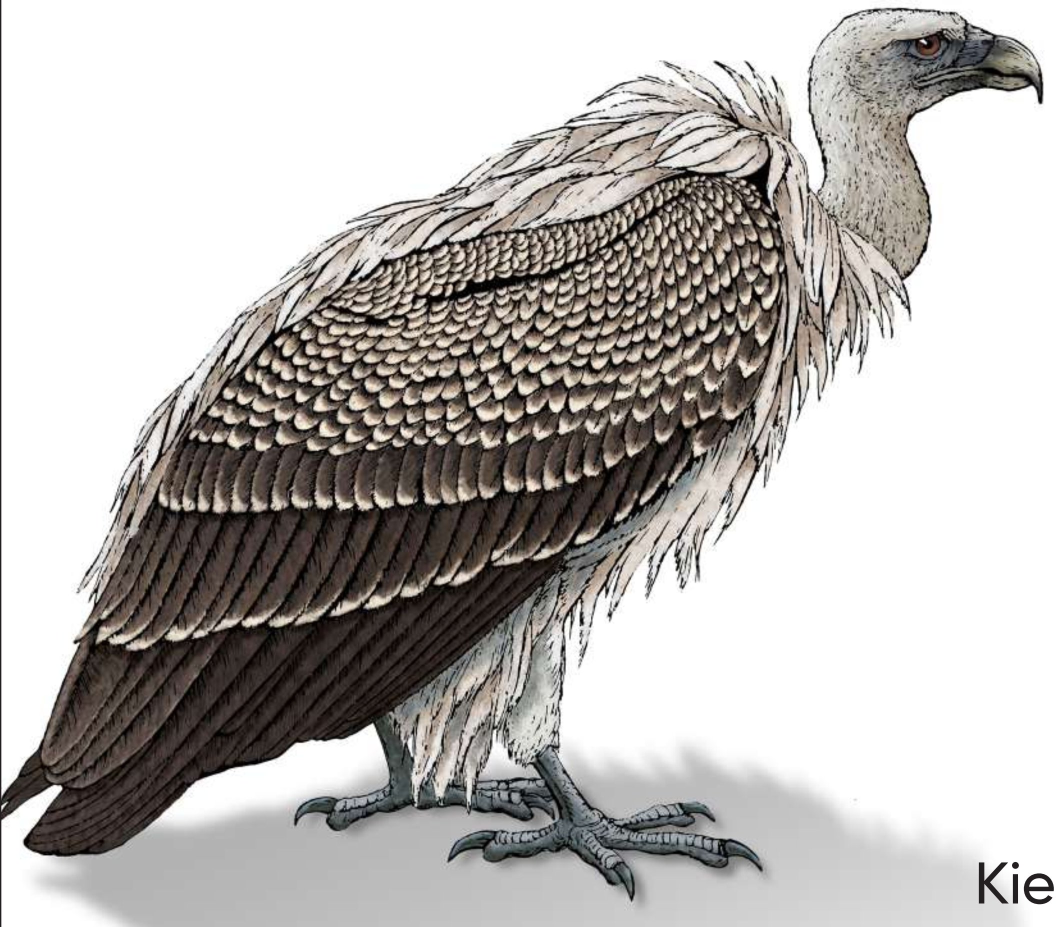
O'REILLY®

Second Edition

Infrastructure as Code

Dynamic Systems for the Cloud Age



Kief Morris

O'REILLY®

Infrastructure as Code

Six years ago, Infrastructure as Code was a new concept. Today, as even banks and other conservative organizations plan moves to the cloud, development teams for companies worldwide are attempting to build large infrastructure codebases. With this practical book, Kief Morris of ThoughtWorks shows you how to effectively use principles, practices, and patterns pioneered by DevOps teams to manage cloud-age infrastructure.

Ideal for system administrators, infrastructure engineers, software developers, team leads, and architects, this updated edition demonstrates how you can exploit cloud and automation technology to make changes easily, safely, quickly, and responsibly. You'll learn how to define everything as code and apply software design and engineering practices to build your system from small, loosely coupled pieces.

This book covers:

- **Foundations:** Use Infrastructure as Code to drive continuous change and raise the bar of operational quality, using tools and technologies to build cloud-based platforms
- **Working with infrastructure stacks:** Learn how to define, provision, test, and continuously deliver changes to infrastructure resources
- **Working with servers and other platforms:** Use patterns to design provisioning and configuration of servers and clusters
- **Working with large systems and teams:** Learn workflows, governance, and architectural patterns to create and manage infrastructure elements

> "Infrastructure as Code practices have evolved from managing servers to managing complete stacks, but this new power comes at the cost of complexity. This book will take you beyond the commands to the design patterns behind good practices and the how-tos of next-level automation."
>
> **—Patrick Debois**
> DevOpsDays Founder

Kief Morris is global director of cloud engineering at ThoughtWorks. He helps organizations and teams explore better ways to apply cloud and infrastructure technology to deliver stronger value more quickly and reliably. He's been designing, building, and running automated IT server infrastructure for over 20 years, starting out with shell scripts and Perl and moving on to CFengine, Puppet, Chef, and Terraform among other technologies as they've emerged.

SYSTEM ADMINISTRATION

US $69.99 CAN $92.99
ISBN: 978-1-098-11467-1

56999

9 781098 114671

Praise for *Infrastructure as Code*

Infrastructure as Code practices have evolved from managing servers to managing complete stacks, but this new power comes at the cost of complexity. This book will take you beyond the commands to the design patterns behind good practices and the how-to's of next-level automation.

—Patrick Debois, DevOpsDays Founder

Infrastructure as Code sits at the intersection of multiple software engineering domains. This book regroups and distills the relevant best practices from each domain to give readers an accelerated course on infrastructure automation.

—Carlos Condé, VP of Engineering at Sweetgreen

A down-to-earth guide for navigating the landscape of cloud infrastructure with principles, practices and patterns.

—Effy Elden, Technologist at ThoughtWorks

SECOND EDITION

Infrastructure as Code

Dynamic Systems for the Cloud Age

Kief Morris

Beijing · Boston · Farnham · Sebastopol · Tokyo O'REILLY®

Infrastructure as Code

by Kief Morris

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*.

Acquisitions Editor: John Devins
Development Editor: Virginia Wilson
Production Editor: Kate Galloway
Copyeditor: Kim Cofer
Proofreader: nSight, Inc.

Indexer: Potomac Indexing, LLC
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: John Francis Amalanathan

June 2016:	First Edition
December 2020:	Second Edition

Revision History for the Second Edition

2020-11-17:	First Release
2021-01-15:	Second Release

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

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.

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

978-1-098-11467-1

[GP]

Table of Contents

Part I. Foundations

Part II. Working with Infrastructure Stacks

Part III. Working with Servers and Other Application Runtime Platforms

Part IV. Designing Infrastructure

Part V. Delivering Infrastructure

Preface

Ten years ago, a CIO at a global bank scoffed when I suggested they look into private cloud technologies and infrastructure automation tooling: "That kind of thing might be fine for startups, but we're too large and our requirements are too complex." Even a few years ago, many enterprises considered using public clouds to be out of the question.

These days cloud technology is pervasive. Even the largest, most hidebound organizations are rapidly adopting a "cloud-first" strategy. Those organizations that find themselves unable to consider public clouds are adopting dynamically provisioned infrastructure platforms in their data centers.[1] The capabilities that these platforms offer are evolving and improving so quickly that it's hard to ignore them without risking obsolescence.

Cloud and automation technologies remove barriers to making changes to production systems, and this creates new challenges. While most organizations want to speed up their pace of change, they can't afford to ignore risks and the need for governance. Traditional processes and techniques for changing infrastructure safely are not designed to cope with a rapid pace of change. These ways of working tend to throttle the benefits of modern, Cloud Age technologies—slowing work down and harming stability.[2]

[1] For example, many government and financial organizations in countries without a cloud presence are prevented by law from hosting data or transactions abroad.

[2] The research published by DORA in the *State of DevOps Report* finds that heavyweight change-management processes correlate to poor performance on change failure rates and other measures of software delivery effectiveness.

In Chapter 1 I use the terms "Iron Age" and "Cloud Age" ("From the Iron Age to the Cloud Age" on page 2) to describe the different philosophies that apply to managing physical infrastructure, where mistakes are slow and costly to correct, and managing virtual infrastructure, where mistakes can be quickly detected and fixed.

Infrastructure as Code tools create the opportunity to work in ways that help you to deliver changes more frequently, quickly, and reliably, improving the overall quality of your systems. But the benefits don't come from the tools themselves. They come from how you use them. The trick is to leverage the technology to embed quality, reliability, and compliance into the process of making changes.

Why I Wrote This Book

I wrote the first edition of this book because I didn't see a cohesive collection of guidance on how to manage Infrastructure as Code. There was plenty of advice scattered across blog posts, conference talks, and documentation for products and projects. But a practitioner needed to sift through everything and piece a strategy together for themselves, and most people simply didn't have time.

The experience of writing the first edition was amazing. It gave me the opportunity to travel and to talk with people around the world about their own experiences. These conversations gave me new insights and exposed me to new challenges. I learned that the value of writing a book, speaking at conferences, and consulting with clients is that it fosters conversations. As an industry, we are still gathering, sharing, and evolving our ideas for managing Infrastructure as Code.

What's New and Different in This Edition

Things have moved along since the first edition came out in June 2016. That edition was subtitled "Managing Servers in the Cloud," which reflected the fact that most infrastructure automation until that point had been focused on configuring servers. Since then, containers and clusters have become a much bigger deal, and the infrastructure action has moved to managing collections of infrastructure resources provisioned from cloud platforms—what I call *stacks* in this book.

As a result, this edition involves more coverage of building stacks, which is the remit of tools like CloudFormation and Terraform. The view I've taken is that we use stack management tools to assemble collections of infrastructure that provide application runtime environments. Those runtime environments may include servers, clusters, and serverless execution environments.

I've changed quite a bit based on what I've learned about the evolving challenges and needs of teams building infrastructure. As I've already touched on in this preface, I see making it safe and easy to change infrastructure as the key benefit of

Infrastructure as Code. I believe people underestimate the importance of this by thinking that infrastructure is something you build and forget.

But too many teams I meet struggle to meet the needs of their organizations; they are not able to expand and scale quickly enough, support the pace of software delivery, or provide the reliability and security expected. And when we dig into the details of their challenges, it's that they are overwhelmed by the need to update, fix, and improve their systems. So I've doubled down on this as the core theme of this book.

This edition introduces three core practices for using Infrastructure as Code to make changes safely and easily:

Define everything as code
: This one is obvious from the name, and creates repeatability and consistency.

Continuously test and deliver all work in progress
: Each change enhances safety. It also makes it possible to move faster and with more confidence.

Build small, simple pieces that you can change independently
: These are easier and safer to change than larger pieces.

These three practices are mutually reinforcing. Code is easy to track, version, and deliver across the stages of a change management process. It's easier to continuously test smaller pieces. Continuously testing each piece on its own forces you to keep a loosely coupled design.

These practices and the details of how to apply them are familiar from the world of software development. I drew on Agile software engineering and delivery practices for the first edition of the book. For this edition, I've also drawn on rules and practices for effective design.

In the past few years, I've seen teams struggle with larger and more complicated infrastructure systems, and I've seen the benefits of applying lessons learned in software design patterns and principles, so I've included several chapters in this book on how to do this.

I've also seen that organizing and working with infrastructure code is difficult for many teams, so I've addressed various pain points. I describe how to keep codebases well organized, how to provide development and test instances for infrastructure, and how to manage the collaboration of multiple people, including those responsible for governance.

What's Next

I don't believe we've matured as an industry in how we manage infrastructure. I'm hoping this book gives a decent view of what teams are finding effective these days. And a bit of aspiration of what we can do better.

I fully expect that in another five years the toolchains and approaches will evolve. We could see more general-purpose languages used to build libraries, and we could be dynamically generating infrastructure rather than defining the static details of environments at a low level. We certainly need to get better at managing changes to live infrastructure. Most teams I know are scared when applying code to live infrastructure. (One team referred to Terraform as "Terrorform," but users of other tools all feel this way.)

What This Book Is and Isn't

The thesis of this book is that exploring different ways of using tools to implement infrastructure can help us to improve the quality of services we provide. We aim to use speed and frequency of delivery to improve the reliability and quality of what we deliver.

So the focus of this book is less on specific tools, and more on how to use them.

Although I mention examples of tools for particular functions like configuring servers and provisioning stacks, you won't find details of how to use a particular tool or cloud platform. You will find patterns, practices, and techniques that should be relevant to whatever tools and platforms you use.

You won't find code examples for real-world tools or clouds. Tools change too quickly in this field to keep code examples accurate, but the advice in this book should age more slowly, and be applicable across tools. Instead, I write pseudocode examples for fictional tools to illustrate concepts. See the book's companion website for references to example projects and code.

This book won't guide you on how to use the Linux operating system, Kubernetes cluster configuration, or network routing. The scope of this book does include ways to provision infrastructure resources to create these things, and how to use code to deliver them. I share different cluster topology patterns and approaches for defining and managing clusters as code. I describe patterns for provisioning, configuring, and changing server instances using code.

You should supplement the practices in this book with resources on the specific operating systems, clustering technologies, and cloud platforms. Again, this book explains approaches for using these tools and technologies that are relevant regardless of the particular tool.

This book is also light on operability topics like monitoring and observability, log aggregation, identity management, and other concerns that you need to support services in a cloud environment. What's in here should help you to manage the infrastructure needed for these services as code, but the details of the specific services are, again, something you'll find in more specific resources.

Some History of Infrastructure as Code

Infrastructure as Code tools and practices emerged well before the term. Systems administrators have been using scripts to help them manage systems since the beginning. Mark Burgess created the pioneering CFEngine system in 1993. I first learned practices for using code to fully automate provisioning and updates of servers from the Infrastructures.org website in the early 2000s.[3]

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 called "Agile Infrastructure" that Clay-Shafer gave at the Velocity conference in 2009, and an article John Willis wrote summarizing the talk. Adam Jacob, who cofounded Chef, and Luke Kanies, founder of Puppet, were also using the phrase around this time.

Who This Book Is For

This book is for people who are involved in providing and using infrastructure to deliver and run software. You may have a background in systems and infrastructure, or in software development and delivery. Your role may be engineering, testing, architecture, or management. I'm assuming you have some exposure to cloud or virtualized infrastructure and tools for automating infrastructure using code.

Readers new to Infrastructure as Code should find this book a good introduction to the topic, although you will get the most out of it if you are familiar with how infrastructure cloud platforms work, and the basics of at least one infrastructure coding tool.

Those who have more experience working with these tools should find a mixture of familiar and new concepts and approaches. The content should create a common language and articulate challenges and solutions in ways that experienced practitioners and teams find useful.

[3] The original content remains on this site as of summer 2020, although it hadn't been updated since 2007.

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 to do it 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 you should 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 being 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.

The ShopSpinner Examples

I use a fictional company called ShopSpinner to illustrate concepts throughout this book. ShopSpinner builds and runs online stores for its customers.

ShopSpinner runs on FCS, the Fictional Cloud Service, a public IaaS provider with services that include FSI (Fictional Server Images) and FKS (Fictional Kubernetes Service). It uses the *Stackmaker* tool—an analog of Terraform, CloudFormation, and

Pulumi—to define and manage infrastructure on its cloud. It configures servers with the *Servermaker* tool, which is much like Ansible, Chef, or Puppet.

ShopSpinner's infrastructure and system design may vary depending on the point I'm using it to make, as will the syntax of the code and command-line arguments for its fictional tools.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

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

`Constant width`

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

`Constant width bold`

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

`Constant width italic`

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

This element signifies a tip or suggestion.

This element signifies a general note.

This element indicates a warning or caution.

O'Reilly Online Learning

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

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

How to Contact Us

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

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

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *https://oreil.ly/infra-as-code-2e*.

Email *bookquestions@oreilly.com* to comment or ask technical questions about this book.

For news and information about our books and courses, visit *http://oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

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

Acknowledgments

As with the first edition, this book is not my product—it's the result of collating and consolidating as best I could what I've learned from more people than I'm able to remember and properly credit. Apologies and thanks to those I've forgotten to name here.

I always enjoy knocking ideas around with James Lewis; our conversations and his writing and talks have directly and indirectly influenced much of what's in this book. He kindly shared his deep experience on software design and broad awareness of various other topics by giving me thoughts on a near-final draft of the book. His suggestions helped me to tighten up the connections I've tried to draw between software engineering and Infrastructure as Code.

Martin Fowler has generously supported my efforts from the beginning. His skill at drawing from various people's experiences, applying his knowledge and perception, and shaping it all into clear, helpful advice inspires me.

Thierry de Pauw has been a most thoughtful and helpful reviewer. He read multiple drafts and shared his reactions, telling me what he found new and useful, which ideas aligned with his own experiences, and which parts didn't come across clearly to him.

I need to thank Abigail Bangser, Jon Barber, Max Griffiths, Anne Simmons, and Claire Walkley for their encouragement and inspiration.

People I've worked with gave me thoughts and ideas that improved this book. James Green shared insights into data engineering and machine learning in the context of infrastructure. Pat Downey explained his use of expand and contract for infrastructure. Vincenzo Fabrizi pointed out to me the value of inversion of control for infrastructure dependencies. Effy Elden is an endless fount of knowledge about the infrastructure tooling landscape. Moritz Heiber directly and indirectly influenced the contents of this book, although it would be too much to hope that he agrees with 100% of it.

At ThoughtWorks I have the opportunity to interact with and discuss Infrastructure as Code and related topics with many colleagues and clients in workshops, projects, and online forums. A few of these people include Ama Asare, Nilakhya Chatterjee, Audrey Conceicao, Patrick Dale, Dhaval Doshi, Filip Fafara, Adam Fahie, John Feminella, Mario Fernandez, Louise Franklin, Heiko Gerin, Jarrad "Barry" Goodwin, Emily Gorcenski, James Gregory, Col Harris, Prince M Jain, Andrew Jones, Aiko Klostermann, Charles Korn, Vishwas Kumar, Punit Lad, Suya Liu, Tom Clement Oketch, Gerald Schmidt, Boss Supanat Pothivarakorn, Rodrigo Rech, Florian Sellmayr, Vladimir Sneblic, Isha Soni, Widyasari Stella, Paul Valla, Srikanth Venugopalan, Ankit Wal, Paul Yeoh, and Jiayu Yi. Also thanks to Kent Spillner—this time I remember why.

Plenty of people reviewed various drafts of this edition of the book and shared feedback, including Artashes Arabajyan, Albert Attard, Simon Bisson, Phillip Campbell, Mario Cecchi, Carlos Conde, Bamdad Dashtban, Marc Hofer, Willem van Ketwich, Barry O'Reilly, Rob Park, Robert Quinlivan, Wasin Watthanasrisong, and Rebecca Wirfs-Brock.

Deep thanks to Virginia Wilson, my editor, who has kept me going throughout the long and grueling process of making this book happen. My colleague, John Amalanathan, turned my wonky diagrams into the slick artwork you see here, with great patience and diligence.

My employer, ThoughtWorks, has been an enormous supporter. Firstly by creating the environment for me to learn from phenomenal people, secondly by fostering a culture that encourages its members to share ideas with the industry, and thirdly, by supporting me as I worked with other ThoughtWorkers and clients to explore and test new ways of working. Ashok Subramanian, Ruth Harrison, Renee Hawkins, Ken Mugrage, Rebecca Parsons, and Gayathri Rao, among others, have helped me make this more than a personal project.

Last and most, everlasting love to Ozlem and Erel, who endured my obsession with this book. Again.

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 fire contained."[2]

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

[1] This is as opposed to what many of the same people said a few years ago, which was that software was “not part of our core business.” After following this advice and outsourcing IT, organizations realized they were being overtaken by those run by people who see better software as a way to compete, rather than as a cost to cut.

[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.”

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 IT, which adds different problems.[3]

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 that 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 test and deliver 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 for quality. Instead, we use speed as a way to improve quality, and we use quality to enable delivery at speed.

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).

[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.

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.

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.

Benefits of Infrastructure as Code

To summarize, organizations adopting Infrastructure as Code to manage dynamic infrastructure 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 system, continuous change is inevitable, and making changes is the only way to improve a system, it makes sense to optimize your capability to make changes both rapidly *and* reliably.[4] Research from the *Accelerate State of DevOps Report* backs this up. Making changes frequently and reliably is correlated to organizational success.[5]

[4] According to Gene Kim, George Spafford, and Kevin Behr in *The Visible Ops Handbook* (IT Process Institute), changes cause 80% of unplanned outages.

[5] Reports from the *Accelerate* research are available in the annual *State of DevOps Report*, and in the book *Accelerate* by Dr. Nicole Forsgren, Jez Humble, Gene Kim (IT Revolution Press).

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 and should 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 not 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 that are supporting actively used systems handle a continuous stream of changes.

Consider these common examples of infrastructure changes:

- An essential new application feature requires you to add a new database.
- A new application feature needs you to upgrade 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: *Stablity 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.

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

There are three 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 many 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.

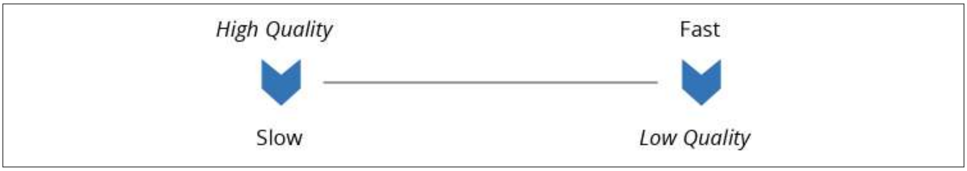


Figure 1-1. The idea that speed and quality are opposite ends of a spectrum is a false dichotomy

However, the *Accelerate* research I mentioned earlier (see "Use Infrastructure as Code to Optimize for Change" on page 4) 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.
>
> —Nicole Forsgren, PhD, *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 quadrant rather than a continuum,[6] as shown in Figure 1-2.

[6] Yes, I do work at a consultancy, why do you ask?

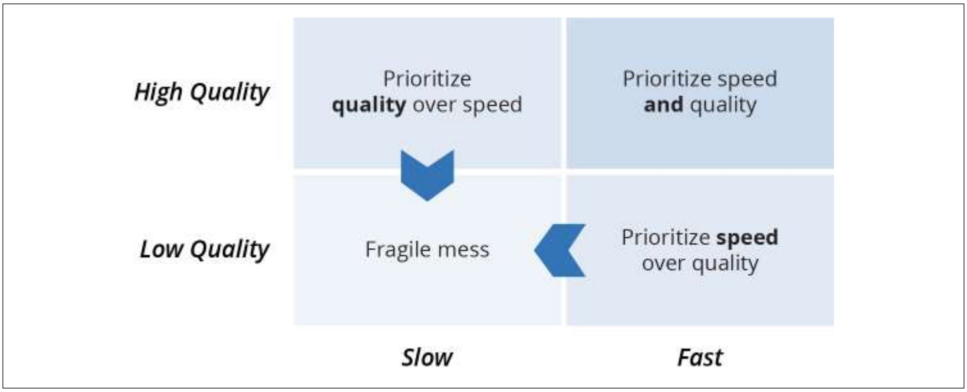

High Quality
Prioritize quality over speed
Prioritize speed and quality
Low Quality
Fragile mess
Prioritize speed over quality
Slow
Fast

Figure 1-2. Speed and quality map to quadrants

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

Lower-right quadrant: Prioritize speed over quality
: This is the "move fast and break things" philosophy. Teams that optimize for speed and sacrifice quality build messy, fragile systems. They slide into the lower-left quadrant because their shoddy systems slow them down. Many startups that 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 create barriers to improvement, so technical debt grows along with lists of "known issues." These teams slump into the lower-left quadrant. They end up with low-quality systems *because* 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 normal,[7] especially those who work in risk-sensitive industries.[8]

[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* (University Of Chicago Press).

[8] It's ironic (and scary) that so many people in industries like finance, government, and health care consider fragile IT systems—and processes that obstruct improving them—to be normal, and even desirable.

The 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."

The Four Key Metrics

DORA's *Accelerate* research team identifies four key metrics for software delivery and operational performance.[9] Its research surveys various measures, and has found that these four have the strongest correlation to how well an organization meets its goals:

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

Deployment frequency
How often you deploy changes to production systems

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

Mean Time to Restore (MTTR)
How long it takes to restore service when there is an unplanned outage or impairment

Organizations that perform well against their goals—whether that's revenue, share price, or other criteria—also perform well against these four metrics, and vice versa. The ideas in this book aim to help your team, and your organization, perform well on these metrics. Three core practices for Infrastructure as Code can help you to achieve this.

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:

[9] DORA, now part of Google, is the team behind the *Accelerate State of DevOps Report*.

- Define everything as code
- Continuously test and deliver 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, and other people can build identical instances of the thing.

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

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, gain insight to use when troubleshooting, and review and audit for compliance.

I'll expand on concepts and implementation principles for defining things as code in Chapter 4.

Core Practice: Continuously Test and Deliver All Work in Progress

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

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

One part of this 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. CI involves merging and testing everyone's code throughout development. CD takes this further, keeping the merged code always production-ready.

I'll go into more detail on how to continuously test and deliver infrastructure code in Chapter 8.

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. The system is composed of small, simple pieces. Each piece is easy to understand and has clearly defined interfaces. The team can easily change each component on its own, and can deploy and test each component in isolation.

I dig more deeply into implementation principles for this core practice in Chapter 15.

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: defining everything as code, continuously testing and delivering, and building small pieces.

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.

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 test and deliver, build small pieces). I also list several common pitfalls that teams make with dynamic infrastructure.

These principles and pitfalls underlie more specific advice on implementing Infrastructure as Code practices 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 hardware.[1]

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, and 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 change.[2]

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.

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

[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 "Chaos Engineering" on page 379.

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 "Data Continuity in a Changing System" on page 382.

The ability to effortlessly build and rebuild any part of the infrastructure is powerful. It removes risk and fear of making changes, and 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 a 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 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 pipelines to prove that it is correct and 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.[3]

Cattle, Not Pets

"Treat your servers like cattle, not pets," is a popular expression about disposability.[4] 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. The team 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.

[3] See "Cloud Native and Application-Driven Infrastructure" on page 156.

[4] 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.

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 one hundred identical servers than five completely different servers.

The reproducibility principle (see "Principle: Make Everything Reproducible" on page 14) 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 on 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, or 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* by Jim Highsmith, Linda Luu, and David Robinson (Addison-Wesley Professsional), 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. Configuration drift makes it harder to maintain consistent automation.

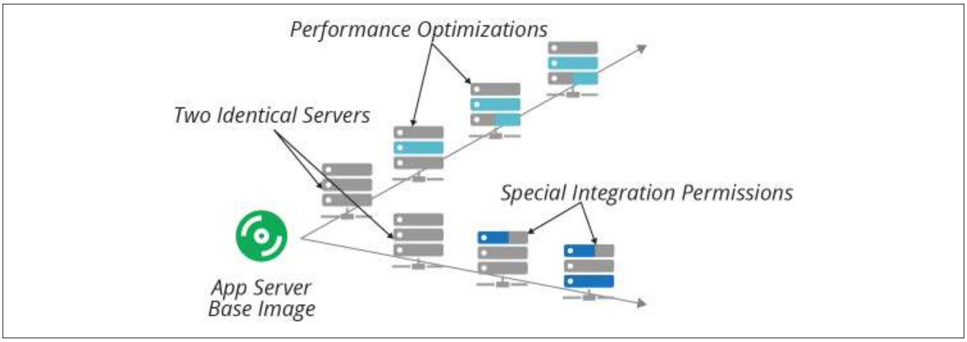


Figure 2-1. Configuration drift is when instances of the same thing become different over time

As an example of how infrastructure can diverge over time, consider the journey of our example team, ShopSpinner.[5]

ShopSpinner runs a separate instance of its application for each store, with each instance configured to use custom branding and product catalog content. In the early days, the ShopSpinner team ran scripts to create a new application server for each new store. The team managed the infrastructure manually or by writing scripts and tweaking them each time it needed to make a change.

One of the customers, Water Works,[6] has far more traffic to its order management application than the others, so the team tweaked the configuration for the Water Works server. The team didn't make the changes to the other customers, because it was busy and it didn't seem necessary.

Later, the ShopSpinner team adopted Servermaker to automate its application server configuration.[7] The team first tested it with the server for Palace Pens,[8] a lower-volume customer, and then rolled it out to the other customers. Unfortunately, the code didn't include the performance optimizations for Water Works, so it stripped those improvements. The Water Works server slowed to a crawl until the team caught and fixed the mistake.

[5] ShopSpinner is a fictional company that allows businesses to set up and run online stores. I'll be using it throughout the book to illustrate concepts and practices.

[6] Water Works sends you bottles of a different craft drinking water every month.

[7] Servermaker is a fictional server configuration tool, similar to Ansible, Chef, or Puppet.

[8] Palace Pens sells the world's finest luxury writing utensils.

The ShopSpinner team overcame this by parameterizing its Servermaker code. It can now set resource levels different for each customer. This way, the team can still apply the same code across every customer, while still optimizing it for each. Chapter 7 describes some patterns and antipatterns for parameterizing infrastructure code for different instances.

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 reapply, 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.

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.

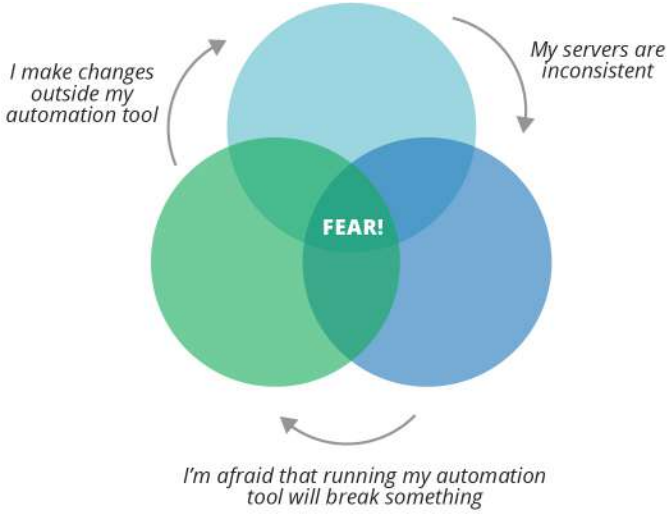

I make changes outside my automation tool
My servers are inconsistent
FEAR!
I'm afraid that running my automation tool will break something

Figure 2-2. The automation fear spiral

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 it.[9] 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.

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.

[9] My colleague Florian Sellmayr says, "If it's worth documenting, it's worth automating."

CHAPTER 3

Infrastructure Platforms

There is a vast landscape of tools relating in some way to modern cloud infrastructure. The question of which technologies to select and how to fit them together can be overwhelming. This chapter presents a model for thinking about the higher-level concerns of your platform, the capabilities it provides, and the infrastructure resources you may assemble to provide those capabilities.

This is not an authoritative model or architecture. You will find your own way to describe the parts of your system. Agreeing on the right place in the diagram for each tool or technology you use is less important than having the conversation.

The purpose of this model is to create a context for discussing the concepts, practices, and approaches in this book. A particular concern is to make sure these discussions are relevant to you regardless of which technology stack, tools, or platforms you use. So the model defines groupings and terminology that later chapters use to describe how, for instance, to provision servers on a virtualization platform like VMware, or an IaaS cloud like AWS.

The Parts of an Infrastructure System

There are many different parts, and many different types of parts, in a modern cloud infrastructure. I find it helpful to group these parts into three platform layers (Figure 3-1):

Applications
: Applications and services provide capabilities to your organization and its users. Everything else in this model exists to enable this layer.

Application runtimes

Application runtimes provide services and capabilities to the application layer. Examples of services and constructs in an application runtime platform include container clusters, serverless environments, application servers, operating systems, and databases. This layer can also be called Platform as a Service (PaaS).

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. This is also known as Infrastructure as a Service (IaaS). I'll elaborate on the resources provided in this layer in "Infrastructure Resources" on page 27.

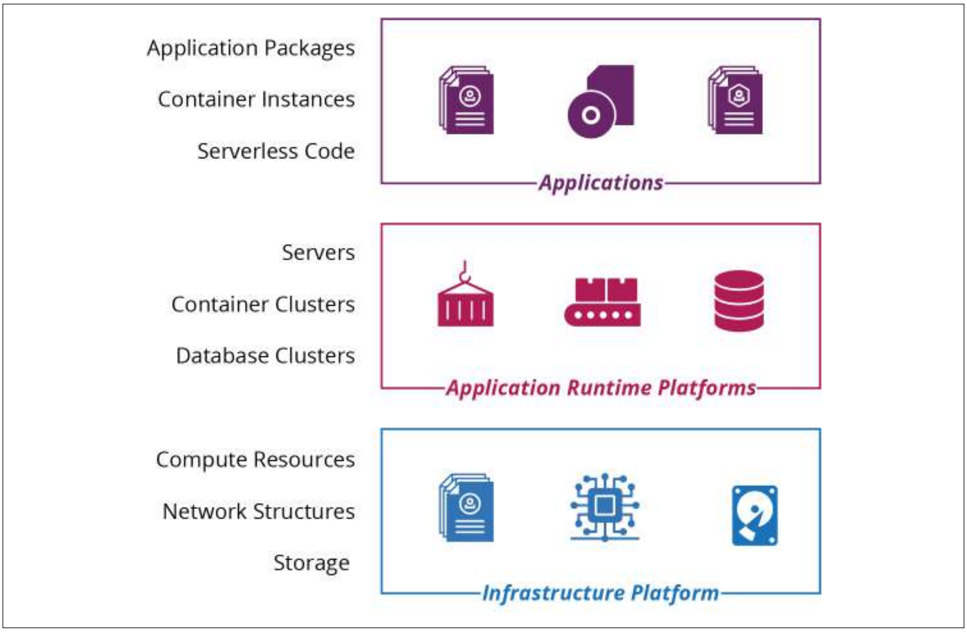


Figure 3-1. Layers of system elements

The premise of this book is using the infrastructure platform layer to assemble infrastructure resources to create the application runtime layer.

Chapter 5 and the other chapters in Part II describe how to use code to build infrastructure stacks. An infrastructure stack is a collection of resources that is defined and managed together using a tool like Ansible, CloudFormation, Pulumi, or Terraform.

Chapter 10 and the other chapters in Part III describe how to use code to define and manage application runtimes. These include servers, clusters, and serverless execution environments.

Infrastructure Platforms

Infrastructure as Code requires a *dynamic* infrastructure platform, something that you can use to provision and change resources on demand with an API. Figure 3-2 highlights the infrastructure platform layer in the platform model. This is the essential definition of a cloud.[1] When I talk about an "infrastructure platform" in this book, you can assume I mean a dynamic, IaaS type of platform.[2]

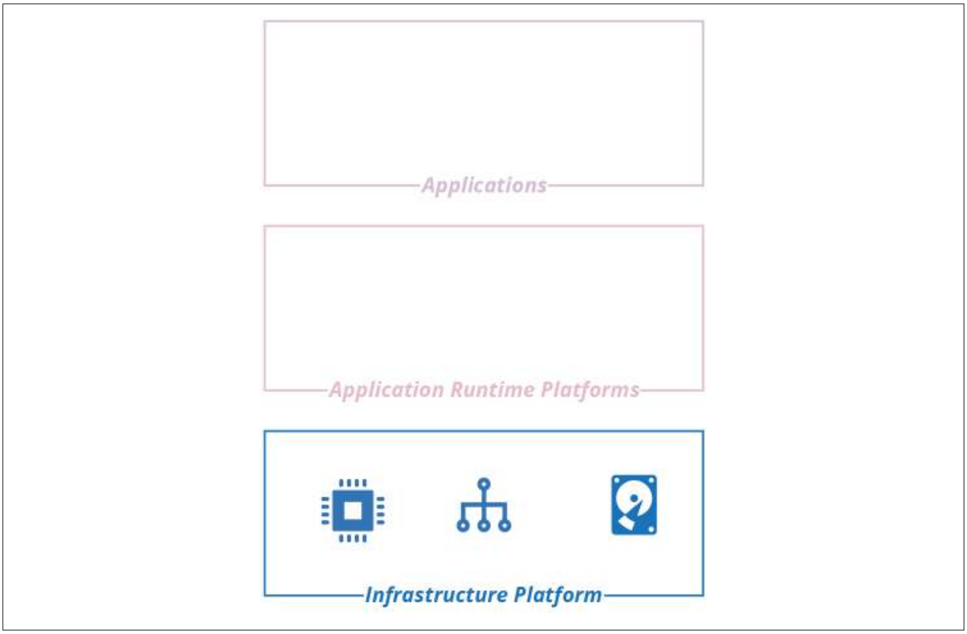


Figure 3-2. The infrastructure platform is the foundation layer of the platform model

[1] The US National Institute of Standards and Technology (NIST) has an excellent definition of cloud computing.

[2] Here's how NIST defines IaaS: "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)."

In the old days—the Iron Age of computing—infrastructure was hardware. Virtualization decoupled systems from the hardware they ran on, and cloud added APIs to manage those virtualized resources.[3] 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. 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, Oracle Cloud, OVH, Scaleway, and Vultr
Private IaaS cloud products	CloudStack, OpenStack, and VMware vCloud
Bare-metal cloud tools	Cobbler, FAI, and Foreman

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.

PaaS

Most public cloud vendors provide resources, or bundles of resources, which combine to provide higher-level services for deploying and managing applications. Examples of hosted PaaS services include Heroku, AWS Elastic BeanStalk, and Azure Devops.[4]

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. This book tends to use generic names that apply to different platforms. Rather than *VPC* and *Subnet*, I use *network address block* and *VLAN*.

[3] Virtualization technology has been around since the 1960s, but emerged into the mainstream world of x86 servers in the 2000s.

[4] 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's talk on DevOps culture.

Multicloud

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.

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. Common compute resource resources include:

Virtual machines (VMs)

The infrastructure platform manages a pool of physical host servers, and runs virtual machine instances on hypervisors across these hosts. Chapter 11 goes into much more detail about how to provision, configure, and manage servers.

Physical servers

Also called *bare metal*, the platform dynamically provisions physical servers on demand. "Using a Networked Provisioning Tool to Build a Server" on page 182 describes a mechanism for automatically provisioning physical servers as a bare-metal cloud.

Server clusters

A server cluster is a pool of server instances—either virtual machines or physical servers—that the infrastructure platform provisions and manages as a group. Examples include AWS Auto Scaling Group (ASG), Azure virtual machine scale set, and Google Managed Instance Groups (MIGs).

Containers

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. Many application runtime platforms are built around containerization. Chapter 10 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.

Application hosting clusters

An application hosting cluster is a pool of servers onto which the platform deploys and manages multiple applications.[5] Examples include Amazon Elastic Container Services (ECS), Amazon Elastic Container Service for Kubernetes (EKS), Azure Kubernetes Service (AKS), and Google Kubernetes Engine (GKE). You can also deploy and manage an application hosting package on an infrastructure platform. See "Application Cluster Solutions" on page 230 for more details.

FaaS serverless code runtimes

An infrastructure platform executes serverless Function as a Service (FaaS) code on demand, in response to an event or schedule, and then terminates it after it has completed its action. See "Infrastructure for FaaS Serverless" on page 246 for more. Infrastructure platform serverless offerings include AWS Lambda, Azure Functions, and Google Cloud Functions.

Serverless Beyond Code

The serverless concept goes beyond running code. Amazon DynamoDB and Azure Cosmos DB are serverless databases: the servers involved in hosting the database are transparent to you, unlike more traditional DBaaS offerings like AWS RDS.

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.

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.

[5] An application hosting cluster should not be confused with a PaaS. This type of cluster manages the provision of compute resources for applications, which is one of the core functions of a PaaS. But a full PaaS provides a variety of services beyond compute.

Typical storage resources found on infrastructure platforms are:

Block storage (virtual disk volumes)
: You can attach a block storage volume to a single 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.

Object storage
: You can use object storage to store and access files from multiple locations. Amazon's S3, Azure Block Blobs, Google Cloud Storage, and OpenStack Swift are all examples. Object storage is usually cheaper and more reliable than block storage, but with higher latency.

Networked filesystems (shared network volumes)
: Many cloud platforms provide shared storage volumes that can be mounted on multiple compute instances using standard protocols, such as NFS, AFS, and SMB/CIFS.[6]

Structured data storage
: Most infrastructure platforms provide a managed Database as a Service (DBaaS) that you can define and manage as code. These may be a standard commercial or open source database application, such as MySQL, Postgres, or SQL Server. Or it could be a custom structured data storage service; for example, a key-value store or formatted document stores for JSON or XML content. The major cloud vendors also 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.

Secrets management
: Any storage resource can be encrypted so you can store passwords, keys, and other information that 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. See "Handling Secrets as Parameters" on page 102 for techniques for managing secrets and infrastructure code.

[6] Network File System, Andrew File System, and Server Message Block, respectively.

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 (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 model secures every service, application, and other resource in a system at the lowest level.[7] 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 nontrivial 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.
- The security relationships between elements of the system are visible, which enables validation, auditing, and reporting.

[7] Google documents its approach to zero-trust (also known as perimeterless) with the BeyondCorp security model.

Typical of networking constructs and services an infrastructure platform provides include:

Network address blocks
: Network address blocks are a fundamental structure for grouping resources to control routing of traffic between them. 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 or VLANs. A certain networking structure, such as AWS subnets, may be associated with physical locations, such as a data center, which you can use to manage availability.

Names, such as DNS entries
: Names are mapped to lower-level network addresses, usually IP addresses.

Routes
: Configure what traffic is allowed between and within address blocks.

Gateways
: May be needed to direct traffic in and out of blocks.

Load balancing rules
: Forward connections coming into a single address to a pool of resources.

Proxies
: Accept connections and use rules to transform or route them.

API gateways
: A proxy, usually for HTTP/S connections, which can carry out activities to handle noncore aspects of APIs, such as authentication and rate throttling. See also "Service Mesh" on page 244.

VPNs (virtual private networks)
: Connect different address blocks across locations so that they appear to be part of a single network.

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

Network access rules (firewall rules)
: Rules that restrict or allow traffic between network locations.

Asynchronous messaging
: Queues for processes to send and receive messages.

Cache

A cache distributes data across network locations 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.

Service mesh

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" on page 23.

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 Network Adminstration* (O'Reilly).

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.

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 test and deliver all work in progress, and build small, simple pieces.

This chapter delves into the first of these core practices, starting with the banal questions. Why would you want to define your Infrastructure as Code? What types of things can and should you define as code?

At first glance, "define everything as code" might seem obvious in the context of this book. But the characteristics of different types of languages are relevant to the following chapters. In particular, Chapter 5 describes using declarative languages to define either low-level ("Low-Level Infrastructure Languages" on page 54) or high-level stacks ("High-Level Infrastructure Languages" on page 55), and Chapter 16 explains when declarative or programmatic code is most appropriate for creating reusable code modules and libraries.

Why You Should Define Your Infrastructure as Code

There are simpler ways to provision infrastructure than writing a bunch of code and then feeding it into 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, including reusability, consistency, and transparency (see "Core Practice: Define Everything as Code" on page 10).

Implementing and managing your systems as code enables you to leverage speed to improve quality. It's the secret sauce that powers high performance as measured by the four key metrics (see "The Four Key Metrics" on page 9).

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 should define as code include:

- An infrastructure stack, which is a collection of elements provisioned from an infrastructure cloud platform. See Chapter 3 for more about infrastructure platforms, and Chapter 5 for an introduction to the infrastructure stack concept.
- Elements of a server's configuration, such as packages, files, user accounts, and services (Chapter 11).
- A server role is a collection of server elements that are applied together to a single server instance ("Server Roles" on page 175).
- A server image definition generates an image for building multiple server instances ("Tools for Building Server Images" on page 209).
- An application package defines how to build a deployable application artifact, including containers (Chapter 10).
- Configuration and scripts for delivery services, which include pipelines and deployment ("Delivery Pipeline Software and Services" on page 123).
- Configuration for operations services, such as monitoring checks.
- Validation rules, which include both automated tests and compliance rules (Chapter 8).

Choose Tools with Externalized Configuration

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.

Noncode infrastructure automation tools store infrastructure definitions as data that you can't directly access. 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 interfaces.

The issue with these closed-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 CI 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.

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 nontrivial systems, developers find that keeping their source code in external files is more powerful.

It is challenging to use Agile engineering practices such as TDD, CI, and CD with closed-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.

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 why.[1] This history is invaluable when debugging problems.

[1] Context about why depends on people writing useful commit messages.

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.

One thing that you should not put into source control is unencrypted secrets, such as passwords and keys. Even if your source code repository is private, history and revisions of code are too easily leaked. Secrets leaked from source code are one of the most common causes of security breaches. See "Handling Secrets as Parameters" on page 102 for better ways to manage secrets.

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 "Domain-Specific Infrastructure Languages" on page 44) 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, using the same declarative DSL model. Declarative languages simplified infrastructure code, by separating the definition of what infrastructure you want from how to implement it.

Recently, there is a trend of new infrastructure tools that use existing general-purpose programming languages to define infrastructure.[2] Pulumi and the AWS CDK (Cloud Development Kit) support languages like Typescript, Python, and Java. These tools have emerged to address some of the limitations of declarative languages.

Mixing Declarative and Imperative Code

Imperative code is a set of instructions that specifies how to make a thing happen. *Declarative code* specifies what you want, without specifying how to make it happen.[3]

Too much infrastructure code in production today suffers from mixing declarative and imperative code. I believe it's an error to insist that one or the other of these two language paradigms should be used for all infrastructure code.

An infrastructure codebase involves many different concerns, from defining infrastructure resources, to configuring different instances of otherwise similar resources, to orchestrating the provisioning of multiple interdependent pieces of a system. Some of these concerns can be expressed most simply using a declarative language. Some concerns are more complex, and are better handled with an imperative language.

As practitioners of the still-young field of infrastructure code, we are still exploring where to draw boundaries between these concerns. Mixing concerns can lead to code that mixes language paradigms. One failure mode is extending a declarative syntax like YAML to add conditionals and loops. The second failure mode is embedding simple configuration data ("2GB RAM") into procedural code, mixing *what* you want with *how* to implement it.

In relevant parts of this book I point out where I believe some of the different concerns may be, and where I think one or another language paradigm may be most appropriate. But our field is still evolving. Much of my advice will be wrong or incomplete. So, my intention is to encourage you, the reader, to think about these questions and help us all to discover what works best.

Infrastructure Scripting

Before standard tools appeared for provisioning cloud infrastructure declaratively, we wrote scripts in general-purpose, procedural languages. Our scripts typically used an SDK (software development kit) to interact with the cloud provider's API.

[2] "Recently" as I write this in mid-2020.

[3] I sometimes refer to imperative code or languages as *programmable* even though it's not the most accurate term, because it's more intuitive than "imperative."

Example 4-1 uses pseudocode, and 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
  }
  if(app_server.ok != true) {
    throw ServerFailedError
  }
  app_server.provision(
    provisioner: servermaker,
    role: tomcat_server
  )
}
```

This script mixes *what* to create and *how* to create it. It specifies attributes of the server, including the CPU and memory resources to provide it, what OS image to start from, and what 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 applying configuration to it.

This example code 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 multipurpose 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.

Declarative Infrastructure Languages

Many infrastructure code tools, including Ansible, Chef, CloudFormation, Puppet, and Terraform, use declarative languages. Your code defines your desired state for your infrastructure, such as which packages and user accounts should be on a server, or how much RAM and CPU resources it should have. The tool handles the logic of how to make that desired state come about.

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 language.[4]

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 the server provisioner. The tool that you run to apply the code takes care of that. The tool also checks the current attributes of infrastructure against what is declared, and works out what changes to make to bring the infrastructure in line. So to increase the RAM of the application server in this example, you edit the file and rerun the tool.

Declarative infrastructure tools like Terraform and Chef separate *what* you want from *how* to create it. As a result, your code is cleaner and more direct. People sometimes describe declarative infrastructure code as being closer to configuration than to programming.

[4] I use this pseudocode language to illustrate the concepts I'm trying to explain, without tying them to any specific tool.

Is Declarative Code Real Code?

Some people dismiss declarative languages as being mere configuration, rather than "real" code.

I use the word *code* to refer to both declarative and imperative languages. When I need to distinguish between the two, I specifically say either "declarative" or "programmable," or some variation.

I don't find the debate about whether a coding language must be Turing-complete to be useful. I even find regular expressions useful for some purposes, and they aren't Turing-complete either. So, my devotion to the purity of "real" programming may be lacking.

Idempotency

Continuously applying code is an important practice for maintaining the consistency and control of your infrastructure code, as described in "Apply Code Continuously" on page 350. This practice involves repeatedly reapplying code to infrastructure to prevent drift. Code must be *idempotent* to be safely applied continuously.

You can rerun idempotent code any number of times without changing the output or outcome. 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. 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.

Programmable, Imperative Infrastructure Languages

Declarative code is fine when you always want the same outcome. But there are situations where you want different results depending on the circumstances. For example, the following code creates a set of VLANs. The ShopSpinner team's cloud provider has a different number of data centers in each country, and the team wants its code to create one VLAN in each data center. So the code needs to dynamically discover how many data centers there are, and create a VLAN in each one:

```
this_country = getArgument("country")
data_centers = CloudApi.find_data_centers(country: this_country)
full_ip_range = 10.2.0.0/16

vlan_number = 0
for $DATA_CENTER in data_centers {
  vlan = CloudApi.vlan.apply(
    name: "public_vlan_${DATA_CENTER.name}"
    data_center: $DATA_CENTER.id
    ip_range: Networking.subrange(
        full_ip_range,
        data_centers.howmany,
        data_centers.howmany++
    )
  )
}
```

The code also assigns an IP range for each VLAN, using a fictional but useful method called `Networking.subrange()`. This method takes the address space declared in `full_ip_range`, divides it into a number of smaller address spaces based on the value of `data_centers.howmany`, and returns one of those address spaces, the one indexed by the `data_centers.howmany` variable.

This type of logic can't be expressed using declarative code, so most declarative infrastructure tools extend their languages to add imperative programming capability. For example, Ansible adds loops and conditionals to YAML. Terraform's HCL configuration language is often described as declarative, but it actually combines three sublanguages. One of these is expressions, which includes conditionals and loops.

Newer tools, such as Pulumi and the AWS CDK, return to using programmatic languages for infrastructure. Much of their appeal is their support for general-purpose programming languages (as discussed in "General-Purpose Languages Versus DSLs for Infrastructure" on page 46). But they are also valuable for implementing more dynamic infrastructure code.

Rather than seeing either declarative or imperative infrastructure languages as the correct paradigm, we should look at which types of concerns each one is most suited for.

Declarative Versus Imperative Languages for Infrastructure

Declarative code is useful for defining the desired state of a system, particularly when there isn't much variation in the outcomes you want. It's common to define the shape of some infrastructure that you would like to repeat with a high level of consistency.

For example, you normally want all of the environments supporting a release process to be nearly identical (see "Delivery Environments" on page 66). So declarative code is good for defining reusable environments, or parts of environments (per the reusable stack pattern discussed in "Pattern: Reusable Stack" on page 72). You can even support limited variations between instances of infrastructure defined with declarative code using instance configuration parameters, as described in Chapter 7.

However, sometimes you want to write reusable, sharable code that can produce different outcomes depending on the situation. For example, the ShopSpinner team writes code that can build infrastructure for different application servers. Some of these servers are public-facing, so they need appropriate gateways, firewall rules, routes, and logging. Other servers are internally facing, so they have different connectivity and security requirements. The infrastructure might also differ for applications that use messaging, data storage, and other optional elements.

As declarative code supports more complex variations, it involves increasing amounts of logic. At some point, you should question why you are writing complex logic in YAML, JSON, XML, or some other declarative language.

So programmable, imperative languages are more appropriate for building libraries and abstraction layers, as discussed in more detail in Chapter 16. And these languages tend to have better support for writing, testing, and managing libraries.

Domain-Specific Infrastructure Languages

In addition to being declarative, many infrastructure tools use their own DSL, or domain-specific language.[5]

A DSL is a language designed to model a specific domain, in our case infrastructure. This makes it easier to write code, and makes the code easier to understand, because it closely maps the things you're defining.

[5] 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).

For example, Ansible, Chef, and Puppet each have a DSL for configuring servers. Their languages provide constructs for concepts like packages, files, services, and user accounts. A pseudocode example of a server configuration DSL is:

```
package: jdk
package: tomcat

service: tomcat
  port: 8443
  user: tomcat
  group: tomcat

file: /var/lib/tomcat/server.conf
  owner: tomcat
  group: tomcat
  mode: 0644
  contents: $TEMPLATE(/src/appserver/tomcat/server.conf.template)
```

This code ensures that two software packages are installed, `jdk` and `tomcat`. It defines a service that should be running, including the port it listens to and the user and group it should run as. Finally, the code defines that a server configuration file should be created from a template file.

The example code is pretty easy for someone with systems administration knowledge to understand, even if they don't know the specific tool or language. Chapter 11 discusses how to use server configuration languages.

Many stack management tools also use DSLs, including Terraform and CloudFormation. They expose concepts from their own domain, infrastructure platforms, so that you can directly write code that refers to virtual machines, disk volumes, and network routes. See Chapter 5 for more on using these languages and tools.

Other infrastructure DSLs model application runtime platform concepts. These model systems like application clusters, service meshes, or applications. Examples include Helm charts and CloudFoundry app manifests.

Many infrastructure DSLs are built as extensions of existing markup languages such as YAML (Ansible, CloudFormation, anything related to Kubernetes) and JSON (Packer, CloudFormation). Some are internal DSLs, written as a subset (or superset) of a general-purpose programming language. Chef is an example of an internal DSL, written as Ruby code. Others are external DSLs, which are interpreted by code written in a different language. Terraform HCL is an external DSL; the code is not related to the Go language its interpreter is written in.

General-Purpose Languages Versus DSLs for Infrastructure

Most infrastructure DSLs are declarative languages rather than imperative languages. An internal DSL like Chef is an exception, although even Chef is primarily declarative.[6]

One of the biggest advantages of using a general-purpose programming language, such as JavaScript, Python, Ruby, or TypeScript, is the ecosystem of tools. These languages are very well supported by IDEs,[7] with powerful productivity features like syntax highlighting and code refactoring. Testing support is an especially useful part of a programming language's ecosystem.

Many infrastructure testing tools exist, some of which are listed in "Verification: Making Assertions About Infrastructure Resources" on page 135 and "Testing Server Code" on page 176. But few of these integrate with languages to support unit testing. As we'll discuss in "Challenge: Tests for Declarative Code Often Have Low Value" on page 110, this may not be an issue for declarative code. But for code that produces more variable outputs, such as libraries and abstraction layers, unit testing is essential.

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.

Separate Declarative and Imperative Code

Code that mixes both declarative and imperative code is a design smell that suggests you should split the code into separate concerns.[8]

[6] You can mix imperative Ruby code in your Chef recipes, but this gets messy. Chef interprets recipes in two phases, first compiling the Ruby code, then running the code to apply changes to the server. Procedural code is normally executed in the compile phase. This makes Chef code that mixes procedural and imperative code hard to understand. On the other hand, imperative code is useful when writing Chef providers, which are a type of library. This reinforces the idea that an imperative language works well for library code, and a declarative language works well for defining infrastructure.

[7] *Integrated Development Environment*, a specialized editor for programming languages.

[8] The term *design smell* derives from code smell. 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 imperative 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.

Treat Infrastructure Code Like Real Code

Many infrastructure codebases evolve from configuration files and utility scripts into an unmanageable mess. 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. To keep an infrastructure codebase maintainable, you need to treat it as a first-class concern.

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.

Chapter 15 describes how to apply various software design principles to infrastructure, such as improving cohesion and reducing coupling. Chapter 18 explains ways to organize and manage infrastructure codebases to make them easier to work with.

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 and parameter references from code. You can put this in a change management pipeline to update documentation every time someone makes a change to the code.

Conclusion

This chapter detailed the core practice of defining your system as code. This included looking at why you should define things as code, and what parts of your system you can define as code. The core of the chapter explored different infrastructure language paradigms. This might seem like an abstract topic. But using the right languages in the right ways is a crucial challenge for creating effective infrastructure, a challenge that the industry hasn't yet solved. So the question of which type of language to use in different parts of our system, and the consequences of those decisions, is a theme that will reappear throughout this book.

PART II

Working with Infrastructure Stacks

CHAPTER 5

Building Infrastructure Stacks as Code

This chapter combines Chapters 3 and 4 by describing ways to use code to manage infrastructure resources provided from an infrastructure platform.

The concept that I use to talk about doing this is the infrastructure stack. A stack is a collection of infrastructure resources defined and changed together. The resources in a stack are provisioned together to create a stack instance, using a stack management tool. Changes are made to stack code and applied to an instance using the same tool.

This chapter describes patterns for grouping infrastructure resources in stacks.

What Is an Infrastructure Stack?

An *infrastructure stack* is a collection of infrastructure resources that you define, provision, and update as a unit (Figure 5-1).

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" on page 28), disk volume ("Storage Resources" on page 29), and a subnet ("Network Resources" on page 31).

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.

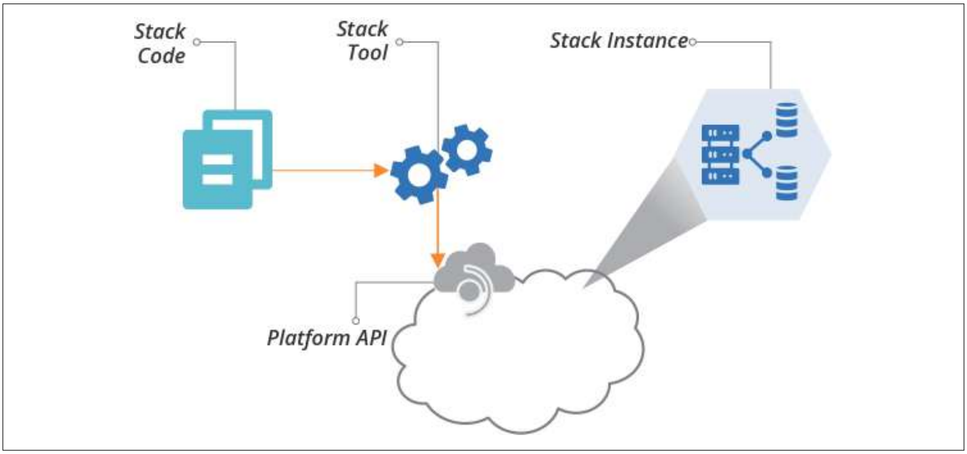


Figure 5-1. An infrastructure stack is a collection of infrastructure elements managed as a group

Examples of stack management tools include:

- HashiCorp Terraform
- AWS CloudFormation
- Azure Resource Manager
- Google Cloud Deployment Manager
- OpenStack Heat
- Pulumi
- Bosh

Some server configuration tools (which I'll talk about much more in Chapter 11) have extensions to work with infrastructure stacks. Examples of these are Ansible Cloud Modules, Chef Provisioning (now end-of-lifed), 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. 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 for the fictional *Stackmaker* tool.

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-waterworks-${environment}
  source_image: shopspinner-base-appserver
  memory: 4GB
  provision:
    tool: servermaker
    parameters:
      maker_server: maker.shopspinner.xyz
      role: appserver
      environment: ${environment}
```

This stack defines an application server instance, created from a server image called `shopspinner-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.

Low-Level Infrastructure Languages

Most of the popular stack management tool languages are low-level infrastructure languages. That is, the language directly exposes the infrastructure resources provided by the infrastructure platform you're working with (the types of resources listed in "Infrastructure Resources" on page 27).

It's your job, as the infrastructure coder, to write the code that wires these resources together into something useful, such as Example 5-3.

Example 5-3. Example of low-level infrastructure stack code

```
address_block:
  name: application_network_tier
  address_range: 10.1.0.0/24"
  vlans:
  - appserver_vlan_A
      address_range: 10.1.0.0/16

virtual_machine:
  name: shopspinner_appserver_A
  vlan: application_network_tier.appserver_vlan_A

gateway:
  name: public_internet_gateway
  address_block: application_network_tier

inbound_route:
  gateway: public_internet_gateway
  public_ip: 192.168.99.99
  incoming_port: 443
  destination:
    virtual_machine: shopspinner_appserver_A
    port: 8443
```

This contrived and simplified pseudocode example defines a virtual machine, an address block and VLAN, and an internet gateway. It then wires them together and defines an inbound connection that routes connections coming into *https://192.168.99.99* to port 8443 on the virtual machine.[1]

The platform may itself provide a higher layer of abstraction; for example, providing an application hosting cluster. The cluster element provided by the platform might automatically provision server instances and network routes. But low-level infrastructure code maps directly to the resources and options exposed by the platform API.

High-Level Infrastructure Languages

A high-level infrastructure language defines entities that don't directly map to resources provided by the underlying platform. For example, a high-level code version of Example 5-3 might declare the basics of the application server, as shown in Example 5-4.

[1] No, that's not actually a public IP address.

Example 5-4. Example of high-level infrastructure stack code

```
application_server:
  public_ip: 192.168.99.99
```

In this example, applying the code either provisions the networking and server resources from the previous example, or it discovers existing resources to use. The tool or library that this code invokes decides what values to use for the network ports and VLAN, and how to build the virtual server.

Many application hosting solutions, such as a PaaS platform or packaged cluster (see "Packaged Cluster Distribution" on page 231), provide this level of abstraction. You write a deployment descriptor for your application, and the platform allocates the infrastructure resources to deploy it onto.

In other cases, you may build your own abstraction layer by writing libraries or modules. See Chapter 16 for more on this.

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 structuring infrastructure stacks.

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 *micro stack* 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 (see Figure 5-2).

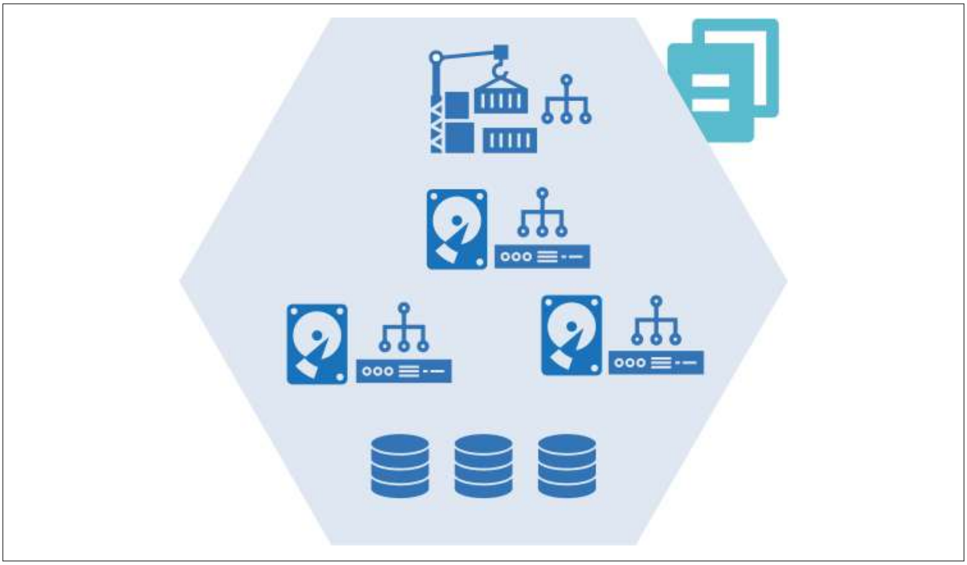


Figure 5-2. A monolithic stack is an infrastructure stack that includes too many elements, making it difficult to maintain

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.

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. For a modestly sized collection of infrastructure elements, a monolithic stack might make sense. But more often, a monolithic stack organically grows out of control.

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 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 immediate *blast radius* is the scope of code that the command to apply your change includes.[2] For example, when you run `terraform apply`, the direct blast radius includes all of the code in your project. The indirect blast radius includes other elements of the system that depend on the resources in your direct blast radius, and which might be affected by breaking those resources.

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 (see "Pattern: Micro Stack" on page 62), which aims to keep stacks small so that they are easier to maintain and improve. A monolithic stack may be an application group stack (see "Pattern: Application Group Stack" on page 59) 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.

[2] Charity Majors popularized the term *blast radius* in the context of Infrastructure as Code. Her post "Terraform, VPC, and Why You Want a tfstate File Per env" describes the scope of potential damage that a given change could inflict on your system.

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.

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 managed as a group, as shown in Figure 5-3.

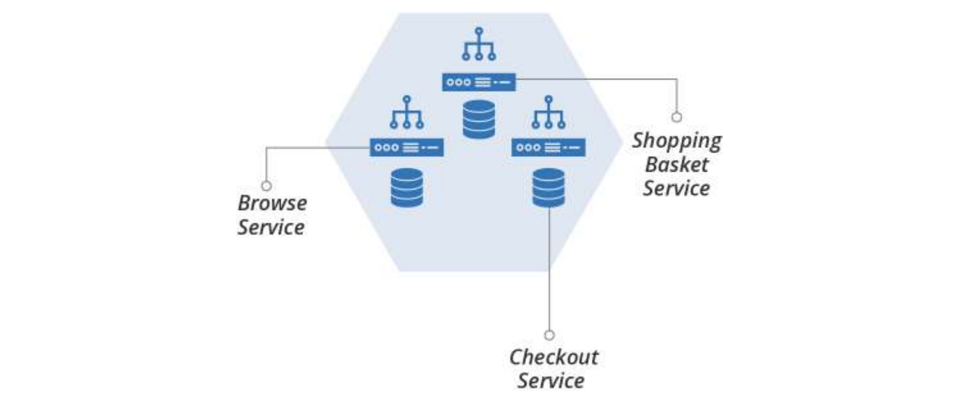


Figure 5-3. An application group stack hosts multiple processes in a single instance of the stack

For example, ShopSpinner's product application stack includes separate services for browsing products, searching for products, and managing a shopping basket. The servers and other infrastructure for all of these are combined in a single stack instance.

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.

Multiservice 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 (see "Antipattern: Monolithic Stack" on page 56). In the other direction, breaking each service in an application group stack into a separate stack creates a service stack.

Pattern: Service Stack

A service stack manages the infrastructure for each deployable application component in a separate infrastructure stack (see Figure 5-4).

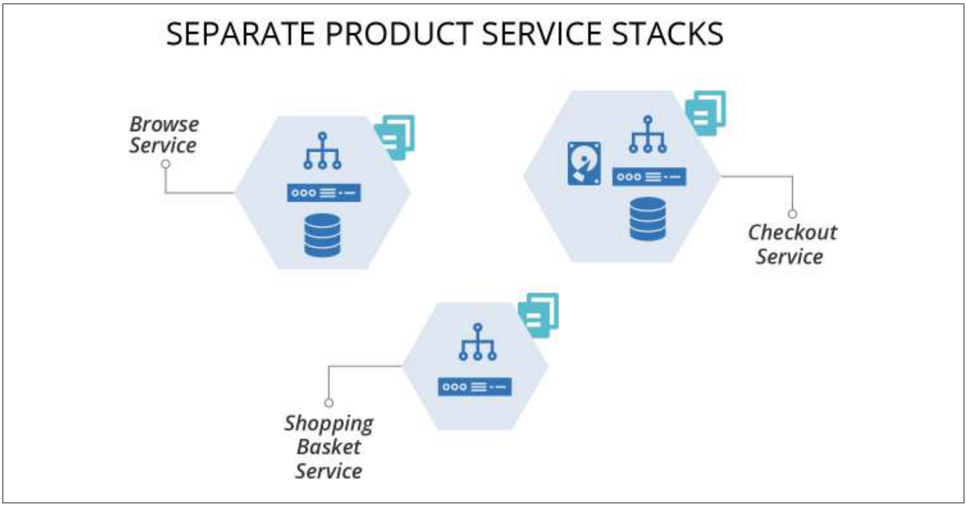


Figure 5-4. A service stack manages the infrastructure for each deployable application component in a separate infrastructure stack

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 architectures.[3] They also help organizations with autonomous teams to ensure each team owns its infrastructure.[4]

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 16).

[3] See "Microservices" by James Lewis.

[4] See "The Art of Building Autonomous Teams" by John Ferguson Smart, among many other references.

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 Chapter 17.

Related patterns

The service stack pattern falls between an application group stack ("Pattern: Application Group Stack" on page 59), which has multiple applications in a single stack, and a 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 (see Figure 5-5).

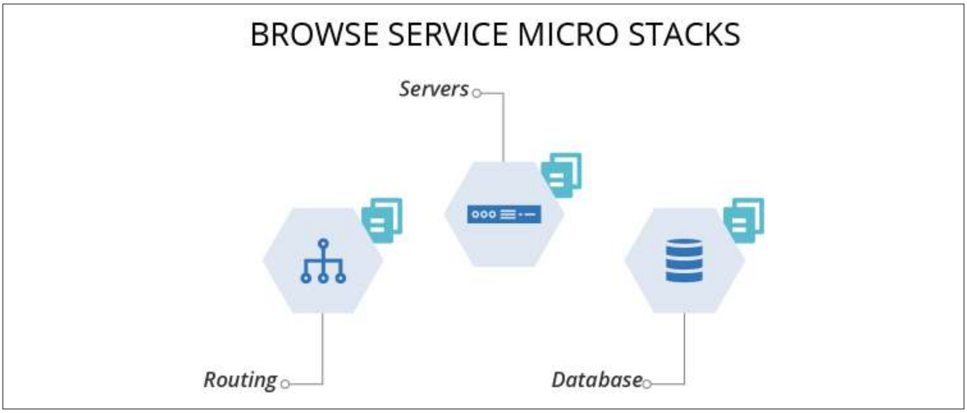


Figure 5-5. Micro stacks divide the infrastructure for a single service across multiple stacks

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 that make them easier to manage separately. For instance, some methods for managing server instances involve frequently destroying and rebuilding them.[5] 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 life cycles, with the server stack being rebuilt much more often than the data stack.

Consequences

Although smaller stacks are themselves simpler, having more moving parts adds complexity. Chapter 17 describes techniques for handling integration between multiple stacks.

Implementation

Adding a new micro stack 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 Chapter 17.

Related Patterns

Micro stacks are the opposite end of the spectrum from a monolithic stack (see "Antipattern: Monolithic Stack" on page 56), 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.

[5] For example, immutable servers (see "Pattern: Immutable Server" on page 195).

CHAPTER 6

Building Environments with Stacks

Chapter 5 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? This chapter explains that maybe it is, but maybe it isn't.

An environment is a collection of software and infrastructure resources organized around a particular purpose, such as to support a testing phase, or to provide service in a geographical region. A stack, or set of stacks, is a means of defining and managing a collection of infrastructure resources. So you use a stack, or multiple stacks, to implement an environment. 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 delivery process, and the other is to run multiple production instances of the system.

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 (Figure 6-1).

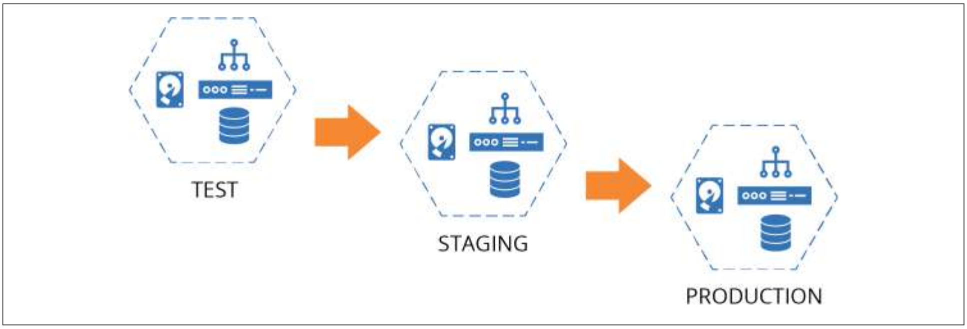


Figure 6-1. ShopSpinner delivery environments

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 "Continuity" on page 376 for continuity strategies that leverage Infrastructure as Code.

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.

ShopSpinner runs a separate application server for each of its ecommerce customers. As it expands to support customers in North America, Europe, and South Asia, it decides to create a separate environment for each of these regions (see Figure 6-2).

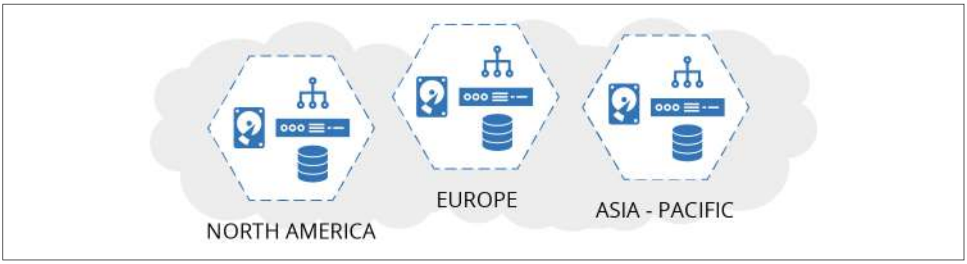


Figure 6-2. ShopSpinner regional environments

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

Later, ShopSpinner lands a contract with a pharmaceutical store chain named The Medicine Barn. The Medicine Barn needs to host its customer data separately from other companies for regulatory reasons. So ShopSpinner's team offers to run a completely separate environment dedicated to The Medicine Barn 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 delivery 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 delivery in Chapter 8.

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" on page 56. I'll explain what this looks like for multiple environments in "Building Environments with Multiple Stacks" on page 74.

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 (Figure 6-3).

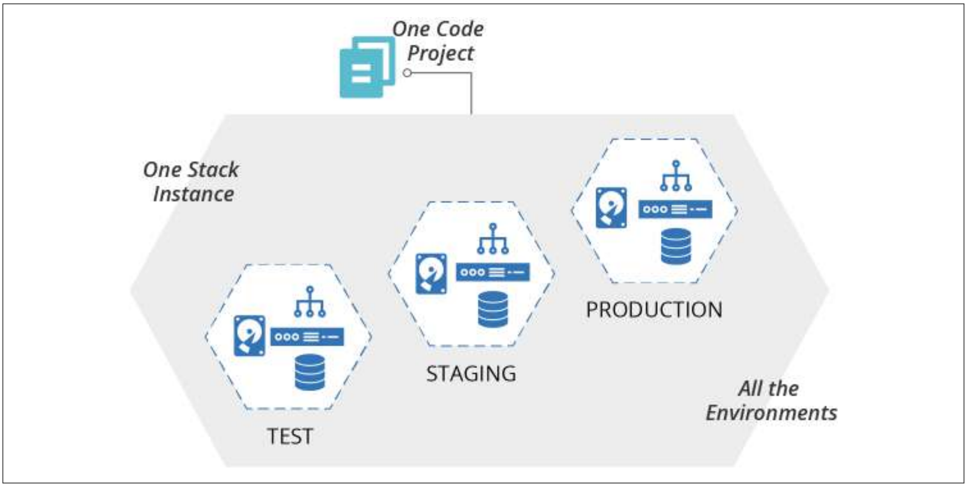


Figure 6-3. A multiple-environment stack manages the infrastructure for multiple environments as a single stack instance

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 vulnerable.[1]

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 (see "Antipattern: Copy-Paste Environments" on page 70), where each environment is a separate stack project, although this is considered an antipattern.

A better approach is the reusable stack pattern (see "Pattern: Reusable Stack" on page 72). A single project is used to define the generic structure for an environment and is

[1] Charity Majors shared her painful experiences of working with a multiple-environment stack in a blog post.

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 (Figure 6-4). Changes are made by editing the code in one environment and then copying the changes into each of the other environments in turn.

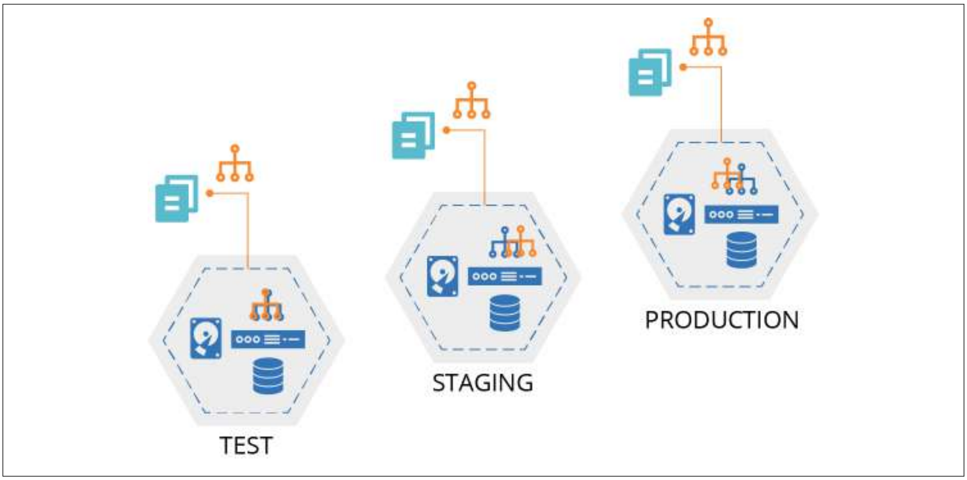


Figure 6-4. A copy-paste environment has a separate copy of the source code project for each instance

Motivation

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

Applicability

Copy-paste environments might be appropriate if 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 (see "Configuration Drift" on page 17). 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.

A copy-paste environment might be consistent when it's first set up, but variations creep in over time.

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

Environment branches (see "Delivering code from a source code repository" on page 325) may be considered a form of copy-paste environments. Each branch has a copy of the code, and people copy code between branches by merging. Continuously applying code (see "Apply Code Continuously" on page 350) may avoid the pitfalls of copy-paste, because it guarantees the code isn't modified from one environment to the next. Editing the code as a part of merging it to an environment branch creates the hazards of the copy-paste antipattern.

The wrapper stack pattern (see "Pattern: Wrapper Stack" on page 90) is also similar to copy-paste environments. A wrapper stack uses a separate stack project for each environment in order to set configuration parameters. But the code for the stack is implemented in stack components, such as reusable module code. That code itself is not copied and pasted for each environment, but promoted much like a reusable stack. However, if people add more than basic stack instance parameters to the wrapper stack projects, it can devolve into the copy-paste environment antipattern.

In cases where stack instances are meant to represent the same stack, the reusable stack pattern is usually more appropriate.

Pattern: Reusable Stack

A reusable stack is an infrastructure source code project that is used to create multiple instances of a stack (Figure 6-5).

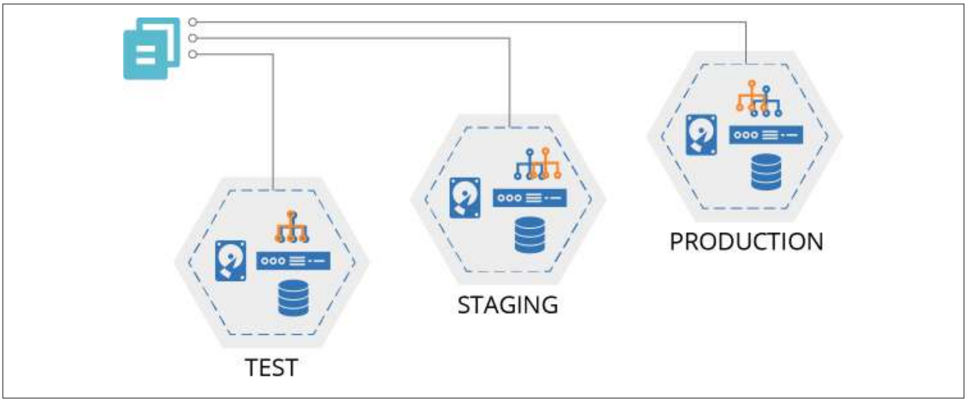


Figure 6-5. Multiple stack instances created from a single reusable stack project

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.

As an example, the ShopSpinner team extracted common code from different stack projects that each use an application server. Team members 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 ShopSpinner was 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 members use 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 7).

You should test your stack project code before you apply changes to business-critical infrastructure. I'll spend multiple chapters on this, including Chapters 8 and 9.

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 (see "Antipattern: Copy-Paste Environments" on page 70), making it easier to keep multiple instances consistent.

The wrapper stack pattern (see "Pattern: Wrapper Stack" on page 90) uses stack components to define a reusable stack, but uses a different stack project to set parameter values for each instance.

Building Environments with Multiple Stacks

The reusable stack pattern describes an approach for implementing multiple environments. In Chapter 5 I described different ways of structuring a system's infrastructure across multiple stacks (see "Patterns and Antipatterns for Structuring Stacks" on page 56). 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 6-5).

But you should split larger systems into multiple stacks. For example, if you follow the service stack pattern ("Pattern: Service Stack" on page 60) you have a separate stack for each service, as shown in Figure 6-6.

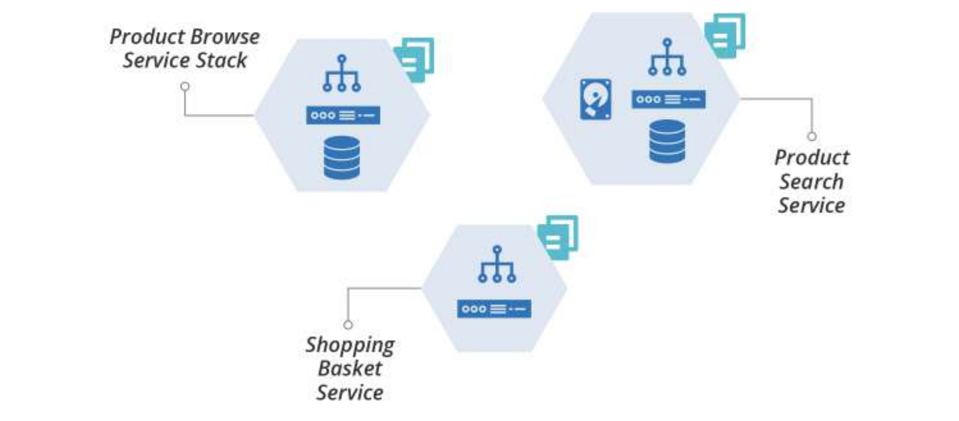


Figure 6-6. Example using a separate infrastructure stack for each service

To create multiple environments, you provision an instance of each service stack for each environment, as shown in Figure 6-7.

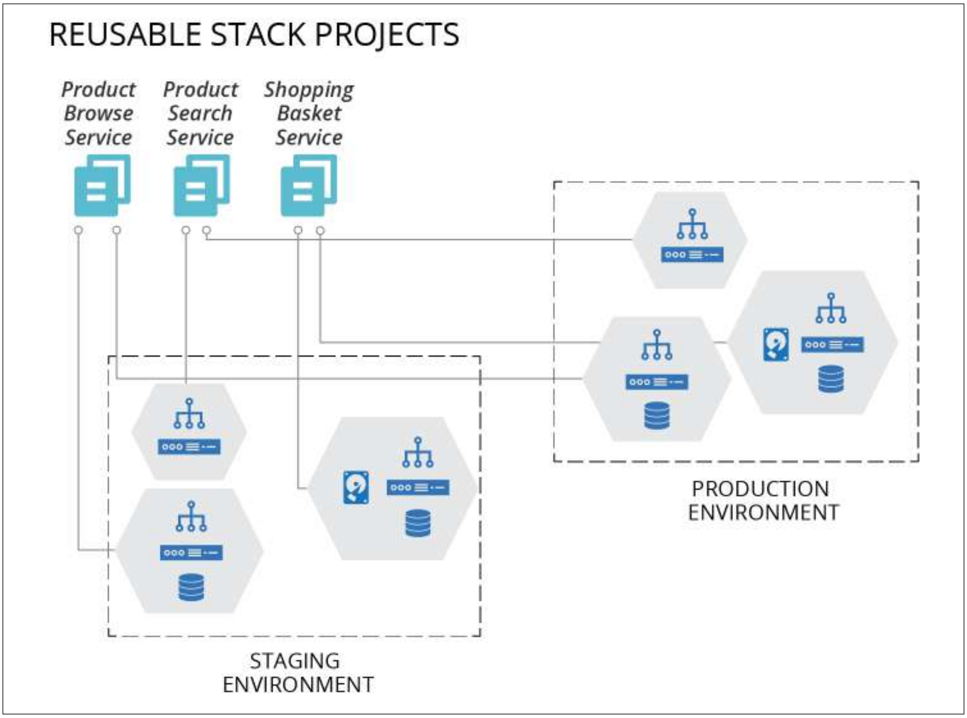


Figure 6-7. Using multiple stacks to build each environment

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
```

Chapter 15 describes strategies for splitting systems into multiple stacks, and Chapter 17 discusses how to integrate infrastructure across stacks.

Conclusion

Reusable stacks should be the workhorse pattern for most teams who need to manage large infrastructures. A stack is a useful unit for testing and delivering changes. It provides assurance that each instance of an environment is defined and built consistently. The comprehensiveness of a stack, rather than modules, as a unit of change enhances the ability to easily deliver changes quickly and frequently.

CHAPTER 7

Configuring Stack Instances

Using a single stack code project makes it easier to maintain multiple consistent instances of infrastructure, as described in "Pattern: Reusable Stack" on page 72. However, you often need to customize different instances of a stack. For example, you might size a cluster with fewer servers in a development environment 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 7-1.

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.

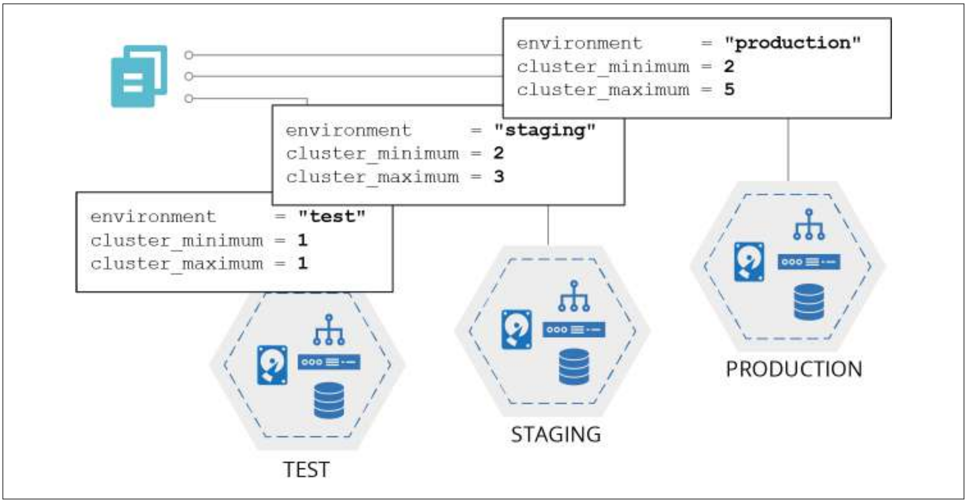


Figure 7-1. Using the same code with different parameter values for each environment

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" on page 17. 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 a 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 indicate 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 (per the reusable stack pattern discussed in "Pattern: Reusable Stack" on page 72), you may see failures from infrastructure resources that require unique identifiers. To see what I mean, look at the following pseudocode 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. Example 7-1 shows the project structure.

Example 7-1. Example project structure for a template stack that defines a container cluster

```
├── src/
│   ├── cluster.infra
│   └── networking.infra
└── test/
```

The cluster stack uses the parameters listed in Example 7-2 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 7-2. 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

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
Hardcode 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 7-3.

Example 7-3. 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. It's also useful to type parameters on the command line to experiment.

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 7-2), 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 tool to store and share them between team members securely, and rotate secrets when people leave the team.[1]

Related patterns

The scripted parameters pattern (see "Pattern: Scripted Parameters" on page 84) takes the command that you would type and puts it into a script. The pipeline stack parameters pattern (see "Pattern: Pipeline Stack Parameters" on page 93) 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, as shown in Example 7-4 (see "Implementation" on page 83 for more on how).

Example 7-4. Setting environment variables

```
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, as shown in Example 7-5.

Example 7-5. Stack code using environment variables

```
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.

[1] Some examples of tools teams can use to securely share passwords include GPG, KeePass, 1Password, Keeper, and LastPass.

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 Example 7-5, 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 (see "Antipattern: Manual Stack Parameters" on page 81). You could set them in a script that runs the stack tool (the scripted parameters pattern, as discussed in "Pattern: Scripted Parameters" on page 84), or have the pipeline tool set them (see "Pattern: Pipeline Stack Parameters" on page 93).

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 (see "Pattern: Stack Configuration Files" on page 87). The script to set the variables would be exactly like Example 7-4, and any command that runs the stack tool would import it into the environment:

```
source ./environments/staging.env
stack up --source ./src
```

Alternatively, 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. Example 7-5 showed how stack code can directly read environment variables.

But you could, instead, use a stack orchestration script (see "Using Scripts to Wrap Infrastructure Tools" on page 335) 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 hardcoding the parameter values into a script that runs the stack tool. You can write a separate script for each environment or a single script that includes the values for all of your environments:

```
if ${ENV} == "test"
  stack up cluster_maximum=1 env="test"
elsif ${ENV} == "staging"
  stack up cluster_maximum=3 env="staging"
elsif ${ENV} == "production"
  stack up cluster_maximum=5 env="production"
end
```

Motivation

Scripts are a simple way to capture the values for each instance, avoiding the problems with the manual stack parameters antipattern (see "Antipattern: Manual Stack Parameters" on page 81). 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 hardcode secrets in scripts, 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 (see "Handling Secrets as Parameters" on page 102 for suggestions).

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. "Using Scripts to Wrap Infrastructure Tools" on page 335 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 hardcoded parameter values for each environment. Example 7-6 is a simple example of this.

Example 7-6. 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

stack up \
    environment=$1 \
    cluster_minimum=${CLUSTER_MINIMUM} \
    cluster_maximum=${CLUSTER_MAXIMUM}
```

Another implementation is a separate script for each stack instance, as shown in Example 7-7.

Example 7-7. 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 hardcoded 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.

"Using Scripts to Wrap Infrastructure Tools" on page 335 discusses the use of scripts to run stack tools in much more detail.

As mentioned earlier, you shouldn't hardcode secrets into scripts, so you'll need to use a different pattern for those. You can use the script to support that pattern. In Example 7-8, a command-line tool fetches the secret from a secrets manager, following the parameter registry pattern (see "Pattern: Stack Parameter Registry" on page 96).

Example 7-8. 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 describe in "Secretless Authorization" on page 103).

Related patterns

Provisioning scripts run the command-line tool for you, so are a way to move beyond the manual stack parameters antipattern (see "Antipattern: Manual Stack Parameters" on page 81). The stack configuration files pattern 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, such as in Example 7-9.

Example 7-9. Example project with a parameter file for each environment

```
├── src/
│   ├── cluster.infra
│   ├── host_servers.infra
│   └── networking.infra
├── environments/
│   ├── test.properties
│   ├── staging.properties
│   └── production.properties
└── test/
```

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?")
- Trace the history for debugging ("When did the maximum cluster size change?")
- 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" on page 143, 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 described in "Infrastructure Delivery Pipelines" on page 119. 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 "Using Scripts to Wrap Infrastructure Tools" on page 335, 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 earlier in Example 7-9.

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 should 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" on page 84. You can avoid some of the limitations of environment configuration files by using the stack parameter registry pattern instead (see "Pattern: Stack Parameter Registry" on page 96). 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 component (see Chapter 16). Each wrapper project defines the parameter values for one instance of the stack. It then imports a component shared by all of the stack instances (see Figure 7-2).

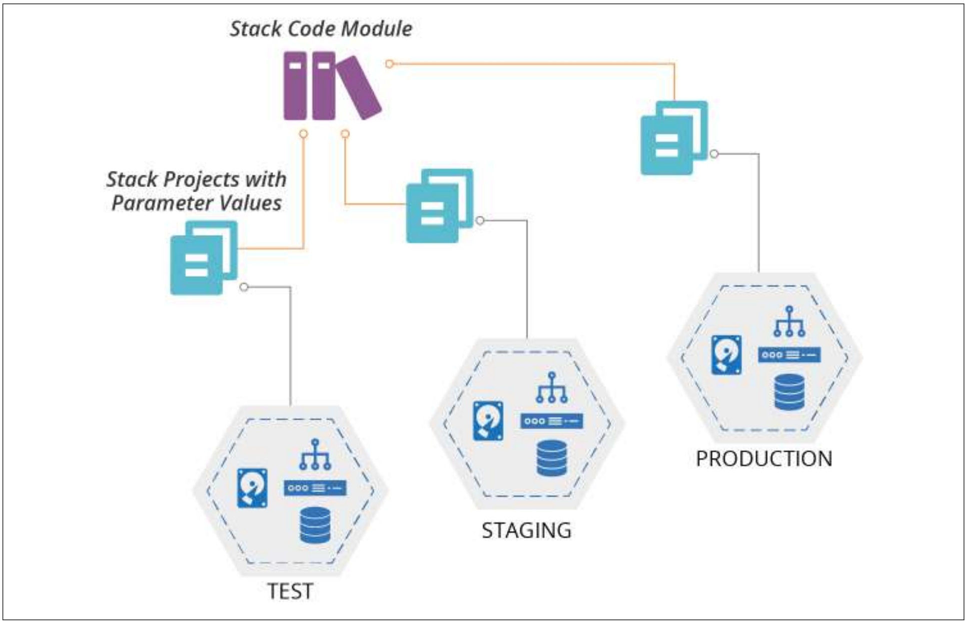


Figure 7-2. A wrapper stack uses an infrastructure stack project for each instance as a wrapper to import a stack module

Motivation

A wrapper stack leverages the stack tool's module functionality or library support to reuse 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 (see "Infrastructure Delivery Pipelines" on page 119). 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 (see "Pattern: Scripted Parameters" on page 84).

Consequences

Components add an extra layer of complexity between your stack and the code contained in the component. You now have two levels: the stack project, which contains the wrapper projects, and the component that 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 (see "Antipattern: Copy-Paste Environments" on page 70), 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 component.

Pattern: Pipeline Stack Parameters

With the pipeline stack parameters pattern, you define values for each instance in the configuration of a delivery pipeline (see Figure 7-3).

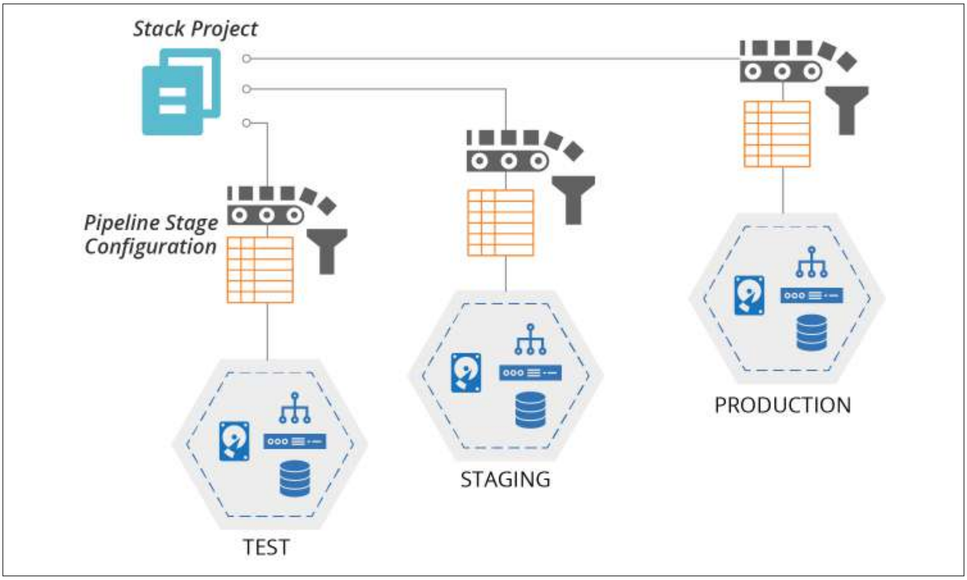


Figure 7-3. Each stage that applies the stack code passes the relevant configuration values for the environment

I explain how to use a change delivery pipeline to apply infrastructure stack code to environments in "Infrastructure Delivery Pipelines" on page 119. You can implement a pipeline using a tool like Jenkins, GoCD, or ConcourseCI (see "Delivery Pipeline Software and Services" on page 123 for more on these tools).

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 7-10 shows a pseudocode pipeline stage configuration.

Example 7-10. 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 shown in Example 7-11 (see also "Pattern: Stack Environment Variables" on page 82).

Example 7-11. 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 tool sets 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, as shown in Example 7-12.

Example 7-12. 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 (see Figure 7-4).

Configuration Registries and Stack Parameter Registries

I use the term *configuration registry* to describe a service that 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" on page 99.

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.

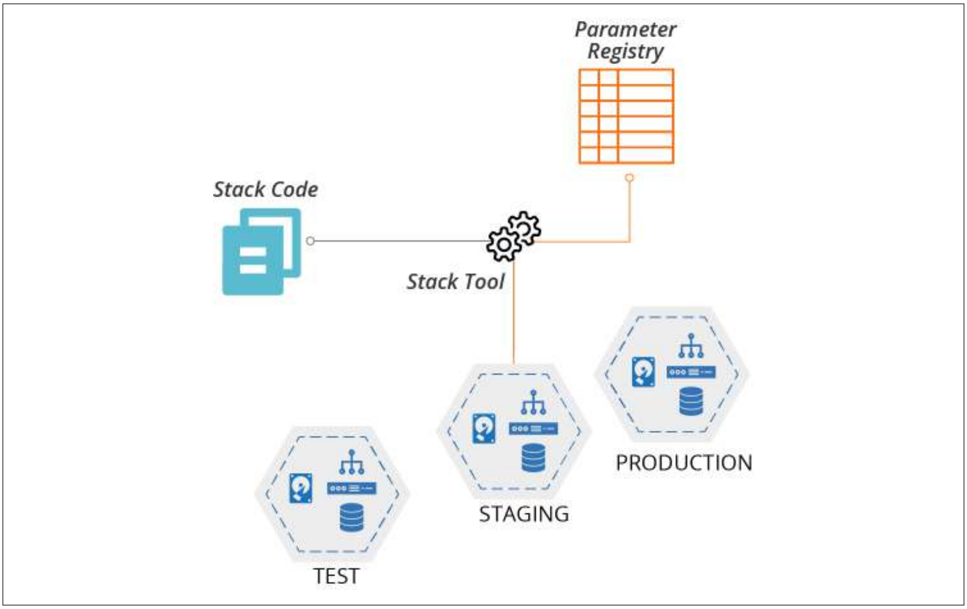


Figure 7-4. Stack instance parameter values stored in a central registry

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). 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 "Discovering Dependencies Across Stacks" on page 287).

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 trade-offs. 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 in "Configuration Registry" on page 99. 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 7-13.

Example 7-13. Example of configuration registration entries

```
└── env/
    ├── test/
    │   └── cluster/
    │        ├── min = 1
    │        └── max = 1
    ├── staging/
    │   └── cluster/
    │        ├── min = 2
    │        └── max = 3
    └── production/
        └── cluster/
             ├── min = 2
             └── max = 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: ${get_value("/env/${environment}/cluster/min")}
  maximum: ${get_value("/env/${environment}/cluster/max")}
```

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, which 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 (see "Storage Resources" on page 29) 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 that 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.

Chaining Configuration Patterns

Most tools support a chain of configuration options, with a predictable hierarchy of precedence—configuration file values overridden by environment variables overridden by command-line parameters. Most stack configuration regimes will support a similar hierarchy.

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" on page 96. It can also be useful for managing integration dependencies across different stack instances, applications, and other services, as I'll explain in "Discovering Dependencies Across Stacks" on page 287.

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. It 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
- etcd

- Consul[2]
- doozerd

These are generally compatible with different tools, languages, and systems, so avoid locking you into any particular toolchain.

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 (for Debian-based or Rad Hat-Based Linux distributions, respectively), 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.

[2] 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.

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.

Handling Secrets as Parameters

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 to connect to a 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.

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

Encrypting Secrets

The exception to the rule of never putting a secret in source code is encrypting it in the source code; git-crypt, blackbox, sops, and transcrypt are a few tools that help you to encrypt secrets in a repository.

The key to decrypt the secret should not be in the repository itself; otherwise, it would be available to attackers who gain access to the code. You'll need to use one of the other approaches described here to enable decryption.

Secretless Authorization

Many services and systems provide ways to authorize an action without using a secret. 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 tool 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 a secret for a stack or other infrastructure code, you can explore ways to inject the secret at runtime. You'll normally implement it as a stack parameter, which is the topic of Chapter 7. 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 code will often keep secrets in a local file that isn't stored in version control.[3] The stack tool could read that file directly, which is especially appropriate if you're using the stack configuration file

[3] I explain how people can work on stack code locally in more detail in "Personal Infrastructure Instances" on page 347.

pattern (see "Pattern: Stack Configuration Files" on page 87). Or the file could be a script that sets the secrets in environment variables, which works well with the stack environment variables pattern (see "Pattern: Stack Environment Variables" on page 82).

These approaches also work on unattended agents, such as those used for CI testing or CD delivery pipelines.[4] 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 (see "Pattern: Pipeline Stack Parameters" on page 93). Another option is to pull secrets from a secrets management service (of the type described in "Storage Resources" on page 29), which aligns to the stack parameter registry pattern (see "Pattern: Stack Parameter Registry" on page 96).

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 rerun 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 service that uses it, such as the application server. One-time passwords take this approach to the extreme of creating a new password every time authentication occurs.

Conclusion

Reusing stack projects requires you to be able to configure different instances of a given stack. Configuration should be minimized. If you find that you need different instances of a stack to be radically different from one another, you should define them as different stacks.

A stack project should define the shape of a stack that is consistent across instances. It's perfectly fine to have two different stack projects that define something superficially similar—application servers, for example—but with different shapes.

[4] I describe how these are used in "Infrastructure Delivery Pipelines" on page 119.

CHAPTER 8
Core Practice: Continuously Test and Deliver

Continuous testing and delivery is the second of the three core practices of Infrastructure as Code, which also include defining everything as code and building small pieces. Testing is a cornerstone of Agile software engineering. Extreme Programming (XP) emphasizes writing tests first with TDD, and frequently integrating code with CI.[1] CD extends this to testing the full production readiness of code as developers work on it, rather than waiting until they finish working on a release.[2]

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. This chapter explores strategies for testing and delivering infrastructure. It draws heavily on Agile engineering approaches to quality, including TDD, CI, and CD. These practices all build quality into a system by embedding testing into the code-writing process, rather than leaving it to be done later.

This chapter focuses on fundamental challenges and approaches for testing infrastructure. The next chapter (Chapter 9) builds on this one with specific guidance on testing infrastructure stack code, while Chapter 11 discusses testing server configuration code (see "Testing Server Code" on page 176).

[1] See "Continuous Integration" by Martin Fowler.

[2] 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.

Why Continuously Test 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 an automated test suite for something you build once?

Creating an automated test 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" on page 4, I explained the rationale for implementing systems for delivering changes to infrastructure. To recap, you'll make far more changes to your infrastructure after you build it than you might expect. Once any nontrivial system is live, you need to patch, upgrade, fix, and improve it.

A key benefit of CD is removing the classic, Iron Age distinction between the "build" and "run" phases of a system's life cycle.[3] Design and implement the delivery systems, including automated testing and code promotion, together with the system itself. Use this system to incrementally build your infrastructure, and to incrementally improve it throughout its operational lifetime. Going "live" is almost an arbitrary event, a change of who is using the system, but not how the system is managed.

What Continuous Testing Means

One of the cornerstones of Agile engineering is testing 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 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

[3] As described in "From the Iron Age to the Cloud Age" on page 2.

periodically during development, such as at the end of a sprint. Teams following Lean or Kanban test each story as they complete it.[4]

Truly continuous testing involves testing even more frequently than this. People write and run tests as they code, even before finishing a story. They frequently push their code into a centralized, automated build system—ideally at least once a day.[5]

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 testing.

Immediate Testing and Eventual Testing

Another way to think of this is to classify each of your testing activities as either immediate or eventual. Immediate testing happens when you push your code. Eventual testing happens after some delay, perhaps after a manual review, or maybe on a schedule.

Ideally, testing is truly immediate, happening as you write code. There are validation activities that run in your editor, such as syntax highlighting, or running unit tests. The Language Server Protocol (LSP) defines a standard for integrating syntax checking into IDEs, supported by implementations for various languages.

People who prefer the command line as a development environment can 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, but 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 tests 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.

[4] See the Mountain Goat Software site for an explanation of Agile stories.

[5] The *Accelerate* research published in the annual *State of 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.

What Should We Test with Infrastructure?

The essence of CI is to test every change someone makes as soon as possible. The essence of CD is to maximize the scope of that testing. 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 code to your system. Will the code break when applied? Does it create the right infrastructure? Does the infrastructure work the way it should? Does it meet operational criteria for performance, reliability, and security? Does it comply with regulatory and governance rules?

CD is about broadening the scope of risks that are immediately tested when pushing a change to the codebase, rather than waiting for eventual testing days, weeks, or even months afterwards. So on every push, a pipeline applies the code to realistic test environments and runs comprehensive tests. 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 testing any given change against those risks. This process takes the form of automated test suites and manual tests. 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. Constraints and requirements beyond the purely functional are often called Non-Functional Requirements (NFRs) or Cross-Functional Requirements (CFRs).[7] 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 and compliance with formatting rules, and run a 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 onto the infrastructure and checking that they run correctly. Doing

[6] See Jez Humble's website for more on CD patterns.

[7] My colleague Sarah Taraporewalla coined the term CFR to emphasize that people should not consider these to be separate from development work, but applicable to all of the work. See her website.

this indirectly tests that the infrastructure is correct, but you can often catch issues before deploying applications. An example of this for infrastructure is network routing. Can an HTTPS connection be made from the public internet to the web servers? You may be able 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 testing 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 testing. See "Governance in a Pipeline-based Workflow" on page 352 for a broader look at doing this.

Performance
Automated tools can test how quickly specific actions complete. Testing the speed of a network connection from point A to point B can surface issues 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 test 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 Testing" on page 115. But first I'd like to address the things that make infrastructure especially difficult to test.

Challenges with Testing Infrastructure Code

Most of the teams I encounter that 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.

The following 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 ("Declarative Infrastructure Languages" on page 41), many infrastructure tools use declarative languages rather than imperative languages. Declarative code typically declares the desired state for some infrastructure, such as this code that defines a networking subnet:

Example 8-1.

```
subnet:
  name: private_A
  address_range: 192.168.0.0/16
```

A test for this would simply restate 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 preceding test can uncover:

- The infrastructure code was never applied.
- The infrastructure code was applied, but the tool failed to apply it correctly, without returning an error.
- 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.

Declarative Tests

The *Given, When, Then* format is useful for writing tests.[8] A declarative test omits the "When" part, having a format more like "Given a thing, Then it has these characteristics." Tests written like this suggest that the code you're testing doesn't create variable outcomes. Declarative tests, like declarative code, have a place in many infrastructure codebases, but be aware that many tools and practices for testing dynamic code may not be appropriate.

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

The previous example of declarative code is simple—the values are hardcoded, so the result of applying the code is clear. Variables introduce the possibility of creating dif-

[8] See Perryn Fowler's post for an explanation of writing *Given, When, Then* tests.

ferent 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: get_networking_subrange(
    get_vpc(${MY_ENVIRONMENT}),
    data_centers.howmany,
    data_centers.howmany++
  )
```

This code has some logic that might be worth testing. It calls two functions, `get_networking_subrange` 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.

> 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 imperative code (as discussed in "Separate Declarative and Imperative Code" on page 46). The tests for the subnet code would need to include various edge cases of the imperative 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 that 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. Chapter 15 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. See "Using Test Fixtures to Handle Dependencies" on page 137 for more about test doubles.

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 Testing" on page 115.

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" on page 143.

Online and offline tests

Some types of tests run *online*, requiring you to provision infrastructure on the "real" cloud platform. Others can run *offline* on your laptop or a build agent. Tests that you can run offline include 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 testing 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. See "Offline Testing Stages for Stacks" on page 131 and "Online Testing Stages for Stacks" on page 134 for more detail on offline and online testing for stacks.

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.

Challenge: Dependencies Complicate Testing Infrastructure

The time needed to set up other infrastructure that your code depends on makes testing even slower. A useful technique for addressing this is to replace dependencies with 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 book *xUnit Test Patterns* (Addison-Wesley) to be useful.[9]

In the context of infrastructure, there is a growing number of tools that allow you to mock the APIs of cloud vendors.[10] You can apply your infrastructure code to a local mocked cloud to test 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.

It's often more useful to use test doubles for other infrastructure components than for the infrastructure platform itself. Chapter 9 gives examples of using test doubles and other test fixtures for testing infrastructure stacks (see "Using Test Fixtures to Handle Dependencies" on page 137). Later chapters in Part IV describe breaking infrastructure into smaller pieces and integrating them. Test fixtures are a key tool for keeping components loosely coupled.

Progressive Testing

Most nontrivial systems use multiple suites of tests to validate changes. Different suites may test different things (as listed in "What Should We Test with Infrastructure?" on page 108). One suite may test 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.

Progressive testing 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 help you think about how to structure validation activities across your test suites.

[9] Martin Fowler's bliki "Mocks Aren't Stubs" is a useful reference for test doubles.

[10] Examples of cloud mocking tools and libraries include Localstack and moto. Do Better As Code maintains a current list of this kind of tool.

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 (Figure 8-1). This way, small errors are quickly made visible so they can be quickly fixed and retested.

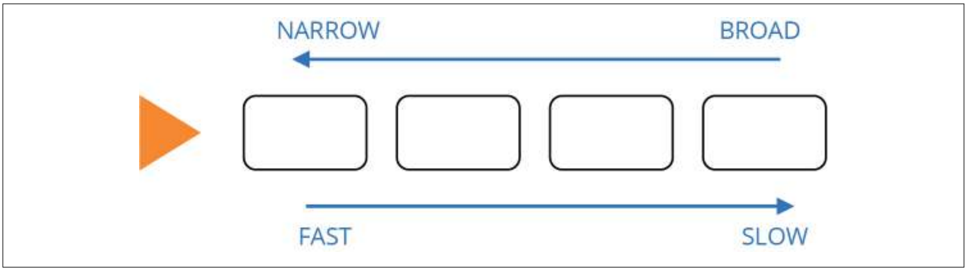


Figure 8-1. Scope versus speed of progressive testing

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 tests 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 testing.[11] 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, and fewer tests in the later stages (see Figure 8-2).

The pyramid was devised for application software development. The lower level of the pyramid is composed of unit tests, each of which tests a small piece of code and runs very quickly.[12] The middle layer is integration tests, each of which covers a collection of components assembled together. The higher stages are journey tests, driven through the user interface, which test the application as a whole.

[11] "The Practical Test Pyramid" by Ham Vocke is a thorough reference.

[12] See the ExtremeProgramming.org definition of unit tests. Martin Fowler's bliki definition of UnitTest discusses a few ways of thinking of unit tests.

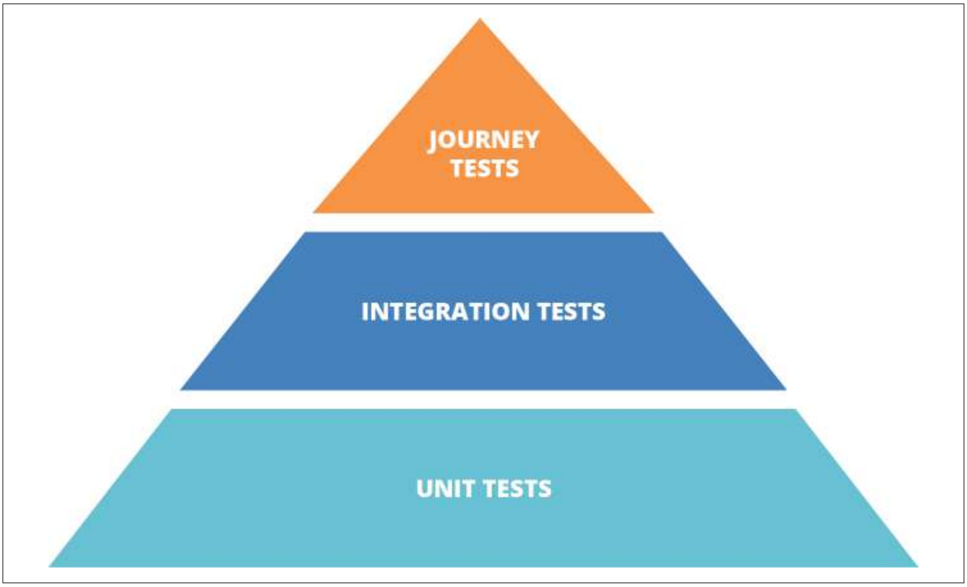


Figure 8-2. The classic test pyramid

The tests in higher levels of the pyramid cover the same scope already covered in lower levels. This means they can be less comprehensive—they only need to test functionality that emerges from the integration of components, rather than proving the behavior of lower-level components.

The testing pyramid is less valuable with declarative infrastructure codebases. Most low-level declarative stack code (see "Low-Level Infrastructure Languages" on page 54) written for tools like Terraform and CloudFormation is too large for unit testing, and depends on the infrastructure platform. Declarative modules (see "Reuse Declarative Code with Modules" on page 272) are difficult to test in a useful way, both because of the lower value of testing declarative code (see "Challenge: Tests for Declarative Code Often Have Low Value" on page 110) and because there is usually not much that can be usefully tested without the infrastructure.

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 for declarative infrastructure may end up looking more like a diamond, as shown in Figure 8-3.

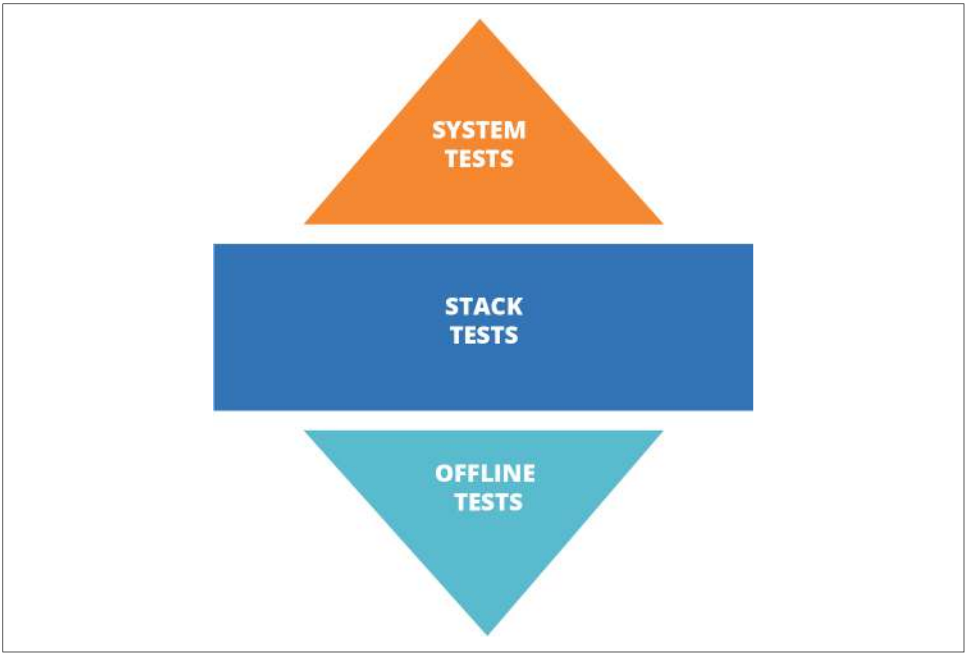


Figure 8-3. The infrastructure test diamond

The pyramid may be more relevant with an infrastructure codebase that makes heavier use of dynamic libraries (see "Dynamically Create Stack Elements with Libraries" on page 273) written in imperative languages (see "Programmable, Imperative Infrastructure Languages" on page 43). These codebases have more small components that produce variable results, so there is more to test.

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 industry. 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 (see Figure 8-4). 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.

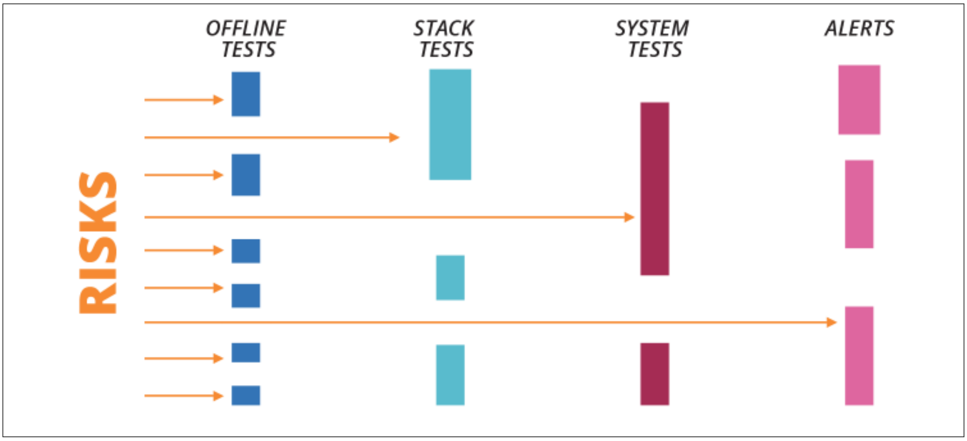


Figure 8-4. Swiss cheese testing model

The key takeaway is to test based on risk rather than based on fitting a formula.

Infrastructure Delivery Pipelines

A CD pipeline combines the implementation of progressive testing with the delivery of code across environments in the path to production.[13] Chapter 19 drills into details of how pipelines can package, integrate, and apply code to environments. This section explains how to design a pipeline for progressive testing.

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.

A pipeline automates processes involved in packaging, promoting, and applying code and tests. Humans may review changes, and even conduct exploratory testing on environments. But they should not run commands by hand to deploy and apply changes. They also shouldn't select configuration options or make other decisions on the fly. These actions should be defined as code and executed by the system.

Automating the process ensures it is carried out consistently every time, for every stage. Doing this improves the reliability of your tests, and creates consistency between instances of the infrastructure.

[13] 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.

Every change should be pushed from the start of 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, as shown in Figure 8-5. This practice ensures that every change is fully tested.

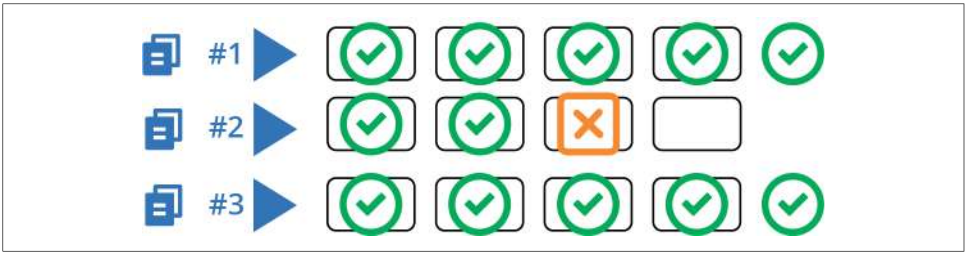


Figure 8-5. Start a new pipeline run to correct a failure

In the 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 in the third run of the pipeline.

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
: An event that causes the stage to start running. It may automatically run when a change is pushed to the code repository, or on 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 the stage is marked as passing or failing. The system could automatically 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

Artifacts or other material produced by the stage. Typical outputs include an infrastructure code package or a test report.

Scope of Components Tested in a Stage

In a progressive testing strategy, earlier stages validate individual components, while later stages integrate components and test them together. Figure 8-6 shows an example of progressively testing the components that lead to a web server running as part of a larger stack.

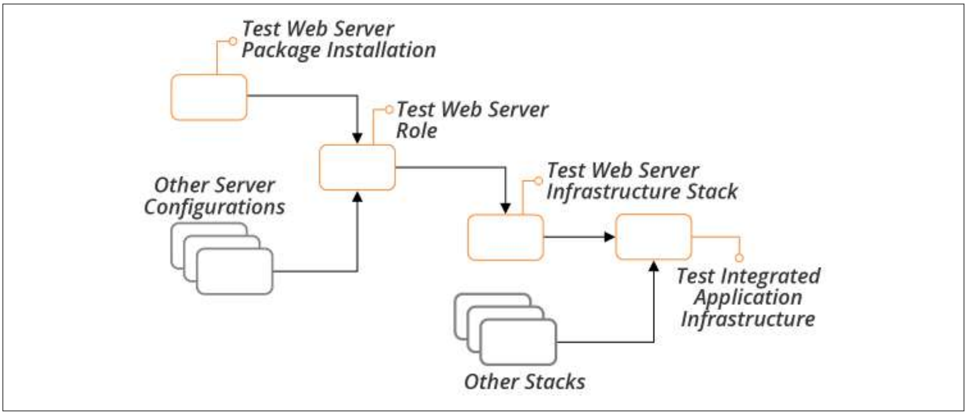

Test Web Server Package Installation
Test Web Server Role
Other Server Configurations
Test Web Server Infrastructure Stack
Test Integrated Application Infrastructure
Other Stacks

Figure 8-6. Progressive integration and testing of components

One stage might run tests for multiple components, such as a suite of unit tests. Or, different components may each have a separate test stage. Chapter 17 outlines different strategies for when to integrate different components, in the context of infrastructure stacks (see "Integrating Projects" on page 326).

Scope of Dependencies Used for a 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 test 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 (see Figure 8-7).

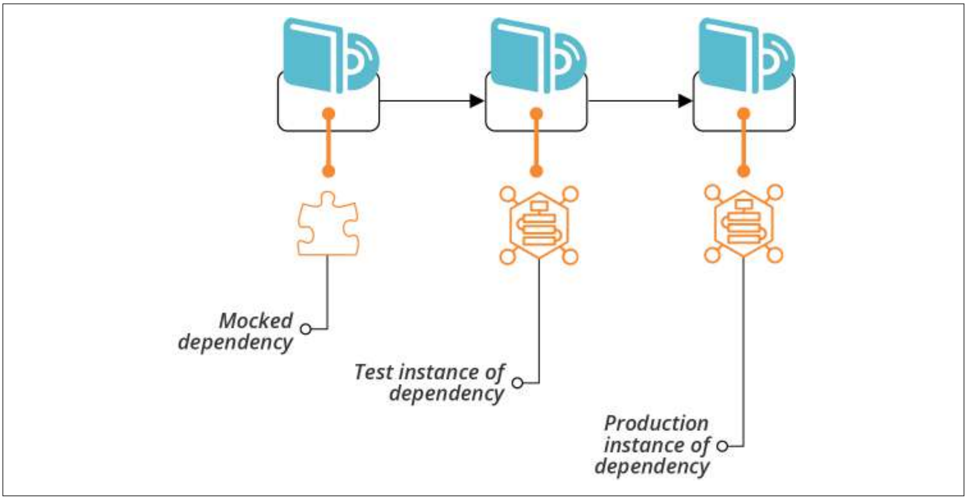

Mocked dependency
Test instance of dependency
Production instance of dependency

Figure 8-7. Progressive integration with dependencies

Only Include Stages That Add Value

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 testing into stages this way only when it adds enough value to be worth the overhead. Some reasons that may drive you to do this include speed, reliability, cost, and control.

Platform Elements Needed for a Stage

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 usefully run and test parts of it offline.

For example, code that defines networking structures needs to provision those structures on the cloud platform for meaningful tests. 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 test stages may be able to run without using the full cloud platform for some components (see Figure 8-8).

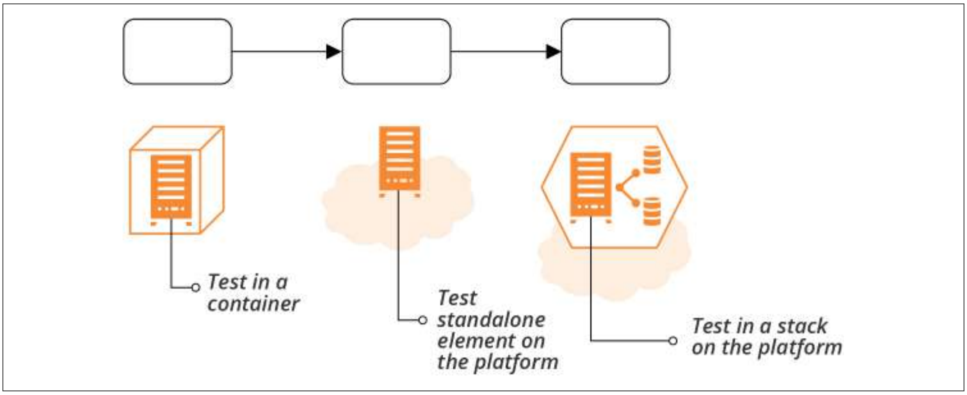

Test in a container
Test standalone element on the platform
Test in a stack on the platform

Figure 8-8. Progressive use of platform elements

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:

Build server
Many teams use a build server such as Jenkins, Team City, Bamboo, or GitHub Actions 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 across multiple 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 forward and backward. CD tools include GoCD,[14] ConcourseCI,[15] 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 Pipelines.

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 here were all designed with application software in mind. You can use most of them 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 is a rapidly changing area, so I suspect that what I have to say about these tools will be 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.

[14] In the interest of full disclosure, my employer, ThoughtWorks, created GoCD. It was previously a commercial product, but it is now fully open source.

[15] In spite of its name, ConcourseCI is designed around pipelines rather than CI jobs.

- 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 it calls GitOps. Even if you don't use WeaveWorks's tools, it's an emerging model that's worth watching. I'll touch on it a bit more in "GitOps" on page 351.

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 releases, even apart from the various technical teams who had to carry out tasks to install and configure various parts of the system.[16]

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 prerelease testing can comprehensively cover your risks leads to:

- Over-investing in prerelease 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)", 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 that 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.

[16] These groups were: change management, information security, risk management, service management, transition management, system integration testing, user acceptance, and the technical governance board.

Traffic

If your system has a nontrivial 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.

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.[17]

Zero-downtime deployment

Being able to deploy and roll back changes quickly and seamlessly helps mitigate the risk of errors (see "Changing Live Infrastructure" on page 368).

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 "Changing Live Infrastructure" on page 368).

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.

Chaos engineering

Lower risk in production environments by deliberately injecting known types of failures to prove that your mitigation systems work correctly (see "Chaos Engineering" on page 379).

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.

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.

[17] Although it's often conflated with monitoring, observability is about giving people ways to understand what's going on inside your system. See Honeycomb's "Introduction to Observability".

CHAPTER 9

Testing Infrastructure Stacks

This chapter applies the core practice of continuously testing and delivering code to infrastructure stacks. It uses the ShopSpinner example to illustrate how to test a stack project. This includes using online and offline test stages and making use of test fixtures to decouple the stack from dependencies.

Example Infrastructure

The ShopSpinner team uses reusable stack projects (see "Pattern: Reusable Stack" on page 72) to create consistent instances of application infrastructure for each of its customers. It can also use this to create test instances of the infrastructure in the pipeline.

The infrastructure for these examples is a standard three-tier system. The infrastructure in each tier consists of:

Web server container 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 Chapter 17.

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

ShopSpinner runs a separate database instance for each customer application instance, using its provider's DBaaS (see "Storage Resources" on page 29). ShopSpinner's 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 Example 9-1.

Example 9-1. Stack project for ShopSpinner customer application

```
stack-project/
   └── src/
       ├── appserver_vm.infra
       ├── appserver_networking.infra
       ├── database.infra
       └── database_networking.infra
```

Within this project, the file *appserver_vm.infra* includes code along the lines of what is shown in Example 9-2.

Example 9-2. 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.shopspinner_java_server_image
  provision:
    tool: servermaker
    parameters:
      maker_server: maker.shopspinner.xyz
      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 7.

As described in Chapter 8, the team uses multiple test stages ("Progressive Testing" on page 115), organized in a sequential pipeline ("Infrastructure Delivery Pipelines" on page 119).

Pipeline for the Example Stack

A simple pipeline for the ShopSpinner application infrastructure stack has two testing stages,[1] followed by a stage that applies the code to each customer's production environment (see Figure 9-1).

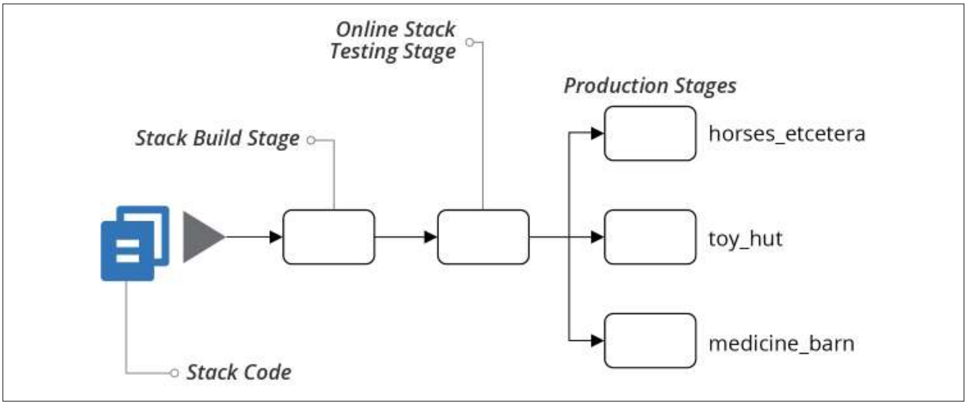


Figure 9-1. Simplified example pipeline for a stack

The first stage of the pipeline is the stack build stage. A *build stage* for an application usually compiles code, runs unit tests (described in "Test Pyramid" on page 116), and builds a deployable artifact. See "Building an Infrastructure Project" on page 322 for more details of a typical build stage for infrastructure code. Because earlier stages in a pipeline should run faster, the first stage is normally used to run offline tests.

The second stage of the example pipeline runs online tests for the stack project. Each of the pipeline stages may run more than one set of tests.

Offline Testing 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" on page 123), rather than needing to provision infrastructure on your infrastructure platform. Strict offline testing runs entirely

[1] 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 "Delivering Infrastructure and Applications" on page 314). You might also need a customer acceptance testing stage before each customer production stage. This also doesn't include governance and approval stages, which many organizations require.

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 tests you can carry out on 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 scriptable syntax checking tools include `terraform validate` and `aws cloudformation validate-template`.

The output of a failing syntax check 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 *linting*.[2] 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, cfn_nag, tfsec, and checkov.

[2] The term *lint* comes from a classic Unix utility that analyzes C source code.

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 (see "Preview: Seeing What Changes Will Be Made" on page 134), 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.shopspinner_java_server_image' doesn't
  exist (validate_server_images)

  on appserver_vm.infra line 5, in resource "virtual_machine":
```

Testing with a 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. Some tools can mock parts of a platform, such as Azurite, which emulates Azure blob and queue storage.

Applying declarative stack code to a local mock can reveal coding errors that syntax or code analysis checks might not find. In practice, testing declarative code with infrastructure platform API mocks isn't very valuable, for the reasons discussed in "Challenge: Tests for Declarative Code Often Have Low Value" on page 110. However, these mocks may be useful for unit testing imperative code (see "Programmable, Imperative Infrastructure Languages" on page 43), especially libraries (see "Dynamically Create Stack Elements with Libraries" on page 273).

Online Testing 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 testing than online tests. 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 "Handling Secrets as Parameters" on page 102 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 ShopSpinner customer application infrastructure works with a shared web server cluster stack. However, the ShopSpinner team members implement their infrastructure, and testing stages, using techniques that allow them to 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 Chapter 15. 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 in "Using Test Fixtures to Handle Dependencies" on page 137.

First, consider how different types of online stack tests work. The tests 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_image"
```

But the preview may not tell you that `"java_server_image"` 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 (see "Antipattern: Copy-Paste Environments" on page 70) often use a preview stage as a minimal test for each environment. But teams using reusable stacks (see "Pattern: Reusable Stack" on page 72) 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. Some examples of frameworks for testing infrastructure resources include:

- Awspec
- Clarity
- Inspec
- Taskcat
- Terratest

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" on page 110), 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" on page 112). 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 16) 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 in "Life Cycle Patterns for Test Instances of Stacks" on page 142, but 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" on page 137).

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: "shopspinner_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. Test doubles, mentioned in

"Challenge: Dependencies Complicate Testing Infrastructure" on page 114, are a type of test fixture.

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* (see Figure 9-2).

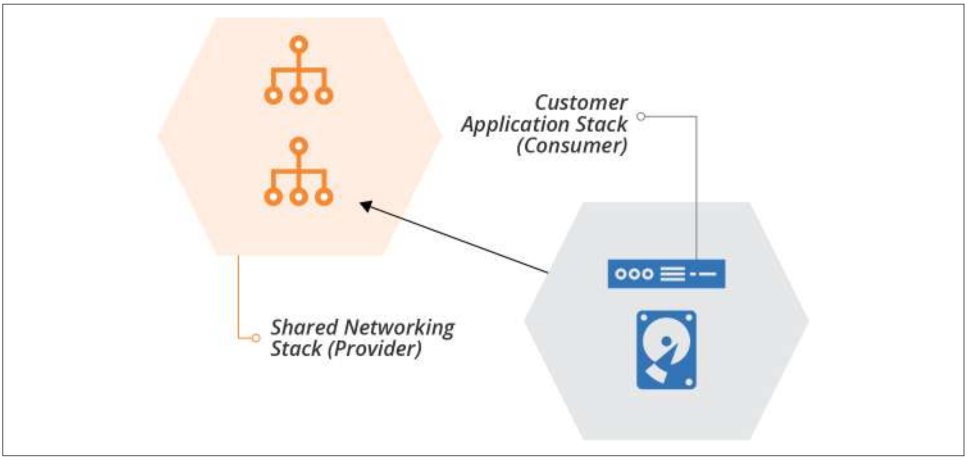


Figure 9-2. Example of a provider stack and consumer stack

Our ShopSpinner example has a provider stack that 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 and assigns it to a network address block.[3]

[3] Chapter 17 explains how to connect stack dependencies.

A given stack may be both a provider and a consumer, consuming resources from another stack and providing resources to other stacks. 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. 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 ShopSpinner 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, or customer data protection regulations. 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 9-3, the stack instance defines networking structures for ShopSpinner, 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.

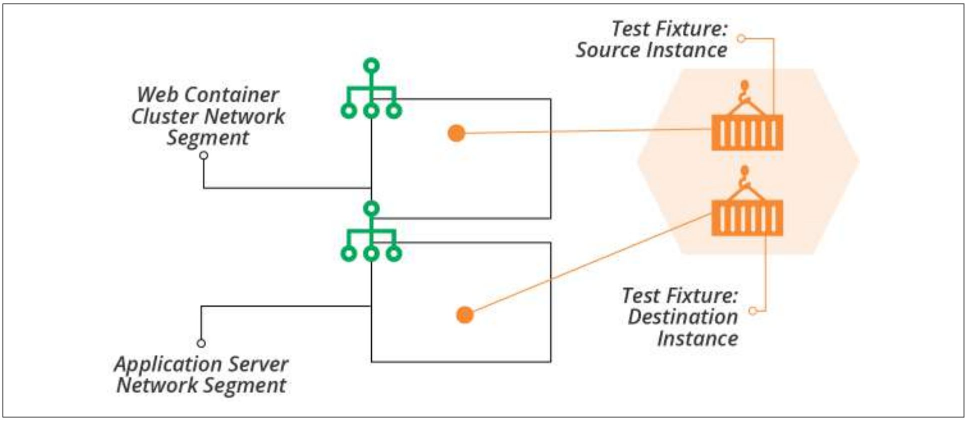


Figure 9-3. Test instance of the ShopSpinner network stack, with test fixtures

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 (see "Verification: Making Assertions About Infrastructure Resources" on page 135) for outcome testing. These example tests use a fictional stack testing DSL:

```
given stack_instance(stack: "shopspinner_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 test 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 9-4.

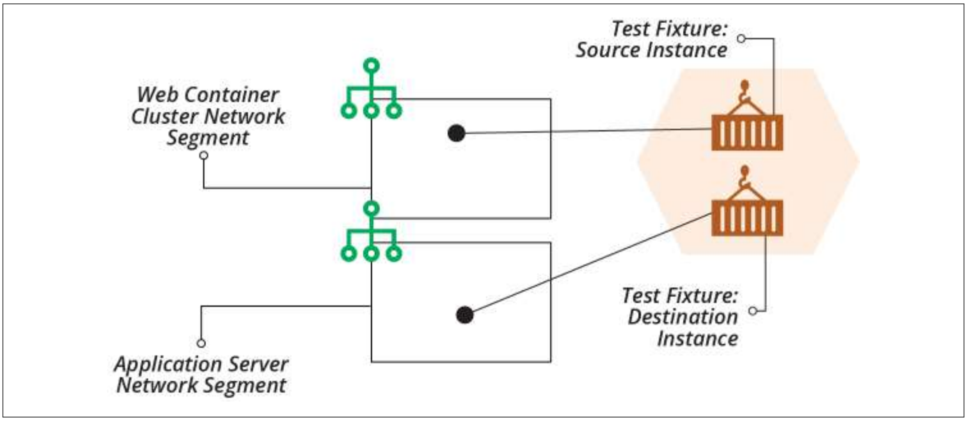


Figure 9-4. Testing connectivity in the ShopSpinner network stack

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. Martin Fowler has written about refactoring and other techniques for improving system architecture. For example, the Strangler Application prioritizes keeping the system fully working while restructuring it over time.

Part IV of this book involves more detailed rules and examples for modularizing and integrating infrastructure.

Life Cycle 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 trade-offs 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

Also known as: static environment.

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 (see Figure 9-5).

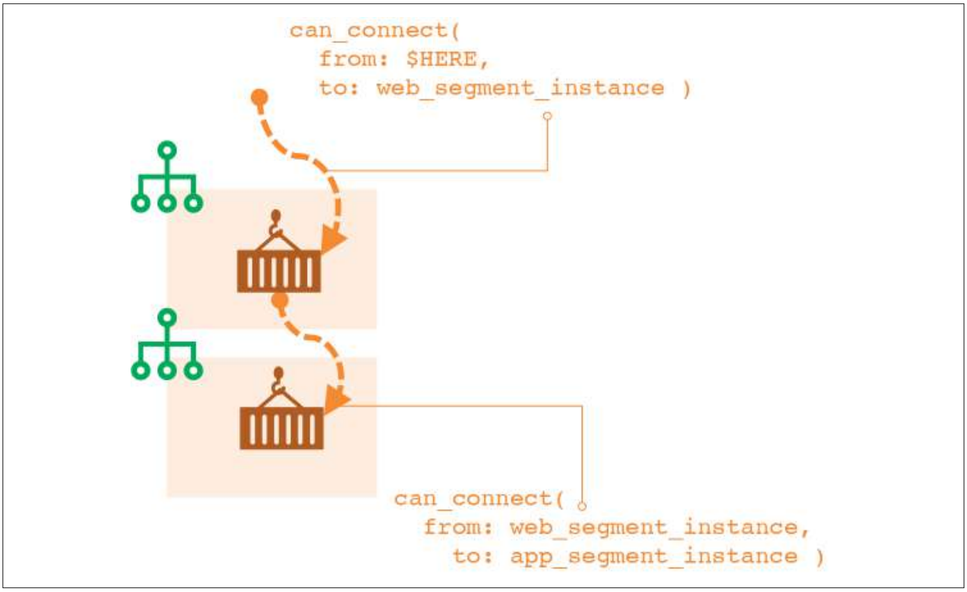

can_connect(
from: $HERE,
to: web_segment_instance)
can_connect(
from: web_segment_instance,
to: app_segment_instance)

Figure 9-5. Persistent test stack instance

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

It's not uncommon for stack instances to become "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 unwedging 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 Chapter 15 for more on 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 discussed in "Pattern: Periodic Stack Rebuild" on page 146 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

Also known as: quick and dirty plus slow and clean.

With the *ephemeral test stack* pattern, the test stage creates and destroys a new instance of the stack every time it runs (see Figure 9-6).

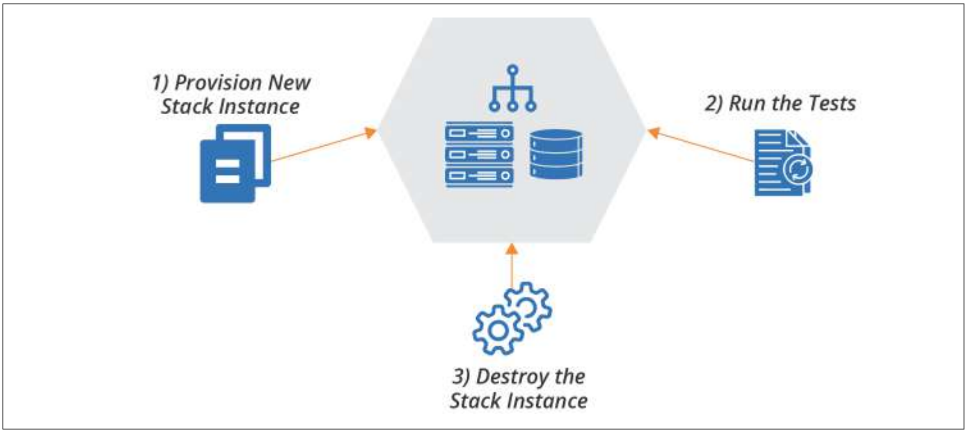

1) Provision New Stack Instance
2) Run the Tests
3) Destroy the Stack Instance

Figure 9-6. Ephemeral test stack instance

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" on page 147) 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

Also known as: nightly rebuild.

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 combines the persistent test stack pattern (see "Pattern: Persistent Test Stack" on page 142) and the ephemeral test stack pattern (see "Pattern: Ephemeral Test Stack" on page 143).

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 life cycle 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, as shown in Figure 9-7. 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.

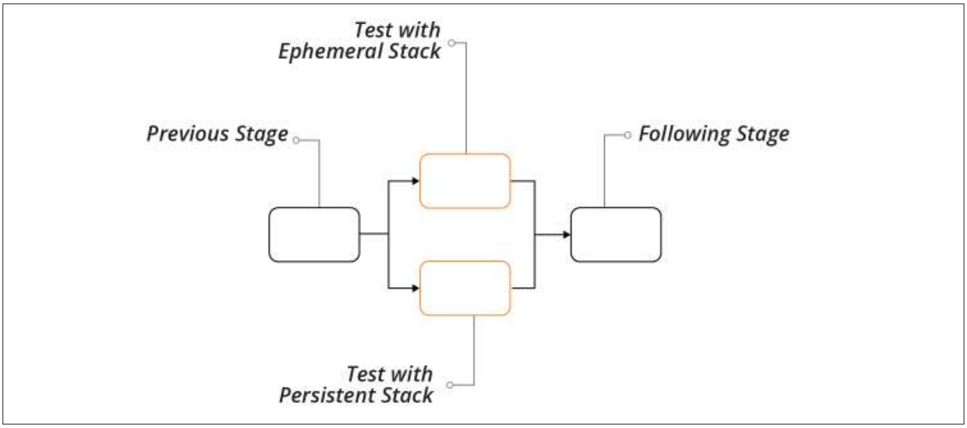


Figure 9-7. Dual persistent and ephemeral stack stages

Related patterns

This antipattern combines the persistent test stack pattern (see "Pattern: Persistent Test Stack" on page 142) and the ephemeral test stack pattern (see "Pattern: Ephemeral Test Stack" on page 143).

Pattern: Periodic Stack Rebuild

Periodic stack rebuild uses a persistent test stack instance (see "Pattern: Persistent Test Stack" on page 142) 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.

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 time zones, 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 (see "Pattern: Persistent Test Stack" on page 142)—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 (see Figure 9-8).

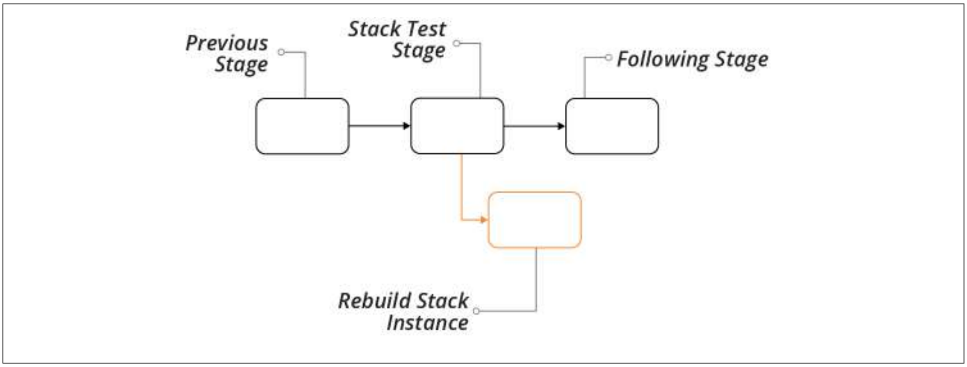


Figure 9-8. Pipeline flow for continuous stack reset

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 (see "Pattern: Persistent Test Stack" on page 142), providing feedback, while having the reliability of the ephemeral test stack pattern (see "Pattern: Ephemeral Test Stack" on page 143).

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 life cycles 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 "Using Scripts to Wrap Infrastructure Tools" on page 335, 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 life cycle 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 life cycles, 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. "Personal

Infrastructure Instances" on page 347 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 machines for packaging development environments that can run on different types of desktop systems.[4] Alternatively, 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 plug-ins 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 "Using Scripts to Wrap Infrastructure Tools" on page 335). 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 and Molecule. 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.

[4] Vagrant is handy for sharing virtual machine configuration between members of a team.

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 gave an example of creating a pipeline with multiple stages to implement the core practice of continuously testing and delivering stack-level code. One of the challenges with testing stack code is tooling. 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.

PART III

Working with Servers and Other Application Runtime Platforms

CHAPTER 10

Application Runtimes

I introduced application runtimes in "The Parts of an Infrastructure System" on page 23, as part of a model that organizes the parts of a system into three layers. In this model, you combine resources from the infrastructure layer to provide runtime platforms that people can deploy applications onto.

Application runtimes are composed of infrastructure stacks that you define and create using infrastructure management tools, as described in Chapter 5, and depicted in Figure 10-1.

The starting point for designing and implementing application runtime infrastructure is understanding the applications that will use it. What language and execution stacks do they run? Will they be packaged and deployed onto servers, in containers, or as FaaS serverless code? Are they single applications deployed to a single location, or multiple services distributed across a cluster? What are their connectivity and data requirements?

The answers to these questions lead to an understanding of the infrastructure resources that the application runtime layer needs to provision and manage to run the applications. The parts of the application runtime layer map to the parts of the infrastructure platform I describe in "Infrastructure Resources" on page 27. These will include an execution environment based around compute resources, data management built on storage resources, and connectivity composed of networking resources.

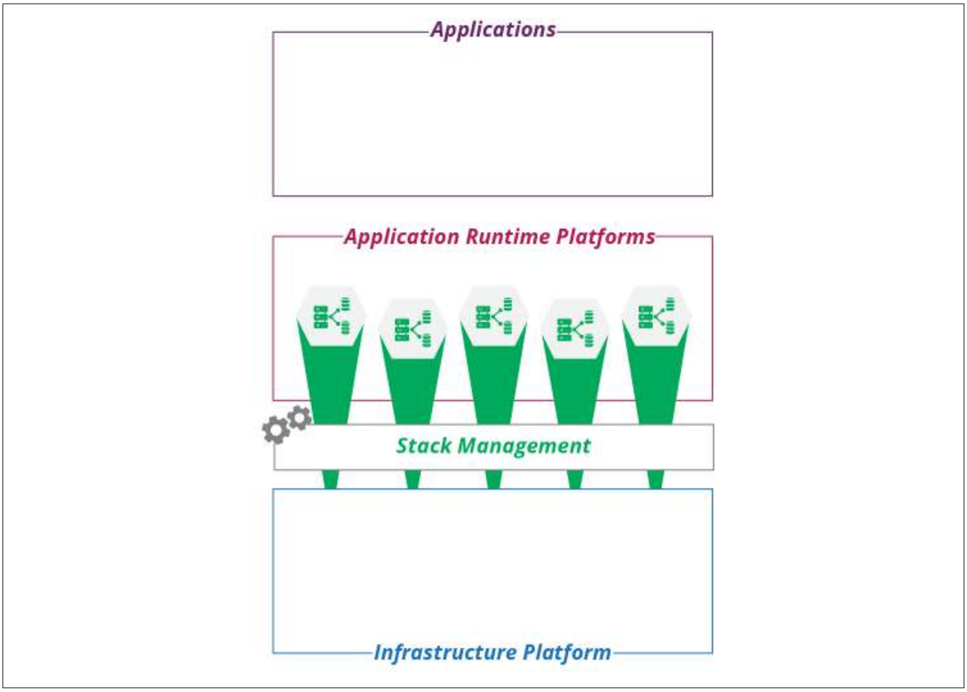


Figure 10-1. The application layer composed of infrastructure stacks

This chapter summarizes each of these relationships, focusing on ways to organize infrastructure resources into runtime platforms for applications. It sets the stage for later chapters, which go into more detail on how to define and manage those resources as code—servers as code (Chapter 11) and clusters as code (Chapter 14).

Cloud Native and Application-Driven Infrastructure

Cloud native software is designed and implemented to exploit the dynamic nature of modern infrastructure. Unlike older-generation software, instances of a cloud native application can be transparently added, removed, and shifted across underlying infrastructure. The underlying platform dynamically allocates compute and storage resources and routes traffic to and from the application. The application integrates seamlessly with services like monitoring, logging, authentication, and encryption.

The folks at Heroku articulated the twelve-factor methodology for building applications to run on cloud infrastructure. Cloud native, as a phrase, is often associated with the Kubernetes ecosystem.[1]

[1] The Cloud Native Computing Foundation aims to set standards for the genre.

Many organizations have existing software portfolios that aren't cloud native. While they may convert or rewrite some of their software to become cloud native, in many cases the cost of doing so isn't justified by the benefits. An application-driven infrastructure strategy involves building application runtime environments for applications using modern, dynamic infrastructure.

Teams provide application runtimes for new applications that run as serverless code or in containers. They also provide infrastructure to support existing applications. All of the infrastructure is defined, provisioned, and managed as code. Application-driven infrastructure may be provided dynamically using an abstraction layer (see "Building an Abstraction Layer" on page 284).

Application Runtime Targets

Implementing an application-driven strategy starts with analyzing the runtime requirements of your application portfolio. You then design application runtime solutions to meet those requirements, implementing reusable stacks, stack components, and other elements teams can use to assemble environments for specific applications.

Deployable Parts of an Application

A deployable release for an application may involve different elements. Leaving aside documentation and other metadata, examples of what may be part of an application deployment include:

Executables
: The core of a release is the executable file or files, whether they are binaries or interpreted scripts. You can consider libraries and other files used by the executables as members of this category.

Server configuration
: Many application deployment packages make changes to server configuration. These can include user accounts that processes will run as, folder structures, and changes to system configuration files.

Data structures
: When an application uses a database, its deployment may create or update schemas. A given version of the schema usually corresponds to a version of the executable, so it is best to bundle and deploy these together.

Reference data
: An application deployment may populate a database or other storage with an initial set of data. This could be reference data that changes with new versions or example data that helps users get started with the application the first time they install it.

Connectivity

An application deployment may specify network configuration, such as network ports. It may also include elements used to support connectivity, like certificates or keys used to encrypt or authenticate connections.

Configuration parameters

An application deployment package may set configuration parameters, whether by copying configuration files onto a server, or pushing settings into a registry.

You can draw the line between application and infrastructure in different places. You might bundle a required library into the application deployment package, or provision it as part of the infrastructure.

For example, a container image usually includes most of the operating system, as well as the application that will run on it. An immutable server or immutable stack takes this even further, combining application and infrastructure into a single entity. On the other end of the spectrum, I've seen infrastructure code provision libraries and configuration files for a specific application, leaving very little in the application package itself.

This question also plays into how you organize your codebase. Do you keep your application code together with the infrastructure code, or keep it separate? The question of code boundaries is explored later in this book (see Chapter 18), but one principle is that it's usually a good idea to align the structure of your codebase to your deployable pieces.

Deployment Packages

Applications are often organized into deployment packages, with the package format depending on the type of the runtime environment. Examples of deployment package formats and associated runtimes are listed in Table 10-1.

Table 10-1. Examples of target runtimes and application package formats

Target runtime	Example packages
Server operating system	Red Hat RPM files, Debian *.deb* files, Windows MSI installer packages
Language runtime engine	Ruby gems, Python pip packages, Java *.jar*, *.war*, and *.ear* files.
Container runtime	Docker images
Application clusters	Kubernetes Deployment Descriptors, Helm charts
FaaS serverless	Lambda deployment package

A deployment package format is a standard that enables deployment tools or runtimes to extract the parts of the application and put them in the right places.

Deploying Applications to Servers

Servers, whether physical or virtual, are the traditional runtime platform. An application is packaged using an operating system packaging format such as an RPM, a *.deb* file, or a Windows MSI. Or it is packaged in a language runtime format, such as a Ruby gem or Java *.war* file. More recently, container images, such as Docker images, have gained popularity as a format for packaging and deploying applications to servers.

Defining and provisioning servers as code is the topic of Chapter 11. This topic overlaps with application deployment, given the need to decide when and how to run deployment commands (see "Configuring a New Server Instance" on page 185).

Packaging Applications in Containers

Containers pull dependencies from the operating system into an application package, the container image, as shown in Figure 10-2.[2]

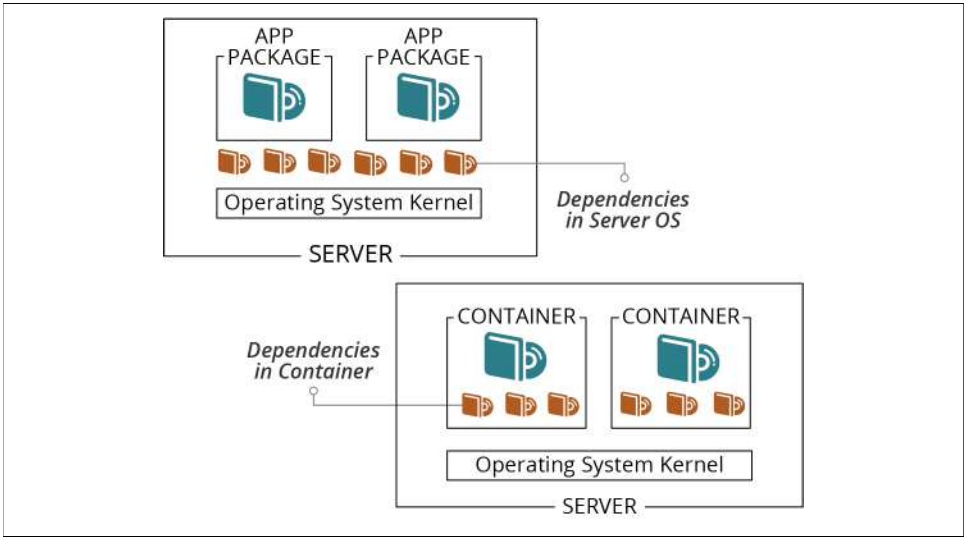


Figure 10-2. Dependencies may be installed on the host OS or bundled in containers

Including dependencies in the container makes it larger than typical operating system or language packages, but has several advantages:

[2] Docker is the dominant container format. Alternatives include CoreOS rkt and Windows Containers.

- A container creates a more consistent environment for running an application. Without a container, the application relies on libraries, configuration, user accounts, and other elements that may be different on different servers. A container bundles the runtime environment along with the application and its dependencies.
- A containerized application is mostly isolated from the server it runs on, which gives you flexibility for where you can run it.
- By packaging the application's operating system context in the container image, you simplify and standardize the requirements for the host server. The server needs to have the container execution tooling installed, but little else.
- Reducing the variability of runtime environments for an application improves quality assurance. When you test a container instance in one environment, you can be reasonably confident it will behave the same in other environments.

Deploying Applications to Server Clusters

People have been deploying applications to groups of servers since before container-based application clusters were a thing. The usual model is to have a server cluster (as described in "Compute Resources" on page 28), and run an identical set of applications on each server. You package your applications the same way you would for a single server, repeating the deployment process for each server in the pool, perhaps using a remote command scripting tool like Capistrano or Fabric. See Figure 10-3.

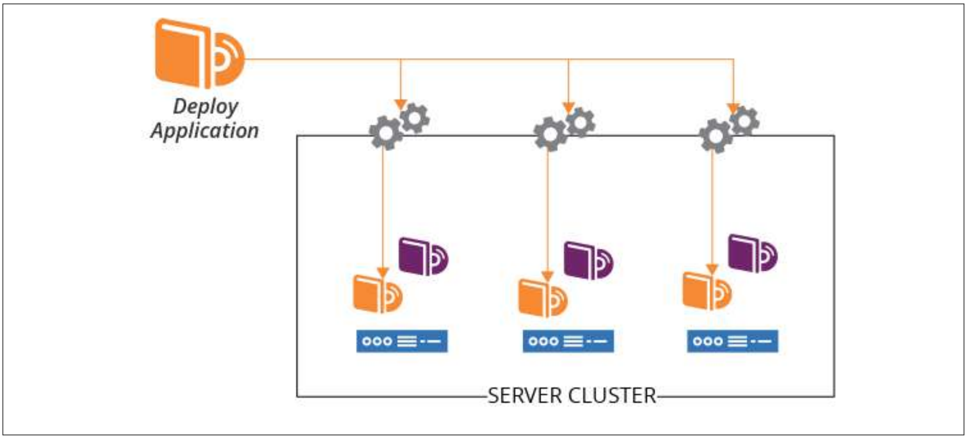


Figure 10-3. Applications are deployed to each server in a cluster

If you deploy an application to multiple servers, you need to decide how to orchestrate the deployment. Do you deploy the application to all of the servers at once? Do you need to take the entire service offline while you do this? Or do you upgrade one server at a time? You can leverage incremental deployment to servers for progressive deployment strategies like the blue-green and canary deployment patterns (see "Changing Live Infrastructure" on page 368 for more on these strategies).

In addition to deploying application code onto servers, you may need to deploy other elements like changes to data structures or connectivity.

Deploying Applications to Application Clusters

As discussed in "Compute Resources" on page 28, an application hosting cluster is a pool of servers that runs one or more applications. Unlike a server cluster, where each server runs the same set of applications, different servers in an application cluster may run different groups of application instances (see Figure 10-4).

When you deploy an application to the cluster, a scheduler decides which host servers to run instances of the application on. The schedule may change this distribution, adding and removing application instances across host servers according to different algorithms and settings.

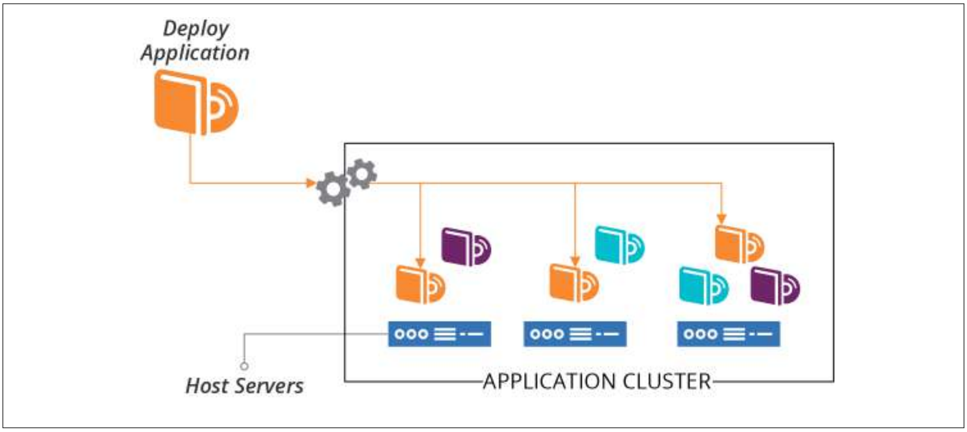


Figure 10-4. Applications are deployed to the cluster and distributed across host nodes

In the old days,[3] the most popular application clusters were Java-based (Tomcat, Websphere, Weblogic, JBoss, and others). A wave of cluster management systems emerged a few years ago, including Apache Mesos and DC/OS, many inspired by Google's Borg.[4] Systems focused on orchestrating container instances have overwhelmed application servers and cluster orchestrators in more recent years.

[3] "The old days" were less than ten years ago.

[4] Borg is Google's internal, proprietary cluster management system, documented in the paper "Large-Scale Cluster Management at Google with Borg".

Defining and provisioning clusters as code is the topic of Chapter 14. Once you have your cluster, it may be simple to deploy a single application onto it. Package the application with Docker and push it to the cluster. But more complex applications, with more moving parts, have more complex requirements.

Packages for Deploying Applications to Clusters

Modern applications often involve multiple processes and components deployed across complex infrastructure. A runtime environment needs to know how to run these various pieces:

- What are the minimum and maximum number of instances to run?
- How should the runtime know when to add or remove instances?
- How does the runtime know whether an instance is healthy or needs to be restarted?
- What storage needs to be provisioned and attached to each instance?
- What are the connectivity and security requirements?

Different runtime platforms provide different functionality, and many have their own packaging and configuration format. Often, these platforms use a *deployment manifest* that refers to the actual deployment artifacts (for example, a container image), rather than an archive file that includes all of the deployable pieces. Examples of cluster application deployment manifests include:

- Helm charts for Kubernetes clusters
- Weave Cloud Kubernetes deployment manifests
- AWS ECS Services, which you can define as code using your favorite stack management tool
- Azure App Service Plans
- CNAB Cloud Native Application Bundle

Different deployment manifests and packages work at different levels. Some are focused on a single deployable unit, so you need a manifest for each application. Some define a collection of deployable services. Depending on the tool, each of the services within the collection may have a separate manifest, with a higher-level manifest defining common elements and integration parameters.

A manifest for the ShopSpinner web server deployment might look like the pseudocode shown in Example 10-1.

Example 10-1. Example of an application cluster deployment manifest

```
service:
  name: webservers
  organization: ShopSpinner
  version: 1.0.3
application:
  name: nginx
  container_image:
    repository: containers.shopspinner.xyz
    path: /images/nginx
    tag: 1.0.3
  instance:
    count_min: 3
    count_max: 10
    health_port: 443
    health_url: /alive
  connectivity:
    inbound:
      id: https_inbound
      port: 443
      allow_from: $PUBLIC_INTERNET
      ssl_cert: $SHOPSPINNER_PUBLIC_SSL_CERT
    outbound:
      port: 443
      allow_to: [ $APPSERVER_APPLICATIONS.https_inbound ]
```

This example specifies where and how to find the container image (the `container_image` block), how many instances to run, and how to check their health. It also defines inbound and outbound connection rules.

Deploying FaaS Serverless Applications

In Chapter 3, I listed some FaaS serverless application runtime platforms as a type of compute resource (see "Compute Resources" on page 28). Most of these have their own format for defining application runtime requirements, and packaging these with code and related pieces for deploying to an instance of the runtime.

When writing a FaaS application, the details of whatever servers or containers your code runs in are hidden from you. But your code probably needs infrastructure to work. For example, you may need network routing for inbound or outbound connections, storage, or message queues. Your FaaS framework may integrate with the underlying infrastructure platform, automatically provisioning the necessary infrastructure. Or you may need to define the infrastructure elements in a separate stack definition tool. Many stack tools like Terraform and CloudFormation let you declare your FaaS code provisioning as a part of an infrastructure stack.

Chapter 14 is relevant for defining and provisioning FaaS runtimes to run your code.

Application Data

Data is often an afterthought for deploying and running applications. We provision databases and storage volumes but, as with many parts of infrastructure, it's making changes to them that turns out to be hard. Changing data and structures is time consuming, messy, and risky.

Application deployment often involves creating or changing data structures, including converting existing data when structures change. Updating data structures should be a concern of the application and application deployment process, rather than the infrastructure platform and runtime. However, the infrastructure and application runtime services need to support maintaining data when infrastructure and other underlying resources change or fail. See "Data Continuity in a Changing System" on page 382 for ways to approach this challenge.

Data Schemas and Structures

Some data storage is strictly structured, such as SQL databases, while others are unstructured or schema-less. A strictly structured, schema-driven database enforces structures on data, refusing to store incorrectly formatted data. Applications that use a schema-less database are responsible for managing the data format.

A new application release may include a change to data structures. For a schema-driven database, this involves changing the definition of the data structures in the database. For example, the release may add a new field to data records, a classic example being splitting a single "name" field into separate fields for first, middle, and last name.

When data structures change, any existing data needs to be converted to the new structure with either type of database. If you are splitting a "name" field, you will need a process that divides the names in the database into their separate fields.

Changing data structures and converting data is called *schema migration*. There are several tools and libraries that applications and deployment tools can use to manage this process, including Flyway, DBDeploy, Liquibase, and db-migrate.[5] Developers can use these tools to define incremental database changes as code. The changes can be checked into version control, and packaged as part of a release. Doing this helps to ensure that database schemas stay in sync with the application version deployed to an instance.

[5] DBDeploy popularized this style of database schema migration, along with Ruby on Rails. But the project isn't currently maintained.

Teams can use database evolution strategies to safely and flexibly manage frequent changes to data and schemas. These strategies align with Agile software engineering approaches, including CI and CD, as well as Infrastructure as Code.[6]

Cloud Native Application Storage Infrastructure

Cloud native infrastructure is dynamically allocated to applications and services on demand. Some platforms provide cloud native storage as well as compute and networking. When the system adds an application instance, it can automatically provision and attach a storage device. You specify the storage requirements, including any formatting or data to be loaded when it is provisioned, in the application deployment manifest (see Example 10-2).

Example 10-2. Example of an application deployment manifest with data storage

```
application:
  name: db_cluster
  compute_instance:
    memory: 2 GB
    container_image: db_cluster_node_application
  storage_volume:
    size: 50 GB
    volume_image: db_cluster_node_volume
```

This simple example defines how to create nodes for a dynamically scaled database cluster. For each node instance, the platform will create an instance of the container with the database software, and attach a disk volume cloned from an image initialized with an empty database segment. When the instance boots, it will connect to the cluster and synchronize data to its local volume.

Application Connectivity

In addition to compute resources to execute code and storage resources to hold data, applications need networking for inbound and outbound connectivity. A server-oriented application package, such as for a web server, might configure network ports and add encryption keys for incoming connections. But traditionally, they have relied on someone to configure infrastructure outside the server separately.

[6] For more on strategies and techniques you can use to evolve your databases, see "Evolutionary Database Design" by Pramod Sadalage, *Refactoring Databases* by Scott Ambler and Pramod Sadalage (Addison-Wesley Professional), and *Database Reliability Engineering*, by Laine Campbell and Charity Majors (O'Reilly).

You can define and manage addressing, routing, naming, firewall rules, and similar concerns as part of an infrastructure stack project, and then deploy the application into the resulting infrastructure. A more cloud native approach to networking is to define the networking requirements as part of the application deployment manifest and have the application runtime allocate the resources dynamically.

You can see this in part of Example 10-1, shown here:

```
application:
  name: nginx
  connectivity:
    inbound:
      id: https_inbound
      port: 443
      allow_from: $PUBLIC_INTERNET
      ssl_cert: $SHOPSPINNER_PUBLIC_SSL_CERT
    outbound:
      port: 443
      allow_to: [ $APPSERVER_APPLICATIONS.https_inbound ]
```

This example defines inbound and outbound connections, referencing other parts of the system. These are the public internet, presumably a gateway, and inbound HTTPS ports for application servers, defined and deployed to the same cluster using their own deployment manifests.

Application runtimes provide many common services to applications. Many of these services are types of service discovery.

Service Discovery

Applications and services running in an infrastructure often need to know how to find other applications and services. For example, a frontend web application may send requests to a backend service to process transactions for users.

Doing this isn't difficult in a static environment. Applications may use a known hostname for other services, perhaps kept in a configuration file that you update as needed.

But with a dynamic infrastructure where the locations of services and servers are fluid, a more responsive way of finding services is needed.

A few popular discovery mechanisms are:

Hardcoded IP addresses

Allocate IP addresses for each service. For example, the monitoring server always runs on `192.168.1.5`. If you need to change the address or run multiple instances of a service (for example, for a controlled rollout of a major upgrade), you need to rebuild and redeploy applications.

Hostfile entries

Use server configuration to generate the */etc/hosts* file (or equivalent) on each server, mapping service names to the current IP address. This method is a messier alternative to DNS, but I've seen it used to work around legacy DNS implementations.[7]

DNS (Domain Name System)

Use DNS entries to map service names to their current IP address, either using DNS entries managed by code, or DDNS (Dynamic DNS). DNS is a mature, well-supported solution to the problem.

Resource tags

Infrastructure resources are tagged to indicate what services they provide, and the context such as environments. Discovery involves using the platform API to look up resources with relevant tags. Care should be taken to avoid coupling application code to the infrastructure platform.

Configuration registry

Application instances can maintain the current connectivity details in a centralized registry (see "Configuration Registry" on page 99), so other applications can look it up. This may be helpful when you need more information than the address; for example, health or other status indications.

Sidecar

A separate process runs alongside each application instance. The application may use the sidecar as a proxy for outbound connections, a gateway for inbound connections, or as a lookup service. The sidecars will need some method to do network discovery itself. This method could be one of the other discovery mechanisms, or it might use a different communication protocol.[8] Sidecars are usually part of a service mesh (I discuss service meshes in "Service Mesh" on page 244), and often provide more than service discovery. For example, a sidecar may handle authentication, encryption, logging, and monitoring.

API gateway

An API gateway is a centralized HTTP service that defines routes and endpoints. It usually provides more services than this; for example, authentication,

[7] A typical example of an "unsophisticated DNS implementation" is organizations that don't give teams access to change DNS entries. This is usually because the organization doesn't have modern automation and governance processes that make it safe to do so.

[8] The documentation for HashiCorp Consul explains how its sidecars communicate.

encryption, logging, and monitoring. In other words, an API gateway is not unlike a sidecar, but is centralized rather than distributed.[9]

Avoid Hard Boundaries Between Infrastructure, Runtime, and Applications

In theory, it could be useful to provide application runtimes as a complete set of services to developers, shielding them from the details of the underlying infrastructure. In practice, the lines are much fuzzier than presented in this model. Different people and teams need access to resources at different levels of abstraction, with different levels of control. So you should not design and implement systems with absolute boundaries, but instead define pieces that can be composed and presented in different ways to different users.

Conclusion

The purpose of infrastructure is to run useful applications and services. The guidance and ideas in this book should help you in providing collections of infrastructure resources in the shapes needed to do this. An application-driven approach to infrastructure focuses on the runtime requirements of applications, helping you to design stacks, servers, clusters, and other middle-tier constructs for running applications.

[9] My colleagues have expressed concerns about the tendency to centralize business logic in API gateways. See Overambitious API gateways in the ThoughtWorks Technology Radar for more detail.

CHAPTER 11

Building Servers as Code

Infrastructure as Code first emerged as a way to configure servers. Systems administrators wrote shell, batch, and Perl scripts. CFEngine pioneered the use of declarative, idempotent DSLs for installing packages and managing configuration files on servers, and Puppet and Chef followed. These tools assumed you were starting with an existing server—often a physical server in a rack, sometimes a virtual machine using VMware, and later on, cloud instances.

Now we either focus on infrastructure stacks in which servers are only one part or else we work with container clusters where servers are an underlying detail.

But servers are still an essential part of most application runtime environments. Most systems that have been around for more than a few years run at least some applications on servers. And even teams running clusters usually need to build and run servers for the host nodes.

Servers are more complex than other types of infrastructure, like networking and storage. They have more moving parts and variation, so most systems teams still spend quite a bit of their time immersed in operating systems, packages, and configuration files.

This chapter explains approaches to building and managing server configuration as code. It starts with the contents of servers (what needs to be configured) and the server life cycle (when configuration activities happen). It then moves on to a view of server configuration code and tools. The central content of this chapter looks at different ways to create server instances, how to prebuild servers so you can create multiple consistent instances, and approaches to applying server configuration across the server life cycle.

It can be helpful to think about the life cycle of a server as having several transition phases, as illustrated in Figure 11-1.

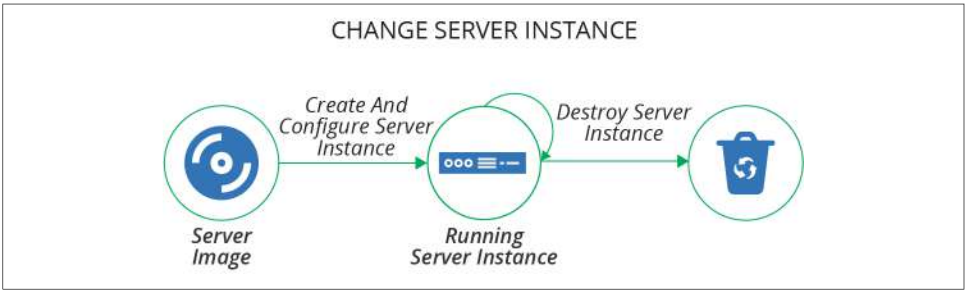


Figure 11-1. The basic server life cycle

The basic life cycle shown here has three transition phases:

1. Create and configure a server instance
2. Change an existing server instance
3. Destroy a server instance

This chapter describes creating and configuring servers. Chapter 12 explains how to make changes to servers, and Chapter 13 discusses creating and updating server images that you can use to create server instances.

What's on a Server

It's useful to think about the various things on a server and where they all come from.

One way of thinking of the stuff on a server is to divide it into software, configuration, and data. These categories, as described in Table 11-1, are useful for understanding how configuration management tools should treat a particular file or set of files.

The difference between configuration and data is whether automation tools automatically manage what's inside the file. So even though a system log is essential for infrastructure, an automated configuration tool treats it as data. Similarly, if an application stores things like its accounts and preferences for its users in files on the server, the server configuration tool treats that file as data.

Table 11-1. *Types of things on a server*

Type of thing	Description	How configuration management treats it
Software	Applications, libraries, and other code. This doesn't need to be executable files; it could be pretty much any files that are static and don't tend to vary from one system to another. An example of this is time zone data files on a Linux system.	Makes sure it's the same on every relevant server; doesn't care what's inside.
Configuration	Files used to control how the system and/or applications work. The contents may vary between servers, based on their roles, environments, instances, and so on. These files are managed as part of the infrastructure, rather than configuration managed by applications themselves. For example, if an application has a UI for managing user profiles, the data files that store the user profiles wouldn't be considered configuration from the infrastructure's point of view; instead, this would be data. But an application configuration file that is stored on the filesystem and managed by the infrastructure would be considered configuration in this sense.	Builds the file contents on the server; ensures it is consistent.
Data	Files generated and updated by the system and applications. The infrastructure may have some responsibility for this data, such as distributing it, backing it up, or replicating it. But the infrastructure treats the contents of the files as a black box, not caring about their contents. Database data files and logfiles are examples of data in this sense.	Naturally occurring and changing; may need to preserve it, but won't try to manage what's inside.

Where Things Come From

The software, configuration, and data that comprise a server instance may be added when the server is created and configured, or when the server is changed. There are several possible sources for these elements (see Figure 11-2):

Base operating system
The operating system installation image could be a physical disk, an ISO file, or a stock server image. The OS installation process may select optional components.

OS package repositories
The OS installation, or a post-installation configuration step, can run a tool that downloads and installs packages from one or more repositories (see Table 10-1 for more on OS package formats). OS vendors usually provide a repository with packages they support. You can use third-party repositories for open source or commercial packages. You may also run an internal repository for packages developed or curated in-house.

Language, framework, and other platform repositories
In addition to OS-specific packages, you may install packages for languages or frameworks like Java libraries or Ruby gems. As with OS packages, you may pull packages from repositories managed by language vendors, third parties, or internal groups.

Nonstandard packages
Some vendors and in-house groups provide software with their own installers, or which involve multiple steps other than running a standard package management tool.

Separate material
You may add or modify files outside of an application or component; for example, adding user accounts or setting local firewall rules.

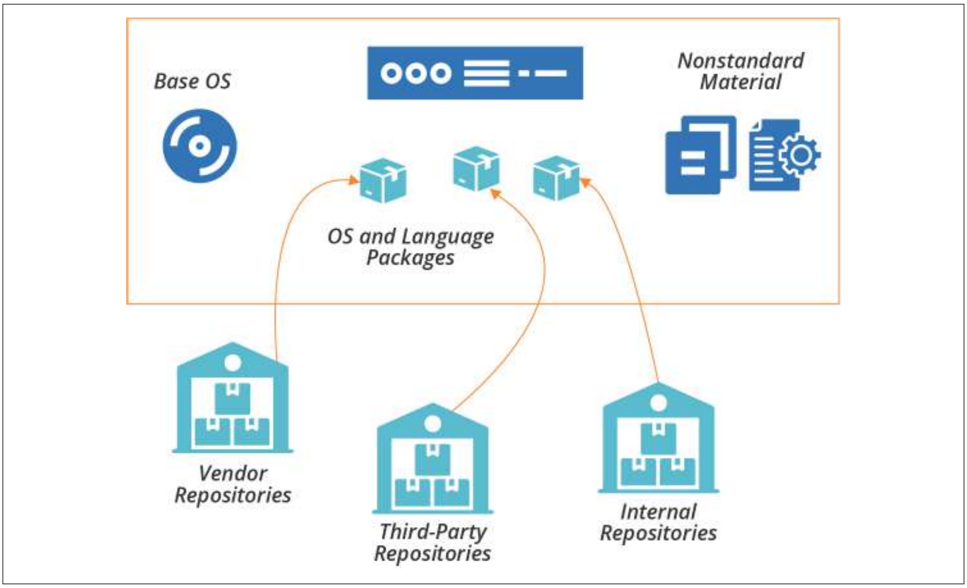


Figure 11-2. Packages and other elements of a server instance

The next question is how to add things to server instances when you create or change them.

Server Configuration Code

The first generation of Infrastructure as Code tools focused on automated server configuration. Some of the tools in this space are:

- Ansible
- CFEngine
- Chef
- Puppet
- Saltstack

Many of these tools use an agent installed on each server following the pull configuration pattern (see "Pattern: Pull Server Configuration" on page 200). They provide an agent that you can either install as a service or run periodically from a cron job. Other tools are designed to run from a central server and connect to each managed server, following the push pattern ("Pattern: Push Server Configuration" on page 198).

You can implement either the push or pull pattern with most of these tools. If you use a tool like Ansible, which is designed for the push model, you can preinstall it on servers and run it from cron. If, on the other hand, you use a tool like Chef or Puppet, which provides an agent to run with the pull model, you can instead run a command from a central server that logs in and executes the client on each machine. So the tool is not a constraint on which pattern you use.

Many server configuration tool makers offer repository servers to host configuration code. Examples of these include Ansible Tower, Chef Server, and Puppetmaster. These may have additional functionality, such as a configuration registry (see "Configuration Registry" on page 99), dashboards, logging, or centralized execution.

Arguably, choosing a vendor that provides an all-in-one ecosystem of tools simplifies things for an infrastructure team. However, it's useful if elements of the ecosystem can be swapped out for different tools so the team can choose the best pieces that fit their needs. For instance, if you use multiple tools that integrate with a configuration registry, you may prefer to use a standalone, general-purpose configuration registry rather than one tied to your server configuration tool.

The codebase is a significant part of a modern infrastructure team's life. The principles and guidelines from Chapter 4 apply to server code, stack code, etc.

Server Configuration Code Modules

Server configuration code, as with any code, seems to enjoy growing into a sprawling mess over time. It takes discipline and good habits to avoid this. Fortunately, these tools give you ways to organize and structure code into different units.

Most server configuration tools let you organize code into modules. For example, Ansible has playbooks, Chef groups recipes into cookbooks, and Puppet has modules containing manifests. You can organize, version, and release these modules individually.

Roles are a way to mark a group of modules to apply to a server, defining its purpose. Unlike other concepts, most of the tool vendors seem to agree on using the term *role* for this, rather than making up their own words.[1]

[1] It would have been entirely in character for the folks at Chef to call a role a *hat*, so they could talk about servers wearing a *chef hat*. I'm grateful they didn't.

Many of these tools also support extensions to their core resource model. A tool's language provides a model of server resource entities, like users, packages, and files. For example, you could write a Chef Lightweight Resource Provider (LWRP) to add a Java application resource that installs and configures applications written by your development teams. These extensions are similar to stack modules (see Chapter 16).

See Chapter 15 for a more general discussion on modularizing infrastructure, including guidance on good design.

Designing Server Configuration Code Modules

You should design and write each server configuration code module (for example, each cookbook or playbook) around a single, cohesive concern. This advice follows the software design principle of *separation of concerns*. A common approach is to have a separate module for each application.

As an example, you might create a module for managing an application server like Tomcat. The code in the module would install the Tomcat software, user accounts and groups for it to run under, and folders with the correct permissions for logs and other files. It would build the Tomcat configuration files, including configuring ports and settings. The module code would also integrate Tomcat into various services such as log aggregation, monitoring, and process management.

Many teams find it useful to design server configuration modules differently based on their use. For example, you can divide your modules into library modules and application modules (with Chef, this would be library cookbooks and application cookbooks).

A library module manages a core concept that can be reused by application modules. Taking the example of a Tomcat module, you could write the module to take parameters, so the module can be used to configure an instance of Tomcat optimized for different purposes.

An application module, also called a wrapper module, imports one or more library modules and sets their parameters for a more specific purpose. The ShopSpinner team could have one Servermaker module that installs Tomcat to run its product catalog application and a second one that installs Tomcat for its customer management application.[2] Both of those modules use a shared library that installs Tomcat.

[2] You may recall from Chapter 4 that Servermaker is a fictional server configuration tool similar to Ansible, Chef, and Puppet.

Versioning and Promoting Server Code

As with any code, you should be able to test and progress server configuration code between environments before applying it to your production systems. Some teams version and promote all of their server code modules together, as a single unit, either building them together (see "Pattern: Build-Time Project Integration" on page 327), or building and testing them separately before combining them for use in production environments (see "Pattern: Delivery-Time Project Integration" on page 330). Others version and deliver each module independently (see "Pattern: Apply-Time Project Integration" on page 333).

If you manage each module as a separate unit, you need to handle any dependencies between them all. Many server configuration tools allow you to specify the dependencies for a module in a descriptor. In a Chef cookbook, for example, you list dependencies in the `depends` field of the cookbook's metadata file. The server configuration tool uses this specification to download and apply dependent modules to a server instance. Dependency management for server configuration modules works the same way as software package management systems like RPMs and Python pip packages.

You also need to store and manage different versions of your server configuration code. Often, you have one version of the code currently applied to production environments, and other versions being developed and tested.

See Chapters 18 and 19 for a deeper discussion of organizing, packaging, and delivering your infrastructure code.

Server Roles

As mentioned earlier, a server role is a way to define a group of server configuration modules to apply to a server. A role may also set some default parameters. For example, you may create an `application-server` role:

```
role: application-server
  server_modules:
    - tomcat
    - monitoring_agent
    - logging_agent
    - network_hardening
  parameters:
    - inbound_port: 8443
```

Assigning this role to a server applies the configuration for Tomcat, monitoring and logging agents, and network hardening. It also sets a parameter for an inbound port, presumably something that the `network_hardening` module uses to open that port in the local firewall while locking everything else down.

Roles can become messy, so you should be deliberate and consistent in how you use them. You may prefer to create fine-grained roles that you combine to compose specific servers. You assign multiple roles to a given server, such as ApplicationServer, MonitoredServer, and PublicFacingServer. Each of these roles includes a small number of server modules for a narrow purpose.

Alternatively, you may have higher-level roles, which include more modules. Each server probably has only one role, which may be quite specific; for example, ShoppingServiceServer or JiraServer.

A common approach is to use role inheritance. You define a base role that includes software and configuration common to all servers. This role might have networking hardening, administrative user accounts, and agents for monitoring and logging. More specific roles, such as for application servers or container hosts, include the base role and then add a few additional server configuration modules and parameters:

```
role: base-role
  server_modules:
    - monitoring_agent
    - logging_agent
    - network_hardening

role: application-server
  include_roles:
    - base_role
  server_modules:
    - tomcat
  parameters:
    - inbound_port: 8443

role: shopping-service-server
  include_roles:
    - application-server
  server_modules:
    - shopping-service-application
```

This code example defines three roles in a hierarchy. The shopping-service-server role inherits everything from the application-server role, and adds a module to install the specific application to deploy. The application-server role inherits from the base-role, adding the Tomcat server and network port configuration. The base-role defines a core set of general-purpose configuration modules.

Testing Server Code

Chapter 8 explained how automated testing and CD theory applies to infrastructure, and Chapter 9 described approaches for implementing this with infrastructure stacks. Many of the concepts in both of those chapters apply to testing server code.

Progressively Testing Server Code

There is a self-reinforcing dynamic between code design and testing. It's easier to write and maintain tests for a cleanly structured codebase. Writing tests, and keeping them all passing, forces you to maintain that clean structure. Pushing every change into a pipeline that runs your tests helps your team keep the discipline of continuously refactoring to minimize technical debt and design debt.

The structure described earlier, of server roles composed of server configuration modules, which may themselves be organized into library modules and application modules, aligns nicely to a progressive testing strategy (see "Progressive Testing" on page 115). A series of pipeline stages can test each of these in increasing stages of integration, as shown in Figure 11-3.

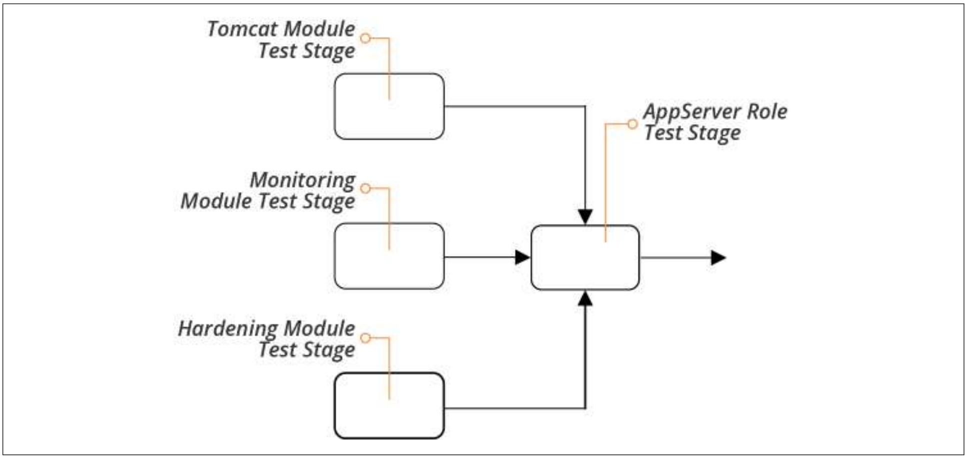

Tomcat Module Test Stage
AppServer Role Test Stage
Monitoring Module Test Stage
Hardening Module Test Stage

Figure 11-3. Progressively testing server code modules

A separate stage tests each code module whenever someone commits a change to that module. The server role also has a test stage. This stage runs the tests whenever one of the modules used by the role changes and passes its own stage. The role-testing stage also runs when someone commits a change to the role code, for example adding or removing a module, or changing a parameter.

What to Test with Server Code

It's tricky to decide what to test with server code. This goes back to the question of whether there's any value in testing declarative code (see "Challenge: Tests for Declarative Code Often Have Low Value" on page 110). Server code modules, especially in a well-designed codebase, tend to be small, simple, and focused. A module that installs a Java JVM, for example, may be a single statement, with a few parameters:

```
package:
  name: java-${JAVA_DISTRIBUTION}
  version: ${JAVA_VERSION}
```

In practice, even simple-seeming package installation modules may have more code than this, customizing file paths, configuration files, and perhaps adding a user account. But there's often not much to test that doesn't simply restate the code itself.

Tests should focus on common issues, variable outcomes, and combinations of code.

Consider common issues, things that tend to go wrong in practice, and how to sanity check. For a simple package installation, you might want to check that the command is available in the default user path. So a test would invoke the command, and make sure it was executed:

```
given command 'java -version' {
  its(exit_status) { should_be 0 }
}
```

If the package may have radically different outcomes depending on the parameters passed to it, you should write tests to assure you that it behaves correctly in different cases. For the JVM installation package, you could run the test with different values for the Java distribution, for example making sure that the `java` command is available no matter which distribution someone chooses.

In practice, the value of testing is higher as you integrate more elements. So you may write and run more tests for the application server role than for the modules the role includes.

This is especially true when those modules integrate and interact with one another. A module that installs Tomcat isn't likely to clash with one that installs a monitoring agent, but a security hardening module might cause an issue. In this case, you may want to have tests that run on the application server role, to confirm that the application server is up and accepting requests even after ports have been locked down.

How to Test Server Code

Most automated server code tests work by running commands on a server or container instance, and checking the results. These tests may assert the existence and status of resources, such as packages, files, user accounts, and running processes. Tests may also look at outcomes; for example, connecting to a network port to prove whether a service returns the expected results.

Popular tools for testing conditions on a server include Inspec, Serverspec, and Terratest.

The strategies for testing full infrastructure stacks include running tests offline and online (see "Offline Testing Stages for Stacks" on page 131 and "Online Testing

Stages for Stacks" on page 134). Online testing involves spinning things up on the infrastructure platform. You can normally test server code offline using containers or local virtual machines.

An infrastructure developer working locally can create a container instance or local VM with a minimal operating system installation, apply the server configuration code, and then run the tests.

Pipeline stages that test server code can do the same, running the container instance or VM on the agent (for example, the Jenkins node running the job). Alternatively, the stage could spin up a standalone container instance on a container cluster. This works well when the pipeline orchestration is itself containerized.

You can follow the guidance for stacks to orchestrate server code testing (see "Test Orchestration" on page 149). This includes writing scripts that set up the testing prerequisites, such as container instances, before running the tests and reporting results. You should run the same scripts to test locally that you use to test from your pipeline service, so the results are consistent.

Creating a New Server Instance

The basic server life cycle described earlier starts with creating a new server instance. A more extended life cycle includes steps for creating and updating server images that you can create server instances from, but we'll leave that topic for another chapter (Chapter 13). For now, our life cycle starts with creating an instance, as shown in Figure 11-4.

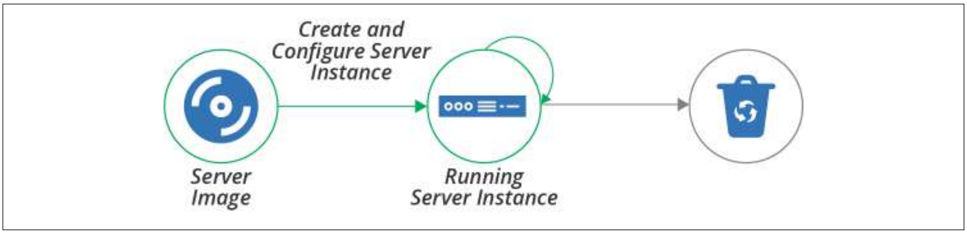


Figure 11-4. Creating a server instance

The full server creation process includes provisioning the server instance so that it is fully ready for use. Some of the activities involved in creating and provisioning a server instance are:

- Allocate infrastructure resources to instantiate the server instance. To do this, a physical server is selected from a pool or a host server is selected to run it as a virtual machine. For a virtual machine, the hypervisor software on the host allocates memory and other resources. The process may also allocate storage for disk volumes for the server instance.
- Install the operating system and initial software. The OS may be copied onto a disk volume, as when booting from a server image like an AMI. Or the new instance may execute an installation process that selects and copies files to the new instance, maybe using a scriptable installer. Examples of scripted OS installers include Red Hat Kickstart, Solaris JumpStart, Debian Preseed, and the Windows installation answer file.
- Additional configuration is applied to the server instance. In this step, the process runs the server configuration tool, following a pattern described in "How to Apply Server Configuration Code" on page 198.
- Configure and attach networking resources and policies. This process can include assigning the server to a network address block, creating routes, and adding firewall rules.
- Register the server instance with services. For example, add the new server to a monitoring system.

These aren't mutually exclusive—your server creation process may use one or more methods for running these different activities.

There are several mechanisms that you might use to trigger the creation of a server instance. Let's go over each of these mechanisms.

Hand-Building a New Server Instance

Infrastructure platforms provide tools to create new servers, usually a web UI, command-line tool, and sometimes a GUI application. Each time you create a new server, you select the options you want, including the source image, resources to allocate, and networking details:

```
$ mycloud server new \
    --source-image=stock-linux-1.23 \
    --memory=2GB \
    --vnet=appservers
```

While it's useful to play with UIs and command-line tools to experiment with a platform, it's not a good way to create servers that people need. The same principles for creating and configuring stacks discussed previously (see "Patterns for Configuring Stacks" on page 80) apply for servers. Setting options manually (as described in "Antipattern: Manual Stack Parameters" on page 81) encourages mistakes, and leads to inconsistently configured servers, unpredictable systems, and too much maintenance work.

Using a Script to Create a Server

You can write scripts that create servers consistently. The script wraps a command-line tool or uses the infrastructure platform's API to create a server instance, setting the configuration options in the script code or configuration files. A script that creates servers is reusable, consistent, and transparent. This is the same concept as the scripted parameters pattern for stacks (see "Pattern: Scripted Parameters" on page 84):

```
mycloud server new \
    --source-image=stock-linux-1.23 \
    --memory=2GB \
    --vnet=appservers
```

This script is the same as the previous command line example. But because it's in a script, other people on my team can see how I created the server. They can create more servers and be confident that they will behave the same as the one I created.

Before Terraform and other stack management tools emerged, most teams I worked with wrote scripts for creating servers. We usually made the scripts configurable with configuration files, as with the stack configuration pattern (see "Pattern: Stack Configuration Files" on page 87). But we spent too much time improving and fixing these scripts.

Using a Stack Management Tool to Create a Server

With a stack management tool, as discussed in Chapter 5, you can define a server in the context of other infrastructure resources, as shown in Example 11-1. The tool uses the platform API to create or update the server instance.

Example 11-1. Stack code that defines a server

```
server:
  source_image: stock-linux-1.23
  memory: 2GB
  vnet: ${APPSERVER_VNET}
```

There are several reasons why using a stack tool to create servers is so handy. One is that the tool takes care of logic that you would have to implement in your own script, like error checking. Another advantage of a stack is that it handles integration with other infrastructure elements; for example, attaching the server to network structures and storage defined in the stack. The code in Example 11-1 sets the `vnet` parameter using a variable, ${APPSERVER_VNET}, that presumably refers to a network structure defined in another part of the stack code.

Configuring the Platform to Automatically Create Servers

Most infrastructure platforms can automatically create new server instances in specific circumstances. Two common cases are *auto-scaling*, adding servers to handle load increases, and *auto-recovery*, replacing a server instance when it fails. You can usually define this with stack code, as in Example 11-2.

Example 11-2. Stack code for automated scaling

```
server_cluster:
  server_instance:
    source_image: stock-linux-1.23
    memory: 2GB
    vnet: ${APPSERVER_VNET}
  scaling_rules:
    min_instances: 2
    max_instances: 5
    scaling_metric: cpu_utilization
    scaling_value_target: 40%
  health_check:
    type: http
    port: 8443
    path: /health
    expected_code: 200
    wait: 90s
```

This example tells the platform to keep at least 2 and at most 5 instances of the server running. The platform adds or removes instances as needed to keep the CPU utilization across them close to 40%.

The definition also includes a health check. This check makes an HTTP request to /health on port 8443, and considers the server to be healthy if the request returns a 200 HTTP response code. If the server doesn't return the expected code after 90 seconds, the platform assumes the server has failed, and so destroys it and creates a new instance.

Using a Networked Provisioning Tool to Build a Server

In Chapter 3, I mentioned bare-metal clouds, which dynamically provision hardware servers. The process for doing this usually includes these steps:

1. Select an unused physical server and trigger it to boot in a "network install" mode supported by the server firmware (e.g., PXE boot).[3]
2. The network installer boots the server from a simple bootstrap OS image to initiate the OS installation.
3. Download an OS installation image and copy it to the primary hard drive.
4. Reboot the server to boot the OS in setup mode, running an unattended scripted OS installer.

There are a number of tools to manage this process, including:

- Crowbar
- Cobbler
- FAI, or Fully Automatic Installation
- Foreman
- MAAS
- Rebar
- Tinkerbell

Instead of booting an OS installation image, you could instead boot a prepared server image. Doing this creates the opportunity to implement some of the other methods for preparing servers, as described in the next section.

FaaS Events Can Help Provision a Server

FaaS serverless code can play a part in provisioning a new server. Your platform can run code at different points in the process, before, during, and after a new instance is created. Examples include assigning security policies to a new server, registering it with a monitoring service, or running a server configuration tool.

Prebuilding Servers

As discussed earlier (see "Where Things Come From" on page 171), there are several sources of content for a new server, including the operating system installation, packages downloaded from repositories, and custom configuration files and application files copied to the server.

[3] Often, triggering a server to use PXE to boot a network image requires pressing a function key while the server starts. 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.

While you can assemble all of these when you create the server, there are also several ways to prepare server content ahead of time. These approaches can optimize the process of building a server, making it faster and simpler to do, and making it easier to create multiple, consistent servers. What follows are a few approaches to doing this. The section after this then explains options for applying configuration before, during, and after the provisioning process.

Hot-Cloning a Server

Most infrastructure platforms make it simple to duplicate a running server. Hot-cloning a server in this way is quick and easy, and gives you servers that are consistent at the point of cloning.

Cloning a running server can be useful when you want to explore or troubleshoot a server without interfering with its operation. But it carries some risks. One is that a copy of a production server that you make for experimentation might affect production. For example, if a cloned application server is configured to use the production database, you might accidentally pollute or damage production data.

Cloned servers running in production usually contain historical data from the original server, such as logfile entries. This data pollution can make troubleshooting confusing. I've seen people spend time puzzling over error messages that turned out not to have come from the server they were debugging.

And cloned servers are not genuinely reproducible. Two servers cloned from the same parent at different points in time will be different, and you can't go back and build a third server and be entirely sure whether and how it is different from any of the others. Cloned servers are a source of configuration drift (see "Configuration Drift" on page 17).

Using a Server Snapshot

Rather than building new servers by directly cloning a running server, you can take a snapshot of a running server and build your servers from that. Again, most infrastructure platforms provide commands and API calls to snapshot servers easily, creating a static image. This way, you can create as many servers as you like, confident that each one is identical to the starting image.

Creating a server from a snapshot taken from a live server has many of the other disadvantages of hot-cloning, however. The snapshot may be polluted with logs, configuration, and other data from the original server. It's more effective to create a clean server image from scratch.

Creating a Clean Server Image

A server image is a snapshot that you create from clean, known sources so that you can create multiple, consistent server instances. You may do this using the same infrastructure platform features that you would for snapshotting a live server, but the original server instance is never used as a part of your overall systems. Doing this ensures that every new server is clean.

A typical process for building a server image is:

1. Create a new server instance from a known source, such as an operating system vendor's installation image, or an image provided by your infrastructure platform for this purpose.
2. Apply server configuration code or run other scripts on the server. These may install standard packages and agents that you want on all of your servers, harden configuration for security purposes, and apply all of the latest patches.
3. Snapshot the server, creating an image that you can use as a source for creating multiple server instances. Doing this may involve steps to mark the snapshot as an image for creating new servers. It may also involve tagging and versioning to help manage a library of multiple server images.

People sometimes call server images *golden images*. Although some teams build these images by hand, perhaps following a written checklist of steps, readers of this book immediately see the benefits of automating this process, managing server images as code. Building and delivering server images is the topic of Chapter 13.

Configuring a New Server Instance

This chapter has described what the elements of a server are, where they come from, ways to create a new server instance, and the value of prebuilding server images. The last piece of the server creation and provisioning process is applying automated server configuration code to new servers. There are several points in the process where you can do this, as illustrated in Figure 11-5.

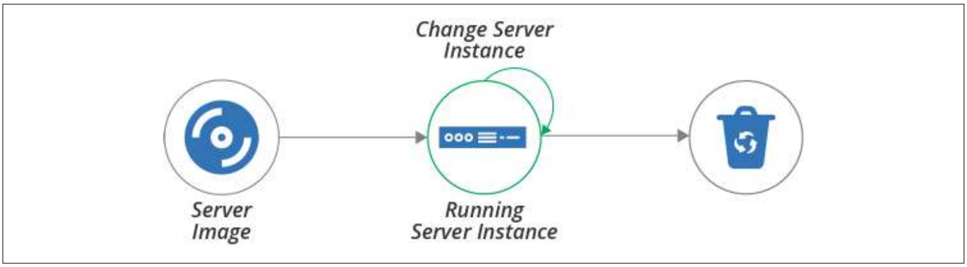


Figure 11-5. Where configuration can be applied in the server life cycle

Configuration can be applied when creating the server image, when creating the server instance from the image, and when the server is running:

Configuring a server image
: You apply configuration when building a server image that will be used to create multiple server instances. Think of this as configuring one time, and using it many times. This is often called *baking* a server image.

Configuring a new server instance
: You apply configuration when creating a new server instance. Think of this as configuring many times. This is sometimes called *frying* a server instance.

Configuring a running server instance
: You apply configuration to a server that is already in use. One common reason to do this is to make a change, like applying security patches. Another reason is to revert any changes made outside the automated configuration to enforce consistency. Some teams also apply configuration to transform an existing server; for example, turning a web server into an application server.

The last of these, configuring a running server instance, is usually done to change a server, which is the topic of Chapter 12. The first two are options for applying configuration when creating a new server, so are firmly in the scope of this chapter. The main question is determining the right time to apply configuration for a new server—frying it into each new server instance, or baking it into the server image?

Frying a Server Instance

As explained, frying a server involves applying configuration when you create the new server instance. You can take this to an extreme, keeping each server image to a bare minimum, and building everything specific to a given server into the live image. Doing this means new servers always have the latest changes, including system patches, the most recent software package versions, and up-to-date configuration options. Frying a server is an example of delivery-time integration (see "Pattern: Delivery-Time Project Integration" on page 330).

This approach simplifies image management. There are not many images, probably only one for each combination of hardware and OS version used in the infrastructure—for example, one for 64-bit Windows 2019 and one each for 32-bit and 64-bit Ubuntu 18.x. The images don't need to be updated very often, as there isn't much on them that changes. And in any case, you can apply the latest patches when you provision each server.

Some teams I've been with have focused on frying early in our adoption of infrastructure automation. Setting up the tooling and processes for managing server images is a lot of effort. We put that work on our backlog to do after building our core infrastructure management.

In other cases, frying makes sense because servers are highly variable. A hosting company I know lets customers choose from a large number of customizations for servers. The company maintains minimalist base server images and puts more effort into the automation that configures each new server.

There are several potential issues with installing and configuring elements of a server when creating each instance. These include:

Speed
: Activities that occur when building a server add to the creation time. This added time is a particular issue for spinning up servers to handle spikes in load or to recover from failures.

Efficiency
: Configuring a server often involves downloading packages over the network from repositories. This can be wasteful and slow. For example, if you need to spin up 20 servers in a short time, having each one of them download the same patches and application installers is wasteful.

Dependencies
: Configuring a server usually depends on artifact repositories and other systems. If any of these are offline or unreachable, you can't create a new server. This can be especially painful in an emergency scenario where you need to rebuild a large number of servers quickly. In these situations, network devices or repositories may also be down, creating a complex ordered graph of which systems to restart or rebuild first to enable the next set of systems to come up.

Baking Server Images

At the other end of the server creation spectrum is configuring nearly everything into the server image. Building new servers is then very quick and straightforward since you only need to apply instance-specific configuration. Baking a server image is an example of build-time integration (see "Pattern: Build-Time Project Integration" on page 327).

You can see the advantages of baking from the disadvantages of frying. Baking configuration into server images is especially appropriate for systems that use a large number of similar server instances, and when you need to be able to create servers frequently and quickly.

One challenge with baking server images is that you need to set up the tooling and automated processes to make it easy to update and roll out new versions. For example, if an important security patch is released for your operating system or for key packages that are baked into your server images, you want to be able to build new images and roll them out to your existing servers quickly and with minimal disruption. Doing this is covered in Chapter 13.

Another issue with baking server images is speed. Even with a mature automated process for updating images, it can take many minutes—10 to 60 minutes is common—to build and release a new image. You can mitigate this by rolling changes out to running servers (the topic of Chapter 12), or by having a process that combines baking and frying.

Combining Baking and Frying

In practice, most teams use a combination of baking and frying to configure new servers. You can balance which activities to configure into server images, and which to apply when creating a server instance. You may apply some configuration elements at both parts of the process.

The main considerations for deciding when is the right time to apply a particular configuration are the time it takes and the frequency of change. Things that take longer to apply, and which change less often, are clear candidates for baking into the server image. For example, you can install application server software on a server image, making it much quicker to spin up multiple server instances and saving network bandwidth in the process.

On the flip side of this trade-off, something that is quicker to install, or changes more often, is better to fry. An example of this is applications developed in-house. One of the most common use cases for spinning up new servers on demand is testing as part of a software release process.

Highly productive development teams may push a dozen or more new builds of their application every day, relying on a CI process to automatically deploy and test each build. Baking a new server image for each new application build is too slow for this kind of working pace, so it's more efficient to deploy the application when creating the test server.

Another way that teams combine baking and frying is to bake as much as possible onto server images, but to fry updated versions. A team may bake updated server images at a slower pace; for example, weekly or monthly. When they need to update something that they usually bake onto the image, such as a security patch or configuration improvement, they can put this into the server creation process to fry it on top of the baked image.

When the time comes to bake an updated image, they fold the updates into it and remove it from the creation process. This method lets the team incorporate new changes quickly, with less overhead. Teams often use this in combination with continuously applying code (see "Pattern: Continuous Configuration Synchronization" on page 194) to update existing servers without rebuilding them.

Applying Server Configuration When Creating a Server

Most of the tooling used to create servers in the ways discussed earlier (see "Creating a New Server Instance" on page 179), whether a command-line tool, platform API call, or a stack management tool, provides a way to apply server configuration code. For example, a stack management tool should have syntax to support popular tools, or to run arbitrary commands on a new server instance, as with Example 11-3.

Example 11-3. Stack code that runs my fictional server configuration tool

```
server:
  source_image: stock-linux-1.23
  memory: 2GB
  vnet: ${APPSERVER_VNET}
  configure:
    tool: servermaker
    code_repo: servermaker.shopspinner.xyz
    server_role: appserver
    parameters:
      app_name: catalog_service
      app_version: 1.2.3
```

This code runs the Servermaker tool, passing it the hostname of the server that hosts server configuration code, the role to apply to the server (`appserver`), and some parameters to pass to the server configuration code (`app_name` and `app_version`).

Some tools also allow you to embed server configuration code directly into the code for the stack, or shell commands to execute. It can be tempting to use this to implement the server configuration logic, and for simple needs, this may be fine. But in most cases, this code grows in size and complexity. So it's better to extract the code to keep your codebase clean and maintainable.

Conclusion

This chapter has covered various aspects of creating and provisioning new servers. The types of things on a server include software, configuration, and data. These are typically drawn from the base operating system installation and packages from various repositories, including those managed by the OS and language vendors, third parties, and internal teams. You typically build a server by combining content from a server image and using a server configuration tool to apply additional packages and configuration.

To create a server, you can run a command-line tool or use a UI, but it’s preferable to use a code-driven process. These days you’re less likely to write a custom script to do this and more likely to use your stack management tool. Although I describe some different approaches for building servers, I usually recommend building server images.

CHAPTER 12
Managing Changes to Servers

Many organizations and teams focus on processes and tools to build servers and other infrastructure, but neglect changes. When a change is needed—to fix a problem, apply security patches, or upgrade software—they treat it as an unusual event. If every change is an exception, then you can't automate it. This mindset is why so many organizations have inconsistent, flaky systems. It's why many of us spend our time switching between picking up dull tickets and fighting fires.

The only constant thing about our systems is that they change. If we define our systems as code, run them on dynamic infrastructure platforms, and deliver that code across our systems with a change pipeline, then we can make changes routine and easy. If our systems are only ever created and changed through code and pipelines, then we can ensure their consistency and be sure they align with whatever policies we need.

"What's on a Server" on page 170 and "Where Things Come From" on page 171 describe what's on a server and where it all comes from. Everything on a given server comes from a defined source, whether it's an OS installation, system package repository, or server configuration code. Any change you need to make to a server is a change to one of these things.

This chapter is about how to change things on servers by changing the code that defines where the thing comes from and applying it one way or another. By implementing a reliable, automated change process for your servers, you ensure that you can roll changes out across your estate rapidly and reliably. You can keep all of your servers up-to-date with the latest approved packages and configuration, with minimal effort.

There are several patterns for applying configuration code to servers, including applying each change as it happens, continuously synchronizing it, and rebuilding servers to change them. Another dimension of making changes is how to run your tool to apply changes to a server, whether to push or pull configuration. Finally, there are several other events in a server's life cycle, from pausing to rebuilding to deleting a server.

Change Management Patterns: When to Apply Changes

There is one antipattern and two patterns for deciding when to apply changes to a server instance.

Antipattern: Apply On Change

Also known as: ad hoc automation.

With the *apply on change* antipattern, configuration code is only applied to a server when there is a specific change to apply.

For example, consider a team that runs multiple Tomcat application servers. The team members run an Ansible playbook to install and configure Tomcat when they create a new server instance, but once the server is running, they don't run Ansible until they need to. When a new version of Tomcat is released, they update their playbook and apply it to their servers.

In the most extreme version of this antipattern, you only apply code to the server that you specifically intend to change.

The team in our example notices that one of its application servers is seeing much higher traffic, and the load is making Tomcat unstable. The team members make some changes to their playbook to optimize their Tomcat configuration for higher load and then apply it to the server that has the issue. But they don't apply the playbook to the other application servers, because those don't need the change.

Motivation

Systems and network administrators have traditionally managed servers by hand. When you need to make a change, you log in to the relevant server and make the change. Why would you do it differently? Even people who use scripts tend to write and run the scripts to make a specific change. The apply on change antipattern is an extension of this way of working that happens to use an Infrastructure as Code tool instead of a manual command or on-off script.

Applicability

Applying code only as needed for a specific change may be fine for a single temporary server. But it's not a suitable method for sustainably managing a group of servers.

Consequences

If you only apply configuration code to make a specific change, you may have long gaps where the code is never applied to a particular server instance. When you finally do apply the code, it might fail because of other differences on the server than the one you meant to change.

Things have a habit of changing on a server when you aren't paying attention. Someone may make a change manually—for example, as a quick fix for an outage. Someone else may have patched the system with newer versions of OS or application packages. These fall into the category of quick changes that we're sure won't break anything. And into the category of changes that, a week later, we don't remember making (because it was only a small change) until we've spent several hours debugging something it broke.

The problem becomes worse when we only apply changes to some servers and not others. Consider the previous example where the team applies code to optimize performance to one server. Later on, someone on the team will need to make a different change to the application servers.

When they do, they also apply the previous performance optimization change to the servers that don't already have it, as a side effect of applying the code for the new change. The earlier change might have unexpected effects on other servers. Worse, the person applying the code may have forgotten about the performance optimization, so they take much longer to discover the cause of any problems it creates.

Implementation

People who are used to making changes by hand, or using one-off scripts, tend to use server configuration code the same way. They see the tool—Ansible, Chef, Puppet—as a scripting tool with awkward syntax. Most often, people who do this run the tool by hand from their local computer, rather than having a pipeline or other orchestration service apply code.

Related patterns

People tend to use the apply on change antipattern with the push configuration pattern ("Pattern: Push Server Configuration" on page 198) rather than using pull ("Pattern: Pull Server Configuration" on page 200). The alternatives to this antipattern are continuous synchronization or immutable servers (see "Pattern: Immutable Server" on page 195).

Pattern: Continuous Configuration Synchronization

Also known as: scheduled server configuration update.

Continuous configuration synchronization involves repeatedly and frequently applying configuration code to a server, regardless of whether the code has changed. Doing this reverts or surfaces any unexpected differences that might creep in, whether to the server, or other resources used by the configuration code.

Motivation

We would like to believe that server configuration is predictable. Once I apply code to a server, nothing should change until the next time I apply the code. And if I haven't changed the code, there's no need to apply it. However, servers, and server code, are sneaky.

Sometimes a server changes for obvious reasons, such as someone logging in and making a change by hand. People often make minor changes this way because they don't think it will cause any problems. They are often mistaken about this. In other cases, teams manage some aspects of a server with a different tool or process. For example, some teams use a specialized tool to update and patch servers, especially for security patches.

Even if a server hasn't changed, applying the same server configuration code multiple times can introduce differences. For example, the code may use parameters from a central configuration registry. If one of those parameters changes, the code may do something different the next time it runs on a server.

Packages are another external source of change. If your configuration code installs a package from a repository, it may update the package to a newer version when it becomes available. You can try to specify package versions, but this leads down one of two terrible roads. Down one road, your system eventually includes many out-of-date packages, including ones with security vulnerabilities well known to attackers. Down the other road, you and your team spend enormous amounts of energy manually updating package version numbers in your server code.

By reapplying your server configuration code regularly, on an automated schedule, you ensure that you keep all of your servers configured consistently. You also ensure that any differences, from whatever sources, are applied sooner rather than later.

Applicability

It is easier to implement continuous synchronization than the main alternative, immutable servers. Most Infrastructure as Code tools—Ansible, Chef, and Puppet, for example—are designed with this pattern in mind. It is quicker and less disruptive to apply changes by updating an existing server instance than by building a new instance.

Consequences

When an automated process applies server configuration across your estate, there is a risk that something will break. All of the things that might change unexpectedly, as described earlier in the "Motivation" section, are things that could break a server. To counter this, you should have an effective monitoring system to alert you to problems, and a good process for testing and delivering code before applying changes to production systems.

Implementation

As mentioned before, most server configuration as code tools are designed to run continuously. The specific mechanisms they use are described in "Pattern: Push Server Configuration" on page 198 and in "Pattern: Pull Server Configuration" on page 200.

Most continuous synchronization implementations run on a schedule. These tend to have some way of varying the runtime on different servers so that all of your servers don't wake up and run their configuration at the same time.[1] However, sometimes you want to apply code more quickly, maybe to apply a fix, or to support a software deployment. Different tools have different solutions for doing this.

Related patterns

Continuous synchronization is implemented using either the push ("Pattern: Push Server Configuration" on page 198) or pull ("Pattern: Pull Server Configuration" on page 200) configuration pattern. The alternative to this pattern is immutable servers ("Pattern: Immutable Server" on page 195).

Pattern: Immutable Server

An *immutable server* is a server instance whose configuration is never changed. You deliver changes by creating a new server instance with the changed configuration and using it to replace the existing server.[2]

[1] For an example, see the chef-client `--splay` option.

[2] My colleague Peter Gillard-Moss and former colleague Ben Butler-Cole used immutable servers when they worked on the ThoughtWorks' Mingle SaaS platform.

Motivation

Immutable servers reduce the risk of making changes. Rather than applying a change to a running server instance, you create a new server instance. You have the opportunity to test the new instance, and then swap it out for the previous instance. You can then check that the new instance is working correctly before destroying the original, or swap it back into place if something goes wrong.

Applicability

Organizations that need tight control and consistency of server configuration may find immutable servers useful. For example, a telecommunications company that runs thousands of server images may decide not to apply changes to running servers, preferring to guarantee the stability of their configuration.

Consequences

Implementing immutable servers requires a robust automated process for building, testing, and updating server images (as described in Chapter 13). Your system and application design must support swapping server instances without interrupting service (see "Changing Live Infrastructure" on page 368 for ideas).

Despite the name, immutable servers do change.[3]

Configuration drift may creep in, especially if people can log in to servers and make changes manually rather than using the configuration change process to build new server instances. So teams using immutable servers should be careful to ensure the freshness of running instances. It's entirely possible to combine immutable servers with the apply on change antipattern (see "Antipattern: Apply On Change" on page 192), which can result in servers that run, unchanged, for a long time, without including patches and improvements made to servers built later. Teams should also consider disabling access to servers, or make the ability to make accessing and manually changing services require a "break glass" procedure.[4]

[3] Some people have protested that immutable infrastructure is an invalid approach because aspects of servers, including logs, memory, and process space, are changeable. This argument is similar to dismissing "serverless computing" because it uses servers behind the scenes. The terminology is a metaphor. Despite the imperfection of the metaphor, the approach it describes is useful for many people.

[4] A "break glass" process can be used to gain elevated access temporarily, in an emergency. The process is usually highly visible, to discourage it being used routinely. If people begin relying on a break glass process, the team should assess what tasks are forcing its use, and look for ways to support those tasks without it. See Derek A. Smith's webinar, "Break Glass Theory" for more.

Implementation

Most teams who use immutable servers handle the bulk of their configuration in server images, favoring baking images (see "Baking Server Images" on page 187) over frying instances. So a pipeline, or set of pipelines, to automatically build and update server images is a foundation for immutable servers.

You can fry configuration onto immutable server instances (as described in "Frying a Server Instance" on page 186), as long as you don't make changes to the instance after you create it. However, a stricter form of immutable servers avoids adding any differences to server instances. With this approach, you create and test a server image, and then promote it from one environment to the next. Because little or nothing changes with each server instance, you lower the risk of issues creeping in from one environment to the next.

Related patterns

People tend to bake servers (see "Baking Server Images" on page 187) to support immutable servers. Continuous synchronization (see "Pattern: Continuous Configuration Synchronization" on page 194) is the opposite approach, routinely applying changes to running server instances. Immutable servers are a subset of immutable infrastructure (see "Immutable Infrastructure" on page 351).

Patching Servers

Many teams are used to patching servers as a special, separate process. However, if you have an automated pipeline that delivers changes to servers, and continuously synchronize server code to existing servers, you can use this same process to keep your servers patched.

Teams I've worked with pull, test, and deliver the latest security patches for their operating systems and other core packages on a weekly basis, and sometimes daily.

This rapid, frequent patching process impressed the CIO at one of my clients when the business press trumpeted a high-profile security vulnerability in a core OS package. The CIO demanded that we drop everything and create a plan to address the vulnerability, with estimates for how long it would take and the cost impact of diverting resources to the effort. They were pleasantly surprised when we told them that the patch fixing the issue had already been rolled out as part of that morning's routine update.

How to Apply Server Configuration Code

Having described patterns to manage when to apply configuration code to servers, this section now discusses patterns for deciding how to apply the code to a server instance.

These patterns are relevant to changing servers, particularly as a part of a continuous synchronization process (see "Pattern: Continuous Configuration Synchronization" on page 194). But they're also used when building new server instances, to fry configuration onto an instance (see "Frying a Server Instance" on page 186). And you also need to choose a pattern to apply configuration when building server images (Chapter 13).

Given a server instance, whether it's a new one you are building, a temporary instance you use to build an image, or an existing instance, there are two patterns for running a server configuration tool to apply code. One is *push*, and the other is *pull*.

Pattern: Push Server Configuration

With the *push server configuration* pattern, a process running outside the new server instance connects to the server and executes, downloading and applying code.

Motivation

Teams use push to avoid the need to preinstall server configuration tooling onto server images.

Applicability

The push pattern is useful when you need a higher degree of control over the timing of configuration updates to existing servers. For example, if you have events, such as software deployments, with a sequence of activities across multiple servers, you can implement this with a central process that orchestrates the process.

Consequences

The push configuration pattern requires the ability to connect to server instances and execute the configuration process over the network. This requirement can create a security vulnerability since it opens a vector that attackers might use to connect and make unauthorized changes to your servers.

It can be awkward to run push configuration for server instances that are created automatically by the platform (see "Configuring the Platform to Automatically Create Servers" on page 182); for example, for autoscaling or automated recovery. However, it can be done.

Implementation

One way to push server configuration is for someone to run the server configuration tool from their local computer. As discussed in "Applying Code from a Centralized Service" on page 345, however, it's better to run tools from a central server or service, to ensure consistency and control over the process.

Some server configuration tools include a server application that manages connections to machine instances, such as Ansible Tower. Some companies offer SaaS services to remotely configure server instances for you, although many organizations prefer not to give third parties this level of control over their infrastructure.

In other cases, you implement a central service yourself to run server configuration tools. I've most often seen teams build this using CI or CD server products. They implement CI jobs or pipeline stages that run the configuration tool against a particular set of servers. The job triggers based on events, such as changes to server configuration code, or creation of new environments.

The server configuration tool needs to be able to connect to server instances. Although some tools use a custom network protocol for this, most use SSH. Each server instance must accept SSH connections from the server tool, allowing the tool to run with enough privileges to apply its configuration changes.

It's essential to have strong authentication and secrets management for these connections. Otherwise, your server configuration system is a huge security hole for your estate.

When creating and configuring a new server instance, you can dynamically generate a new authentication secret, such as an SSH key. Most infrastructure platforms provide a way to set a key when creating a new instance. Your server configuration tool can then use this secret and potentially disable and discard the key once it's no longer needed.

If you need to apply configuration changes to existing server instances, as with continuous synchronization, then you need a longer-term method to authenticate connections from the configuration tool. The simplest method is to install a single key on all of your server instances. But this single key is a vulnerability. If it is exposed, then an attacker may be able to access every server in your estate.

An alternative is to set a unique key for each server instance. It's essential to manage access to these keys in a way that allows the server configuration tool to access them while reducing the risk of an attacker gaining the same access—for example, by compromising the server that runs the tool. The advice in "Handling Secrets as Parameters" on page 102 is relevant here.

One approach many organizations use is to have multiple servers or services that manage server configuration. The estate is divided into different security realms, and each server configuration service instance only has access to one of these. This division can reduce the scope of a compromise.

Related patterns

The alternative to the push pattern is the pull server configuration pattern.

Pattern: Pull Server Configuration

The *pull server configuration* pattern involves a process running on the server instance itself to download and apply configuration code. This process triggers when a new server instance is created. For existing instances under the continuous synchronization pattern, the process typically runs on a schedule, periodically waking up and applying the current configuration.

Motivation

Pull-based server configuration avoids the need for server instances to accept incoming connections from a central server, so it helps to reduce the attack surface. The pattern simplifies configuring instances created automatically by the infrastructure platform, such as autoscaling and automated recovery (see "Configuring the Platform to Automatically Create Servers" on page 182).

Applicability

You can implement pull-based server configuration when you can build or use server images that have your server configuration tool preinstalled.

Implementation

Pull configuration works by using a server image that has the server configuration tool preinstalled. If you are pulling configuration for new server instances, configure the image to run the tool when it first boots.

Cloud-init is a widely used tool for automatically running this kind of process. You can pass parameters to the new server instance using your infrastructure platform's API, even including commands to execute, and parameters to pass to the server configuration tool (see Example 12-1).

Example 12-1. Example stack code that runs a setup script

```
server:
  source_image: stock-linux-1.23
  memory: 2GB
  vnet: ${APPSERVER_VNET}
```

```
instance_data:
  - server_tool: servermaker
  - parameter: server_role=appserver
  - parameter: code_repo=servermaker.shopspinner.xyz
```

Configure the script to download configuration code from a central repository and apply it on startup. If you use continuous synchronization to update running servers, the setup process should configure this, whether it's running a background process for the server configuration tool, or a cron job to execute the tool on a schedule.

Even if you don't build your own server images, most of the images provided by public cloud vendors have cloud-init and popular server configuration tools preinstalled.

A few other tools, notably Saltstack, use a messaging and event-based approach to trigger server configuration. Each server instance runs an agent that connects to a shared service bus from which it receives commands to apply configuration code.

Related patterns

The alternative to the pull pattern is the push server configuration pattern (see "Pattern: Push Server Configuration" on page 198).

Decentralized Configuration

Most server configuration tools provide a central service, which you run on a machine or cluster to centrally control the distribution of configuration code and parameters, and to manage other activities. Some teams prefer to run without a central service.

The leading reason teams decentralize configuration is to simplify their infrastructure management. A configuration server is another set of pieces to manage and can be a single point of failure. If the configuration server is down, you can't build new machines, which makes it a dependency for disaster recovery. Configuration servers may also be a performance bottleneck, needing to scale to handle connections from (and possibly to) hundreds or thousands of server instances.

To implement decentralized configuration, install and run the server configuration tool in offline mode, such as using `chef-solo` rather than `chef-client`. You may run the tool from a script, which checks a central file repository to download the latest versions of server configuration code. The code is stored locally on server instances, so the tool can still run if the file repository is not available.

A central file repository might be a single point of failure or a performance bottleneck, just like a configuration server. But in practice, there are many simple, highly reliable, and performant options for hosting static files like server configurations.

These include web servers, networked file servers, and object storage services like AWS S3.

Another way teams implement the decentralized pattern is by bundling server configuration code into a system package like an *.rpm* or *.deb* file and hosting it in their private package repository. A regular process runs `yum update` or `apt-get update`, which installs or updates the package, copying the server configuration code to a local directory.

Other Server Life Cycle Events

Creating, changing, and destroying server instances makes up the basic life cycle for a server. But there are other interesting phases in the extended life cycle, including stopping and restarting a server (Figure 12-1), replacing a server, and recovering a failed server.

Stopping and Restarting a Server Instance

When most of our servers were physical devices plugged into an electrical supply, we would commonly shut them down to upgrade hardware, or reboot them when upgrading certain operating system components.

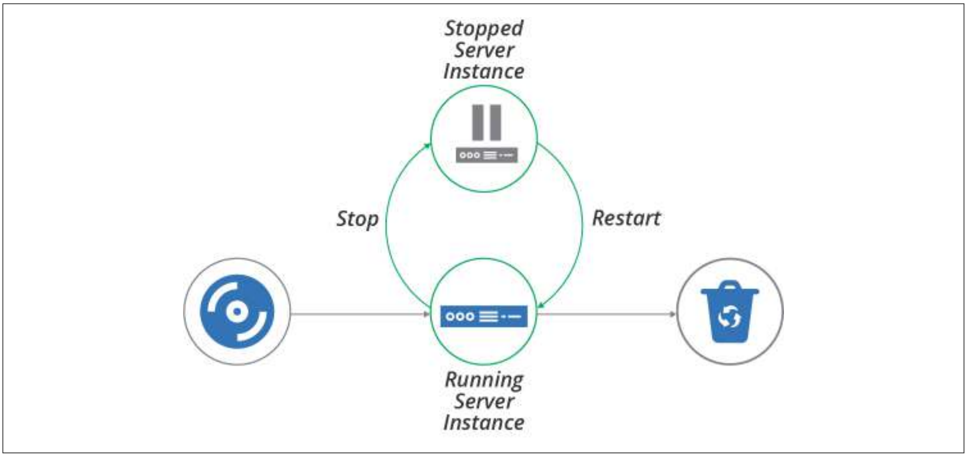


Figure 12-1. Server life cycle—stopping and restarting

People still stop and reboot virtual servers, sometimes for the same reasons—to reconfigure virtual hardware or upgrade an OS kernel. We sometimes shut servers down to save hosting costs. For example, some teams shut down development and test servers during evenings or weekends if nobody uses them.

If servers are easy to rebuild, however, many teams simply destroy servers when they aren't in use, and create new servers when they're needed again. They do this partly because it's easy and may save a bit more money than simply shutting them down.

But destroying and rebuilding servers rather than stopping and keeping them around also plays to the philosophy of treating servers like "cattle" rather than "pets" (as mentioned in "Cattle, Not Pets" on page 16). Teams may stop and restart rather than rebuilding because they aren't able to rebuild their servers with confidence. Often the challenge is retaining and restoring application data. See "Data Continuity in a Changing System" on page 382 for ways to handle this challenge.

So having a policy not to stop and restart servers forces your team to implement reliable processes and tools to rebuild servers and to keep them working.

Having servers stopped can also complicate maintenance activities. Configuration updates and system patches won't be applied to stopped servers. Depending on how you manage those types of updates, servers may receive them when starting again, or they may miss out.

Replacing a Server Instance

One of the many benefits of moving from physical to virtual servers is how easy it is to build and replace server instances. Many of the patterns and practices described in this book, including immutable servers (see "Pattern: Immutable Server" on page 195) and frequently updating server images, rely on the ability to replace a running server by building a new one (Figure 12-2).

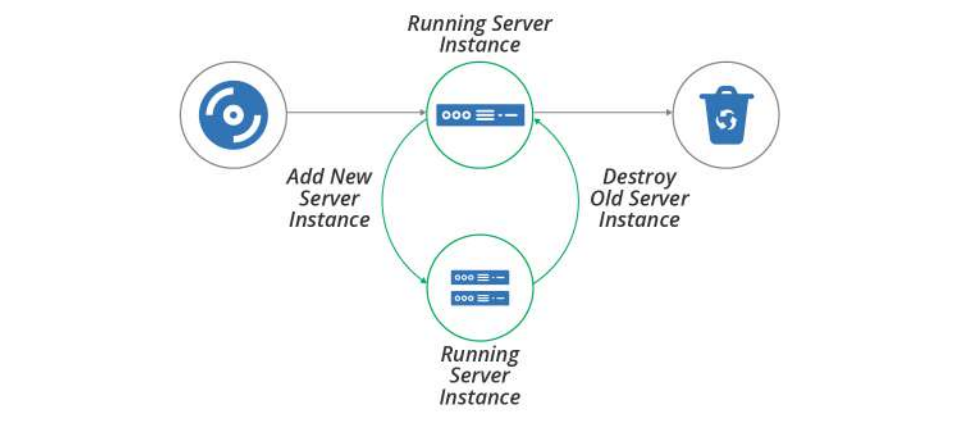


Figure 12-2. Server life cycle—replacing a server instance

The essential process for replacing a server instance is to create a new instance, validate its readiness, reconfigure other infrastructure and systems to use the new instance, test that it's working correctly, and then destroy the old instance.

Depending on the applications and systems that use the server instance, it may be possible to carry out the replacement without downtime, or at least with minimal downtime. You can find advice on this in "Changing Live Infrastructure" on page 368.

Some infrastructure platforms have functionality to automate the server replacement process. For example, you might apply a configuration change to the definition of servers in an autoscaling server cluster, specifying that it should use a new server image that includes security patches. The platform automatically adds new instances, checks their health, and removes old ones.

In other cases, you may need to do the work to replace servers yourself. One way to do this within a pipeline-based change delivery system is to expand and contract. You first push a change that adds the new server, then push a change that removes the old server afterward. See "Expand and Contract" on page 372 for more details on this approach.

Recovering a Failed Server

Cloud infrastructure is not necessarily reliable. Some providers, including AWS, explicitly warn that they may terminate server instances without warning—for example, when they decide to replace the underlying hardware.[5] Even providers with stronger availability guarantees have hardware failures that affect hosted systems.

The process for recovering a failed server is similar to the process for replacing a server. One difference is the order of activities—you create the new server after destroying the old one rather than before (see Figure 12-3). The other difference is that you typically replace a server on purpose, while failure is less intentional.[6]

As with replacing a server instance, recovery may need a manual action, or you may be able to configure your platform and other services to detect and recover failures automatically.

[5] Amazon provides documentation for its instance retirement policy.

[6] An exception to this is chaos engineering, which is the practice of deliberately creating failure to prove recoverability. See "Chaos Engineering" on page 379.

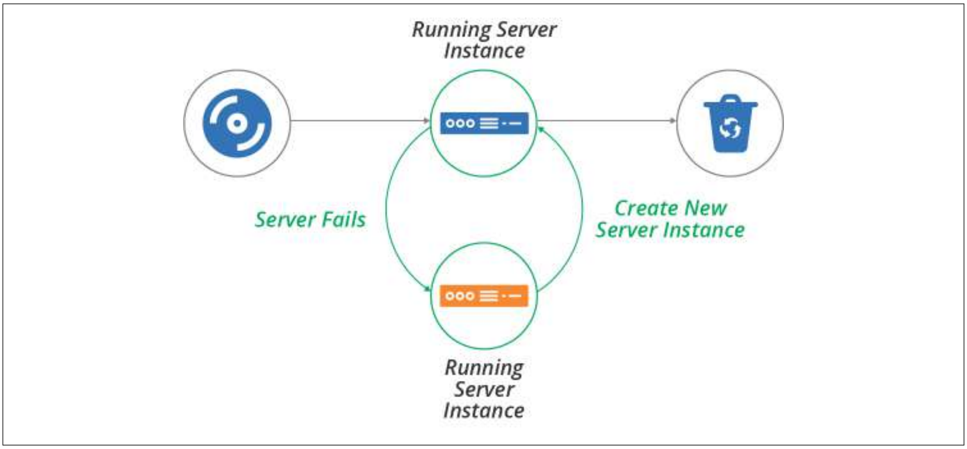


Figure 12-3. Server life cycle—recover a failed server instance

Conclusion

We've considered some patterns for when to apply configuration changes to servers—when making a change, by applying code continuously to running servers, or by creating new instances. We've also looked at patterns for how to apply changes—push and pull. Finally, we touched on some of the other server life cycle events, including stopping, replacing, and recovering servers.

This chapter, combined with the previous chapter on creating servers, covers the core events in a server's life. However, many of the approaches to creating and changing servers that we've discussed work by customizing server images, which you then use to create or update multiple server instances. So the next chapter explores approaches to defining, building, and updating server images.

CHAPTER 13

Server Images as Code

Chapter 11 touched on the idea of creating multiple server instances from a single source ("Prebuilding Servers" on page 183), and recommended using cleanly built server images ("Creating a Clean Server Image" on page 185). If you wondered what's the best way to follow that advice, this chapter aims to help.

As with most things in your system, you should use code to build and update server images in a repeatable way. Whenever you make a change to an image, such as installing the latest operating system patches, you can be sure you are building it consistently.

By using an automated, code-driven process to build, test, deliver, and update server images, you can more easily keep your servers up-to-date, and ensure their compliance and quality.

The basic server image life cycle involves building a custom image from an origin source (see Figure 13-1).

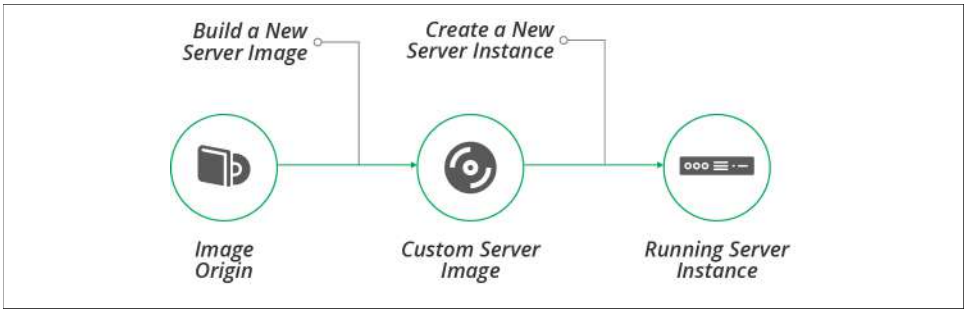


Figure 13-1. Server image life cycle

Building a Server Image

Most infrastructure management platforms have a format for server images to use for creating server instances. Amazon has AMIs, Azure has managed images, and VMware has VM templates. Hosted platforms provide prepackaged stock images with common operating systems and distributions so that you can create servers without needing to build images yourself.

Why Build a Server Image?

Most teams eventually build custom images rather than using the stock images from the platform vendor. Common reasons for doing this include:

Compliance with organizational governance
: Many organizations, especially those in regulated industries, need to ensure that they build servers following stricter guidelines.

Hardening for security
: Stock server images tend to have many different packages and utilities installed, to make them useful for a wide variety of cases. You can build custom images stripped to the minimum elements. Hardening can include disabling or removing unused user accounts and system services, disabling all network ports not strictly needed for operation, and locking down filesystem and folder permissions.[1]

Optimizing for performance
: Many of the steps you take to harden images for security, such as stopping or removing unneeded system services, also reduce the CPU and memory resources a server uses. Minimizing the size of the server image also makes it faster to create server instances, which can be helpful with scaling and recovery scenarios.

Installing common packages
: You can install a standard set of services, agents, and tools on your server images, to make sure they're available on all of your instances. Examples include monitoring agents, system user accounts, and team-specific utilities and maintenance scripts.

[1] For more information on security hardening, see "Proactively Hardening Systems Against Intrusion: Configuration Hardening" on the Tripwire website, "25 Hardening Security Tips for Linux Servers", and "20 Linux Server Hardening Security Tips".

Building images for server roles

Many of the examples in this chapter go further than building a standard, general-purpose image. You can build server images tailored for a particular purpose, such as a container cluster node, application server, or CI server agent.

How to Build a Server Image

Chapter 11 described a few different ways of creating a new server instance, including hot-cloning an existing instance or using a snapshot from an existing instance (see "Prebuilding Servers" on page 183). But instances built from a well-managed server image are cleaner and more consistent.

There are two approaches to building a server image. The most common, and most straightforward, is *online image building*, where you create and configure a new server instance, and then turn that into an image. The second approach, *offline image building*, is to mount a disk volume and copy the necessary files to it, and then turn that into a bootable image.

The main trade-off between building images online and offline is speed and difficulty. It's easier to build a server image online, but it can be slow to boot and configure a new instance, on the order of minutes or tens of minutes. Mounting and preparing a disk offline is usually much faster, but needs a bit more work.

Before discussing how to implement each of these options, it's worth noting some tools for building server images.

Tools for Building Server Images

Tools and services for building server images generally work by orchestrating the process of creating a server instance or disk volume, running a server configuration tool or scripts to customize it, and then using the infrastructure platform API to convert the server instance into an image. You should be able to define this process as code, for all of the usual reasons. I'll share examples of image building code shortly, to illustrate the online and offline approaches.

Netflix pioneered the heavy use of server images, and open sourced the Aminator tool it created for doing it. Aminator is specific to Netflix's ways of working, building CentOS and Red Hat images for AWS EC2.[2]

The most popular tool for building images, as of this writing, is HashiCorp's Packer. Packer supports a variety of operating systems and infrastructure platforms.

[2] Netflix described its approach for building and using AMIs in a blog post.

Some of the infrastructure platforms offer services for building their server images, including AWS Image Builder and Azure Image Builder.

The examples of server image building code in this chapter use a simple, fictional image builder language.

Online Image Building Process

The online method for building server images involves booting a new, clean server instance, configuring it, and then converting it into the infrastructure platform's server image format, as shown in Figure 13-2.

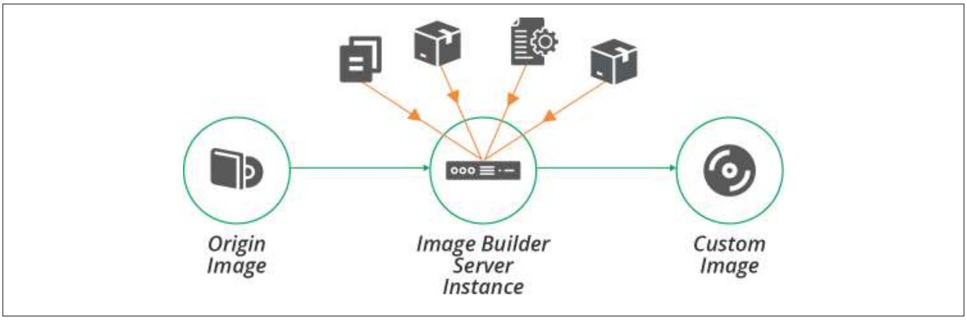


Figure 13-2. Online process for building a server image

The process starts by booting a new server instance from an origin image, as shown in Example 13-1. See "Origin Content for a Server Image" on page 214 for more about the options for origin images. If you use a tool like Packer to build your image, it uses the platform API to create the server instance.

Example 13-1. Example image builder code for booting from an origin image

```
image:
  name: shopspinner-linux-image
  platform: fictional-cloud-service
  origin: fci-12345678
  region: europe
  instance_size: small
```

This example code builds an FCS image named shopspinner-linux-image, booting it from the existing FCI whose ID is fci-12345678.[3] This is a stock Linux distribution provided by the cloud vendor. This code doesn't specify any configuration actions, so the resulting image is a fairly straight copy of the origin image.

The code also defines a few characteristics of the server instance to boot for building the image, including the region and instance type.

Infrastructure for the builder instance

You probably need to provide some infrastructure resources for your server image building instance—for example, an address block. Make sure to provide an appropriately secure and isolated context for this instance.

It might be convenient to run the builder instance using existing infrastructure, but doing this might expose your image building process to compromise. Ideally, create disposable infrastructure for building your image, and destroy it afterward.

Example 13-2 adds a subnet ID and SSH key to the image builder instance.

Example 13-2. Dynamically assigned infrastructure resources

```
image:
  name: shopspinner-linux-image
  origin: fci-12345678
  region: europe
  size: small
  subnet: ${IMAGE_BUILDER_SUBNET_ID}
  ssh_key: ${IMAGE_BUILDER_SSH_KEY}
```

These new values use parameters, which you can automatically generate and pass to the image builder tool. Your image building process might start by creating an instance of an infrastructure stack that defines these things, then destroys them after running the image builder tool. This is an example of an infrastructure tool orchestration script (see "Using Scripts to Wrap Infrastructure Tools" on page 335 for more on this topic).

You should minimize the resources and access that you provision for the image builder server instance. Only allow inbound access for the server provisioning tool, and only do this if you are pushing the configuration (see "Pattern: Push Server Configuration" on page 198). Only allow outbound access necessary to download packages and configuration.

[3] FCS is the Fictional Cloud Service, similar to AWS, Azure, and others mentioned in "Infrastructure Platforms" on page 25. An FSI is a Fictional Server Image, similar to an AWS AMI or Azure Managed Image.

Configuring the builder instance

Once the server instance is running, your process can apply server configuration to it. Most server image building tools support running common server configuration tools (see "Server Configuration Code" on page 172), in a similar way to stack management tools (see "Applying Server Configuration When Creating a Server" on page 189). Example 13-3 extends Example 13-2 to run the Servermaker tool to configure the instance as an application server.

Example 13-3. Using the Servermaker tool to configure the server instance

```
image:
  name: shopspinner-linux-image
  origin: fci-12345678
  region: europe
  instance_size: small
  subnet: ${IMAGE_BUILDER_SUBNET_ID}
  configure:
    tool: servermaker
    code_repo: servermaker.shopspinner.xyz
    server_role: appserver
```

The code applies a server role, as in "Server Roles" on page 175, which may install an application server and related elements.

Using Simple Scripts to Build a Server Image

When you're already using a server configuration tool to configure new server instances, it makes sense to use the same tool, and often the same server configuration code, when building server images. This fits the baking and frying process described in "Combining Baking and Frying" on page 188.

But for teams who bake more complete server images (see "Baking Server Images" on page 187), a full-blown server configuration tool may be overkill. These tools and their languages add complexity so that you can apply them repeatedly to servers with unknown, variable starting conditions.

But each time you build a server image, you usually build it from a known, consistent origin source. So a simpler scripting language may be more appropriate—Bash, batch scripts, or PowerShell, for example.

If you do write scripts to configure server images, you should write small, simple scripts, each focused on a single task, rather than having large complex scripts with conditional loops. I like to prefix scripts with a number so that it's clear what order they run in, as shown in Example 13-4.

Example 13-4. Example image builder using shell scripts

```
image:
  name: shopspinner-linux-image
  origin: fci-12345678
  configure:
    commands:
      - 10-install-monitoring-agent.sh
      - 20-install-jdk.sh
      - 30-install-tomcat.sh
```

Offline Image Building Process

The process of booting a server instance, applying the configuration, and then shutting the instance down to create the server image can be slow. An alternative is to use a source image that is mountable as a disk volume—for example, an EBS-backed AMI in AWS. You can mount a copy of this image as a disk volume, and then configure the files on that disk image, as shown in Figure 13-3. Once you have finished configuring the disk image, you can unmount it and convert it to the platform's server image format.

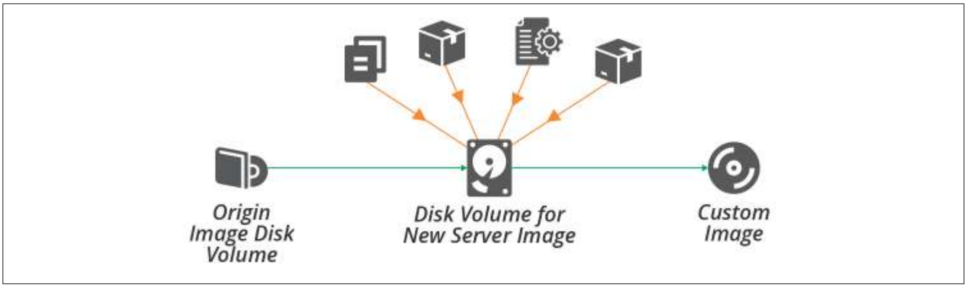


Figure 13-3. Offline process for building a server image

To build an image on a disk volume, you need to mount the volume on the compute instance where your image building tool runs. For example, if you build images using a pipeline as described later in this chapter, this could be a CD tool agent.

Another requirement is a tool or scripts that can properly configure the disk volume. This can be tricky. Server configuration tools apply configuration code to the server instance on which they are running. Even if you write simpler scripts, like a shell script, those scripts often run a tool like a package installer, which defaults to installing files in the running server's filesystem:

```
yum install java
```

Many tools have command-line or configuration options that you can set to install files to a different path. You can use this to install packages to a different path (for example, `yum install java --prefix /mnt/image_builder/`):

```
yum install --prefix /mnt/image_builder/ java
```

However, this is tricky and easy to get wrong. And not all tools support an option like this one.

Another approach is to use the `chroot` command, which runs a command with a different filesystem root directory:

```
chroot /mnt/image_builder/ yum install java
```

The advantage of using the `chroot` command is that it works for any tool or command. Popular image building tools like Packer support `chroot` out of the box.[4]

Origin Content for a Server Image

Several elements come together for building server images:

Image builder code
: You write code to define the image, which your builder tool uses. A Packer template file is one example. This code defines the other elements to assemble into the image.

Origin source image
: In many cases, you build an image starting with another image. You may need a source image in your platform's format, like an AMI. Or it may be an operating system installation image, whether physical like a DVD or data like an ISO image. In some cases, you build the image from scratch, such as when you run an OS installer. Each of these is described in more detail later.

Server configuration code
: You may use a server configuration tool or scripts to customize the image, as explained in the previous section.

Other sever elements
: You may use any or all of the elements, and sources of those elements, described in "What's on a Server" on page 170 to build your server image. It's common to use packages from public or internal repositories when building images.

[4] See Packer's chroot builders for AWS AMIs and Azure server images.

This set of elements comes into play when you use a pipeline to automate the image building process, as described later in this chapter. Types of origin source images are worth some more specific attention.

Building from a Stock Server Image

Infrastructure platform vendors, operating system vendors, and open source projects often build server images and make them available to users and customers. As mentioned earlier, many teams use these stock images directly rather than building their own. But when you do build your own, these images are usually a good starting point.

One issue with stock images is that many of them are over-provisioned. Makers often install a wide variety of tools and packages, so that they are immediately useful to the broadest number of users. But you should minimize what you install on your custom server images, both for security and for performance.

You can either strip unneeded content from the origin image as part of your server image configuration process or find smaller origin images. Some vendors and organizations build JEOS (Just Enough Operating System) images. Adding the things you need to a minimal base image is a more reliable approach for keeping your images minimal.

Building a Server Image from Scratch

Your infrastructure platform may not provide stock images, especially if you use an internal cloud or virtualization platform. Or, your team may simply prefer not to use a stock image. The alternative is to build custom server images from scratch.

An OS installation image gives you a clean, consistent starting point for building a server template. The template building process starts by booting a server instance from the OS image and running a scripted installation process.[5]

Provenance of a Server Image and its Content

Another concern with base images, as with any content from third parties, is their provenance and security. You should be sure you understand who provides the image, and the process they use to ensure that they build the images safely. Some things to consider:

[5] Some scripted OS installers include Red Hat Kickstart, Solaris JumpStart, Debian Preseed, and the Windows installation answer file.

- Do images and packages include software with known vulnerabilities?
- What steps does the vendor take to scan their code for potential problems, such as static code analysis?
- How do you know whether any included software or tools collect data in ways that contradict your organizational policies or local laws?
- What processes does the vendor use to detect and prevent illicit tampering, and do you need to implement anything on your end, such as checking package signatures?

You shouldn't completely trust the content that comes from a vendor or third party, so you should implement your own checks. "Test Stage for a Server Image" on page 223 suggests how to build checking into a pipeline for server images.

Changing a Server Image

Server images become stale over time as updated packages and configurations are released. Although you can apply patches and updates every time you create a new server, this process takes longer as time goes by, reducing the benefit of using a server image. Regularly refreshing your images keeps everything running smoothly.

In addition to keeping images up-to-date with the latest patches, you often need to improve the base configurations, add and remove standard packages, and make other routine changes to your server images. The more you bake into your custom server images rather than frying into server instances as you create them, the more often you need to update your images with changes.

The next section explains how to use a pipeline to make, test, and deliver changes to your server images. There are a few prerequisites to cover before getting into pipelines, however. One is how to make updates—whether to use the existing server image as the base for your change or rebuild the image from first principles. Another prerequisite is versioning your server images.

Reheating or Baking a Fresh Image

When you want to build a new version of a server image—for example, to update it with the latest operating system patches—you can use the same tools and processes that you used to build the first version of the image.

You could use the previous version of the image as the origin source image—*reheating* the image. This ensures that the new version is consistent with the previous version, because the only changes are the ones you explicitly make when you apply server configuration to the new image.

The alternative is to *bake a fresh image*, building the new version of the image from the original sources. So, if you built the first image version from scratch, you build each new version from scratch as well.

While reheating images may limit the changes from one version to the next, baking a fresh image for the new version should give the same results, given that you are using the same sources. Fresh builds are arguably cleaner and more reliably reproducible because they don't include anything that may have been left over by previous builds.

For example, your server configuration may install a package that you later decide to remove. That package is still found on newer server images if you reheat from the older images. You need to add code to explicitly remove the package, which leaves you with superfluous code to maintain and clean out later. If instead, you bake fresh images each time, the package will simply not exist on newer server images, so there's no need to write any code to remove it.

Versioning a Server Image

It's essential that everyone who uses server images, and servers created from them, can track the versions. They should be able to understand what version was used to create any given server instance, which version is the latest for a given image, and what source content and configuration code were used to create the image.

It's also helpful to be able to list which server image versions were used to create instances currently in use—for example, so you can identify instances that need to be updated when you discover a security vulnerability.

Many teams use different server images. For example, you may build separate images for application servers, container host nodes, and a general-purpose Linux OS image. In this case, you would manage each of these images as a separate component, each with its separate version history, as illustrated in Figure 13-4.

In this example, the team updated the application server image to upgrade the version of Tomcat installed on it, bumping that image to version 2, while the others remained at version 1. The team then updated all of the images with Linux OS updates. This bumped the application server image to version 3, and the other images to version 2. Finally, the team upgraded the container software version on the host node image, bumping it to version 3.

Most infrastructure platforms don't directly support version numbering for server images. In many cases, you can embed version numbers in the name of the server image. For the examples in Figure 13-4, you might have images named `appserver-3`, `basic-linux-2`, and `container-node-3`.

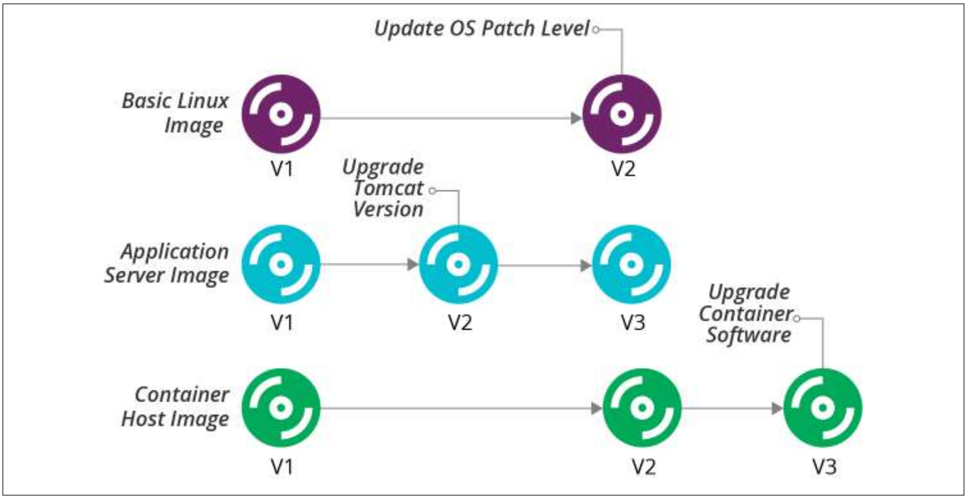


Figure 13-4. Example version history for multiple server images

Another option is to put version numbers and image names in tags if your platform supports it. An image could have a tag named `Name=appserver` and another named `Version=3`.

Use whatever mechanism makes it easiest to search, discover, and report on image names and versions. Most of the teams I've worked with use both of those methods, since tags are easy to search, and names are easy for humans to see.

You can use different version numbering schemes. Semantic versioning is a popular approach, using different fields of a three-part version number (such as `1.0.4`) to indicate how significant the changes are from one version to the next. I like to add a date and time stamp into the version (for example, `1.0.4-20200229_0420`, to make it easy to see when a particular version was built.

In addition to putting the version number on the server image itself, you can tag or label server instances with the server image name and version used to create it. You can then map each instance back to the image used to create it. This ability is useful for rolling out new image versions.

Updating Server Instances When an Image Changes

When you build a new version of a server image, you may replace all of the server instances based on that image, or wait for them to be replaced naturally over time.

A policy of rebuilding existing servers to replace old image versions can be disruptive and time-consuming. But it also ensures the consistency of servers and continuously exercises your system's resilience (see also "Continuous Disaster Recovery" on page 379). A good pipeline makes it easier to manage this process, and zero-downtime changes (see "Zero Downtime Changes" on page 375) make it less disruptive. So this policy is favored by many teams with mature and pervasive automation.

It's easier to wait to replace servers with the new image version when they are rebuilt for other reasons. This might be the case if you deploy software updates or other changes by rebuilding server instances (see "Pattern: Immutable Server" on page 195).

The drawback of waiting to update servers with new image versions is that this may take a while, and leave your estate with servers built from a wide variety of image versions.

This situation creates inconsistency. For example, you may have application servers with different versions of OS package updates and configuration, leading to mysteriously inconsistent behavior. In some cases, the older image versions, and the servers built from them, may have security vulnerabilities or other issues.

There are a few strategies you can use to mitigate these issues. One is to track running instances against the image versions used to create them. This could be a dashboard or report, similar to Table 13-1.

Table 13-1. Example report of instances and their image versions

Image	Version	Instance count
basic-linux	1	4
basic-linux	2	8
appserver	1	2
appserver	2	11
appserver	3	8
container-node	1	2
container-node	2	15
container-node	3	5

If you have this information readily available, and you can drill into the list of specific servers, then you can identify server instances that urgently need rebuilding. For example, you learn about a security vulnerability that was fixed in the latest update to your Linux distribution. You included the patch in `basic-linux-2`, `appserver-3`, and `container-node-2`. The report shows you that you need to rebuild 19 of your server instances (4 basic Linux servers at version 1, 13 application servers at versions 1 and 2, and 2 container nodes at version 1).

You may also have age limits. For example, you could have a policy to replace any running server instances built from a server image older than three months. Your report or dashboard should then show the number of instances that are past this date.

Providing and Using a Server Image Across Teams

In many organizations, a central team builds server images and makes them available for other teams to use. This situation adds a few wrinkles to managing server image updates.

The team using the image needs to be sure that it is ready to use new versions of an image. For example, an internal application team may install bug tracking software onto the base Linux image provided by the compute team. The compute team may produce a new version of the base Linux image with updates that have an incompatibility with the bug tracking software.

Ideally, server images feed into each team's infrastructure pipeline. When the team owning the image releases a new version through its pipeline, each team's infrastructure pipeline pulls that version and updates its server instances. The internal team's pipeline should automatically test that the bug tracking software works with the new image version before rebuilding servers that its users depend on.

Some teams may take a more conservative approach, *pinning* the image version number they use. The internal application team could pin their infrastructure to use version 1 of the basic Linux image. When the compute team releases version 2 of the image, the internal application team continues to use version 1 until they are ready to test and roll out version 2.

Many teams use the pinning approach when their automation for testing and delivering changes to server infrastructure isn't as mature. These teams need to do more manual work to be sure of image changes.

Even with mature automation, some teams pin images (and other dependencies) whose changes are riskier. The dependencies may be more fragile because the software the team deploys onto images is more sensitive to changes in the operating system. Or the team providing the image—perhaps an external group—has a track record of releasing versions with problems, so trust is low.

Handling Major Changes to an Image

Some changes to server images are more significant, and so more likely to need manual effort to test and modify downstream dependencies. For example, a new version of the application server image may include a major version upgrade for the application server software.

Rather than treating this as a minor update to the server image, you may either use semantic versioning to indicate it is a more significant change, or even build a different server image entirely.

When you use semantic versioning, most changes increment the lowest digit of the version, for example, `1.2.5` to `1.2.6`. You indicate a major change by incrementing the second or first digit. For a minor application server software update that shouldn't create compatibility issues for applications, you might increment version `1.2.6` of the server image to `1.3.0`. For a major change that may break applications, you increment the highest digit, so `1.3.0` would be replaced by `2.0.0`.

In some cases, especially where you expect the older image version to be used by teams for some time, you may create a new image entirely. For example, if your basic Linux image uses Centos 9.x, but you want to start testing and rolling out Centos 10.x, rather than incrementing the version number of the image, you could create a new image, `basic-linux10-1.0.0`. This makes migration to the new OS version a more explicit undertaking than a routine image update.

Using a Pipeline to Test and Deliver a Server Image

Chapter 8 describes using pipelines to test and deliver infrastructure code changes ("Infrastructure Delivery Pipelines" on page 119). Pipelines are an excellent way to build server images. Using a pipeline makes it easy to build new versions of an image, ensuring they are built consistently. With a mature pipeline integrated with pipelines for other parts of a system, you can safely roll out operating system patches and updates across your estate weekly or even daily.

It's a good idea to build new server images often, such as weekly, rather than leaving it for long periods, such as every few months. This fits with the core practice of continuously testing and delivering changes. The longer you wait to build a new server image, the more changes it includes, which increases the work needed to test, find, and fix problems.

A basic server image pipeline would have three stages, as shown in Figure 13-5.

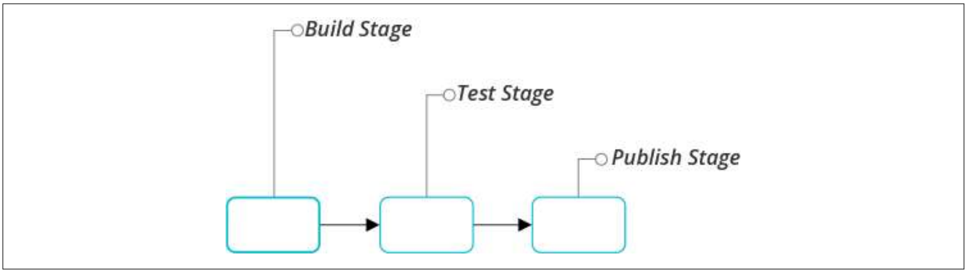


Figure 13-5. A simple server image pipeline

Each of these stages for building, testing, and publishing a server image merits a closer look.

Build Stage for a Server Image

The build stage of a server image pipeline automatically implements the online image building process ("Online Image Building Process" on page 210) or offline image building process ("Offline Image Building Process" on page 213). The stage produces a server image, which the following stages treat as a release candidate (see Figure 13-6).

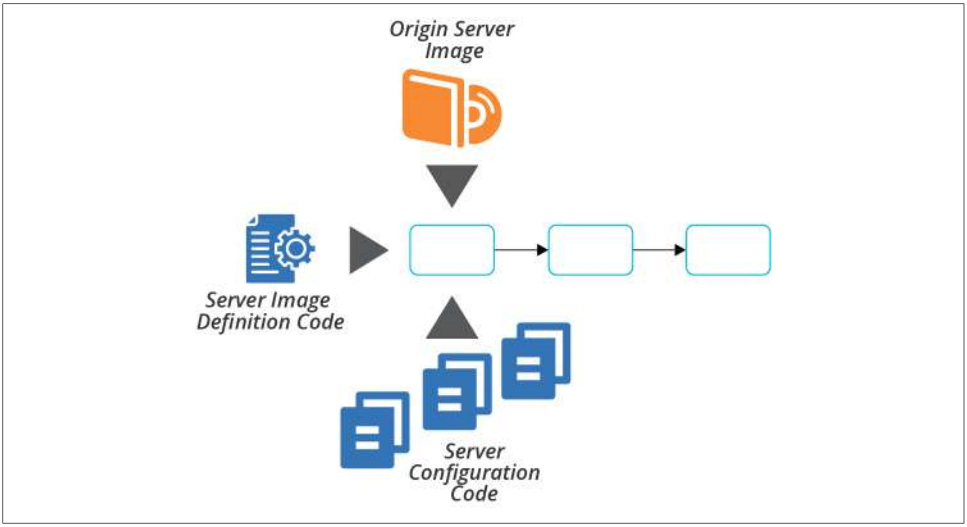


Figure 13-6. Server image build stage

You can configure the server image build stage to run automatically on changes to any of the inputs to the image listed in "Origin Content for a Server Image" on page 214. For example:

- Someone commits a change to the image builder code, such as Packer template
- The vendor publishes a new version of your origin source image, such as an AMI provided by the OS vendor
- You make changes to server configuration code used for the server image
- Package maintainers publish new versions of packages installed on the server image

In practice, many changes to source elements like OS packages are difficult to automatically detect and use to trigger a pipeline. You may also not want to build a new server image for every package update, especially if the image uses dozens or even hundreds of packages.

So you could automatically trigger new images only for major changes, like the source server image or image builder code. You can then implement scheduled builds—for example, weekly—to roll up all of the updates to smaller source elements.

As mentioned, the result of the build stage is a server image that you can use as a release candidate. If you want to test your image before making it available—recommended for any nontrivial use case—then you should not mark the image for use at this point. You can tag the image with a version number, according to the guidance given earlier. You might also tag it to indicate that it is an untested release candidate; for example, `Release_Status=Candidate`.

Test Stage for a Server Image

Given the emphasis on testing throughout this book, you probably understand the value of automatically testing new images as you build them. People usually test server images using the same testing tools that they use to test server code, as described in "Testing Server Code" on page 176.

If you build server images online (see "Online Image Building Process" on page 210), you could run your automated tests on the server instance after you configure it and before you shut it down and convert it to an image. But there are two concerns with this. First, your tests might pollute the image, leaving test files, log entries, or other residue as a side effect of your tests. They might even leave user accounts or other avenues for accessing servers created from the image.

The other concern with testing the server instance before you turn it into an image is that it may not reliably replicate servers created from the image. The process of converting the instance into an image might change important aspects of the server, such as user access permissions.

The most reliable method for testing a server image is to create and test a new instance from the final image. The drawback of doing this is that it takes longer to get the feedback from your tests.

So a typical server image test stage, as shown in Figure 13-7, takes the identifier of the image created in the build stage, uses it to create a temporary instance, and runs automated tests against that. If the tests pass, then the stage tags the image to indicate that it's ready for the next stage of the pipeline.

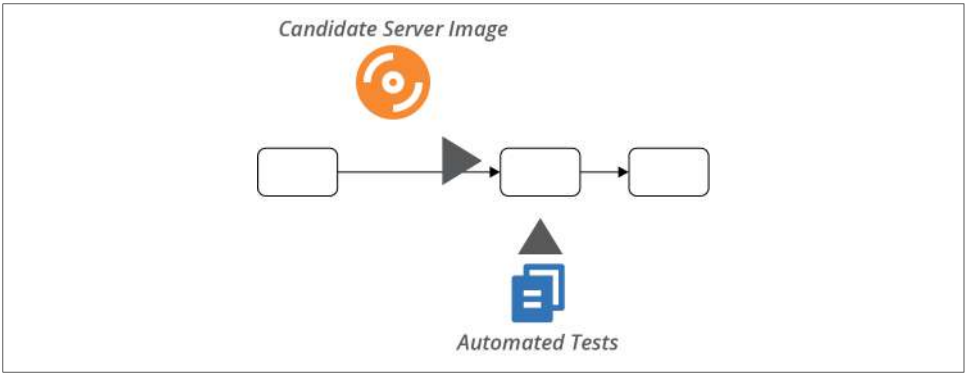


Figure 13-7. Server image test stage

Delivery Stages for a Server Image

Your team's image pipeline may have additional stages for other activities—for example, security testing. Ultimately, an image version that passes its test stages is tagged as ready for use.

In some systems, the pipeline that creates the image includes the stages to rebuild servers with the new image (see Figure 13-8). It could run a stage for each delivery environment in the progression to production, perhaps triggering application deployment and test stages.

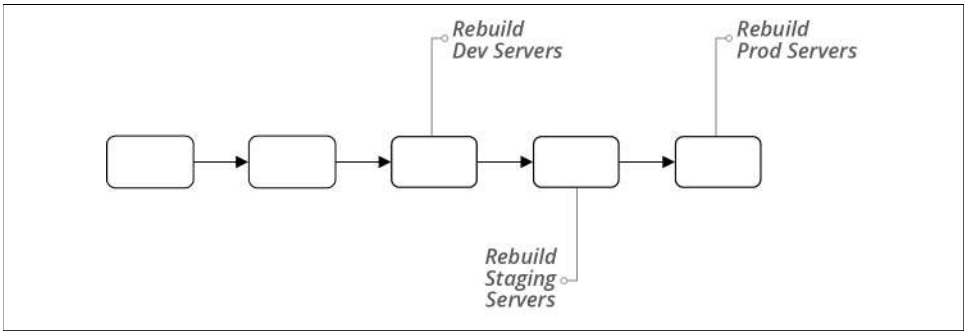


Figure 13-8. Pipeline that delivers images to environments

In systems where different teams are responsible for managing infrastructure and environments, the image pipeline may end once it has marked the image as ready for use. Other teams' pipelines take new image versions as inputs, as described in "Providing and Using a Server Image Across Teams" on page 220.

Using Multiple Server Images

Some teams only maintain a single server image. These teams create the different types of servers they need by applying server configuration roles (see "Server Roles" on page 175) when creating server instances. Other teams, however, find it useful or necessary to maintain multiple server images.

You need multiple images to support multiple infrastructure platforms, multiple server hardware architectures, or multiple server hardware architectures. And multiple images can be useful to optimize server creation time, following the strategy of baking over frying ("Baking Server Images" on page 187).

Let's examine each of these scenarios—supporting multiple platforms, and baking roles into server images—and then discuss strategies for maintaining multiple server images.

Server Images for Different Infrastructure Platforms

Your organization may use more than one infrastructure platform as part of a multicloud, poly-cloud, or hybrid cloud strategy. You need to build and maintain separate server images for each of these platforms.

Often, you can use the same server configuration code, perhaps with some variations that you manage using parameters, across platforms. You might start with different source images on each platform, although in some cases, you can use images from your OS vendor or a trusted third party that builds them consistently for each.

Each infrastructure platform should have a separate pipeline to build, test, and deliver new versions of the image. These pipelines may take server configuration code as input material. For example, if you update the server configuration for your application servers, you would push the change into each of the platform pipelines, building and testing a new server image for each.

Server Images for Different Operating Systems

If you support multiple operating systems or distributions, such as Windows, Red Hat Linux, and Ubuntu Linux, you'll need to maintain separate images for each OS. Also, you probably need a separate image for each major version of a given OS that your organization uses. You need a separate source image to build each of these OS images. You might be able to reuse server configuration code across some of these OS images.

But in many cases, writing server configuration code to handle different operating systems adds complexity, which requires more complex testing. The testing pipeline needs to use server (or container) instances for each variation. For some configurations, it can be simpler to write a separate configuration code module for each OS or distribution.

Server Images for Different Hardware Architectures

Some organizations run server instances on different CPU architectures, such as x86 and ARM. Most often, you can build nearly identical server images for each architecture, using the same code. Some applications exploit specific hardware features or are more sensitive to their differences. In these cases, your pipelines should test images more thoroughly across the different architectures to detect issues.

Server Images for Different Roles

Many teams use a general-purpose server image and apply role configuration (see "Server Roles" on page 175) when creating server instances. However, when configuring a server for a role involves installing too much software, such as an application server, it can be better to bake the role configuration into a dedicated image.

The example earlier in this chapter ("Versioning a Server Image" on page 217) used custom images for application servers and container host nodes, and a general-purpose Linux image. This example shows that you don't necessarily need to have a separate image for each server role you use.

To decide whether it's worth maintaining a separate image for a particular role, consider the time and cost of having pipelines, tests, and storage space, and weigh these against the drawbacks of configuring server instances when you create them—speed, efficiency, and dependencies—as explained in "Frying a Server Instance" on page 186.

Layering Server Images

Teams that have a large number of role-based server images may consider building them up in layers. For example, create a base OS image with the default packages and configurations that you want to install on all servers. Then use this image as the source image to build more specific role-based images for application servers, container nodes, and others, as shown in Figure 13-9.

In this example, the ShopSpinner team uses a base Linux image as the source image for building an application server image and a container host node image. The application server image has Java and Tomcat preinstalled. The team uses this image, in turn, as the source image for building images with specific applications preinstalled.

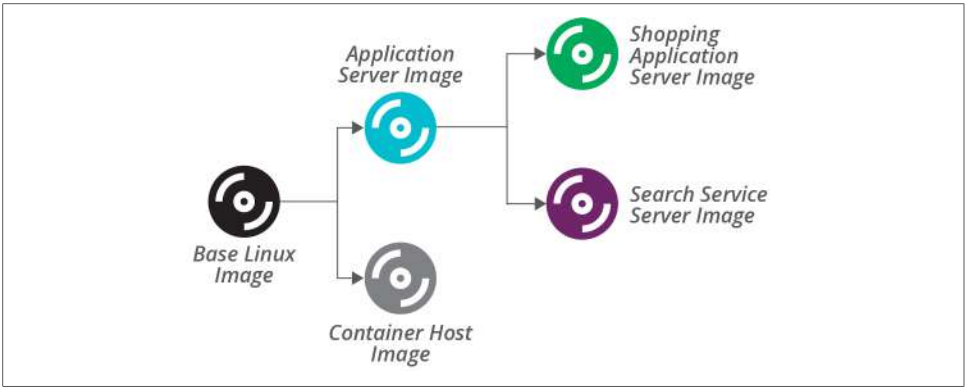

Application Server Image
Shopping Application Server Image
Base Linux Image
Search Service Server Image
Container Host Image

Figure 13-9. Layering server images

Governance and Server Images

Traditional approaches to governance often include a manual review and approval process each time a new server image is built. As "Governance in a Pipeline-based Workflow" on page 352 explains, defining servers as code and using pipelines to deliver changes creates opportunities for stronger approaches.

Rather than inspecting new server images, people responsible for governance can inspect the code that creates them. Better yet, they can collaborate with infrastructure and other teams involved to create automated checks that can run in pipelines to ensure compliance. Not only can your pipelines run these checks every time a new image is built, but they can also run them against new server instances to ensure that applying server configuration code hasn't "unhardened" it.

The ShopSpinner shopping application and search service frequently need to be scaled up and down to handle variable traffic loads at different times of the day. So it's useful to have the images baked and ready. The team builds server instances for other applications and services from the more general application server image because the team doesn't create instances of those as often.

A separate pipeline builds each server image, triggering when the input image changes. The feedback loops can be very long. When the team changes the base Linux image, its pipeline completes and triggers the application server and container host pipelines. The application server image pipeline then triggers the pipelines to build new versions of the server images based on it. If each pipeline takes 10–15 minutes to run, this means it takes up to 45 minutes for the changes to ripple out to all of the images.

An alternative is to use a shallower image building strategy.

Sharing Code Across Server Images

Even given an inheritance-based model where server roles inherit configuration from one another, you don't necessarily need to build images in the layered model we just described. You can instead layer the server configuration code into server role definitions, and apply all of the code directly to build each image from scratch, as shown in Figure 13-10.

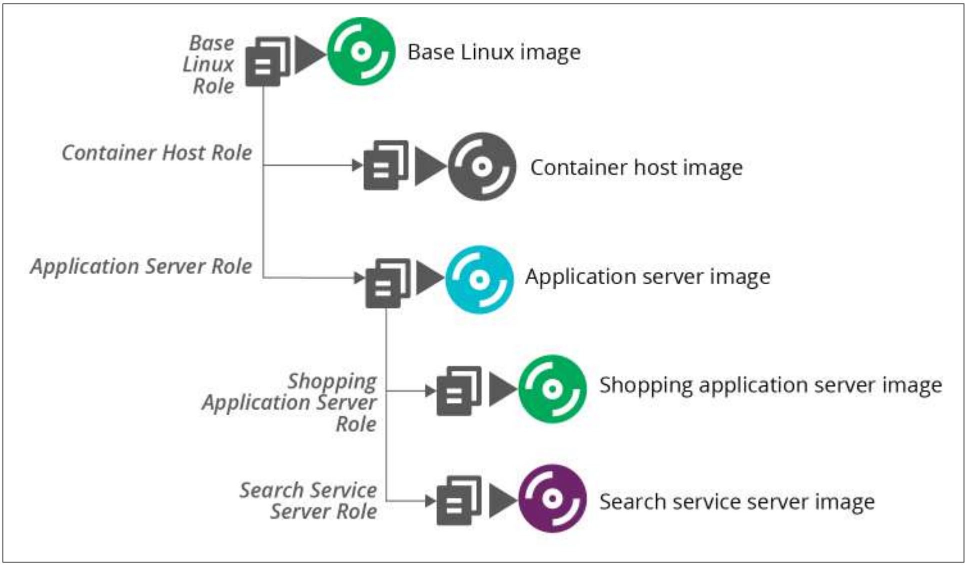

Base Linux Role
Base Linux image
Container Host Role
Container host image
Application Server Role
Application server image
Shopping Application Server Role
Shopping application server image
Search Service Server Role
Search service server image

Figure 13-10. Using scripts to layer roles onto server images

This strategy implements the same layered mode. But it does so by combining the relevant configuration code for each role and applying it all at once. So building the application server image doesn't use the base server image as its source image, but instead applies all of the code used for the base server image to the application server image.

Building images this way reduces the time it takes to build images at the bottom of the hierarchy. The pipeline that builds the image for the search service immediately runs when someone makes a change to the base Linux server configuration code.

Conclusion

You can build and maintain servers without creating custom images. But there are benefits from creating and maintaining an automated system for building server images. These benefits are mainly around optimization. You can create server instances more quickly, using less network bandwidth, and with fewer dependencies by pre-baking more of your configuration into server images.

CHAPTER 14

Building Clusters as Code

Chapter 3 described an application hosting cluster as a service that dynamically deploys and runs application instances across a pool of servers (see "Compute Resources" on page 28). Examples of application cluster systems include Kubernetes, AWS ECS, HashiCorp Nomad, Mesos, and Pivotal Diego. This model separates the concerns of orchestrating applications from the concerns of provisioning and configuring the servers they run on (see Figure 14-1).

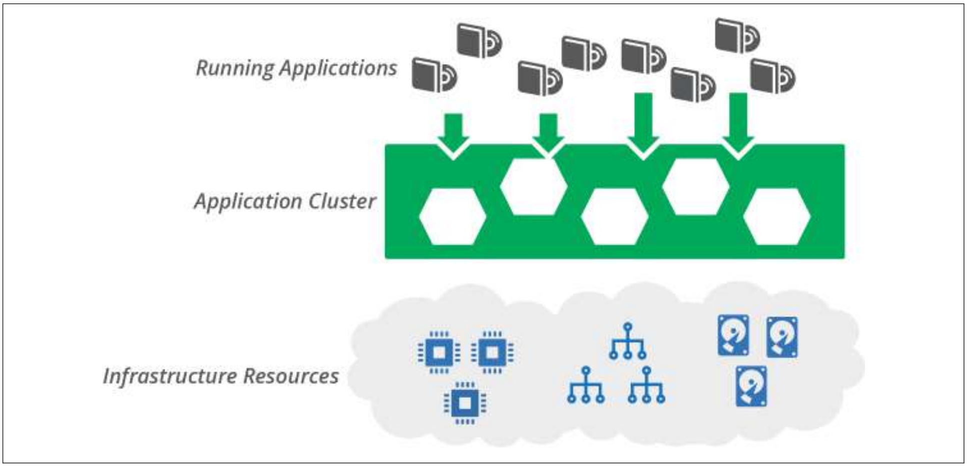


Figure 14-1. An application cluster creates a layer between infrastructure resources and the applications running on them

You should, of course, use code to define and manage your application cluster. As explained in Chapter 10, you can build an application cluster as one or more application stacks as code. In this chapter, I give some examples of what this can look like. I show some different stack topologies, including a single stack for a whole cluster, and breaking clusters across multiple stacks (see "Stack Topologies for Application Clusters" on page 232). I include diagrams for pipelines to deliver infrastructure code changes for these topologies.

I also discuss strategies for sharing clusters, or not sharing them, across environments and teams (see "Sharing Strategies for Application Clusters" on page 241). I close the chapter with a discussion of infrastructure for serverless applications (see "Infrastructure for FaaS Serverless" on page 246).

Containers as Code

One of the many strengths of containers is that they are defined as code. A container image is built once from a definition file, then used to create multiple instances. To use containers effectively, it's essential that you treat them as immutable and stateless.

Don't make changes to the contents of a container instance; instead, change the definition, create a new image, and then replace the instance. And don't store state within a container. Instead, store state and data using other infrastructure resources (see "Data Continuity in a Changing System" on page 382 in Chapter 21).

The immutable and stateless nature of containers makes them perfectly suited to the dynamic nature of cloud platforms, which is why they are so strongly associated with cloud native architectures (see "Cloud Native and Application-Driven Infrastructure" on page 156).

Application Cluster Solutions

There are two main approaches for implementing and maintaining an application cluster. One is to use a managed cluster as a service, usually provided as part of your infrastructure platform. The other is to deploy a packaged cluster solution onto lower-level infrastructure resources.

Cluster as a Service

Most infrastructure platforms provide a managed cluster service. You can define, provision, and change a cluster as code using your stack configuration tool. Using a cluster as a service is a matter of creating a stack that includes the cluster and supporting elements.

Many of these clusters are based on Kubernetes, including EKS, AKS, and GKE. Others are based on a proprietary container service, such as ECS.

Managed Kubernetes Clusters Are Not a Cloud Abstraction Layer

At first glance, vendor-managed Kubernetes clusters appear to be a great solution for developing and running applications transparently across different cloud platforms. Now you can build applications for Kubernetes and run them on whatever cloud you like!

In practice, although an application cluster might be useful as one part of your application runtime layer, much more work is required to create a full runtime platform. And achieving true abstraction from the underlying cloud platforms is not trivial.

Applications need access to other resources than the compute provided by a cluster, including storage and networking. These resources will be provided differently by different platforms, unless you build a solution to abstract those.

You also need to provide services such as monitoring, identity management, and secrets management. Again, you will either use a different service on each cloud platform, or build a service or abstraction layer that you can deploy and maintain on each cloud.

So the application cluster is actually a small piece of your overall application hosting platform. And even that piece tends to vary from one cloud to the next, with different versions, implementations, and tooling for the core Kubernetes system.

Packaged Cluster Distribution

Many teams prefer to install and manage application clusters themselves, rather than using managed clusters. Kubernetes is the most popular core cluster solution.

You can use an installer such as kops, Kubeadm, and kubespray to deploy Kubernetes onto infrastructure you've provisioned for it.

There are also packaged Kubernetes distributions that bundle other services and features. There are literally dozens of these, some of which are:[1]

- HPE Container Platform (Hewlett Packard)
- OpenShift (Red Hat)
- Pivotal Container Services (PKS) (VMware/Pivotal)
- Rancher RKE

Some of these products started out with their own container formats and application scheduling and orchestration services. Many decided to rebuild their core products

[1] You can find a list of certified Kubernetes distributions on the Kubernetes website.

around the overwhelmingly popular Docker and Kubernetes rather than to continue developing their own solutions.

But a handful of products have resisted assimilation into the Kubernetes borg. These include HashiCorp Nomad and Apache Mesos, both of which can orchestrate container instances as well as noncontainerized applications across different compute resources. The Cloud Foundry Application Runtime (CFAR) has its own container orchestration (Diego), although it can also be used with Kubernetes.[2]

Stack Topologies for Application Clusters

An application cluster is comprised of different moving parts. One set of parts is the applications and services that manage the cluster. Some of these cluster management services include:

- Scheduler, which decides how many instances of each application to run, and where to run them
- Monitoring, to detect issues with application instances so that they can be restarted or moved if necessary
- Configuration registry, for storing information needed to manage the cluster and configure applications
- Service discovery, enabling applications and services to find the current location of application instances
- Management API and UI, enabling tools and users to interact with the cluster

Many cluster deployments run management services on dedicated servers, separate from the services that host application instances. These services should probably also run clustered, for resilience.

The other main parts of an application cluster are the application hosting nodes. These nodes are a pool of servers where the scheduler runs application instances. It's common to set these up as a server cluster (see "Compute Resources" on page 28) to manage the number and location of server instances. A service mesh (see "Service Mesh" on page 244) may run sidecar processes on the host nodes, alongside application instances.

An example application cluster (Figure 14-2) includes servers to run cluster management services, a server cluster to run application instances, and network address blocks.

[2] See "CF Container Runtime" for more details.

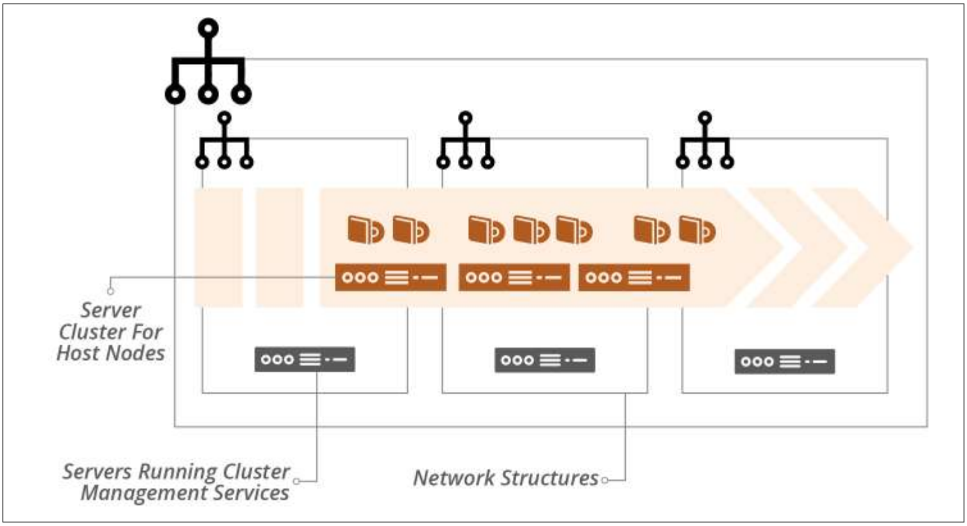


Figure 14-2. An example application cluster

Networking structures for an application cluster may be flat. The cluster services assign network addresses to application instances. It should also handle network security, including encryption and connection management, often using a service mesh for this.

You use infrastructure stacks to provision this infrastructure. Chapter 5 explained that the size of a stack and the scope of its contents have implications for the speed and risk of changes.

"Patterns and Antipatterns for Structuring Stacks" on page 56 in particular lists patterns for arranging resources across stacks. The following examples show how these patterns apply to cluster infrastructure.

Monolithic Stack Using Cluster as a Service

The simplest design is to define all of the parts of your cluster in a single stack, following the monolithic stack antipattern (see "Antipattern: Monolithic Stack" on page 56). Although monoliths become an antipattern at scale, a single stack can be useful when starting out with a small and simple cluster.

Example 14-1 uses a cluster as a service, similar to AWS EKS, AWS ECS, Azure AKS, and Google GKE. So the code defines the cluster, but it doesn't need to provision servers to run the cluster management, because the infrastructure platform handles that behind the scenes.

Example 14-1. Stack code that defines everything for a cluster

```
address_block:
  name: cluster_network
  address_range: 10.1.0.0/16"
  vlans:
    - vlan_a:
        address_range: 10.1.0.0/8
    - vlan_b:
        address_range: 10.1.1.0/8
    - vlan_c:
        address_range: 10.1.2.0/8

application_cluster:
  name: product_application_cluster
  address_block: $address_block.cluster_network

server_cluster:
  name: "cluster_nodes"
  min_size: 1
  max_size: 3
  vlans: $address_block.cluster_network.vlans
  each_server_node:
    source_image: cluster_node_image
    memory: 8GB
```

This example leaves out many of the things you'd have for a real cluster, like network routes, security policies, and monitoring. But it shows that the significant elements of networking, cluster definition, and server pool for host nodes are all in the same project.

Monolithic Stack for a Packaged Cluster Solution

The code in Example 14-1 uses an application cluster service provided by the infrastructure platform. Many teams instead use a packaged application cluster solution (as described in "Packaged Cluster Distribution" on page 231). These solutions have installers that deploy the cluster management software onto servers.

When using one of these solutions, your infrastructure stack provisions the infrastructure that the installer needs to deploy and configure the cluster. Running the installer should be a separate step. This way, you can test the infrastructure stack separately from the application cluster. Hopefully, you can define the configuration for your cluster as code. If so, you can manage that code appropriately, with tests and a pipeline to help you to deliver updates and changes easily and safely.

It may be useful to use server configuration code (as in Chapter 11) to deploy your cluster management system onto your servers. Some of the packaged products use standard configuration tools, such as Ansible for OpenShift, so you may be able to incorporate these into your stack building process. Example 14-2 shows a snippet that you would add to the monolithic stack code in Example 14-1 to create a server for the cluster management application.

Example 14-2. Code to build a cluster management server

```
virtual_machine:
  name: cluster_manager
  source_image: linux-base
  memory: 4GB
  provision:
    tool: servermaker
    parameters:
      maker_server: maker.shopspinner.xyz
      role: cluster_manager
```

The code configures the server by running the fictitious `servermaker` command, applying the `cluster_manager` role.

Pipeline for a Monolithic Application Cluster Stack

Since there is only one stack, a single pipeline can test and deliver code changes to instances of the application cluster. However, there are other elements involved, including the server image for the host nodes and the applications themselves. Figure 14-3 shows a potential design for these pipelines.

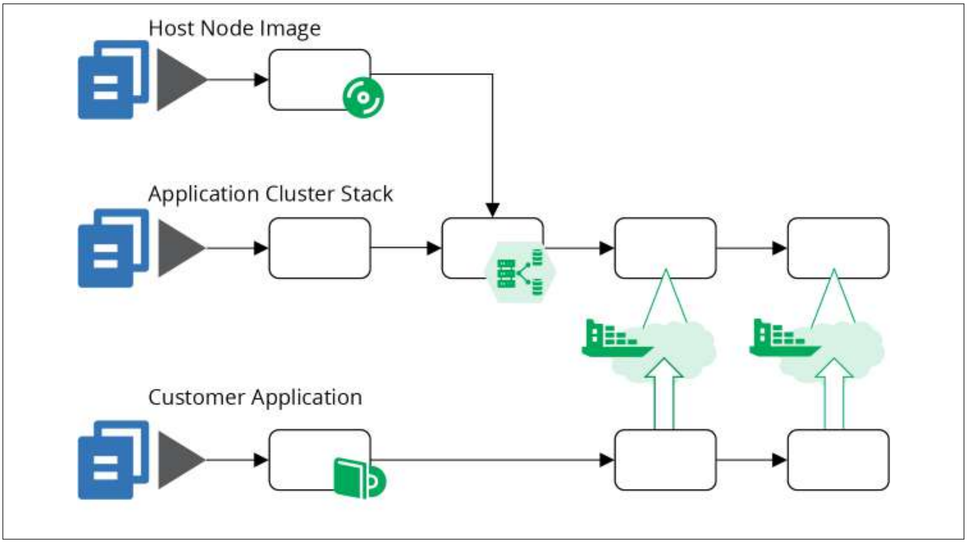


Figure 14-3. An example of pipelines for a cluster using a monolithic stack

The top pipeline (Figure 14-4) builds a server image for the host nodes, as described in "Using a Pipeline to Test and Deliver a Server Image" on page 221. The result of this pipeline is a server image that has been tested in isolation. The tests for that image probably check that the container management software is installed and that it complies with security policies.

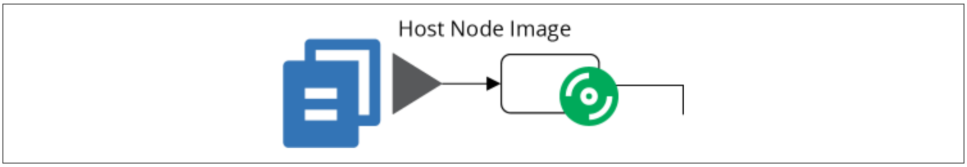


Figure 14-4. The pipeline for the host node server image

The bottom pipeline (Figure 14-5) is for an application that deploys onto the cluster. In practice, you'll have a number of these, one for every separately deployed application. This pipeline includes at least one early stage to build and test the application on its own. It then has stages that deploy the application to the cluster in each environment. The application can be tested, reviewed, and made available for production use in these environments. The pipelines for the applications are very loosely coupled with the pipelines for the cluster instances. You may choose to trigger application testing stages after cluster updates. Doing this helps you to find any issues that the change to the cluster causes for the application by running the application-specific tests.

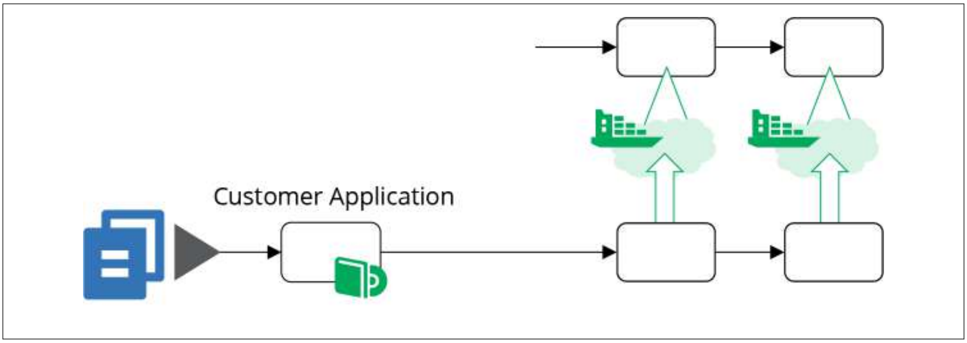


Figure 14-5. Pipelines for delivering applications to the cluster

The pipeline for the application cluster stack in Figure 14-6 starts with an offline stage ("Offline Testing Stages for Stacks" on page 131) that runs some syntax checking, and applies the stack code to a local mock of the infrastructure platform ("Testing with a Mock API" on page 133). These tests can catch problems at the coding level, without needing to use infrastructure platform resources, so they run quickly.

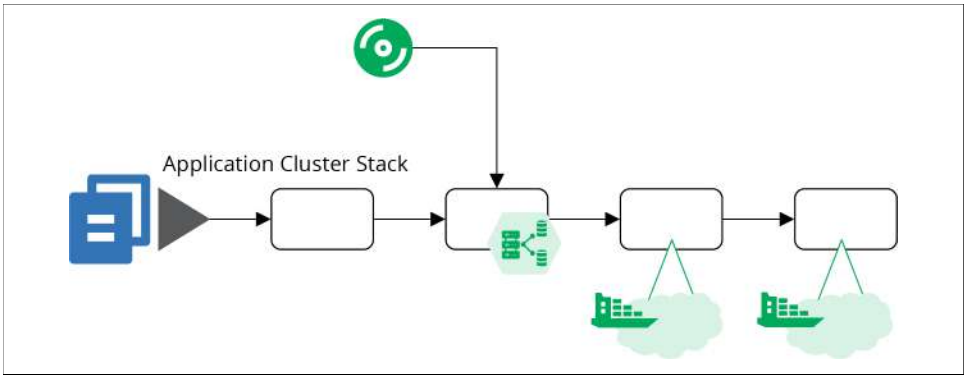


Figure 14-6. The pipeline for the cluster stack code

The second stage of this pipeline is an online stage (see "Online Testing Stages for Stacks" on page 134), and creates an instance of the stack on the infrastructure platform. The instance might be persistent (see "Pattern: Persistent Test Stack" on page 142) or ephemeral (see "Pattern: Ephemeral Test Stack" on page 143). The tests in this stage can check that the cluster management services are correctly created and accessible. You can also test for security issues—for instance, making sure that access to cluster management endpoints is locked down.[3]

[3] Kubernetes has historically had issues allowing the management API to be used without authentication. Follow guides to ensure you're taking measures to secure your cluster, and write tests to stop code and configuration changes that accidentally create an opening for attackers.

Because this monolithic cluster stack includes the code to create the host node servers, the online testing stage can test these as well. A test could deploy a sample application to the cluster and prove that it's working. The advantage of using a sample application rather than a real application is that you can keep it simple. Strip it down to a minimal set of dependencies and configurations, so you can be sure any test failures are caused by issues with the cluster provisioning, rather than due to complexities of deploying a real-world application.

Note that this pipeline stage is heavy. It tests both the cluster configuration and the host node server cluster. It also tests the server image in the context of the cluster. There are plenty of different things that can cause this stage to fail, which complicates troubleshooting and fixing failures.

Most of the elapsed time for this stage will almost certainly be taken up by provisioning everything, far more than the time to execute the tests. These two issues—the variety of things tested in this one stage, and the time it takes to provision—are the main drivers for breaking the cluster into multiple stacks.

Example of Multiple Stacks for a Cluster

Breaking the infrastructure code for a cluster into multiple stacks can improve the reliability and speed of your process for making changes. Aim to design each stack so that you can provision and test it in isolation, rather than needing to provision instances of other stacks.

Start by pulling the host node server pool into a separate stack, as in Example 14-3. You can provision and test an instance of this stack without the application cluster. Test that the platform successfully boots servers from your images and that network routes work correctly. You can also test reliability, triggering a problem with one of the servers and proving whether the platform automatically replaces it.

Example 14-3. Stack code that defines the server pool of host nodes

```
server_cluster:
  name: "cluster_nodes"
  min_size: 1
  max_size: 3
  vlans: $address_block.host_node_network.vlans
  each_server_node:
    source_image: cluster_node_image
    memory: 8GB

address_block:
  name: host_node_network
  address_range: 10.2.0.0/16"
  vlans:
    - vlan_a:
        address_range: 10.2.0.0/8
    - vlan_b:
        address_range: 10.2.1.0/8
    - vlan_c:
        address_range: 10.2.2.0/8
```

This code adds separate VLANs for the host nodes, unlike the earlier code for the monolithic stack (see Example 14-1). It's good practice to split the host nodes and the cluster management into different network segments, which you could do within the monolithic stack. Breaking the stacks apart leads us to do this, if only to reduce coupling between the two stacks.

Breaking the stacks apart adds a new pipeline for the host node cluster stack, as shown in Figure 14-7.

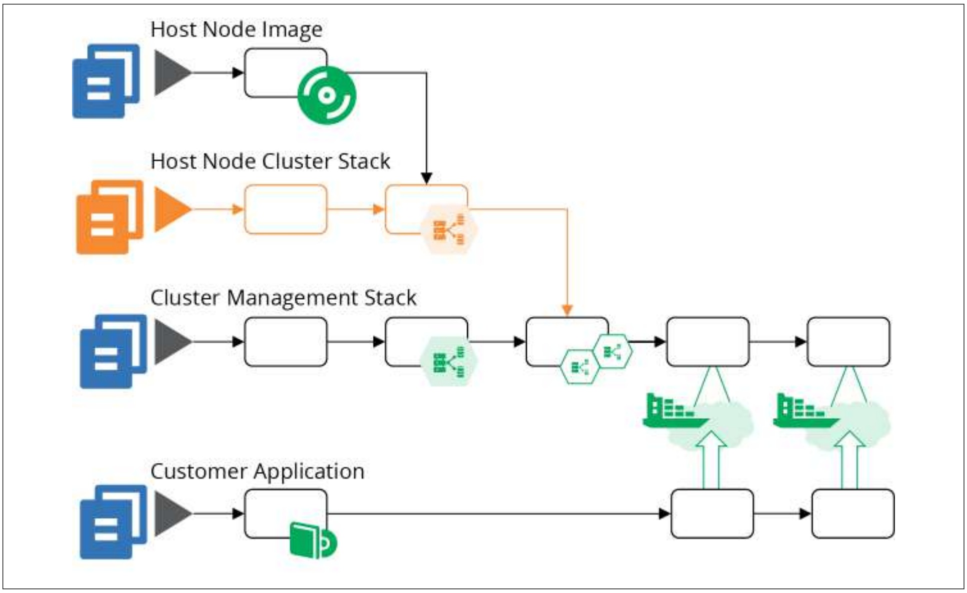


Figure 14-7. Added pipeline for the host node pool

Although there are a few more stages in this combined pipeline, they are lighter and faster. The online test stage for the cluster management stack (highlighted in Figure 14-8) only provisions the cluster management infrastructure, which is faster than the online stage in the monolithic stack pipeline. This stack no longer depends on the pipeline for host node server images, and doesn't include the server nodes. So the tests in this stage can focus on checking that cluster management is configured and secured correctly.

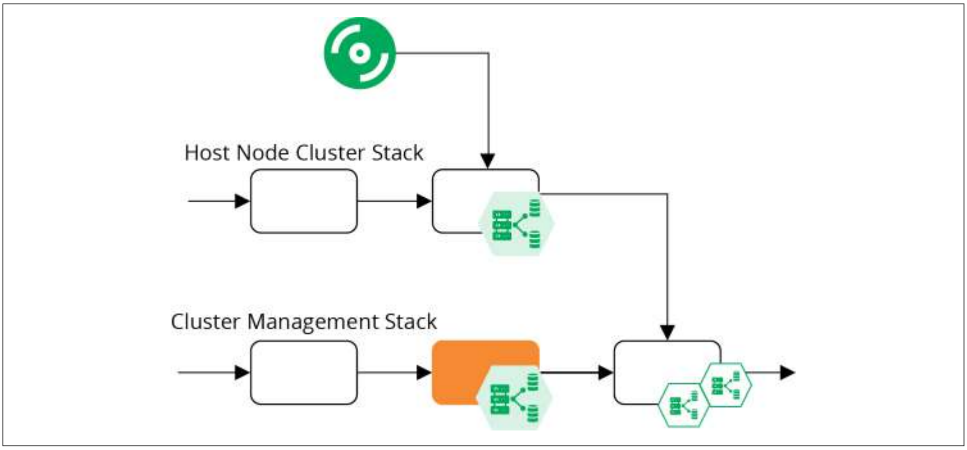


Figure 14-8. Online test stages for the cluster pipelines

This revised design joins the pipeline for the host node server stack together with the pipeline for the cluster management stack in a stack integration stage, as shown in Figure 14-9.

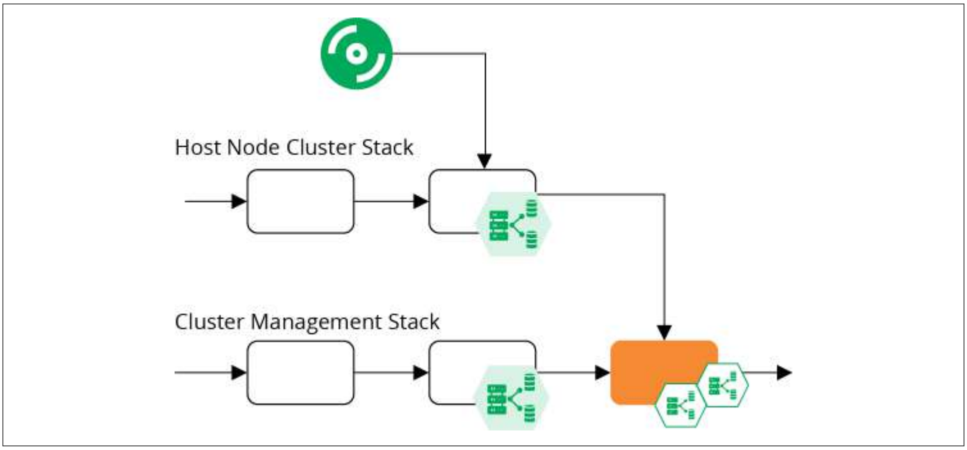


Figure 14-9. Stack integration test stage for a cluster

This is an online test stage that provisions and tests instances of both stacks together. These tests can focus on issues that only emerge with this combination, so shouldn't duplicate testing activities from previous stages. This is the stage where you would deploy a sample application and prove that it runs correctly on the cluster. You could also test reliability and scaling by triggering failures in your test application and creating the conditions needed to add additional instances.

You might decide to split this into more stacks; for example, breaking the common networking infrastructure out of the management stack. Chapters 15 and 17 go into more details of decomposing and integrating infrastructure across stacks.

Sharing Strategies for Application Clusters

How many clusters should you run, how big should they be, and how much should you run on each?

In theory, you could run a single cluster, manifesting environments and other application boundaries within the cluster instance. However, there are many reasons why a single cluster may not be practical:[4]

Managing changes

You need to update, upgrade, fix, and change your cluster. So at the least, you need somewhere to test these changes that won't interrupt services. For disruptive changes, like those that require or risk downtime, scheduling a time that meets the needs of all teams, applications, and regions is challenging. Running multiple clusters makes it easier to schedule maintenance windows, and reduces the impact of a failed change.

Segregation

Many clustering implementations don't provide strong enough segregation between applications, data, and configuration. You may also have different governance regimes for the cluster implementation based on the services running on it. For example, services that handle credit card numbers may have stricter compliance requirements, so running them on a separate cluster simplifies the requirements for your other clusters.

Configurability

Some applications or teams have different configuration requirements for the clusters they use. Providing them with separate cluster instances reduces configuration conflict.

Performance and scalability

Clustering solutions have different scaling characteristics. Many don't cope with higher latency, which makes it impractical to run a single cluster across geographical regions. Applications may hit resource limitations or contention issues with one another when they scale up on a single cluster.

[4] Rob Hirschfeld explores the trade-offs between cluster sizes and sharing in his article, "The Optimal Kubernetes Cluster Size? Let's Look at the Data".

Availability
: A single cluster is a single point of failure. Running multiple clusters can help cope with various failure scenarios.

There are a few potential strategies for sizing and sharing cluster instances. To choose the right strategy for your system, consider your requirements for change segregation, configurability, performance, scale, distribution, and availability. Then test your application clustering solution against those requirements.

One Big Cluster for Everything

A single cluster can be simpler to manage than multiple clusters. The likely exception is in managing changes. So at a minimum, you should use at least one separate cluster instance to test changes, using a pipeline to deploy and test cluster configuration changes there before applying them to your production cluster.

Separate Clusters for Delivery Stages

You can run different clusters for different parts of your software delivery process. Doing this could be as simple as running one cluster per environment (see Figure 14-10).

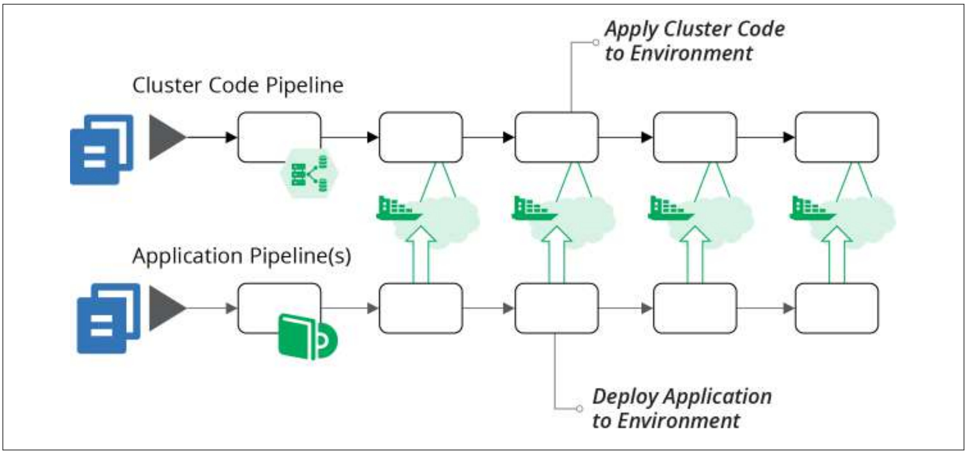

Apply Cluster Code to Environment
Cluster Code Pipeline
Application Pipeline(s)
Deploy Application to Environment

Figure 14-10. Pipelines managing one cluster for each application deployment environment

A dedicated cluster for every environment avoids inconsistencies you might get from having applications from multiple environments sharing resources. However, it may be difficult and expensive to maintain a separate instance for every delivery stage. For instance, if your delivery process dynamically creates test instances, you may need to dynamically create cluster instances to run them, which can be very slow.

A variation of separating clusters by delivery stage is to share clusters across multiple stages. For example, you might use different clusters based on governance requirements. In Figure 14-11, there are three clusters. The `DEV` cluster is for development, running instances where people create and use various data sets for more exploratory testing scenarios. The `NON-PROD` cluster is for more rigorous delivery stages, with managed test data sets. The `PROD` cluster hosts the `PREPROD` and `PROD` environments, both of which contain customer data and so have more rigorous governance requirements.

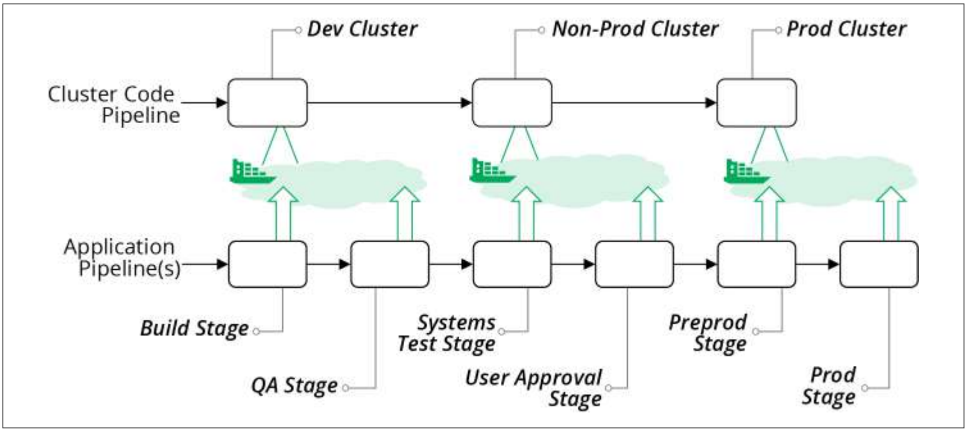


Figure 14-11. Cluster instances shared across multiple environments

When hosting multiple environments in a shared cluster, you should aim to keep each environment segregated as much as possible. Ideally, applications and operational services should not be able to see or interact with instances from other environments. Many of the mechanisms that application clustering solutions provide for separating applications are "soft." For instance, you may be able to tag instances to indicate the environment, but this is purely a convention. You should look for stronger methods to segregate applications.

Clusters for Governance

One of the benefits of having separate clusters for different parts of the delivery process is that the governance requirements are usually different for different stages in the process. Production has tighter requirements because services running there are the most business-critical, and the data is the most sensitive.

Quite often, different parts of a system have different governance and compliance requirements that cut across delivery stages. The most common example is services that handle credit card numbers, which are subject to PCI standards. Other examples include services that deal with customer personal data, which may be subject to regimes such as GDPR.

Hosting services that are subject to stricter standards on dedicated clusters can simplify and strengthen compliance and auditing. You can impose stronger controls on these clusters, applications running on them, and delivery of code changes to them. Clusters hosting services with less strict compliance requirements can have streamlined governance processes and controls.

As an example, you could have two clusters, one used for development, testing, and production hosting for regulated services, and one for the unregulated services. Or you may split cluster instances by delivery stage and by regulation requirements, as illustrated in Figure 14-12.

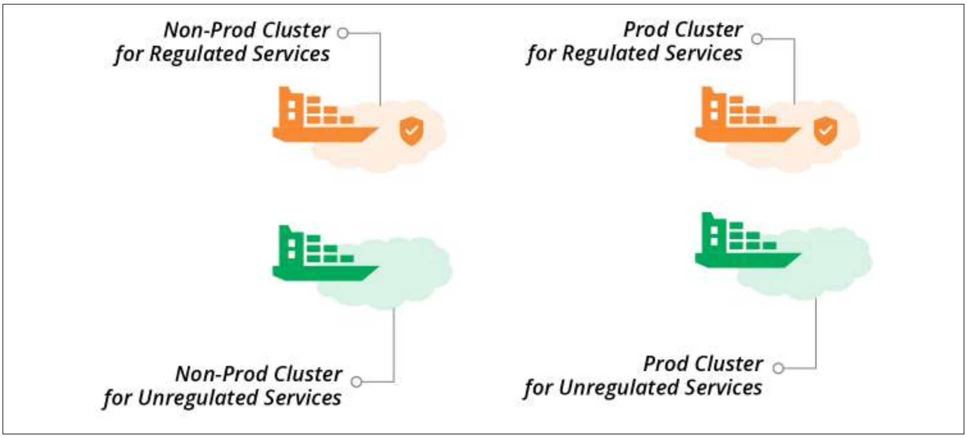


Figure 14-12. Separate clusters for delivery stage and regulation requirements

Clusters for Teams

Yet another factor for organizing multiple clusters is team ownership. Often, different teams are responsible for delivering and running different types of applications and services, which may have different hosting requirements. For example, a team owning customer-facing services may have different requirements for governance and availability than a team that owns services for an internal department. A cluster allocated to a team can be optimized for the requirements of that team and its applications.

Service Mesh

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 described in "The Parts of an Infrastructure System" on page 23. In a typical service mesh implementation, each application instance delegates communication with other instances to a sidecar process (see Figure 14-13).

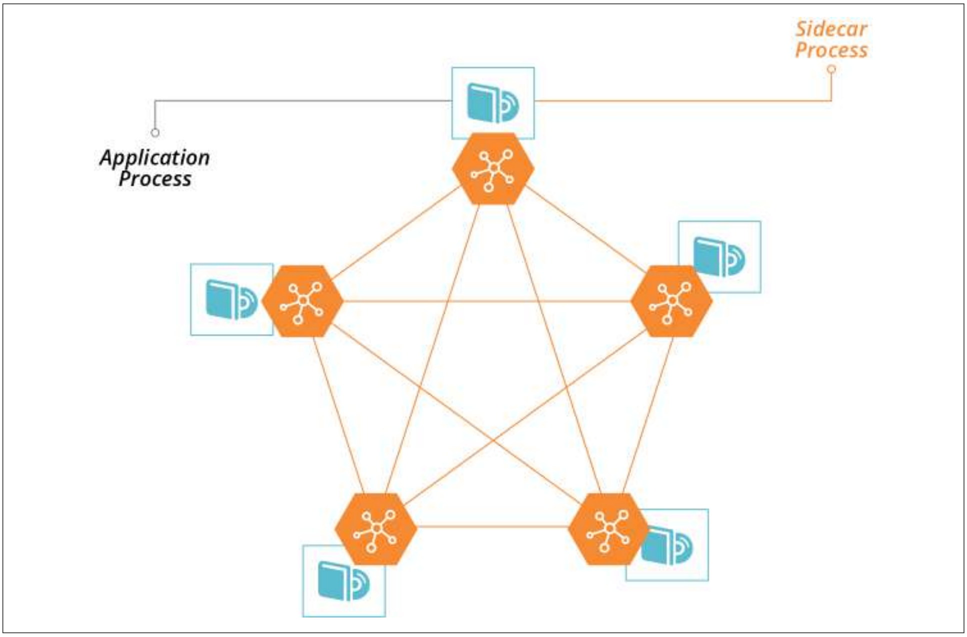


Figure 14-13. Sidecars enable communication with other processes in a service mesh

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 "Changing Live Infrastructure" on page 368.

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 an application hosting cluster. The application cluster dynamically provides compute resources decoupled from lower-level resources. The service mesh dynamically manages application communication decoupled from lower-level networking resources. The benefits of this model are:

- It simplifies application development, by moving common concerns out of the application and into the sidecar.
- It makes 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.
- It handles 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 noncontainerized systems; for example, by deploying sidecar processes onto virtual machines.

A service mesh adds 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.

Infrastructure for FaaS Serverless

Chapter 3 lists FaaS serverless as one of the ways a platform can provide compute resources to applications (see "Compute Resources" on page 28). The normal model for application code is to run it continuously in a container or server. FaaS executes application code on demand, in response to an event or schedule.

FaaS 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.

FaaS 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 FaaS rather than serverless. This also disambiguates FaaS from other uses of the term serverless, which can also mean *Backend as a Service* (BaaS), which is an externally hosted service.[5]

FaaS runtimes follow the same models as application clusters—FaaS runtime provided as a service by your infrastructure platform, and packaged FaaS, which requires you to provision and configure infrastructure and management tools.

Examples of FaaS runtime provided as a service include:

- AWS Lambda
- Azure Functions
- Google Cloud Functions

Examples of packaged FaaS runtime solutions include:

- Fission
- Kubeless
- OpenFaaS
- Apache OpenWhisk

You can use the same strategies described earlier in this chapter for provisioning infrastructure for a packaged FaaS solution, such as server pools and management services. Be sure you understand how your FaaS solution works in depth, so you are aware of whether and how code may "leak" data. For example, it may leave temporary files and other remnants in locations that may be available to other FaaS code, which can create issues for security and compliance. How well the FaaS solution meets your needs for segregating data, and whether it can scale, should drive your decisions on whether to run multiple instances of your FaaS runtime.

The FaaS services provided by cloud vendors usually don't leave as much for you to configure as application clusters do. For example, you normally won't need to specify the size and nature of the host servers the code executes on. This drastically reduces the amount of infrastructure you need to define and manage.

[5] See Mike Roberts's definitive article "Serverless Architectures" for more.

However, most FaaS code does interact with other services and resources. You may need to define networking for inbound requests that trigger a FaaS application, and for outbound requests the code makes. FaaS code often reads and writes data and messages to storage devices, databases, and message queues. These all require you to define and test infrastructure resources. And of course, FaaS code should be delivered and tested using a pipeline, as with any other code. So you still need all of the practices around defining and promoting infrastructure code and integrating it with application testing processes.

Conclusion

Computing infrastructure exists to support services. Services are provided by application software, which runs on a runtime environment. This chapter has described how to use code to define infrastructure that provides runtime environments for your organization's applications.

PART IV

Designing Infrastructure

CHAPTER 15
Core Practice: Small, Simple Pieces

A successful system tends to grow over time. More people use it, more people work on it, more things are added to it. As the system grows, changes become riskier and more complex. This often leads to more complicated and time-consuming processes for managing changes. Overhead for making changes makes it harder to fix and improve the system, allowing technical debt to grow, eroding the quality of the system.

This is the negative version of the cycle of speed of change driving better quality, and better quality enabling faster speed of change described in Chapter 1.

Applying the three core practices of Infrastructure as Code—defining everything as code, continuously testing and delivering, and building small pieces ("Three Core Practices for Infrastructure as Code" on page 9)—enables the positive version of the cycle.

This chapter focuses on the third practice, composing your system from smaller pieces so that you can maintain a faster rate of change while improving quality even as your system grows. Most infrastructure coding tools and languages have features that support modules, libraries, and other types of components. But infrastructure design thinking and practice hasn't yet reached the level of maturity of software design.

So this chapter draws on design principles for modularity learned from decades of software design, considering them from the point of view of code-driven infrastructure. It then looks at different types of components in an infrastructure system, with an eye to how we can leverage them for better modularity. Building on this, we can look at different considerations for drawing boundaries between infrastructure components.

Designing for Modularity

The goal of modularity is to make it easier and safer to make changes to a system. There are several ways modularity supports this goal. One is to remove duplication of implementations, to reduce the number of code changes you need to make to deliver a particular change. Another is to simplify implementation by providing components that can be assembled in different ways for different uses.

A third way to make changes easier and safer is to design the system so that you can make changes to a smaller component without needing to change other parts of the system. Smaller pieces are easier, safer, and faster to change than larger pieces.

Most design rules for modularity have a tension. Followed carelessly, they can actually make a system more brittle and harder to change. The four key metrics from "The Four Key Metrics" on page 9 are a useful lens for considering the effectiveness of modularizing your system.

Characteristics of Well-Designed Components

Designing components is the art of deciding which elements of your system to group together, and which to separate. Doing this well involves understanding the relationships and dependencies between the elements. Two important design characteristics of a component are coupling and cohesion. The goal of a good design is to create low coupling and high cohesion.

Coupling describes how often a change to one component requires a change to another component. Zero coupling isn't a realistic goal for two parts of a system. Zero coupling probably means they aren't a part of the same system at all. Instead, we aim for low, or loose coupling.

A stack and a server image are coupled, because you may need to increase the memory allocated to the server instance in the stack when you upgrade software on the server. But you shouldn't need to change code in the stack every time you update the server image. Low coupling makes it easier to change a component with little risk of breaking other parts of the system.

Cohesion describes the relationship between the elements within a component. As with coupling, the concept of cohesion relates to patterns of change. Changes to a resource defined in a stack with low cohesion are often not relevant to other resources in the stack.

An infrastructure stack that defines separate networking structures for servers provisioned by two other stacks has low cohesion. Components with high cohesion are easier to change because they are smaller, simpler, and cleaner, with a lower blast radius ("Blast Radius" on page 58), than components that include a mash of loosely related things.

Four Rules of Simple Design

Kent Beck, the creator of XP and TDD, often cites four rules for making the design of a component simple. According to his rules, simple code should:

- Pass its tests (do what it is supposed to do)
- Reveal its intention (be clear and easy to understand)
- Have no duplication
- Include the fewest elements

Rules for Designing Components

Software architecture and design includes many principles and guidelines for designing components with low coupling and high cohesion.

Avoid duplication

The DRY (Don't Repeat Yourself) principle says, "Every piece of knowledge must have a single, unambiguous, authoritative representation within a system."[1] Duplication forces people to make a change in multiple places.

For example, all of ShopSpinner's stacks use a `provisioner` user account to apply configuration to server instances. Originally, the login details for the account are specified in every stack as well as the code that builds their base server image. When someone needs to change the login details for the user account, they need to find and change it in all of these locations in the codebase. So the team moves the login details in a central location, and each stack, plus the server image builder, refers to that location.

[1] The DRY principle, and others, can be found in *The Pragmatic Programmer: From Journeyman to Master* by Andrew Hunt and David Thomas (Addison-Wesley).

Duplication Considered Useful

The DRY principle discourages duplicating the implementation of a concept, which is not the same as duplicating literal lines of code. Having multiple components depend on shared code can create tight coupling, making it hard to change.

I've seen teams insist on centralizing any code that looks similar; for example, having all virtual servers created using a single module. In practice, servers created for different purposes, such as application servers, web servers, and build servers, usually need to be defined differently. A module that needs to create all of these different types of servers can become overly complicated.

When considering whether code is duplicated and should be centralized, consider whether the code truly represents the same concept. Does changing one instance of the code always mean the other instance should change as well?

Also consider whether it's a good idea to lock the two instances of code together into the same change cycle. Forcing every application server in the organization to upgrade at the same time may be unrealistic.

Reuse increases coupling. So a good rule of thumb for reuse is to be DRY within a component, and wet across components.

Rule of composition

To make a composable system, make independent pieces. It should be easy to replace one side of a dependency relationship without disturbing the other.[2]

The ShopSpinner team starts with a single Linux application server image that they provision from different stacks. Later they add a Windows application server image. They design the code that provisions server images from any given stack so that they can easily switch between these two server images as needed for a given application.

Single responsibility principle

The single responsibility principle (SRP) says that any given component should have responsibility for one thing. The idea is to keep each component focused, so that its contents are cohesive.[3]

[2] The rule of composition is one of the basic rules of the Unix philosophy.

[3] My colleague James Lewis applies the SRP to the question of how big a microservice should be. His advice is that a component should fit in your head, conceptually, which applies to infrastructure as well as software components.

An infrastructure component, whether it's a server, configuration library, stack component, or a stack, should be organized with a single purpose in mind. That purpose may be layered. Providing the infrastructure for an application is a single purpose that can be fulfilled by an infrastructure stack. You could break that purpose down into secure traffic routing for an application, implemented in a stack library; an application server, implemented by a server image; and a database instance, implemented by a stack module. Each component, at each level, has a single, easily understood purpose.

Design components around domain concepts, not technical ones

People are often tempted to build components around technical concepts. For example, it might seem like a good idea to create a component for defining a server, and reuse that in any stack that needs a server. In practice, any shared component couples all of the code it uses.

A better approach is to build components around a domain concept. An application server is a domain concept that you may want to reuse for multiple applications. A build server is another domain concept, which you may want to reuse to give different teams their own instance. So these make better components than servers, which are probably used in different ways.

Law of Demeter

Also called the *principle of least knowledge*, the Law of Demeter says that a component should not have knowledge of how other components are implemented. This rules pushes for clear, simple interfaces between components.

The ShopSpinner team initially violates this rule by having a stack that defines an application server cluster, and a shared networking stack that defines a load balancer and firewall rules for that cluster. The shared networking stack has too much detailed knowledge of the application server stack.

> **Providers and Consumers**
>
> In a dependency relationship between components a *provider* component creates or defines a resource that a *consumer* component uses.
>
> A shared networking stack may be a provider, creating networking address blocks such as subnets. An application infrastructure stack may be a consumer of the shared networking stack, provisioning servers and load balancers within the subnets managed by the provider.
>
> A key topic of this chapter is defining and implementing interfaces between infrastructure components.

No circular dependencies

As you trace relationships from a component that provides resources to consumers, you should never find a loop (or cycle). In other words, a provider component should never consume resources from one of its own direct or indirect consumers.

The ShopSpinner example of a shared network stack has a circular dependency. The application server stack assigns the servers in its cluster to network structures in the shared networking stack. The shared networking stack creates load balancer and firewall rules for the specific server clusters in the application server stack.

The ShopSpinner team can fix the circular dependencies, and reduce the knowledge the networking stack has of other components, by moving the networking elements that are specific to the application server stack into that stack. This also improves cohesion and coupling, since the networking stack no longer contains elements that are most closely related to the elements of another stack.

Use Testing to Drive Design Decisions

Chapters 8 and 9 describe practices for continuously testing infrastructure code as people work on changes. This heavy focus on testing makes testability an essential design consideration for infrastructure components.

Your change delivery system needs to be able to create and test infrastructure code at every level, from a server configuration module that installs a monitoring agent, to stack code that builds a container cluster. Pipeline stages must be able to quickly create an instance of each component in isolation. This level of testing is impossible with a spaghetti codebase with tangled dependencies, or with large components that take half an hour to provision.

These challenges derail many initiatives to introduce effective automated testing regimes for infrastructure code. It's hard to write and run automated tests for a poorly designed system.

And this is the secret benefit of automated testing: it drives better design. The only way to continuously test and deliver code is to implement and maintain clean system designs with loose coupling and high cohesion.

It's easier to implement automated testing for server configuration modules that are loosely coupled. It's easier to build and use mocks for a module with cleaner, simpler interfaces (see "Using Test Fixtures to Handle Dependencies" on page 137). You can provision and test a small, well-defined stack in a pipeline more quickly.

Modularizing Infrastructure

An infrastructure system involves different types of components, as described in Chapter 3, each of which can be composed of different parts. A server instance may be built from an image, using a server configuration role that references a set of server configuration modules, which in turn may import code libraries. An infrastructure stack may be composed of server instances, and may use stack code modules or libraries. And multiple stacks may combine to comprise a larger environment or estate.

Stack Components Versus Stacks as Components

The infrastructure stack, as defined in Chapter 5, is the core deployable unit of infrastructure. The stack is an example of an *Architectural Quantum*, which Ford, Parsons, and Kua define as, "an independently deployable component with high functional cohesion, which includes all the structural elements required for the system to function correctly."[4] In other words, a stack is a component that you can push into production on its own.

As mentioned previously, a stack can be composed of components, and a stack may itself be a component. Servers are one potential component of a stack, which are involved enough to drill into later in this chapter. Most stack management tools also support putting stack code into modules, or using libraries to generate elements of the stack.

Figure 15-1 shows two stacks, the imaginatively named `StackA` and `StackB`, which use a shared code module that defines a networking structure.

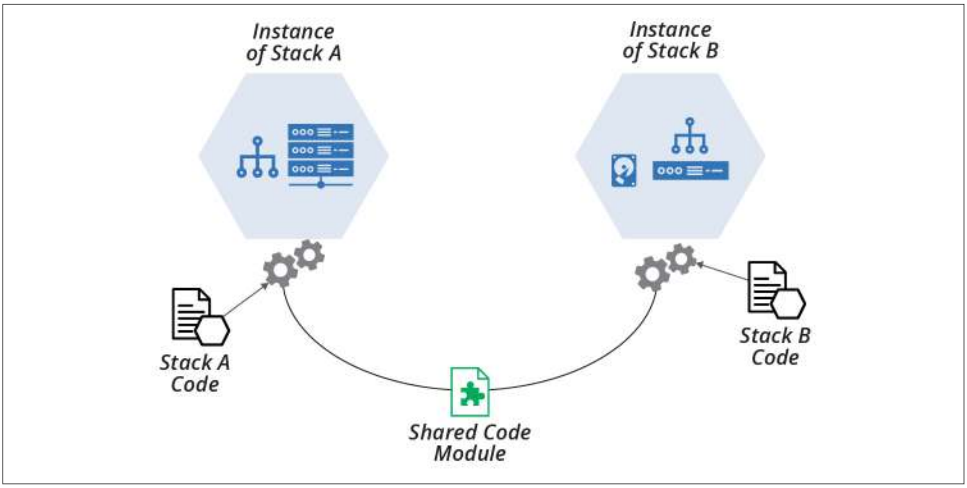


Figure 15-1. Shared code module used by two stacks

[4] See chapter 4 of *Building Evolutionary Architectures* by Neal Ford, Rebecca Parsons, and Patrick Kua (O'Reilly).

Chapter 16 describes some patterns and antipatterns for using stack code modules and libraries.

Stack modules and libraries are useful for reusing code. However, they are less helpful for making stacks easy to change. I've seen teams try to improve a monolithic stack (see "Antipattern: Monolithic Stack" on page 56) by breaking the code into modules. While modules made the code easier to follow, each stack instance was just as large and complex as before.

Figure 15-2 shows that code separated into separate modules are combined into the stack instance.

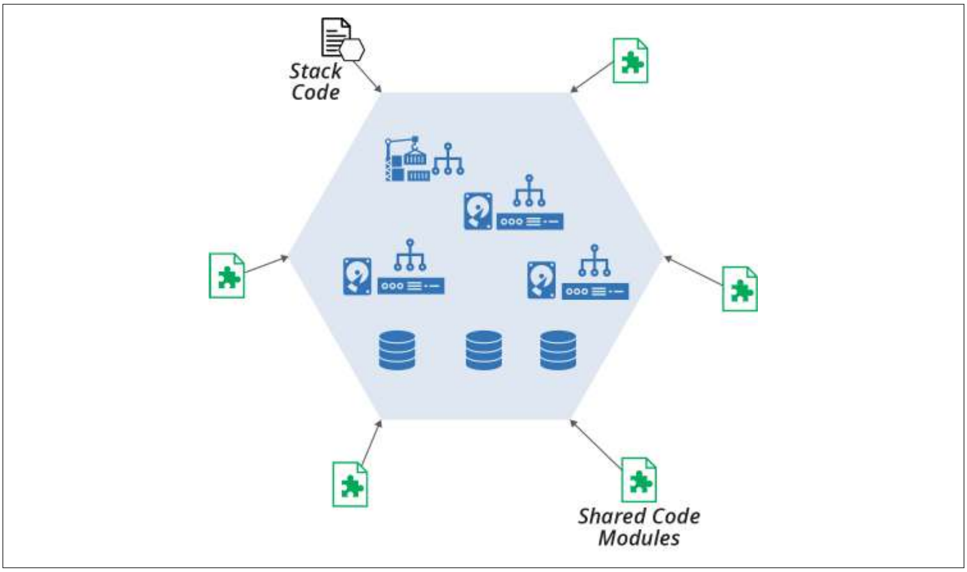


Figure 15-2. Stack modules add complexity to the stack instance

In addition to adding to the elements with each stack instance, a module that is also used by other stacks creates coupling between those stacks. Changing the module to accommodate a requirement in one stack may affect other stacks that use the module. This coupling may add friction for making changes.

A more fruitful approach to making a large stack more manageable is to break it into multiple stacks, each of which can be provisioned, managed, and changed independently of others. "Patterns and Antipatterns for Structuring Stacks" on page 56 lists a few patterns for considering the size and contents of a stack. Chapter 17 goes into more detail of managing the dependencies between stacks.

Using a Server in a Stack

Servers are a common type of stack component. Chapter 11 explained the various components of a server and its life cycle. Stack code normally incorporates servers through some combination of server images (see Chapter 13) and server configuration modules (see "Server Configuration Code" on page 172), often by specifying a role (see "Server Roles" on page 175).

The ShopSpinner team's codebase includes an example of using a server image as a component of a stack. It has a stack called `cluster_of_host_nodes`, which builds a cluster of servers to act as container host nodes, as shown in Figure 15-3.

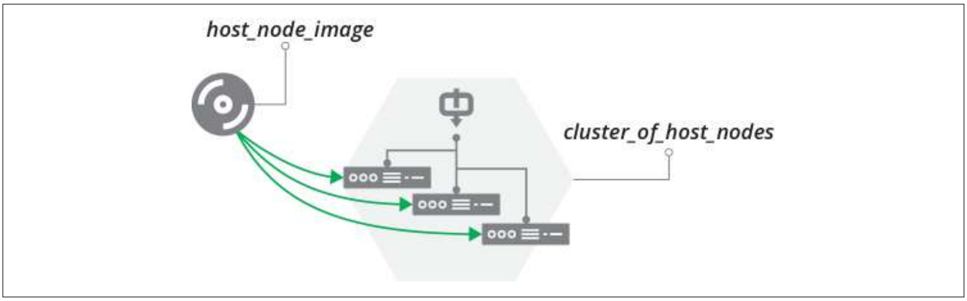


Figure 15-3. Server image as a provider to a stack

The code that defines the server cluster specifies the name of the server image, `host_node_image`:

```
server_cluster:
  name: "cluster_of_host_nodes"
  min_size: 1
  max_size: 3
  each_server_node:
    source_image: host_node_image
    memory: 8GB
```

The team uses a pipeline to build and test changes to the server image. Another pipeline tests changes to `cluster_of_host_nodes`, integrating it with the latest version of `host_node_image` that has passed its tests (see Figure 15-4).

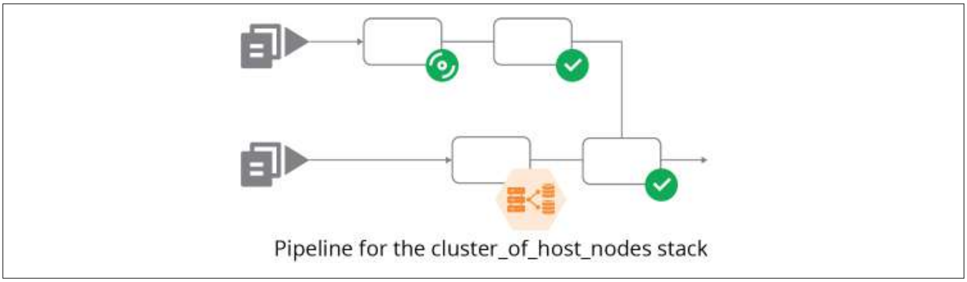


Figure 15-4. Pipeline to integrate the server image and its consumer stack

"Infrastructure Delivery Pipelines" on page 119 explained how pipelines can work for infrastructure.

This example has a slight issue, however. The first pipeline stage for the `cluster_of_host_nodes` stack doesn't use the `host_node_image`. But the stack code example includes the name of the server image, so it can't be run as an online test stage ("Online Testing Stages for Stacks" on page 134). Testing the stack code without the image might be useful, so the team can find problems with the stack code without having to also provision the full host node servers, which are heavy.

The ShopSpinner team addresses the problem by extracting the hardcoded `host_node_image` out of the stack code, using a stack parameter instead (Chapter 7). This code is more testable:

```
server_cluster:
  name: "cluster_of_host_nodes"
  min_size: 1
  max_size: 3
  each_server_node:
    source_image: ${HOST_NODE_SERVER_IMAGE}
    memory: 8GB
```

The online test stage for the `cluster_of_host_nodes` stack can set the `HOST_NODE_SERVER_IMAGE` parameter with the ID for a stripped-down server image. The team can run tests in this stage to validate that the server cluster works correctly, scaling up and down and recovering failed instances. The stripped-down server image is an example of a test double (see "Using Test Fixtures to Handle Dependencies" on page 137).

The simple change of replacing the hardcoded reference to the server image with a parameter reduces coupling. It also follows the rule of composition (see "Rule of composition" on page 254). The team can easily create instances of `cluster_of_host_nodes` using a different server image, which would come in handy if people on the team want to test and incrementally roll out a different operating system for their clusters.

Shared-Nothing Infrastructure

In the distributed computing field, a *shared-nothing architecture* enables scaling by ensuring that new nodes can be added to a system without adding contention for any resources outside the node itself.

The typical counter-example is a system architecture where processors share a single disk. Contention for the shared disk limits the scalability of the system when adding more processors. Removing the shared disk from the design means the system can scale closer to linearly by adding processors.

A shared-nothing design with infrastructure code moves resources from a shared stack to each stack that requires them, removing the provider-consumer relationship. For example, the ShopSpinner team could combine the `application-infrastructure-stack` and `shared-network-stack` into a single stack.

Each instance of the application infrastructure has its own full set of networking structures. Doing this duplicates networking structures, but keeps each application instance independent of the others. As with distributed system architecture, this removes limitations on scaling. For example, the ShopSpinner team can add as many instances of the application infrastructure as they need without using up the address space allocated with a single shared networking stack.

But a more common driver for using a shared-nothing infrastructure code design is to make it easier to modify, rebuild, and recover the networking resources for an application stack. The shared networking stack design increases the blast radius and management overhead of working with the networking.

Shared-nothing infrastructure also supports the zero-trust security model (see "Zero-Trust Security Model with SDN" on page 31), since each stack can be secured separately.

A shared-nothing design doesn't require putting everything into a single stack instance. For the ShopSpinner team, an alternative to combining the networking and application infrastructure in a single stack is to define the networking and application infrastructure in different stacks, as before, but create a separate instance of the networking stack for each instance of the application stack (see Figure 15-5).

With this approach, application stack instances don't share networking with other stacks, but the two parts can be independently managed. The limitation of doing this is that all network stack instances are still defined by the same code, so any change to the code needs extra work to ensure it doesn't break any of the instances.

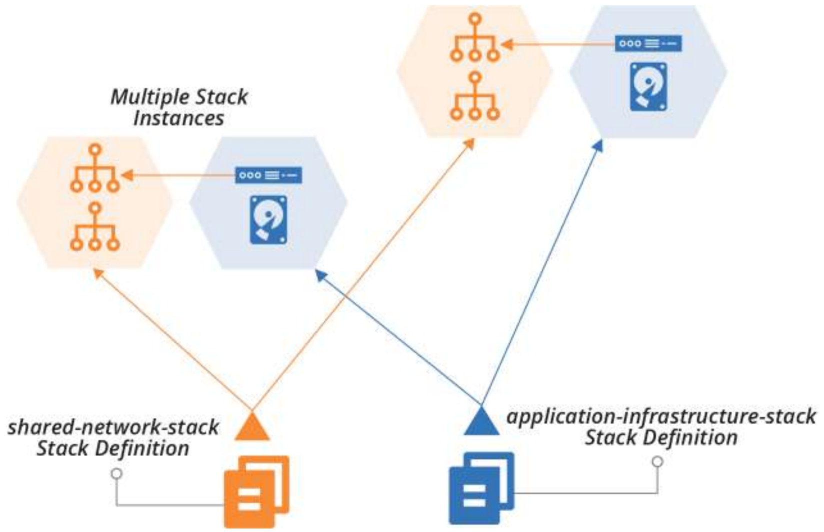

Multiple Stack Instances
shared-network-stack Stack Definition
application-infrastructure-stack Stack Definition

Figure 15-5. Multiple stacks with a shared-nothing deployment model

Drawing Boundaries Between Components

To divide infrastructure, as with any system, you should look for seams. A *seam* is a place where you can alter behavior in your system without editing in that place.[5] The idea is to find natural places to draw boundaries between parts of your systems, where you can create simple, clean integration points.

Each of the following strategies groups infrastructure elements based on a specific concern: patterns of change, organizational structures, security and governance, and resilience and scaling. These strategies, as with most architectural principles and practices, come down to optimizing for change. It's a quest to design components so that you can make changes to your system more easily, safely, and quickly.

Align Boundaries with Natural Change Patterns

The most basic approach to optimizing component boundaries for change is to understand their natural patterns of change. This is the idea behind finding seams—a seam is a natural boundary.

[5] Michael Feathers introduced the term *seam* in his book *Working Effectively with Legacy Code* (Addison-Wesley). More information is also available online.

With an existing system, you may learn about which things typically change together by examining historical changes. Finer-grained changes, like code commits, give the most useful insight. The most effective teams optimize for frequent commits, fully integrating and testing each one. By understanding what components tend to change together as part of a single commit, or closely related commits across components, you can find patterns that suggest how to refactor your code for more cohesion and less coupling.

Examining higher levels of work, such as tickets, stories, or projects, can help you to understand what parts of the system are often involved in a set of changes. But you should optimize for small, frequent changes. So be sure to drill down to understand what changes can be made independently of one another, to enable incremental changes within the context of larger change initiatives.

Align Boundaries with Component Life Cycles

Different parts of an infrastructure may have different life cycles. For example, servers in a cluster (see "Compute Resources" on page 28) are created and destroyed dynamically, perhaps many times a day. A database storage volume changes less frequently.

Organizing infrastructure resources into deployable components, particularly infrastructure stacks, according to their life cycle can simplify management. Consider a ShopSpinner application server infrastructure stack composed of networking routes, a server cluster, and a database instance.

The servers in this stack are updated at least every week, rebuilt using new server images with the latest operating system patches (as discussed in Chapter 13). The database storage device is rarely changed, although new instances may be built to recover or replicate instances of the application. The team occasionally changes networking in other stacks, which requires updating the application-specific routing in this stack.

Defining these elements in a single stack can create some risk. An update to the application server image may fail. Fixing the issue might require rebuilding the entire stack, including the database storage device, which in turn requires backing up the data to restore to the new instance (see "Data Continuity in a Changing System" on page 382). Although it's possible to manage this within a single stack, it would be simpler if the database storage was defined in a separate stack, as shown in Figure 15-6.

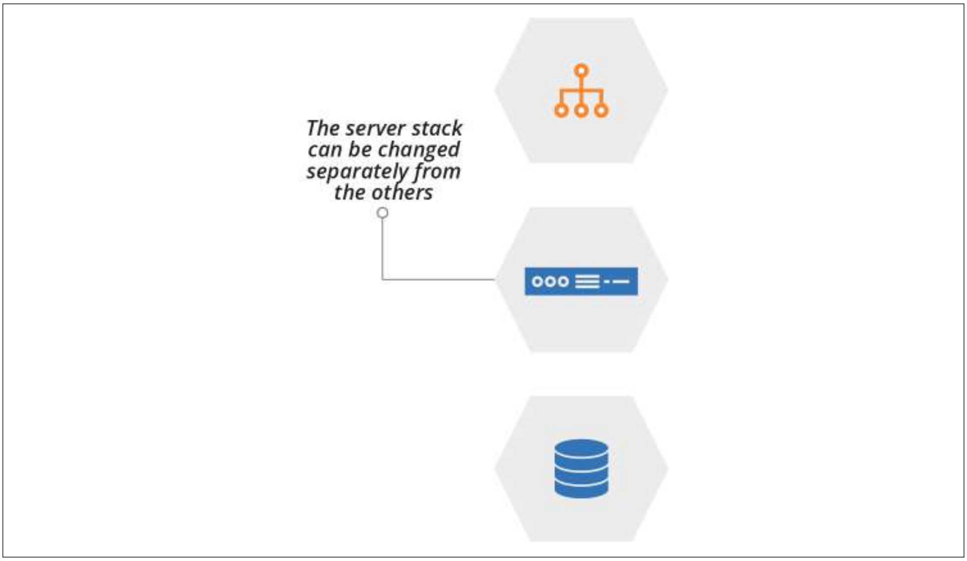


Figure 15-6. Different stacks have different life cycles

Changes are made to any of these micro stacks (see "Pattern: Micro Stack" on page 62) without directly affecting the others. This approach can enable stack-specific management events. For instance, any change to the database storage stack might trigger a data backup, which would probably be too expensive to trigger with every change to the other elements of the first, combined stack.

Optimizing stack boundaries for life cycles is particularly useful for automated testing in pipelines. Pipeline stages often run multiple times every day while people are working on infrastructure changes, so they need to be optimized to give fast feedback and keep a good working rhythm. Organizing infrastructure elements into separate stacks based on their life cycle can reduce the time taken to apply changes for testing.

For example, when working on the infrastructure code for the application servers, some pipeline stages might rebuild the stack each time (see "Pattern: Ephemeral Test Stack" on page 143). Rebuilding networking structures or large data storage devices can be slow, and may not be needed for many of the changes involved in the work. In this case, the micro-stack design shown earlier (Figure 15-6) can streamline the testing and delivery process.

A third use case for separating stacks by life cycle is cost management. Shutting down or destroying and rebuilding infrastructure that isn't needed in quiet periods is a common way to manage public cloud costs. But some elements, such as data storage, may be more challenging to rebuild. You can split these into their own stacks and leave them running while other stacks are destroyed to reduce costs.

Align Boundaries with Organizational Structures

Conway's Law says that systems tend to reflect the structure of the organization that creates it.[6] One team usually finds it easier to integrate software and infrastructure that it owns wholly, and the team will naturally create harder boundaries with parts of the system owned by other teams.

Conway's Law has two general implications for designing systems that include infrastructure. One is to avoid designing components that multiple teams need to make changes to. The other is to consider structuring teams to reflect the architectural boundaries you want, according to the "Inverse Conway Maneuver".

Legacy Silos Create Disjointed Infrastructure

Legacy organizational structures that put build and run into separate teams often create inconsistent infrastructure across the path to production, increasing the time, cost, and risk of delivering changes. At one organization I worked with, the application development team and operations team used different configuration tools to build servers. They wasted several weeks for every software release getting the applications to deploy and run correctly in the production environments.

For infrastructure, in particular, it's worth considering how to align design with the structure of the teams that use the infrastructure. In most organizations these are product or service lines and applications. Even with infrastructure used by multiple teams, such as a DBaaS service (see "Storage Resources" on page 29), you may design your infrastructure so you can manage separate instances for each team.

Aligning infrastructure instances with the teams that use them makes changes less disruptive. Rather than needing to negotiate a single change window with all of the teams using a shared instance, you can negotiate separate windows for each team.

Create Boundaries That Support Resilience

When something fails in your system, you can rebuild an independently deployable component like an infrastructure stack. You can repair or rebuild elements within a stack manually, conducting *infrastructure surgery*. Infrastructure surgery requires someone with a deep understanding of the infrastructure to carefully intervene. A simple mistake can make the situation far worse.

[6] The complete definition is, "Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure."

Some people take pride in infrastructure surgery, but it's a fallback to compensate for gaps in your infrastructure management systems.

An alternative to infrastructure surgery is rebuilding components using well-defined processes and tooling. You should be able to rebuild any stack instance by triggering the same automated process you use to apply changes and updates. If you can do this, you don't need to wake your most brilliant infrastructure surgeon when something fails in the middle of the night. In many cases, you can automatically trigger a recovery.

Infrastructure components need to be designed to make it possible to quickly rebuild and recover them. If you organize resources into components based on their life cycle ("Align Boundaries with Component Life Cycles" on page 263), then you can also take rebuild and recovery use cases into account.

The earlier example of splitting infrastructure that includes persistent data (Figure 15-6) does this. The process to rebuild the data storage should include steps that automatically save and load data, which will come in handy in a disaster recovery scenario. See "Data Continuity in a Changing System" on page 382 for more on this.

Dividing infrastructure into components based on their rebuild process helps to simplify and optimize recovery. Another approach to resilience is running multiple instances of parts of your infrastructure. Strategies for redundancy can also help with scaling.

Create Boundaries That Support Scaling

A common strategy for scaling systems is to create additional instances of some of its components. You may add instances in periods of higher demand, and you might also consider deploying instances in different geographical regions.

Most people are aware that most cloud platforms can automatically scale server clusters (see "Compute Resources" on page 28) up and down as load changes. A key benefit of FaaS serverless (see "Infrastructure for FaaS Serverless" on page 246) is that it only executes instances of code when needed.

However, other elements of your infrastructure, such as databases, message queues, and storage devices, can become bottlenecks when compute scales up. And different parts of your software system may become bottlenecks, even aside from the infrastructure.

For example, the ShopSpinner team can deploy multiple instances of the product browsing service stack to cope with higher load, because most user traffic hits that part of the system at peak times. The team keeps a single instance of its frontend traffic routing stack, and a single instance of the database stack that the application server instances connect to (see Figure 15-7).

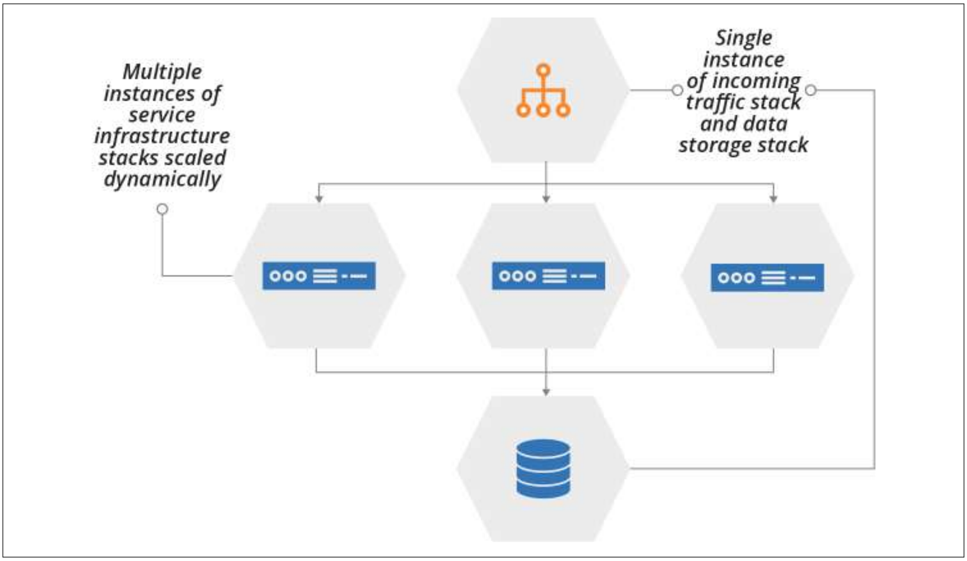

Multiple instances of service infrastructure stacks scaled dynamically
Single instance of incoming traffic stack and data storage stack

Figure 15-7. Scaling the number of instances of different stacks

Other parts of the system, such as order checkout and customer profile management services, probably don't need to scale together with the product browsing service. Splitting those services into different stacks helps the team to scale them more quickly. It reduces the waste that replicating everything would create.

Prefer Vertical Groupings Over Horizontal Groupings

Traditionally, many architects organized systems functionally. Networking stuff lives together, database stuff lives together, and operating system stuff lives together. This is often a result of organization design, as predicted by Conway's Law—when teams are organized around these technical, functional specialties, they will divide the infrastructure based on what they manage.

The pitfall with this approach is that a service provided to users cuts across many functions. This is often shown as a vertical service crossing horizontal functional layers, as shown in Figure 15-8.

Organizing system elements into cross-cutting, functional infrastructure stacks has two drawbacks. One is that a change to the infrastructure for one service may involve making changes to multiple stacks. These changes need to be carefully orchestrated to ensure a dependency isn't introduced in a consumer stack before it appears in the provider stack (see "Providers and Consumers" on page 255).

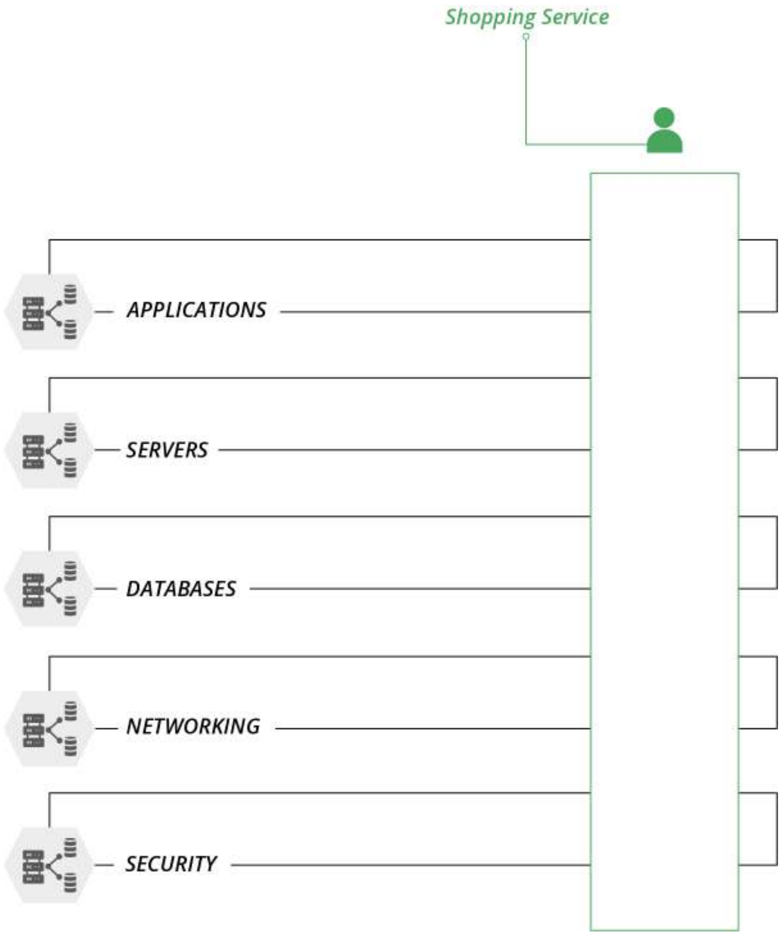


Figure 15-8. The infrastructure for each service is split across multiple stacks

Spreading ownership of infrastructure for a single service across multiple functional teams adds considerable communication overhead and process for changes to any one service.

The second drawback of functional stacks emerges when they are shared across services. In Figure 15-9, a server team manages a single stack instance with servers for multiple services.

When the team changes the server for one of the services, there's a risk of breaking other services, because a stack boundary represents the blast radius for a change, as described in "Antipattern: Monolithic Stack" on page 56.

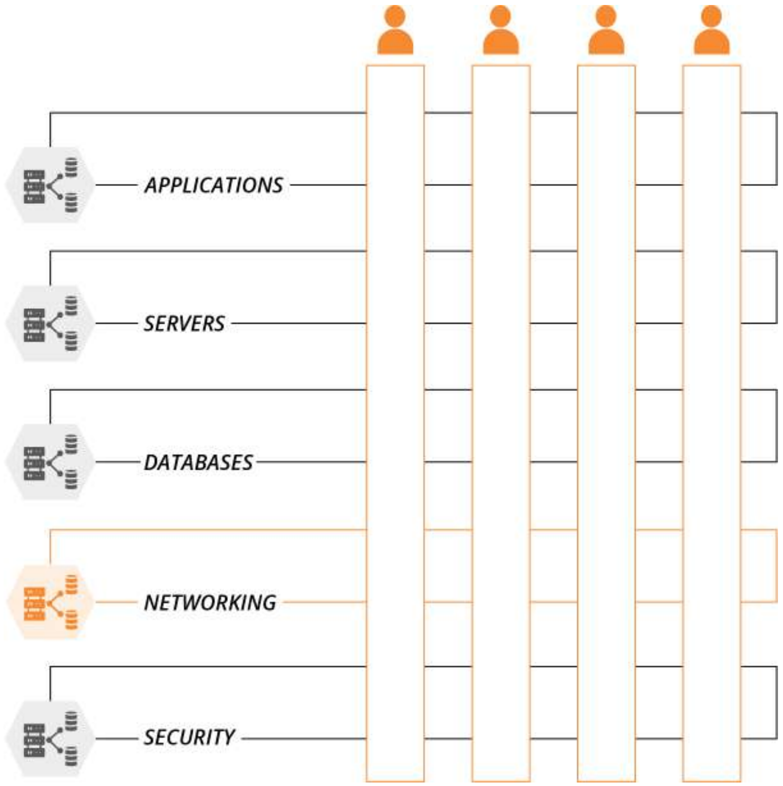


Figure 15-9. Changing a horizontal layer impacts multiple user-facing services

Align Boundaries to Security and Governance Concerns

Security, compliance, and governance protect data, transactions, and service availability. Different parts of a system will have different rules. For example, the PCI security standard imposes requirements on parts of a system that handle credit card numbers or payment processing. Personal data of customers and employees often needs to be handled with stricter controls.

Many organizations divide their infrastructure according to the regulations and policies that apply to the services and data it hosts. Doing this creates clarity for assessing what measures need to be taken with a given infrastructure component. The process for delivering changes can be tailored to the governance requirements. For example, the process can enforce and record reviews and approvals and generate change reports that simplify auditing.

Network Boundaries Are Not Infrastructure Stack Boundaries

People often divide infrastructure into networking security zones. Systems running in a frontend zone are directly accessible from the public internet, protected by firewalls and other mechanisms. Other zones, for application hosting and databases, for example, are only accessible from certain other zones, with additional layers of security.

While these boundaries are important for protecting against network-based attacks, they are usually not appropriate for organizing infrastructure code into deployable units. Putting the code for web servers and load balancers into a “frontend” stack doesn’t create a layer of defense against malicious changes to code for application servers or databases. The threat model for exploiting infrastructure code and tools is different from the threat model for network attacks.

By all means, do use your infrastructure code to create layered network boundaries.[7] But don’t assume that it’s a good idea to apply the networking security model to structuring infrastructure code.

Conclusion

This chapter started the conversation of how to manage larger, more complex infrastructure defined as code by breaking it into smaller pieces. It built on the previous content of the book, including the organizational units of infrastructure (stacks and servers), and the other core practices of defining things as code and continuously testing.

[7] Although you should consider a zero-trust model over pure perimeter-based security models, as mentioned in “Zero-Trust Security Model with SDN” on page 31.

CHAPTER 16

Building Stacks from Components

Chapter 15 explained how well-designed components can make an infrastructure system easier and safer to change. This message supports this book's theme of using speed of change to continuously improve the quality of a system, and using high quality to enable faster change.

This chapter focuses on modularizing infrastructure stacks; that is, breaking stacks into smaller pieces of code. There are several reasons to consider modularizing a stack:

Reuse
: Put knowledge of how to implement a particular construct into a component so you can reuse it across different stacks.

Composition
: Create the ability to swap different implementations of a concept, so you have flexibility in building your stacks.

Testability
: Improve the speed and focus of testing by breaking a stack into pieces that can be tested separately before integrating them. If a component is composable, you can replace them with test doubles ("Using Test Fixtures to Handle Dependencies" on page 137) to further improve the isolation and speed of testing.

Sharing
: Share composable, reusable, and well-tested components between teams, so people can build better systems more quickly.

As mentioned in "Stack Components Versus Stacks as Components" on page 257, breaking a stack into modules and libraries simplifies the code, but it doesn't make stack instances any smaller or simpler. Stack components have the potential to make things worse by obscuring the number and complexity of infrastructure resources they add to your stack instance.

So you should be sure you understand what lies beneath the abstractions, libraries, and platforms that you use. These things are a convenience that can let you focus on higher-level tasks. But they shouldn't be a substitute for fully understanding how your system is implemented.

Infrastructure Languages for Stack Components

Chapter 4 describes different types of infrastructure code languages. The two main styles of language for defining stacks are declarative (see "Declarative Infrastructure Languages" on page 41) and imperative (see "Programmable, Imperative Infrastructure Languages" on page 43). That chapter mentions that these different types of languages are suitable for different types of code (see "Declarative Versus Imperative Languages for Infrastructure" on page 44).

These differences often collide with stack components, when people write them using the wrong type of language. Using the wrong type of language usually leads to a mixture of both declarative and imperative languages, which, as explained previously, is a Bad Thing (see "Separate Declarative and Imperative Code" on page 46).

The decision of which language to use tends to be driven by which infrastructure stack management tool you're using, and the languages it supports.[1]

The patterns defined later in this chapter should encourage you to think about what you're trying to achieve with a particular stack and its components. To use this to consider the type of language, and potentially the type of stack tool, that you should use, consider two classes of stack component based on language type.

Reuse Declarative Code with Modules

Most stack management tools with declarative languages let you write shared components using the same language. CloudFormation has nested stacks and Terraform has modules. You can pass parameters to these modules, and the languages have at least some programmability (such as the HCL expressions sublanguage for Terraform). But the languages are fundamentally declarative, so nearly all complex logic written in them is barbaric.

So declarative code modules work best for defining infrastructure components that don't vary very much. A declarative module works well for a facade module (see "Pattern: Facade Module" on page 274), which wraps and simplifies a resource provided

[1] As I write this, in mid-2020, tool vendors are rapidly evolving their strategies around types of languages. I'm hopeful that the next few years will see a maturing of stack componentization.

by the infrastructure platform. These modules get nasty when you use them for more complex cases, creating spaghetti modules (see "Antipattern: Spaghetti Module" on page 280).

As mentioned in "Challenge: Tests for Declarative Code Often Have Low Value" on page 110, testing a declarative module should be fairly simple. The results of applying a declarative module don't vary very much, so you don't need comprehensive test coverage. This doesn't mean you shouldn't write tests for these modules. When a module combines multiple declarations to create a more complex entity, you should test that it fulfills its requirement.

Dynamically Create Stack Elements with Libraries

Some stack management tools, like Pulumi and the AWS CDK, use general-purpose, imperative languages. You can use these languages to write reusable libraries that you can call from your stack project code. A library can include more complex logic that dynamically provisions infrastructure resources according to how it's used.

For example, the ShopSpinner team's infrastructure includes different application server infrastructure stacks. Each of these stacks provisions an application server and networking structures for that application. Some of the applications are public facing, and others are internally facing.

In either case, the infrastructure stack needs to assign an IP address and DNS name to the server and create a networking route from the relevant gateway. The IP address and DNS name will be different for a public-facing application versus an internally facing one. And a public-facing application needs a firewall rule to allow the connection.

The checkout_service stack hosts a public-facing application:

```
application_networking = new ApplicationServerNetwork(PUBLIC_FACING, "checkout")

virtual_machine:
  name: appserver-checkout
  vlan: $(application_networking.address_block)
  ip_address: $(application_networking.private_ip_address)
```

The stack code creates an ApplicationServerNetwork object from the application_networking library, which provisions or references the necessary infrastructure elements:

```
class ApplicationServerNetwork {

  def vlan;
  def public_ip_address;
  def private_ip_address;
  def gateway;
  def dns_hostname;
```

```
    public ApplicationServerNetwork(application_access_type, hostname) {
      if (application_access_type == PUBLIC_FACING) {
        vlan = get_public_vlan()
        public_ip_address = allocate_public_ip()
        dns_hostname = PublicDNS.set_host_record(
          "${hostname}.shopspinners.xyz",
          this.public_ip_address
        )
      } else {
        // Similar stuff but for a private VLAN
      }

      private_ip_address = allocate_ip_from(this.vlan)
      gateway = get_gateway(this.vlan)
      create_route(gateway, this.private_ip_address)

      if (application_access_type == PUBLIC_FACING) {
        create_firewall_rule(ALLOW, '0.0.0.0', this.private_ip_address, 443)
      }
    }
  }
```

This pseudocode assigns the server to a public VLAN that already exists and sets its private IP address from the VLAN's address range. It also sets a public DNS entry for the server, which in our example will be *checkout.shopspinners.xyz*. The library finds the gateway based on the VLAN that was used, so this would be different for an internally facing application.

Patterns for Stack Components

The following set of patterns and antipatterns give ideas for designing stack components and evaluating existing components. It's not a complete list of ways you should or shouldn't build modules and libraries; rather, it's a starting point for thinking about the subject.

Pattern: Facade Module

Also known as: wrapper module.

A *facade module* creates a simplified interface to a resource from the stack tool language or the infrastructure platform. The module exposes a few parameters to the calling code (see Example 16-1).

Example 16-1. Example code using a facade module

```
use module: shopspinner-server
  name: checkout-appserver
  memory: 8GB
```

The module uses these parameters to call the resource it wraps, and hardcodes values for other parameters needed by the resource (Example 16-2).

Example 16-2. Code for the example facade module

```
declare module: shopspinner-server
  virtual_machine:
    name: ${name}
    source_image: hardened-linux-base
    memory: ${memory}
    provision:
      tool: servermaker
      maker_server: maker.shopspinner.xyz
      role: application_server
    network:
      vlan: application_zone_vlan
```

This example module allows the caller to create a virtual server, specifying the name and the amount of memory for the server. Every server created using the module uses a source image, role, and networking defined by the module.

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 that use it.

Applicability

Facade modules work best for simple use cases, usually involving a basic 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 better and more secure implementations. But it limits flexibility, so won’t apply to every use case.

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 hardcoded values, and a small number of values that are passed through from the code that uses the module. A declarative infrastructure language is appropriate for a facade module.

Related patterns

An obfuscation module is a facade module that doesn't hide much, adding complexity without adding much value. A bundle module ("Pattern: Bundle Module" on page 278) declares multiple related infrastructure resources, so is like a facade module with more parts.

Antipattern: Obfuscation Module

An *obfuscation module* wraps the code for an infrastructure element defined by the stack language or infrastructure platform, but does not simplify it or add any particular value. In the worst cases, the module complicates the code. See Example 16-3.

Example 16-3. Example code using an obfuscation module

```
use module: any_server
  server_name: checkout-appserver
  ram: 8GB
  source_image: base_linux_image
  provisioning_tool: servermaker
  server_role: application_server
  vlan: application_zone_vlan
```

The module itself passes the parameters directly to the stack management tool's code, as shown in Example 16-4.

Example 16-4. Code for the example obfuscation 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}
```

Motivation

An obfuscation module may be a facade module (see "Pattern: Facade Module" on page 274) gone wrong. Sometimes people write this kind of module aiming to follow the DRY principle (see "Avoid duplication" on page 253). 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.

Other people create obfuscation modules in a quest to design their own language for referring to infrastructure elements, "improving" the one provided by their stack tool.

Applicability

Nobody intentionally writes an obfuscation module. You may debate whether a given module obfuscates or is a facade, and that debate 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 adds more code to maintain, cognitive overhead to learn, and extra moving parts in your build and delivery process. A component should add enough value to make the overhead worthwhile.

Implementation

If a module neither simplifies the resources it defines nor adds value over the underlying stack language code, consider replacing usages by directly using the stack language.

Related patterns

An obfuscation module is similar to a facade module (see "Pattern: Facade Module" on page 274), but doesn't noticeably simplify the underlying code.

Antipattern: Unshared Module

An *unshared module* is only used once in a codebase, rather than being reused by multiple stacks.

Motivation

People usually create unshared modules as a way to organize the code within a stack project.

Applicability

As a stack project's code grows, you may be tempted to divide the code into modules. If you divide the code so that you can write tests for each module, this can make it easier to work with the code. Otherwise, there may be better ways to improve the codebase.

Consequences

Organizing a single stack's code into modules adds overhead to the codebase, probably including versioning and other moving parts. Building a reusable module when you don't need to reuse it is an example of *YAGNI* ("You Aren't Gonna Need It"), investing effort now for a benefit that you may or may not need in the future.

Implementation

When a stack project becomes too large, there are several alternatives to moving its code into modules. It's often better to split the stack into multiple stacks, using an appropriate stack structural pattern (see Chapter 17). If the stack is fairly cohesive (see "Characteristics of Well-Designed Components" on page 252), you could instead simply organize the code into different files and, if necessary, different folders. Doing this can make the code easier to navigate and understand without the overhead of the other options.

The *rule of three* for software reuse suggests that you should turn something into a reusable component when you find three places that you need to use it.[2]

Related patterns

An unshared module may map closely to lower-level infrastructure elements, like a facade module ("Pattern: Facade Module" on page 274), or to a higher-level entity, like an infrastructure domain entity ("Pattern: Infrastructure Domain Entity" on page 283).

Pattern: Bundle Module

A *bundle module* declares a collection of related infrastructure resources with a simplified interface. The stack code uses the module to define what it needs to provision:

[2] The rule of three was defined in Robert Glass's book, *Facts and Fallacies of Software Engineering* (Addison-Wesley). Jeff Atwood also commented on the rule of three in his post on the delusion of reuse.

```
  use module: application_server
    service_name: checkout_service
    application_name: checkout_application
    application_version: 1.23
    min_cluster: 1
    max_cluster: 3
    ram_required: 4GB
```

The module code declares multiple infrastructure resources, usually centered on a core resource. In Example 16-5, the resource is a server cluster, but also includes a load balancer and DNS entry.

Example 16-5. Module code for an application server

```
declare module: application_server

  server_cluster:
    id: "${service_name}-cluster"
    min_size: ${min_cluster}
    max_size: ${max_cluster}
    each_server_node:
      source_image: base_linux
      memory: ${ram_required}
      provision:
        tool: servermaker
        role: appserver
        parameters:
          app_package: "${checkout_application}-${application_version}.war"
          app_repository: "repository.shopspinner.xyz"

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

  dns_entry:
    id: "${service_name}-hostname"
    record_type: "A"
    hostname: "${service_name}.shopspinner.xyz"
    ip_address: {$load_balancer.ip_address}
```

Motivation

A bundle module is useful to define a cohesive collection of infrastructure resources. It avoids verbose, redundant code. These modules are useful to capture knowledge about the various elements needed and how to wire them together for a common purpose.

Applicability

A bundle module is suitable when you're working with a declarative stack language, and when the resources involved don't vary in different use cases. If you find that you need the module to create different resources or configure them differently depending on the usage, then you should either create separate modules, or else switch to an imperative language and create an infrastructure domain entity (see "Pattern: Infrastructure Domain Entity" on page 283).

Consequences

A bundle module may provision more resources than you need in some situations. Users of the module should understand what it provisions, and avoid using the module if it's overkill for their use case.

Implementation

Define the module declaratively, including infrastructure elements that are closely related to the declared purpose.

Related patterns

A facade module ("Pattern: Facade Module" on page 274) wraps a single infrastructure resource, while a bundle module includes multiple resources, although both are declarative in nature. An infrastructure domain entity ("Pattern: Infrastructure Domain Entity" on page 283) is similar to a bundle module, but dynamically generates infrastructure resources. A spaghetti module is a bundle module that wishes it was a domain entity but descends into madness thanks to the limitations of its declarative language.

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 (see Example 16-6).

Example 16-6. 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
        ...
    ...
```

This example code 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.

Motivation

As with other antipatterns, people create a spaghetti module by accident, often over time. You may create a facade module ("Pattern: Facade Module" on page 274) or a bundle module ("Pattern: Bundle Module" on page 278), that grows in complexity to handle divergent use cases that seem similar on the surface.

Spaghetti modules often result from trying to implement an infrastructure domain entity ("Pattern: Infrastructure Domain Entity" on page 283) using a declarative language.

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 Chapter 8, 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 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 Example 16-6, might look like Example 16-7.

Example 16-7. Example of using decomposed modules rather than a single spaghetti module

```
use module: java-application-servers
  name: checkout_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 test than the original spaghetti module.

Related patterns

A spaghetti module is often an attempt at building an infrastructure domain entity using declarative code. It could also be a facade module ("Pattern: Facade Module" on page 274) or a bundle module ("Pattern: Bundle Module" on page 278) that people tried to extend to handle different use cases.

Pattern: Infrastructure Domain Entity

A *infrastructure domain entity* implements a high-level stack component by combining multiple lower-level infrastructure resources. An example of a higher-level concept is the infrastructure needed to run an application.

This example shows how a library that implements a Java application infrastructure instance might be used from stack project code:

```
use module: application_server
  service_name: checkout_service
  application_name: checkout_application
  application_version: 1.23
  traffic_level: medium
```

The code defines the application and version to deploy, and also a traffic level. The domain entity library code could look similar to the bundle module example (Example 16-5), but includes dynamic code to provision resources according to the `traffic_level` parameter:

```
...
  switch (${traffic_level}) {
    case ("high") {
      $appserver_cluster.min_size = 3
      $appserver_cluster.max_size = 9
    } case ("medium") {
      $appserver_cluster.min_size = 2
      $appserver_cluster.max_size = 5
    } case ("low") {
      $appserver_cluster.min_size = 1
      $appserver_cluster.max_size = 2
    }
  }
...
```

Motivation

A domain entity is often part of an abstraction layer (see "Building an Abstraction Layer" on page 284) that people can use to define and build infrastructure based on higher-level requirements. An infrastructure platform team builds components that other teams can use to assemble stacks.

Applicability

Because an infrastructure domain entity dynamically provisions infrastructure resources, it should be written in an imperative language rather than a declarative one. See "Declarative Versus Imperative Languages for Infrastructure" on page 44 for more on why.

Implementation

On a concrete level, implementing an infrastructure domain entity is a matter of writing the code. 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 infrastructure domain entity pattern derives from *Domain Driven Design* (DDD), which creates a conceptual model for the business domain of a software system, and uses that to drive the design of the system itself.[3] Infrastructure, especially one designed and built as software, should 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.

Related patterns

A bundle module (see "Pattern: Bundle Module" on page 278) is similar to a domain entity in that it creates a cohesive collection of infrastructure resources. But a bundle module normally creates a fairly static set of resources, without much variation. The mindset of a bundle module is usually bottom-up, starting with the infrastructure resources to create. A domain entity is a top-down approach, starting with what's required for the use case.

Most spaghetti modules (see "Antipattern: Spaghetti Module" on page 280) for infrastructure stacks are a result of pushing declarative code to implement dynamic logic. But sometimes an infrastructure domain entity becomes overly complicated. A domain entity with poor cohesion becomes a spaghetti module.

Building an Abstraction Layer

An *abstraction layer* provides a simplified interface to lower-level resources. A set of reusable, composable stack components can act as an abstraction layer for infrastructure resources. Components can implement the knowledge of how to assemble

[3] See *Domain-Driven Design: Tackling Complexity in the Heart of Software*, by Eric Evans (Addison-Wesley).

low-level resources exposed by the infrastructure platform into entities that are useful for people focused on higher-level tasks.

For example, an application team may need to define an environment that includes an application server, database instance, and access to message queues. The team can use components that abstract the details of assembling rules for routing and infrastructure resource permissions.

Components can be useful even for a team that has the skills and experience to implement the low-level resources. An abstraction helps to separate different concerns, so that people can focus on the problem at a particular level of detail. They should also be able to drill down to understand and potentially improve or extend the underlying components as needed.

You might be able to implement an abstraction layer for some systems using more static components like facade modules (see "Pattern: Facade Module" on page 274) or bundle modules (see "Pattern: Bundle Module" on page 278). But more often you need the components of the layer to be more flexible, so dynamic components like infrastructure domain entities (see "Pattern: Infrastructure Domain Entity" on page 283) are more useful.

An abstraction layer might emerge organically as people build libraries and other components. But it's useful to have a higher-level design and standards so that the components of the layer work well together and fit into a cohesive view of the system.

The components of an abstraction layer are normally built using a low-level infrastructure language ("Low-Level Infrastructure Languages" on page 54). Many teams find it useful to build a higher-level language ("High-Level Infrastructure Languages" on page 55) for defining stacks with their abstraction layer. The result is often a higher-level, declarative language that specifies the requirements for part of the application runtime environment, which calls to dynamic components written in a low-level, imperative language.

Application Abstraction Models

The Open Application Model is an example of an attempt to define a standard architecture that decouples application, runtime, and infrastructure.

Conclusion

Building stacks from components can be useful when you have multiple people and teams working on and using infrastructure. But be wary of the complexity that comes with abstraction layers and libraries of components, and be sure to tailor your use of these constructs to match the size and complexity of your system.

CHAPTER 17

Using Stacks as Components

A stack is usually the highest-level component in an infrastructure system. It's the largest unit that can be defined, provisioned, and changed independently. The reusable stack pattern (see "Pattern: Reusable Stack" on page 72) encourages you to treat the stack as the main unit for sharing and reusing infrastructure.

Infrastructures composed of small stacks are more nimble than large stacks composed of modules and libraries. You can change a small stack more quickly, easily, and safely than a large stack. So this strategy supports the virtuous cycle of using speed of change to improve quality, and high quality to enable fast change.

Building a system from multiple stacks requires keeping each stack well-sized and well-designed, cohesive and loosely coupled. The advice in Chapter 15 is relevant to stacks as well as other types of infrastructure components. The specific challenge with stacks is implementing the integration between them without creating tight coupling.

Integration between stacks typically involves one stack managing a resource that another stack uses. There are many popular techniques to implement discovery and integration of resources between stacks, but many of them create tight coupling that makes changes more difficult. So this chapter explores different approaches from a viewpoint of how they affect coupling.

Discovering Dependencies Across Stacks

The ShopSpinner system includes a consumer stack, `application-infrastructure-stack`, that integrates with networking elements managed by another stack, `shared-network-stack`. The networking stack declares a VLAN:

```
vlan:
  name: "appserver_vlan"
  address_range: 10.2.0.0/8
```

The application stack defines an application server, which is assigned to the VLAN:

```
virtual_machine:
  name: "appserver-${ENVIRONMENT_NAME}"
  vlan: "appserver_vlan"
```

This example hardcodes the dependency between the two stacks, creating very tight coupling. It won't be possible to test changes to the application stack code without an instance of the networking stack. Changes to the networking stack are constrained by the other stack's dependence on the VLAN name.

If someone decides to add more VLANs for resiliency, they need consumers to change their code implementation in conjunction with the change to the consumer.[1] Otherwise, they can leave the original name, adding messiness that makes the code and infrastructure harder to understand and maintain:

```
vlans:
  - name: "appserver_vlan"
    address_range: 10.2.0.0/8
  - name: "appserver_vlan_2"
    address_range: 10.2.1.0/8
  - name: "appserver_vlan_3"
    address_range: 10.2.2.0/8
```

Hardcoding integration points can make it harder to maintain multiple infrastructure instances, as for different environments. This might work out depending on the infrastructure platform's API for the specific resources. For instance, perhaps you create infrastructure stack instances for each environment in a different cloud account, so you can use the same VLAN name in each. But more often, you need to integrate with different resource names for multiple environments.

So you should avoid hardcoding dependencies. Instead, consider using one of the following patterns for discovering dependencies.

Pattern: Resource Matching

A consumer stack uses *resource matching* to discover a dependency by looking for infrastructure resources that match names, tags, or other identifying characteristics. For example, a provider stack could name VLANs by the types of resources that belong in the VLAN and the environment of the VLAN (see Figure 17-1)

[1] See "Changing Live Infrastructure" on page 368 for less disruptive ways to manage this kind of change.

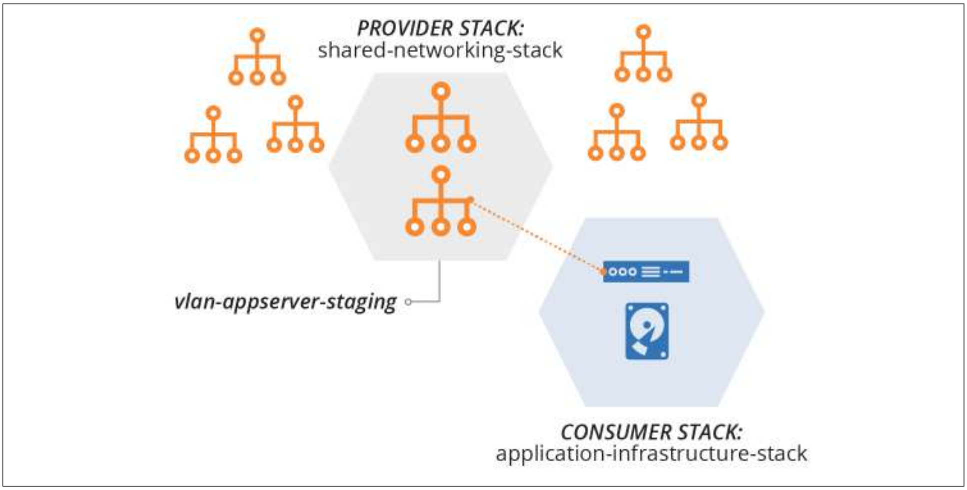


Figure 17-1. Resource matching for discovering dependencies

In this example, `vlan-appserver-staging` is intended for application servers in the staging environment. The `application-infrastructure-stack` code finds this resource by matching the naming pattern:

```
virtual_machine:
  name: "appserver-${ENVIRONMENT_NAME}"
  vlan: "vlan-appserver-${ENVIRONMENT_NAME}"
```

A value for `ENVIRONMENT_NAME` is passed to the stack management tool when applying the code, as described in Chapter 7.

Motivation

Resource matching is straightforward to implement with most stack management tools and languages. The pattern mostly eliminates hardcoded dependencies, reducing coupling.

Resource matching also avoids coupling on tools. The provider infrastructure and consumer stack can be implemented using different tools.

Applicability

Use resource matching for discovering dependencies when the teams managing provider and consumer code both have a clear understanding of which resources should be used as dependencies. Consider switching to an alternative pattern if you experience issues with breaking dependencies between teams.

Resource matching is useful in larger organizations, or across organizations, where different teams may use different tooling to manage their infrastructure, but still need to integrate at the infrastructure level. Even where everyone currently uses a single tool, resource matching reduces lock-in to that tool, creating the option to use new tools for different parts of the system.

Consequences

As soon as a consumer stack implements resource matching to discover a resource from another stack, the matching pattern becomes a contract. If someone changes the naming pattern of the VLAN in the shared networking stack, the consumer's dependency breaks.

So a consumer team should only discover dependencies by matching resources in ways that the provider team explicitly supports. Provider teams should clearly communicate what resource matching patterns they support, and be sure to maintain the integrity of those patterns as a contract.

Implementation

There are several ways to discover infrastructure resources by matching. The most straightforward method is to use variables in the name of the resource, as shown in the example code from earlier:

```
virtual_machine:
  name: "appserver-${ENVIRONMENT_NAME}"
  vlan: "vlan-appserver-${ENVIRONMENT_NAME}"
```

The string `vlan-appserver-${ENVIRONMENT_NAME}` will match the relevant VLAN for the environment.

Most stack languages have features to match other attributes than the resource name. Terraform has data sources and AWS CDK's supports resource importing.

In this example (using pseudocode) the provider assigns tags to its VLANs:

```
vlans:
  - appserver_vlan
    address_range: 10.2.0.0/8
    tags:
      network_tier: "application_servers"
      environment: ${ENVIRONMENT_NAME}
```

The consumer code discovers the VLAN it needs using those tags:

```
external_resource:
  id: appserver_vlan
  match:
    tag: name == "network_tier" && value == "application_servers"
    tag: name == "environment" && value == ${ENVIRONMENT_NAME}

virtual_machine:
  name: "appserver-${ENVIRONMENT_NAME}"
  vlan: external_resource.appserver_vlan
```

Related patterns

The resource matching pattern is similar to the stack data lookup pattern. The main difference is that resource matching doesn't depend on the implementation of the same stack tool across provider and consumer stacks.

Pattern: Stack Data Lookup

Also known as: remote statefile lookup, stack reference lookup, or stack resource lookup.

Stack data lookup finds provider resources using data structures maintained by the tool that manages the provider stack (Figure 17-2).

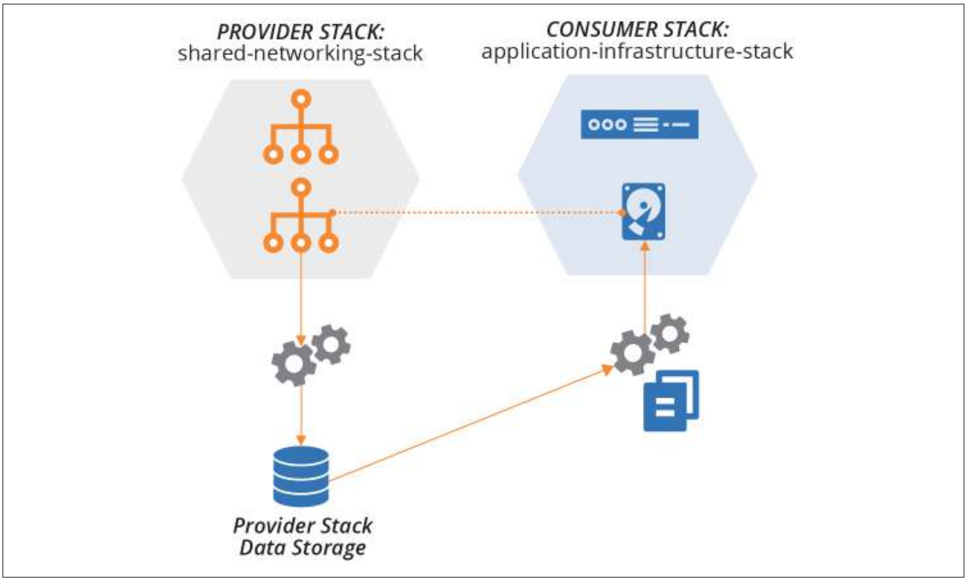


Figure 17-2. Stack data lookup for discovering dependencies

Many stack management tools maintain data structures for each stack instance, which include values exported by the stack code. Examples include Terraform and Pulumi remote state files.

Motivation

Stack management tool vendors make it easy to use their stack data lookup features to integrate different projects. Most implementations of data sharing across stacks require the provider stack to explicitly declare which resources to publish for use by other stacks. Doing this discourages consumer stacks from creating dependencies on resources without the provider's knowledge.

Applicability

Using stack data lookup functionality to discover dependencies across stacks works when all of the infrastructure in a system is managed using the same tool.

Consequences

Stack data lookup tends to lock you into a single stack management tool. It's possible to use the pattern with different tools, as described in the implementation of this pattern. But this adds complexity to your implementation.

This pattern sometimes breaks across different versions of the same stack tool. An upgrade to the tool may involve changing the stack data structures. This can cause problems when upgrading a provider stack to the newer version of the tool. Until you upgrade the consumer stack to the new version of the tool as well, it may not be possible for the older version of the tool to extract the resource values from the upgraded provider stack. This stops you from rolling out stack tool upgrades incrementally across stacks, potentially forcing a disruptive coordinated upgrade across your estate.

Implementation

The implementation of stack data lookup uses functionality from your stack management tool and its definition language.

Terraform stores output values in a remote state file. Pulumi also stores resource details in a state file that can be referenced in a consumer stack using a StackReference. CloudFormation can export and import stack output values across stacks, which AWS CDK can also access.[2]

The provider usually explicitly declares the resources it provides to consumers:

```
stack:
  name: shared_network_stack
  environment: ${ENVIRONMENT_NAME}

vlans:
  - appserver_vlan
    address_range: 10.2.0.0/8

export:
  - appserver_vlan_id: appserver_vlan.id
```

[2] See the AWS CDK Developer Guide.

The consumer declares a reference to the provider stack and uses this to refer to the VLAN identifier exported by that stack:

```
external_stack:
  name: shared_network_stack
  environment: ${ENVIRONMENT_NAME}

virtual_machine:
  name: "appserver-${ENVIRONMENT_NAME}"
  vlan: external_stack.shared_network_stack.appserver_vlan.id
```

This example embeds the reference to the external stack in the stack code. Another option is to use dependency injection ("Dependency Injection" on page 296) so that your stack code is less coupled to the dependency discovery. Your orchestration script looks up the output value from the provider stack and passes it to your stack code as a parameter.

Although stack data lookups are tied to the tool that manages the provider stack, you can usually extract these values in a script, so you can use them with other tools, as shown in Example 17-1.

Example 17-1. Using a stack tool to discover a resource ID from a stack instance's data structure

```
#!/usr/bin/env bash

VLAN_ID=$(
  stack value \
    --stack_instance shared_network-staging \
    --export_name appserver_vlan_id
)
```

This code runs the fictional `stack` command, specifying the stack instance (`shared_network-staging`) to look in, and the exported variable to read and print (`appserver_vlan_id`). The shell command stores the command's output, which is the ID of the VLAN, in a shell variable named `VLAN_ID`. The script can then use this variable in different ways.

Related patterns

The main alternative patterns are resource matching ("Pattern: Resource Matching" on page 288) and registry lookup.

Pattern: Integration Registry Lookup

Also known as: integration registry.

A consumer stack can use *integration registry lookup* to discover a resource published by a provider stack (Figure 17-3). Both stacks refer to a registry, using a known location to store and read values.

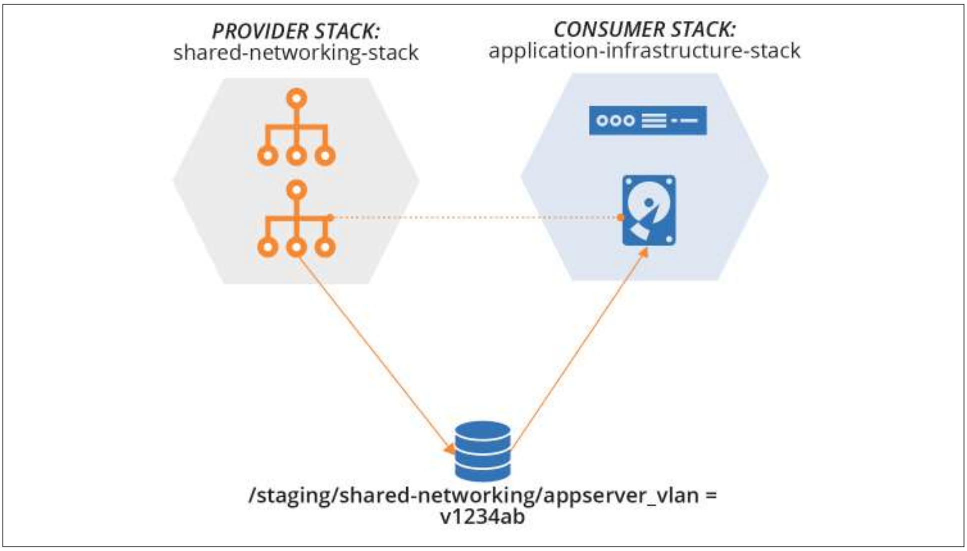

PROVIDER STACK:
shared-networking-stack
CONSUMER STACK:
application-infrastructure-stack
/staging/shared-networking/appserver_vlan = v1234ab

Figure 17-3. Integration registry lookup for discovering dependencies

Many stack tools support storing and retrieving values from different types of registries within definition code. The `shared-networking-stack` code sets the value:

```
vlans:
  - appserver_vlan
    address_range: 10.2.0.0/8

registry:
  host: registry.shopspinner.xyz
  set:
    /${ENVIRONMENT_NAME}/shared-networking/appserver_vlan: appserver_vlan.id
```

The `application-infrastructure-stack` code then retrieves and uses the value:

```
registry:
  id: stack_registry
  host: registry.shopspinner.xyz
  values:
    appserver_vlan_id: /${ENVIRONMENT_NAME}/shared-networking/appserver_vlan

virtual_machine:
  name: "appserver-${ENVIRONMENT_NAME}"
  vlan: stack_registry.appserver_vlan_id
```

Motivation

Using a configuration registry decouples the stack management tools for different infrastructure stacks. Different teams can use different tools as long as they agree to use the same configuration registry service and naming conventions for storing values in it. This decoupling also makes it easier to upgrade and change tools incrementally, one stack at a time.

Using a configuration registry makes the integration points between stacks explicit. A consumer stack can only use values explicitly published by a provider stack, so the provider team can freely change how they implement their resources.

Applicability

The integration registry lookup pattern is useful for larger organizations, where different teams may use different technologies. It's also useful for organizations concerned about lock-in to a tool.

If your system already uses a configuration registry ("Configuration Registry" on page 99), for instance, to provide configuration values to stack instances following the stack parameter registry pattern ("Pattern: Stack Parameter Registry" on page 96), it can make sense to use the same registry for integrating stacks.

Consequences

The configuration registry becomes a critical service when you adopt the integration registry lookup pattern. You may not be able to provision or recover resources when the registry is unavailable.

Implementation

The configuration registry is a prerequisite for using this pattern. See "Configuration Registry" on page 99 in Chapter 7 for a discussion of registries. Some infrastructure tool vendors provide registry servers, as mentioned in "Infrastructure automation tool registries" on page 100. With any registry product, be sure it's well-supported by the tools you use, and those you might consider using in the future.

It's essential to establish a clear convention for naming parameters, especially when using the registry to integrate infrastructure across multiple teams. Many organizations use a hierarchical namespace, similar to a directory or folder structure, even when the registry product implements a simple key/value mechanism. The structure typically includes components for architectural units (such as services, applications, or products), environments, geography, or teams.

For example, ShopSpinner could use a hierarchical path based on the geographical region:

```
/infrastructure/
 ├── au/
 │   ├── shared-networking/
 │   │   └── appserver_vlan=
 │   └── application-infrastructure/
 │       └── appserver_ip_address=
 └── eu/
     ├── shared-networking/
     │   └── appserver_vlan=
     └── application-infrastructure/
         └── appserver_ip_address=
```

The IP address of the application server for the European region, in this example, is found at the location */infrastructure/eu/application-infrastructure/appserver_ip_address*.

Related patterns

Like this pattern, the stack data lookup pattern (see "Pattern: Stack Data Lookup" on page 291) stores and retrieves values from a registry. That pattern uses data structures specific to the stack management tool, whereas this pattern uses a general-purpose registry implementation. The parameter registry pattern (see "Pattern: Stack Parameter Registry" on page 96) is essentially the same as this pattern, in that a stack pulls values from a registry to use in a given stack instance. The only difference is that with this pattern, the value comes from another stack, and is used explicitly to integrate infrastructure resources between stacks.

Dependency Injection

These patterns describe strategies for a consumer to discover resources managed by a provider stack. Most stack management tools support directly using these patterns in your stack definition code. But there's an argument for separating the code that defines a stack's resources from code that discovers resources to integrate with.

Consider this snippet from the earlier implementation example for the dependency matching pattern (see "Pattern: Resource Matching" on page 288):

```
external_resource:
  id: appserver_vlan
  match:
    tag: name == "network_tier" && value == "application_servers"
    tag: name == "environment" && value == ${ENVIRONMENT_NAME}

virtual_machine:
  name: "appserver-${ENVIRONMENT_NAME}"
  vlan: external_resource.appserver_vlan
```

The essential part of this code is the declaration of the virtual machine. Everything else in the snippet is peripheral, implementation details for assembling configuration values for the virtual machine.

Issues with mixing dependency and definition code

Combining dependency discovery with stack definition code adds cognitive overhead to reading or working with the code. Although it doesn't stop you from getting things done, the subtle friction adds up.

You can remove the cognitive overhead by splitting the code into separate files in your stack project. But another issue with including discovery code in a stack definition project is that it couples the stack to the dependency mechanism.

The type and depth of coupling to a dependency mechanism and the kind of impact that coupling has will vary for different mechanisms and how you implement them. You should avoid or minimize coupling with the provider stack, and with services like a configuration registry.

Coupling dependency management and definition can make it hard to create and test instances of the stack. Many approaches to testing use practices like ephemeral instances ("Pattern: Ephemeral Test Stack" on page 143) or test doubles ("Using Test Fixtures to Handle Dependencies" on page 137) to enable fast and frequent testing. This can be challenging if setting up dependencies involves too much work or time.

Hardcoding specific assumptions about dependency discovery into stack code can make it less reusable. For example, if you create a core application server infrastructure stack for other teams to use, different teams may want to use different methods to configure and manage their dependencies. Some teams may even want to swap out different provider stacks. For example, they might use different networking stacks for public-facing and internally facing applications.

Decoupling dependencies from their discovery

Dependency injection (DI) is a technique where a component receives its dependencies, rather than discovering them itself. An infrastructure stack project would declare the resources it depends on as parameters, the same as instance configuration parameters described in Chapter 7. A script or other tool that orchestrates the stack management tool ("Using Scripts to Wrap Infrastructure Tools" on page 335) would be responsible for discovering dependencies and passing them to the stack.

Consider how the `application-infrastructure-stack` example used to illustrate the dependency discovery patterns earlier in this chapter would look with DI:

```
parameters:
  - ENVIRONMENT_NAME
  - VLAN
```

```
virtual_machine:
  name: "appserver-${ENVIRONMENT_NAME}"
  vlan: ${VLAN}
```

The code declares two parameters that must be set when applying the code to an instance. The `ENVIRONMENT_NAME` parameter is a simple stack parameter, used for naming the application server virtual machine. The `VLAN` parameter is the identifier of the VLAN to assign the virtual machine to.

To manage an instance of this stack, you need to discover and provide a value for the `VLAN` parameter. Your orchestration script can do this, using any of the patterns described in this chapter. It might set the parameter based on tags, by finding the provider stack project's outputs using the stack management tool, or it could look up the values in a registry.

An example script using stack data lookup (see "Pattern: Stack Data Lookup" on page 291) might use the stack tool to retrieve the VLAN ID from the provider stack instance, as in Example 17-1, and then pass that value to the stack command for the consumer stack instance:

```
#!/usr/bin/env bash

ENVIRONMENT_NAME=$1

VLAN_ID=$(
  stack value \
    --stack_instance shared_network-${ENVIRONMENT_NAME} \
    --export_name appserver_vlan_id
)

stack apply \
  --stack_instance application_infrastructure-${ENVIRONMENT_NAME} \
  --parameter application_server_vlan=${VLAN_ID}
```

The first command extracts the `appserver_vlan_id` value from the provider stack instance named `shared_network-${ENVIRONMENT_NAME}`, and then passes it as a parameter to the consumer stack `application_infrastructure-${ENVIRONMENT_NAME}`.

The benefit of this approach is that the stack definition code is simpler and can be used in different contexts. When you work on changes to the stack code on your laptop, you can pass whatever VLAN value you like. You can apply your code using a local API mock (see "Testing with a Mock API" on page 133), or to a personal instance on your infrastructure platform (see "Personal Infrastructure Instances" on page 347). The VLANs you provide in these situations may be very simple.

In more production-like environments, the VLAN may be part of a more comprehensive network stack. This ability to swap out different provider implementations makes it easier to implement progressive testing (see "Progressive Testing" on page 115), where earlier pipeline stages run quickly and test the consumer component in isolation, and later stages test a more comprehensively integrated system.

Origins of Dependency Injection

DI originated in the world of object-oriented (OO) software design in the early 2000s. Proponents of XP found that unit testing and TDD were much easier with software written for DI. Java frameworks like PicoContainer and the Spring framework pioneered DI. Martin Fowler's 2004 article "Inversion of Control Containers and the Dependency Injection Pattern" explains the rationale for the approach for OO software design.

Conclusion

Composing infrastructure from well-designed, well-sized stacks that are loosely coupled makes it easier, faster, and safer to make changes to your system. Doing this requires following the general design guidance for modularizing infrastructure (as in Chapter 15). It also requires making sure that stacks don't couple themselves too closely together when sharing and providing resources.

PART V

Delivering Infrastructure

CHAPTER 18

Organizing Infrastructure Code

An infrastructure codebase may include various types of code, including stack definitions, server configurations, modules, libraries, tests, configuration, and utilities.

How should you organize this code across and within projects? How should you organize projects across repositories? Do infrastructure and application code belong together, or should they be separated? How should you organize code for an estate with multiple parts?

Organizing Projects and Repositories

In this context, a *project* is a collection of code used to build a discrete component of the system. There is no hard rule on how much a single project or its component can include. "Patterns and Antipatterns for Structuring Stacks" on page 56 describes different levels of scope for an infrastructure stack, for instance.

A project may depend on other projects in the codebase. Ideally, these dependencies and the boundaries between projects are well-defined, and clearly reflected in the way project code is organized.

Conway's Law (see "Align Boundaries with Organizational Structures" on page 265) says that there is a direct relationship between the structure of the organization and the systems that it builds. Poor alignment of team structures and ownership of systems, and the code that defines those systems, creates friction and inefficiency.

The flip side of drawing boundaries between projects is integrating projects when there are dependencies between them, as described for stacks in Chapter 17.

See "Integrating Projects" on page 326 for a discussion of how and when different dependencies may be integrated with a project.

There are two dimensions to the problem of how to organize code. One is where to put different types of code—code for stacks, server configuration, server images, configuration, tests, delivery tooling, and applications. The other is how to arrange projects across source code repositories. This last question is a bit simpler, so let's start there.

One Repository, or Many?

Given that you have multiple code projects, should you put them all in a single repository in your source control system, or spread them among more than one? If you use more than one repository, should every project have its own repository, or should you group some projects together into shared repositories? If you arrange multiple projects into repositories, how should you decide which ones to group and which ones to separate?

There are some trade-off factors to consider:

- Separating projects into different repositories makes it easier to maintain boundaries at the code level.
- Having multiple teams working on code in a single repository can add overhead and create conflicts.
- Spreading code across multiple repositories can complicate working on changes that cross them.
- Code kept in the same repository is versioned and can be branched together, which simplifies some project integration and delivery strategies.
- Different source code management systems (such as Git, Perforce, and Mercurial) have different performance and scalability characteristics and features to support complex scenarios.

Let's look at the main options for organizing projects across repositories in the light of these factors.

One Repository for Everything

Some teams, and even some larger organizations, maintain a single repository with all of their code. This requires source control system software that can scale to your usage level. Some software struggles to handle a codebase as it grows in size, history, number of users, and activity level.[1] So splitting repositories becomes a matter of managing performance.

A single repository can be easier to use. People can check out all of the projects they need to work on, guaranteeing they have a consistent version of everything. Some version control software offers features, like sparse-checkout, which let a user work with a subset of the repository.

Monorepo—One Repository, One Build

A single repository works well with build-time integration (see "Pattern: Build-Time Project Integration" on page 327). The monorepo strategy uses the build-time integration pattern for projects maintained in a single repository. A simplistic version of monorepo builds all of the projects in the repository, as shown in Figure 18-1.

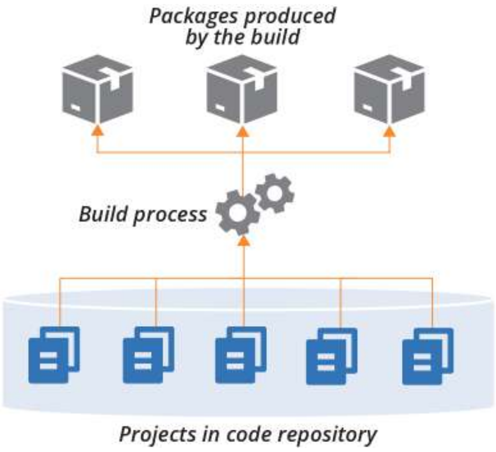


Figure 18-1. Building all projects in a repository together

Although the projects are built together, they may produce multiple artifacts, such as application packages, infrastructure stacks, and server images.

[1] Facebook, Google, and Microsoft all use very large repositories. All three have either made custom changes to their version control software or built their own. See "Scaling version control software" for more. Also see "Scaled trunk-based development" by Paul Hammant for insight on this history of Google's approach.

One repository, multiple builds

Most organizations that keep all of their projects in a single repository don't necessarily run a single build across them all. They often have a few different builds to build different subsets of their system (see Figure 18-2).

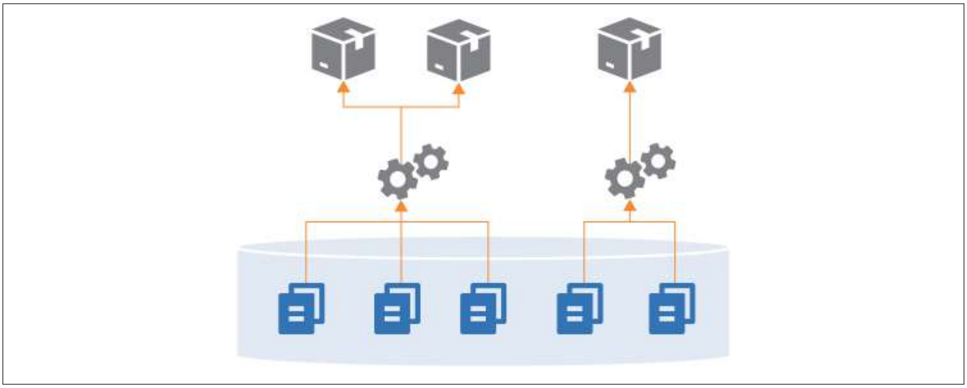


Figure 18-2. Building different combinations of projects from one repository

Often, these builds will share some projects. For instance, two different builds may use the same shared library (see Figure 18-3).

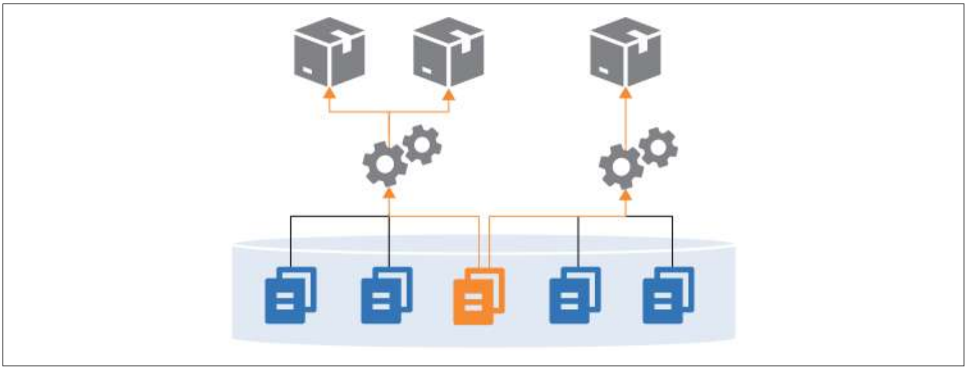


Figure 18-3. Sharing a component across builds in a single repository

One pitfall of managing multiple projects is that it can blur the boundaries between projects. People may write code for one project that refers directly to files in another project in the repository. Doing this leads to tighter coupling and less visibility of dependencies. Over time, projects become tangled and hard to maintain, because a change to a file in one project can have unexpected conflicts with other projects.

A Separate Repository for Each Project (Microrepo)

Having a separate repository for each project is the other extreme (Figure 18-4).

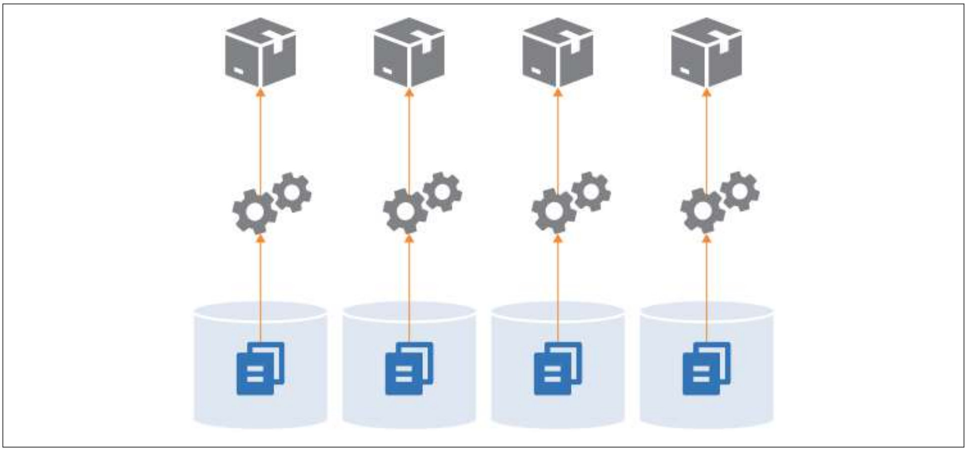


Figure 18-4. Each project in a separate repository

This strategy ensures a clean separation between projects, especially when you have a pipeline that builds and tests each project separately before integrating them. If someone checks out two projects and makes a change to files across projects, the pipeline will fail, exposing the problem.

Technically, you could use build-time integration across projects managed in separate repositories, by first checking out all of the builds (see Figure 18-5).

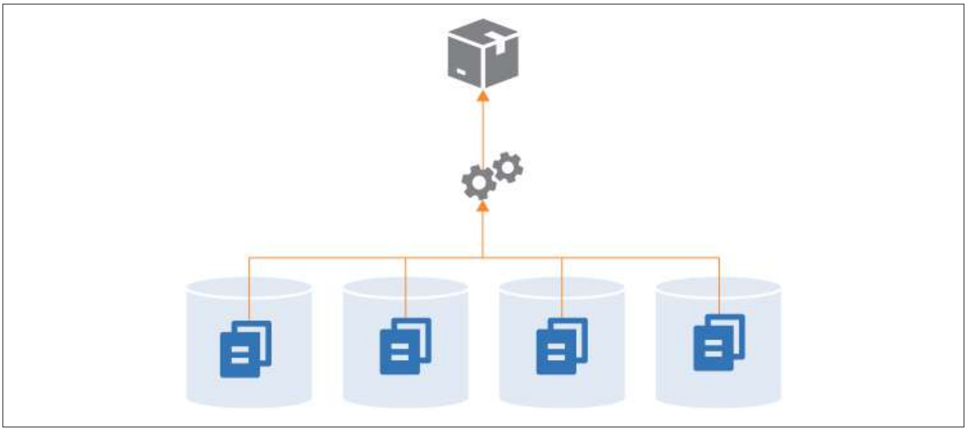


Figure 18-5. A single build across multiple repositories

In practice, it's more practical to build across multiple projects in a single repository, because their code is versioned together. Pushing changes for a single build to multiple repositories complicates the delivery process. The delivery stage would need some way to know which versions of all of the involved repositories to check out to create a consistent build.

Single-project repositories work best when supporting delivery-time and apply-time integration. A change to any one repository triggers the delivery process for its project, bringing it together with other projects later in the flow.

Multiple Repositories with Multiple Projects

While some organizations push toward one extreme or the other—single repository for everything, or a separate repository for each project—most maintain multiple repositories with more than one project (see Figure 18-6).

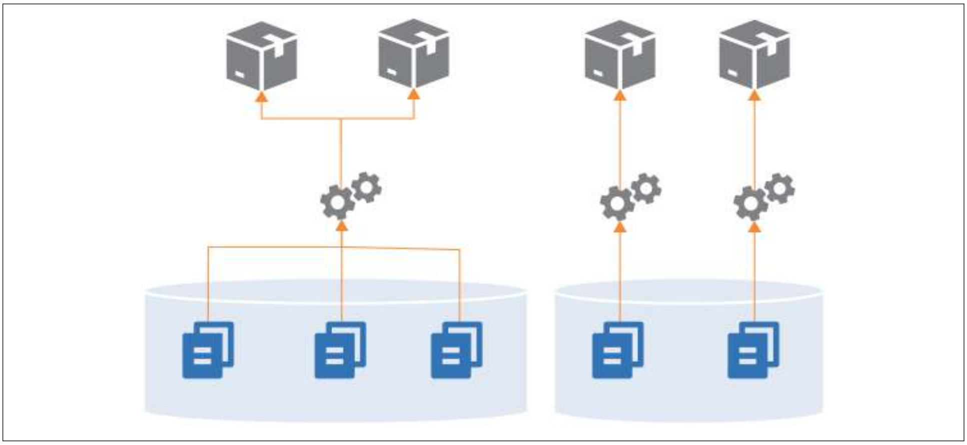


Figure 18-6. Multiple repositories with multiple projects

Often, the grouping of projects into repositories happens organically, rather than being driven by a strategy like monorepo or microrepo. However, there are a few factors that influence how smoothly things work.

One factor, as seen in the discussions of the other repository strategies, is the alignment of a project grouping with its build and delivery strategy. Keep projects in a single repository when they are closely related, and especially when you integrate the projects at build time. Consider separating projects into separate repositories when their delivery paths aren't tightly integrated.

Another factor is team ownership. Although multiple people and teams can work on different projects in the same repository, it can be distracting. Changelogs intermingle commit history from different teams with unrelated workstreams. Some organizations restrict access to code. Access control for source control systems is often managed by repository, which is another driver for deciding which projects go where.

As mentioned for single repositories, projects within a repository more easily become tangled together with file dependencies. So teams might divide projects between repositories based on where they need stronger boundaries from an architectural and design perspective.

Organizing Different Types of Code

Different projects in an infrastructure codebase define different types of elements of your system, such as applications, infrastructure stacks, server configuration modules, and libraries. And these projects may include different types of code, including declarations, imperative code, configuration values, tests, and utility scripts. Having a strategy for organizing these things helps to keep your codebase maintainable.

Project Support Files

Generally speaking, any supporting code for a specific project should live with that project's code. A typical project layout for a stack might look like Example 18-1.

Example 18-1. Example folder layout for a project

```
├── build.sh
├── src/
├── test/
├── environments/
└── pipeline/
```

This project's folder structure includes:

`src/`
: The infrastructure stack code, which is the heart of the project.

`test/`
: Test code. This folder can be divided into subfolders for tests that run at different phases, such as offline and online tests. Tests that use different tools, like static analysis, performance tests, and functional tests, probably have dedicated subfolders as well.

`environments/`
: Configuration. This folder includes a separate file with configuration values for each stack instance.

`pipeline/`
: Delivery configuration. The folder contains configuration files to create delivery stages in a delivery pipeline tool (see "Delivery Pipeline Software and Services" on page 123).

`build.sh/`
: Script to implement build activities. See "Using Scripts to Wrap Infrastructure Tools" on page 335 for a discussion of scripts like this one.

Of course, this is only an example. People organize their projects differently and include many other things than what's shown here.

The key takeaway is the recommendation that files specific to a project live with the project. This ensures that when someone checks out a version of the project, they know that the infrastructure code, tests, and delivery are all the same version, and so should work together. If the tests are stored in a separate project it would be easy to mismatch them, running the wrong version of the tests for the code you're testing.

However, some tests, configuration, or other files might not be specific to a single project. How should you handle these?

Cross-Project Tests

Progressive testing (see "Progressive Testing" on page 115) involves testing each project separately before testing it integrated with other projects, as shown in Figure 18-7.

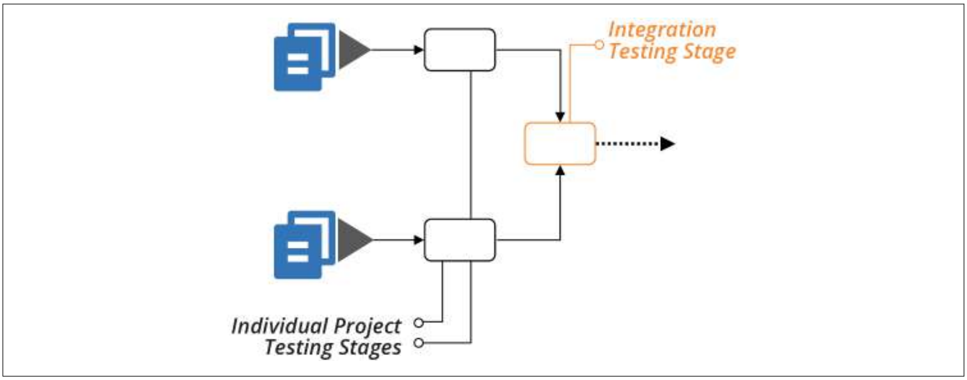


Figure 18-7. Testing projects separately, then together

You can comfortably put the test code to run in the individual stages for each project with that project. But what about the test code for the integration stage? You can put these tests in one of the projects, or create a separate project for the integration tests.

Keeping integration tests within a project

In many cases where you integrate multiple projects, one project is an obvious entry point for certain types of tests. For example, many functional tests connect to a frontend service to prove that the entire system works. If a backend component, such as a database, isn't configured correctly, the frontend service can't connect to it, so the test fails.

In these cases, the integration test code lives comfortably with the project that provisions the frontend service. Most likely, the test code is coupled to that service. For example, it needs to know the hostname and port for that service.

Separate these tests from tests that run in earlier delivery stages—for example, when testing with test doubles. You can keep each set of tests in a separate subfolder in the project.

Integration tests also fit well in projects that consume other projects (see "Using Test Fixtures to Handle Dependencies" on page 137), rather than in the provider project. The ShopSpinner example includes one stack project that defines application infrastructure instances, sharing network structures defined in a different stack.

Putting integration tests into the shared network stack project goes against the direction of the dependency. The network project needs to know specific details of the application stack, and any other stacks that use it, to test that integration works correctly. The application infrastructure stack already knows about the shared network stack, so keeping the integration tests with the application stack code avoids dependency loops between the projects.

Dedicated Integration Test Projects

An alternative approach is to create a separate project for integration tests, perhaps one for each integration stage. This approach is common when a different team owns the integration tests, as predicted by Conway's Law. Other teams do this when it's not clear which project aligns with the integration tests.

Versioning can be challenging when managing integration test suites separately from the code they test. People may confuse which version of the integration tests to run for a given version of the system code. To mitigate this, be sure to write and change tests along with the code, rather than separating those activities. And implement a way to correlate project versions; for example, using the fan-in approach described in the delivery-time integration pattern (see "Pattern: Delivery-Time Project Integration" on page 330).

Organize Code by Domain Concept

Code within a single project can include multiple pieces. The application infrastructure project in the ShopSpinner example defines a server cluster and a database instance, and networking structures and security policies for each. Many teams define networking structures and security policies in their own files, as shown in Example 18-2.

Example 18-2. Source files organized by technology

```
└── src/
    ├── cluster.infra
    ├── database.infra
    ├── load_balancer.infra
    ├── routing.infra
    ├── firewall_rules.infra
    └── policies.infra
```

The *firewall_rules.infra* file includes firewall rules for the virtual machines created in *cluster.infra* as well as rules for the database instance defined in *database.infra*.

Organizing code this way focuses on the functional elements over how they're used. It's often easier to understand, write, change, and maintain the code for related elements when they're in the same file. Imagine a file with thirty different firewall rules for access to eight different services, versus a file that defines one service, and the three firewall rules related to it.

This concept follows the design principle of designing around domain concepts rather than technical ones (see "Design components around domain concepts, not technical ones" on page 255).

Organizing Configuration Value Files

Chapter 7 described the configuration files pattern for managing parameter values for different instances of a stack project (see "Pattern: Stack Configuration Files" on page 87). The description suggested two different ways to organize per-environment configuration files across multiple projects. One is to store them within the relevant project:

```
├── application_infra_stack/
│   ├── src/
│   └── environments/
│       ├── test.properties
│       ├── staging.properties
│       └── production.properties
│
└── shared_network_stack/
    ├── src/
    └── environments/
        ├── test.properties
        ├── staging.properties
        └── production.properties
```

The other is to have a separate project with the configuration for all of the stacks, organized by environment:

```
├── application_infra_stack/
│   └── src/
│
├── shared_network_stack/
│   └── src/
│
└── configuration/
    ├── test/
    │   ├── application_infra.properties
    │   └── shared_network.properties
    ├── staging/
    │   ├── application_infra.properties
    │   └── shared_network.properties
    └── production/
        ├── application_infra.properties
        └── shared_network.properties
```

Storing configuration values with the code for a project mixes generalized, reusable code (assuming it's a reusable stack, per "Pattern: Reusable Stack" on page 72) with details of specific instances. Ideally, changing the configuration for an environment shouldn't require modifying the stack project.

On the other hand, it's arguably easier to trace and understand configuration values when they're close to the projects they relate to, rather than mingled in a monolithic configuration project. Team ownership and alignment is a factor, as usual. Separating infrastructure code and its configuration can discourage taking ownership and responsibility across both.

Managing Infrastructure and Application Code

Should application and infrastructure code be kept in separate repositories, or together? Each answer seems obviously correct to different people. The right answer depends on your organization's structure and division of ownership.

Managing infrastructure and application code in separate repositories supports an operating model where separate teams build and manage infrastructure and applications. But it creates challenges if your application teams have responsibility for infrastructure, particularly infrastructure specific to their applications.

Separating code creates a cognitive barrier, even when application team members are given responsibility for elements of the infrastructure that relate to their application.

If that code is in a different repository than the one they most often work in, they won't have the same level of comfort digging into it. This is especially true when it's code that they're less familiar with, and when it's mixed with code for infrastructure for other parts of the system.

Infrastructure code located in the team's own area of the codebase is less intimidating. There's less feeling that a change might break someone else's applications or even fundamental parts of the infrastructure.

DevOps and Team Structures

The DevOps movement encourages organizations to experiment with alternatives to the traditional divide between development and operations. See Matthew Skelton and Manuel Pais's writings on Team Topologies for more in-depth thoughts on structuring application and infrastructure teams.

Delivering Infrastructure and Applications

Regardless of whether you manage application and infrastructure code together, you ultimately deploy them into the same system.[2] Changes to infrastructure code should be integrated and tested with applications throughout the application delivery flow (Figure 18-8).

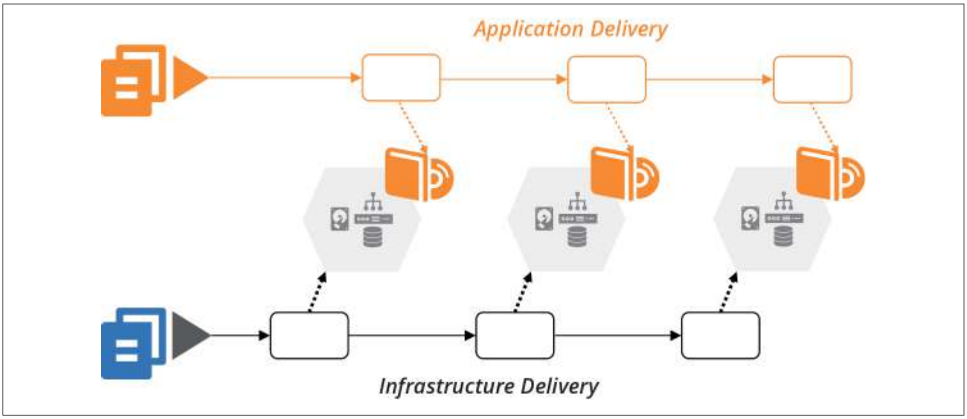


Figure 18-8. Application delivery to environments managed by infrastructure code

As a counter-example, many organizations have a legacy view of production infrastructure as a separate silo. Quite often, one team owns the production infrastructure,

[2] The questions—and patterns—around when to integrate projects are relevant to integrating applications with infrastructure. See "Integrating Projects" on page 326 for more on this.

including a staging or preproduction environment, but doesn't have responsibility for development and testing environments (Figure 18-9).

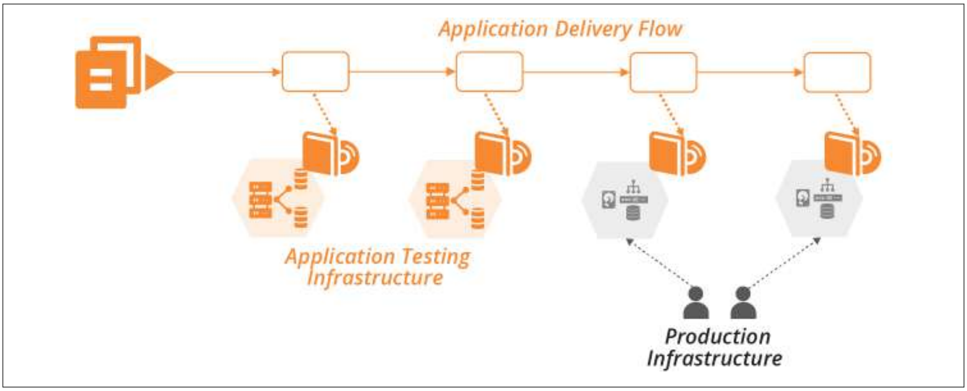

Application Delivery Flow
Application Testing Infrastructure
Production Infrastructure

Figure 18-9. Separate ownership of production infrastructure

This separation creates friction for application delivery, and also for infrastructure changes. Teams don't discover conflicts or gaps between the two parts of the system until late in the delivery process. As explained in Chapter 8, continuously integrating and testing all parts of the system as people work on changes is the most effective way to ensure high quality and reliable delivery.

So your delivery strategy should deliver changes to infrastructure code across all environments. There are a few options for the flow of infrastructure changes.

Testing Applications with Infrastructure

If you deliver infrastructure changes along the application delivery path, you can leverage automated application tests. At each stage, after applying the infrastructure change, trigger the application test stage (see Figure 18-10).

The progressive testing approach ("Progressive Testing" on page 115) uses application tests for integration testing. The application and infrastructure versions can be tied together and progressed through the rest of the delivery flow following the delivery-time integration pattern (see "Pattern: Delivery-Time Project Integration" on page 330). Or the infrastructure change can be pushed on to downstream environments without integrating with any application changes in progress, using apply-time integration (see "Pattern: Apply-Time Project Integration" on page 333).

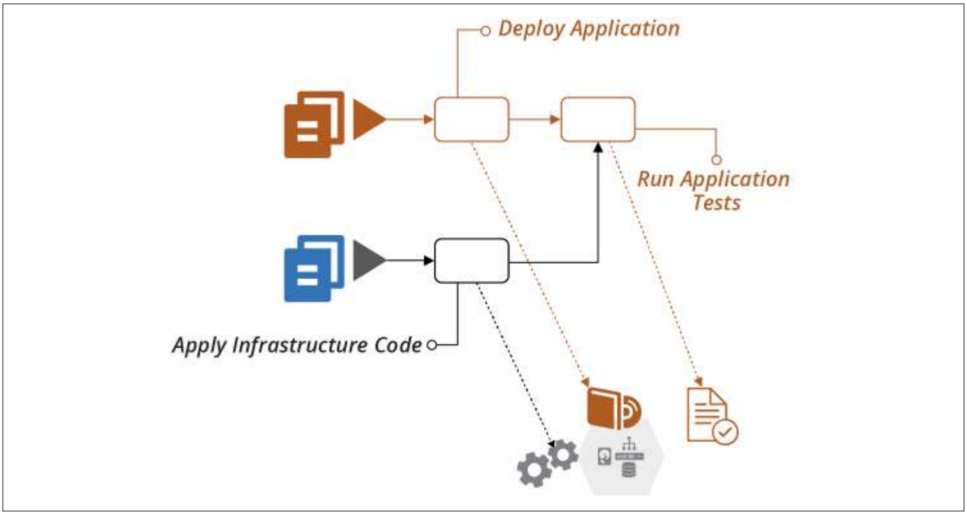


Figure 18-10. Running application tests when infrastructure changes

Pushing changes to applications and infrastructure through as quickly as possible is ideal. But in practice, it's not always possible to remove the friction from all types of changes in an organization. For example, if stakeholders require a deeper review of user-facing application changes, you may need to push routine infrastructure changes faster. Otherwise, your application release process may tie up urgent changes like security patches and minor changes like configuration updates.

Testing Infrastructure Before Integrating

A risk of applying infrastructure code to shared application development and test environments is that breaking those environments impacts other teams. So it's a good idea to have delivery stages and environments for testing infrastructure code on their own, before promoting them to shared environments (see Figure 18-11).

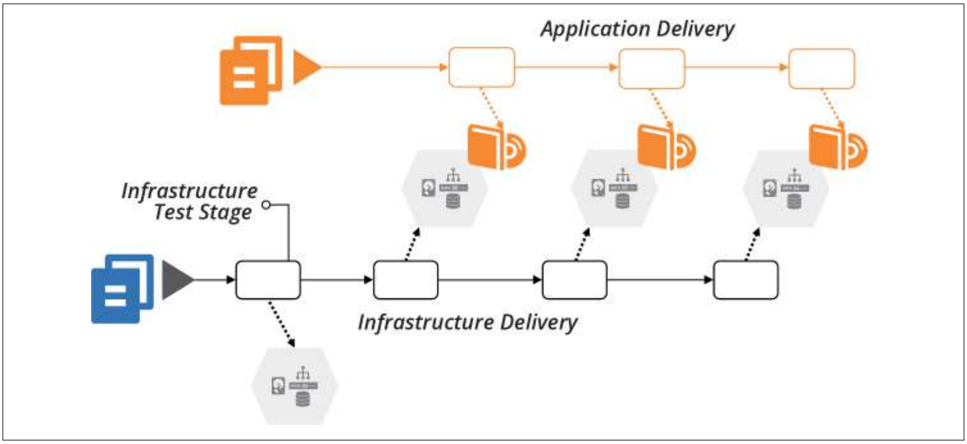


Figure 18-11. Infrastructure testing stage

This idea is a specific implementation of progressive testing ("Progressive Testing" on page 115) and delivery-time project integration ("Pattern: Delivery-Time Project Integration" on page 330).

Using Infrastructure Code to Deploy Applications

Infrastructure code defines what goes onto servers. Deploying an application involves putting things onto servers. So it may seem sensible to write infrastructure code to automate an application's deployment process. In practice, mixing the concerns of application deployment and infrastructure configuration becomes messy. The interface between application and infrastructure should be simple and clear.

Operating system packaging systems like RPMs, *.deb* files, and *.msi* files are a well-defined interface for packaging and deploying applications. Infrastructure code can specify the package file to deploy, then let the deployment tool take over.

Trouble comes when deploying an application involves multiple activities, and especially when it involves multiple moving parts. For example, I once wrote a Chef cookbook to deploy my team's Dropwizard Java applications onto Linux virtual machines. The cookbook needed to:

1. Download and unpack the new application version to a new folder
2. Stop the process for the previous version of the application if it was running
3. Update configuration files if required
4. Update the symlink that points to the current version of the application to the new folder

5. Run database schema migration scripts[3]
6. Start the process for the new application version
7. Check that the new process is working correctly

This cookbook was troublesome for our team, sometimes failing to detect when the previous process hadn't terminated, or that the new process crashed a minute or so after starting.

Fundamentally, this was a procedural script within a declarative infrastructure codebase. We had more success after deciding to package our applications as RPMs, which meant we could use tools and scripts specifically intended for deploying and upgrading applications. We wrote tests for our RPM packaging process, which didn't rely on the rest of our Chef codebase, so we could drill in on the specific issues that made our deployment unreliable.

Another challenge with using infrastructure code to deploy applications is when the deployment process requires orchestrating multiple pieces. My team's process worked fine when deploying our Dropwizard applications to a single server. It wasn't suitable when we moved to load balancing an application across multiple servers.

Even after moving to RPM packages, the cookbooks didn't manage the deployment order across multiple servers. So the cluster would run mixed application versions during the deployment operation. And the database schema migration script should only run once, so we needed to implement locking to ensure that only the first server's deployment process would run it.

Our solution was to move the deployment operation out of the server configuration code, and into a script that pushed applications onto servers from a central deployment location—our build server. This script managed the order of deployment to servers and database schema migrations, implementing zero-downtime deployment by modifying the load balancer's configuration for a rolling upgrade.[4]

Distributed, cloud native applications increase the challenge of orchestrating application deployments. Orchestrating changes to dozens, hundreds, or thousands of application instances can become messy indeed. Teams use deployment tools like Helm or Octopus Deploy to define the deployment of groups of applications. These tools enforce separation of concerns by focusing on deploying the set of applications, leaving the provisioning of the underlying cluster to other parts of the codebase.

[3] See "Using Migration Scripts in Database Deployments" from Red Gate.

[4] We used the expand and contract pattern to make database schema changes without downtime.

However, the most robust application deployment strategy is to keep each element loosely coupled. The easier and safer it is to deploy a change independently of other changes, the more reliable the entire system is.

Conclusion

Infrastructure as Code, as the name suggests, drives the architecture, quality, and manageability of a system's infrastructure from the codebase. So the codebase needs to be structured and managed according to the business requirements and system architecture. It needs to support the engineering principles and practices that make your team effective.

CHAPTER 19

Delivering Infrastructure Code

The software delivery life cycle is a prominent concept in our industry. Infrastructure delivery often follows a different type process in an Iron Age context, where it can arguably be less rigorous. It's common for a change to be made to production infrastructure without being tested first; for example, hardware changes.

But using code to define infrastructure creates the opportunity to manage changes with a more comprehensive process. A change to a manually built system might seem silly to replicate in a development environment, such as changing the amount of RAM in a server. But when the change is implemented in code you can easily roll it across a path to production using a pipeline (see "Infrastructure Delivery Pipelines" on page 119). Doing this would not only catch a problem with the change itself, which might seem trivial (adding more RAM has a pretty small risk of breaking something), but would also catch any issues with the process of applying the change. It also guarantees that all environments across the path to production are consistently configured.

Delivering Infrastructure Code

The pipeline metaphor describes how a change to infrastructure code progresses from the person making the change to production instances. The activities required for this delivery process influence how you organize your codebase.

A pipeline for delivering versions of code has multiple types of activities, including build, promote, apply, and validate. Any given stage in the pipeline may involve multiple activities, as shown in Figure 19-1.

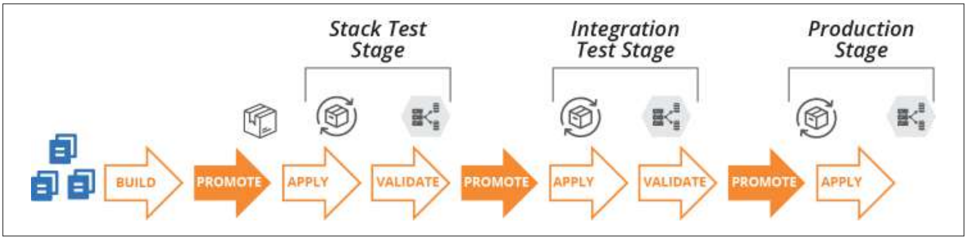


Figure 19-1. Infrastructure code project delivery phases

Building
: Prepares a version of the code for use and makes it available for other phases. Building is usually done once in a pipeline, each time the source code changes.

Promoting
: Moves a version of the code between delivery stages, as described in "Progressive Testing" on page 115. For example, once a stack project version has passed its stack test stage, it might be promoted to show it's ready for the system integration test stage.

Applying
: Runs the relevant tooling to apply the code to the relevant infrastructure instance. The instance could be a delivery environment used for testing activities or a production instance.

"Infrastructure Delivery Pipelines" on page 119 has more detail on pipeline design. Here, we're focusing on building and promoting code.

Building an Infrastructure Project

Building an infrastructure project prepares the code for use. Activities can include:

- Retrieving build-time dependencies, such as libraries, including those from other projects in the codebase and external libraries
- Resolving build-time configuration, for example pulling in configuration values that are shared across multiple projects
- Compiling or transforming code, such as generating configuration files from templates
- Running tests, which include offline and online tests ("Offline Testing Stages for Stacks" on page 131 and "Online Testing Stages for Stacks" on page 134)
- Preparing the code for use, putting it into the format that the relevant infrastructure tool uses to apply it
- Making the code available for use

There are a few different ways to prepare infrastructure code and make it available for use. Some tools directly support specific ways to do this, such as a standard artifact package format or repository. Others leave it to the teams that use the tool to implement their own way to deliver code.

Packaging Infrastructure Code as an Artifact

With some tools, "preparing the code for use" involves assembling the files into a package file with a specific format, an artifact. This process is typical with general-purpose programming languages like Ruby (gems), JavaScript (NPM), and Python (Python packages used with the pip installer). Other package formats for installing files and applications for specific operating systems include *.rpm*, *.deb*, *.msi*, and NuGet (Windows).

Not many infrastructure tools have a package format for their code projects. However, some teams build their own artifacts for these, bundling stack code or server code into ZIP files or "tarballs" (tar archives compressed with gzip). Some teams use OS packaging formats, creating RPM files that unpack Chef Cookbook files onto a server, for example. Other teams create Docker images that include stack project code along with the stack tool executable.

Other teams don't package their infrastructure code into artifacts, especially for tools that don't have a native package format. Whether they do this depends on how they publish, share, and use their code, which depends on what kind of repository they use.

Using a Repository to Deliver Infrastructure Code

Teams use a source code repository for storing and managing changes to their infrastructure source code. They often use a separate repository for storing code that is ready for delivery to environments and instances. Some teams, as we'll see, use the same repository for both purposes.

Conceptually, the build stage separates these two repository types, taking code from the source code repository, assembling it, and then publishing it to the delivery repository (see Figure 19-2).

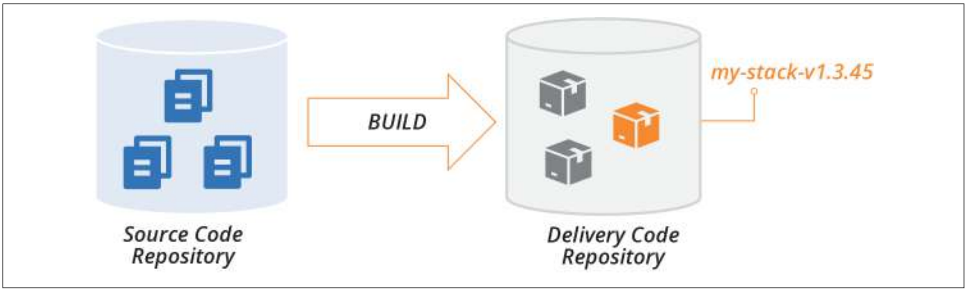


Figure 19-2. Build stage publishes code to the delivery repository

The delivery repository usually stores multiple versions of a given project's code. Promotion phases, as described shortly, mark versions of the project code to show which stage they have progressed to; for example, whether it's ready for integration testing or production.

An apply activity pulls a version of the project code from the delivery repository and applies it to a particular instance, such as the SIT environment or the PROD environment.

There are a few different ways to implement a delivery repository. A given system may use different repository implementations for different types of projects. For example, they may use a tool-specific repository, like Chef Server, for projects that use that tool. The same system might use a general file storage service, such as an S3 bucket, for a project using a tool like Packer that doesn't have a package format or specialized repository.

There are several types of delivery code repository implementations.

Specialized artifact repository

Most of the package formats discussed in the previous section have a package repository product, service, or standard that multiple products and services may implement. There are multiple repository products and hosted services for *.rpm*, *.deb*, *.gem*, and *.npm* files.

Some repository products, like Artifactory and Nexus, support multiple package formats. Teams in an organization that runs one of these sometimes use them to store their artifacts, such as ZIP files and tarballs. Many cloud platforms include specialized artifact repositories, such as server image storage.

The ORAS (OCI Registry As Storage) project offers a way to use artifact repositories originally designed for Docker images as a repository for arbitrary types of artifacts.

If an artifact repository supports tags or labels, you can use these for promotion. For example, to promote a stack project artifact to the system integration test stage, you might tag it with `SIT_STAGE=true`, or `Stage=SIT`.

Alternatively, you might create multiple repositories within a repository server, with one repository for each delivery stage. To promote an artifact, you copy or move the artifact to the relevant repository.

Tool-specific repository

Many infrastructure tools have a specialized repository that doesn't involve packaged artifacts. Instead, you run a tool that uploads your project's code to the server, assigning it a version. This works nearly the same way as a specialized artifact repository, but without a package file.

Examples of these include Chef Server (self-hosted), Chef Community Cookbooks (public), and Terraform Registry (public modules).

General file storage repository

Many teams, especially those who use their own formats to store infrastructure code projects for delivery, store them in a general-purpose file storage service or product. This might be a file server, web server, or object storage service.

These repositories don't provide specific functionality for handling artifacts, such as release version numbering. So you assign version numbers yourself, perhaps by including it in the filename (for example, *my-stack-v1.3.45.tgz*). To promote an artifact, you might copy or link it to a folder for the relevant delivery stage.

Delivering code from a source code repository

Given that source code is already stored in a source repository, and that many infrastructure code tools don't have a package format and toolchain for treating their code as a release, many teams simply apply code to environments from their source code repository.

Applying code from the main branch (trunk) to all of the environments would make it difficult to manage different versions of the code. So most teams doing this use branches, often maintaining a separate branch for each environment. They promote a version of the code to an environment by merging it to the relevant branch. GitOps combines this practice with continuous synchronization (see "Apply Code Continuously" on page 350 and "GitOps" on page 351 for more detail).

Using branches for promoting code can blur the distinction between editing code and delivering it. A core principle of CD is never changing code after the build stage.[1] While your team may commit to never editing code in branches, it's often difficult to maintain this discipline.

Integrating Projects

As mentioned in "Organizing Projects and Repositories" on page 303, projects within a codebase usually have dependencies between them. The next question is when and how to combine different versions of projects that depend on each other.

As an example, consider several of the projects in the ShopSpinner team's codebase. They have two stack projects. One of these, `application_infrastructure-stack`, defines the infrastructure specific to an application, including a pool of virtual machines and load balancer rules for the application's traffic. The other stack project, `shared_network_stack`, defines common networking shared by multiple instances of `application_infrastructure-stack`, including address blocks (VPC and subnets) and firewall rules that allow traffic to application servers.

The team also has two server configuration projects, `tomcat-server`, which configures and installs the application server software, and `monitor-server`, which does the same for a monitoring agent.

The fifth infrastructure project, `application-server-image`, builds a server image using the `tomcat-server` and `monitor-server` configuration modules (see Figure 19-3).

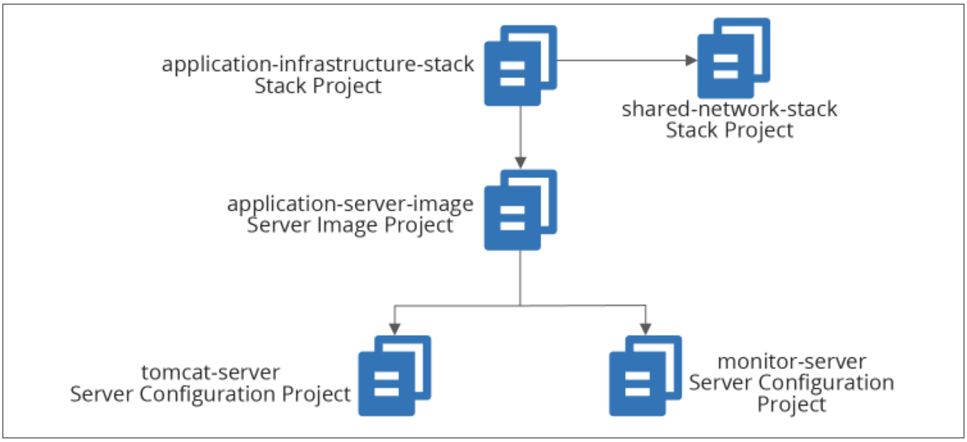


Figure 19-3. Example of dependencies across infrastructure projects

[1] "Only build packages once." See Jez Humble's CD patterns.

The `application_infrastructure-stack` project creates its infrastructure within networking structures created by `shared_network_stack`. It also uses server images built by the `application-server-image` project to create servers in its application server cluster. And `application-server-image` builds server images by applying the server configuration definitions in `tomcat-server` and `monitor-server`.

When someone makes a change to one of these infrastructure code projects, it creates a new version of the project code. That project version must be integrated with a version of each of the other projects. The project versions can be integrated at build time, delivery time, or apply time.

Different projects of a given system can be integrated at different points, as illustrated by the ShopSpinner example in the following pattern descriptions.

Linking and Integrating

Combining the elements of a computer program is referred to as *linking*. There are parallels between linking and integrating infrastructure projects as described in this chapter. Libraries are *statically linked* when building an executable file, similarly to build-time project integration.

Libraries installed on a machine are *dynamically linked* whenever a call is made from an executable, which is somewhat like apply-time project integration. In both cases, either the provider or the consumer can be changed to a different version in the runtime environment.

The analogy is imperfect, as changing a program affects an executable's behavior, while changing infrastructure (generally speaking) affects the state of resources.

Pattern: Build-Time Project Integration

The *build-time project integration* pattern carries out build activities across multiple projects. Doing this involves integrating the dependencies between them and setting the code versions across the projects.

The build process often involves building and testing each of the constituent projects before building and testing them together (see Figure 19-4). What distinguishes this pattern from alternatives is that it produces either a single artifact for all of the projects, or a set of artifacts that are versioned, promoted, and applied as a group.

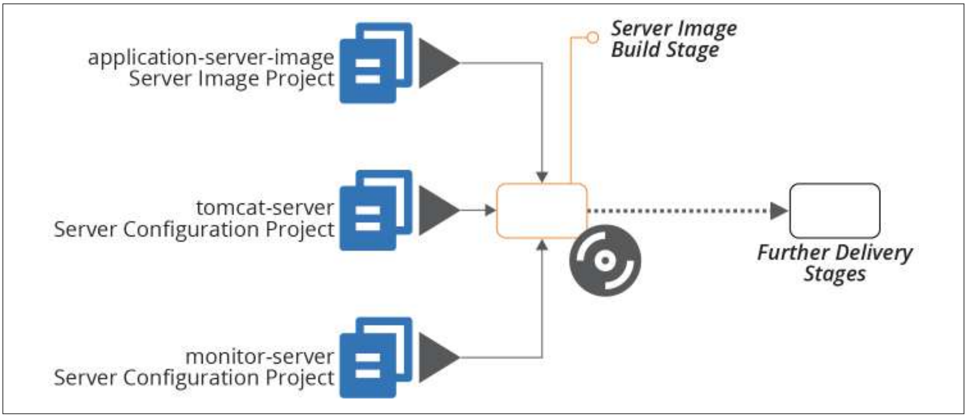


Figure 19-4. Example of integrating projects at build time

In this example, a single build stage produces a server image using multiple server configuration projects.

The build stage may include multiple steps, such as building and testing the individual server configuration modules. But the output—the server image—is composed of code from all of its constituent projects.

Motivation

Building the projects together resolves any issues with dependencies early. Doing this gives fast feedback on conflicts, and creates a high level of consistency across the codebase through the delivery process into production. Project code integrated at build time is consistent throughout the entire delivery cycle. The same version of code is applied at every stage of the process through to production.

Applicability

The use of this pattern versus one of the alternatives is mostly a matter of preference. It depends on which set of trade-offs you prefer, and on your team's ability to manage the complexity of cross-project builds.

Consequences

Building and integrating multiple projects at runtime is complex, especially for very large numbers of projects. Depending on how you implement the build, it may lead to slower feedback times.

Using build-time project integration at scale requires sophisticated tooling to orchestrate builds. Larger organizations that use this pattern across large codebases, such as Google and Facebook, have teams dedicated to maintaining in-house tooling.

Some build tools are available to build very large numbers of software projects, as discussed under implementation. But this approach is not used as widely in the industry as building projects separately, so there are not as many tools and reference materials to help.

Because projects are built together, boundaries between them are less visible than with other patterns. This may lead to tighter coupling across projects. When this happens, it can be hard to make a small change without affecting many other parts of the codebase, increasing the time and risk of changes.

Implementation

Storing all of the projects for the build in a single repository, often called a monorepo, simplifies building them together by integrating code versioning across them (see "Monorepo—One Repository, One Build" on page 305).

Most software build tools, like Gradle, Make, Maven, MSBuild, and Rake, are used to orchestrate builds across a modest number of projects. Running builds and tests across a very large number of projects can take a long time.

Parallelization can speed up this process by building and testing multiple projects in different threads, processes, or even across a compute grid. But this requires more compute resources.

A better way to optimize large-scale builds is using a directed graph to limit building and testing to the parts of the codebase that have changed. Done well, this should reduce the time needed to build and test after a commit so that it only takes a little longer than running a build for separate projects.

There are several specialized build tools designed to handle very large-scale, multiproject builds. Most of these were inspired by internal tools created at Google and Facebook. Some of these tools include Bazel, Buck, Pants, and Please.

Related patterns

The alternatives to integrating project versions at build time are to do so at delivery time (see "Pattern: Delivery-Time Project Integration" on page 330) or apply time (see "Pattern: Apply-Time Project Integration" on page 333).

The strategy of managing multiple projects in a single repository ("Monorepo—One Repository, One Build" on page 305), although not a pattern, supports this pattern. The example I've used for this pattern, Figure 19-4, applies server configuration code when creating a server image (see "Baking Server Images" on page 187). The immutable server pattern ("Pattern: Immutable Server" on page 195) is another example of build-time integration over delivery-time integration.

Although not documented as a pattern in this book, many project builds resolve dependencies on third-party libraries at build time, downloading them and bundling them with the deliverable. The difference is that those dependencies are not built and tested with the project that uses them. When those dependencies come from other projects within the same organization, it's an example of delivery-time project integration (see "Pattern: Delivery-Time Project Integration" on page 330).

Is It Monorepo or Build-Time Project Integration?

Most descriptions of the monorepo strategy for organizing a codebase include building all of the projects in the repository together—what I've called *build-time project integration*. I chose not to name this pattern *monorepo* because that name hinges on the use of a single code repository, which is an implementation option.

I've known teams to manage multiple projects in a single repository without building them together. Although these teams often call their codebase a monorepo, they aren't using the pattern described here.

On the other side, it's technically possible to check out projects from separate repositories and build them together. This fits the pattern described here since it integrates project versions at build time. However, doing this complicates correlating and tracing source code versions to builds, for example, when debugging production issues.

Pattern: Delivery-Time Project Integration

Given multiple projects with dependencies between them, *delivery-time project integration* builds and tests each project individually before combining them. This approach integrates versions of code later than with build-time integration.

Once the projects have been combined and tested, their code progresses together through the rest of the delivery cycle.

As an example, the ShopSpinner `application-infrastructure-stack` project defines a cluster of virtual machines using the server image defined in the `application-server-image` project (see Figure 19-5).

When someone makes a change to the infrastructure stack code, the delivery pipeline builds and tests the stack project on its own, as described in Chapter 9.

If the new version of the stack project passes those tests, it proceeds to the integration test phase, which tests the stack integrated with the last server image that passed its own tests. This stage is the integration point for the two projects. The versions of the projects then progress to later stages in the pipeline together.

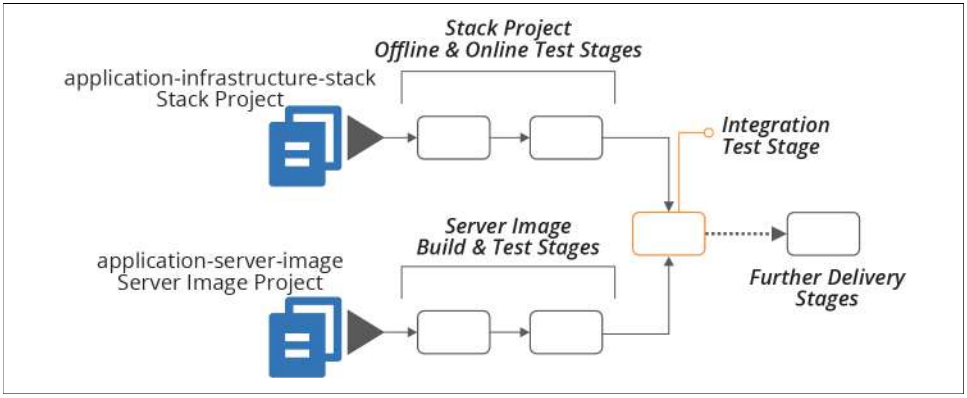


Figure 19-5. Example of integrating projects during delivery

Motivation

Building and testing projects individually before integrating them is one way to enforce clear boundaries and loose coupling between them.

For example, a member of the ShopSpinner team implements a firewall rule in `application-infrastructure-stack`, which opens a TCP port defined in a configuration file in `application-server-image`. They write code that reads the port number directly from that configuration file. But when they push their code, the test stage for the stack fails, because the configuration file from the other project is not available on the build agent.

This failure is a good thing. It exposes coupling between the two projects. The team member can change their code to use a parameter value for the port number to open, setting the value later (using one of the patterns described in Chapter 7). The code will be more maintainable than a codebase with direct references to files across projects.

Applicability

Delivery-time integration is useful when you need clear boundaries between projects in a codebase, but still want to test and deliver versions of each project together. The pattern is difficult to scale to a large number of projects.

Consequences

Delivery-time integration puts the complexity of resolving and coordinating different versions of different projects into the delivery process. This requires a sophisticated delivery implementation, such as a pipeline (see "Delivery Pipeline Software and Services" on page 123).

Implementation

Delivery pipelines integrate different projects using the "fan-in" pipeline design. The stage that brings different projects together is called a fan-in stage, or project integration stage.[2]

How the stage integrates different projects depends on what types of projects it combines. In the example of a stack project that uses a server image, the stack code would be applied and passed a reference to the relevant version of the image. Infrastructure dependencies are retrieved from the code delivery repository (see "Using a Repository to Deliver Infrastructure Code" on page 323).

The same set of combined project versions need to be applied in later stages of the delivery process. There are two common approaches to handling this.

One approach is to bundle all of the project code into a single artifact to use in later stages. For example, when two different stack projects are integrated and tested, the integration stage could zip the code for both projects into a single archive, promoting that to downstream pipeline stages. A GitOps flow would merge the projects to the integration stage branch, and then merge them from that branch to downstream branches.

Another approach is to create a descriptor file with the version numbers of each project. For example:

```
descriptor-version: 1.9.1

stack-project:
  name: application-infrastructure-stack
  version: 1.2.34

server-image-project:
  name: application-server-image
  version: 1.1.1
```

The delivery process treats the descriptor file as an artifact. Each stage that applies the infrastructure code pulls the individual project artifact from the delivery repository.

A third approach would be to tag relevant resources with an aggregated version number.

[2] The fan-in pattern is similar to (or perhaps a different name for) the Aggregate Artifact pipeline pattern.

Related patterns

The build-time project integration pattern (see "Pattern: Build-Time Project Integration" on page 327) integrates the projects at the beginning, rather than after some delivery activities have already taken place on the separate projects. The apply-time project integration pattern integrates the projects at each delivery stage that uses them together but doesn't "lock" the versions.

Pattern: Apply-Time Project Integration

Also known as: decoupled delivery or decoupled pipelines.

Apply-time project integration involves pushing multiple projects through delivery stages separately. When someone changes a project's code, the pipeline applies the updated code to each environment in the delivery path for that project. This version of the project code may integrate with different versions of upstream or downstream projects in each of these environments.

In the ShopSpinner example, the `application-infrastructure-stack` project depends on networking structures created by the `shared-network-stack` project. Each project has its own delivery stages, as illustrated in Figure 19-6.

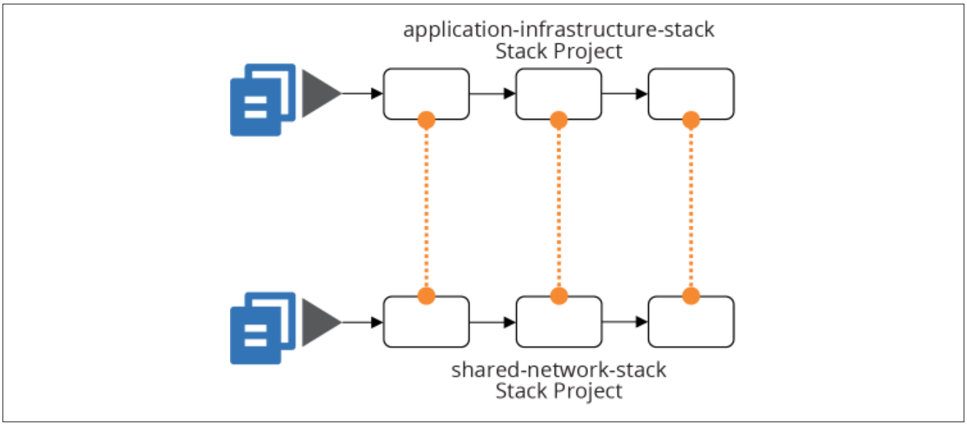


Figure 19-6. Example of integrating projects when applying them

Integration between the projects takes place by applying the `application-infrastructure-stack` code to an environment. This operation creates or changes a server cluster that uses network structures (for example, subnets) from the shared networking.

This integration happens regardless of which version of the shared networking stack is in place in a given environment. So the integration of versions happens separately each time the code is applied.

Motivation

Integrating projects at apply time minimizes coupling between projects. Different teams can push changes to their systems to production without needing to coordinate, and without being blocked by problems with a change to another team's projects.

Applicability

This level of decoupling suits organizations with an autonomous team structure. It also helps with larger-scale systems, where coordinating releases and delivering them in lock-step across hundreds or thousands of engineers is impractical.

Consequences

This pattern moves the risk of breaking dependencies across projects to the apply-time operation. It doesn't ensure consistency through the pipeline. If someone pushes a change to one project through the pipeline faster than changes to other projects, they will integrate with a different version in production than they did in test environments.

Interfaces between projects need to be carefully managed to maximize compatibility across different versions on each side of any given dependency. So this pattern requires more complexity in designing, maintaining, and testing dependencies and interfaces.

Implementation

In some respects, designing and implementing decoupled builds and pipelines using apply-time integration is simpler than alternative patterns. Each pipeline builds, tests, and delivers a single project.

Chapter 17 discusses strategies for integrating different infrastructure stacks. For example, when a stage applies the `application-infrastructure-stack`, it needs to reference networking structures created by the `shared-network-stack`. That chapter explains some techniques for sharing identifiers across infrastructure stacks.

There is no guarantee of which version of another project's code was used in any given environment. So teams need to identify dependencies between projects clearly and treat them as contracts. The `shared-network-stack` exposes identifiers for the networking structures that other projects may use. It needs to expose these in a standardized way, using one of the patterns described in Chapter 17.

As explained in "Using Test Fixtures to Handle Dependencies" on page 137, you can use test fixtures to test each stack in isolation. With the ShopSpinner example, the team would like to test the `application-infrastructure-stack` project without using an instance of `shared-network-stack`. The network stack defines redundant and complex infrastructure that isn't needed to support test cases. So the team's test setup can create a stripped-down set of networking. Doing this also reduces the risk that the application stack evolves to assume details of the network stack's implementation.

A team that owns a project that other projects depend on can implement contract tests that prove its code meets expectations. The `shared-network-stack` can verify that the networking structures—subnets—are created and that their identifiers are exposed using the mechanism that other projects use to consume them.

Make sure that contract tests are clearly labeled. If someone makes a code change that causes the test to fail, they should understand that they may be breaking other projects, rather than thinking they only need to update the test to match their change.

Many organizations find consumer-driven contract (CDC) testing useful. With this model, teams working on a *consumer* project that depends on resources created in a *provider* project write tests that run in the provider project's pipeline. This helps the provider team to understand the expectations of consumer teams.

Related patterns

The build-time project integration pattern ("Pattern: Build-Time Project Integration" on page 327) is at the opposite end of the spectrum from this pattern. That pattern integrates projects once, at the beginning of the delivery cycle, rather than each time. Delivery-time project integration ("Pattern: Delivery-Time Project Integration" on page 330) also integrates projects once, but at a point during the delivery cycle rather than at the beginning.

"Frying a Server Instance" on page 186 illustrates apply-time integration used for server provisioning. Dependencies like server configuration modules are applied each time a new server instance is created, usually taking the latest version of the server module that was promoted to the relevant stage.

Using Scripts to Wrap Infrastructure Tools

Most teams managing infrastructure code create custom scripts to orchestrate and run their infrastructure tools. Some use software build tools like Make, Rake, or Gradle. Others write scripts in Bash, Python, or PowerShell. In many cases, this supporting code becomes at least as complicated as the code that defines the infrastructure, leading teams to spend much of their time debugging and maintaining it.

Teams may run these scripts at build time, delivery time, or apply time. Often, the scripts handle more than one of these project phases. The scripts handle a variety of tasks, which can include:

Configuration
Assemble configuration parameter values, potentially resolving a hierarchy of values. More on this shortly.

Dependencies
Resolve and retrieve libraries, providers, and other code.

Packaging
Prepare code for delivery, whether packaging it into an artifact or creating or merging a branch.

Promotion
Move code from one stage to the next, whether by tagging or moving an artifact or creating or merging a branch.

Orchestration
Apply different stacks and other infrastructure elements in the correct order, based on their dependencies.

Execution
Run the relevant infrastructure tools, assembling command-line arguments and configuration files according to the instance the code is applied to.

Testing
Set up and run tests, including provisioning test fixtures and data, and gathering and publishing results.

Assembling Configuration Values

Marshaling and resolving configuration values can be one of the more complex wrapper script tasks. Consider a system like the ShopSpinner example that involves multiple delivery environments, multiple production customer instances, and multiple infrastructure components.

A simple, one-level set of configuration values, with one file per combination of component, environment, and customer, requires quite a few files. And many of the values are duplicated.

Imagine a value of `store_name` for each customer, which must be set for every instance of each component. The team quickly decides to set that value in one location with shared values and add code to its wrapper script to read values from the shared configuration and the per-component configuration.

They soon discover they need some shared values across all of the instances in a given environment, creating a third set of configuration. When a configuration item has different values in multiple configuration files, the script must resolve it following a precedence hierarchy.

This type of parameter hierarchy is a bit messy to code. It's harder for people to understand when introducing new parameters, configuring the right values, and tracing and debugging the values used in any given instance.

Using a configuration registry puts a different flavor onto the complexity. Rather than chasing parameter values across an array of files, you chase them through various subtrees of the registry. Your wrapper script might handle resolving values from different parts of the registry, as with the configuration files. Or you might use a script to set registry values beforehand, so it owns the logic for resolving a hierarchy of default values to set the final value for each instance. Either approach creates headaches for setting and tracing the origin of any given parameter value.

Simplifying Wrapper Scripts

I've seen teams spend more time dealing with bugs in their wrapper scripts than on improving their infrastructure code. This situation arises from mixing and coupling concerns that can and should be split apart. Some areas to consider:

Split the project life cycle

A single script shouldn't handle tasks during the build, promotion, and apply phases of a project's life cycle. Write and use different scripts for each of these activities. Make sure you have a clear understanding of where information needs to be passed from one of these phases to the next. Implement clear boundaries between these phases, as with any API or contract.

Separate tasks

Break out the various tasks involved in managing your infrastructure, such as assembling configuration values, packaging code, and executing infrastructure tools. Again, define the integration points between these tasks and keep them loosely coupled.

Decouple projects

A script that orchestrates actions across multiple projects should be separate from scripts that execute tasks within the projects. And it should be possible to execute tasks for any project on its own.

Keep wrapper code ignorant

Your scripts should not know anything about the projects they support. Avoid hardcoding actions that depend on what the infrastructure code does into your wrapper scripts. Ideal wrapper scripts are generic, usable for any infrastructure project of a particular shape (e.g., any project that uses a particular stack tool).

Treating wrapper scripts like "real" code helps with all of this. Test and validate scripts with tools like shellcheck. Apply the rules of good software design to your scripts, such as the rule of composition, single responsibility principle, and designing around domain concepts. See Chapter 15 and the references to other sources of information on good software design.

Conclusion

Creating a solid, reliable process for delivering infrastructure code is a key to achieving good performance against the four key metrics (see "The Four Key Metrics" on page 9). Your delivery system is the literal implementation of fast and reliable delivery of changes to your system.

CHAPTER 20

Team Workflows

Using code to build and change infrastructure is a radically different way of working from traditional approaches. We make changes to virtual servers and network configuration indirectly, rather than by typing commands at prompts or directly editing live configuration. Writing code and then pushing it out to be applied by automated systems is a bigger shift than learning a new tool or skill.

Infrastructure as Code changes how everyone involved in designing, building, and governing infrastructure works as individuals and as a group. This chapter aims to explain how different people work on infrastructure code. Processes for working on infrastructure involve designing, defining, and applying code.

Some characteristics of effective processes for teams who manage Infrastructure as Code include:

- The automated process is the easiest, most natural way for team members to make changes.
- People have a clear way to ensure quality, operability, and alignment to policies.
- The team keeps its systems up-to-date with little effort. Things are consistent where appropriate, and where variations are required, they are clear and well-managed.
- The team's knowledge of the system is embedded in code, and its ways of working are articulated in the automation.
- Errors are quickly visible and easily corrected.
- It's easy and safe to change the code that defines the system and the automation that tests and delivers that code.

Overall, a good automated workflow is fast enough to get a fix through to systems in an emergency, so that people aren't tempted to jump in and make a manual change to fix it. And it's reliable enough that people trust it more than they trust themselves to twiddle configuration on a live system by hand.

This chapter and the next one both discuss elements of how a team works on infrastructure code. This chapter focuses on what people do in their workflows, while the following one looks at ways of organizing and managing infrastructure codebases.

Measuring the Effectiveness of Your Workflow

The four key metrics from the Accelerate research, as mentioned in "The Four Key Metrics" on page 9, are a good basis for deciding how to measure your team's effectiveness. The evidence is that organizations that perform well on these metrics tend to perform well against their core organizational goals, such as profitability and share price.

Your team might use these metrics to create SLIs (Service Level Indicators), which are things to measure, SLOs (Service Level Objectives), which are targets your team uses for itself, and SLAs (Service Level Agreements), which are commitments to other people.[1] The specific things you measure depend on your team's context and specific ways you're trying to improve higher-level outcomes.

The People

A reliable automated IT system is like Soylent Green—its secret ingredient is people.[2] While human hands shouldn't be needed to get a code change through to production systems, other than perhaps reviewing test results and clicking a few buttons, people are needed to continuously build, fix, adapt, and improve the system.

There are a handful of roles involved with most infrastructure systems, automated or otherwise. These roles don't often map one to one to individuals—some people play more than one role, and some people share roles with others:

[1] See Google's "SRE Fundamentals" for more on SLOs, SLAs, and SLIs.

[2] Soylent Green is a food product in the classic dystopian science fiction movie of the same name. Spoiler: "Soylent Green is people!" Although, my lawyers advise me to point out that, for a reliable automated IT system, the secret ingredient is *living* people.

Users

Who directly uses the infrastructure? In many organizations, application teams do. These teams may develop the applications, or they may configure and manage third-party applications.

Governance specialists

Many people set policies for the environment across various domains, including security, legal compliance, architecture, performance, cost control, and correctness.

Designers

People who design the infrastructure. In some organizations, these people are architects, perhaps divided into different domains, like networking or storage.

Toolmakers

People who provide services, tools, and components that other teams use to build or run environments. Examples include a monitoring team or developers who create reusable infrastructure code libraries.

Builders

People who build and change infrastructure. They could do this manually through consoles or other interfaces, by running scripts, or by running tools that apply infrastructure code.

Testers

People who validate infrastructure. This role isn't limited to QAs (quality analysts). It includes people who test or review systems for a governance domain like security or performance.

Support

People who make sure the system continues to run correctly and fix it when it doesn't.

Figure 20-1 shows a classic structure, with a dedicated team for each part of the workflow for changing a system.

Many roles may be divided across different infrastructure domains, such as networking, storage, or servers. They are also potentially split across governance domains like security, compliance, architecture, and performance. Many larger organizations create baroque organizational structures of micro-specialties.[3]

[3] I worked with a group at an international bank that had four different release testing environments, one for each stage in their release process. For each of these environments, one team configured the infrastructure, another team deployed and configured the application, and then a third team tested it. Some of these twelve teams were unaware that their counterparts existed. The result was little knowledge sharing and no consistency across the release process.

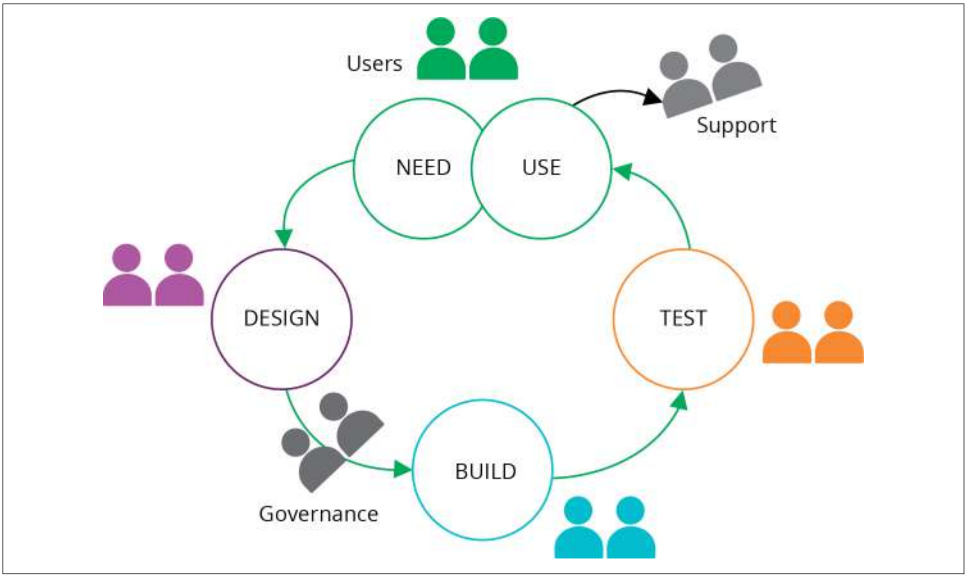


Figure 20-1. A classic mapping of a dedicated team to each part of a workflow

However, it's also common for a person or team to work across these roles. For example, an infosec (information security) team might set standards, provide scanning tools, and conduct security audits. A bit later in this chapter we'll look at ways to reshuffle responsibilities (see "Reshuffling Responsibilities" on page 352).

Who Writes Infrastructure Code?

Here are a few different ways organizations answer the question of who writes and edits infrastructure code:

Builders write code
: Some organizations try to keep traditional processes and team structures. So the team that builds (and perhaps supports) infrastructure uses Infrastructure as Code tools to optimize its work. Users request an environment, and the build team uses its tools and scripts to build it for them. See "Using Value Stream Mapping to Improve Workflows" on page 343 for an example of how optimizing a build team's process tends not to improve the end-to-end process, either for speed or quality.

Users write code
: Many organizations enable application teams to define the infrastructure that their applications use. This aligns user needs to the solution. However, it either requires every team to include people with strong infrastructure expertise or tooling that simplifies defining infrastructure. The challenge with tooling is

ensuring that it meets the needs of application teams, rather than constraining them.

Toolmakers write code

Specialist teams can create platforms, libraries, and tools that enable users to define the infrastructure they need. In these cases, the users tend to write more configuration than code. The difference between toolmakers and builders writing code is self-service. A builder team writes and uses code in response to requests from users to create or change an environment. A toolmaker team writes code that users can use to create or change their own environments. See "Building an Abstraction Layer" on page 284 as an example of what toolmakers might build.

Governance and testers write code

People who set policies and standards, and those who need to assure changes, can create tools that help other people to verify their own code. These people may become toolmakers or work closely with toolmakers.

Using Value Stream Mapping to Improve Workflows

Value stream mapping is a useful way to break down your lead time, so you can understand where the time goes.[4]

By measuring the time spent on various activities, including waiting, you can focus your improvements on areas that make the most difference. Too often, we optimize parts of our process that seem the most obviously inefficient but which have little impact on the total lead time. For example, I've seen teams implement automation to cut the time it takes to provision a server from eight hours to ten minutes. This is a massive 98% decrease in the time to provision a server. However, if users typically wait ten days to get their new server, the total decrease is a much less exciting 10% decrease. If server request tickets wait in a queue for an average of eight days, you should focus your efforts there instead.

Value stream mapping makes the time to complete an action visible so that you can find the best opportunities for improvement. Continue to measure the end-to-end lead time, and other metrics such as failure rates, while you make improvements. Doing this helps avoid optimizations to one part of your process that make the full flow worse.

[4] The book *Value Stream Mapping* by Karen Martin and Mike Osterling (McGraw-Hill Education) is a good reference.

Applying Code to Infrastructure

A typical workflow for making a change to infrastructure starts with code in a shared source repository. A member of the team pulls the latest version of the code to their working environment and edits it there. When they're ready, they push the code into the source repository, and apply the new version of the code to various environments.

Many people run their tools from the command line in their working environment when they are starting with infrastructure automation. However, doing that has pitfalls.

Applying Code from Your Local Workstation

Applying infrastructure code from the command line can be useful for a test instance of the infrastructure that nobody else uses. But running the tool from your local work environments creates problems with shared instances of infrastructure, whether it's a production environment or a delivery environment (see "Delivery Environments" on page 66).

The person might make changes to their local version of the code before applying it. If they apply the code before pushing the changes to the shared repository, then nobody else has access to that version of the code. This can cause problems if someone else needs to debug the infrastructure.

If the person who applied their local version of the code does not immediately push their changes, someone else might pull and edit an older version of the code. When they apply that code, they'll revert the first person's changes. This situation quickly becomes confusing and hard to untangle (see Figure 20-2).

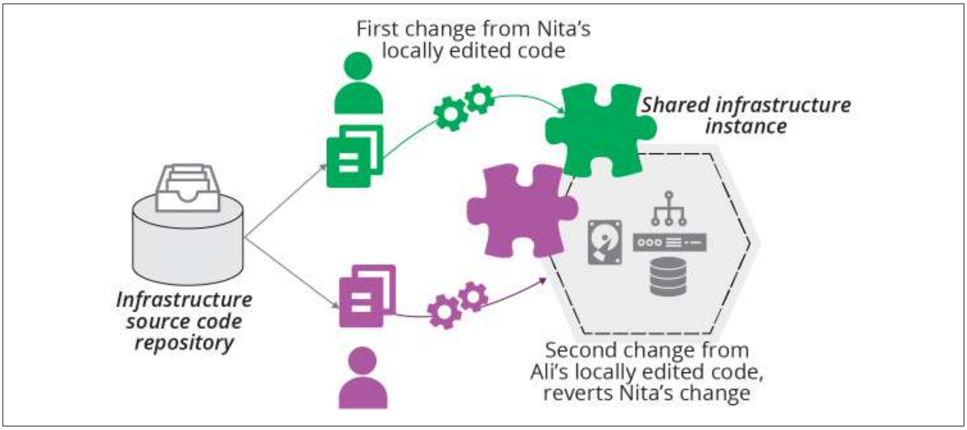

First change from Nita's locally edited code
Shared infrastructure instance
Infrastructure source code repository
Second change from Ali's locally edited code, reverts Nita's change

Figure 20-2. Locally editing and applying code leads to conflicts

Note that locking solutions, such as Terraform's state locking, don't prevent this situation. Locking stops two people from applying their code to the same instance simultaneously. But, as of this writing, locking solutions don't stop people from applying divergent versions of code, as long as they each wait their turn.

So the lesson is that, for any instance of infrastructure, code should always be applied from the same location. You could designate an individual to be responsible for each instance. But this has many pitfalls, including a risky dependency on one person and their workstation. A better solution is to have a central system that handles shared infrastructure instances.

Applying Code from a Centralized Service

You can use a centralized service to apply infrastructure code to instances, whether it's an application that you host yourself or a third-party service. The service pulls code from a source code repository or an artifact repository (see "Packaging Infrastructure Code as an Artifact" on page 323) and applies it to the infrastructure, imposing a clear, controlled process for managing which version of code is applied.

If two people pull and edit code, they must resolve any differences in their code when they integrate their code with the branch that the tool uses. When there is a problem, it's easy to see which version of the code was applied and correct it (see Figure 20-3).

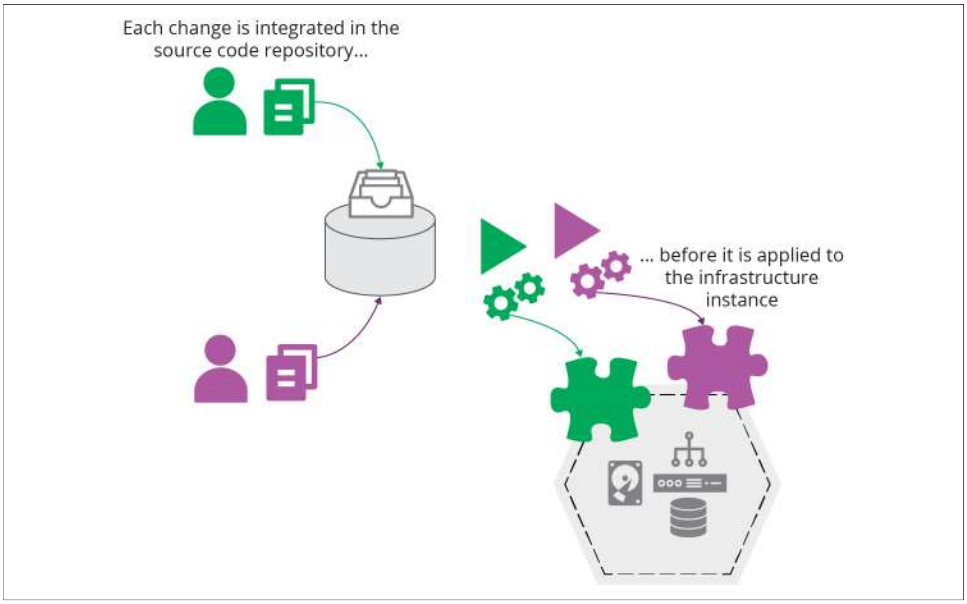

Each change is integrated in the source code repository...
... before it is applied to the infrastructure instance

Figure 20-3. Centrally integrating and applying code

A central service also ensures that the infrastructure tool is run consistently, rather than assuming that a person doesn't make a mistake or "improve" (deviate from) the workflow. It uses the same versions of the tool, scripts, and supporting utilities every time.

Using a central service aligns well with the pipeline model for delivering infrastructure code across environments (see "Infrastructure Delivery Pipelines" on page 119). Whatever tool or service you use to orchestrate changes through your pipeline takes on the responsibility for running your infrastructure tool to apply the latest version of code to each environment.

Another benefit of having a central service execute your infrastructure code is that it forces you and your team to automate the entire process. If you run the tool from your workstation, it's easy to leave a few loose ends, steps that you need to do manually before or after running the tool. A central service gives you no choice other than making sure the task is entirely automated.

Tools and Services That Run Your Infrastructure Tool for You

There are several options for a centralized service to apply infrastructure code. If you use a build server like Jenkins or a CD tool like GoCD or ConcourseCI, you can implement jobs or stages to execute your infrastructure tool. These tools support managing different versions of code from a source repository and can promote code between stages. These multipurpose tools also make it easy to integrate workflows across applications, infrastructure, and other parts of your system. You can use self-hosted instances of these services, or use a hosted offering. See "Delivery Pipeline Software and Services" on page 123 for more on pipeline servers and software.

Several vendors offer products or services specialized for running infrastructure tools. Examples include Terraform Cloud, Atlantis, and Pulumi for Teams. WeaveWorks provides Weave Cloud, which applies infrastructure code to Kubernetes clusters.

Personal Infrastructure Instances

In most of the workflows discussed in this book, you pull code, edit it, and then push it into a shared code repository.[5] Then a pipeline delivery process applies it to relevant environments.

Ideally, you can test your code changes before you push them into the shared repository. Doing this gives you a way to make sure your change does what you expect and is faster than waiting for a pipeline to run your code through to an online test stage (see "Online Testing Stages for Stacks" on page 134). It also helps avoid breaking the build when one of your changes fails a pipeline stage, disrupting everyone working on the codebase.

Your team can do a few things to make it easier to test code changes before pushing.

First, make sure each person working on the infrastructure code can create their own instance of the infrastructure. There's a limit to what people can test locally, without your cloud platform, as discussed in "Offline Testing Stages for Stacks" on page 131. You might be tempted to run shared "dev" instances of the infrastructure. But as explained earlier, having multiple people apply locally edited code to a shared instance becomes messy. So create a way for people to spin up their own infrastructure instances, and destroy them when they're not actively using them.

Second, keep the pieces of your infrastructure small. This is, of course, one of the three core practices in this book (see Chapter 15). You should be able to spin up an instance of any component of your system on its own, perhaps using test fixtures to handle dependencies (see "Using Test Fixtures to Handle Dependencies" on page 137). It's hard for people to work on personal instances if they need to spin up the entire system, unless the system is exceptionally small.

Third, people should use the same tools and scripts to apply and test their instances of infrastructure that are used with shared instances, for example, in the pipeline. It helps to create packages for the tooling and scripts that you can use across these different locations.[6]

[5] Particularly in Chapters 8 and 9.

[6] batect and Dojo are examples of tools that build a repeatable, shareable container for developing applications and infrastructure.

Centrally Managed Personal Instances

It's safer for people to apply code to personal instances from their workstations than it is for shared instances. But there may be advantages to using a centralized service for personal instances. I've seen a team struggle to tear down a personal instance that someone left running when they went on vacation. They created the instance using a local version of the infrastructure code that they didn't push to the repository, making it hard to destroy.

So some teams establish a practice where each person pushes changes to a personal branch, which the central service applies to their personal infrastructure instance that they can test. In this arrangement, the personal branch emulates local code, so people don't consider a change committed until they've merged it to the shared branch or trunk. But the code is centrally available for other people to view and use in their absence.

Source Code Branches in Workflows

Branches are a powerful feature of shared source repositories that make it easier for people to make changes to different copies of a codebase—*branches*—and then integrate their work when they're ready. There are many strategies and patterns for using branches as part of a team's workflow. Rather than elaborating on them here, I refer you to Martin Fowler's article, "Patterns for Managing Source Code Branches".

It's worth highlighting a few distinctions of branching strategy in the context of Infrastructure as Code. One is the difference between path to production patterns and integration patterns for branching. The other is the importance of integration frequency.

Teams use *path to production* branching patterns to manage which versions of code to apply to environments.[7] Typical path to production patterns include release branches and environment branches (environment branches are discussed in "Delivering code from a source code repository" on page 325).

Integration patterns for branching describes ways for people working on a codebase to manage when and how they integrate their work.[8] Most teams use the mainline integration pattern, either with feature branching or continuous integration.

The specific pattern or strategy is less important than how you use it. The most important factor in the effectiveness of a team's use of branches is *integration frequency*, how often everyone merges all of their code into the same (main) branch of

[7] See the path to production section of Fowler's article, "Patterns for Managing Source Code Branches".

[8] Fowler also describes integration patterns in the article.

the central repository.[9] The DORA Accelerate research finds that more frequent integration of all of the code within a team correlates to higher commercial performance. Their results suggest that everyone in the team should integrate all of their code together—for example, to main or trunk—at least once a day.

Merging Is Not Integration

People sometimes confuse a build server automatically running tests on branches with continuous integration. The practice of continuous integration, and the correlation with higher team performance, is based on fully integrating all of the changes that everyone is working on in the codebase.

Although someone using the feature branch pattern may frequently merge the current main branch to their own branch, they don't normally integrate their own work back into main until they've finished working on their feature. And if other people are working the same way on their own feature branches, then the code is not fully integrated until everyone finishes their feature and merges their changes to the main branch.

Integration involves merging in both directions—individuals merging their own changes to the main branch as well as merging main back to their own branch or local copy of the code. Continuous integration, therefore, means everyone doing this as they work, at least once a day.

Preventing Configuration Drift

Chapter 2 described the perils of configuration drift (see "Configuration Drift" on page 17), where similar infrastructure elements become inconsistent over time. Configuration drift often results when teams use infrastructure coding tools to automate parts of older ways of working, rather than fully adapting their ways of working.

There are several things you can do in your workflows to avoid configuration drift.

Minimize Automation Lag

Automation lag is the time that passes between instances of running an automated process, such as applying infrastructure code. The longer it's been since the last time

[9] See the section on integration frequency for a detailed examination.

the process ran, the more likely it will fail.[10] Things change over time, even when nobody has consciously made a change.

Even if code hasn't changed, applying it after a long gap in time can still fail, for various reasons:

- Someone has changed another part of the system, such as a dependency, in a way that only breaks when your code is reapplied.
- An upgrade or configuration change to a tool or service used to apply your code might be incompatible with your code.
- Applying unchanged code might nevertheless bring in updates to transitive dependencies, such as operating system packages.
- Someone may have made an manual fix or improvement that they neglected to fold back into the code. Reapplying your code reverts the fix.

The corollary to automation lag is, the more frequently you apply infrastructure code, the less likely it is to fail. When failures do occur, you can discover the cause more quickly, because less has changed since the last successful run.

Avoid Ad Hoc Apply

One habit that some teams carry over from Iron Age ways of working is only applying code to make a specific change. They might only use their infrastructure code to provision new infrastructure, but not for making changes to existing systems. Or they may write and apply infrastructure code to make ad hoc changes to specific parts of their systems. For example, they code a one-off configuration change for one of their application servers. Even when teams use code to make changes, and apply code to all instances, sometimes they may only apply code when they make a change to the code.

These habits can create configuration drift or automation lag.

Apply Code Continuously

A core strategy for eliminating configuration drift is to continuously apply infrastructure code to instances, even when the code hasn't changed. Many server configuration tools, including Chef and Puppet, are designed to reapply configuration on a schedule, usually hourly.[11]

[10] Automation lag applies to other types of automation as well. For example, if you only run your automated application test suite at the end of a long release cycle, you will spend days or weeks updating the test suite to match the code changes. If you run the tests every time you commit a code change, you only need to make a few changes to match the code, so the full test suite is always ready to run.

[11] See ConfigurationSynchronization for an early articulation of this concept.

The GitOps methodology (see "GitOps" on page 351) involves continuously applying code from a source code branch to each environment. You should be able to use a central service to apply code (as described in "Applying Code from a Centralized Service" on page 345) to continuously reapply code to each instance.

Immutable Infrastructure

Immutable infrastructure solves the problem of configuration drift in a different way. Rather than applying configuration code frequently to an infrastructure instance, you only apply it once, when you create the instance. When the code changes, you make a new instance and swap it out for the old one.

Making changes by creating a new instance requires sophisticated techniques to avoid downtime ("Zero Downtime Changes" on page 375), and may not be applicable to all use cases. Automation lag is still potentially an issue, so teams that use immutable infrastructure tend to rebuild instances frequently, as with phoenix servers.[12]

GitOps

GitOps is a variation of Infrastructure as Code that involves continuously synchronizing code from source code branches to environments. GitOps emphasizes defining systems as code (see "Core Practice: Define Everything as Code" on page 10).

GitOps doesn't prescribe an approach to testing and delivering infrastructure code, but it is compatible with using a pipeline to deliver code ("Infrastructure Delivery Pipelines" on page 119). However, GitOps discourages the use of delivery artifacts ("Packaging Infrastructure Code as an Artifact" on page 323), instead promoting code changes by merging them to source code branches (see "Delivering code from a source code repository" on page 325).

Another key element of GitOps is continuously synchronizing code to systems ("Apply Code Continuously" on page 350). Rather than having a build server job or pipeline stage apply the code when it changes ("Avoid Ad Hoc Apply" on page 350), GitOps uses a service that continuously compares the code to the system, reducing configuration drift (see "Configuration Drift" on page 17).

Some teams describe their process as GitOps, but only implement the branches for environments practice without continuously synchronizing code to environments. This makes it to easy to fall into an ad hoc change process, and bad habits of copying, pasting, and editing code changes for each environment, per the copy-paste antipattern (see "Antipattern: Copy-Paste Environments" on page 70.

[12] A phoenix server is frequently rebuilt, in order to ensure that the provisioning process is repeatable. This can be done with other infrastructure constructs, including infrastructure stacks.

Governance in a Pipeline-based Workflow

Governance is a concern for most organizations, especially larger ones and those that work in regulated industries like finance and health care. Some people see governance as a dirty word that means adding unneeded friction to getting useful work done. But governance just means making sure things are done responsibly, according to the organization's policies.

Chapter 1 explained that quality—governance being an aspect of quality—can enable delivery speed and that the ability to deliver changes quickly can improve quality (see "Use Infrastructure as Code to Optimize for Change" on page 4.) Compliance as Code[13] leverages automation and more collaborative working practices to make this positive loop work.

Reshuffling Responsibilities

Defining systems as code creates opportunities to reshuffle the responsibilities of the people involved in working on infrastructure (the people listed in "The People" on page 340) and the way those people engage with their work. Some factors that create these opportunities are:

Reuse
: Infrastructure code can be designed, reviewed, and reused across multiple environments and systems. You don't need a lengthy design, review, and signoff exercise for each new server or environment if you use code that has already been through that process.

Working code
: Because code is quick to write, people can review and make decisions based on working code and example infrastructure. This makes for faster and more accurate feedback loops than haggling over diagrams and specifications.

Consistency
: Your code creates environments far more consistently than humans following checklists. So testing and reviewing infrastructure in earlier environments gives faster and better feedback than doing these later in the process.

Automated testing
: Automated testing, including for governance concerns like security and compliance, gives people working on infrastructure code fast feedback. They can correct many problems as they work, without needing to involve specialists for routine issues.

[13] See the O'Reilly website for articles on compliance as code.

Democratize quality

People who aren't specialists can make changes to the code for potentially sensitive areas of infrastructure, such as networking and security policies. They can use tools and tests created by specialists to sanity check their changes. And specialists can still review and approve changes before they're applied to production systems. Reviewing this way is more efficient because the specialist can directly view code, test reports, and working test instances.

Governance channels

The infrastructure codebase, and pipelines used to deliver changes to production instances, can be organized based on their governance requirements. So a change to security policies goes through a review and signoff step not necessarily required for changes to less sensitive areas.

Many of the ways we can change how people manage systems involve shifting responsibilities left in the process.

Shift Left

Chapter 8 explains principles and practices for implementing automated testing and pipelines to deliver code changes to environments. The term *shift left* describes how this impacts workflows and delivery practices.

Code is rigorously tested during implementation, at the "left" end of the flow shown in most process diagrams. So organizations can spend less time on heavyweight processes at the "right" end, just before applying code to production.

People involved in governance and testing focus on what happens during implementation, working with teams, providing tools, and enabling practices to test early and often.

An Example Process for Infrastructure as Code with Governance

ShopSpinner has a reusable stack ("Pattern: Reusable Stack" on page 72) that it can use to create the infrastructure for an application server to host an instance of its service for a customer. When someone changes the code for this stack, it can affect all of its customers.

Its technical leadership group, which is responsible for architectural decisions, defines CFRs (as described in "What Should We Test with Infrastructure?" on page 108) that the application server infrastructure must support. These CFRs include the number and frequency of orders that users can place on a customer instance, the response times for the interface, and recovery times for server failures.

The infrastructure team and the application team join with a pair of Site Reliability Engineers (SREs) and a QA to implement some automated tests that check the performance of the application server stack against the CFRs. They build these tests into the several stages of the pipeline, progressively testing different components of the stack (as per "Progressive Testing" on page 115).

Once the group has these tests in place, people don't need to submit infrastructure changes for review by the technical leadership group, SREs, or anyone else. When an engineer changes the networking configuration, for example, the pipeline automatically checks whether the resulting infrastructure still meets the CFRs before they can apply it to production customer instances. If the engineer makes a mistake that breaks a CFR, they find out within minutes when a pipeline stage goes red and can immediately correct it.

In some cases, a change may cause a problem with a customer instance that isn't caught by the automated tests. The group can conduct a blameless postmortem to review what happened. Perhaps the problem was that none of the CFRs covered the situation, so they need to change or add a CFR to their list. Or their testing may have a gap that missed the issue, in which case they improve the test suite.

Normalize Your Emergency Fix Process

Many teams have a separate process for emergency changes so that they can deliver fixes quickly. Needing a separate process for faster fixes is a sign that the normal change process could be improved.

An emergency change process speeds things up in one of two ways. One is to leave out unnecessary steps. The other is to leave out necessary steps. If you can safely leave a step out in an emergency, when the pressure is on and the stakes are high, you can probably leave it out of your normal process. If skipping a step is unacceptably risky, then find a way to handle it more efficiently and do it every time.[14]

Conclusion

When an organization defines its Infrastructure as Code, its people should find themselves spending less time carrying out routine activities and playing gatekeeper. They should instead spend more time continuously improving their ability to improve the system itself. Their efforts will be reflected in the four metrics for software delivery and operational performance.

[14] Steve Smith defines this as the *dual value streams* antipattern.

CHAPTER 21

Safely Changing Infrastructure

The theme of making changes frequently and quickly runs throughout this book. As mentioned at the very start ("Objection: "We must choose between speed and quality"" on page 7), far from making systems unstable, speed is an enabler for stability, and vice versa. The mantra is not "move fast and break things," but rather, "move fast and improve things."

However, stability and quality don't result from optimizing purely from speed. The research cited in the first chapter shows that trying to optimize for either speed or quality achieves neither. The key is to optimize for both. Focus on being able to make changes frequently, quickly, and safely, and on detecting and recovering from errors quickly.

Everything this book recommends—from using code to build infrastructure consistently, to making testing a continuous part of working, to breaking systems into smaller pieces—enables fast, frequent, and safe changes.

But making frequent changes to infrastructure imposes challenges for delivering uninterrupted services. This chapter explores these challenges and techniques for addressing them. The mindset that underpins these techniques is not to see changes as a threat to stability and continuity, but to exploit the dynamic nature of modern infrastructure. Exploit the principles, practices, and techniques described throughout this book to minimize disruptions from changes.

Reduce the Scope of Change

Agile, XP, Lean, and similar approaches optimize the speed and reliability of delivery by making changes in small increments. It's easier to plan, implement, test, and debug a small change than a large one, so we aim to reduce batch sizes.[1] Of course, we often need to make significant changes to our systems, but we can do this by breaking things up into a small set of changes that we can deliver one at a time.

As an example, the ShopSpinner team initially built its infrastructure with a single infrastructure stack. The stack included its web server cluster and an application server. Over time, the team members added more application servers and turned some into clusters. They realized that running the web server cluster and all of the application servers in a single VLAN was a poor design, so they improved their network design and shifted these elements into different VLANs. They also decided to take the advice of this book and split their infrastructure into multiple stacks to make it easier to change them individually.

ShopSpinner's original implementation was a single stack with a single VLAN (see Figure 21-1).

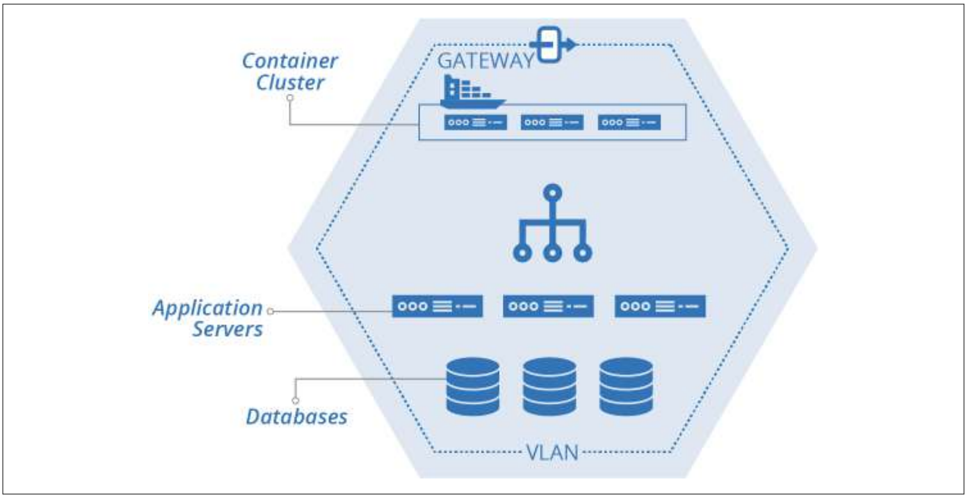


Figure 21-1. Example starting implementation, single stack, single VLAN

The team plans to split its stack out into multiple stacks. These include the `shared-networking-stack` and `application-infrastructure-stack` seen in examples from previous chapters of this book. The plan also includes a `web-cluster-stack` to manage the container cluster for the frontend web servers, and an `application-database-stack` to manage a database instance for each application (Figure 21-2).

[1] Donald G. Reinertsen describes the concept of reducing batch size in his book, *The Principles of Product Development Flow* (Celeritas Publishing).

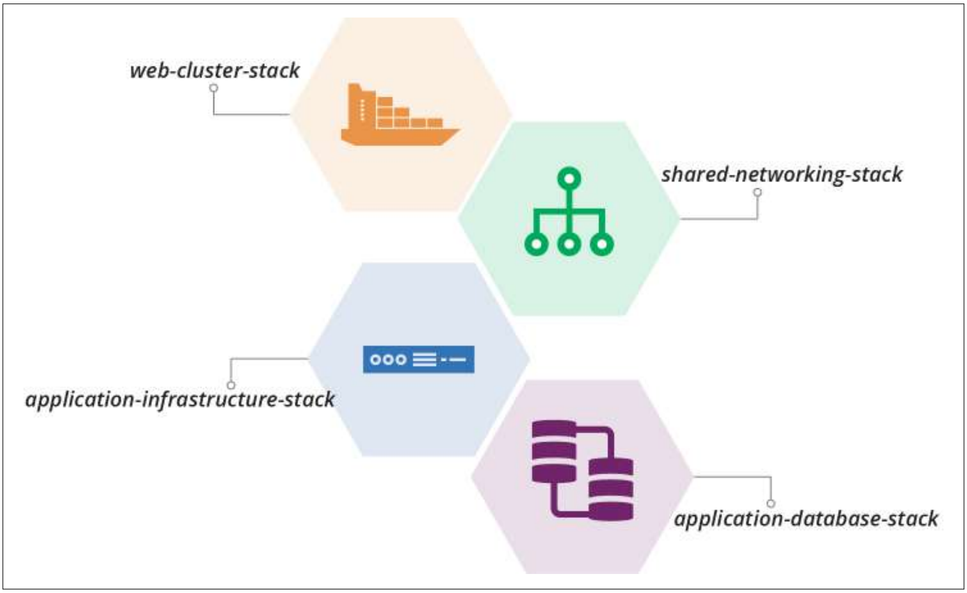


Figure 21-2. Plan to split out multiple stacks

The team will also split its single VLAN into multiple VLANs. The application servers will be spread out across these VLANs for redundancy (see Figure 21-3).

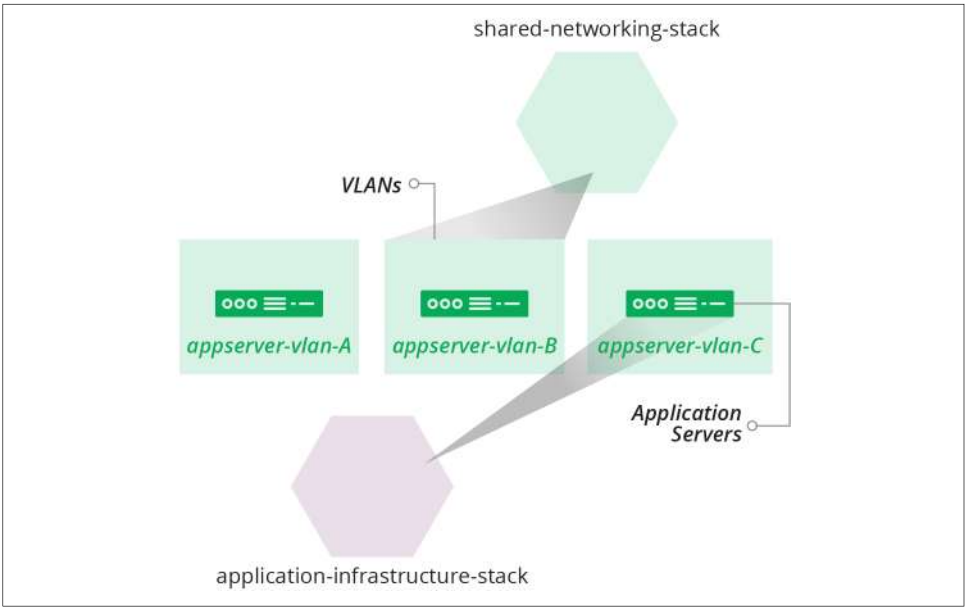


Figure 21-3. Plan to create multiple VLANs

Chapter 17 described the design choices and some implementation patterns for dividing these example stacks. Now we can explore ways to move from one implementation to another in a production system.

Small Changes

The biggest messes I've made in code were when I built up too much work locally before pushing. It's tempting to focus on completing the full piece of work that you have in mind. It's harder to make a small change that only takes you a little further toward that full thing. Implementing large changes as a series of small changes requires a new mindset and new habits.

Fortunately, the software development world has shown the path. I've included many of the techniques that support building systems a piece at a time throughout this book, including TDD, CI, and CD. Progressively testing and delivering code changes using a pipeline, as described in Chapter 8 and referenced throughout the book, is an enabler. You should be able to make a small change to your code, push it, get feedback on whether it works, and put it into production.

Teams that use these techniques effectively push changes very frequently. A single engineer may push changes every hour or so, and each change is integrated into the main codebase and tested for production readiness in a fully integrated system.

People knock around various terms and techniques for making a significant change as a series of small changes:

Incremental

An incremental change is one that adds one piece of the planned implementation. You could build the example ShopSpinner system incrementally by implementing one stack at a time. First, create the shared networking stack. Then, add the web cluster stack. Finally, build the application infrastructure stack.

Iterative

An iterative change makes a progressive improvement to the system. Start building the ShopSpinner system by creating a basic version of all three stacks. Then make a series of changes, each one expanding what the stacks can do.

Walking skeleton

A walking skeleton is a basic implementation of the main parts of a new system that you implement to help validate its general design and structure.[2] People often create a walking skeleton for an infrastructure project along with similar initial implementations of applications that will run on it, so teams can see how

[2] *Growing Object-Oriented Software, Guided by Tests* by Steve Freeman and Nat Pryce (Addison-Wesley) devotes a chapter to walking skeletons.

delivery, deployment, and operations might work. The initial implementation and selection of tools and services for the skeleton are often not the ones that are planned for the long term. For example, you might plan to use a full-featured monitoring solution, but build your walking skeleton using more basic services provided out of the box by your cloud provider.

Refactoring
: Refactoring involves changing the design of a system, or a component of the system, without changing its behavior. Refactoring is often done to pave the way for changes that do change behavior.[3] A refactoring might improve the clarity of the code so that it's easier to change, or it might reorganize the code so that it aligns with planned changes.

Example of Refactoring

The ShopSpinner team decides to decompose its current stack into multiple stacks and stack instances. Its planned implementation includes one stack instance for the container cluster that hosts its web servers and another for shared networking structures. The team will also have a pair of stacks for each service, one for the application server and its associated networking, and one for the database instance for that service (see Figure 21-4).

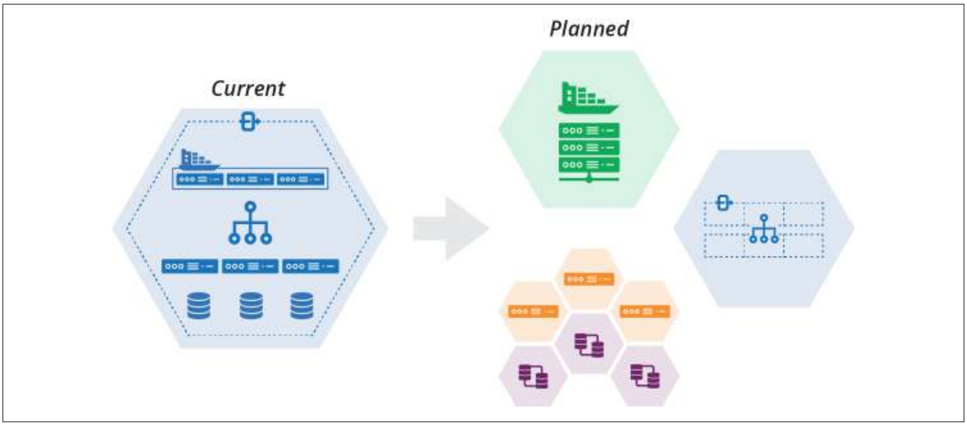


Figure 21-4. Plan to decompose a stack

[3] Kent Beck describes workflows for making "large changes in small, safe steps" in his article "SB Changes". This involves making a series of changes, some that tidy the code to prepare for a behavioral change, others that make behavioral changes. The key point is that each change does one or the other, never both.

The team members also want to replace their container cluster product, moving from a Kubernetes cluster that they deploy themselves onto virtual machines to use Containers as a Service provided by their cloud vendor (see "Application Cluster Solutions" on page 230).

The team decides to incrementally implement its decomposed architecture. The first step is to extract the container cluster into its own stack, and then replace the container product within the stack (Figure 21-5).

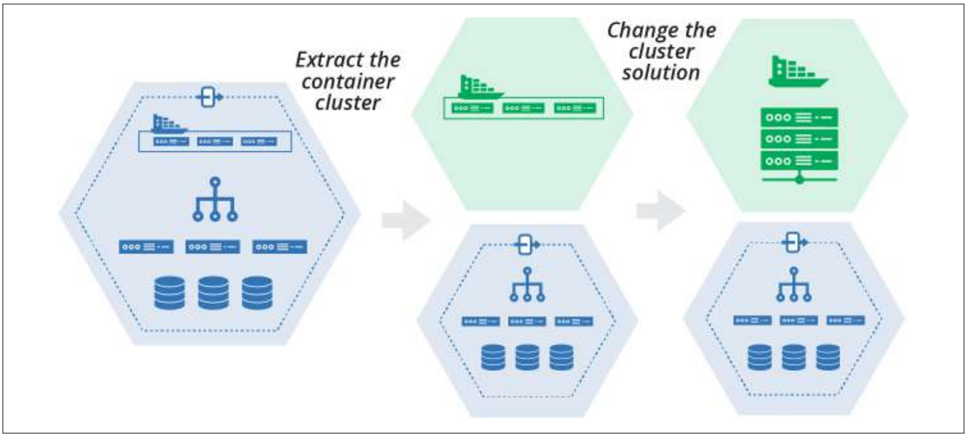


Figure 21-5. Plan to extract and replace the container cluster

This plan is an example of using refactoring to enable a change. Changing the container cluster solution will be easier when it is isolated into its own stack than when it's a part of a larger stack with other infrastructure. When the team members extract the cluster into its own stack, they can define its integration points for the rest of the system. They can write tests and other validations that keep the separation and integration clean. Doing this gives the team confidence that they can safely change the contents of the cluster stack.

Building New

Rather than incrementally changing your existing production system, you could build the new version of your system separately and swap users over when you're finished. Building the new version may be easier if you're drastically changing your system design and implementation. Even so, it's useful to get your work into production as quickly as possible. Extracting and rebuilding one part of your system at a time is less risky than rebuilding everything at once. It also tests and delivers the value of improvements more quickly. So even a substantial rebuild can be done incrementally.

Pushing Incomplete Changes to Production

How can you deliver a significant change to your production system as a series of small, incremental changes while keeping the service working? Some of those small changes may not be useful on their own. It may not be practical to remove existing functionality until the entire set of changes is complete.

"Example of Refactoring" on page 359 showed two incremental steps, extracting a container cluster from one stack into its own stack, and then replacing the cluster solution within the new stack. Each of these steps is large, so would probably be implemented as a series of smaller code pushes.

However, many of the smaller changes would make the cluster unusable on its own. So you need to find ways to make those smaller changes while keeping the existing code and functionality in place. There are different techniques you can use, depending on the situation.

Parallel Instances

The second step of the cluster replacement example starts with the original container solution in its own stack, and ends with the new container solution (see Figure 21-6).

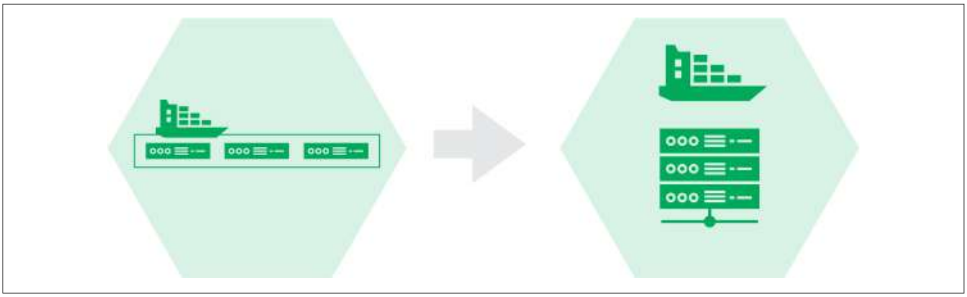


Figure 21-6. Replacing the cluster solution

The existing solution is a packaged Kubernetes distribution called KubeCan.[4] The team is switching to FKS, a managed cluster service provided by its cloud platform.[5] See "Application Cluster Solutions" on page 230 for more on clusters as a service and packaged cluster distributions.

[4] KubeCan is yet another of the fictitious products the fictitious ShopSpinner team prefers.

[5] FKS stands for *Fictional Kubernetes Service*.

It isn't practical to turn the KubeCan cluster into an FKS cluster in small steps. But the team can run the two clusters in parallel. There are a few different ways to run the two different container stacks in parallel with a single instance of the original stack.

One option is to have a parameter for the main stack to choose which cluster stack to integrate with (see Figure 21-7).

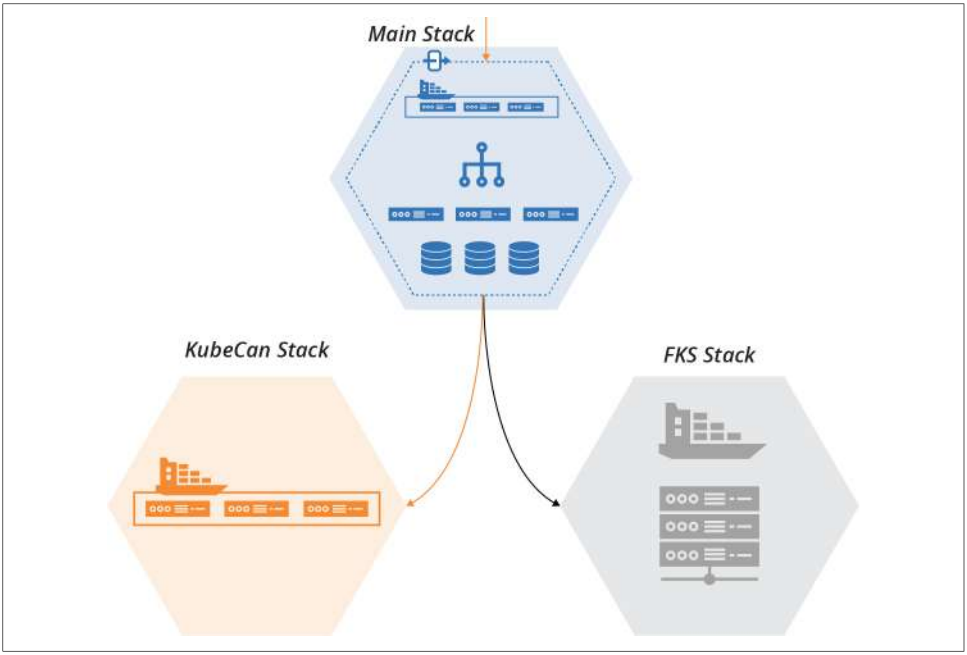


Figure 21-7. One stack is enabled, and one is disabled

With this option, one of the stacks is enabled and will handle live workloads. The second stack is disabled but still present. The team can test the second stack in a fully operational environment, exercise and develop its delivery pipeline and test suite, and integrate it with other parts of the infrastructure in each environment.

Why Extract the Old Container Solution at All?

Considering that we've ended up creating a standalone stack with the new container solution, we arguably could have skipped the step of extracting the old solution into its own stack. We could have just created the new stack with the new solution from scratch.

By extracting the old solution, it's easier to make sure our new solution matches the old solution's behavior. The extracted stack clearly defines how the cluster integrates with other infrastructure. By using the extracted stack in production, we guarantee the integration points are correct.

Adding automated tests and a new pipeline for the extracted stack ensures that we find out immediately when one of our changes breaks something.

If we leave the old cluster solution in the original stack and build the new one separately, swapping it out will be disruptive. We won't know until the end if we've made an incompatible design or implementation decision. It would take time to integrate the new stack with other parts of the infrastructure, and to test, debug, and fix problems.

Another option is to integrate both stacks with the main stack (see Figure 21-8).

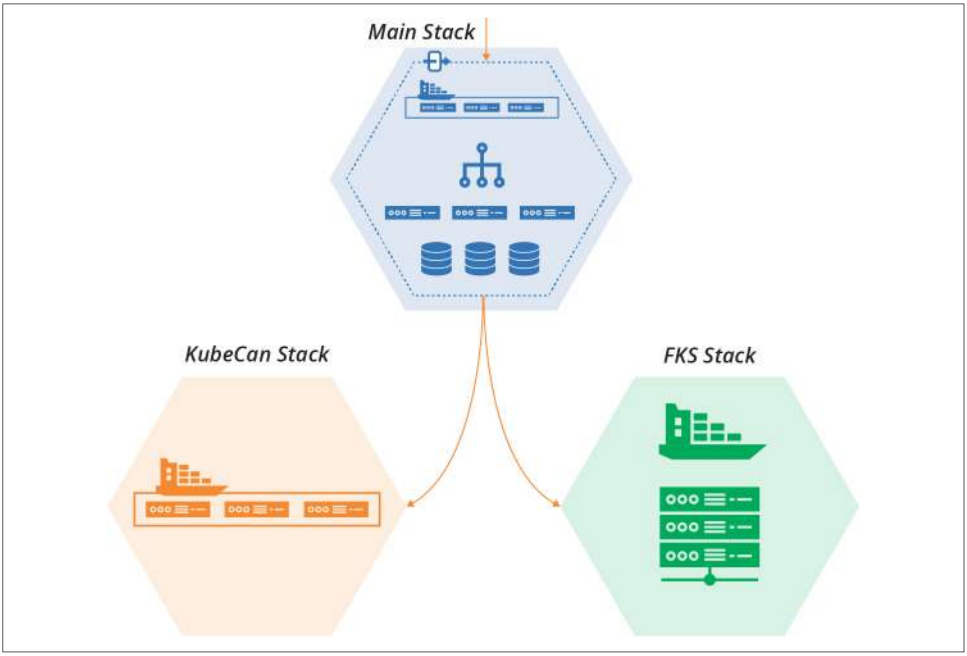


Figure 21-8. Each cluster implementation runs in its own stack instance

With this arrangement, you can direct part of your workload to each of the cluster stacks. You can divide the workload in different ways:

Workload percentage
: Direct part of the workload to each stack. Usually, the old stack handles most of the load at first, with the new stack taking a small percentage to evaluate how well it works. As the new stack beds in, you can dial the load up over time. After the new stack successfully manages 100% of the load and everyone is ready, you can decommission the old stack. This option assumes the new stack has all of the capabilities of the old stack, and that there aren't any issues with data or messaging being split across the stacks.

Service migration
: Migrate services one by one to the new cluster. Workloads in the main stack, such as network connections or messages, are directed to whichever stack instance the relevant service is running on. This option is especially useful when you need to modify service applications to move them to the new stack. It often requires more complex integration, perhaps even between the old and new cluster stacks. This complexity may be justified for migrating a complex service portfolio.[6]

User partitioning
: In some cases, different sets of users are directed to different stack implementations. Testers and internal users are often the first group. They can conduct exploratory tests and exercise the new system before risking "real" customers. In some cases, you might follow this by giving access to customers who opt-in to alpha testing or preview services. These cases make more sense when the service running on the new stack has changes that users will notice.

Running new and old parts of the system conditionally, or in parallel, is a type of branch by abstraction. Progressively shifting portions of a workload onto new parts of a system is a canary release. *Dark launching* describes putting a new system capability into production, but not exposing it to production workloads, so your team can test it.

Backward Compatible Transformations

While some changes might require building and running a new component in parallel with the old one until it's complete, you can make many changes within a component without affecting users or consumers.

[6] This type of complex migration scenario with applications integrated across two hosting clusters is common when migrating a large estate of server-based applications hosted in a data center to a cloud-based hosting platform.

Even when you add or change what you provide to consumer components, you can often add new integration points while maintaining existing integration points unchanged. Consumers can switch to using the new integration points on their own schedule.

For example, the ShopSpinner team plans to change its `shared-networking-stack` to move from a single VLAN to three VLANs (Figure 21-9).

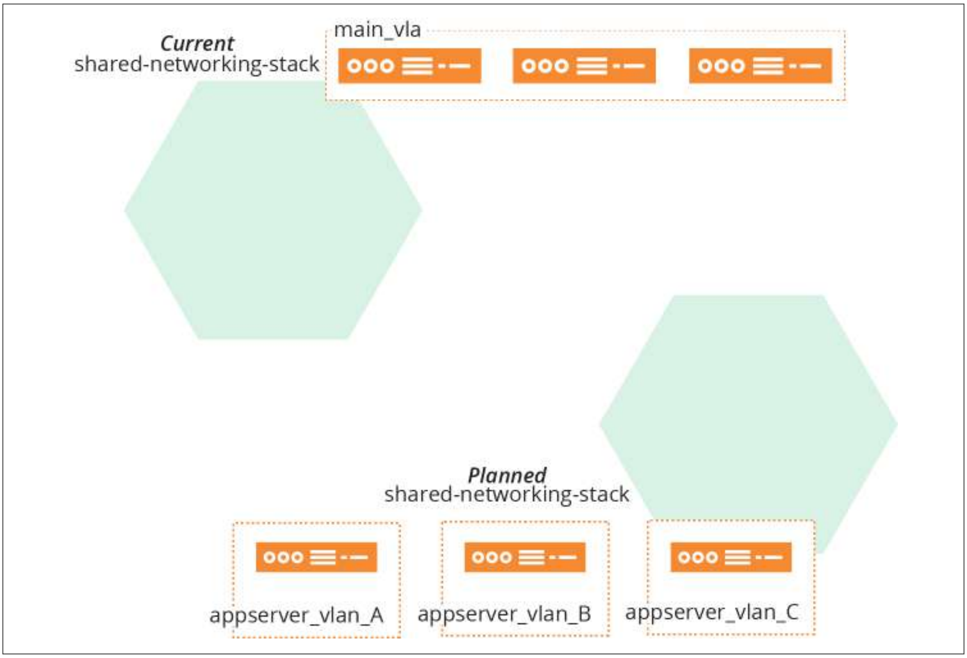


Figure 21-9. Change from single VLAN to three VLANs

Consumer stacks, including `application-infrastructure-stack`, integrate with the single VLAN managed by the networking stack using one of the discovery methods described in "Discovering Dependencies Across Stacks" on page 287. The `shared-networking-stack` code exports the VLAN identifier for its consumer stacks to discover:

```
vlans:
  - main_vlan
    address_range: 10.2.0.0/8

export:
  - main_vlan: main_vlan.id
```

The new version of `shared-networking-stack` creates three VLANs and exports their identifiers under new names. It also exports one of the VLAN identifiers using the old identifier:

```
vlans:
  - appserver_vlan_A
    address_range: 10.1.0.0/16
  - appserver_vlan_B
    address_range: 10.2.0.0/16
  - appserver_vlan_C
    address_range: 10.3.0.0/16

export:
  - appserver_vlan_A: appserver_vlan_A.id
  - appserver_vlan_B: appserver_vlan_B.id
  - appserver_vlan_C: appserver_vlan_C.id
  # Deprecated
  - main_vlan: appserver_vlan_A.id
```

By keeping the old identifier, the modified networking stack still works for consumer infrastructure code. The consumer code should be modified to use the new identifiers, and once all of the dependencies on the old identifier are gone, it can be removed from the networking stack code.

Feature Toggles

When making a change to a component, you often need to keep using the existing implementation until you finish the change. Some people branch the code in source control, working on the new change in one branch and using the old branch in production. The issues with this approach include:

- It takes extra work to ensure changes to other areas of the component, like bugfixes, are merged to both branches.
- It takes effort and resources to ensure both branches are continuously tested and deployed. Alternatively, the work in one branch goes with less rigorous testing, increasing the chances of errors and rework later on.
- Once the change is ready, changing production instances over is more of a "big bang" operation, with a higher risk of failure.

It's more effective to work on changes to the main codebase without branching. You can use feature toggles to switch the code implementation for different environments. Switch to the new code in some environments to test your work, and switch to the existing code for production-line environments. Use a stack configuration parameter (as described in Chapter 7) to specify which part of the code to apply to a given instance.

Once the ShopSpinner team finishes adding VLANs to its `shared-networking-stack`, as described previously, the team needs to change the `application-infrastructure-stack` to use the new VLANs. The team members discover that this isn't as simple a change as they first thought.

The application stack defines application-specific network routes, load balancer VIPs, and firewall rules. These are more complex when application servers are hosted across VLANs rather than in a single VLAN.

It will take the team members a few days to implement the code and tests for this change. This isn't long enough that they feel the need to set up a separate stack, as described in "Parallel Instances" on page 361. But they are keen to push incremental changes to the repository as they work, so they can get continuous feedback from tests, including system integration tests.

The team decides to add a configuration parameter to the application-infrastructure-stack that selects different parts of the stack code depending on whether it should use a single VLAN or multiple VLANs.

This snippet of the source code for the stack uses three variables—appserver_A_vlan, appserver_B_vlan, and appserver_C_vlan—to specify which VLAN to assign to each application server. The value for each of these is set differently depending on the value of the feature toggle parameter, toggle_use_multiple_vlans:

```
input_parameters:
  name: toggle_use_multiple_vlans
  default: false

variables:
  - name: appserver_A_vlan
    value:
      $IF(${toggle_use_multiple_vlans} appserver_vlan_A ELSE main_vlan)
  - name: appserver_B_vlan
    value:
      $IF(${toggle_use_multiple_vlans} appserver_vlan_B ELSE main_vlan)
  - name: appserver_C_vlan
    value:
      $IF(${toggle_use_multiple_vlans} appserver_vlan_C ELSE main_vlan)

virtual_machine:
  name: appserver-${SERVICE}-A
  memory: 4GB
  address_block: ${appserver_A_vlan}

virtual_machine:
  name: appserver-${SERVICE}-B
  memory: 4GB
  address_block: ${appserver_B_vlan}

virtual_machine:
  name: appserver-${SERVICE}-C
  memory: 4GB
  address_block: ${appserver_C_vlan}
```

If the `toggle_use_multiple_vlans` toggle is set to `false`, the `appserver_X_vlan` parameters are all set to use the old VLAN identifier, `main_vlan`. If the toggle is `true`, then each of the variables is set to one of the new VLAN identifiers.

The same toggle parameter is used in other parts of the stack code, where the team works on configuring the routing and other tricky elements.

Advice on Feature Toggles

See "Feature Toggles (aka Feature Flags)" by Pete Hodgson for advice on using feature toggles, and examples from software code. I have a few recommendations to add.

Firstly, minimize the number of feature toggles you use. Feature toggles and conditionals clutter code, making it hard to understand, maintain, and debug. Keep them short-lived. Remove dependencies on the old implementation as soon as possible, and remove the toggles and conditional code. Any feature toggle that remains past a few weeks is probably a configuration parameter.

Name feature toggles according to what they do. Avoid ambiguous names like `new_networking_code`. The earlier example toggle, `toggle_use_multiple_vlans`, tells the reader that it is a feature toggle, to distinguish it from a configuration parameter. It tells the reader that it enables multiple VLANs, so they know what it does.

And the name makes it clear which way the toggle works. Reading a name like `toggle_multiple_vlans`, or worse, `toggle_vlans`, leaves you uncertain whether it enables or disables the multiple VLAN code. This leads to errors, where someone uses the conditional the wrong way around in their code.

Changing Live Infrastructure

These techniques and examples explain how to change infrastructure code. Changing running instances of infrastructure can be trickier, especially when changing resources that are being consumed by other infrastructure.

For example, when the ShopSpinner team applies the change to the `shared-networking-stack` code that replaces the single VLAN with three VLANs, as explained in "Backward Compatible Transformations" on page 364, what happens to resources from other stacks that are assigned to the first VLAN (see Figure 21-10)?

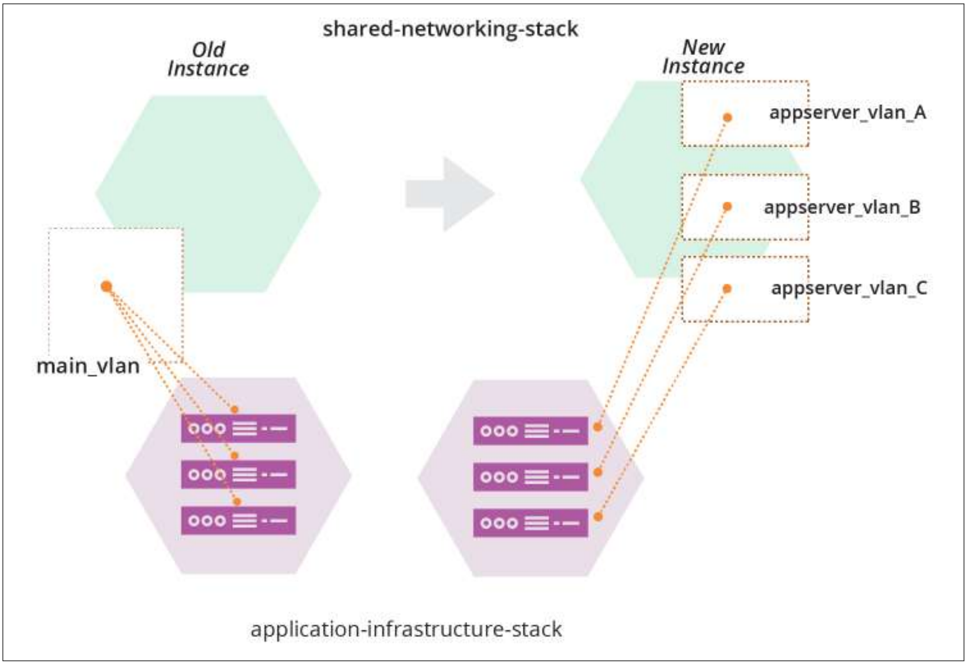


Figure 21-10. Changing networking structures that are in use

Applying the networking code destroys `main_vlan`, which contains three server instances. In a live environment, destroying those servers, or detaching them from the network, will disrupt whatever services they provide.

Most infrastructure platforms will refuse to destroy a networking structure with server instances attached, so the operation would fail. If the code change you apply removes or changes other resources, the operation might implement those changes to the instance, leaving the environment in a halfway state between the old and new versions of your stack code. This will almost always be a bad thing.

There are a few ways to handle this kind of live infrastructure change. One would be to keep the old VLAN, `main_vlan`, and add the two new VLANs, `appserver_vlan_B`, and `appserver_vlan_C`.

Doing this leaves you with three VLANs, as intended, but one of them is named differently from the others. Keeping the existing VLAN probably prevents you from changing other aspects of it, such as its IP address range. Again, you might decide to compromise by keeping the original VLAN smaller than the new ones.

These kinds of compromises are a bad habit, leading to inconsistent systems and code that is confusing to maintain and debug.

You can use other techniques to change live systems and leave them in a clean, consistent state. One is to edit infrastructure resources using infrastructure surgery. The other is to expand and contract infrastructure resources.

Infrastructure Surgery

Some stack management tools, like Terraform, give you access to the data structures that map infrastructure resources to code. These are the same data structures used in the stack data lookup pattern for dependency discovery (see "Pattern: Stack Data Lookup" on page 291).

Some (but not all) stack tools have options to edit their data structures. You can leverage this capability to make changes to live infrastructure.

The ShopSpinner team can use its fictional `stack` tool to edit its stack data structures. Members of the team will use this to change their production environment to use the three new VLANs. They first create a second instance of their `shared-networking-stack` with the new version of their code (see Figure 21-11).

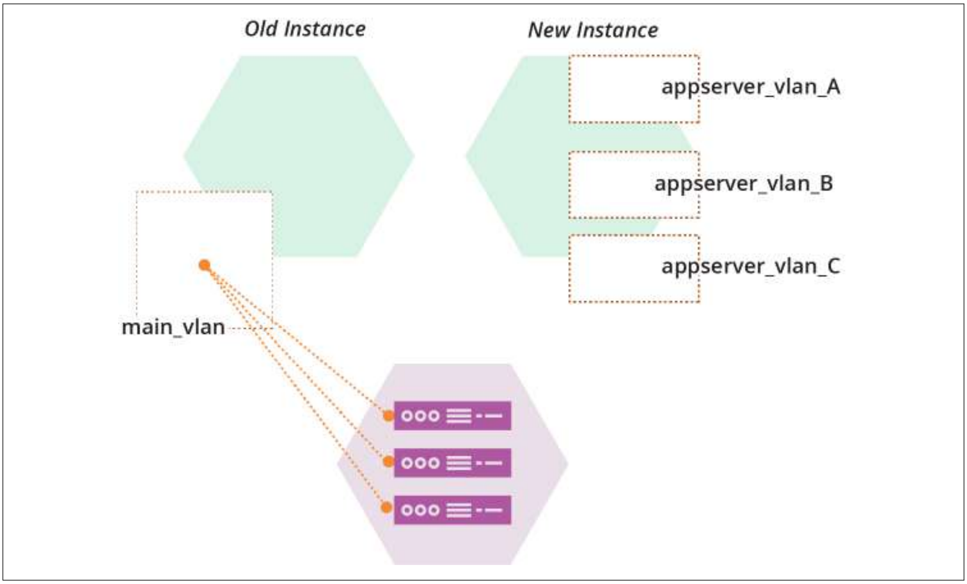


Figure 21-11. Parallel instances of the production networking stack

Each of these three stacks' instances—the `application-infrastructure-stack` instance, and the old and new instances of the `shared-networking-stack`—has a data structure that indicates which resources in the infrastructure platform belong to that stack (see Figure 21-12).

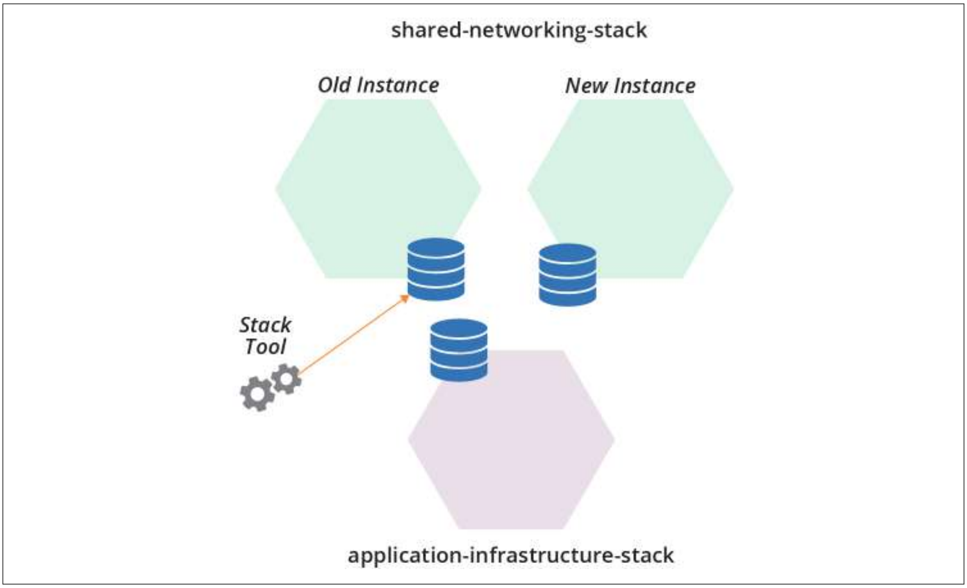


Figure 21-12. Each stack instance has its own stack data structure

The ShopSpinner team will move `main_vlan` from the old stack instance's data structures into the data structure for the new stack instance. The team will then use it to replace `appserver_vlan_A`.

The VLAN in the infrastructure platform won't change in any way, and the server instances will be completely untouched. These changes are entirely a bookkeeping exercise in the stack tool's data structures.

The team runs the stack tool command to move `main_vlan` from the old stack into the new stack instance:

```
$ stack datafile move-resource \
    source-instance=shared-networking-stack-production-old \
    source-resource=main_vlan \
    destination-instance=shared-networking-stack-production-new
Success: Resource moved
```

The next step is to remove `appserver_vlan_A`. How to do this varies depending on the actual stack management tool. The fictional `stack` command happens to make this operation incredibly simple. Running the following command destroys the VLAN in the infrastructure platform and removes it from the data structure file:

```
$ stack datafile destroy-resource \
    instance=shared-networking-stack-production-new \
    resource=appserver_vlan_A
Success: Resource destroyed and removed from the datafile
```

Note that the team members have not removed `appserver_vlan_A` from the stack source code, so if they apply the code to the instance now, it will re-create it. But they won't do that. Instead, they'll run a command to rename the `main_vlan` resource that they moved from the old stack instance:

```
$ stack datafile rename-resource \
    instance=shared-networking-stack-production-new \
    from=main_vlan \
    to=appserver_vlan_A
Success: Resource renamed in the datafile
```

When the team applies the `shared-networking-stack` code to the new instance, it shouldn't change anything. As far as it's concerned, everything in the code exists in the instance.

Note that the ability to edit and move resources between stacks depends entirely on the stack management tool. Most of the tools provided by cloud vendors, at least as of this writing, don't expose the ability to edit stack data structures.[7]

It's easy to make a mistake when editing stack data structures by hand, so the risk of causing an outage is high. You could write a script to implement the commands and test it in upstream environments. But these edits are not idempotent. They assume a particular starting state and running the script can be unpredictable if something differs.

Viewing stack data structures can be useful for debugging, but you should avoid editing them. Arguably, it could be necessary to edit structures to resolve an outage. But the pressure of these situations often makes mistakes even likelier. You should not edit stack data routinely. Any time you resort to editing the structures, your team should follow up with a blameless postmortem to understand how to avoid repeating it.

A safer way to make changes to live infrastructure is to expand and contract.

Expand and Contract

Infrastructure teams use the expand and contract pattern (also called Parallel Change) for changing interfaces without breaking consumers. The idea is that changing a provider's interface involves two steps: change the provider, then change the consumers. The expand and contract pattern decouples these steps.

The essence of the pattern is to first add the new resource while keeping the existing one, then change the consumers over to the new resource, and finally, remove the old

[7] See the terraform mv command and pulumi state command for two examples of tools that do support editing stack data structures.

unused resource. Each of these changes is delivered using a pipeline (see "Infrastructure Delivery Pipelines" on page 119), so it's thoroughly tested.

Making a change by expanding and contracting is similar to a backward compatible transformation (see "Backward Compatible Transformations" on page 364). That technique replaced the old resource and re-pointed the old interface to one of the new resources. However, applying the new code to a running instance would attempt to destroy the old resource, which could either disrupt any consumers attached to it or fail to complete. So a few extra steps are required.

The first step for the ShopSpinner team to use expand and contract for its VLAN change is to add the new VLANs to the `shared-networking-stack`, while leaving the old `main_vlan` in place:

```
vlans:
  - main_vlan
    address_range: 10.2.0.0/8
  - appserver_vlan_A
      address_range: 10.1.0.0/16
  - appserver_vlan_B
      address_range: 10.2.0.0/16
  - appserver_vlan_C
      address_range: 10.3.0.0/16

export:
  - main_vlan: main_vlan.id
  - appserver_vlan_A: appserver_vlan_A.id
  - appserver_vlan_B: appserver_vlan_B.id
  - appserver_vlan_C: appserver_vlan_C.id
```

Unlike the parallel instances technique ("Parallel Instances" on page 361) and infrastructure surgery ("Infrastructure Surgery" on page 370), the ShopSpinner team doesn't add a second instance of the stack, but only changes the existing instance.

After applying this code, the existing consumer instances are unaffected—they are still attached to the `main_vlan`. The team can add new resources to the new VLANs, and can make changes to the consumers to switch them over as well.

How to switch consumer resources to use new ones depends on the specific infrastructure and the platform. In some cases, you can update the definition for the resource to attach it to the new provider interface. In others, you may need to destroy and rebuild the resource.

The ShopSpinner team can't reassign existing virtual server instances to the new VLANs. However, the team can use the expand and contract pattern to replace the servers. The `application-infrastructure-stack` code defines each server with a static IP address that routes traffic to the server:

```
virtual_machine:
  name: appserver-${SERVICE}-A
  memory: 4GB
  vlan: external_stack.shared_network_stack.main_vlan

static_ip:
  name: address-${SERVICE}-A
  attach: virtual_machine.appserver-${SERVICE}-A
```

The team's first step is to add a new server instance attached to the new VLAN:

```
virtual_machine:
  name: appserver-${SERVICE}-A2
  memory: 4GB
  vlan: external_stack.shared_network_stack.appserver_vlan_A

virtual_machine:
  name: appserver-${SERVICE}-A
  memory: 4GB
  vlan: external_stack.shared_network_stack.main_vlan

static_ip:
  name: address-${SERVICE}-A
  attach: virtual_machine.appserver-${SERVICE}-A
```

The first `virtual_machine` statement in this code creates a new server instance named `appserver-${SERVICE}-A2`. The team's pipeline delivers this change to each environment. The new server instance isn't used at this point, although the team can add some automated tests to prove that it's running OK.

The team's next step is to switch user traffic to the new server instance. The team makes another change to the code, modifying the `static_ip` statement:

```
virtual_machine:
  name: appserver-${SERVICE}-A2
  memory: 4GB
  vlan: external_stack.shared_network_stack.appserver_vlan_A

virtual_machine:
  name: appserver-${SERVICE}-A
  memory: 4GB
  vlan: external_stack.shared_network_stack.main_vlan

static_ip:
  name: address-${SERVICE}-A
  attach: virtual_machine.appserver-${SERVICE}-A2
```

Pushing this change through the pipeline makes the new server active, and stops traffic to the old server. The team can make sure everything is working, and easily roll the change back to restore the old server if something goes wrong.

Once the team has the new server working OK, it can remove the old server from the stack code:

```
virtual_machine:
  name: appserver-${SERVICE}-A2
  memory: 4GB
  vlan: external_stack.shared_network_stack.appserver_vlan_A

static_ip:
  name: address-${SERVICE}-A
  attach: virtual_machine.appserver-${SERVICE}-A2
```

Once this change has been pushed through the pipeline and applied to all environments, `application-infrastructure-stack` no longer has a dependency on `main_vlan` in `shared-networking-stack`. After all consumer infrastructure has changed over, the ShopSpinner team can remove `main_vlan` from the provider stack code:

```
vlans:
  - appserver_vlan_A
      address_range: 10.1.0.0/16
  - appserver_vlan_B
      address_range: 10.2.0.0/16
  - appserver_vlan_C
      address_range: 10.3.0.0/16

export:
  - appserver_vlan_A: appserver_vlan_A.id
  - appserver_vlan_B: appserver_vlan_B.id
  - appserver_vlan_C: appserver_vlan_C.id
```

The VLAN change is complete, and the last remnants of `main_vlan` have been swept away.[8]

Zero Downtime Changes

Many of the techniques described in this chapter explain how to implement a change incrementally. Ideally, you'd like to apply the change to existing infrastructure without disrupting the services it provides. Some changes will inevitably involve destroying resources, or at least changing them in a way that might interrupt service. There are a few common techniques for handling these situations.

Blue-green changes

A blue-green change involves creating a new instance, switching usage to the new instance, and then removing the old instance. This is conceptually similar to expand

[8] Don't worry, fear will keep the local systems in line.

and contract ("Expand and Contract" on page 372), which adds and removes resources within an instance of a component such as a stack. It's a key technique for implementing immutable infrastructure (see "Immutable Infrastructure" on page 351).

Blue-green changes require a mechanism to handle the switchover of a workload from one instance to another, such as a load balancer for network traffic. Sophisticated implementations allow the workload to "drain," allocating new work to the new instance, and waiting for all work on the old instance to complete before destroying it. Some automated server clustering and application clustering solutions provide this as a feature, for example, enabling "rolling upgrades" to instances in a cluster.

Blue-green is implemented with static infrastructure by maintaining two environments. One environment is live at any point in time, the other being ready to take the next version. The names *blue* and *green* emphasize that these are equal environments that take turns at being live, rather than a primary and secondary environment.

I worked with one organization that implemented blue-green data centers. A release involved switching workloads for its entire system from one data center to the other. This scale became unwieldy, so we helped the organization to implement deployment at a smaller scale, so it would do a blue-green deployment only for the specific service that was being upgraded.

Continuity

Chapter 1 discussed the contrast between traditional, "Iron Age" approaches to managing infrastructure and modern, "Cloud Age" approaches (see "From the Iron Age to the Cloud Age" on page 2). When we worked more with physical devices and managed them manually, the cost of making changes was high.

The cost of making a mistake was also high. When I provisioned a new server without enough memory, it took me a week or more to order more RAM, take it to the data center, power the server down and pull it from the rack, open it up and add the extra RAM, then rerack and boot the server again.

The cost of making a change with Cloud Age practices is much lower, as are the cost and time needed to correct a mistake. If I provision a server without enough memory, it only takes a few minutes to correct it by editing a file and applying it to my virtual server.

Iron Age approaches to continuity emphasize prevention. They optimize for MTBF, *Mean Time Between Failure*, by sacrificing speed and frequency of change. Cloud Age approaches optimize for MTTR, *Mean Time to Recover*. Although even some enthusiasts for modern methods fall into the trap of thinking that focusing on MTTR means sacrificing MTBF, this is untrue, as explained in "Objection: "We must choose between speed and quality"" on page 7. Teams who focus on the four key metrics (speed and frequency of changes, MTTR, and change failure rate, as described in "The Four Key Metrics" on page 9) achieve strong MTBF as a side effect. The point is not to "move fast and break things," but rather, to "move fast and fix things."

There are several elements to achieving continuity with modern infrastructure. Prevention, the focus of Cloud Age change management practices, is essential, but cloud infrastructure and automation enable the use of more effective Agile engineering practices for reducing errors. Additionally, we can exploit new technologies and practices to recover and rebuild systems to achieve higher levels of continuity than imaginable before. And by continuously exercising the mechanisms that deliver changes and recover systems, we can ensure reliability and readiness for a wide variety of disasters.

Continuity by Preventing Errors

As mentioned, Iron Age approaches to governing changes were mainly preventative. Because the cost of fixing a mistake was high, organizations invested heavily in preventing mistakes. Because changes were mainly manual, prevention involved restricting who could make changes. People needed to plan and design changes in detail, and other people exhaustively reviewed and discussed each change. The idea was that having more people take more time to consider a change ahead of time would catch mistakes.

One problem with this approach is the gap between design documents and implementation. Something that looks simple in a diagram can be complicated in reality. People make mistakes, especially when carrying out substantial, infrequent upgrades. The result is that traditional low-frequency, highly planned large batch change operations have a high failure rate, and often lengthy recovery times.

The practices and patterns described throughout this book aim to prevent errors without sacrificing the frequency and speed of change. Changes defined as code represent their implementation better than any diagram or design document possibly can. Continuously integrating, applying, and testing changes as you work proves their readiness for production. Using a pipeline to test and deliver changes ensures steps aren't skipped, and enforces consistency across environments. This reduces the likelihood of failures in production.

The core insight of Agile software development and Infrastructure as Code is to flip the attitude toward change. Rather than fearing change and doing it as little as possible, you can prevent errors by making changes frequently. The only way to get better at making changes is to make changes frequently, continuously improving your systems and processes.

Another key insight is that as systems become more complex, our ability to replicate and accurately test how code will behave in production shrinks. We need to stay aware of what we can and cannot test before production, and how to mitigate risks by improving our visibility of production systems (see "Testing in Production" on page 125).

Continuity by Fast Recovery

The practices described so far in this chapter can reduce downtime. Limiting the size of changes, making them incrementally, and testing changes before production can lower your change failure rate. But it's unwise to assume that errors can be prevented entirely, so we also need to be able to recover quickly and easily.

The practices advocated throughout this book make it easy to rebuild any part of your system. Your system is composed of loosely coupled components, each defined as idempotent code. You can easily repair, or else destroy and rebuild, any component instance by reapplying its code. You'll need to ensure the continuity of data hosted on a component if you rebuild it, which is discussed in "Data Continuity in a Changing System" on page 382.

In some cases, your platform or services can automatically rebuild failed infrastructure. Your infrastructure platform or application runtime destroys and rebuilds individual components when they fail a health check. Continuously applying code to instances ("Apply Code Continuously" on page 350) automatically reverts any deviation from the code. You can manually trigger a pipeline stage ("Infrastructure Delivery Pipelines" on page 119) to reapply code to a broken component.

In other failure scenarios, these systems might not automatically fix a problem. A compute instance might malfunction in such a way that it still passes its health check. An infrastructure element might stop working correctly while still matching the code definition, so reapplying the code doesn't help.

These scenarios need some kind of additional action to replace failed components. You might flag a component so that the automated system considers it to be failed, and destroys and replaces it. Or, if recovery uses a system that reapplies code, you might need to destroy the component yourself and allow the system to build a new instance.

For any failure scenario that needs someone to take an action, you should make sure to have tools, scripts, or other mechanisms that are simple to execute. People shouldn't need to follow a sequence of steps; for example, backing up data before destroying an instance. Instead, they should invoke an action that carries out all of the required steps. The goal is that, in an emergency, you don't need to think about how to correctly recover your system.

Continuous Disaster Recovery

Iron Age infrastructure management approaches view *disaster recovery* as an unusual event. Recovering from the failure of static hardware often requires shifting workloads to a separate set of hardware kept on standby.

Many organizations test their recovery operation infrequently—every few months at best, in some cases once a year. I've seen plenty of organizations that rarely test their failover process. The assumption is that the team will work out how to get its backup system running if it ever needs to, even if it takes a few days.

Continuous disaster recovery leverages the same processes and tools used to provision and change infrastructure. As described earlier, you can apply your infrastructure code to rebuild failed infrastructure, perhaps with some added automation to avoid data loss.

One of the principles of Cloud Age infrastructure is to assume systems are unreliable ("Principle: Assume Systems Are Unreliable" on page 14). You can't install software onto a virtual machine and expect it will run there as long as you like. Your cloud vendor might move, destroy, or replace the machine or its host system for maintenance, security patches, or upgrades. So you need to be ready to replace the server if needed.[9]

Treating disaster recovery as an extension of normal operations makes it far more reliable than treating it as an exception. Your team exercises your recovery process and tools many times a day as it works on infrastructure code changes and system updates. If someone makes a change to a script or other code that breaks provisioning or causes data loss on an update, it usually fails in a pipeline test stage, so you can quickly fix it.

Chaos Engineering

Netflix was a pioneer of continuous disaster recovery and Cloud Age infrastructure management.[10] Its Chaos Monkey and Simian Army took the concept of continuous disaster recovery a step further, proving the effectiveness of its system's continuity mechanisms by injection error into production systems. This evolved into the field of

[9] Compute instances have become more reliable since the early days of cloud computing. Originally, you couldn't shut an AWS EC2 instance down and boot it later—when the instance stopped, it was gone forever. The ephemeral nature of compute forced cloud users to adopt new practices to run reliable services on unreliable infrastructure. This was the origin of Infrastructure as Code, chaos engineering, and other Cloud Age infrastructure practices.

[10] See "5 Lessons We've Learned Using AWS", written in 2010, for insight into early lessons of building highly reliable services on public cloud at scale.

Chaos Engineering, “The discipline of experimenting on a system to build confidence in the system’s capability.”[11]

To be clear, chaos engineering is not about irresponsibly causing production service outages. Practitioners experiment with specific failure scenarios that their system is expected to handle. These are essential production tests that prove that detection and recovery mechanisms work correctly. The intention is to gain fast feedback when some change to the system has a side effect that interferes with these mechanisms.

Planning for Failure

Failures are inevitable. While you can and should take steps to make them less likely, you also need to put measures in place to make them less harmful and easier to handle.

A team holds a failure scenario mapping workshop to brainstorm types of failures that may occur, and then plan mitigations.[12] You can create a map of likelihood and impact of each scenario, build a list of actions to address the scenarios, and then prioritize these into your team’s backlog of work appropriately.

For any given failure scenario, there are several conditions to explore:

Causes and prevention

What situations could lead to this failure, and what can you do to make them less likely? For example, a server might run out of disk space when usage spikes. You could address this by analyzing disk usage patterns and expanding the disk size, so there is enough for the higher usage levels. You might also implement automated mechanisms to continuously analyze usage levels and make predictions, so disk space can be added preemptively if patterns change. A further step would be to automatically adjust disk capacity as usage increases.

Failure mode

What happens when the failure occurs? What can you do to reduce the consequence without human intervention? For example, if a given server runs out of disk space, the application running on it might accept transactions but fail to record them. This could be very harmful, so you might modify the application to stop accepting transactions if it can’t record them to disk. In many cases, teams don’t actually know what will happen when a given error occurs. Ideally, your failure mode keeps the system fully operational. For example, when an application stops responding, your load balancer may stop directing traffic to it.

[11] This definition is from the principles of chaos engineering web site.

[12] See also “Failure mode and effects analysis”.

Detection

How will you detect when the failure occurs? What can you do to detect it faster, maybe even beforehand? You might detect that the disk has run out of space when the application crashes and a customer calls your CEO to complain. Better to receive a notification when the application crashes. Better still to be notified when disk space runs low, before it actually fills up.

Correction

What steps do you need to take to recover from the failure? In some scenarios, as described earlier, your systems may automatically correct the situation, perhaps by destroying and rebuilding an unresponsive application instance. Others require several steps to repair and restart a service.

If your system automatically handles a failure scenario, such as restarting an unresponsive compute instance, you should consider the deeper failure scenario. Why did the instance become unresponsive in the first place? How will you detect and correct an underlying issue? It shouldn't take you several days to realize that application instances are being recycled every few minutes.

Failure planning is a continuous process. Whenever you have an incident with your system, including in a development or test environment, your team should consider whether there is a new failure scenario to define and plan for.

You should implement checks to prove your failure scenarios. For example, if you believe that when your server runs out of disk space the application will stop accepting transactions, automatically add new server instances, and alert your team, you should have an automated test that exercises this scenario. You could test this in a pipeline stage (availability testing as described in "What Should We Test with Infrastructure?" on page 108) or using a chaos experiment.

Incrementally Improving Continuity

It's easy to define ambitious recovery measures, where your system gracefully handles every conceivable failure without interrupting service. I've never known a team that had the time and resources to build even half of what it would like.

When mapping failure scenarios and mitigations, you can define an incremental set of measures you could implement. Break them into separate implementation stories and prioritize them on your backlog, based on the likelihood of the scenario, potential damage, and cost to implement. For example, although it would be nice to automatically expand the disk space for your application when it runs low, getting an alert before it runs out is a valuable first step.

Data Continuity in a Changing System

Many Cloud Age practices and techniques for deploying software and managing infrastructure cheerfully recommend the casual destruction and expansion of resources, with only a hand wave to the problem of data. You can be forgiven for thinking that DevOps hipsters consider the whole idea of data to be a throwback to the Iron Age—a proper twelve-factor application is stateless, after all. Most systems out in the real world involve data, and people can be touchingly attached to it.

Data can present a challenge when incrementally changing a system, as described in "Pushing Incomplete Changes to Production" on page 361. Running parallel instances of storage infrastructure may create inconsistencies or even corrupt your data. Many approaches to incrementally deploying changes rely on being able to roll them back, which might not be possible with data schema changes.

Dynamically adding, removing, and rebuilding infrastructure resources that host data is particularly challenging. However, there are ways to manage it, depending on the situation. Some approaches include lock, segregate, replicate, and reload.

Lock

Some infrastructure platforms and stack management tools allow you to lock specific resources so they won't be deleted by commands that would otherwise destroy them. If you specify this setting for a storage element, then the tool will refuse to apply changes to this element, allowing team members to make the change manually.

There are a few issues with this, however. In some cases, if you apply a change to a protected resource, the tool may leave the stack in a partially modified state, which can cause downtime for services.

But the fundamental problem is that protecting some resources from automated changes encourages manual involvement in changes. Manual work invites manual mistakes. It's much better to find a way to automate a process that changes your infrastructure safely.

Segregate

You can segregate data by splitting the resources that host them from other parts of the system; for example, by making a separate stack (an example of this is given in "Pattern: Micro Stack" on page 62). You can destroy and rebuild a compute instance with impunity by detaching and reattaching its disk volume.

Keeping data in a database gives even more flexibility, potentially allowing you to add multiple compute instances. You still need a data continuity strategy for the stack that hosts the data, but it narrows the problem's scope. You may be able to offload data continuity completely using a hosted DBaaS service.

Replicate

Depending on the data and how it's managed, you may be able to replicate it across multiple instances of infrastructure. A classic example is a distributed database cluster that replicates data across nodes.

With the right replication strategy, data is reloaded to a newly rebuilt node from other nodes in the cluster. This strategy fails if too many nodes are lost, which can happen with a major hosting outage. So this approach works as the first line of defense, with another mechanism needed for harder failure scenarios.

Reload

The best-known data continuity solution is backing up and restoring data from more reliable storage infrastructure. When you rebuild the infrastructure that hosts data, you first back the data up. You reload the data to the new instance after creating it. You may also take periodic backups, which you can reload in recovery scenarios, although you will lose any data changes that occurred between the backup and the recovery. This can be minimized and possibly eliminated by streaming data changes to the backup, such as writing a database transaction log.

Cloud platforms provide different storage services, as described in "Storage Resources" on page 29, with different levels of reliability. For example, object storage services like AWS S3 usually have stronger guarantees for the durability of data than block storage services like AWS EBS. So you could implement backups by copying or streaming data to an object storage volume.

You should automate the process for not only backing up data but also for recovering it. Your infrastructure platform may provide ways to easily do this. For example, you can automatically take a snapshot of a disk storage volume before applying a change to it.

You may be able to use disk volume snapshots to optimize the process of adding nodes to a system like a database cluster. Rather than creating a new database node with an empty storage volume, attaching it to a clone of another node's disk might make it faster to synchronize and bring the node online.

"Untested backups are the same as no backups" is a common adage in our industry. You're already using automated testing for various aspects of your system, given that you're following Infrastructure as Code practices. So you can do the same thing with your backups. Exercise your backup restore process in your pipeline or as a chaos experiment, whether in production or not.

Mixing Data Continuity Approaches

The best solution is often a combination of segregate, replicate, and reload. Segregating data creates room to manage other parts of the system more flexibly. Replication keeps data available most of the time. And reloading data is a backstop for more extreme situations.

Conclusion

Continuity is often given short shrift by advocates of modern Cloud Age infrastructure management practices. The most familiar approaches to keeping systems running reliably are based on the Iron Age premise that making changes is expensive and risky. These approaches tend to undermine the benefits of cloud, Agile, and other approaches that focus on a fast pace of change.

I hope this chapter has explained how to leverage Cloud Age thinking to make systems more reliable, not despite a fast pace of change, but because of it. You can leverage the dynamic nature of modern infrastructure platforms, and implement the rigorous focus on testing and consistency that comes from Agile engineering practices. The result is a high level of confidence that you can continuously deliver improvements to your system and exploit failures as an opportunity to learn and improve.

Index

About the Author

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

Kief ran his first online system, a bulletin board system (BBS) in Florida in the early 1990s. He later enrolled in an 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, living with his wife, son, and cat.

Most of the companies Kief worked for before ThoughtWorks were post-startups, looking to build and scale. The titles he's been given or self-applied include Software Developer, Systems Administrator, Deputy Technical Director, R&D Manager, Hosting Manager, Technical Lead, Technical Architect, Consultant, and Director of Cloud Engineering.

Colophon

The animal on the cover of *Infrastructure as Code* is Rüppell's vulture (*Gyps rueppellii*), native to 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.

Color illustration by Karen Montgomery, based on a black and white engraving from Cassell's *Natural History*. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.