test environments and subjects it to comprehensive validation. Ideally, once the code has run through the automated stages of the pipeline, it’s fully proven as production-ready.

Teams should identify the risks that come from making changes to their infrastructure code, and create a repeatable process for validating any given change against those risks. This process takes the form of automated test suites and manual validation activities. A test suite is a collection of automated tests that are run as a group.

When people think about automated testing, they generally think about functional tests like unit tests and UI-driven journey tests. But the scope of risks is broader than functional defects, so the scope of validation is broader as well. Examples of things that you may want to validate, whether automatically or manually, include:

Code quality

Is the code readable and maintainable? Does it follow the team's standards for how to format and structure code?

Depending on the tools and languages you're using, some tools can scan code for syntax errors, compliance with formatting rules, and run complexity analysis. Depending on how long they've been around, and how popular they are, infrastructure languages may not have many (or any!) of these tools. Manual review methods include gated code review processes, code showcase sessions, and pair programming.

Functionality

Does it do what it should? Ultimately, functionality is tested by deploying the applications and checking that they work correctly. Doing this indirectly tests that the infrastructure is correct, but it's useful to catch some types of issues more quickly. An example of this for infrastructure is network routing. Can an HTTPS connection be made from the public Internet to the web servers? It's often possible to test this kind of thing using a subset of the entire infrastructure.

Security

You can test security at a variety of levels, from code scanning to unit testing to integration testing and production monitoring. There are some tools specific to security testing, such as vulnerability scanners. It may also be useful to write security tests into standard test suites. For example, unit tests can make assertions about open ports, user account handling, or access permissions.

Compliance

Systems may need to comply with laws, regulations, industry standards, contractual obligations, or organizational policies. Ensuring and proving compliance can be time-consuming for infrastructure and operations teams. Automated validation can be enormously useful with this, both to catch violations

quickly and to provide evidence for auditors. As with security, you can do this at multiple levels of validation, from code-level to production validation.

Performance

Automated tools can validate how quickly specific actions complete. Testing the speed of a network connection from point A to point B can surface issues to do with the network configuration or the cloud platform if run before you even deploy an application. Finding performance issues on a subset of your system is another example of how you can get faster feedback.

Scalability

Automated tests can prove that scaling works correctly, for example, checking that an auto-scaled cluster adds nodes when it should. Tests can also check whether scaling gives you the outcomes that you expect. For example, perhaps adding nodes to the cluster doesn't improve capacity, due to a bottleneck somewhere else in the system. Having these tests run frequently means you'll discover quickly if a change to your infrastructure breaks your scaling.

Availability

Similarly, automated testing can prove that your system would be available in the face of potential outages. Your tests can destroy resources, such as nodes of a cluster, and verify that the cluster automatically replaces them. You can also test that scenarios that aren't automatically resolved are handled gracefully, for example, showing an error page and avoiding data corruption.

Operability

You can automatically validate any other system requirements needed for operations. Teams can test monitoring (inject errors and prove that monitoring detects and reports them), logging, and automated maintenance activities.

Each of these types of validations can be applied at more than one level of scope, from server configuration to stack code to the fully integrated system. I'll discuss this in “[Progressive validation](#)”. But first I'd like to address the things which make infrastructure especially difficult to test.

Challenges with testing infrastructure code

Most of the teams I encounter who work with infrastructure as code struggle to implement the same level of automated testing and delivery for their infrastructure code as they have for their application code. And many teams without a background in agile software engineering find it even more difficult.

The premise of infrastructure as code is that we can apply software engineering practices such as agile testing to infrastructure. But there are significant differences between infrastructure code and application code. So we need to adapt some of the techniques and mindsets from application testing to make them practical for infrastructure.

Here are a few challenges that arise from the differences between infrastructure code and application code.

Challenge: Tests for declarative code often have low value

As mentioned in [Chapter 4 \(“Building infrastructure with declarative code”\)](#), many infrastructure tools use declarative languages, rather than procedural languages. Declarative code

typically declares the desired state for some infrastructure, such as this code that defines a networking subnet:

```
subnet:  
  name: private_A  
  address_range: 192.168.0.0/16
```

A test for this would simply re-state the code:

```
assert:  
  subnet("private_A").exists  
assert:  
  subnet("private_A").address_range is("192.168.0.0/16")
```

A suite of low-level tests of declarative code can become a bookkeeping exercise. Every time you change the infrastructure code, you change the test to match. What value do these tests provide? Well, testing is about managing risks, so let's consider what risks the example test above can uncover:

1. The infrastructure code was never applied
2. The infrastructure code was applied, but the tool failed to apply it correctly, without returning an error
3. Someone changed the infrastructure code but forgot to change the test to match

The first risk may be a real one, but it doesn't require a test for every single declaration. Assuming you have code that does multiple things on a server, a single test would be enough to reveal that, for whatever reason, the code wasn't applied.

The second risk boils down to protecting yourself against a bug in the tool you're using. The tool developers should fix that bug or your team should switch to a more reliable tool. I've seen teams

use tests like this in cases where they found a specific bug, and wanted to protect themselves against it. Testing for this is okay to cover a known issue, but it is wasteful to blanket your code with detailed tests just in case your tool has a bug.

The last risk is circular logic. Removing the test would remove the risk it addresses, and also remove work for the team.

There are some situations when it's useful to test declarative code. Two that come to mind are when the declarative code can create different results, and when you combine multiple declarations.

TESTING VARIABLE DECLARATIVE CODE

That example of declarative code is simple - the values are hard-coded, so the result of applying the code is clear. Variables introduce the possibility of creating different results, which may create risks that make testing more useful. Variables don't always create variation that needs testing. What if we add some simple variables to the earlier example?

```
subnet:  
  name: ${MY_APP}-${MY_ENVIRONMENT}  
  address_range: ${SUBNET_IP_RANGE}
```

There isn't much risk in this code that isn't already managed by the tool that applies it. If someone sets the variables to invalid values, the tool should fail with an error.

The code becomes riskier when there are more possible outcomes. Let's add some conditional code to the example:

```
subnet:  
  name: ${MY_APP}-${MY_ENVIRONMENT}
```

```
address_range: choose_unique_subset(  
    get_vpc(${MY_ENVIRONMENT}).address_range, 16)
```

This code has some logic which might be worth testing. It calls two functions, `choose_unique_subset` and `get_vpc`, either of which might fail or return a result that interacts in unexpected ways with the other function.

The outcome of applying this code varies based on inputs and context, which makes it worth writing tests.

NOTE

Imagine that instead of calling these functions, you wrote the code to select a subset of the address range as a part of this declaration for your subnet. This is an example of mixing declarative and functional code (as I discussed in “[Implementation Principle: Avoid mixing different types of code](#)”). The tests for the subnet code would need to include various edge cases of the functional code—for example, what happens if the parent range is smaller than the range needed?

If your declarative code is complex enough that it needs complex testing, it is a sign that you should pull some of the logic out of your declarations and into a library written in a procedural language. You can then write clearly separate tests for that function, and simplify the test for the subnet declaration.

TESTING COMBINATIONS OF DECLARATIVE CODE

Another situation where testing is more valuable is when you have multiple declarations for infrastructure that combine into more complicated structures. For example, you may have code that defines multiple networking structures - an address block, load balancer, routing rules, and gateway. Each piece of code would probably be simple enough that tests would be unnecessary. But

the combination of these produces an outcome that is worth testing

- that someone can make a network connection from point A to point B.

Testing that the tool created the things declared in code is usually less valuable than testing that they enable the outcomes you want.

Challenge: Testing infrastructure code is slow

To test infrastructure code, you need to apply it to relevant infrastructure. And provisioning an instance of infrastructure is often slow, especially when you need to create it on a cloud platform. Most teams who struggle to implement automated infrastructure testing find that the time to create test infrastructure is a barrier for fast feedback.

The solution is usually a combination of strategies:

Divide infrastructure into more tractable pieces

It's useful to include testability as a factor in designing a system's structure, as it's one of the key ways to make the system easy to maintain, extend, and evolve. Making pieces smaller is one tactic, as smaller pieces are usually faster to provision and test. It's easier to write and maintain tests for smaller, more loosely coupled pieces since they are simpler and have less surface area of risk. [Link to Come] discusses this topic in more depth.

Clarify, minimize, and isolate dependencies

Each element of your system may have dependencies on other parts of your system, on platform services, and on services and systems that are external to your team, department, or organization. These impact testing, especially if you need to rely on someone else to provide instances to support your test.

They may be slow, expensive, unreliable, or have inconsistent test data, especially if other users share them. Test doubles are a useful way to isolate a component so that you can test it quickly. You may use test doubles as part of a progressive testing strategy, first testing your component with test doubles, and later testing it integrated with other components and services.

Progressive testing

You'll usually have multiple test suites to test different aspects of the system. You can run faster tests first, to get quicker feedback if they fail, and only run slower, broader-scoped tests after those have passed. I'll delve into this in “[Progressive validation](#)”.

Choice of ephemeral or persistent instances

You may create and destroy an instance of the infrastructure each time you test it (an *ephemeral instance*), or you may leave an instance running in between runs (*persistent instances*). Using ephemeral instances can make tests significantly slower, but are cleaner and give more consistent results. Keeping persistent instances cuts the time needed to run tests, but may leave changes and accumulate inconsistencies over time. Choose the appropriate strategy for a given set of tests, and revisit the decision based on how well it's working. I provide more concrete examples of implementing ephemeral and persistent instances in “[Pattern: Ephemeral test stack](#)”.

Online and offline tests

Some types of validation are *online*, requiring you to provision infrastructure on the “real” cloud platform. Others can run *offline* on your laptop or a build agent. Validation that you can run offline includes code syntax checking and tests that run in a virtual machine or container instance. Consider the nature of your various tests, and be aware of which ones can run where. Offline validation is usually much faster, so you'll tend to run

them earlier. You can use test doubles to emulate your cloud API offline for some tests.

With any of these strategies, you should regularly assess how well they are working. If tests are unreliable, either failing to run correctly or returning inconsistent results, then you should drill into the reasons for this and either fix them or replace them with something else. If tests rarely fail, or if the same tests almost always fail together, you may be able to strip them out to simplify your test suite. If you spend more time finding and fixing problems that originate in your tests rather than in the code you’re testing, look for ways to simplify and improve them.

TEST DOUBLES

Mocks, fakes, and stubs are all types of *test doubles*. A test double replaces a dependency needed by a component so you can test it in isolation. These terms tend to be used in different ways by different people, but I’ve found the definitions used by Gerard Meszaros in his [xUnit patterns book](#) to be useful.⁷

In the context of infrastructure, there are a growing number of tools⁸ that allow you to mock the APIs of cloud vendors. You can apply your infrastructure code to a local mocked cloud to validate some aspects of the code. These won’t tell you whether your networking structures work correctly, but they should tell you whether they’re roughly valid.

Progressive validation

Most non-trivial systems use multiple suites of tests to validate changes. Different suites may validate different things (as listed in “[What should we validate with infrastructure?](#)”). One suite may validate one concern offline, such as checking for security vulnerabilities by scanning code syntax. Another suite may run online checks for the same concern, for example, by probing a running instance of an infrastructure stack for security vulnerabilities.

In theory, you could run all of your test suites in parallel when you change your code. In practice, this is wasteful. Firstly, you probably need to provision a lot of infrastructure to test everything at once. Secondly, it's potentially wasteful. If the code fails to pass the faster tests, there's no point in running the slower, more expensive tests until you fix the problem.

Progressive validation involves running test suites in a sequence. The sequence builds up, starting with simpler tests that run more quickly over a smaller scope of code, then building up to more comprehensive tests over a broader set of integrated components and services.

Models like the *test pyramid* and *Swiss cheese testing* offer visual metaphors for progressive validation. A delivery pipeline ([Link to Come]) is a technical implementation of progressive testing, with a series of stages, each running a set of test suites.

Validation stages

Each system needs a unique test strategy, which includes defining the validation stages and deciding the order for running them. There are several characteristics for each validation stage, including the components under test, the dependencies involved, and the environment required.

SCOPE OF COMPONENTS BEING VALIDATED

In a progressive validation strategy, earlier stages validate individual components, while later stages integrate components and test them together. For example, an earlier stage might test code that installs and configures a web server package onto a

virtual machine. A later stage tests a server with this package installed, running in an infrastructure stack with firewall rules and network routes.

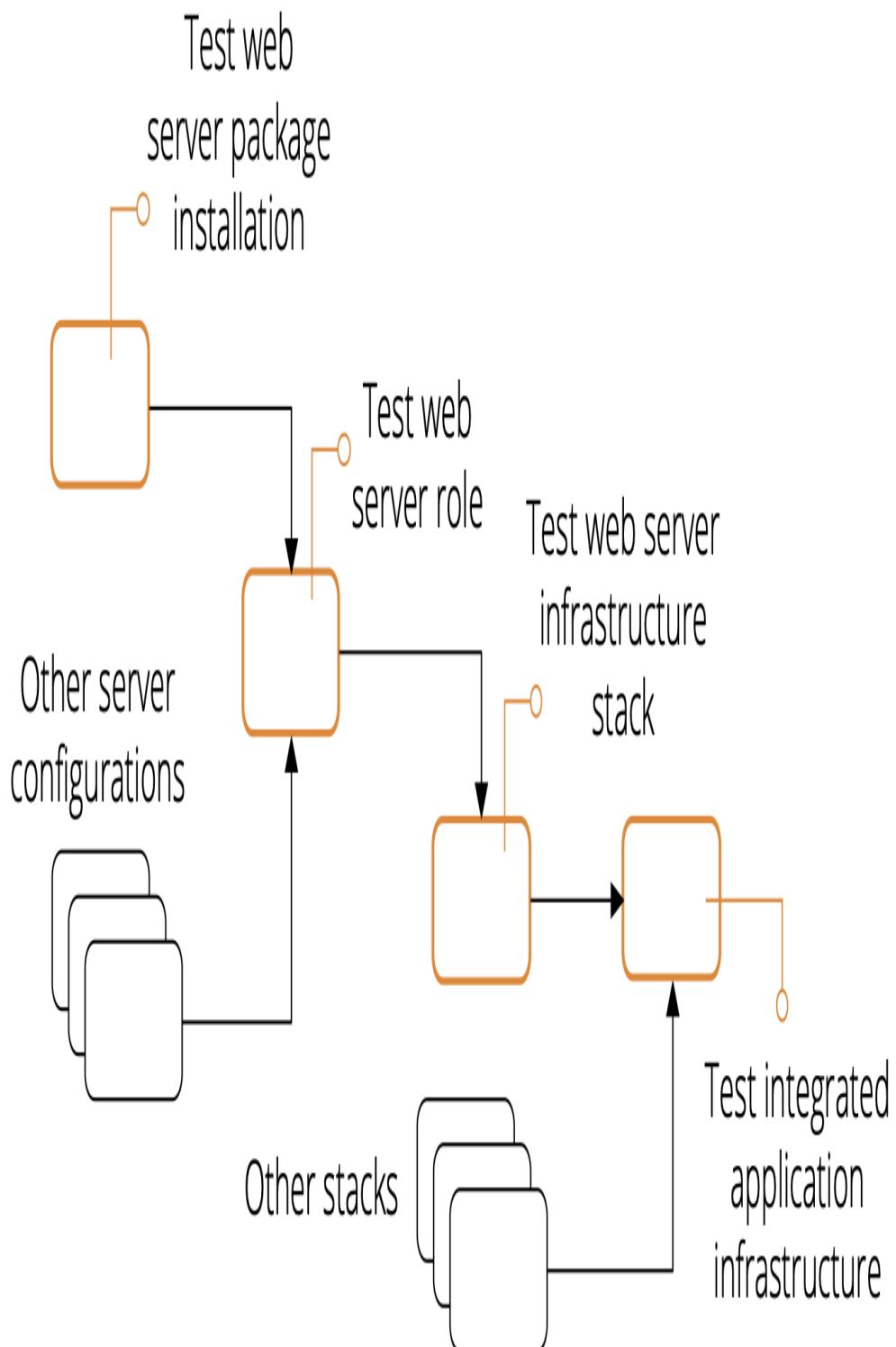


Figure 9-1. Progressive integration and testing of components

One stage might run validations for multiple components, such as a suite of unit tests. Or, different components may each have a separate validation stage.

UNIT TESTS

A unit test⁹ validates a subsection of a system, usually testing a single component at the lowest level for that system. With object oriented software this is often a class, or related set of classes. With infrastructure, this might be a single server configuration element-Chef cookbook, Puppet manifest, or Ansible playbook, for example.

A unit test in software code executes a small subsection of a program, on the order of a one or two classes, to make sure that they run correctly. Most infrastructure tools have some kind of unit testing tool or framework, such as `rspec-puppet` and `ChefSpec`. Saltstack even comes with its own built-in unit testing support.

Tools like this allow a particular configuration definition to be run without actually applying it to a server. They usually include functionality to emulate other parts of a system well enough to check that the definition behaves correctly. This requires ensuring that each of your definitions, scripts, and other low-level elements can be independently executed. Restructuring things to make test isolation possible may be challenging, but results in a cleaner design.

SCOPE OF DEPENDENCIES USED FOR THE STAGE

Many elements of a system depend on other services. An application server stack might connect to an identity management service to handle user authentication. To progressively validate

this, you might first run a stage that tests the application server without the identity management service, perhaps using a mock service to stand in for it. A later stage would run additional tests on the application server integrated with a test instance of the identity management service, and the production stage would integrate with the production instance:

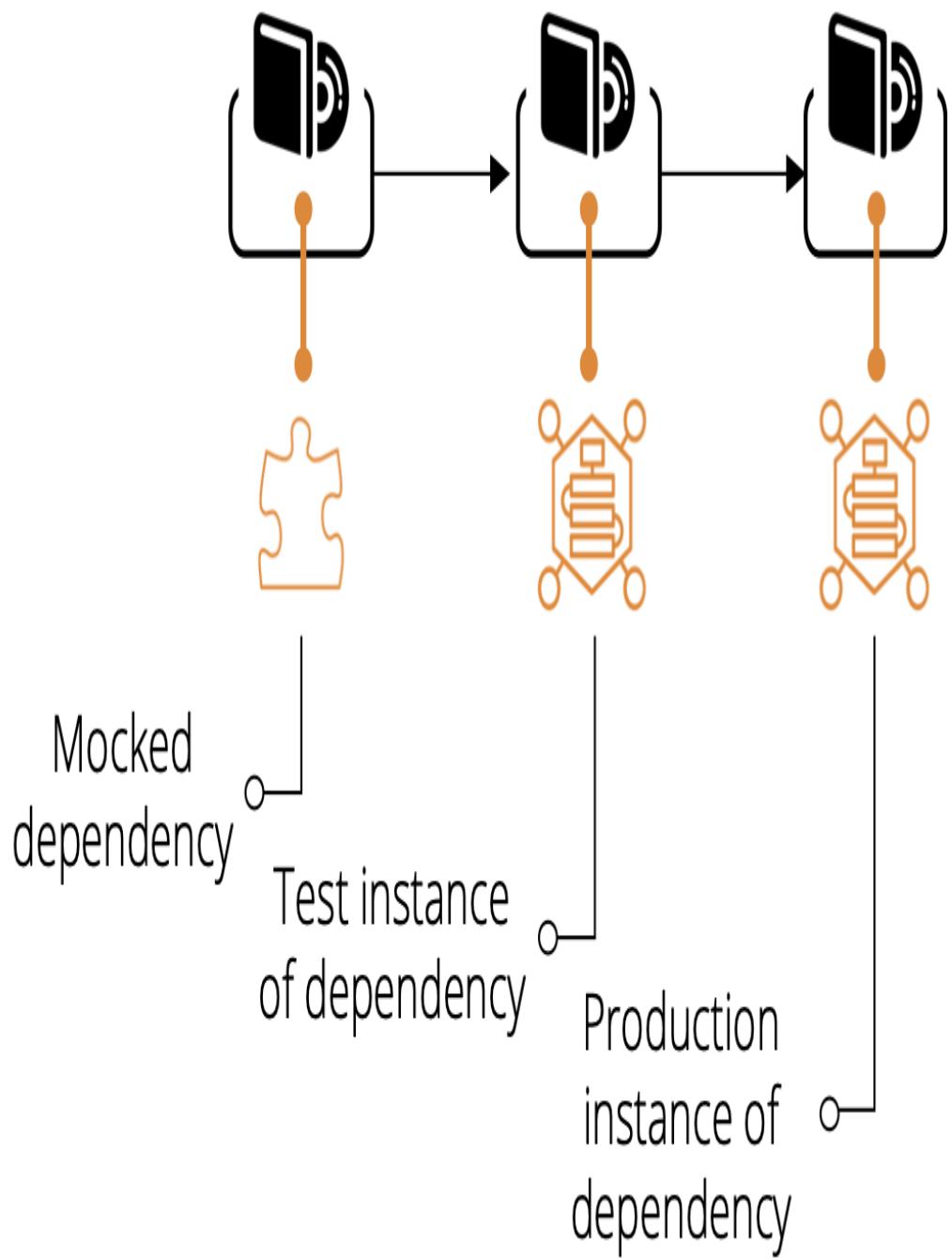


Figure 9-2. Progressive integration with dependencies

Avoid creating unnecessary stages in your pipeline, as each stage adds time and cost to your delivery process. So, don't create separate stages for each component and integration just for completeness. Split validation into stages this way only when it adds enough value to be worth the overhead. Some reasons which

may drive you to do this include speed, reliability, cost, and control.

Provisioning a mock may be much faster than provisioning the external service. Some third -party services are not very reliable, making it unclear whether a test has failed because of your coding error, or a blip with the service. The third party service may be expensive, or it may not give you enough flexibility to set up data and configuration for your tests.

PLATFORM ELEMENTS NEEDED FOR VALIDATION

Platform services are a particular type of dependency for your system. Your system may ultimately run on your infrastructure platform, but you may be able to run and test parts of it offline usefully.

For example, code that defines networking structures needs to provision those structures on the cloud platform for meaningful validation. But you may be able to test code that installs an application server package in a local virtual machine, or even in a container, rather than needing to stand up a virtual machine on your cloud platform.

So earlier validation stages may be able to run without using the full cloud platform for some components:

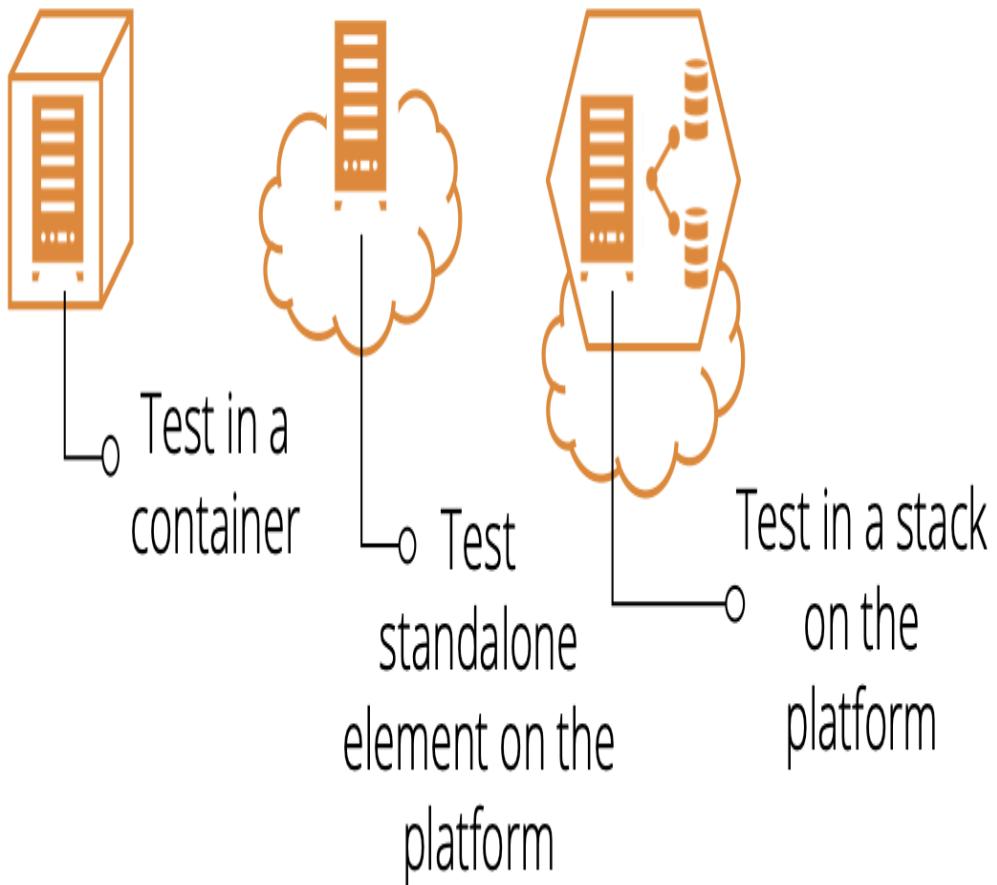
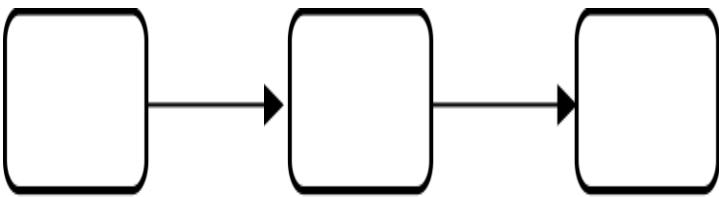


Figure 9-3. Progressive use of platform elements

Testing in production

Testing releases and changes before applying them to production is a big focus in our industry. At one client, I counted eight groups that needed to review and approve releases¹⁰, even apart from the various technical teams who had to carry out tasks to install and configure various parts of the system.

As systems increase in complexity and scale, the scope of risks that you can practically check for outside of production shrinks. This isn't to say that there is no value in testing changes before applying them to production. But believing that pre-release testing can comprehensively cover your risks leads to:

- Over-investing in pre-release testing, well past the point of diminishing returns,
- Under-investing in testing in your production environment.

GOING DEEPER ON TESTING IN PRODUCTION

For more on testing in production, I recommend watching Charity Majors' talk, *[Yes, I Test in Production (And So Should You)]* (<https://www.infoq.com/presentations/testing-production-2018/>), which is a key source of my thinking on this topic.

WHAT YOU CAN'T REPLICATE OUTSIDE PRODUCTION

There are several characteristics of production environments which you can't realistically replicate outside of production:

Data

Your production system may have larger data sets than you can replicate, and will undoubtedly have unexpected data values and combinations, thanks to your users.

Users

Due to their sheer numbers, your users are far more creative at doing strange things than your testing staff.

Traffic

If your system has a non-trivial level of traffic, you can't replicate the number and types of activities it will regularly experience. A week-long soak test is trivial compared to a year of running in production.

Concurrency

Testing tools can emulate multiple users using the system at the same time, but they can't replicate the unusual combinations of things that your users do concurrently.

The two challenges that come from these characteristics are that they create risks that you can't predict, and they create conditions that you can't replicate well enough to test anywhere other than production.

By running tests in production, you take advantage of the conditions that exist there—large natural data sets and unpredictable concurrent activity.

WHY TEST ANYWHERE OTHER THAN PRODUCTION?

Obviously testing in production is not a substitute for testing changes before you apply them to production. It helps to be clear on what you realistically can (and should!) test beforehand:

- Does it work?
- Does my code run?
- Does it fail in ways I can predict?
- Does it fail in the ways it has failed before?

Testing changes before production addresses the *known unknowns*, the things that you know might go wrong. Testing changes in production addresses the *unknown unknowns*, the more unpredictable risks.

MANAGING THE RISKS OF TESTING IN PRODUCTION

Testing in production creates new risks. There are a few things that help manage these risks:

Monitoring

Effective monitoring gives confidence that you can detect problems caused by your tests so you can stop them quickly. This includes detecting when tests are causing issues so you can stop them quickly (see [Link to Come]).

Observability

Observability gives you visibility into what's happening within the system at a level of detail that helps you to investigate and fix problems quickly, as well as improving the quality of what you can test.¹¹

Zero-Downtime Deployment

Being able to deploy and roll back changes quickly and seamlessly helps mitigate the risk of errors (see [Link to Come]).

Progressive Deployment

If you can run different versions of components concurrently, or have different configurations for different sets of users, you can test changes in production conditions before exposing them to users (see [Link to Come]).

Data management

Your production tests shouldn't make inappropriate changes to data or expose sensitive data. You can maintain test data records, such as users and credit card numbers, that won't trigger real-world actions.

MONITORING AS TESTING

Monitoring can be seen as passive testing in production. It's not true testing, in that you aren't taking an action and checking the result. Instead, you're observing the natural activity of your users and watching for undesirable outcomes.

Monitoring should form a part of the testing strategy, because it is a part of the mix of things you do to manage risks to your system.

Progressive validation models

With the different types of things you can validate and the various ways to group tests into stages for progressive validation, a model can help think about how to structure validation and testing activities for changes to your particular system. I'll describe the most popular model for this, the testing pyramid, and an interesting alternative, the Swiss cheese model.

The guiding principle for a progressive feedback strategy is to get fast, accurate feedback. As a rule, this means running faster tests with a narrower scope and fewer dependencies first and then running tests that progressively add more components and integration points. This way, small errors are quickly made visible so they can be quickly fixed and re-tested.

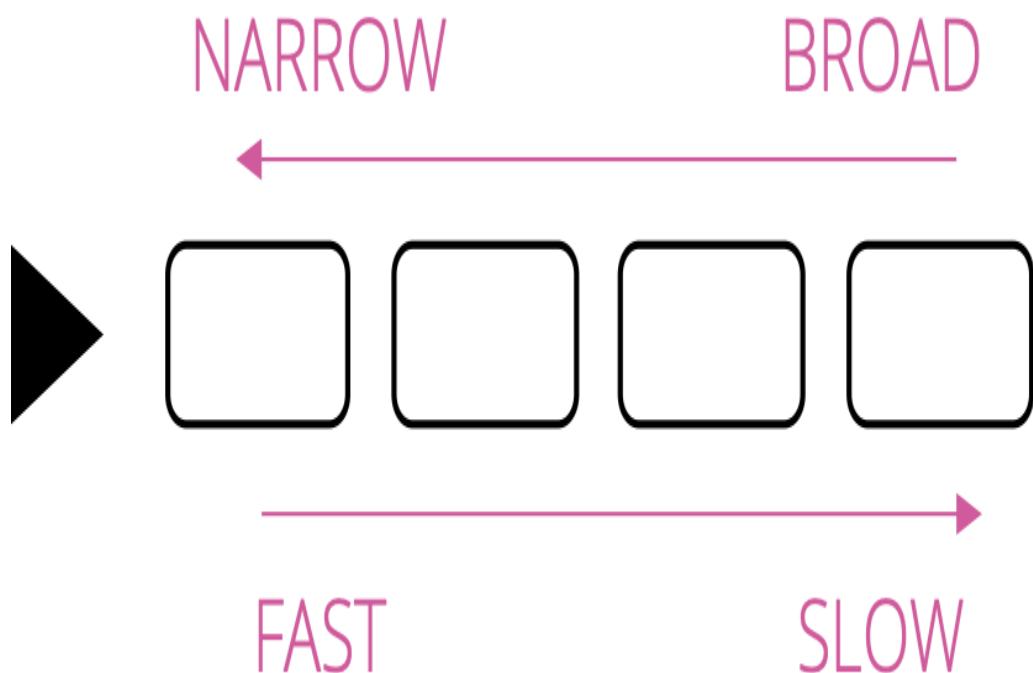


Figure 9-4. Scope vs. speed of progressive testing

When a broadly-scoped test fails, you have a large surface area of components and dependencies to investigate. So you should try to find any potential area at the earliest point, with the smallest scope that you can.

Another goal of a test strategy is to keep the overall test suite manageable. Avoid duplicating validations at different levels. For example, you may test that your application server configuration code sets the correct directory permissions on the log folder. This test would run in an earlier stage, that explicitly tests the server configuration. You should not have a test that checks file permissions in the stage that tests the full infrastructure stack provisioned in the cloud.

Test pyramid

The test pyramid is a well-known model for software testing¹²<https://martinfowler.com/articles/practical-test-pyramid.html> by Ham Vocke is a thorough reference.], so I won't spend much time describing it.

The key idea of the test pyramid is that you should have more tests at the lower layers, which are the earlier stages in your progression. With software, it makes sense to have many unit tests, as these are fast and can catch errors quickly:

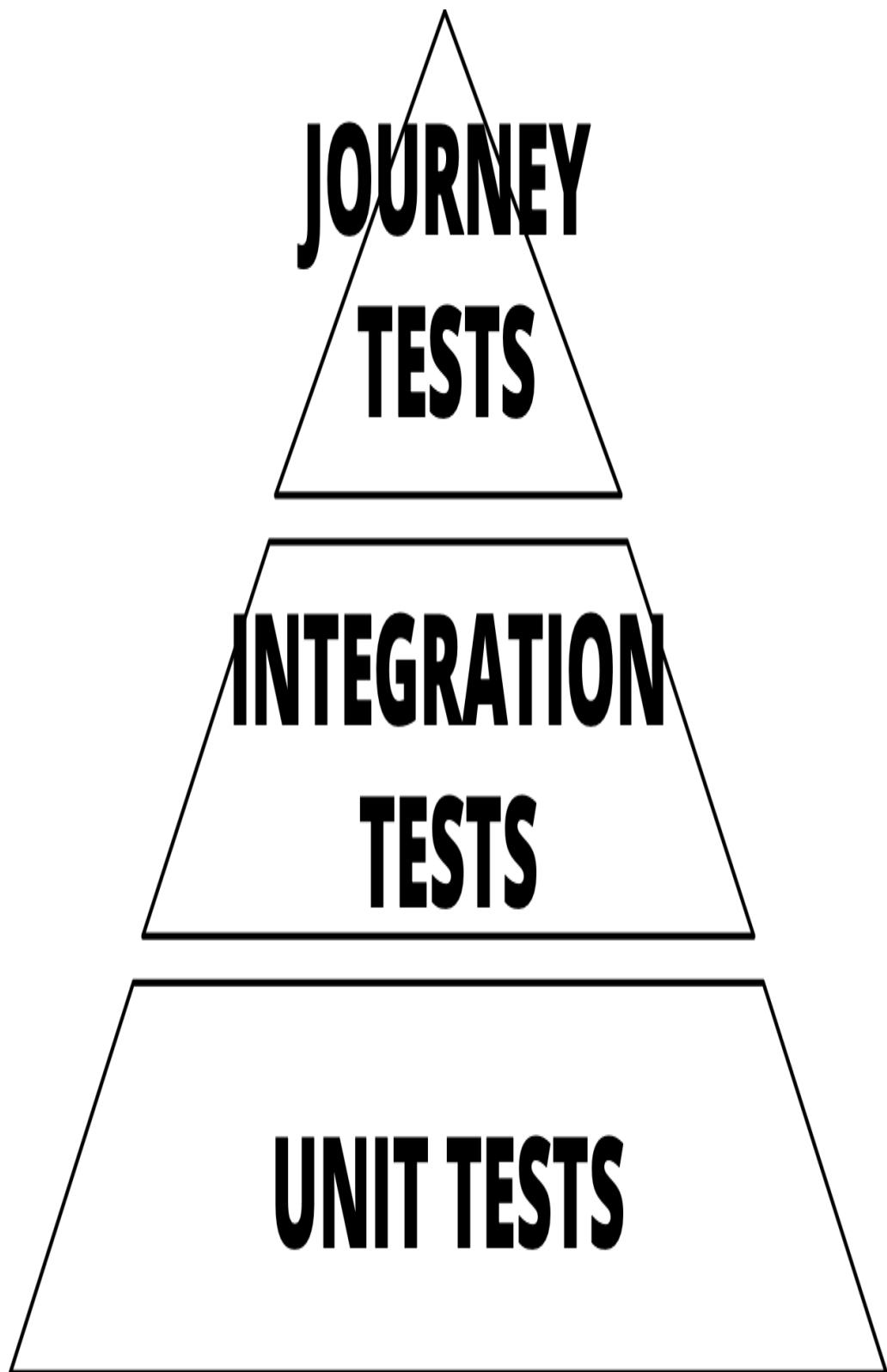


Figure 9-5. The classic test pyramid

But lower-level infrastructure tests tend to have less value (as I discussed in “Challenge: Tests for declarative code often have low value”). This means that, although you’ll almost certainly have

low-level infrastructure tests, there may not be as many as the pyramid model suggests. So an infrastructure test suite may end up looking more like a diamond:

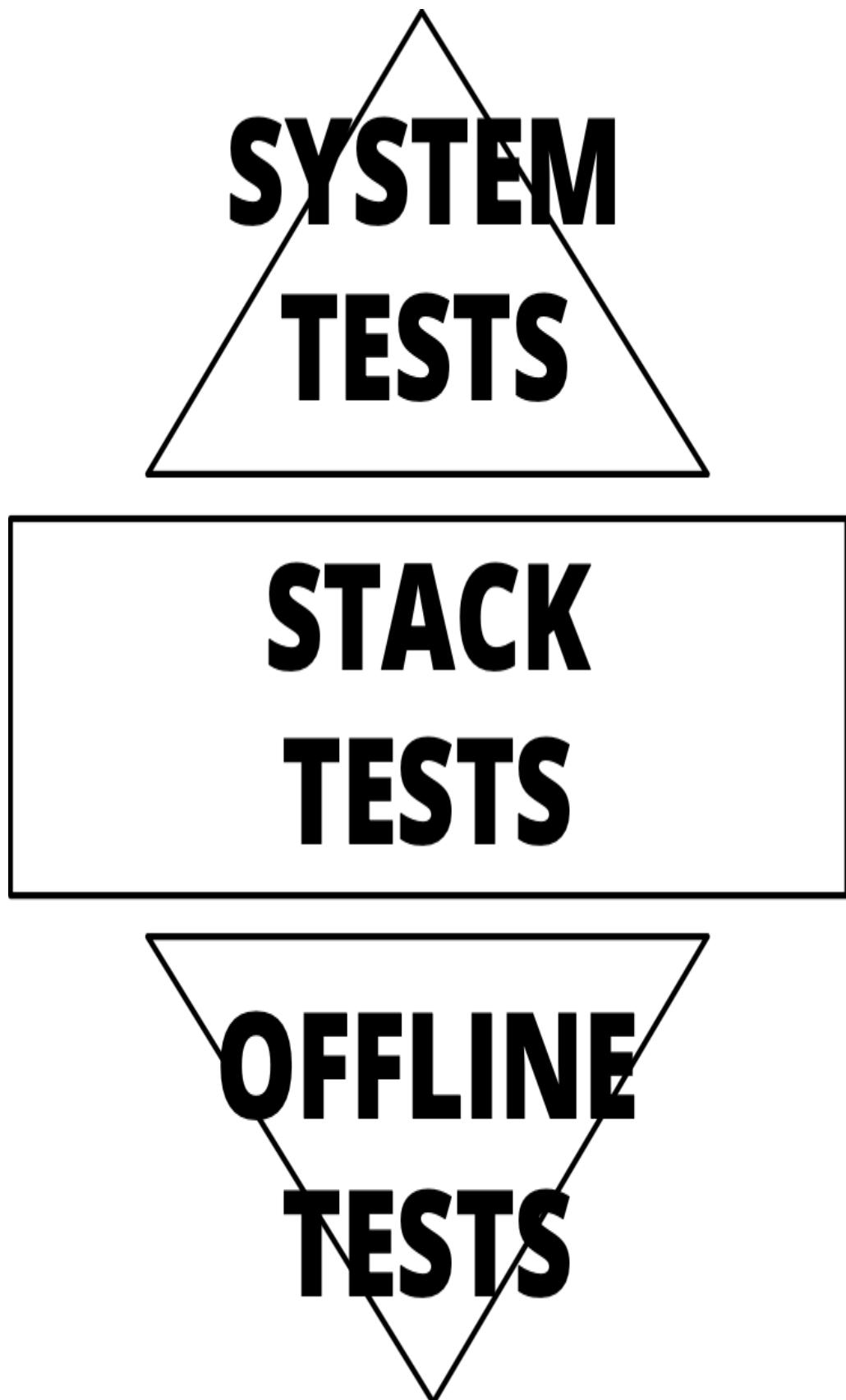


Figure 9-6. The infrastructure test diamond

It can be helpful to keep this in mind when discussing the right balance for a test suite with people who come from a software testing background. They may be uncomfortable with the different balance of tests.

Swiss cheese testing model

Another way to think about how to organize progressive tests is the Swiss cheese model. This concept for risk management comes from outside the software industry¹³. The idea is that a given layer of testing may have holes, like one slice of Swiss cheese, that can miss a defect or risk. But when you combine multiple layers, it looks more like a block of Swiss cheese, where no hole goes all the way through.

The point of using the Swiss cheese model when thinking about infrastructure testing is that you focus on where to catch any given risk. You still want to catch issues in the earliest layer where it is feasible to do so, but the important thing is that it is tested somewhere in the overall model:

OFFLINE TESTS STACK TESTS SYSTEM TESTS ALERTS

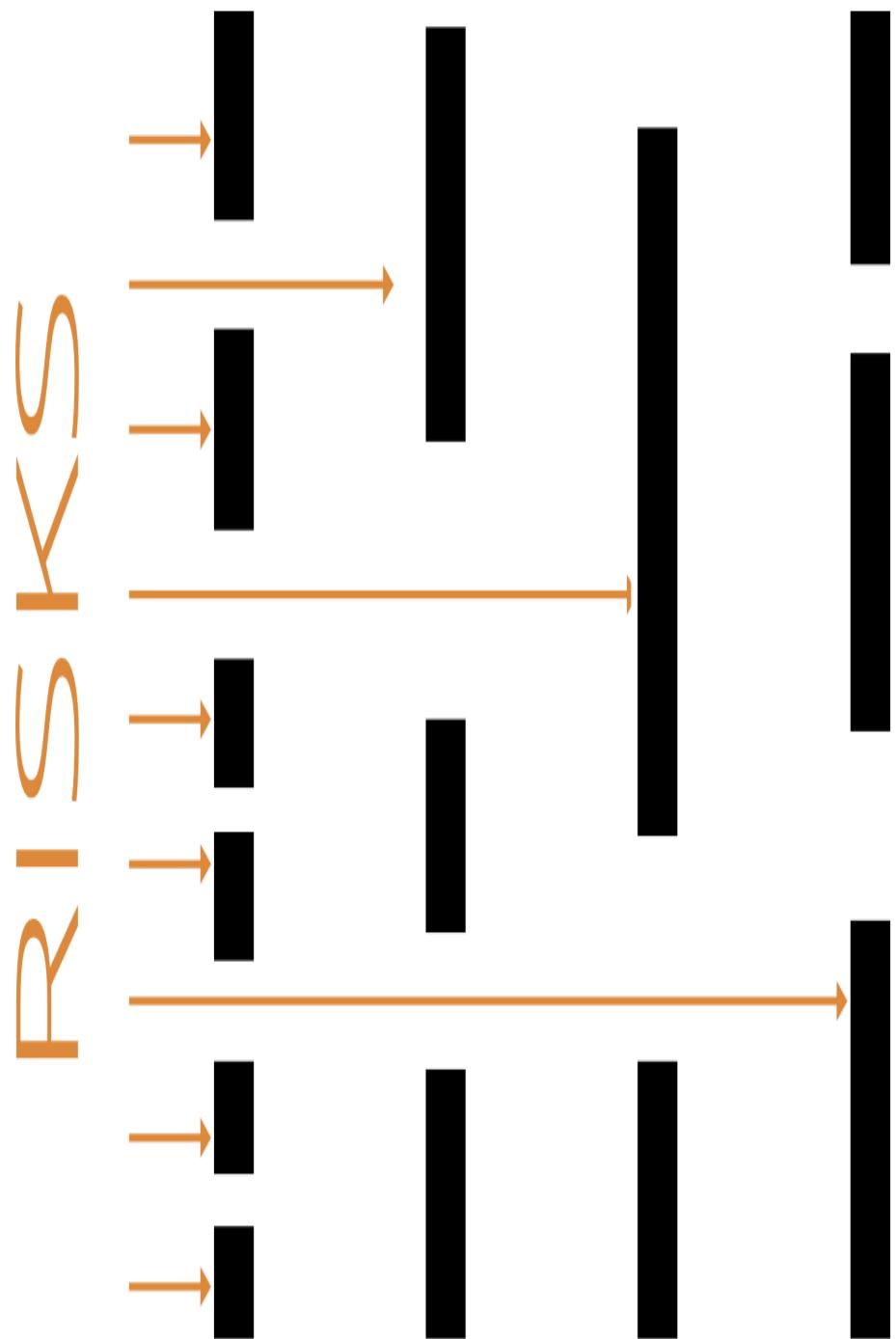


Figure 9-7. Swiss cheese testing model

The key takeaway is, test based on risk, rather than based on fitting a formula.

Pipelines for validation

You can implement progressive validation and delivery of infrastructure elements using a *continuous delivery pipeline*¹⁴. I often call this a *change delivery pipeline* to emphasize its relationship to the change management process.

The idea is that when someone pushes a code change to the source control repository, the team uses a central system to progress the change through a series of stages to test and deliver the change. This process is automated, although people may be involved to trigger or approve activities. Some principles for a change delivery pipeline include:

Automate processes

The pipeline system runs scripts to apply infrastructure code, deploy software, and execute automated tests. Humans may review changes, and even conduct exploratory testing on environments. But they should not be running commands by hand to deploy and apply changes. They also shouldn't be selecting configuration options or making other decisions on the fly. These actions should be defined as code and executed by the system. Automating processes ensures they are carried out consistently every time, for every stage. Doing this improves the reliability of your tests, and creates consistency between instances of the infrastructure.

Push every change from the start of the pipeline

Never change the code once it's progressing through the pipeline. If you find an error in a "downstream" (later) stage in a pipeline, don't fix it in that stage and continue through the rest of the pipeline. Instead, fix the code in the repository and push the new change from the start of the pipeline. This practice ensures that the change is fully tested.

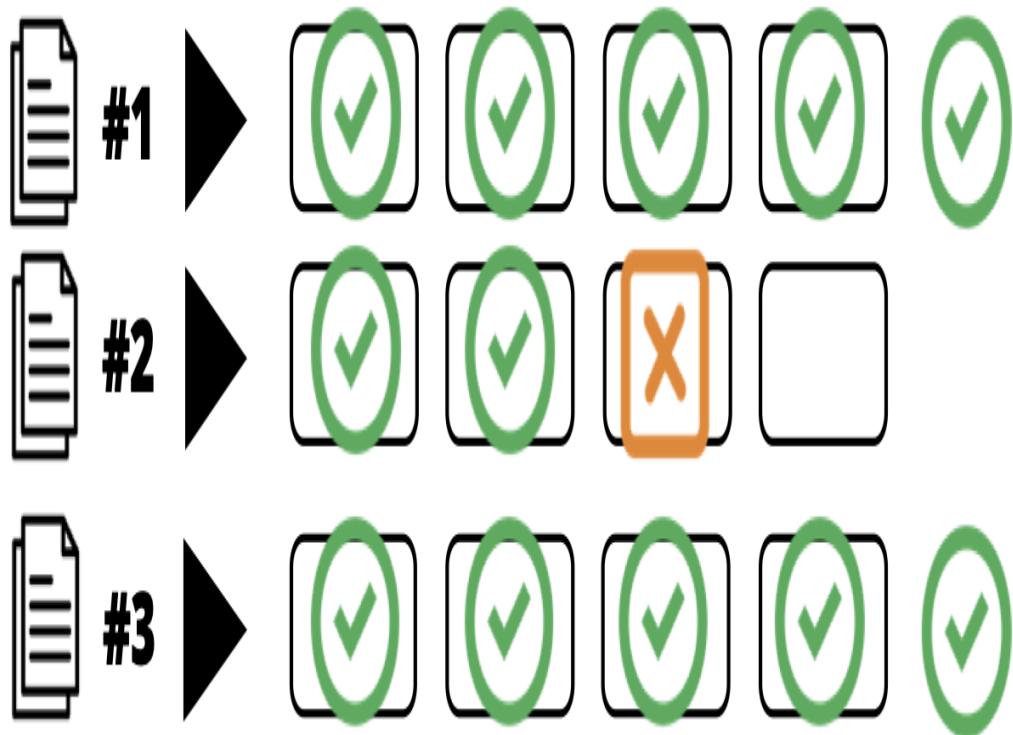


Figure 9-8. Start a new pipeline run to correct failures

In the above figure, one change successfully passes through the pipeline. The second change fails in the middle of the pipeline. A fix is made and pushed through to production as the third pipeline run.

Pipeline stages

Each stage of the pipeline may do different things and may trigger in different ways. Some of the characteristics of a given pipeline stage include:

Trigger

What makes the stage run? It may automatically run when there is an event, such as a change pushed to the code repository or the successful execution of the stage before it in the pipeline. Or someone may trigger the stage manually, as when a tester or release manager decides to apply a code change to a given environment.

Activity

What happens when the stage runs? Multiple actions could execute for a stage. For example, a stage might apply code to provision an infrastructure stack, run tests, and then destroy the stack.

Approval

How is the stage marked as passing or failing? The system could mark the stage as passing (often referred to as “green”) when commands run without errors, and automated tests all pass. Or a human may need to mark the stage as approved. For example, a tester may approve the stage after carrying out exploratory testing on the change. You can also use manual approval stages to support governance sign-offs.

Output

Does the stage produce an artifact or other material? Typical outputs include archive files containing the infrastructure code and test results.

Delivery pipeline software and services

You need software or a hosted service to build a pipeline. A pipeline system needs to do a few things:

- Give you a way to configure the pipeline stages.
- Trigger stages from different actions, including automated events and manual triggers. The tool should support more complex relationships such as *fanning in* (one stage with

multiple input stages) and *fanning out* (one stage with multiple output stages).

- Support any actions you may need for your stages, including applying infrastructure code and running tests. You should be able to create custom activities rather than having a fixed set of supported ones.
- Handle artifacts and other outputs of stages, including being able to pass them from one stage to the next.
- Help you trace and correlate specific versions and instances of code, artifacts, outputs, and infrastructure.

There are a few options for a pipeline system:

CI Software

Many teams use Continuous Integration software such as [Jenkins](#), [Team City](#), and [Bamboo](#), to create pipelines. These are often “job-oriented” rather than “stream-oriented.” The core design doesn’t inherently correlate versions of code, artifacts, and runs of different jobs. Most of these products have added support for pipelines as an overlay in their UI and configuration.

CD Software

CD software is built around the pipeline concept. You define each stage as part of a pipeline, and code versions and artifacts are associated with the pipeline so you can trace them forwards and backward. CD tools include [GoCD](#)¹⁵, [ConcourseCI](#)¹⁶, and [BuildKite](#).

SaaS Services

Hosted CI and CD services include [CircleCI](#), [TravisCI](#), [AppVeyor](#), [Drone](#), and [BoxFuse](#).

Cloud Platform Services

Most cloud vendors include CI and CD services, including [AWS CodeBuild](#) (CI) and [AWS CodePipeline](#) (CD), and [Azure DevOps](#)¹⁷

Source Code Repository Services

Many source code repository products and vendors have added CI support that you can use to create pipelines. Two prominent examples are [Github actions](#), and [Gitlab CI and CD](#).

The products I mentioned above were all designed with application software in mind. You can use most of them¹⁸ to build pipelines for infrastructure, although they may need extra work.

A few products and services designed for infrastructure as code are emerging as I write this. This is a rapidly changing area, so I suspect what I have to say about these tools is out of date by the time you read this, and missing newer tools. But it's worth looking at what exists now, to give context for evaluating tools as they emerge and evolve:

- [Atlantis](#) is a product that helps you to manage pull requests for Terraform projects, and to run `plan` and `apply` for a single instance. It doesn't run tests, but you can use it to create a limited pipeline that handles code reviews and approvals for infrastructure changes.
- [Terraform Cloud](#) is evolving rapidly. It is Terraform-specific, and it includes more features (such as a module registry) than CI and pipelines. You can use Terraform cloud to create a limited pipeline that plans and applies a project's code to multiple environments. But it doesn't run tests other than policy validations with Hashicorp's own [Sentinel](#) product.
- [WeaveWorks](#) makes products and services for managing Kubernetes clusters. These include tools for managing the

delivery of changes to cluster configuration as well as applications using pipelines based around git branches, an approach they call [GitOps](#). Their solutions don't seem applicable to general infrastructure, but even if you don't use their stack, it's an emerging model for pipelines that's worth watching. I'll touch on it a bit more in [Link to Come].

The next chapter ([Chapter 10](#)) includes details on how to structure pipelines for testing and delivering infrastructure stacks. I also discuss pipeline implementation for building servers ([Link to Come]) and clusters ([Link to Come]).

Conclusion

This chapter has discussed general challenges and approaches for testing infrastructure. I've avoided going very deeply into the subjects of testing, quality, and risk management. If these aren't areas you have much experience with, this chapter may give you enough to get started. I encourage you to read more, as testing and QA are fundamental to infrastructure as code.

While I've touched on aspects of testing that are specific to infrastructure, such as the challenges of writing tests for infrastructure, I haven't given many concrete examples. The next chapter, [Chapter 10](#), should do this for infrastructure stacks. Likewise, [Link to Come] explains how to write tests and build pipelines for servers, as [Link to Come] does for clusters.

¹ I listed these three core practices in [Chapter 1](#)

² See <https://martinfowler.com/articles/continuousIntegration.html>

- 3 Jez Humble and David Farley's book *Continuous Delivery* (Addison-Wesley) defined the principles and practices for CD, raising it from an obscure phrase in the Agile Manifesto to a widespread practice among software delivery teams.
- 4 See <https://www.mountaingoatsoftware.com/agile/user-stories> for an explanation of agile stories.
- 5 The *Accelerate* research published in the annual *State of the DevOps Report* finds that teams where everyone merges their code at least daily tend to be more effective than those who do so less often. In the most effective teams I've seen, developers push their code multiple times a day, sometimes as often as every hour or so.
- 6 <https://continuousdelivery.com/>
- 7 Martin Fowler's biki *Mocks Aren't Stubs* is a useful reference for test doubles.
- 8 Examples of cloud mocking tools and libraries include Localstack and moto. <https://dobetterascode.com/types/mock/> maintains a current list of this kind of tool.
- 9 See the extremeprogramming.org definition of unit tests. Martin Fowler's biki definition of *UnitTest* discusses a few ways of thinking of unit tests.
- 10 These groups were: change management, infosec, risk management, service management, transition management, system integration testing, user acceptance, and the technical governance board.
- 11 Although it's often conflated with monitoring, observability is about giving people ways to understand what's going on inside your system. See [honeycomb.io's Introduction to Observability](https://honeycomb.io/introduction-to-observability).
- 12 [The Practical Test Pyramid
- 13 See https://en.wikipedia.org/wiki/Swiss_cheese_model
- 14 Sam Newman described the concept of build pipelines in several blog posts starting in 2005, which he recaps in a 2009 blog post, *A Brief and Incomplete History of Build Pipelines*. Jez Humble and Dave Farley's *Continuous Delivery* book (referenced earlier in this chapter) popularized pipelines. Jez has documented the *deployment pipeline pattern* on his website.
- 15 In the interests of full disclosure, my employer, ThoughtWorks created GoCD. It was previously a commercial product, but it is now fully open source.
- 16 In spite of its name, ConcourseCI is designed around pipelines rather than CI jobs.
- 17 I can't mention Azure DevOps without pointing out the terribleness of that name for a service. DevOps is about culture first, not tools and technology first. Read Matt Skelton's review of John Willis' talk on *DevOps culture*.

18 Although some of the products I listed are designed for building container images, so may be harder to adapt for infrastructure.

Chapter 10. Testing Infrastructure Stacks

[Chapter 5](#) describes an infrastructure stack as a collection of infrastructure resources that you define, provision, and update as a unit from an infrastructure platform. I cover general topics for testing infrastructure in [Chapter 9](#). In this chapter, I combine these two topics, delving into specific techniques for testing infrastructure code at the level of the stack.

I use the Foodspin example¹ to illustrate how to test a stack. I start by outlining the example infrastructure involved, to set the context for specific examples later in the chapter.

I then give examples of types of testing activities, grouped into offline validation, which you can run without actually provisioning infrastructure, and online validation which does provision infrastructure on your platform. I give examples use pseudocode for both stack definitions and tests.

You can use test fixtures, such as mocks and fakes, to help you test your stacks as standalone entities rather than needing to provision large swathes of infrastructure for each test.

In addition to test fixtures, you need to provision instances of your stack for online validation. I describe several patterns and one antipattern for managing the lifecycle of these test instances.

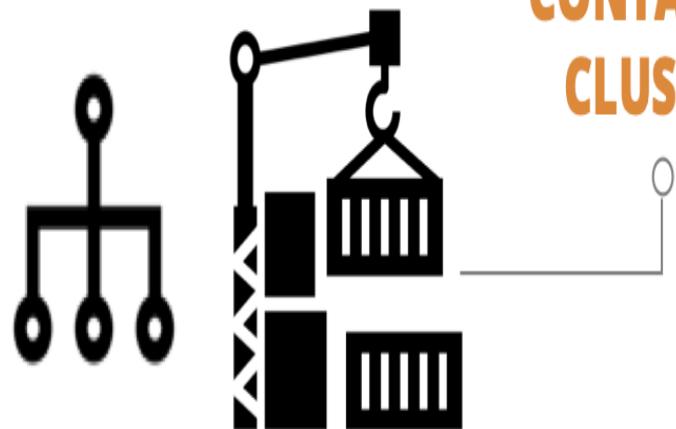
To wrap this up, you need a way to orchestrate test instances and fixtures for running tests. I describe how people use scripts and tools for this, with some pseudocode examples.

Example infrastructure

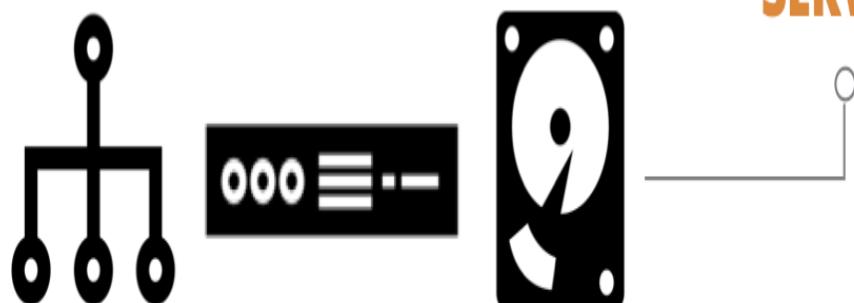
The Foodspin team uses reusable stack projects (“[Pattern: Reusable Stack](#)”) to create consistent instances of application infrastructure for each of their customers. They can also use this to create test instances of the infrastructure in the pipeline.

The infrastructure for these examples includes the following elements (based on the examples from [Chapter 3](#)):

WEB SERVER CONTAINER CLUSTER



APPLICATION SERVER



DATABASE



Figure 10-1. Example infrastructure used for a Foodspin application instance

Web Server Cluster

The team runs a single web server container cluster for each region and in each test environment. Applications in the region or environment share this cluster. The examples in this chapter focus on the infrastructure that is specific to each customer, rather than shared infrastructure. So the shared cluster is a dependency in the examples here. For details of how changes are coordinated and tested across this infrastructure, see [Link to Come].

Application Server

The infrastructure for each application instance includes a virtual machine, a persistent disk volume, and networking. The networking includes an address block, gateway, routes to the server on its network port, and network access rules.

Database Server

Foodspin runs a separate database instance for each customer application instance, using their provider's DBaaS² service. Their infrastructure code also defines an address block, routing, and database authentication and access rules.

The example stack

To start, we can define a single reusable stack that has all of the infrastructure other than the web server cluster. The project structure could look like this:

Example 10-1. Stack project for Foodspin customer application

```
stack-project/
  └── src/
    ├── appserver_vm.infra
    ├── appserver_networking.infra
    ├── database.infra
    └── database_networking.infra
```

Within this project, the file `appserver_vm.infra` would include code along these lines:

Example 10-2. Partial contents of appserver_vm.infra

```
virtual_machine:  
    name: appserver-${customer}-${environment}  
    ram: 4GB  
    address_block: ADDRESS_BLOCK.appserver-${customer}-${environment}  
    storage_volume: STORAGE_VOLUME.app-storage-${customer}-${environment}  
    base_image: SERVER_IMAGE.foodspin_java_server_image  
    provision:  
        tool: servermaker  
        parameters:  
            maker_server: maker.foodspin.io  
            role: appserver  
            customer: ${customer}  
            environment: ${environment}  
  
storage_volume:  
    id: app-storage-${customer}-${environment}  
    size: 80GB  
    format: xfs
```

A team member or automated process can create or update an instance of the stack by running the `stack` tool. They pass values to the instance using one of the patterns from [Chapter 8](#).

As described in [Chapter 9](#), the team uses multiple validation stages (“Progressive validation”), organized in a sequential pipeline (“Pipelines for validation”).

Pipeline for the example stack

A simple pipeline³ for the Foodspin application infrastructure stack has two testing stages, followed by a stage that applies the code to each customer’s production environment:

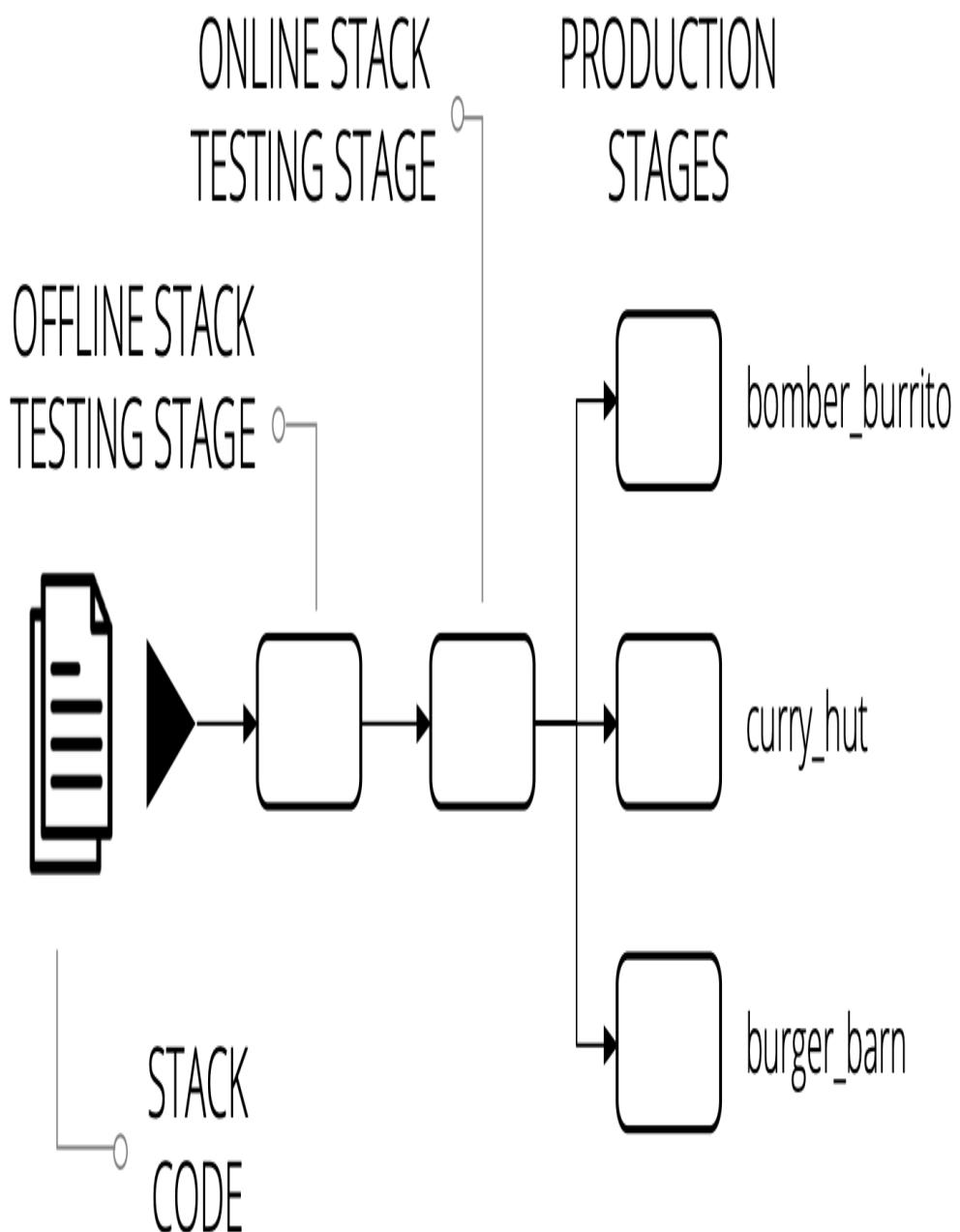


Figure 10-2. Simplified example pipeline for a stack

The first stage of the pipeline is an offline stage, and the second is an online stage. Each of these stages can run several different validation activities.

Offline validation stages for stacks

An *offline stage* runs “locally” on an agent node of the service that runs the stage (see “Delivery pipeline software and services”), rather than needing to provision infrastructure on your infrastructure platform. Strict offline validation runs entirely within the local server or container instance, without connecting to any external services such as a database. A softer offline stage might connect to an existing service instance, perhaps even a cloud API, but doesn’t use any real stack infrastructure.

An offline stage should:

- Run quickly, giving fast feedback if something is incorrect,
- Validate the correctness of components in isolation, to give confidence in each component, and to simplify debugging failures,
- Prove the component is cleanly decoupled.

Some of the things you can check for your stack code in an offline stage are syntax checking, offline static code analysis, static code analysis with the platform API, and testing with a mock API.

Syntax checking

With most stack tools, you can run a *dry run* command that parses your code without applying it to infrastructure. The command exits with an error if there is a syntax error. The check tells you very quickly when you’ve made a typo in your code change, but misses many other errors. Examples of syntax checking include `terraform validate` and `aws cloudformation validate-template`.

The output of a failing syntax validation might look like this:

```
$ stack validate  
  
Error: Invalid resource type  
  
  on appserver_vm.infra line 1, in resource "virtual_mahcine":  
  
    stack does not support resource type "virtual_mahcine".
```

Offline static code analysis

Some tools can parse and analyze stack source code for a wider class of issues than just syntax, but still without connecting to an infrastructure platform. This analysis is often called linting⁴—comes from a classic Unix utility that analyzes C source code]. This kind of tool may look for coding errors, confusing or poor coding style, adherence to code style policy, or potential security issues. Some tools can even modify code to match a certain style, such as the `terraform fmt` command. There are not as many tools that can analyze infrastructure code as there are for application programming languages. Examples include `tflint`, `CloudFormation Linter`, and `cfn_nag`.

Here's an example of an error from a fictional analysis tool:

```
$ stacklint  
1 issue(s) found:  
  
Notice: Missing 'Name' tag (vms_must_have_standard_tags)  
  
  on appserver_vm.infra line 1, in resource "virtual_machine":
```

In this example, we have a custom rule named `vms_must_have_standard_tags` that requires all virtual machines to have a set of tags, including one called `Name`.

Static code analysis with API

Depending on the tool, some static code analysis checks may connect to the cloud platform API to check for conflicts with what the platform supports. For example, tflint can check Terraform project code to make sure that any instance types (virtual machine sizes) or AMIs (server images) defined in the code actually exist. Unlike previewing changes (“Preview: Seeing what changes will be made”), this type of validation tests the code in general, rather than against a specific stack instance on the platform.

The following example output fails because the code declaring the virtual server specifies a sever image that doesn’t exist on the platform:

```
$ stacklint
1 issue(s) found:

Notice: base_image 'SERVER_IMAGE.foodspin_java_server_image' doesn't
exist (validate_server_images)

on appserver_vm.infra line 5, in resource "virtual_machine":
```

Testing with mock API

You may be able to apply your stack code to a local, mock instance of your infrastructure platform’s API. There are not many tools for mocking these APIs. The only one I’m aware of as of this writing is Localstack. Applying your stack code to a local mock can reveal coding errors that syntax or code analysis checks might not find. Depending on how functional the mocks are, you may be able to carry out some online-type validations (as I describe shortly). But in practice, what you can do is limited.

Online validation stages for stacks

An *online stage* involves using the infrastructure platform to create and interact with an instance of the stack. This type of stage is slower but can carry out more meaningful validations than online validation. The delivery pipeline service usually runs the stack tool on one of its nodes or agents, but it uses the platform API to interact with an instance of the stack. The service needs to authenticate to the platform’s API, see “[Secrets and source code](#)” for ideas on how to handle this securely.

Although an online test stage depends on the infrastructure platform, you should be able to test the stack with a minimum of other dependencies. In particular, you should design your infrastructure, stacks, and tests so that you can create and test an instance of a stack without needing to integrate with instances of other stacks.

For example, the Foodspin customer application infrastructure works with a shared web server cluster stack. However, they implement their infrastructure, and testing stages, so that they can test the application stack code without an instance of the web server cluster.

I cover techniques for splitting stacks and keeping them loosely coupled in [Link to Come]. Assuming you have built your infrastructure in this way, you can use test fixtures to make it possible to test a stack on its own, as described a bit later in this chapter (“[Using test fixtures to handle dependencies](#)”).

First, consider how different types of online stack tests work. The validations that an online stage can run include previewing

changes, verifying that changes are applied correctly, and proving the outcomes.

Preview: Seeing what changes will be made

Some stack tools can compare stack code against a stack instance to list changes it would make without actually changing anything. Terraform's *plan* subcommand is a well-known example.

Most often, people preview changes against production instances as a safety measure, so someone can review the list of changes to reassure themselves that nothing unexpected will happen.

Applying changes to a stack can be done with a two-step process in a pipeline. The first step runs the preview, and a person triggers the second step, to apply the changes, once they've reviewed the results of the preview.

Having people review changes isn't very reliable. People might misunderstand or not notice a problematic change. You can write automated tests that check the output of a preview command. This kind of test might check changes against policies, failing if the code creates a deprecated resource type, for example. Or it might check for disruptive changes-fail if the code will rebuild or destroy a database instance.

Another issue is that stack tool previews are usually not deep. A preview tells you that this code will create a new server:

```
virtual_machine:  
  name: myappserver  
  base_image: "java_server_master"
```

But the preview may not tell you that "java_server_master" doesn't exist, although the `apply` command will fail to create the server.

Previewing stack changes is useful for checking a limited set of risks immediately before applying a code change to an instance. But it is less useful for testing code that you intend to reuse across multiple instances, such as across test environments for release delivery. Teams using copy-paste environments ("Antipattern: [Copy-Paste Environments](#)") often use a preview stage as a minimal validation for each environment. But teams using reusable stacks ("Pattern: Reusable Stack") can use test instances for more meaningful validation of their code.

Verification: Making assertions about infrastructure resources

Given a stack instance, you can have tests in an online stage that make assertions about the infrastructure in the stack. Testing frameworks such as Awspec⁵, Inspec⁶, and Terratest⁷ use the infrastructure platform's API, making requests to gather information about infrastructure and then making assertions about it.

A set of tests for the virtual machine from the example stack code earlier in this chapter could look like this:

```
given virtual_machine(name: "appserver-testcustomerA-staging") {
  it { exists }
  it { is_running }
  it { passes_healthcheck }
  it { has_attached storage_volume(name: "app-storage-testcustomerA-
staging") }
}
```

Most stack testing tools provide libraries to help write assertions about the types of infrastructure resources I describe in [Chapter 3](#). This example test uses a `virtual_machine` resource to identify the VM in the stack instance for the staging environment. It makes several assertions about the resource, including whether it has been created (`exists`), whether it's running rather than having terminated (`is_running`), and whether the infrastructure platform considers it healthy (`passes_healthcheck`).

Simple assertions often have low value (see [“Challenge: Tests for declarative code often have low value”](#)), since they simply restate the infrastructure code they are testing. A few basic assertions (such as `exists`) help to sanity check that the code was applied successfully. These quickly identify basic problems with pipeline stage configuration or test setup scripts. Tests such as `is_running` and `passes_healthcheck` would tell you when the stack tool successfully creates the VM, but it crashes or has some other fundamental issue. Simple assertions like these save you time in troubleshooting.

Although you can create assertions that reflect each of the VM's configuration items in the stack code, like the amount of RAM or the network address assigned to it, these have little value and add overhead.

The fourth assertion in the example, `has_attached storage_volume()` is more interesting. The assertion checks that the storage volume defined in the same stack is attached to the VM. Doing this validates that the combination of multiple declarations works correctly (as discussed in [“Testing combinations of declarative code”](#)). Depending on your platform

and tooling, the stack code might apply successfully but leave the server and volume correctly tied together. Or you might make an error in your stack code that breaks the attachment.

Another case where assertions can be useful is when the stack code is dynamic. When passing different parameters to a stack can create different results, you may want to make assertions about those results. As an example, this code creates the infrastructure for an application server that is either public facing or internally facing:

```
virtual_machine:  
  name: appserver-${customer}-${environment}  
  address_block:  
    if(${network_access} == "public")  
      ADDRESS_BLOCK.public-${customer}-${environment}  
    else  
      ADDRESS_BLOCK.internal-${customer}-${environment}  
    end
```

You could have a testing stage that creates each type of instance and asserts that the networking configuration is correct in each case. You should move more complex variations into modules or libraries (see [Chapter 6](#)) and test those modules separately from the stack code. Doing this simplifies testing the stack code.

Asserting that infrastructure resources are created as expected is useful up to a point. But the most valuable testing is proving that they do what they should.

Outcomes: Proving infrastructure works correctly

Functional testing is an essential part of testing application software. The analogy with infrastructure is proving that you can

use the infrastructure as intended. Examples of outcomes you could test with infrastructure stack code include:

- Can you make a network connection from the web server networking segment to an application hosting network segment on the relevant port?
- Can you deploy and run an application on an instance of your container cluster stack?
- Can you safely reattach a storage volume when you rebuild a server instance?
- Does your load balancer correctly handle server instances as they are added and removed?

Testing outcomes is more complicated than verifying that things exist. Not only do your tests need to create or update the stack instance, as I discuss next (“Lifecycle patterns for test instances of stacks”), you may also need to provision test fixtures. A **test fixture** is an infrastructure resource that is used only to support a test (I talk about test fixtures in “Using test fixtures to handle dependencies”).

This test makes a connection to the server to check that the port is reachable, and returns the expected HTTP response:

```
given stack_instance(stack: "foodspin_networking",
                     instance: "online_test") {

    can_connect(ip_address: stack_instance.appserver_ip_address,
                port:443)

    http_request(ip_address: stack_instance.appserver_ip_address,
                 port:443,
                 url: '/').response.code is('200')

}
```

The testing framework and libraries implement the details of validations like `can_connect` and `http_request`. You'll need to read the documentation for your test tool to see how to write actual tests.

Using test fixtures to handle dependencies

Many stack projects depend on resources created outside the stack, such as shared networking defined in a different stack project. A test fixture is an infrastructure resource that you create specifically to help you provision and test a stack instance by itself, without needing to have instances of other stacks.

Using test fixtures makes it much easier to manage tests, keep your stacks loosely coupled, and have fast feedback loops. Without test fixtures, you may need to create and maintain complicated sets of test infrastructure.

A test fixture is not a part of the stack that you are testing. It is additional infrastructure that you create to support your tests. You use test fixtures to represent a stack's dependencies.

A given dependency is either **upstream**, meaning the stack you're testing uses resources provided by another stack, or it is **downstream**, in which case other stacks use resources from the stack you're testing. People sometimes call a stack with downstream dependencies the **provider**, since it provides resources. A stack with upstream dependencies is then called the **consumer**.

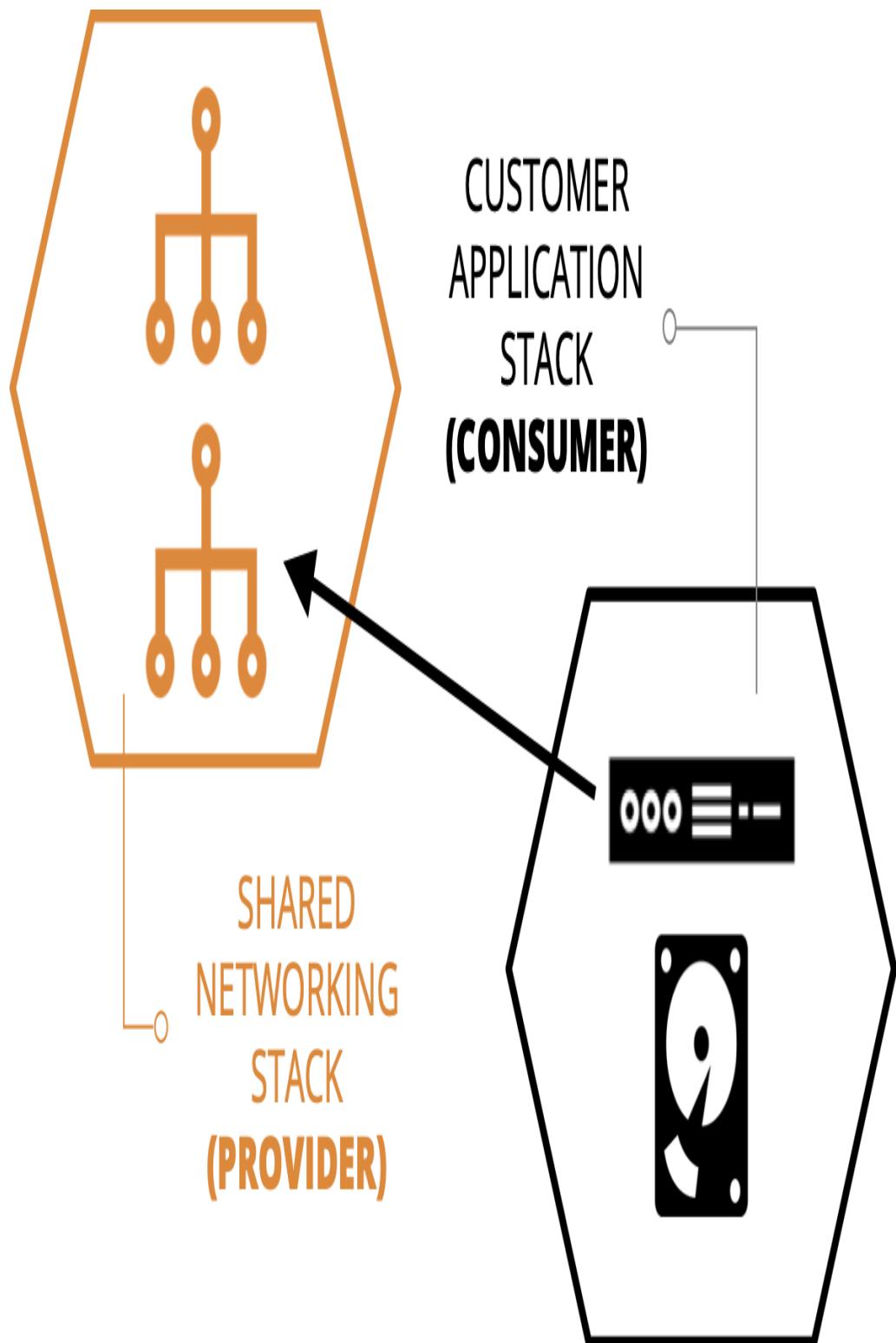


Figure 10-3. Example of a provider stack and consumer stack

Our Foodspin example has a provider stack which defines shared networking structures. These structures are used by consumer stacks, including the stack that defines customer application

infrastructure. The application stack creates a server that it assigns to a network address block created by the networking stack⁸.

A given stack may be both a provider and a consumer, consuming resources from another stack and providing resources to other stacks.

NEVER CREATE CIRCULAR DEPENDENCIES

Although a stack may have both upstream and downstream dependencies, you should never have circular dependencies. A circular dependency exists when a stack has both upstream and downstream dependencies to another stack, or when there is a loop of dependencies between a group of stacks. It should always be possible to follow dependencies from one stack to another in a group, without returning to a previous stack.

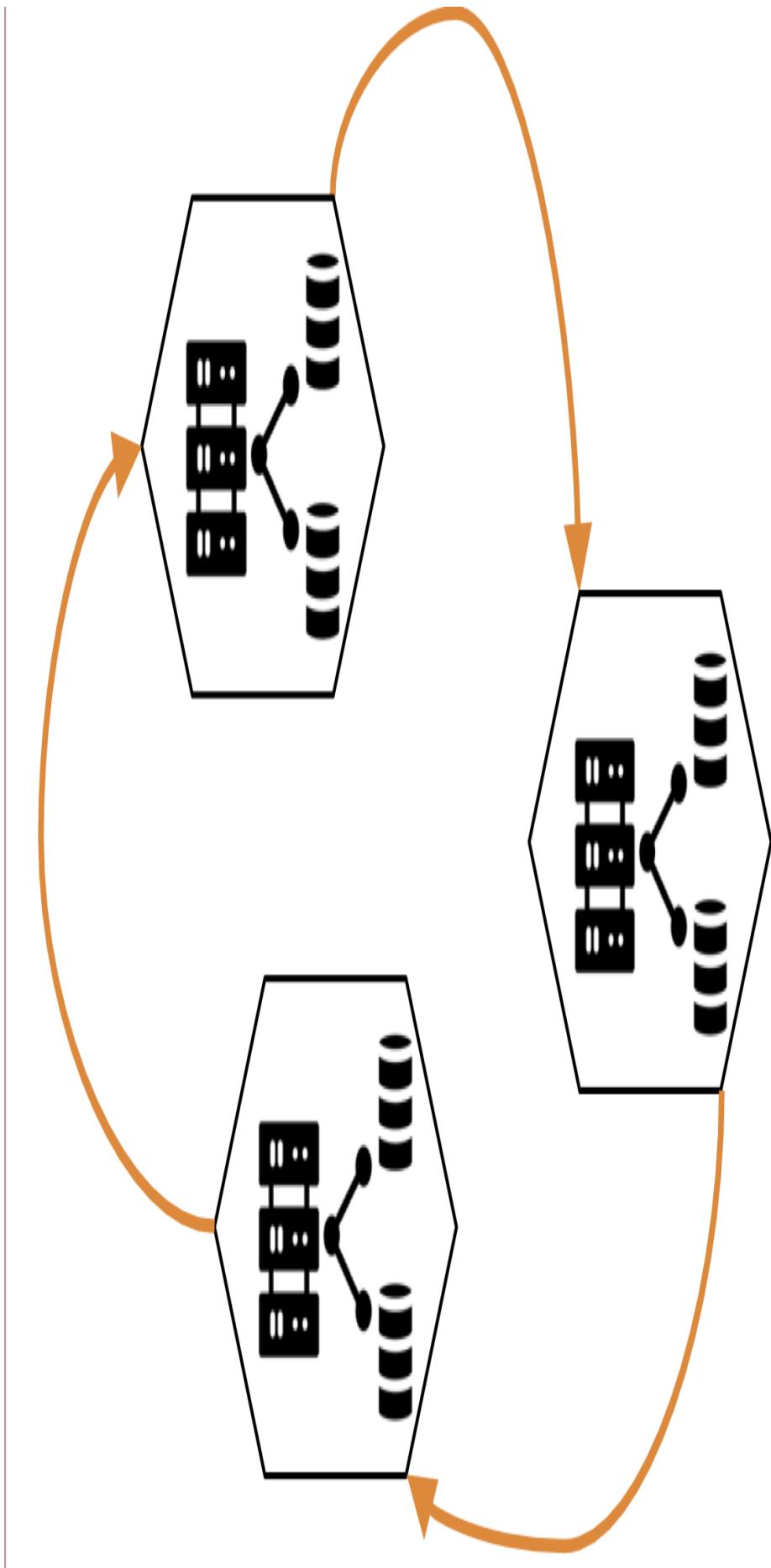


FIGURE 10-4. NEVER CREATE STACKS WITH CIRCULAR DEPENDENCIES

If you find you do have a circular dependency in your system, you should redesign your stacks to remove it.

You can use test fixtures to stand in for either upstream or downstream integration points of a stack.

Test doubles for upstream dependencies

When you need to test a stack that depends on another stack, you can create a test double (see “[Test Doubles](#)”). For stacks, this typically means creating some additional infrastructure. In our example of the shared network stack and the application stack, the application stack needs to create its server in a network address block that is defined by the network stack. Your test setup may be able to create an address block as a test fixture to test the application stack on its own.

It may be better to create the address block as a test fixture rather than creating an instance of the entire network stack. The network stack may include extra infrastructure that isn’t necessary for testing. For instance, it may define network policies, routes, auditing, and other resources for production that are overkill for a test.

Also, creating the dependency as a test fixture within the consumer stack project decouples it from the provider stack. If someone is working on a change to the networking stack project, it doesn’t impact work on the application stack.

A potential benefit of this type of decoupling is to make stacks more reusable and composable. The Foodspin team might want to create different network stack projects for different purposes. One stack creates tightly controlled and audited networking for services that have stricter compliance needs, such as payment processing subject to the [PCI standard]<https://www.pcisecuritystandards.org/>. Another stack creates networking that doesn't need to be PCI compliant. By testing application stacks without using either of these stacks, the team makes it easier to use the stack code with either one.

Test fixtures for downstream dependencies

You can also use test fixtures for the reverse situation, to test a stack that provides resources for other stacks to use. In Figure 10-5, the stack instance defines networking structures for Foodspin, including segments and routing for the web server container cluster and application servers. The network stack doesn't provision the container cluster or application servers, so to test the networking, the setup provisions a test fixture in each of these segments.

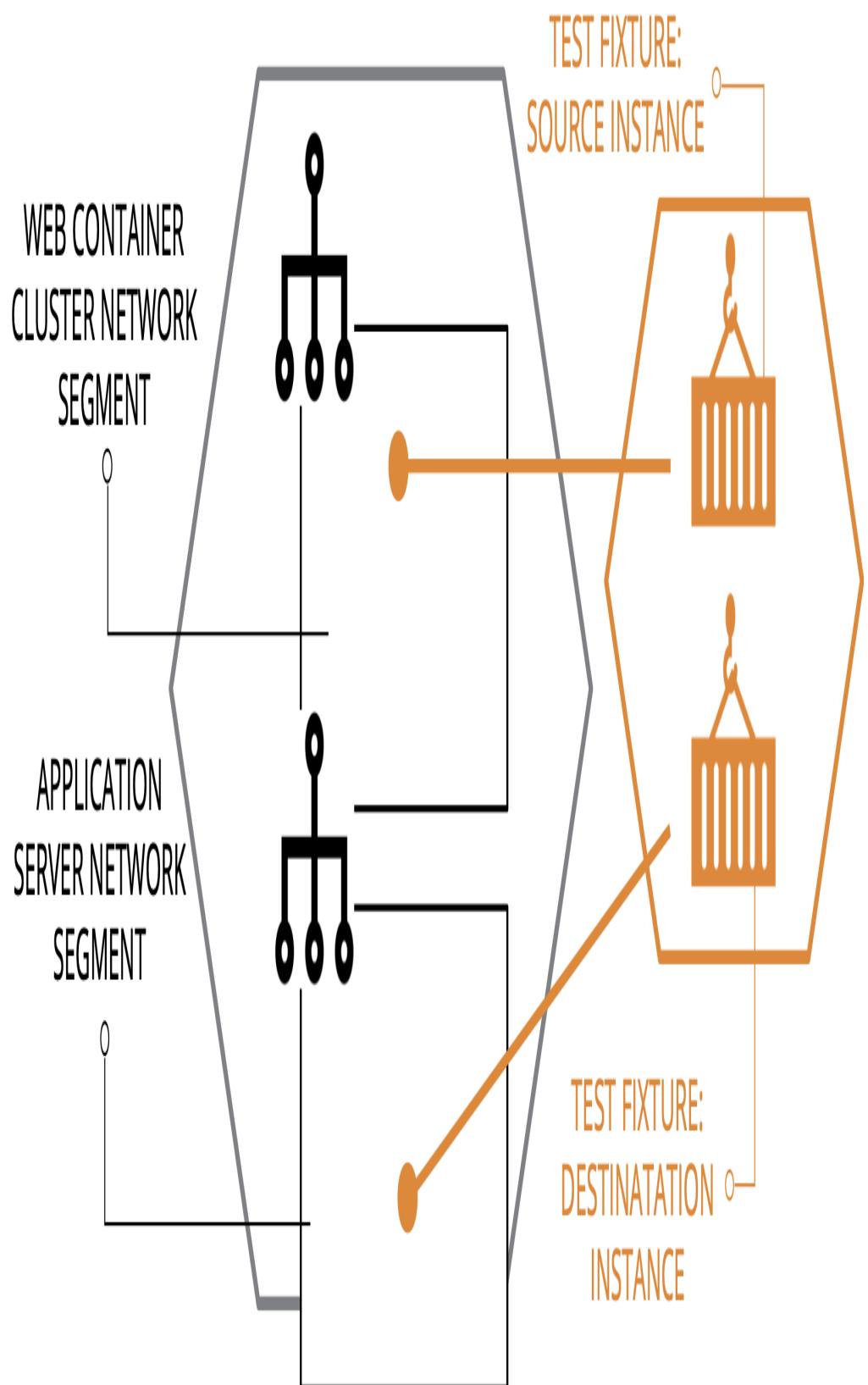


Figure 10-5. Test instance for the Foodspin network stack, with test fixtures

The test fixtures in these examples are a pair of container instances, one assigned to each of the network segments in the stack. You can often use the same testing tools that you use for verification testing (“[Verification: Making assertions about infrastructure resources](#)”) for outcome testing. These example tests use a fictional stack testing DSL:

```
given stack_instance(stack: "foodspin_networking",
                     instance: "online_test") {

    can_connect(from: $HERE,
                to: get_fixture("web_segment_instance").address,
                port:443)

    can_connect(from: get_fixture("web_segment_instance"),
                to: get_fixture("app_segment_instance").address,
                port: 8443)

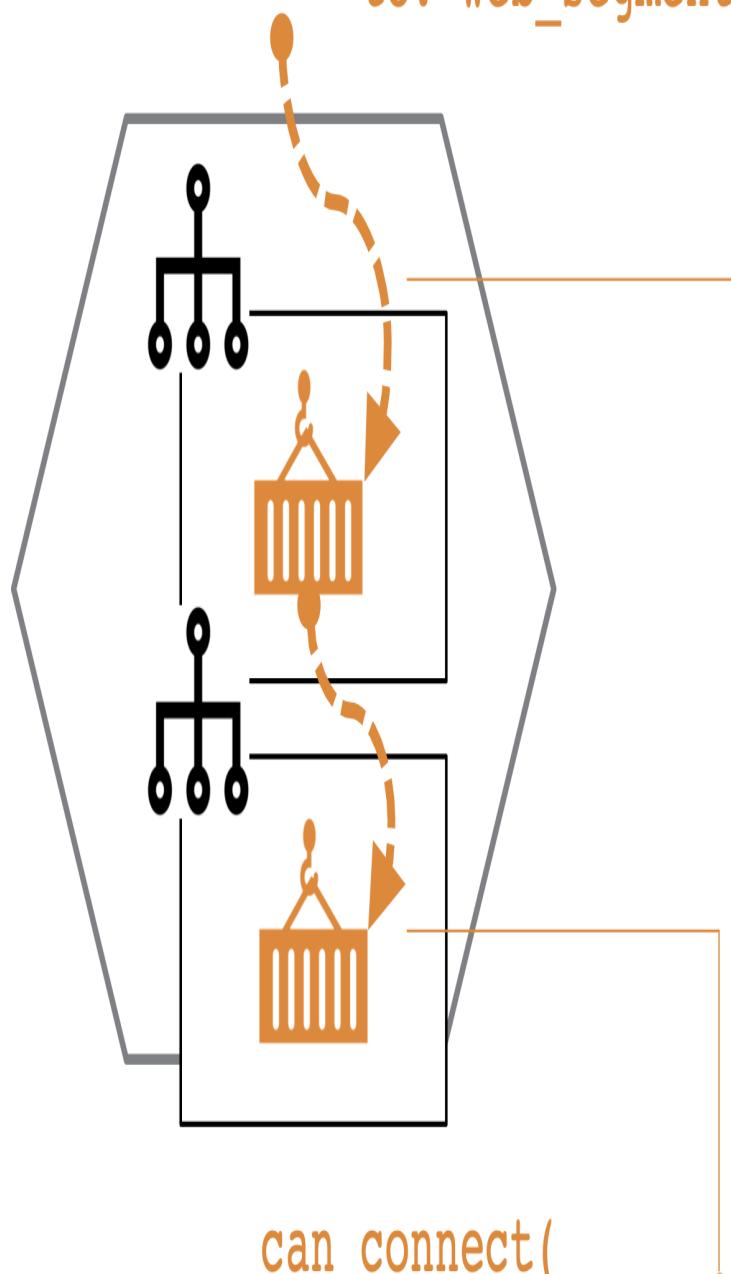
}
```

The method `can_connect` executes from `$HERE`, which would be the agent where the test code is executing, or from a container instance. It attempts to make an HTTPS connection on the specified port to an IP address. The `get_fixture()` method fetches the details of a container instance created as a text fixture.

The test framework might provide the method `can_connect`, or it could be a custom method that the team writes.

You can see the connections that the example test code makes in [Figure 10-6](#).

```
can_connect(  
    from: $HERE,  
    to: web_segment_instance )
```



```
can_connect(  
    from: web_segment_instance,  
    to: app_segment_instance )
```

Figure 10-6. Testing connectivity in the Foodspin network stack

The diagram shows the paths for both tests. The first test connects from outside the stack to the test fixture in the web segment. The second test connects from the fixture in the web segment to the fixture in the application segment.

REFACTOR COMPONENTS SO THEY CAN BE ISOLATED

Sometimes a particular component can't be easily isolated. Dependencies on other components may be hardcoded or simply too messy to pull apart. One of the benefits of writing tests while designing and building systems, rather than afterward, is that it forces you to improve your designs. A component that is difficult to test in isolation is a symptom of design issues. A well-designed system should have loosely coupled components.

So when you run across components that are difficult to isolate, you should fix the design. You may need to completely rewrite components, or replace libraries, tools, or applications. As the saying goes, this is a feature, not a bug. Clean design and loosely coupled code is a byproduct of making a system testable.

There are several strategies for restructuring systems. Refactoring^{footnote:}[Martin Fowler has written about [refactoring](#) as well as other patterns and techniques for improving system architecture. The [Strangler Application](#) prioritizes keeping the system fully working throughout the process of restructuring a system.

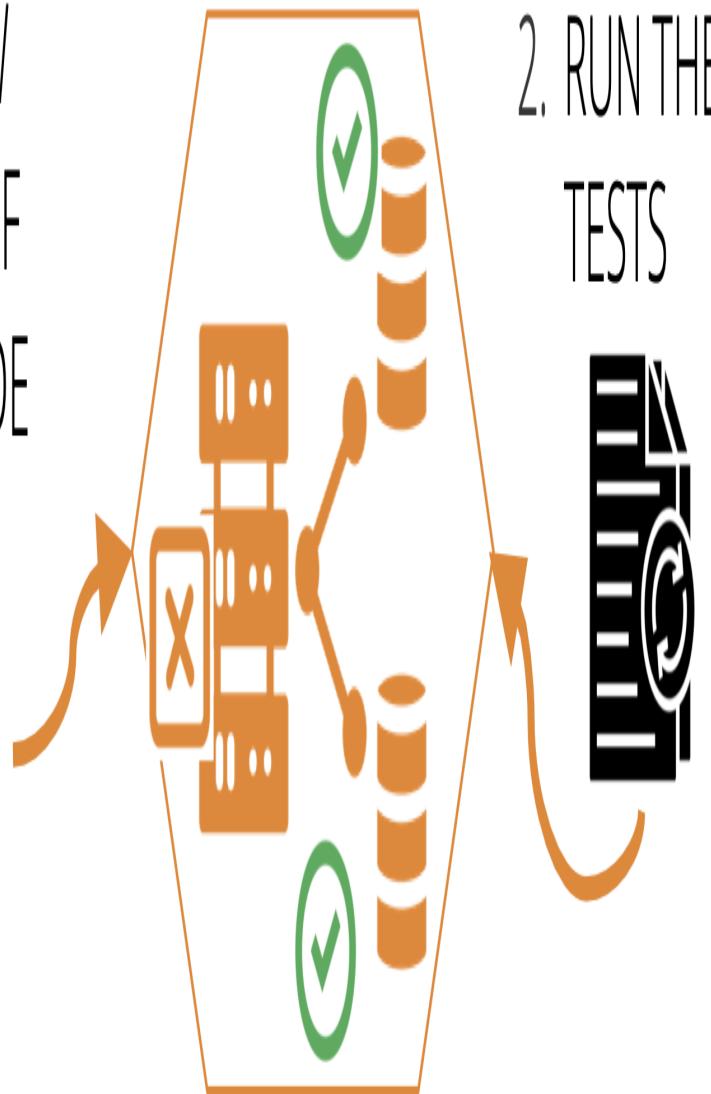
Lifecycle patterns for test instances of stacks

Before virtualization and cloud, everyone maintained static, long-lived test environments. Although many teams still have these environments, there are advantages to creating and destroying environments on demand. The following patterns describe the tradeoffs of keeping a persistent stack instance, creating an ephemeral instance for each test run, and ways of combining both approaches. You can also apply these patterns to application and full system test environments as well as to testing infrastructure stack code.

Pattern: Persistent test stack

A testing stage can use a Persistent Test Stack instance that is always running. The stage applies each code change as an update to the existing stack instance, runs the tests, and leaves the resulting modified stack in place for the next run.

1. APPLY NEW
VERSION OF
STACK CODE



3. DONE!

Figure 10-7. Persistent test stack instance

ALSO KNOWN AS

Static environment.

MOTIVATION

It's usually much faster to apply changes to an existing stack instance than to create a new instance. So the persistent test stack can give faster feedback, not only for the stage itself but for the full pipeline.

APPLICABILITY

A persistent test stack is useful when you can reliably apply your stack code to the instance. If you find yourself spending time fixing broken instances to get the pipeline running again, you should consider one of the other patterns in this chapter.

CONSEQUENCES

Most stack tools, at least as of this writing, have a pretty high rate of failure. A stack instance frequently gets “wedged,” when a change fails and leaves it in a state where any new attempt to apply stack code also fails. Often, an instance gets wedged so severely that the stack tool can't even destroy the stack so you can start over. So your team spends too much time manually un-wedging broken test instances.

You can often reduce the frequency of wedged stacks through better stack design. Breaking stacks down into smaller and simpler stacks, and simplifying dependencies between stacks can lower your wedge rate. See [Link to Come] for more on how to do this.

IMPLEMENTATION

It's easy to implement a persistent test stack. Your pipeline stage runs the stack tool command to update the instance with the relevant version of the stack code, runs the tests, and then leaves the stack instance in place when finished.

You may rebuild the stack completely as an ad-hoc process, such as someone running the tool from their local computer, or using an extra stage or job outside the routine pipeline flow.

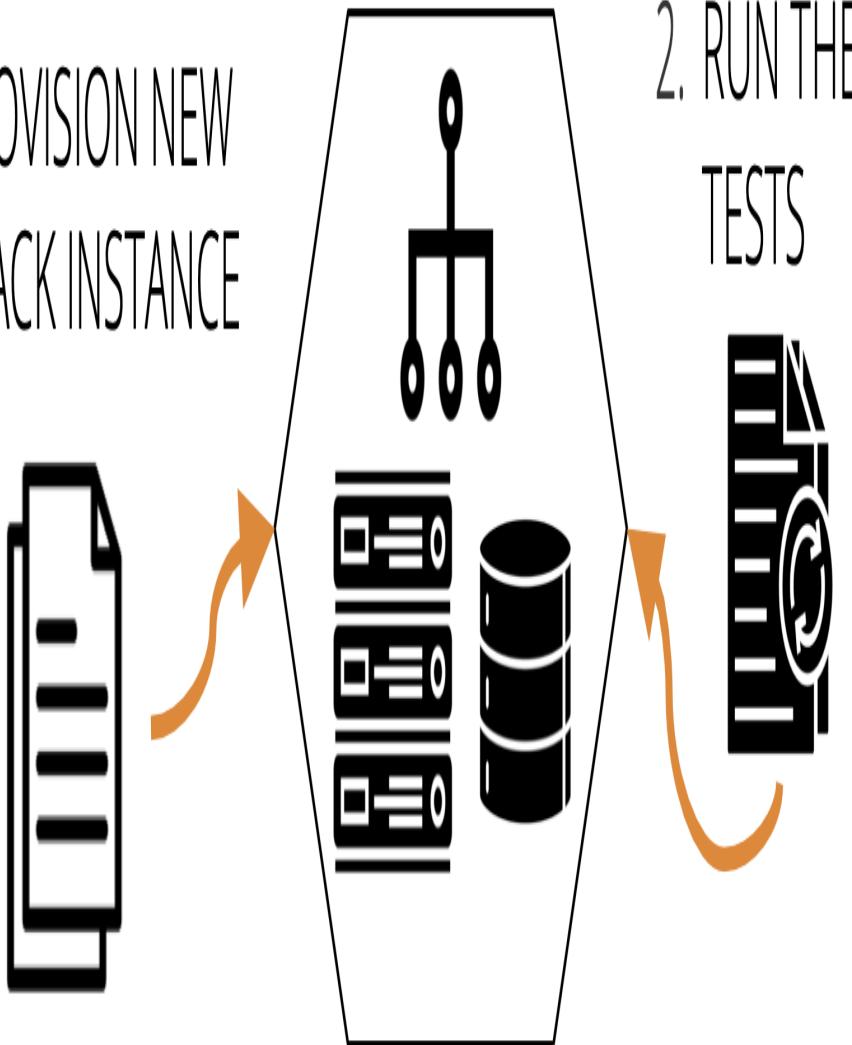
RELATED PATTERNS

The periodic stack rebuild pattern “[Pattern: Periodic stack rebuild](#)” is a simple tweak to this pattern, tearing the instance down at the end of the working day and building a new one every morning.

Pattern: Ephemeral test stack

With the Ephemeral Test Stack pattern, the test stage creates and destroys a new instance of the stack every time it runs.

1. PROVISION NEW
STACK INSTANCE



2. RUN THE
TESTS



3. DESTROY THE
STACK INSTANCE



Figure 10-8. Ephemeral test stack instance

ALSO KNOWN AS

Temporary stack, Ad-hoc stack.

MOTIVATION

An ephemeral test stack provides a clean environment for each run of the tests. There is no risk from data, fixtures, or other “cruft” left over from a previous run.

APPLICABILITY

You may want to use ephemeral instances for stacks that are quick to provision from scratch. “Quick” is relative to the feedback loop you and your teams need. For more frequent changes, like commits to application code during rapid development phases, the time to build a new environment is probably longer than people can tolerate. But less frequent changes, such as OS patch updates, may be acceptable to test with a complete rebuild.

CONSEQUENCES

Stacks generally take a long time to provision from scratch. So stages using ephemeral stack instances make feedback loops and delivery cycles slower.

IMPLEMENTATION

To implement an ephemeral test instance, your test stage should run the stack tool command to destroy the stack instance when testing and reporting have completed. You may want to configure the stage to stop before destroying the instance if the tests fail so that people can debug the failure.

RELATED PATTERNS

The continuous stack reset pattern (“Pattern: Continuous stack reset”) is similar, but runs the stack creation and destruction commands out of band from the stage, so the time taken doesn’t affect feedback loops.

Antipattern: Dual Persistent and Ephemeral Stack Stages

With Persistent and Ephemeral Stack Stages, the pipeline sends each change to a stack to two different stages, one that uses an ephemeral stack instance, and one that uses a persistent stack instance. This combined the persistent test stack pattern (“Pattern: Persistent test stack”) and the ephemeral test stack pattern (“Pattern: Ephemeral test stack”).

ALSO KNOWN AS

Quick and dirty plus slow and clean.

MOTIVATION

Teams usually implement this to work around the disadvantages of each of the two patterns it combines. If all works well, the “quick and dirty” stage (the one using the persistent instance) provides fast feedback. If that stage fails because the environment becomes wedged, you will get feedback eventually from the “slow and clean” stage (the one using the ephemeral instance).

APPLICABILITY

It might be worth implementing both types of stages as an interim solution while moving to a more reliable solution.

CONSEQUENCES

In practice, using both types of stack lifecycle combines the disadvantages of both. If updating an existing stack is unreliable, then your team will still spend time manually fixing that stage when it goes wrong. And you probably wait until the slower stage passes before being confident that a change is good.

This antipattern is also expensive, since it uses double the infrastructure resources, at least during the test run.

IMPLEMENTATION

You implement dual stages by creating two pipeline stages, both triggered by the previous stage in the pipeline for the stack project. You may require both stages to pass before promoting the stack version to the following stage, or you may promote it when either of the stages passes.

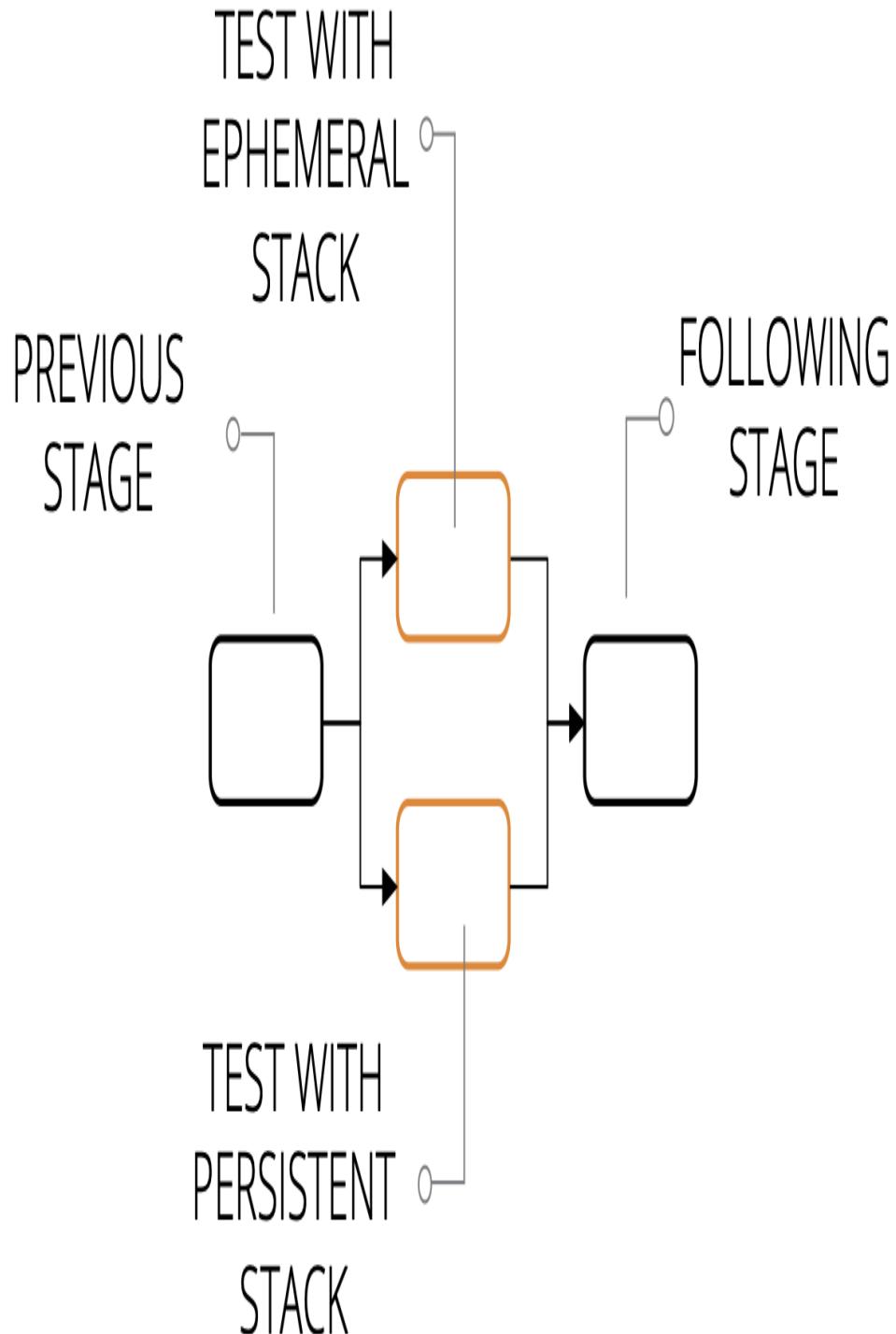


Figure 10-9. Dual Persistent and Ephemeral Stack Stages

RELATED PATTERNS

This antipattern combines the persistent test stack pattern (“Pattern: Persistent test stack”) and the ephemeral test stack

pattern (“[Pattern: Ephemeral test stack](#)”).

Pattern: Periodic stack rebuild

Periodic Stack Rebuild uses a persistent test stack instance (“[Pattern: Persistent test stack](#)”) for the stack test stage, and then has a process that runs -out-of-band to destroy and rebuild the stack instance on a schedule, such as nightly.

ALSO KNOWN AS

Nightly rebuild.

MOTIVATION

People often use periodic rebuilds to reduce costs. They destroy the stack at the end of the working day and provision a new one at the start of the next day.

Periodic rebuilds might help with unreliable stack updates, depending on why the updates are unreliable. In some cases, the resource usage of instances builds up over time, such as memory or storage that accumulates across test runs. Regular resets can clear these out.

APPLICABILITY

Rebuilding a stack instance to work around resource usage usually masks underlying problems or design issues. In this case, this pattern is, at best, a temporary hack, and at worst, a way to allow problems to build up until they cause a disaster.

Destroying a stack instance when it isn’t in use to save costs is sensible, especially when using metered resources such as with

public cloud platforms.

CONSEQUENCES

If you use this pattern to free up idle resources, you need to consider how you can be sure they aren't needed. For example, people working outside of office hours, or in other timezones, may be blocked without test environments.

IMPLEMENTATION

Most pipeline orchestration tools make it easy to create jobs that run on a schedule to destroy and build stack instances. A more sophisticated solution would run based on activity levels. For example, you could have a job that destroys an instance if the test stage hasn't run in the past hour.

There are three options for triggering the build of a fresh instance after destroying the previous instance. One is to rebuild it right away after destroying it. This approach clears resources but doesn't save costs.

A second option is to build the new environment instance at a scheduled point in time. But it may stop people from working flexible hours.

The third option is for the test stage to provision a new instance if it doesn't currently exist. Create a separate job that destroys the instance, either on a schedule or after a period of inactivity. Each time the testing stage runs, it first checks whether the instance is already running. If not, it provisions a new instance first. With this approach, people occasionally need to wait longer than usual to get

test results. If they are the first person to push a change in the morning, they need to wait for the system to provision the stack.

RELATED PATTERNS

This pattern can play out like the persistent test stack pattern (“[Pattern: Persistent test stack](#)”)-if your stack updates are unreliable, people spend time fixing broken instances.

Pattern: Continuous stack reset

With the Continuous Stack Reset pattern, every time the stack testing stage completes, an out-of-band job destroys and rebuilds the stack instance.

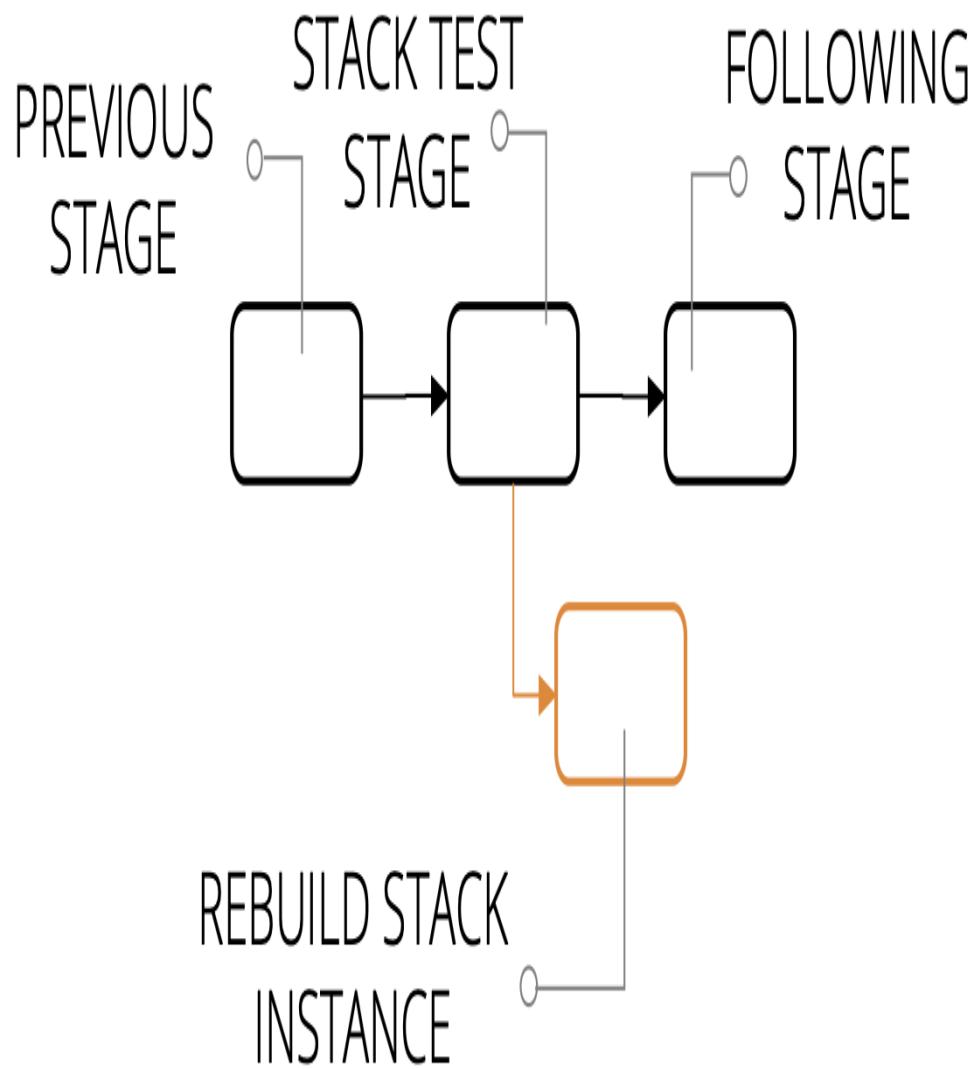


Figure 10-10. Pipeline flow for continuous stack reset

ALSO KNOWN AS

Behind the scenes rebuild

MOTIVATION

Destroying and rebuilding the stack instance every time provides a clean slate to each testing run. It may automatically remove a broken instance unless it is too broken for the stack tool to destroy. And it removes the time it takes to create and destroy the stack instance from the feedback loop.

Another benefit of this pattern is that it can reliably test the update process that would happen for the given stack code version in production.

APPLICABILITY

Destroying the stack instance in the background can work well if the stack project doesn't tend to break and need manual intervention to fix.

CONSEQUENCES

Since the stack is destroyed and provisioned outside the delivery flow of the pipeline, problems may not be visible. The pipeline can be green, but the test instance may break behind the scenes. When the next change reaches the test stage, it may take time to realize it failed because of the background job rather than because of the change itself.

IMPLEMENTATION

When the test stage passes, it promotes the stack project code to the next stage. It also triggers a job to destroy and rebuild the stack instance. When someone pushes a new change to the code, the test stage applies it to the instance as an update.

You need to decide which version of the stack code to use when rebuilding the instance. You could use the same version that has just passed the stage. An alternative is to pull the last version of the stack code applied to the production instance. This way, each version of stack code is tested as an update to the current production version. Depending on how your infrastructure code

typically flows to production, this may be a more accurate representation of the production upgrade process.

RELATED PATTERNS

Ideally, this pattern resembles the persistent test stack pattern (“Pattern: Persistent test stack”), providing feedback, while having the reliability of the ephemeral test stack pattern (“Pattern: Ephemeral test stack”).

Test orchestration

I’ve described each of the moving parts involved in testing stacks: the types of tests and validations you can apply, using test fixtures to handle dependencies, and lifecycles for test stack instances. But how should you put these together to set up and run tests?

Most teams use scripts to orchestrate their tests. Often, these are the same scripts they use to orchestrate running their stack tools. In [Link to Come], I’ll dig into these scripts, which may handle configuration, coordinating actions across multiple stacks, and other activities as well as testing.

Test orchestration may involve:

- Creating test fixtures,
- Loading test data (more often needed for application testing than infrastructure testing),
- Managing the lifecycle of test stack instances,
- Providing parameters to the test tool,
- Running the test tool,

- Consolidating test results,
- Cleaning up test instances, fixtures, and data.

Most of these topics, such as test fixtures and stack instance lifecycle, are covered earlier in this chapter. Others, including running the tests and consolidating the results, depend on the particular tool.

Two guidelines to consider for orchestrating tests are supporting local testing and avoiding tight coupling to pipeline tools.

Support local testing

People working on infrastructure stack code should be able to run the tests themselves before pushing code into the shared pipeline and environments. [Link to Come] discusses approaches to help people work with personal stack instances on an infrastructure platform. Doing this allows you to code and run online tests before pushing changes.

As well as being able to work with personal test instances of stacks, people need to have the testing tools and other elements involved in running tests on their local working environment.

Many teams use code-driven development environments, which automate installing and configuring tools. You can use containers or virtual machines⁹ for packaging development environments that can run on different types of desktop systems. Alternately, your team could use hosted workstations (hopefully configured as code), although these may suffer from latency, especially for distributed teams.

A key to making it easy for people to run tests themselves is using the same test orchestration scripts across local work and pipeline stages. Doing this ensures that tests are set up and run consistently everywhere.

Avoid tight coupling with pipeline tools

Many CI and pipeline orchestration tools have features or plugins for test orchestration, even configuring and running the tests for you. While these features may seem convenient, they make it difficult to set up and run your tests consistently outside the pipeline. Mixing test and pipeline configuration can also make it painful to make changes.

Instead, you should implement your test orchestration in a separate script or tool. The test stage should call this tool, passing a minimum of configuration parameters. This approach keeps the concerns of pipeline orchestration and test orchestration loosely coupled.

Test orchestration tools

Many teams write custom scripts to orchestrate tests. These scripts are similar to or may even be the same scripts used to orchestrate stack management (as described in [Link to Come]). People use Bash scripts, batch files, Ruby, Python, Make, Rake, and others I've never heard of.

There are a few tools available that are specifically designed to orchestrate infrastructure tests. Two I know of are [Test Kitchen]<https://kitchen.ci/> and [Molecule]<https://molecule.readthedocs.io/en/stable/>. Test Kitchen

is an open-source product from Chef that was originally aimed at testing Chef cookbooks. Molecule is an open-source tool designed for testing Ansible playbooks. You can use either tool to test infrastructure stacks, for example, using [Kitchen-Terraform]<https://newcontext-oss.github.io/kitchen-terraform/>.

The challenge with these tools is that they are designed with a particular workflow in mind, and can be difficult to configure to suit the workflow you need. Some people tweak and massage them, while others find it simpler to write their own scripts.

Conclusion

This chapter pulls together the topics of infrastructure stacks and testing. These approaches and patterns may guide you in implementing automated testing for your infrastructure code, hopefully helping you to cope with some of the challenges which make this harder than testing application code.

One of the challenges this book can't help with is tooling for testing infrastructure. Although there are some tools available, many of which I mention in this chapter, TDD, CI, and automated testing are not very well established for infrastructure as of this writing. You have a journey to discover the tools that you can use, and may need to cover gaps in the tooling with custom scripting. Hopefully, this will improve over time.

The next section of this book moves on to the next layer of infrastructure, application runtimes. You use stacks to provision servers, clusters, and other resources for deploying and running

applications. These chapters explain how to integrate testing and pipelines for these things with testing and pipelines for stacks.

- 1 I give the background on Foodspin in “[Foodspin Example: Diverging infrastructure](#)”
- 2 See “[Structured data storage](#)” for a bit more on DBaaS
- 3 This pipeline is much simpler than what you’d use in reality. You would probably have at least one stage to test the stack and the application together (see [Link to Come]). You might also need a customer acceptance testing stage before each customer production stage. And this doesn’t include governance and approval stages, which many organizations require.
- 4 The term [https://en.wikipedia.org/wiki/Lint_\(software\)](https://en.wikipedia.org/wiki/Lint_(software))[lint]
- 5 [Awspec](#) is an rspec-based framework for testing AWS infrastructure
- 6 [Inspec](#) is another rspec-based framework, which tests multiple cloud platforms, as well as other types of infrastructure, such as servers and containers.
- 7 [Terratest](#) is a Go-based framework specific to Terraform but applicable to other types of infrastructure, including servers and containers.
- 8 [Link to Come] explains how to connect stack dependencies, in [Link to Come].
- 9 [Vagrant](#) is handy for sharing virtual machine configuration between members of a team.

About the Author

Kief Morris is a consultant with ThoughtWorks, specializing in cloud, infrastructure, and agile IT operations. This gives him an opportunity to spend time with teams in a variety of 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 organizational structures and processes. Kief ran his first online system, a bulletin board system (BBS) in Florida in the early 1990s. He later enrolled in a MSc 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. 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 Deputy Technical Director, R&D Manager, Hosting Manager, Technical Lead, Technical Architect, Consultant, and Infrastructure Engineering Practices Champion.

Colophon

The animal on the cover of *Infrastructure as Code* is Rüppell's vulture (*Gyps rueppellii*), native to the Sahel region of Africa (a geographic zone that serves as a transition between the Sahara Desert and the savanna). It is named in honor of a 19th-century German explorer and zoologist, Eduard Rüppell.

It is a large bird (with a wingspan of 7–8 feet and weighing 14–20 pounds) with 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. While normally silent, these are very social birds who will voice a loud squealing call at colony nesting sites or when fighting over food.

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 lined with grass and leaves (and often use it for multiple years). Only one egg is laid each year—by the time the next breeding season begins, the chick is just becoming independent. This vulture does not fly very fast (about 22 mph), but will venture 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 them flying 37,000 feet above sea level, as high as commercial aircraft. They have a special hemoglobin in their blood that allows them to absorb oxygen more efficiently at high altitudes.

This species is considered endangered and populations have been in decline. 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. As vultures identify a meal by sight and gather around it in flocks, hundreds of birds can be killed each time.

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 Cassell's *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.