O'REILLY®

Compliments of NGINX

PREVIEW EDITION

Infrastructure as Code

MANAGING SERVERS IN THE CLOUD

Kief Morris

Deploy your apps with code

Flawless application delivery with NGINX Plus

- Advanced load balancing and automated routing
- On-the-fly reconfiguration for scalable service discovery
- Application-aware health checks and container monitoring
- Content caching for better availability and performance
- Access controls and rate limiting to secure your applications

Learn more at:
nginx.com/code

This Preview Edition of *Infrastructure as Code*, *Chapters 1 and 9*, is a work in progress. The final book is currently scheduled for release in June 2016 and will be available at *oreilly.com* and other retailers once it is published.

Infrastructure as Code

Managing servers in the Cloud

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://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com .

Editor: Brian Anderson
Production Editor: FILL IN PRODUCTION EDITOR

Copyeditor: FILL IN COPYEDITOR
Proofreader: FILL IN PROOFREADER
Indexer: FILL IN INDEXER
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

October 2015: First Edition

Revision History for the First Edition
2015-08-03: First Early Release
2015-11-02: Second Early Release
2015-12-15: Third Early Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491924358* for release details.

978-1-491-92435-8

[FILL IN]

Table of Contents

Preface

Feedback wanted!

The book you're looking at now is still an evolving draft. I would love to hear your comments, suggestions, and puzzles. I'm mostly interested in your thoughts on the content; I'm not worried about the poor grammar, spelling, and mechanical issues (links, etc.), which will get professional attention in due time.

Please consider filling out a brief survey to help me understand where I should be focusing my efforts on improving this book: *http://bit.ly/1DOI4z1*

Send your general comments and feedback to *feedback@infrastructure-as-code.com*.

Thanks for reading!

Infrastructure teams and software development teams are increasingly building and managing infrastructure using automated tools which have been described as "Infrastructure as Code". These tools expect users to define their servers, networking, and other elements of an infrastructure in files which are modeled after software source code. The tools then compile and interpret these files to decide what action to take.

This class of tool has grown naturally with the DevOps movement[1]. The DevOps movement is mainly about culture and collaboration between software developers and software operations people. Tooling that manages infrastructure based on a software development paradigm has naturally helped to bring these communities together.

Managing infrastructure as code is very different from classic infrastructure management. I've met many teams who have struggled to work out how to make this shift.

1 Andrew Clay Shafer and Patrick Debois triggered the DevOps movement with a talk at the Agile 2008 conference. The movement grew, mainly driven by the series of DevOpsDays conferences organized by Debois.

But ideas, patterns, and practices for using these tools effectively have been scattered across conference talks, blog posts, and articles. I've been waiting for someone to write a book to pull these ideas together into a single place. I haven't seen any sign of this, so finally tool matters into my own hands. You now hold the results of this in your own hands!

How I learned to stop worrying and love the cloud

I set up my first server, a dialup BBS[2], in 1992. This started me on a journey that led to Unix systems administration, and then to building and running hosted software systems (before we called it SaaS - "Software as a Service") for various companies, from startups to enterprises. I've been on a journey to Infrastructure as Code the entire time, before I'd ever heard the term.

Things came to a head with virtualization. The story of my stumbling adoption of virtualization and cloud may be familiar. I believe it illustrates the role that Infrastructure as Code has to play in modern IT operations.

My first virtual server farm

I was thrilled when my team got the budget to buy a pair of beefy HP rack servers and licenses for VMware ESX Server back in 2007.

We had around twenty 1U and 2U servers in our office's server racks, named after fruits (Linux servers) and berries (Windows database servers), running test environments for our development teams. Stretching these servers to test various releases, branches, and high priority proof of concept applications was a way of life. Network services like DNS, file servers, and email were crammed onto servers running multiple application instances, web servers, and database servers.

So we were sure these new virtual servers would change our lives. We could cleanly split each of these services onto their own virtual machines (VMs), and the ESX hypervisor software would help us to squeeze the most out of the multi-core server machines and gobs of RAM we'd allocated. We could easily duplicate servers to create new environments, and archive those servers that weren't needed onto disk, confident they could be restored in the future if needed.

Those servers did change our lives. But although many of our old problems went away, we discovered new ones, and we had to learn completely different ways of thinking about our infrastructure.

2 A BBS is a Bulletin Board System.

Virtualization made creating and managing servers much easier. The flip side of this was that we ended up creating far more servers than we could have imagined. The product and marketing people were delighted that we could give them a new environment to demo things in well under a day, rather than needing them to find budget and then wait a few weeks for us to order and set up hardware servers.

The sorcerer's apprentice

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

But we found ourselves a bit like Mickey Mouse in "The Sorcerer's Apprentice" from *Fantasia*. We spawned virtual servers, then more, then ever more. They overwhelmed us. When something broke, we tracked down the VM and fixed whatever was wrong with it, but we couldn't keep track of what changes we'd made where.

> Well, a perfect hit! See how he is split! Now there's hope for me, and I can breathe free!
>
> Woe is me! Both pieces come to life anew, now, to do my bidding I have servants two! Help me, o great powers! Please, I'm begging you!
>
> —Excerpted from Brigitte Dubiel's translation of "Der Zauberlehrling" ("The Sorcerer's Apprentice") by Johann Wolfgang von Goethe

As new updates came out, to operating systems, web servers, app servers, database servers, JVMs, and various other software packages, we would struggle to upgrade them across all of our systems. We would upgrade some, but on others the upgrades broke things, and we didn't have time to stomp out every incompatibility. Over time we ended up with many combinations of versions of things strewn across hundreds of servers.

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

The problem was that, although Puppet (and Chef and the others) should have been set up and left running unattended across all of our servers, we couldn't trust it. Our servers were just too different. We would write manifests to configure and manage a particular application server. But when we ran it against another, theoretically similar app server, we found that different versions of Java, application software, and OS

components would cause the Puppet run to fail, or worse, break the application server.

So we ended up using Puppet ad-hoc. We could safely run it against new VMs, although we might need to make some tweaks after it ran. We would write manifests for a specific task, and then run them against servers one at a time, carefully checking the result and making fixes as needed.

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

Cloud from scratch

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

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

The new regime wasn't painless. We had to learn new habits, and we had to find ways of coping with the challenges of a highly automated infrastructure. As the members of the team moved on to other organizations and got involved with communities such as DevOpsDays, we learned and grew. Over time, we reached the point where we were habitually working with automated infrastructures with hundreds of servers, with much less effort and headache than we had been in our Sorcerer's Apprentice days.

Joining ThoughtWorks was an eye opener for me. The development teams I worked with were passionate about using XP engineering practices like Test Driven Development (TDD), Continuous Integration (CI) and Continuous Delivery (CD). Since I had already learned to manage infrastructure scripts and configuration files in source control systems, it was natural to apply these rigorous development and testing approaches to them.

Working with ThoughtWorks has also brought me into contact with many IT Operations teams, most of whom are using virtualization, cloud, and automation tools to handle a variety of challenges. Working with them to share and learn new ideas and techniques has been a fantastic experience.

Why I'm writing this book

I've run across many teams who are in the same place I was a few years ago, people who are using cloud, virtualization, and automation tools, but haven't got it all running as smoothly as they know they could.

Much of the challenge is time. Day to day life for systems administrators is coping with a never-ending flow of critical work. Fighting fires, fixing problems, and setting up new business critical projects doesn't leave much time to work on the fundamental improvements that will make the routine work easier.

My hope is that this book provides a practical vision for how to manage IT infrastructure, with techniques and patterns that teams can try and use. I will avoid the details of configuring and using specific tools, so that the content will be useful for working with different tools, including ones that may not exist yet. On the other hand, I will use examples from current tools to illustrate points I make.

The Infrastructure as Code approach is essential for managing cloud infrastructure of any real scale or complexity, but it's not exclusive to organizations using public cloud providers. The techniques and practices in this book have proven effective in virtualized environments, and even for "bare metal" servers that aren't virtualized.

Infrastructure as Code is one of the cornerstones of DevOps. It is the "A" in "http://itrevolution.com/devops-culture-part-1/[CAMS]" - Culture, Automation, Measurement, and Sharing.

Who this book is for

This book is for people who work with IT infrastructure, particularly at the level of managing servers and collections of servers. You may be a system administrator, infrastructure engineer, team lead, architect, or a manager with technical interest. You might also be a software developer who wants to build and use infrastructure.

I'm assuming you have some exposure to virtualization or IaaS (Infrastructure as a Service) cloud, so you know how to create a server, and the concepts of configuring operating systems. You've probably at least played with configuration automation software like Ansible, Chef, or Puppet.

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

What tools are covered

This book doesn't offer instructions in using specific scripting languages or tools. There are code examples from specific tools, but these are intended to illustrate con-

cepts and approaches, rather than to provide instruction. This book should be helpful to you regardless of whether you use Chef on OpenStack, Puppet on AWS, Ansible on bare metal, or a completely different stack.

The specific tools that I do mention are ones which I'm aware of, and which seem to have a certain amount of traction in the field. But this is a constantly changing landscape, and there are plenty of other relevant tools.

The tools I use in examples tend to be ones with which I am familiar enough to write examples that demonstrate the point I'm trying to make. For example, I use Terraform for examples of infrastructure definitions because it has a nice, clean syntax, and I've used it on multiple projects. Many of my examples use Amazon's AWS cloud platform because it is likely to be the most familiar to readers.

How to read this book

Read chapter 1, or at least skim it, to understand the terms this book uses and the principles this book advocates. You can then use this to decide which parts of the book to focus on.

If you're new to this kind of automation, cloud, and infrastructure orchestration tooling, then you'll want to focus on Part I, and then move on to Part II. Get comfortable with those topics before coming back to Part III.

If you've been using the types of automation tools described here, but don't feel like you're using them the way they're intended after reading chapter 1, then you may want to skip or skim the rest of Part I. Focus on Part II, which describes ways of using dynamic and automated infrastructure that align with the principles outlined in chapter 1.

If you're comfortable with the dynamic infrastructure and automation approaches described in chapter 1, then you may want to skim Parts I and II and focus on Part III. Part III gets more deeply into the infrastructure management regime - architectural approaches as well as team workflow.

CHAPTER 1

Challenges and principles

The new generation of infrastructure management technologies promises to transform the way we manage IT infrastructure. But many organizations today aren't seeing any dramatic differences, and some are finding that these tools only make life messier. As we'll see, Infrastructure as Code is an approach that provides principles, practices, and patterns for using these technologies effectively.

Why we need Infrastructure as Code

Virtualization, cloud, containers, server automation, software defined networking - these should simplify IT operations work. It should take less time and effort to provision, configure, update, and maintain services. Problems should be quickly detected and resolved, systems should all be consistently configured and up to date. IT staff should spend less time on routine drudgery, having time to rapidly make changes and improvements to help their organizations meet the ever-changing needs of the modern world.

But even with the latest and best new tools and platforms IT operations teams still find that they can't keep up with their daily workload. They don't have the time to fix longstanding problems with their systems, much less revamp them to make the best use of new tools. In fact, cloud and automation often makes things worse. The ease of provisioning new infrastructure leads to an ever-growing portfolio of systems, and it takes an ever-increasing amount of time just to keep everything from collapsing.

Adopting cloud and automation tools immediately lowers barriers for making changes to infrastructure. But managing changes in a way that improves consistency and reliability doesn't come out of the box with the software. It takes people to think through how they will use the tools, and put in place the systems, processes, and habits to use them effectively.

Some IT organizations respond to this challenge by applying the same types of processes, structures, and governance that they used to manage infrastructure and software before cloud and automation became commonplace. But the principles that applied in a time when it took days or weeks to provision a new server struggle to cope now that it takes minutes or seconds.

Legacy change management processes are commonly ignored, bypassed, or overruled by people who need to get things done[1]. Organizations which are more successful in enforcing these processes are increasingly seeing themselves outrun by more technically nimble competitors.

Legacy change management approaches struggle to cope with the pace of change offered by cloud and automation. But we still need a way to cope with the ever-growing, continuously changing landscape of systems created by cloud and automation tools. This is where Infrastructure as Code[2] comes in.

The Iron Age and the Cloud Age

In the "Iron Age" of IT, systems were directly bound to physical hardware. Provisioning and maintaining infrastructure was manual work, forcing humans to spend their time pointing, clicking, and typing to keep the gears turning. Because changes involved so much work, change management processes emphasized careful up-front consideration, design, and review work. This made sense because getting it wrong was expensive.

In the "Cloud Age" of IT, systems have been decoupled from the physical hardware. Routine provisioning and maintenance can be delegated to software systems, freeing the humans from drudgery. Changes can be made in minutes, if not seconds. Change management can exploit this speed, providing better reliability along with faster time to market.

1 "Shadow IT" is when people bypass formal IT governance to bring in their own devices, buy and install unapproved software, or adopt cloud-hosted services. These are typically a sign that internal IT is not able to keep up with the needs of the organization it serves.

2 The phrase "Infrastructure as Code" doesn't have a clear origin or author. While writing this book I followed a chain of people who have influenced thinking around the concept, each of whom said it wasn't them, but offered suggestions. This chain had a number of loops. The earliest reference I could find was from the Velocity conference in 2009, in a talk by Andrew Clay-Shafer and Adam Jacob. John Willis may be the first to document the phrase, in an article about the conference. Luke Kaines has admitted that he may have been involved, the closest anyone has come to accepting credit.

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. Changes are made to definitions, and then rolled out to systems through unattended processes that include thorough validation.

The premise is that modern tooling allows us to treat infrastructure as if it were software and data. This allows us to apply software development tools such as Version Control Systems (VCS), automated testing libraries, and deployment orchestration to manage infrastructure. We can also exploit development practices such as Test Driven Development (TDD), Continuous Integration (CI), and Continuous Delivery (CD).

Infrastructure as Code has been proven in the most demanding environments. For companies like Amazon, Netflix, Google, Facebook, and Etsy, IT systems are not just business critical, they *are* the business. There is no tolerance for downtime. Amazon's systems handle hundreds of millions of dollars in transactions every day. So it's no surprise that organizations like these are pioneering new practices for large scale, highly reliable IT infrastructure.

This book aims to explain how to take advantage of the cloud-era Infrastructure as Code approaches to IT infrastructure management. This chapter explores the pitfalls that organizations often fall into when adopting the new generation of infrastructure technology. It describes the core principles and key practices of Infrastructure as Code that are used to avoid these pitfalls.

Goals of Infrastructure as Code

The types of outcomes which many teams and organizations look to achieve through Infrastructure as Code include:

- IT infrastructure supports and enables change, rather than being an obstacle, or a constraint.
- Changes to the system are routine, without drama or stress for users or IT staff.
- IT staff spends their time on valuable things which engage their abilities, not on routine, repetitive tasks.
- Users are able to define, provision, and manage the resources they need, without needing IT staff to do it for them.
- Teams are able to easily and quickly recover from failures, rather than assuming failure can be completely prevented.

- Improvements are made continuously, rather than done through expensive and risky "big bang" projects.
- Solutions to problems are proven through implementing, testing, and measuring them, rather than by discussing them in meetings and documents.

Infrastructure as Code is not just for the cloud

Infrastructure as Code has come into its own with cloud, because it's difficult to manage servers in the cloud well without it. But the principles and practices of infrastructure as code can be applied to infrastructure whether it runs on cloud, virtualized systems, or even directly on physical hardware.

I use the phrase "Dynamic Infrastructure" to refer to the ability to create and destroy servers programmatically - ??? is dedicated to this topic. Cloud does this naturally, and virtualization platforms can be configured to do the same. But even hardware can be automatically provisioned so that it can be used in a fully dynamic fashion. This is sometimes referred to as "bare metal cloud".

It is possible to use many of the concepts of Infrastructure as Code with static infrastructure. Servers that have been manually provisioned can be configured and updated using server configuration tools. However, the ability to effortlessly destroy and rebuild servers is essential for many of the more advanced practices described in this book.

Challenges with dynamic infrastructure

In this section, we'll look at some of the problems teams often see when they adopt dynamic infrastructure and automated configuration tools. These are the problems that Infrastructure as Code addresses, so understanding them lays the groundwork for the principles and concepts that follow.

Server sprawl

Cloud and virtualization can make it trivial to provision new servers from a pool of resources. This can lead to the number of servers growing faster than the ability of the team to manage them as well as they would like.

When this happens, teams struggle to keep servers patched and up to date, leaving systems vulnerable to known exploits. When problems are discovered, fixes may not be rolled out to all of the systems that could be affected by them. Differences in versions and configurations across servers mean that software and scripts that work on some machines don't work on others.

This leads to inconsistency across the servers - configuration drift.

Configuration drift

Even when servers are initially created and configured consistently, differences can creep in over time.

- Someone makes a fix to one of the Oracle servers to fix a specific user's problem, and now it's different from the other Oracle servers.
- A new version of Jira needs a newer version of Java, but there's not enough time to test all of the other Java-based applications so that everything can be upgraded.
- Three different people install IIS on three different web servers over the course of a few months, and each person configures it differently.
- One JBoss server gets more traffic than the others and starts struggling, so someone tunes it, and now it's configuration is different from the other JBoss servers.

Being different isn't bad. The heavily loaded JBoss server probably should be tuned differently from ones with lower levels of traffic. But variations should be captured and managed in a way that makes it easy to reproduce and to rebuild servers and services.

Unmanaged variation between servers leads to snowflake servers and automation fear.

Snowflake servers

A snowflake server is different from any other server on your network. It's special in ways that can't be replicated.

Years ago I ran servers for a company that built web applications for clients, most of which were monstrous collections of Perl CGI. (Don't judge us, this was the dot-com days, everyone was doing it.) We started out using Perl 5.6, but at some point the best libraries moved to Perl 5.8, and couldn't be used on 5.6. Eventually almost all of our newer applications were built with 5.8 as well, but there was one, particularly important client application, which simply wouldn't run on 5.8.

It was actually worse than this. The application worked fine when we upgraded our shared staging server to 5.8, but crashed when we upgraded the staging environment. Don't ask why we upgraded production to 5.8 without discovering the problem with staging, but that's how we ended up. We had one special server that could run the application with Perl 5.8, but no other server would.

We ran this way for a shamefully long time, keeping Perl 5.6 on the staging server and crossing our fingers whenever we deployed to production. We were terrified to

touch anything on the production server; afraid to disturb whatever magic made it the only server that could run the client's application.

This situation led us to discover www.infrastructures.org[3], a site that introduced me to ideas that were a precursor to Infrastructure as Code. We made sure that all of our servers were built in a repeatable way, using FAI for PXE Boot installations, Cfengine to configure servers, and everything checked into CVS.

As embarrassing as this story is, most IT Ops teams have similar stories of special servers that couldn't be touched, much less reproduced. It's not always a mysterious fragility; sometimes there is an important software package that runs on an entirely different OS than everything else in the infrastructure. I recall an accounting package that needed to run on AIX, and a PBX system running on a Windows NT 3.51 server specially installed by a long forgotten contractor.

Once again, being different isn't bad. The problem is when the team that owns the server doesn't understand how and why it's different, and wouldn't be able to rebuild it. An IT Ops team should be able to confidently and quickly rebuild any server in their infrastructure. If any server doesn't meet this requirement, constructing a new, reproducible process that can build a server to take its place should be a leading priority for the team.

Jenga infrastructure

A Jenga infrastructure is easily disrupted and not easily fixed. This is the snowflake server problem expanded to an entire portfolio of systems.

The solution is to migrate everything in the infrastructure to a reliable, reproducible infrastructure, one step at a time. The Visible Ops Handbook[4] outlines an approach for bringing stability and predictability to a difficult infrastructure.

Don't touch that server. Don't point at it. Don't even look at it.

There is the possibly apocryphal story of the data center with a server that nobody had the login details for, and nobody was certain what the server did. Someone took the bull by the horns and unplugged the server from the network. The network failed completely, the cable was re-plugged, and nobody ever touched the server again.

3 Sadly, as of early 2016 the www.infrastructures.org site hadn't been updated since 2007.

4 The Visible Ops Handbook by Gene Kim, George Spafford, and Kevin Behr. The book was originally written before DevOps, virtualization, and automated configuration became mainstream, but it's easy to see how Infrastructure as Code can be used within the framework described by the authors.

Automation fear

At an open spaces session on configuration automation at a DevOpsDays conference, I asked the group how many of them were using an automation tools like Puppet or Chef. 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.

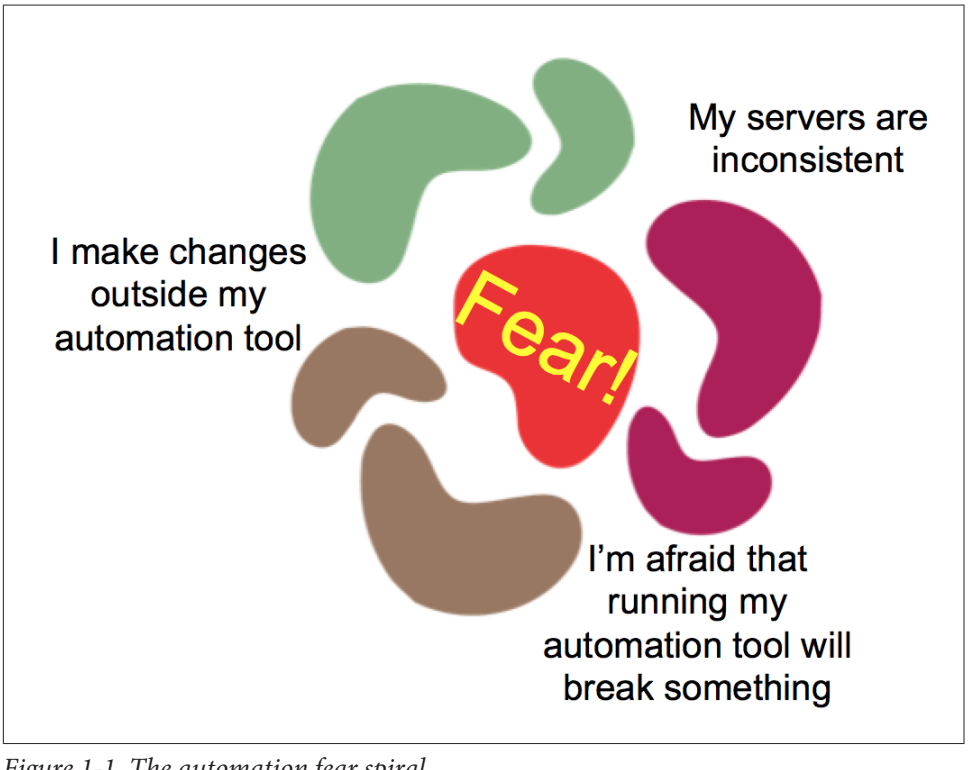

Figure 1-1. The automation fear spiral

This is the automation fear spiral, and infrastructure teams need to break this spiral to use automation successfully. The most effective way to break the spiral is to face your fears. Pick a set of servers, tweak the configuration definitions so that you know they work, and schedule them to run unattended, at least once an hour. Then pick another set of servers and repeat the process, and so on until all of your servers are continuously updated.

Good monitoring, and effective automated testing regimes as described in Part III of this book will help build confidence that configuration can be reliably applied and problems caught quickly.

Erosion

In an ideal world we would never need to touch an automated infrastructure once we've built it, other than to support something new or fix things that break. Sadly, the forces of entropy mean that even without a new requirement, our infrastructure will decay over time. The folks at Heroku call this erosion. Erosion is the idea that problems will creep into a running system over time.

The Heroku folks give these examples of forces that can erode a system over time:

- Operating system upgrades, kernel patches, and infrastructure software (e.g. Apache, MySQL, ssh, OpenSSL) updates to fix security vulnerabilities.
- The server's disk filling up with logfiles.
- One or more of the application's processes crashing or getting stuck, requiring someone to log in and restart them.
- Failure of the underlying hardware causing one or more entire servers to go down, taking the application with it.

Principles of Infrastructure as Code

This section describes principles that can help teams overcome the challenges described earlier in this chapter.

Principle: Systems can be easily reproduced

It should be possible to effortlessly and reliably rebuild any element of an infrastructure. Effortlessly means that there is no need to make any significant decisions about how to rebuild the thing. Decisions about which software and versions to install on a server, how to choose a hostname, and so on should be captured in the scripts and tooling that provision it.

The ability to effortlessly build and rebuild any part of the infrastructure is powerful. It removes much of the risk, and fear, when making changes. Failures can be handled quickly and with confidence. New services and environments can be provisioned with little effort.

Approaches for reproducibly provisioning servers and other infrastructure elements are discussed in Part II of this book.

Principle: Systems are disposable

One of the benefits of dynamic infrastructure is that resources can be easily created, destroyed, replaced, resized, and moved. In order to take advantage of this, systems should be designed to assume that the infrastructure will always be changing. Software should continue running even when servers disappear, appear, and when they are resized.

The ability to handle changes gracefully makes it easier to make improvements and fixes to running infrastructure. It also makes services more tolerant to failure. This becomes especially important when sharing large scale cloud infrastructure, where the reliability of the underlying hardware can't be guaranteed.

Cattle not pets

A popular expression is to "Treat your servers like cattle, not pets"[5] I miss the days of having themes for server names, and carefully selecting names for each new server we provisioned. But I don't miss having to manually tweak and massage every server in our estate.

A fundamental difference between the Iron Age and Cloud Age is the move from unreliable software, which depends on the hardware to be very reliable, to software that runs reliably on unreliable hardware[6]. See ???) for more on how embracing disposable infrastructure can be used to improve service continuity.

The case of the disappearing file server

The idea that servers aren't permanent things can take time to sink in. On one team, we set up an automated infrastructure using VMWare and Chef, and got into the

5 CloudConnect CTO Randy Bias attributed this expression to former Microsoft employee Bill Baker, from his presentation Architectures for open and scalable clouds. I first heard it in Gavin McCance's presentation CERN Data Centre Evolution. Both of these presentations are excellent.

6 Sam Johnson described in this view of the reliability of hardware and software in his article, Simplifying Cloud: Reliability

habit of casually deleting and replacing VMs. A developer, needing a web server to host files for teammates to download, installed a web server onto a server in the development environment and put the files there. He was surprised when his web server and its files disappeared a few days later.

After a bit of confusion, the developer added the configuration for his file repository to the chef configuration, taking advantage of tooling we had to persist data to a SAN. The team ended up with a highly-reliable, automatically configured file sharing service.

To borrow a cliche, the disappearing server is a feature, not a bug. The old world where people installed ad-hoc tools and tweaks in random places leads straight to the old world of snowflakes and untouchable jenga infrastructure. Although it was uncomfortable at first, the developer learned how to use Infrastructure as Code to build services - a file repository in this case - that are reproducible and reliable.

Principle: Systems are consistent

Given two infrastructure elements providing a similar service - for example two application servers in a cluster - the servers should be nearly identical. Their system software and configuration should be exactly the same, except for those bits of configuration that differentiate them from one another, like their IP addresses.

Letting inconsistencies slip into an infrastructure keeps you from being able to trust your automation. If one file server has an 80 GB partition, while another's 100 GB, and a third has 200 GB, then you can't rely on an action to work the same on all of them. This encourages doing special things for servers that don't quite match, which leads to unreliable automation.

Teams that implement the reproducibility principle can easily build multiple identical infrastructure elements. If one of these elements needs to be changed, such as finding that one of the file servers needs a larger disk partition, there are two ways that keep consistency. One is to change the definition, so that all file servers are built with a large enough partition to meet the need. The other is to add a new class, or role, so that there is now an "xl-file-server" with a larger disk than the standard file server. Either type of server can be built repeatedly and consistently.

Being able to build and rebuild consistent infrastructure helps with configuration drift. But clearly, changes that happen after servers are created need to be dealt with. Ensuring consistency for existing infrastructure is the topic of ???.

Principle: Processes are repeatable

Building on the reproducibility principle, any action you carry out on your infrastructure should be repeatable. This is an obvious benefit of using scripts and config-

uration management tools rather than making changes manually, but it can be hard to stick to doing things this way, especially for experienced system administrators.

For example, if I'm faced with what seems like a one-off task like partitioning a hard drive, I find it easier to just log in and do it, rather than to than write and test a script. I can look at the system disk, consider what the server I'm working on needs, and use my experience and knowledge to decide how big to make each partition, what file system to use, and so on.

The problem is that later on, someone else on my team might partition a disk on another machine, and make slightly different decisions. Maybe I made an 80 GB `/var` partition using ext3 on one file server, but Priya made `/var` 100 GB on another file server in the cluster, and used xfs. We're failing the consistency principle, which will eventually undermine our ability to automate things.

Effective infrastructure teams have a strong scripting culture. If a task can be scripted, script it. If a task is hard to script, drill down and see if there's a technique or tool that can help, or whether the problem the task is addressing can be handled in a different way.

Principle: Design is always changing

With Iron Age IT, making a change to an existing system is difficult and expensive. So limiting the need to make changes to the system once it's built makes sense. This leads to the need for comprehensive initial designs that take various possible requirements and situations into account.

Because it's impossible to accurately predict how a system will be used in practice, and how its requirements will change over time, this approach naturally creates overly-complex systems. Ironically, this complexity makes it more difficult to change and improve the system, which makes it less likely to cope well in the long run.

With Cloud Age dynamic infrastructure, making a change to an existing system can be easy and cheap. However, this assumes everything is designed to facilitate change. Software and infrastructure must be designed as simply as possible to meet current requirements. Change management must be able to deliver changes safely and quickly.

The most important measure to ensure that a system can be changed safely and quickly is to make changes frequently. This forces everyone involved to learn good habits for managing changes, to develop efficient, streamlined processes, and to implement tooling that supports doing so.

Practices

The previous section outlined high level principles. This section describes some of the general practices of Infrastructure as Code.

Practice: Use definition files

The cornerstone practice of Infrastructure as Code is the use of definition files. A definition specifies infrastructure elements and how they should be configured. The definition file is used as input for a tool that carries out the work to provision and/or configure instances of those elements.

The infrastructure element could be a server; a part of a server such as a user account; network configuration such as a load balancer rule; or many other things. Different tools have different terms for this - for example, Playbooks (Ansible), Recipes (Chef), or Manifests (Puppet). The term "configuration definition file" is used in this book as a generic term for these.

Example 1-1. Example of a definition file using a DSL

```
server: dbnode
  base_image: centos72
  chef_role: dbnode
  network_segment: prod_db
  allowed_inbound:
    from_segment: prod_app
    port: 1521
  allowed_inbound:
    from_segment: admin
    port: 22
```

Definition files are managed as text files. They may use a standard format such as JSON, YAML, or XML. Or they may define their own Domain Specific Language (DSL)[7].

Keeping specifications and configurations in text files makes them more accessible than storing them in a tool's internal configuration database. The files can also be treated like software source code, bringing a wide ecosystem of development tools to bear.

7 As defined by Martin Fowler and Rebecca Parsons in *http://martinfowler.com/books/dsl.html*, "DSLs are small languages, focused on a particular aspect of a software system. You can't build a whole program with a DSL, but you often use multiple DSLs in a system mainly written in a general purpose language." Their book, Domain Specific Languages, is a good reference on Domain Specific Languages, although it's written more for people thinking about implementing one than for people using them.

Practice: Self-documented systems and processes

IT teams commonly struggle to keep their documentation relevant, useful, and accurate. Someone might write up a comprehensive document for a new process, but it's rare for such documents to be kept up to date as changes and improvements are made to the way things are done. And documents still often leave gaps. Different people find their own shortcuts and improvements. Some people write their own personal scripts to make parts of the process easier.

So although documentation is often seen as a way to enforce consistency, standards and even legal compliance, in practice it's a fictionalized version of what really happens.

With Infrastructure as Code, the steps to carry out a process are captured in the scripts, definition files, and tools that actually implement the process. Only a minimum of added documentation is needed to get people started. The documentation that does exist should be kept close to the code it documents, to make sure it's close to hand and mind when people make changes.

Automatically generating documentation

On one project, my colleague Tom Duckering found that the team responsible for deploying software to production insisted on doing it manually. Tom had implemented an automated deployment using Apache Ant, but the production team wanted written documentation for a manual process.

So Tom wrote a custom Ant task that printed out each step of the automated deployment process. This way, a document was generated with the exact steps, down to the command lines to type. His team's Continuous Integration server generated this document for every build, so they could deliver a document that was accurate and up to date. Any changes to the deployment script were automatically included in the document without any extra effort.

Practice: Version all the things

The Version Control System (VCS) is a core part of infrastructure that is managed as code. The VCS is the source of truth for the desired state of infrastructure. Changes to infrastructure are driven by changes committed to the VCS.

Reasons why VCS is essential for infrastructure management include:

- Traceability: VCS provides a history of changes that have been made, who made them, and ideally, context about why. This is invaluable when debugging problems.

- Rollback: When a change breaks something - and especially when multiple changes break something - it's useful to be able to restore things to exactly how they were before.
- Correlation: When scripts, configuration, artifacts, and everything across the board are in version control and correlated by tags or version numbers, it can be useful for tracing and fixing more complex problems.
- Visibility: Everyone can see when changes are committed to a version control system, which helps situational awareness for the team. 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: VCSs can automatically trigger actions when a change is committed. This is a key to enabling Continuous Integration and Continuous Delivery pipelines.

??? explains how VCS works with configuration management tools, and ??? discusses approaches to managing your infrastructure code and definitions.

Practice: Continuously test systems and processes

Effective automated testing is one of the most important practices that infrastructure teams can borrow from software development. Automated testing is a core practice of high performing development teams. They implement tests along with their code, and run them continuously, typically dozens of times a day as they make incremental changes to their codebase.

It's difficult to write automated tests for an existing, legacy system. A system's design needs to be decoupled and structured in a way that facilitates independently testing components. Writing tests while implementing the system tends to drive clean, simple design, with loosely coupled components.

Running tests continuously during development gives fast feedback on changes. Fast feedback gives people the confidence to make changes quickly and more often. This is especially powerful with automated infrastructure, because a small change can do a lot of damage very quickly (aka DevOops, as described in ???). Good testing practices are the key to eliminating automation fear.

??? explores practices and techniques for implementing testing as part of the system, and particularly how this can be done effectively for infrastructure.

Practice: Small changes rather than batches

When I first got involved in developing IT systems, my instinct was to implement a complete piece of work before putting it live. It made sense to wait until it was "done" before spending the time and effort on testing it, cleaning it up, and generally making

it "production ready". The work involved in finishing it up tended to take a lot of time and effort, so why do the work before it's really needed?

However, over time I've learned to the value of small changes. Even for a big piece of work, it's useful to find incremental changes that can be made, tested, and pushed into use, one by one. There are a lot of good reasons to prefer small, incremental changes over big batches:

- It's easier, and less work, to test a small change and make sure it's solid
- If something goes wrong with a small change, it's easier to find the cause than if something goes wrong with a big batch of changes
- It's faster to fix or reverse a small change
- One small problem can delay everything in a large batch of changes from going ahead, even when most of the other changes in the batch are fine
- Getting fixes and improvements out the door is motivating. Having large batches of unfinished work piling up, going stale, is demotivating.

As with many good working practices, once you get the habit it's hard to *not* do the right thing. You get much better at releasing changes. These days, I get uncomfortable if I've spent more than an hour working on something without pushing it out.

Practice: Keep services available continuously

We need to make sure our service is always able to handle requests, in spite of what might be happening to the infrastructure. If a server disappears, we need to have other servers already running, and be able to quickly start up new ones, so that service is not interrupted. This is nothing new in IT, although virtualization and automation can make it easier.

Data management, broadly defined, can be trickier. Service data can be kept intact in spite of what happens to the servers hosting it through replication and other approaches that have been around for decades. When designing a cloud-based system, it's important to widen the definition of data that needs to be persisted, usually including things like application configuration, logfiles, and more.

The chapter on continuity (???) goes into techniques for keeping service and data continuously available.

Antifragility - Beyond "robust"

We typically aim to build robust infrastructure, meaning systems will hold up well to shocks - failures, load spikes, attacks, etc. However, Infrastructure as Code lends itself to taking infrastructure beyond robust, becoming antifragile.

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

The effect of physical stress on the human body is an example of antifragility in action. Exercise puts stress on muscles and bones, essentially damaging them, causing them to become stronger. Protecting the body by avoiding physical stress and exercise actually weakens it, making it more likely to fail in the face of extreme stress.

Similarly, protecting an IT system by minimizing the number of changes made to it will not make it more robust. Teams which are constantly changing and improving their systems are much more ready to handle disasters and incidents.

The key to an antifragile IT infrastructure is making sure that the default response to incidents is improvement. When something goes wrong, the priority is not simply to fix it, but to improve the ability of the system to cope with similar incidents in the future.

The secret ingredient of anti-fragile IT systems

People are the part of the system that can cope with unexpected situations, and modify the other elements of the system to handle similar situations better the next time around. This means the people running the system need to understand it quite well, and be able to continuously modify it.

This doesn't fit the idea of automation as a way to run things without humans. Someday we might be able to buy a standard corporate IT infrastructure off the shelf and run it as a black box, without needing to look inside, but this isn't possible today. IT technology and approaches are constantly evolving, and even in non-technology businesses, the most successful companies are the ones continuously changing and improving their IT.

The key to continuously improving an IT system is the people who build and run it. So the secret to designing a system that can adapt as needs change is to design it around the people.[8]

8 Brian L. Troutwin gave a talk at DevOpsDays Ghent in 2014 on Automation with humans in mind. He gave an example from NASA of how humans were able to modify the systems on the Apollo 13 spaceflight to cope with disaster. He also gave many details of how the humans at the Chernobyl nuclear power plant were prevented from interfering with the automated systems there, which kept them from taking steps to stop or contain disaster.

Conclusion: What good looks like

The hallmark of an infrastructure team's effectiveness is how well it handles changing requirements. Highly effective teams can handle changes and new requirements easily, breaking down requirements into small pieces and piping them through in a rapid stream of low-risk, low-impact changes.

Some signals that a team is doing well:

- Every element of the infrastructure can be rebuilt quickly, with little effort.
- All systems are kept patched, consistent, and up to date.
- Standard service requests, including provisioning standard servers and environments, can be fulfilled within minutes, with no involvement from infrastructure team members. SLAs are unnecessary.
- Maintenance windows are rarely, if ever, needed. Changes take place during working hours, including software deployments and other high risk activities.
- The team tracks MTTR (Mean Time to Recover) and focuses on ways to improve this. Although MTBF (Mean Time Between Failure) may also be tracked, the team does not rely on avoiding failures.[9]
- Team members feel their work is adding measurable value to the organization.

The next three chapters discuss the core toolchains for building Infrastructure as Code. The dynamic infrastructure management platform, infrastructure definition tools, and server configuration tools. Part II describes how to use these tools in a way that follows infrastructure as code approaches.

What's next?

The next four chapters focus on the tooling involved in Infrastructure as Code. Readers who are already familiar with these tools may choose to skim or skip these chapters, and go straight to ???, which describes Infrastructure as Code patterns for using the tools.

I have grouped tools into four chapters. As with any model for categorizing things, this division of tools is not absolute. Many tools will cross these boundaries, or have a fuzzy relationship to these definitions. This grouping is a convenience, to make it easier to discuss the many tools involved in running a dynamic infrastructure.

- **Dynamic Infrastructure Platforms** are used to provide and manage basic infrastructure resources, particularly compute (servers), storage, and networking.

9 See John Allspaw's seminal blog post, MTTR is more important than MTBF (for most types of F).

These include public and private cloud infrastructure services, virtualization, and automated configure of physical devices. This is the topic of ???.

- **Infrastructure Definition Tools** are used to mange the allocation and configuration servers, storage, and networking resources. These tools provision and configure infrastructure at a high level. This is the subject of ???.
- **Server Configuration Tools** deal with the details of servers themselves. This includes software packages, user accounts, and various types of configuration. This group, which is typified by specific tools including Cfengine, Puppet, Chef, and Ansible, are what many people think of first when discussing infrastructure automation and Infrastructure as Code. These tools are discussed in ???.
- **Infrastructure Services** are tools and services that help to manage infrastructure and application services. Topics such as monitoring, distributed process management, and software deployment are the subject of ???.

CHAPTER 2

Patterns for defining infrastructure

Most of Part II of this book has focused on provisioning and configuring servers. This chapter will look at how to provision and configure larger groups of infrastructure elements. As infrastructure grows in size, complexity, and number of users, it becomes harder to get the benefits we look for with Infrastructure as Code.

- The larger scope of what may be affected by a given change makes it difficult to make changes frequently, quickly, and safely
- People spend more time on routine maintenance and firefighting, less on making more valuable improvements to services
- Allowing users to provision and manage their own resources can risk disruption to other users and services

The typical reaction is to centralize control over infrastructure. This leads to more time spent on meetings, documentation, and change process, and less time spent on activities which add value to the organization.

But an alternative to centralizing control is to design infrastructure to minimize the scope of impact of a given change. An effective approach to defining, provisioning, and managing infrastructure will enable changes to be made frequently and confidently even as the size of the infrastructure grows. This in turn allows ownership of application and service infrastructure to be safely delegated.

What is a stack?

A stack[1] is a collection of infrastructure elements that are defined as a unit.

A stack can be any size, from a single server, to a pool of servers and its networking and storage, to all of the servers and infrastructure involved in an application, or even everything in an entire data-center. What makes a set of infrastructure elements a stack isn't the size, but whether it's defined and changed as a unit.

The concept of a stack hasn't been commonly used with manually managed infrastructures. Elements are added organically, and networking boundaries are naturally used to think about infrastructure groupings.

But automation tools force more explicit groupings of infrastructure elements. It's certainly possible to put everything into one large group. And it's also possible to structure stacks by following network boundaries. But these aren't the only ways to organize stacks.

Rather than simply replicating patterns and approaches that worked with Iron Age static infrastructure, it's worth considering whether there are more effective patterns to use.

This chapter is concerned with ways of structuring infrastructure into "stacks" to support easier and safer change management. What are good ways of dividing infrastructure into stacks? What is a good size for a stack? What are good approaches to defining and implementing stacks?

Environments

The term "environment" is typically used when there are multiple stacks that are actually different instances of the same service or set of services. The most common use of environments is for testing. An application or service may have "development", "test", "pre-production", and "production" environments. The infrastructure for these should generally be the same, with perhaps some variations due to scale.

Keeping multiple environments configured consistently is a challenge for most IT organizations, and one which Infrastructure as Code is particularly well-suited to address.

1 The inspiration for choosing the term stack comes mainly from the term's use by AWS CloudFormation.

Antipattern: Hand-crafted infrastructure

It's not unusual for teams which automatically provision and configure their servers to manually define surrounding infrastructure elements, such as networking and storage. This is especially common when using virtualized servers with non-virtualized storage and networking. Quite often, teams will script server creation, but use an interactive interface to configure networking and allocate storage.

For example, setting up a database cluster for a new environment may involve someone allocating disk space on a SAN (Storage Area Network), and then configuring firewall rules and selecting IP addresses for the servers in the cluster.

Changes made manually with an interactive GUI or tool aren't easily repeatable, and lead to inconsistency between environments. A firewall change made by hand in a test environment may not be made the same way for the production environment.

Organizations which work this way often find they need to take extra time to manually test and inspect each environment after it's built. Signs that this is an issue is if it is common practice to allocate time on project plans for correcting mistakes after setting up new environments, or else that it's common for implementation schedules to slip due to unexpected errors.

Snowflake environments

As with snowflake servers, an environment which is "special" can be difficult to rebuild, which makes it scary to change and improve. It's very common to see this for infrastructure services like monitoring, and other internally-facing services such as bug tracking systems. Resist the temptation to treat setting up this kind of system as a one-off task that isn't worth properly automating.

Defining infrastructure stacks as code

The following example is a stack with a simple web server cluster with front-end networking.

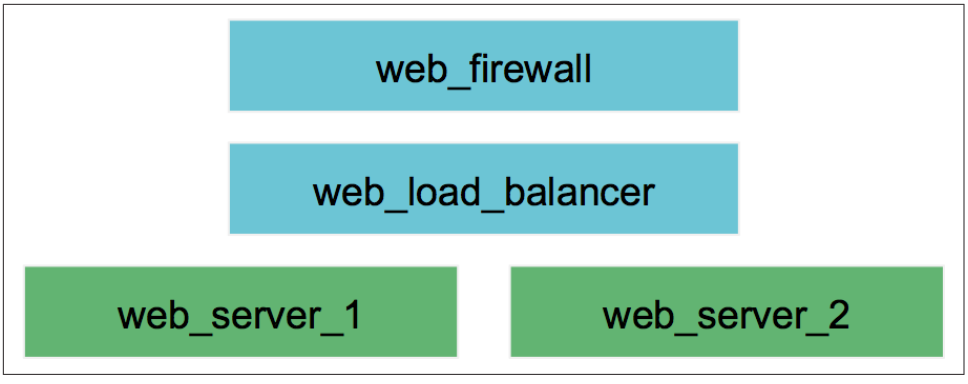

Figure 2-1. A simple environment

The Terraform file excerpt below shows how this stack might be defined on AWS. The definition file would be committed to a VCS, so that it can be easily rebuilt or replicated.

The first two blocks each define a web server, specifying the server template (AMI) and the size of the server. The third block defines a load balancer (ELB). This defines the port to take requests on and forward them to, and references to the server instances defined in the earlier blocks as the destination for requests. The ELB block also refers to the security group, defined in the last block of the example, which is the firewall rule to allow traffic in[2].

Example 2-1. Terraform definition for a simple stack

```
resource "aws_instance" "web_server_1" {
  ami = "ami-47a23a30"
  instance_type = "t2.micro"
}

resource "aws_instance" "web_server_2" {
  ami = "ami-47a23a30"
  instance_type = "t2.micro"
}

resource "aws_elb" "web_load_balancer" {
  name = "weblb"

  listener {
    instance_port = 80
```

2 Note that this is not a well-designed example. In practice, we should use a dynamic pool of servers - an Autoscaling Group in AWS - rather than statically defined ones. We would probably prefer to have the AMIs defined as variables, so they can be changed without having to edit and push changes to this definition file.

```
    instance_protocol = "http"
    lb_port = 80
    lb_protocol = "http"
  }

  instances = [
    "${aws_instance.web_server_1.id}",
    "${aws_instance.web_server_2.id}"
  ]

  security_groups = ["${aws_security_group.web_firewall.id}"]
}

resource "aws_security_group" "web_firewall" {
  name = "web_firewall"

  ingress {
      from_port = 80
      to_port = 80
      protocol = "tcp"
      cidr_blocks = ["0.0.0.0/0"]
  }
}
```

Antipattern: Per-Environment Definition Files

A simplistic way of managing multiple, similar environments is to have a separate definition file for each environment, even when the environments are used to run different instances of the same application or service.

With this approach, a new environment can be made by copying the definition file from another environment, and editing it to reflect differences such as networking details. When a fix, improvement, or other change is made, it is made by hand to each of the separate definition files.

This dependency on manual effort to keep multiple files up to date and consistent is the weakness of per-environment definition files. There is no way to ensure that changes is made consistently. Also, because the change is made manually to each environment, it isn't possible to use an automated pipeline to test and promote the change from one environment to the next.

Pattern: Re-usable Definition Files

A more effective approach is to use a single definition file across all of the environments which represent the same application or service. The file can be parameterized to set options specific to each environment. This makes it easy to build and run multiple instances of an infrastructure stack

The earlier Terraform definition (Example 2-1) can be modified to use environment-specific parameters. The snippet below shows part of the previous example, adding tags to the servers to indicate the environment. A variable `environment` is declared, signalling that it should be provided to the Terraform command line tool. This variable is then used to set the value of a tag on the AWS EC2 instance.

Example 2-2. Parameterized Terraform environment definition

```
variable "environment" {}

resource "aws_instance" "web_server_1" {
  ami = "ami-47a23a30"
  instance_type = "t2.micro"
  tags {
    Name = "Web Server 1"
    Environment = "${var.environment}"
  }
}

resource "aws_instance" "web_server_2" {
  ami = "ami-47a23a30"
  instance_type = "t2.micro"
  tags {
    Name = "Web Server 2"
    Environment = "${var.environment}"
  }
}
```

The definition tool is run separately for each environment.

First for QA:

Example 2-3. Create or update the QA environment

```
$ terraform apply -state=qa-env.tfstate -var environment=qa
```

Then the tool is run again for PROD:

Example 2-4. Create or update the PROD environment

```
$ terraform apply -state=prod-env.tfstate -var environment=prod
```

This creates two sets of servers, and assuming we've similarly parameterized the load balancer, it creates separate ELBs as well.

Separate state files or IDs per environment

These Terraform command line examples use the "-state" parameter to define a separate state data file for each environment. Terraform stores information about infrastructure that it manages in a state file. When managing multiple instances of an infrastructure with a single definition file, each instance needs to have its state stored in a separate file.

Other definition tools, such as CloudFormation, store the state server side, and allow each instance to be identified by a parameter - `StackName` in the case of CloudFormation.

Practice: Test and promote stack definitions

A reusable, parameterized definition file can be reliably tested before being applied to production infrastructure. Rather making a networking configuration change directly to a production environment, the change can be made to the relevant definition file, and applied to a test environment first. Automated tests can validate whether there are any errors or issues with the change. Once testing has been carried out, the change can be safely rolled out to the production infrastructure.

Because the process for applying the configuration is automated, there is a far higher confidence in the reliability of testing. The ease of creating and modifying a test environment also lowers barriers. When people need to manually set up and run tests, they are tempted to skip it when they're in a hurry. "It's a small change that shouldn't break anything".

This leads to test environments that aren't configured consistently with production, which makes testing less reliable. This in turn gives more reason not to bother with testing. This is the automation fear spiral[3] all over again. Using a single definition file across environments makes it trivial to keep the environments consistently configured.

3 "Automation fear" on page 17

Simplifying change management process for environments

In larger organizations with a need for strong governance around changes to infrastructure, making a configuring change and rolling it out across the pipeline of test, staging, and production environments can be lengthy.

Making a single change may require a separate change process for each environment, with detailed designs, tickets, change requests, and change reviews. In many cases, different people may implement the same change in different environments. When this is a manual process, no amount of documentation or inspection can guarantee the change will be implemented consistently.

The proper use of a re-usable environment definition, in conjunction with an automated pipeline, can make this process simpler, faster, and more consistent. The amount of process overhead needed can be dramatically reduced. What's more, because changes, and the process used to apply them, are explicitly defined in files and managed in a VCS, it is easy to provide traceability and assurance that legal and other requirements are being complied with.

Self-service environments

Another benefit of a parameterize environment definitions is that new instances can be easily created.

For example, development and testing teams don't need to coordinate access to shared environments, slowing their pace of work. A new environment can be built in minutes. Environments can even be created and destroyed on demand.

This can also be used with standard services, such as a ticketing application like Jira. Each team can easily be given their own instance, avoiding the need to coordinate configuration changes across multiple teams.

Environment and service instances defined through a common definition file can be provisioned through self-service. Rather than requiring infrastructure team members to take time to set up services and infrastructure, a team can use a pre-defined, well-tested template to spin up their own instance.

It's important that self-service provisioning be supported by a process for updates. When a change is made to a shared infrastructure definition, it should be rolled out to all existing instances. A change management pipeline can ensure that changes are tested before being rolled out to teams using them, and that they are applied and tested to test environments for applications before being applied to production environments.

Structuring infrastructure

An infrastructure is normally composed of a variety of interrelated stacks. Some stacks share infrastructure like networking devices. Others have inter-dependencies which require networking connections between them. And there is normally a continuous stream of changes which need to be made to some or all stacks, which can potentially create issues.

This section looks at some ways to structure infrastructure stacks to address these concerns.

Antipattern: Monolithic Stack

When first adopting an infrastructure definition tool, it's easy to create large, sprawling infrastructure in a single definition. Different services and applications, even different testing environments, can all be defined and managed together.

This simplifies integration of different elements of the infrastructure, and sharing them. In the monolithic stack shown below, the shared pool of web servers needs to be configured to access two different pools of CMS (Content Management System) servers. Having all of these configured in one file is convenient. If a new CMS server is added to the definition, it can be referred to by the web servers and network routing.

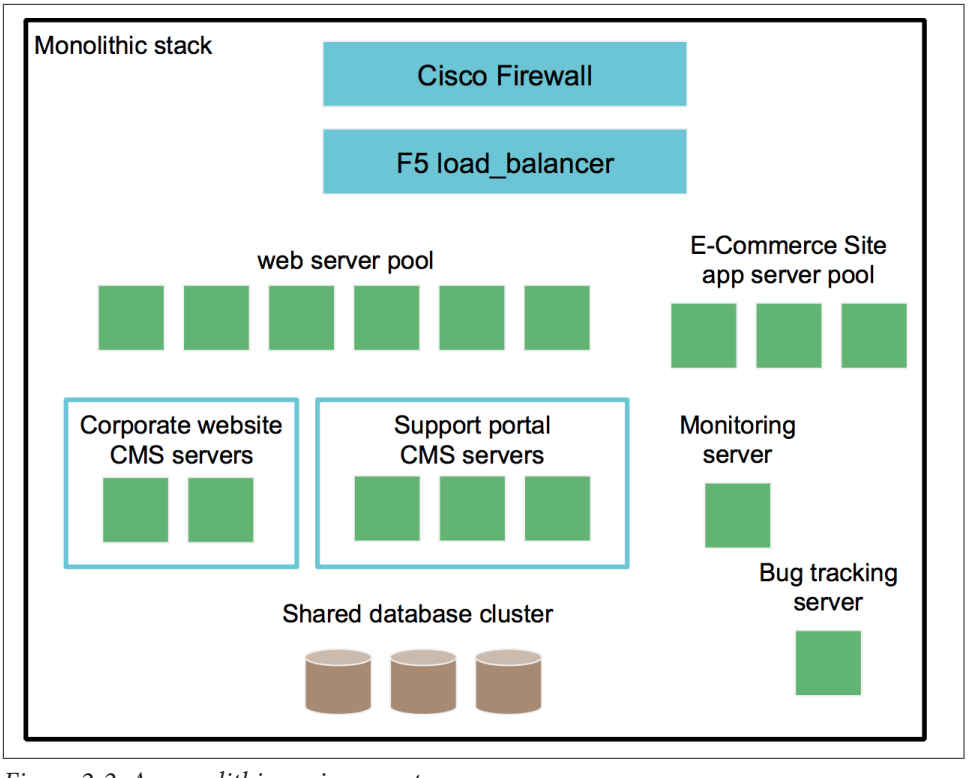

Figure 2-2. A monolithic environment

The difficulty of a monolithic definition is that it becomes cumbersome to change. With most definition tools the file can be organized into separate files. But if making a change involves running the tool against the entire infrastructure stack, things become dicey.

- It's easy for a small change to break many things
- It's hard to avoid tight coupling between the parts of the infrastructure
- Each instance of the environment is large, and expensive
- Developing and testing changes requires testing an entire stack at once, which is cumbersome
- If many people can make changes to the infrastructure, there is a high risk someone will break something
- On the other hand, if changes are limited to a small group to minimize the risk, then there are likely to be long delays waiting for changes to be made

You know your infrastructure definition is becoming monolithic when people become afraid to make a change. When you find yourself considering adding organizational processes to coordinate and schedule changes - stop! There are ways to structure your infrastructure to make it easier and safer to make changes. Rather than adding organizational and process complexity to manage complex infrastructure designs, redesign the infrastructure to eliminate unnecessary complexity.

Avoid "lift and shift" when migrating infrastructure

Organizations which have existing, static infrastructure will have become used to their architectural patterns and implementations. It's tempting to assume these are still relevant, and even necessary to keep when moving to a dynamic infrastructure platform.

But in many cases applying existing patterns will, at best, miss out on opportunities to leverage newer technology to simplify and improve the architecture. At worst, replicating existing patterns with the newer platforms will involve adding even more complexity.

It can be difficult to see "outside the box" of an existing, proven infrastructure implementation, but it's well worth the effort.

Dividing an application environment into multiple stacks

Multi-tier applications can be divided into multiple stacks, to allow each tier to be changed independently. The example below has a stack each for web servers, application servers, database cluster, and networking.

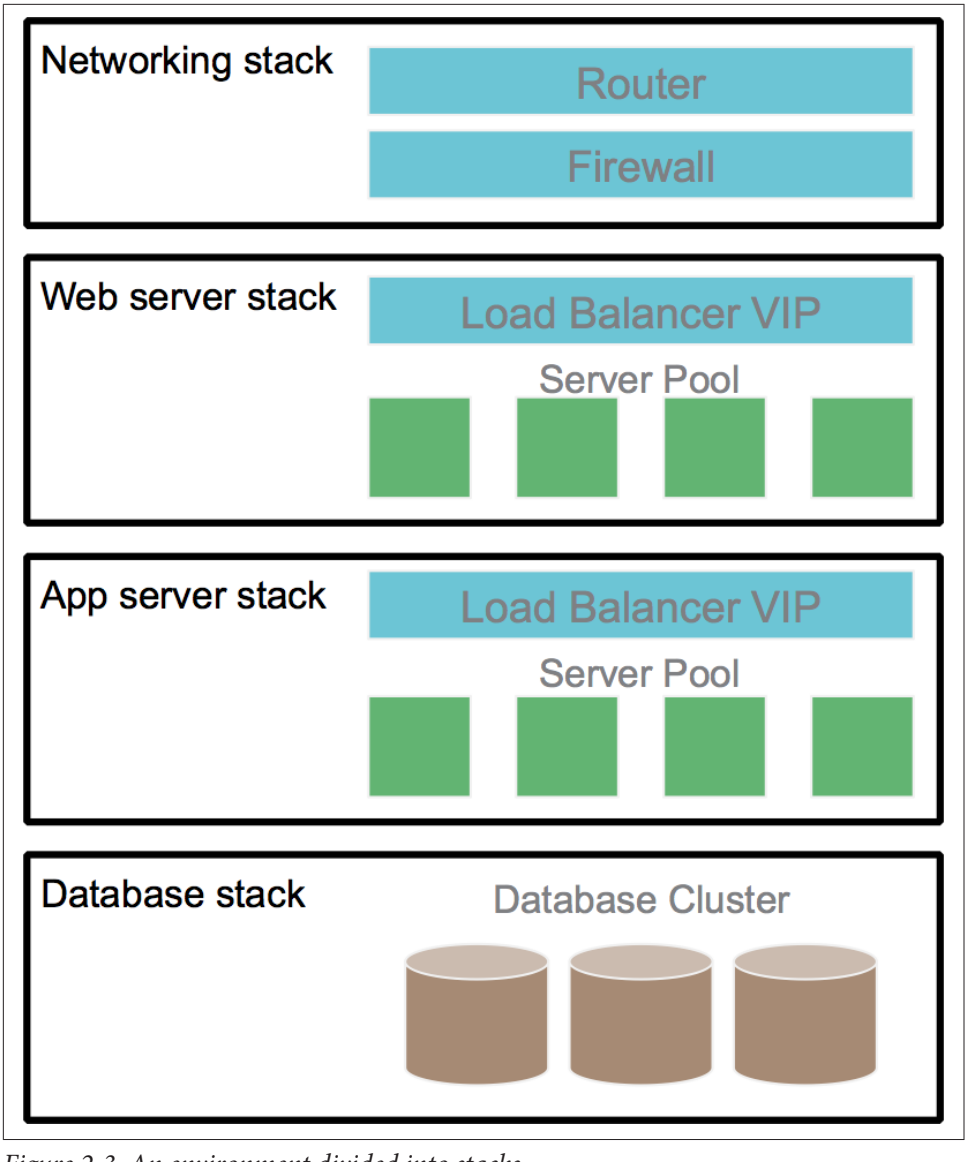

Figure 2-3. An environment divided into stacks

These stacks have inter-dependencies. The web server stack needs to be added to the firewall and subnets defined in the network stack. It also needs to know the IP address of the load balancer VIP for the application server stack, so the web servers can be configured to proxy requests to the application server. The application server stack will similarly need the IP addresses of the database cluster.

Example 2-5 shows parts of a Terraform definition[4] for the web server stack from Figure 2-3. It assumes the networking stack, application server stack, and dtabase stack are all defined in separate files, and each built and updated separately.

So this definition only defines the infrastructure elements that are specific to the web server pool, including the autoscaling group and load balancer. Changes can be made to this file and applied, with only minimal coupling to the other stacks in an environment.

Example 2-5. Web server stack definition

```
# Declare a variable for passing the app server pool's IP address
variable "app_server_vip_ip" {}

# Define the web server pool
resource "aws_autoscaling_group" "web_servers" {
  launch_configuration = "${aws_launch_configuration.webserver.name}"
  min_size = 2
  max_size = 5
}

# Define how to create a web server instance for the pool
resource "aws_launch_configuration" "webserver" {
  image_id = "ami-12345678"
  instance_type = "t2.micro"
  # Pass the IP address of the app server pool so it can be used
  # by the server configuration tool
  user_data { app_server_vip_ip = "${app_server_vip_ip}" }
}

# Define the load balancer for the web server pool
resource "aws_elb" "web_load_balancer" {
  instances = [ "${aws_asg.web_servers.id}" ]
  listener {
    instance_port = 80
    instance_protocol = "http"
    lb_port = 80
    lb_protocol = "http"
  }
}
```

Managing configuration parameters between stacks

A basic, and not very scalable technique for coordinating information between infrastructure stacks is to have a script that runs the definition tool for all of the different

4 In the interests of clarity, this example excludes a number of arguments that would be needed for it to actually work.

stacks. It makes sure to run them in the appropriate order, and captures the outputs of each stack so they can be passed as variables to other stacks. However, this is brittle, and loses the advantage of being able to run the definition tool independently for different stacks.

This is where a configuration registry[5] comes in handy. It provides a structured way to capture information about a stack, and make it available in a central location for other stacks to use.

Many infrastructure definition tools have built-in support for setting and fetching values from a registry.

Example 2-6 below modifies Example 2-5 to use a Consul configuration registry. It finds the IP address of the application server stack's load balancer VIP in the registry, looking up the key path `myapp/${var.environment}/appserver/vip_ip`. Using `var.environment` allows this definition file to be shared across multiple environments, each storing its own app server VIP in a unique location.

Example 2-6. Application server stack definition using a registry

```
# The "environment" parameter must be passed in
variable "environment" {}

# Import keys from the Consul registry
resource "consul_keys" "app_server" {
  key {
    name = "vip_ip"
    path = "myapp/${var.environment}/appserver/vip_ip"
  }
}

resource "aws_launch_configuration" "webserver" {
  image_id = "ami-12345678"
  instance_type = "t2.micro"
  # Pass the IP address of the app server pool so it can be used
  # by the server configuration tool
  user_data { app_server_vip_ip = "${consul_keys.app_server.var.vip_ip}" }
}
```

Lightweight registry approaches for sharing configuration between stacks

Some teams prefer to use a lightweight registry[6], although it may require more work to implement than an off the shelf tool. For example, the script that runs the defini-

5 As described in ??? (???), there are different types of tooling and approaches that can be used for a configuration registry.

6 As discussed in ??? (???)

tion tool might fetch structured registry files (e.g. JSON or YAML files) from a static file sharing service (e.g. an AWS S3 bucket), and pass it as parameters for applying the stack definition. The output of the definition tool is then captured and stored in the file sharing service, where it will be used for other stacks.

This can be highly scalable and robust, since it can take advantage of standard tools and techniques for hosting static files, including caching and distribution.

Packaging the outputs of a stack into a system package, like an RPM or .deb file, is another technique that leverages existing, well-proven tooling, as does committing the output files into a VCS.

Sharing infrastructure elements

By itself, the example above of splitting a single application environment into several stacks may not be especially useful. Most definition tools only apply changes to the part of the stack that has actually changed, anyway. So if you are making a change to the web server tier, having the other tiers defined in separate files doesn't buy much. Worse, we've seen how much complexity is added to the system in order to pass information between stacks.

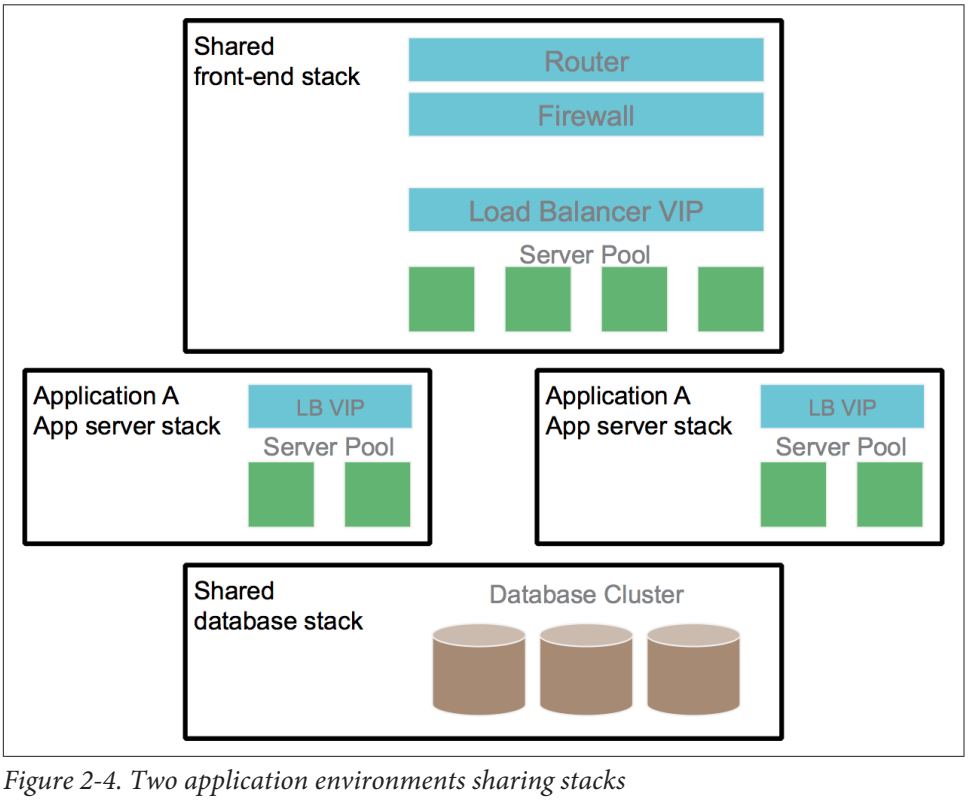

Figure 2-4. Two application environments sharing stacks

However, dividing stacks makes more sense as the size and complexity of the infrastructure grows.

One use is to make it easier for different application environments to share infrastructure resources. Figure 2-4 shows two applications sharing a single front-end stack, including a pool of web servers, and also sharing a database stack. Each application might have its own database schema and virtual web host in each of these stacks. The application stacks are independently managed.

Pitfalls of sharing infrastructure elements

Sharing stacks should be avoided whenever possible, because it creates dependencies which can make changes expensive and risky.

Sharing is usually driven by hardware such as networking devices and storage. Hardware devices tend to be expensive, and changing them may be high effort. But virtualized and cloud infrastructure, driven by automated configuration, makes it cheaper and easier to give each application dedicated resources.

The overhead of changes to infrastructure increases exponentially with the number of applications and services that share it. This overhead is reflected in time, cost, and risk.

In the earlier example with the database cluster shared between two applications, a change to the cluster, such as upgrading the database server, affects both applications. The change needs to be coordinated for both of these, which can make scheduling the upgrade more challenging.

Both applications may need to be taken offline, have their schemas updated, brought up, and then tested. If things go wrong, both applications may be offline for an extended time. If one application works fine after the upgrade but the other doesn't, both will need to be rolled back.

Because changing shared infrastructure has a higher risk, organizations must put more effort into planning and managing the change. This can involve meetings, documentation, and project plans. A larger number of people need to schedule time for planning, implementation, and testing. Project managers and analysts get involved to handle all of this. Entire teams and departments may be created to oversee this type of process.

One indication that this is an issue is when a simple change takes several months and a budget running as high as five and six figures. Another indication is when the organization is still running outdated versions of key system and middleware, having been unable to muster the time and money to carry out upgrades that impact many applications and services.

But by allocated dedicated infrastructure to each application, upgrades can be carried out in more manageable chunks, with less overhead.

At a glance, upgrading the database for each application one at a time may seem less efficient than upgrading a single shared instance all in one go. But upgrading the database for a single application has a much smaller risk area. Automation can cut the time and improve the reliability of the upgrade process. The upgrade can be made even more reliable by having it automatically applied and tested in a test environment first.

This is another case where automation and dynamic infrastructure, when used the right way, can dramatically simplify the change process while also making it less risky.

Practice: Manage application code and infrastructure code together

The default approach in many organizations is to manage the definitions for a given application's infrastructure separately from the code for that application. Application code is kept in its own VCS project, and infrastructure definitions are kept in another. Often, each is managed by a separate team. This follows the tradition of separating development and operations concerns, but can add technical and organizational overhead.

Consider a change that involves both the application and its infrastructure. An application might read environment-specific configuration options from a file that is added to the system by a server configuration definition, such as an Ansible playbook. The file contains the IP address of the database server, which may be different in each environment the application is deployed to. If the playbook is managed in a separate VCS repository from the application code, then a new version of the application may be deployed to an environment that hasn't had a new configuration option added to the file yet.

Dividing ownership of an application from the configuration of its infrastructure between teams adds even more complexity. In addition to technical integration issues, the time and work of both teams need to be coordinated. When the infrastructure is shared by multiple development teams, we see the same problems described earlier in this chapter ("Pitfalls of sharing infrastructure elements" on page 44).

It's certainly possible to manage the dependencies between an application and its infrastructure even when they are managed separately, and by separate teams. But this involves adding complexity, including technical mechanisms and organizational processes.

An alternative is to manage application software and its supporting infrastructure as a single concern.[7] Putting the definitions for the application-specific infrastructure elements into the same project folder as the development code makes it easy to manage, test, and deliver them as a unit.

Example 2-7. Project structure with combined application and infrastructure code

```
./our-app
./our-app/build.sh
./our-app/deploy.sh
./our-app/pom.xml
./our-app/Vagrantfile
./our-app/src
./our-app/src/java
./our-app/src/java/OurApp.java
./our-app/src/infra
./our-app/src/infra/terraform
./our-app/src/infra/terraform/our-app.tf
./our-app/src/infra/ansible
./our-app/src/infra/ansible/roles
./our-app/src/infra/ansible/roles/our_app_server
./our-app/src/infra/ansible/roles/our_app_server/files
./our-app/src/infra/ansible/roles/our_app_server/files/service.sh
./our-app/src/infra/ansible/roles/our_app_server/tasks
./our-app/src/infra/ansible/roles/our_app_server/tasks/main.yml
./our-app/src/infra/ansible/roles/our_app_server/templates
./our-app/src/infra/ansible/roles/our_app_server/templates/config.yml
```

Managing application and infrastructure definitions together means they can be tested together. The example project structure above includes configuration to build and deploy the application to a local virtual machine with Vagrant. This allows developers, testers, and operations people to run the application in a server that is configured the same way as the production servers. Changes and fixes can be made and tested locally with a high level of confidence.

A good rule of thumb for what code can and should be managed together is to consider the scope of a typical change. If it's common for changes to involve modifying code that lives in different VCS repositories, then putting them into the same repository is likely to simplify the process for making a change.

7 My former colleague Ben Butler-Cole put it well, "Build systems, not software".

Approaches to sharing definitions

With many development teams, there are often benefits from sharing infrastructure code. Getting the balance right between sharing good code and keeping team agility can be tricky.

Central code libraries are a common approach to sharing infrastructure definitions and code. Most server configuration tools support using a centralized library for cookbooks or modules. So all of the application teams in an organization could use the same Puppet module for building and configuring nginx web servers, for example. Improvements and upgrades can be made to the module, and all of the teams can take advantage of it.

Of course, this risks the pitfalls of shared infrastructure, since a change can potentially impact multiple teams. The overhead and risks this creates often discourages frequent updates.

A few approaches that some teams use to get around this:

- Versioning of shared modules in a way that allows teams to opt-in to updates. This allows teams to adopt newer versions of modules as time permits.
- Copy rather than share. Teams maintain templates for application and infrastructure code, and copy the files as needed to create new applications, and fold in improvements as they become available.
- Optional sharing. Teams have the option of using a shared module, but have the option to write alternative modules if it makes more sense for their needs.
- Overridable modules. Modules are designed in a way that makes it easy for application teams to customize them and override behavior as needed.

As a rule, sharing functionality and definitions makes more sense for things where the requirements are truly common, don't change very often, and are low-risk to change. It also makes sense when there is a clear, easy way for teams to customize and override the shared capability.

As an example, many organizations find that the base operating system distribution can be shared fairly easily. So a server template with the base OS can be centrally created, and made available to teams. Teams are able to customize this base, either by building a new template that layers changes onto the base template[8], or by writing server configuration definitions (Chef, Puppet, etc.) that makes changes to the base image when provisioning a new server instance.

8 ???)

In those rare cases where a team needs a different base image - for example, if they're running third party software which requires a different OS distribution - then they will ideally have the option to create and manage their own images, potentially sharing them with other teams.

Practice: Align infrastructure design with the scope of change

Clearly there are tradeoffs to consider when deciding how to split infrastructure into stacks. A division that makes sense in one organization may not make sense in another, depending on the team structures and on the services that the infrastructure supports.

However, a good guiding principle when considering the breakdown is the scope of change. Consider types of changes that are made often, and those which are the most difficult and risky. Then look at the parts of the infrastructure that are impacted by each of those changes. The most effective structures will put those parts together.

Keeping the scope of a change together simplifies the technical integration, which in turn simplifies the design and implementation of the systems involved. Simplified design is easier to understand, test, secure, support, and fix. "Pitfalls of sharing infrastructure elements" on page 44 outlined the reasoning behind this principle.

When it's not possible to keep the infrastructure affected by a change in a single stack, or within a single team, then it's important to make the interfaces between them simple. Interfaces should be designed to be tolerant, so that a change or problem with one side of the integration doesn't break the other side.

Teams which provide services to other teams should make sure those other teams can customize and configure them without involving the providing team. The model for this is public cloud vendors. These vendors provide APIs and configuration, rather than having a central team make configuration changes on behalf of their customers.

When considering different types of changes and their scope, it's important to cast a wide net. It's important to optimize the most common and simple changes, such as creating a new server of a standard type, or using a Chef cookbook for a common service following the most common use case. But also consider the changes which don't fit into the common use cases. These are the ones which can take the most work to implement if the infrastructure design is not aligned to make them simple.

Example: An infrastructure design for microservices

Microservices is proving to be a popular style of software architecture for deploying to the cloud. Its approach of breaking systems into small-ish, independently deployable pieces of software aligns well with the approach to defining infrastructure described in this chapter. The following describes a design for implementing infrastructure for an example microservices-based application on AWS.

A typical microservice may be composed of an application and a corresponding database schema. The microservice application is deployed to a pool of application servers implemented as an AWS Autoscaling Group.

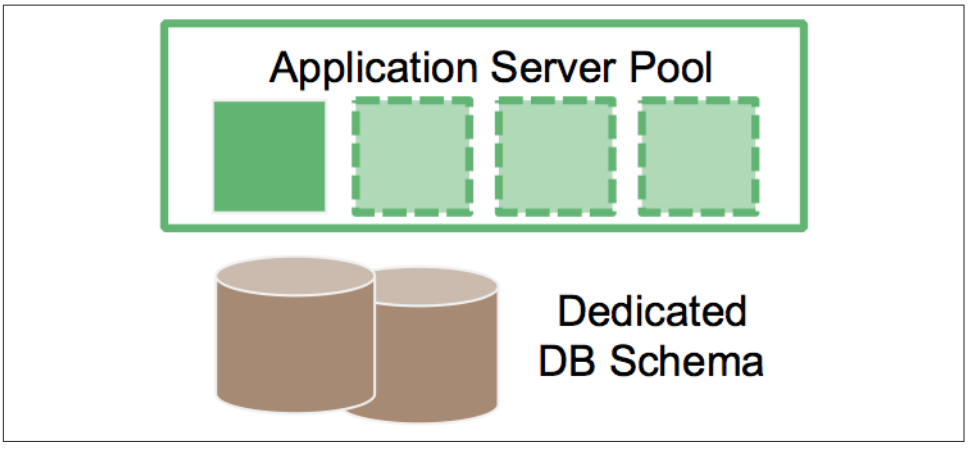

Figure 2-5. Deployable parts of a microservice

The Terraform definition in Example 2-8 can be used to create a pool of application servers running a microservice application. When it is run, parameters are passed in giving it the name and version of the microservice to run, and the name of the environment to run it in. This single definition can be used multiple times to create multiple microservices in multiple environments.

Terraform passes the contents of the `user_data` string to the new server instance using the EC2 API. The AMI `ami-12345678` has the cloud-init tool pre-installed[9].

Cloud-init automatically runs when a new server instance boots, examines the contents of the user_data string, and runs the named script `/usr/local/sbin/microservice-bootstrap.sh`, passing it the arguments `environment`, `microservice_name`, and `microservice_version`. The `microservice-bootstrap.sh` script is written by the infrastructure team, and is built into the AMI image as part of the template building process. This script uses the `microservice_name` and `microservice_version` arguments to download and install the relevant microservice application package. It may also use the `environment` argument to look up configuration parameters in a configuration registry.

9 Cloud-init has become a standard tool - most stock server images for cloud platforms have it pre-installed and configured to run at startup.

Example 2-8. Terraform definition for a microservice autoscaling group

```
variable "microservice_name" {}
variable "microservice_version" {}
variable "environment" {}

resource "aws_launch_configuration" "abc" {
  name = "abc_microservice"
  image_id = "ami-12345678"
  instance_type = "m1.small"
  user_data = "#!/bin/bash
/usr/local/sbin/microservice-bootstrap.sh \
  environment=${var.environment} \
  microservice_name=${var.microservice_name} \
  microservice_version=${var.microservice_version}"
}

resource "aws_autoscaling_group" "abc" {
  availability_zones = ["us-east-1a", "us-east-1b", "us-east-1c"]
  name = "abc_microservice"
  max_size = 5
  min_size = 2
  launch_configuration = "${aws_launch_configuration.abc.name}"
}
```

Each microservice application can use this definition, or one similar to it, to define how it is to be deployed in an environment. This definition doesn't define the networking infrastructure for the application. Although each application could define its own AWS VPC, subnets, security groups, etc., this duplication is potentially excessive as the system grows. These elements should not need to change very often, and should not be very different between applications. So this is a good candidate for sharing.[10]

The diagram below shows the elements of an AWS infrastructure which might be managed in their own global stack.

10 Sharing these kinds of network structures is also recommended by Amazon. See this presentation from Gary Silverman for typical Amazon recommendations.

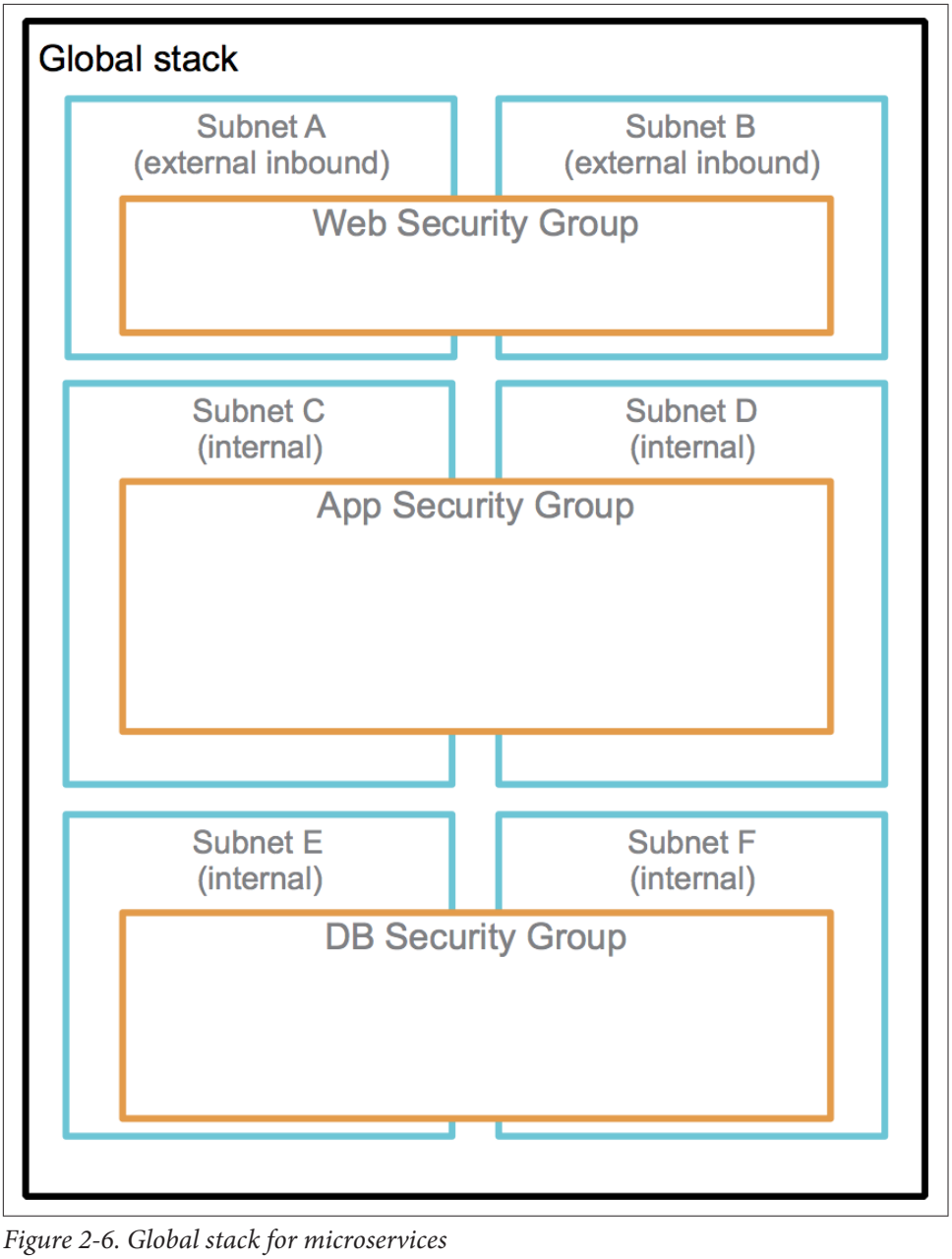

Figure 2-6. Global stack for microservices

This stack would have its own Terraform definition file, which is applied separately for each environment. The IDs of the infrastructure elements provisioned by Terraform will be recorded in a configuration registry.

Each microservice application then has its own Terraform definition file, similar to the example above. The microservice infrastructure would also add an ELB to route traffic to the autoscaling group, and an RDS definition to create a database schema instance for the application. The definition would also reference the configuration registry to pull in the IDs for the subnets and security groups to assign the various parts of the infrastructure to. The result would look like the diagram below.

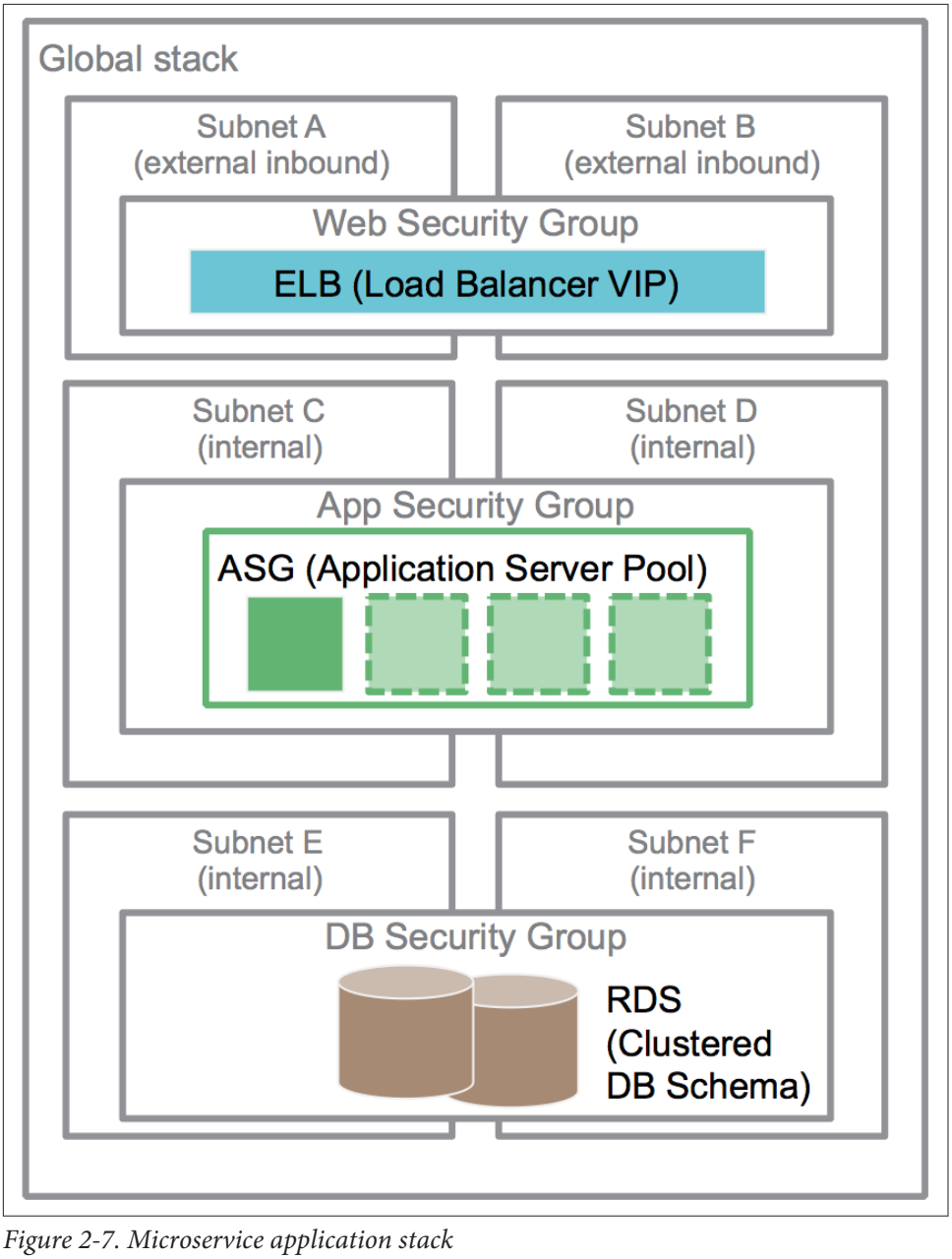

Figure 2-7. Microservice application stack

This architecture is appropriate for a set of microservices which don't need to be segregated from one another. In other situations, segregation might be needed. For example, if a payment service involves handling credit card details it would be subject

to PCI-DSS regulations. In this case, putting the payment service into its own global stack makes it easier to keep it segregated. This can then be managed with more tightly controlled processes to ensure and demonstrate compliance.

Reverse engineered definitions vs. build-forward

One popular feature for a definition tool is a way to point it at existing infrastructure and automatically generate definition files. This offers a quick way to start using a tool like Terraform, CloudFormation, Heat, etc. for teams which already have extensive infrastructure in place.

The risk with this approach is that there is no guarantee the tool will actually be able to rebuild the infrastructure if needed. There may be aspects of the infrastructure which the tool did not detect, or even doesn't support.

The best way to have confidence in the ability to reliably rebuild infrastructure is to "build-forward". That is, write the definition files to match the existing infrastructure, and then run them to create new instances of the infrastructure. The new instances can be tested to make sure they work correctly. Then, the old infrastructure should be replaced by the newly built infrastructure.

If your team does not believe the old infrastructure can safely be replaced by infrastructure that has been built from scratch using the automated definition tools, then this is a sign of a snowflake or even jenga infrastructure ("Jenga infrastructure" on page 16). Consider rebuilding elements of the legacy infrastructure using Infrastructure as Code, one piece at a time, until the entire infrastructure can be easily and reliably rebuilt with confidence.

Automatically generating configuration definitions from existing infrastructure can be useful for learning about infrastructure. Generated definitions can even be used as a starting point for automatically-built infrastructure. But don't make the mistake of considering them a useful recovery point for hand-crafted infrastructure.

Running definition tools

Engineers can run definition tools from their local workstation or laptop to manage infrastructure. But it's better to ensure that infrastructure is provisioned and updated by running tools from centrally managed systems, such as an orchestration agent. An orchestration agent is a server which is used to execute tools for provisioning and updating infrastructure. These are often controlled by a CI or CD server, as part of a change management pipeline.

There are a number of reasons to run definition tools from an agent:

- The agent server can itself be automatically built and configured, ensuring that it can be reliably rebuilt in an emergency situation.
- It avoids relying on snowflake configuration for executing the tool, which often happens when tools are installed and run on an engineer's desktop.
- It avoids dependency on a particular person's desktop.
- It makes it easier to control, log, and track what changes have been applied, by whom, and when.
- It's easier to automatically trigger the tool to be applied for a change management pipeline.
- It removes the need (or temptation) to keep credentials and tools for changing infrastructure on a laptop which could be lost or stolen.

Running a tool locally is fine for working with sandbox infrastructure (???) to develop, test, and debug automation scripts and tooling. But any infrastructure used for important purposes, including change management, should be run from centralized agents.

Conclusion

Part II of this book built on the technical concepts from Part I to outline patterns and practices for using automation tooling effectively for Infrastructure as Code. It should have provided clear guidance on good approaches, as well as pitfalls to avoid.

Part III will explore a variety of practices that can help to make this work. These will delve into practices and techniques derived from software development, testing, and change management.

About the Author

Kief Morris has been designing, building, and running automated IT server infrastructure for nearly twenty years, having started out with shell scripts and Perl, moving on to CFEngine, Puppet, Chef, and Ansible among other technologies as they've emerged. He is a consultant for ThoughtWorks, helping clients with ambitious missions take advantage of cloud, infrastructure, and Continuous Delivery. Originally from Tennessee and Florida, he is now based in London.