Infrastructure as Code

Kief Morris

Beijing • Cambridge • Farnham • Köln • 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: FIX ME!
Copyeditor: FIX ME!
Proofreader: FIX ME!

Indexer: FIX ME!
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Rebecca Demarest

September 2015: First Edition

Revision History for the First Edition:

2015-03-06: First early release revision

2015-03-25: Second early release revision

2015-04-15: Third early release revision

2015-07-20: Fourth early release revision

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

ISBN: 978-1-491-92435-8

[?]

Table of Contents

CHAPTER 1
Challenges and Principles

The new generation of infrastructure management technologies promises to transform the way we manage IT infrastructure. Virtualization, cloud, containers, server automation, software defined networking - these should take the drudgery out of our IT operations work. Routine maintenance will happen automatically, basic problems will be automatically corrected, and systems will keep themselves up to date. We'll spend our time on interesting, technically challenging work, making forward-looking improvements that are truly valuable for our organization.

Unfortunately, while most IT teams are adopting at least some of these technologies, they often don't make a dramatic difference to the way they spend their time. The team may find the new tools helpful, but still find themselves swamped. They spend most of their time firefighting, handling routine user requests, and carrying out repetitive tasks.

The problem is that these teams haven't really changed how they work. They're using new generation tools to work in old generation ways. So they aren't seeing particularly different results. However, some IT teams are taking advantage of the new technologies to radically change the way they handle their work. This can be described as Infrastructure as Code[1].

Virtualization, cloud, and containers have turned infrastructure into software and data. This allows infrastructure to be treated like a software system. What's more, it has become dynamic. Its elements are ephemeral, portable, and constantly changing. The people who manage infrastructure don't need to personally make changes to it, instead

1. 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.

they can build systems that do it for them. They can develop, test, and deploy these systems using the same engineering practices and tooling that have been been proven effective for application development.

Teams who treat their infrastructure as a software system can scale the number of systems they manage without needing to scale their team size to match. They can make changes and improvements to their production infrastructure frequently and rapidly, with short turnaround times, low failure rates, and fast fix times.

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.

Organizations of all types are discovering that software is essential to what they do. As a result, their IT infrastructure is continuously growing in size, complexity, and importance. The software they manage is constantly evolving. This requires infrastructure to not just cope with change, but to enable continuous, rapid change as a BAU. So the principles and practices of Infrastructure as Code that have emerged from large-scale online businesses can help nearly any type of organization become more effective.

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.

The new wave of infrastructure technology

Systems administrators have been using scripts and tools to carry out routine infrastructure management tasks for decades. But the new generation of infrastructure technology makes automation fundamental to the way infrastructure is managed.

Virtualization lays the groundwork for infrastructure as code, by decoupling infrastructure concepts like compute, storage, and networking from the hardware resources that provide them. This has several benefits:

- Applications which don't use much resource can share hardware with other applications, to keep utilization higher,
- Workloads can be shifted across a pool of hardware to utilize it all more efficiently,
- Computing resources can be provisioned quickly and easily, as long as there is capacity somewhere in the infrastructure.

Adding a cloud interface model for managing virtualized resources adds more benefits:

- Computing resources can be allocated, provisioned, and freed up programmatically, using scripts or software systems,
- Resources can be allocated on demand by the users who need them, in a self-service model,
- The cost of computing resources can be spread across larger groups of users, either inside an organization, between related organizations, or even with the public.

So virtualization and cloud have the potential to make hardware resources more directly available and responsive to the needs of the people who are more directly using them. Implemented well, this frees those of us managing the infrastructure from having getting involved in the routine usage of the services we manage. (to come) goes into more detail on the tooling and technology to manage computing resources dynamically.

Automated configuration management tools are another category of tools which have become mainstream over the past decade or so. The new generation of tools - pioneered by CFengine, popularized by Puppet and Chef, and carried on by tools like Ansible and Saltstack - bring a software development sensibility to infrastructure management. The way computing resources are allocated and configured is captured in files which can be managed in source control systems, automatically tested, and orchestrated in the same way that software releases are managed. Chapter 2 discusses these tools in more detail.

One benefit of these tools is that software engineering practices can be applied to infrastructure. Test Driven Development (TDD), Continuous Integration (CI), Continuous Delivery (CD), and similar practices provide rigorous quality controls with a rapid pace of change. The secret to the success of high performing infrastructure teams is that they leverage principles that have been proven in software development to achieve high quality through rapid change.

The Iron Age and the Cloud Era

I often refer to the "Iron Age of IT", and the "Cloud Era". In the iron age, computing was bound directly to the hardware. A server was a physical thing. Making changes took a while. Adding a server to our infrastructure involved a purchasing process, and days or weeks of waiting for packages to show up so we could assemble them. The pace of change was slow, and the cost of a mistake was high. So we took our time to make decisions, and to plan and design. If we realized we needed to change a decision we'd made earlier, we'd have meetings to review documents to understand the impact.

In the Cloud Era, computing has been unshackled from the hardware. A server is a process that can be started or stopped in minutes or seconds. Mistakes are quickly caught

and corrected, which means they're less expensive. This lets us choose the right design and implementation for something by building the simplest thing, and then iterating it.

Is the cloud emperor really wearing new clothes?

But while most IT teams have adopted some degree of virtualization, cloud, and configuration automation, not many get the step-change improvements that we hear about in presentations and blog posts. There are a few common situations teams find themselves in.

New tools, old habits

One common situation is that not much changes. It may be easier to physically build a new server, or to make a configuration change across many servers, but it still takes a person's time to make it happen. So user requests still go into a queue, and engineers still spend their time firefighting, handling routine user requests, and carrying out repetitive tasks.

This happens when the team carries on with their old working habits, rather than using the tools to help them change the way they manage their work.

Strangled automation

A variation of this often happens with larger organizations. The time to carry out a common task like provisioning a server is cut to minutes using expensive automation and cloud tooling. But a large number of different departments and stakeholders need to be involved to deliver the task, and complex governance requirements must be met. As a result, even when the technical implementation takes minutes, the end to end time to deliver may take days or weeks.

Again, this can be avoided by using the new tools in new ways. Automation can be implemented to allow changes to be delivered quickly, while providing stronger controls and governance than traditional change management processes.

Server sprawl

A contrasting situation is where people can use cloud or virtualization to quickly create new servers on demand, resulting in too many servers to manage effectively. Servers can't be easily kept up to date with patches, especially as custom configurations are made so servers can carry out different roles. Problems pop up and are fixed tactically, leading to an inconsistent sprawl of servers. This inconsistency means that automation tools don't run easily across all of the servers, making it still harder to keep everything together.

But a strong infrastructure as code approach can ensure even a large infrastructure with diverse user requirements can be kept consistent and well-maintained.

What is Infrastructure as Code?

Infrastructure as code is an approach to using cloud era technologies to build and manage dynamic infrastructure. It treats the infrastructure, and the tools and services that manage the infrastructure itself, as a software system, adapting software engineering practices to manage changes to the system in a structured, safe way. This results in infrastructure with well tested functionality to manage routine operational tasks, and a team that has clear, reliable processes for making changes to the system.

Dynamic infrastructure

The components of a dynamic infrastructure[2] change continuously and automatically. Servers may appear or disappear so that capacity is matched with load, to recover from failures, or to enable services. These changes could be triggered by humans, for example a tester needing an environment to test a new software build. Or the system itself may apply changes in response to events such as a hardware failure. But the process of creating, configuring, and destroying the infrastructure elements happens without a person being involved.

The foundation of infrastructure as code is the ability to treat infrastructure elements as if they were data. Virtualization and cloud hosting platforms decouple infrastructure from its underlying hardware, providing a programmatic interface for managing servers, storage, and network devices. LOM (Lights Out Management) and other tooling also make it possible to manage operating systems installed directly on hardware in a similar way, which makes it possible to use infrastructure as code in non-virtualized environments as well. Automated infrastructure management platforms are discussed in more detail in (to come).

Automated configuration management tools like Ansible, Cfengine, Chef, Puppet, and Salt (among others) allow infrastructure elements themselves to be configured using specialized programming languages. IT operations teams can manage configuration definitions using using tools and practices which have been proven effective for software development, including Version Control Systems (VCS), Continuous Integration (CI), Test Driven Development (TDD), and Continuous Delivery (CD). Configuration management tools are discussed in Chapter 2, and software engineering practices are covered throughout this book.

2. Some further reading on dynamic infrastructures include: Web Operations, by Allspaw, Robbins, et. al.; and The Practice of Cloud System Administration, by Limoncelli, Chalup, and Hogan.

So an automated infrastructure management platform and server configuration tools are a starting point, but they aren't enough. The practices and approaches of the Iron Age of Infrastructure - the days when infrastructure was bound to the hardware it ran on - don't cope well with dynamic infrastructure, as we'll see in the next section. Infrastructure as code is a new way of approaching dynamic infrastructure management.

Goals of infrastructure as code

These are some of the goals that infrastructure as code support:

- IT infrastructure should support and enable change, not be an obstacle or a constraint.
- IT staff should spend their time on valuable things which engage their abilities, not on routine, repetitive tasks.
- Users should be able to provision and manage the resources they need, without needing IT staff to do it for them.
- Teams should know how to recover quickly from failure, rather than depending on avoiding failure.
- Changes to the system should be routine, without drama or stress for users or IT staff.
- Improvements should be made continuously, rather than done through expensive and risky "big bang" projects.
- Solutions to problems should be proven through implementing, testing, and measuring them, rather than by discussing them in meetings and documents.

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.

Configuration drift

Servers may be consistent when they're created, but over time differences creep in.

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, and now the Jira server is different from other servers that have Java installed. Three different people install IIS on three different web servers over 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 the ones with low 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

Years ago the company I was with 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 not on our staging environment. Don't ask why we upgraded production to 5.8 without discovering the problem with our test environment, but that's how we ended up. We had a 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 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 stories that can'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 infra-

3. Sadly, the infrastructures.org site hadn't been updated since 2007 when I last looked at it.

structure. 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, scaled up.

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.

> 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 touched the server again.

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.

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.

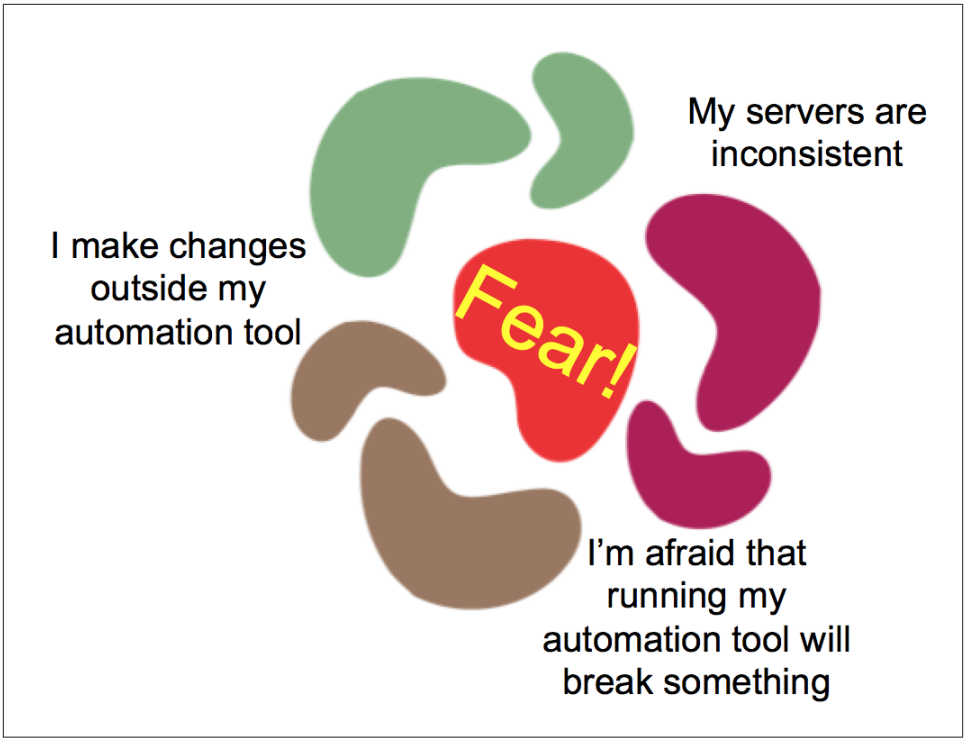


Figure 1-1. The automation fear spiral

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.

Habits that lead to these problems

In this section, we'll look at some of the habits that teams fall into that can lead to problems like snowflake servers and configuration drift. These habits tend to reinforce each other, each habit creating conditions which encourage the others.

Doing routine tasks manually

When people have to spend their time doing routine things like setting up servers, applying updates, and making configuration changes across a group of servers, not only does it take their time and attention away from more important work, it also leads to things being done inconsistently. This in turn leads to configuration drift, snowflakes, and the various other evils we've discussed.

Running scripts manually

Most infrastructure teams write scripts for at least some of their routine work, and quite a few have adopted automated configuration tools to help with this. But many teams run their scripts by hand, rather than having them run unattended on a schedule, or automatically triggered by events. A human may need to pass parameters to the script, or just keep a close eye on it and check things afterwards to make sure everything has worked correctly. Using a script may help keep things consistent, but still takes time and attention away from more useful work.

Needing humans to babysit scripts suggests that there are not enough controls - tests and monitoring - to make the script properly safe to run. This topic is touched on throughout this book, but especially in Part III. Team members also need to learn habits for writing code that is safe to run unattended, such as error detection and handling.

Applying changes directly to important systems

Most mature IT organizations would never allow software developers to make code changes directly to production systems, without testing the changes in a development environment first. But this standard doesn't always apply to infrastructure. I've been to organizations which put their software releases through half a dozen stages of testing and staging, and seen IT operations staff routinely make changes to server configurations directly on production systems.

I've even seen teams do this with configuration management tools. They make changes to Puppet manifests and apply them to business critical systems without any testing at all. Editing production server configuration by hand is like playing Russian Roulette with your business. Running untested configuration scripts against a groups of servers is like playing Russian Roulette with a machine gun.

Running automation infrequently

Many teams who start using an automated configuration tool only run it when they need to make a specific change. Sometimes they only run the tool against the specific servers they want to make the change on.

The problem with this is that the longer you wait to run the configuration tool, the more difficult and risky it is to run the tool. Someone may have made a change to the server - or some of the servers - since the last time it was run, and that change may be incompatible with the definitions you are applying now. Or someone else may have changed the configuration definitions, but not applied them to the server you're changing now, which again means more things that can go wrong. If your team is in the habit of making changes to scripts for use with specific servers, there's an even greater chance of incompatibilities.

This of course leads to automation fear, configuration drift, and snowflakes.

Bypassing automation

When scripts and configuration tools are run infrequently and inconsistently, and there is little confidence in the automation tools, then it's common to make changes outside of the tools. These are done as a supposed one-off to deal with an urgent need, with the hope that you'll go back and update the automation scripts later on.

Of course, these out-of-band changes will cause later automation runs to fail, or to revert important fixes. The more this happens, the more unreliable and unusable our configuration scripts become.

Good habit - rolling changes into automation

Even with a comprehensive automated infrastructure, it's sometimes useful to just log onto a server and fix something by hand. When critical services are down the first priority is to get things working, *now*. And sometimes the easiest way to find the right configuration setting is to manually tweak it on a running test server.

But, as soon as things are stable or the right setting is found, changes need to be folded back into the configuration system immediately. This ensures the change is rolled out and made consistent across all of the relevant servers. It also avoids regressions, where automation reverts that critical fix.

However, making configuration changes directly on production servers is an obvious smell. If people feel that production is the only place they can reliably test whether a change works, then you need an easier and more reliable workflow for developing and testing infrastructure changes. See the chapter on infrastructure developer workflow ((to come)) for more approaches to this.

So a key habit for effective infrastructure teams is to ensure that any change or task that needs to be carried out manually is rolled into the automation.

Principles of Infrastructure as Code

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

Principle: Reproducibility

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 enables many powerful capabilities of infrastructure as code. It cuts down on much of the fear and risk of infrastructure operations. Servers are trivial to replace, so they can be treated as disposable. Reproducibility supports continuous operations (as discussed in (to come)), automated scaling, and creating new environments and services on demand.

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

Principle: Consistency

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 repeatably 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 Chapter 7.

Principle: Repeatability

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 configuration 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 as 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: Disposability

> "Treat your servers like cattle, not pets."[5]
>
> — Bill Baker

We should assume that any infrastructure element can and will be destroyed at any time, without notice. It could be destroyed unexpectedly, when hardware fails, or it could be deliberately destroyed in order to reduce capacity or to replace it. If we design our services so that infrastructure can disappear without drama, then we can freely destroy and replace elements whenever we need, to upgrade, to reconfigure resources, to reduce capacity when demand is low, or for any other reason.

This idea that we can't expect a particular server to be around tomorrow, or even in a few minutes time, is a fundamental shift. Logging into a server and making changes manually is pointless except for debugging or testing potential changes. This requires that any change that matters be made through the automated systems that create and configure servers.

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 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.

5. The cattle/pets analogy was been attributed to former Microsoft employee Bill Baker, according to Cloud-Connect CTO Randy Bias in his presentation Architectures for open and scalable clouds. I first heard the analogy in Gavin McCance's presentation CERN Data Centre Evolution. Both of these presentations are excellent.

Principle: Service continuity

Given a service hosted on our infrastructure, that service must be continuously available to its users even when individual infrastructure elements disappear.

To run a continuously available service on top of disposable infrastructure components, we need to identify which things are absolutely required for the service, and make sure they are decoupled from the infrastructure. This tends to come down to request handling and data management.

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 data management chapter ((to come)) has ideas for this.

State and transactions are an especially tricky consideration. Although there are technologies and tools to manage distributed state and transactions, in practice the most reliable approach is to design software not to depend on this. The twelve-factor application methodology is a helpful approach. For larger, enterprise-scale systems, microservice architecture has proven effective for many organizations.

The chapter on continuity ((to come)) goes into techniques for continuous service and disaster recovery.

Principle: Self-testing systems

Effective automated testing is one of the most important practices infrastructure operations teams can adopt from software development. The highest performing software development teams I've worked with make automated testing a core part of their development process. 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.

The main benefit they see from this is fast feedback as to whether the changes they're making work correctly and without breaking other parts of the system. This immediate feedback gives the team confidence that errors will usually be caught quickly, which then gives them 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 (to come)). Good testing practices are the key to eliminating automation fear.

However while most software development teams aspire to use automated testing, in my experience the majority struggle to do it rigorously. There are two factors that make it difficult. The first is that it takes time to build up the skills, habits, and discipline that make test writing a routine, easy thing to do. Until these are built up, writing tests tends to make things feel slow, which discourages teams and drives them to drop test writing in order to get things done.

The second factor that makes effective automated testing difficult is that many teams don't integrate test writing into their daily working process, but instead write them after much of their code is already implemented. In fact, tests are often written by a separate QA team rather than by the developers themselves, which means the testing is not integrated with the development process, and not integrated with the software.

Writing tests separately from the code means the development team doesn't get that core benefit of testing - they don't get immediate feedback when they make a change that breaks something, and so they don't get the confidence to code and deliver quickly.

Chapter 9 explores practices and techniques for implementing testing as part of the system, and particularly how this can be done effectively for infrastructure.

Principle: Self-documenting systems

A common pattern with IT teams is the struggle to keep documentation relevant, useful, and accurate. When a new tool or system is implemented someone typically takes the time to create a beautiful, comprehensive document, with color glossy screenshots with circles and arrows and a paragraph for each step. But it's difficult to take the time to update documentation every time a script or tool is tweaked, or when a better way is discovered to carry out a task. Over time, the documentation becomes less accurate. It doesn't seem to matter what tool or system is used to manage the documentation - the pattern repeats itself with expensive document management systems, sophisticated collaboration tools, and simple, quick to edit wikis.

And of course, not everyone on the team follows the documented process the same way. People tend to find their own shortcuts and improvements, or write their own little scripts to make parts of the process easier. So although documenting a process is often seen as a way to enforce consistency, standards and even legal compliance, it's generally a fictionalized version of reality.

A great benefit of the infrastructure as code approach is that the steps to carry out a process are captured in the scripts and tooling that actually carry out that process. Documentation outside the scripts can be minimal, indicating the entry points to the tools, how to set them up (although this should ideally be a scripted process itself), and where to find the source code to learn how it works in detail.

Some people in an organization, especially those who aren't directly involved with the automation systems, may feel more comfortable having extensive documentation outside the tools themselves. A useful exercise is to consider the use cases for documentation, and then agree on how each use case will be addressed. For example, a common use case is a new technical team member joins and needs to learn the system. This can be addressed by fairly lightweight documentation and whiteboarding sessions to give an overview, and learning by reading and working with the automated scripts.

One use case that tends to need a bit more documentation is non-technical end users of a system, who won't read configuration scripts to learn how to use it. Ideally, the tooling for these users should have a clear, straightforward user experience, with the information needed to make decisions included in the interface. A rule of thumb for any documentation should be, if it becomes out of date, and work goes on without anyone noticing, then that document is probably unnecessary.

Principle: Small changes

When I first got involved in developing IT systems, my instinct was to complete the whole chunk of work I was doing 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
- When something goes wrong with a big batch of changes, you often need to delay everything, including useful changes, while you fix the small broken thing
- 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.

Principle: Version all the things

Versioning of infrastructure configuration is the cornerstone of infrastructure of code. It makes it possible to automate processes around making changes, including tests and auditing. Reasons why VCS is essential for infrastructure management include:

- Traceability - you get a history of changes that were made, who made them, and potentially context about why. This can be very helpful when debugging problems. I often forget about a minor change I've made recently when things blow up, because it really shouldn't have anything to do with the explosion - until we go back over the commit history and think again.
- Rollback - when a change breaks something, and especially when multiple changes break something, it's nice to be able to restore things to exactly how they were before, rather than struggling to restore them to how we *think* they were before.
- Correlation - when scripts, configuration, artifacts, and all the things 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.
- Transparency - the types of files which are normally committed to a VCS tend to be more human-readable - scripts, configuration files, definitions, etc. - which makes them easier for people to understand than things expressed through a GUI and stored in a black box tool.
- Actionability - version control systems support automatically triggering actions when a change is committed. This is a key to enabling Continuous Integration and Continuous Delivery pipelines.

Chapter 2 explains how VCS works with configuration management tools, and Chapter 8 discusses approaches to managing your infrastructure code and definitions.

Selecting tools

Too many organizations start with tools. It's appealing for managers to select a vendor that promises to make everything work well, and it's appealing to technical folks to find a shiny new tool to learn. But this leads to failed projects, often expensive ones.

I recommend going through this book and other resources, and deciding what infrastructure management strategy makes sense for you. Then choose the simplest, least expensive tools you can use to explore how to implement the strategy. As things progress

and your team learns more, they will inevitably find that many of the tools they've chosen aren't the right fit, and new tools will emerge that may be better.

The best decision you can make on tools is for everyone to agree that tools will change over time.

This suggests that paying a large fee for an expensive tool, or set of tools, is unwise unless you've been using it successfully for a while. I've seen organizations which are struggling to keep up with technically-savvy competition sign ten-year, seven-figure licensing agreements with vendors. This makes as much sense as pouring concrete over your Formula One car at the start of a race.

Select each tool with the assumption that you will need to throw it out within a year. Hopefully this won't be true of every tool in your stack, but avoid having certain tools which are locked in, because you can guarantee those will be the tools that hold you back.

When are we finished building the infrastructure?

Reading through the principles outlined above may give the impression that automation should (and can) be used to create a perfect, factory-like machine. You'll build it, and then sit back, occasionally intervening to oil the gears and replace parts that break. Management can probably cut staffing down to a minimal number of people, maybe less technically skilled ones since all they need to do is watch some gauges and follow a checklist to fix routine problems.

But this isn't the real world. The needs of an organization aren't static, so the infrastructure that supports the organization can't be static, either. There is always a need to add new capabilities and support new services. The value of infrastructure as code is that it makes it easy to change and adapt the infrastructure, quickly and with confidence.

It doesn't help that IT infrastructure, as a field, is still in the pioneering phase. An IT infrastructure today isn't a standard, off the shelf commodity, as much as we wish it was. Many software vendors and systems integrators pitch their wares as if their consultants will show up, plug everything in, and hand over the keys. But they're only playing to our wishful thinking. A look at the history of their products, and planned roadmap, shows how much they are in flux. As of mid-2015, vendors are rushing to support lightweight containerization technologies that they hadn't heard of two years ago.

Any system installed today either needs to be continuously evolved by staff who understand it to a deep level, or thrown away in two or three years. Often both.

When planning to build of an automated infrastructure, make sure the people who will run and maintain it are involved in its design and implementation. Running the infrastructure is really just a continuation of building it, so the team needs to know how it

was built, and should have made the decisions about how to build it, so they will be in the best position for continued ownership.

So what's the answer to the question in the title of this section is, we'll finish building the infrastructure when we finish using it.

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 apply to IT architecture.

The key to an antifragile 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 prevent it from happening again.

A team typically handles an incident by first making a quick fix, so service can resume, then working out the changes needed to fix the underlying cause, to prevent the issue from happening again. Tweaking monitoring to alert when the issue happens again is often an afterthought, something nice to have but easily neglected.

A team striving for antifragility will make monitoring, and even automated testing, the second step, after the quick fix and before implementing the long term fix.

This may be counter-intuitive. Some systems administrators have told me it's a waste of time to implement automated checks for an issue that has already been fixed, since by definition it won't happen again. But in reality, fixes don't always work, may not resolve related issues, and can even be reversed by well-meaning team members who weren't involved in the previous incident.

Add a monitoring check to alert the team if the issue happens again. Implement automated tests that run when someone changes configuration related to the parts of the system that broke.

Implement these checks and tests before you implement the fix to the underlying problem, then reproduce the problem and prove that your checks really do catch it. Then implement the fix, re-run the tests and checks, and you will prove that your fix works. This is Test Driven Development (TDD) for infrastructure!

The Software Practices chapter and the Pipeline chapter go into more details on how to do this sort of thing routinely.

The secret ingredient of anti-fragile IT systems*

It's people! People are the element that can cope with unexpected situations and adapt 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.

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.

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.[6]
- Team members feel their work is adding measurable value to the organization.

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

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

CHAPTER 2
Server Management Tools

Using scripts and automation tools to create, provision, and update servers is not especially new, but the landscape of tools and approaches for using them has changed quite a bit over the past decade or so. A new generation of specialized tools for automatically configuring servers has emerged over the past decade or so. The rise of virtualization and cloud has driven the need for these kinds of tools. Infrastructure teams who are constantly creating servers have picked up these tools to quickly apply common configurations.

Part II of this book goes into details of patterns and practices for managing servers. This chapter discusses the types of tools used to implement those patterns and practices. The goal is to understand the characteristics of tools that enable managing infrastructure as code. There are many tools available for automating infrastructure, but not all of them are designed with an infrastructure as code approach. Understanding how tools can support this approach not only helps to select suitable tools, but also to understand how to apply the concepts.

This chapter is focused on tools for creating and configuring servers. The next chapter will go into services and tools that involve multiple servers.

Goals for automated server management

Using infrastructure as code to manage server configuration should result in the following:

- A new server can be completely provisioned on demand, without waiting more than a few minutes
- A new server can be completely provisioned without human involvement, for example in response to events

- Once a server configuration change is defined, it is applied to servers without human involvement
- Each change is applied to all the servers it is relevant to, and is reflected in all new servers provisioned after the change has been made
- The processes for provisioning and for applying changes to servers are repeatable, consistent, self-documented, and transparent
- It is easy and safe to make changes to the processes used to provision servers and change their configuration
- Automated tests can be run every time a change is made to server configurations, and to the processes for provisioning and changing servers, can be automatically tested
- Changes to configuration, and changes to the processes that carry out tasks on an infrastructure, can be versioned and applied to different environments, to support controlled testing and staged release strategies

Server management tools need to support these goals. The next section discusses how to identify tools that do this well.

Automatically generating documentation

I've seen a support team that insisted that a software deployment script written in Ant wasn't sufficient documentation, since they wanted to be able to replicate the deployment process manually. My colleague Tom wrote a simple task for Ant that generated a document with the exact steps used to deploy, down to the command lines to type, from the deployment script. His team's automated build process 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 extra work to document.

Guidelines for tooling to support automated server management

The core idea of infrastructure as code is in the name - infrastructure is treated as a software system. Infrastructure components, configuration, and the processes that manage them are defined and run like software, so that software engineering and testing practices can be applied to them. This is a well-proven approach to having an infrastructure which is not only easy to manage, but also easy to change safely.

Tools are a common friction point in implementing infrastructure as code. There are many infrastructure management tools which aren't designed to fit into an open auto-

mation system. Even with tools which are designed to work this way, teams don't always understand how to change their working practices to take full advantage of them.

Here are a few principles around tooling for infrastructure as code. These are described as "prefer A over B"[1], because they aren't absolutes. There are good reasons for preferring the first option, but there are always cases where the second option makes sense.

Prefer unattended execution over interactive use

Most systems administrators start out writing scripts to help them carry out tasks. If the series of commands used to configure a new virtual host on a web server can be captured in a script, this meets many of the server automation goals mentioned earlier. The process becomes repeatable, consistent, and transparent, no matter who runs the script.

But a script doesn't free infrastructure team members to focus on other things, because someone still needs to take the time to run the script. And because someone is there to run it, it's easy to leave some surrounding manual activities. I've seen scripts that need to be edited each time they're run in order to set parameters. Tasks like editing DNS entries can be trickier to automate, so tend to be left as a manual step. And it's down to the human to see whether the script ran correctly, and clean up any left over problems and loose ends.[2].

Tasks that can be reliably carried out without human involvement are building blocks of an automated infrastructure. They can be automatically applied and validated in a test environment before being applied to production systems. They can be used to automatically add, remove, recover, and scale infrastructure elements as needed.

Ad-hoc synchronization leads to the automation fear spiral

I visited a financial services company's systems team, who had been using Puppet for nearly a year. They were writing manifests and checking them into source control. But they only applied the manifests when they had a specific change to make. They would tweak the manifests, making the specific change they wanted and choosing which machines they would apply to. More often than not, Puppet would run with errors or even fail on at least some of the machines.

1. This "prefer A over B" form is inspired by the values section of the Agile Manifesto
2. In The Practice of Cloud System Administration, Limoncelli, Chalup, and Hogan talk about tool building versus automation. Tool building is writing a script or tools to make a manual task easier. Automation is eliminating the need for a human to carry out the task. They make the comparison to an auto factory worker using a high powered paint sprayer to paint car doors, versus having an unattended robotic painting system.

This was a perfect example of "Running scripts manually" on page 10, one of the bad habits described in Chapter 1 that leads to the automation fear spiral.

The systems team I visited decided it was time to move beyond treating Puppet as a scripting tool, and learn how to run it continuously. They sent their team on Puppet training, and began migrating their servers into a continuously synchronized regime. A year later they had far more confidence in their automation, and were exploring ways to increase their use of automated testing to gain even more confidence.

These are some characteristics of scripts and tasks that for reliable unattended execution:

- Idempotent. It should be possible to execute the same script or task multiple times without bad effects.
- Pre-checks. A task should validate its starting conditions are correct, and fail with a visible and useful error if not.
- Post-checks. A task should check that it has succeeded in making the changes. This isn't just a matter of checking return codes on commands, but proving that the end-result is there. For example, checking that a virtual host has been added to a web server could involve making an http request to the web server.
- Visible failure. When a task fails to execute correctly, it should be visible to the team. This may involve an information radiator ("What is an information radiator?" on page 56) and/or integration with monitoring services ("Alerting - tell me when something is wrong" on page 55).
- Parameterized. Tasks should be applicable to multiple operations of a similar type. For example, a single script can be used to configure multiple virtual hosts, even ones with different characteristics. The script will need a way to find the parameters for a particular virtual host, and some conditional logic or templating to configure it for the specific situation.

Implementing this takes discipline for the infrastructure team. Be ruthless about finding manual tasks that can be automated. Use good coding practices to ensure scripts are robust. And uproot tasks that are difficult to automate, even if it means replacing major pieces of the infrastructure with ones that offer better support for unattended automation.

Idempotency

Any tool which will be used for continuous synchronization of server configuration needs to be idempotent. The result of running the tool should be the same no matter how many times it's run.

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

```
echo "spock:*:1010:1010:User account for Spock:/home/spock:/bin/sh" >> /etc/passwd
```

Running this script once may have the result we want - the user *spock* is added to the /etc/passwd file. But running it multiple times will end up with many duplicate entries for the user.

A good DSL for server configuration works by having you define the state you want something to be in, and then doing whatever is needed to bring it into that state, without side effects from being applied to the same server many times, as with the following Puppet example:

```
user { "spock":
  ensure => present,
  gid    => "science",
  home   => "/home/spock",
  shell  => "/bin/sh"
}
```

Prefer externalized configuration over black boxes

The configuration of an infrastructure management tool can be managed externally, in files that are accessible and editable by common text editing tools. Or its configuration can be managed internally, in a "black box" editable only through the tool itself.

The black box configuration pattern is intended to simplify management. The tool can offer a helpful interface to users, only presenting valid options. But the externalized configuration pattern tends to be more flexible, especially when used as part of an ecosystem of similar infrastructure tooling.

Lessons from software source code

The externalized configuration pattern mirrors the way software source code works. There are some development environments which keep the source code hidden away, including scripting inside Microsoft Office, and even comprehensive development systems like Visual Basic, (Borland stuff, others??). But the dominant model is keeping programming source code in external files.

An IDE (Integrated Development Environment) can provide an excellent user experience for managing and editing externalized source files, while still keeping the source available to other tools and systems outside the IDE. This gives development teams, and even individual developers in a team, the freedom to choose the tools they prefer.

Some IDE's have support for server configuration definitions like Chef recipes and Puppet manifests. This support doesn't tend to be as strong as support for general purpose programming languages like Java and C#. But it's possible that the increasing popularity of infrastructure automation will bring better IDE support.

The biggest advantage of externalized configuration files is that they can be managed in a standard VCS like Subversion, Git, Perforce, etc. This supports the principle of "Principle: Version all the things" on page 18, with the following benefits to the team using it:

- Changes are versioned. The VCS keeps a full history of changes, the ability to compare changes between arbitrary versions, and to roll back to previous versions if needed.
- Work can be safely shared. Each team member can run their own instance of the tool, loading in the current configuration versions from the VCS. They can then try out changes and commit them back to the VCS when they're ready. Tools with black box configurations often force people to work on a shared instance, which become polluted with unfinished and abandoned experiments over time.
- Changes can be easily replicated and promoted between instances or environments by version. This supports a staging model, where changes are applied to a test environment and automatically validated before being promoted to production.
- Changes are transparent. A VCS makes it easy to review the history of commits to see what's been done. This can be helpful in untangling problems, and making everyone on the team aware of what changes are made.
- A VCS may allow teams to correlate the versions of changes across different tools, to provide stronger auditability and tracking.

- Triggerability. VCS tools can be configured to run actions when changes are committed. This is the basis for Continuous Integration (CI), and an efficient and maintainable automating testing suite (seeChapter 9).

Some black box automation tools embed their own VCS functionality. But since this isn't the tool's core function, it's unlikely to be as fully featured as a standalone VCS. Externalized configuration allows teams to select the VCS with the features they need, and to swap it out later if they find a better one.

Externalizing black box configuration

Many black box automation tools allow their configuration to be exported, which can be used to integrate them into an infrastructure as code regime. Dumps of the configuration can be managed in a VCS or an artifact repository (see (to come)). Team members can pull the latest configuration dump from the repository and load it into an instance of the automation tool to make changes. Once they changes are done, they can be exported and pushed back into the repository, to be loaded into downstream testing and production instances.

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

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

Another way to approach black box configuration is by injecting configuration automatically. Ideally this would be done using an API provided by the tool. However, I have seen it done by automating interaction with the tool's UI, for example using scripted tools to make HTTP requests and form posts. This tends to be brittle, however, since the tool vendor doesn't guarantee consistency for a UI as they will for a supported API.

With this approach, the team defines the configuration they want in external files. These may simply be scripts, or may involve a configuration format or DSL, as described below in "Prefer definitions over scripts" on page 31. Changes are not made using the tool's UI, to avoid conflicts and keep configuration consistent and repeatable from one instance to the next. This approach can be clunky, and comes with a maintenance overhead, but effectively turns a black box tool into one with externalized configuration that is compatible with infrastructure as code.

Another powerful advantage of tools with externalized configuration is that it's easier to integrate them with other tools, write your own tools, or even to use off the shelf tools designed to work with text files. For example, you can use basic command line tools like find, grep, and wc to analyze and report on your configuration. You can use scripting tools like bash, sed, batch scripts, or awk, or more advanced scripting languages like Perl, PowerShell, Python, or Ruby to carry out maintenance and administrative tasks.

The use of these kinds of tools to take full ownership of customizing infrastructure management is common, arguably universal, among the most effective teams. Teams which allow themselves to be constrained by the options exposed by a black box tool's UI struggle to get the same levels of productivity.

The basics of using a VCS

Modern development teams use a VCS without even thinking about it - aside from debating the merits of which one to use. For many systems administration teams, however, it's not a natural part of their workflow.

The VCS acts as the hub of infrastructure as code. Anything that can be done to infrastructure is captured in scripts, configuration files, and definition files checked into version control. When someone wants to make a change to any of these, they pull the latest version of the files, make the changes, and then commit them back in. They might test their changes on a throwaway environment before committing, but the changes are only applied to anything serious - even test environments - after being committed to the repository.

Committing a change into the repository makes it available for use. Teams which use a change management pipeline (as described in (to come)) will have these changes automatically applied and tested on test infrastructure before they are promoted to be used in more important environments.

I've seen at least one team that only commits their changes to version control after they've applied them to systems. For them, the VCS is just a library of scripts that they modify and run manually.

The reason for using the VCS as a hub to drive changes from, rather than as a library, is that it acts as a single source of truth. If two different team members check out files and work on changes to the infrastructure, they could make incompatible changes. For example, I might be adding configuration for an application server while Jim is making a firewall rule change that blocks access to my server. If we each apply our change from our local workstation, we have no visibility of what the other is doing. We may reapply our changes to try to force what we want, but we'll keep undoing each other's work.

But if we each need to commit our change in order for it to be applied, then we should be able to see what's going on very quickly. The current state of the files in the VCS are an accurate representation of what's being applied to our infrastructure. If we use Con-

tinuous Integration to automatically apply our changes to test infrastructure every time we commit, then we'll be notified as soon as someone makes a change that breaks something. And if we have good habits of pulling the latest changes before we commit our own changes, then the second person to commit will discover - before they commit - that someone has changed something that may affect their own work.

Workflows for infrastructure teams are discussed in more detail in (to come).

Prefer definitions over scripts

General purpose scripting languages like Bash, Perl, Powershell, Ruby, and Python are an essential part of an infrastructure team's toolkit. But there are advantages to special purpose tools like Ansible, CFEngine, Chef, and Puppet. These tools are designed to simplify and standardize the problem of configuring many servers in a repeatable way. One of the main ways they do this is by separating the definitions of server configuration from the way they are executed and applied to servers.

These configuration definition formats are Domain-Specific Languages (DSLs)[3], as opposed to a general purpose programming language. They are special purpose languages whose constructions are tailored to their domain, in this case, server configuration.

Example 2-1. A configuration definition to define a user account

```
user "spock"
  state active
  gid   "science"
  home  "/home/spock"
  shell "/bin/sh"
```

The example above defines a user account. The configuration tool will read this definition and ensure that it exists on the relevant server, and has the specified attributes. There are a number of advantages over defining the account in definition file, rather than writing a script that explicitly creates the user account.

One advantage is clarity. The definition is easy to understand, since it only contains the key information for the user account, rather than embedding it within scripting logic. This makes it easier to understand and easier to find errors.

Another advantage is that the logic for applying each type of changes is separated and re-usable. Most configuration tools have an extensive library of definition types for things like managing user accounts, software packages, files and directories, and many other things. These are written to be comprehensive, so that out of the box they do the

3. A good reference on Domain Specific Languages, albeit written more for people thinking about implementing them than using them, is Domain Specific Languages, written by my colleagues Martin Fowler and Rebecca Parsons

right thing across different operating systems, distributions, and versions. They tend to have options for handling the various systems an infrastructure team may use, for example different authentication systems for users, network file system mounts, DNS, etc.

This pre-built logic saves infrastructure teams from the work of writing fairly standard logic themselves, and tends to be well-written and well-tested. Even if the off the shelf definition types available from a tool aren't enough, almost all of these tools allow teams to write their own custom definitions and implementations.

Declarative languages

Most DSLs for server configuration tools, as well as other infrastructure automation tools, are *declarative* languages rather than *procedural*.

Code written in a procedural language is a series of instructions that are executed step by step. Most scripting and programming languages are procedural, from Bash scripts and batch files to C++ and Java. Here is an example of procedural code:

```
if [ ! -d /var/www/repo ] ; then
  mkdir -p -m 0755 /var/www/repo
fi
cp /mnt/repository/index.html /var/www/repo
chown -R www:www /var/www/repo
chmod 0644 /var/www/repo/index.html
```

A declarative language is more like a configuration file. It defines the desired state for things, without specifying the steps to make it happen. Here is an example of declarative code:

```
directory '/var/www/repo' do
  mode '0755'
  owner 'www'
  group 'www'
end

file '/var/www/repo/index.html' do
  source '/mnt/repository/index.html'
  mode '0644'
  owner 'www'
  group 'www'
end
```

A declarative language defines what to do, while a procedural language defines how to do it. The advantage of a declarative language for configuration definitions is that it doesn't need to be concerned about the different things that might need to be done, or what may be optimized, depending on the state of the server before it's applied. If the definition has already been applied, the work doesn't need to be repeated, which can make it run more quickly.

Functional programing is a third paradigm, known from languages such as Lisp, Clojure, and F#, and influencing languages including Ruby. Functional code is structured similarly to mathematical functions, and treats data as immutable. Functional languages don't have much traction in infrastructure management as yet. But there are a few functional tools around, including Riemann, a monitoring message routing tool, and the Nix package manager.

API's and command line tools should be first class interfaces

Many infrastructure management tools are sold on the basis of a slick user interface. The idea is that dragging and dropping objects and selecting items from catalogs simplifies the job of infrastructure management. There's even the possibility that less technically experienced people can carry out infrastructure management tasks, and the tools will make sure everything is correct.

Giving infrastructure users the ability to provision and manage their own infrastructure is a powerful and valuable way to free up infrastructure team members to focus on work which needs their deeper skills. However, in order for self-service tooling to work effectively, it must be possible for technical people to get under the hood of the tools. The automobile industry is over 100 years old, but we still wouldn't buy a car with the hood welded shut.

For IT infrastructure, command line tools and programmable APIs are the means for infrastructure teams to get under the hood. This is necessary not only to tweak and fix things, but also to integrate different tools and services.

Command line interface (CLI) tools should be easy to script. That is, they should be designed to be run within an unattended shell or batch script, and take input and provide output that can be used with other command line tools and scripting functionality.

The common Unix command line tools are the exemplar of good CLI design. Rather than a single tool that does everything, they are a collection of small tools, each of which does one thing well. "grep" includes or excludes lines, "sed" alters the content of lines, "sort" sorts them, etc. These simple tools can be chained together in pipelines to carry out complex tasks.

A CLI tool should be designed to be run unattended. If it needs user input, it should have arguments that can be set in a script to provide this. A tool that insists on stopping for interactive input, such as accepting license terms, is a sign of a vendor that doesn't understand automation.

For tools which run as servers, a good API is essential. The guidelines for APIs for dynamic infrastructure platforms, discussed in [Link to Come], apply here.

Tools for different server management functions

Part II of this book goes into detailed patterns and practices for managing servers, based on the following lifecycle (see "A server's life" on page 67 for more details):

Server management lifecycle

- Package a server template

- Create a new server
- Update a server
- Replace a server
- Delete a server

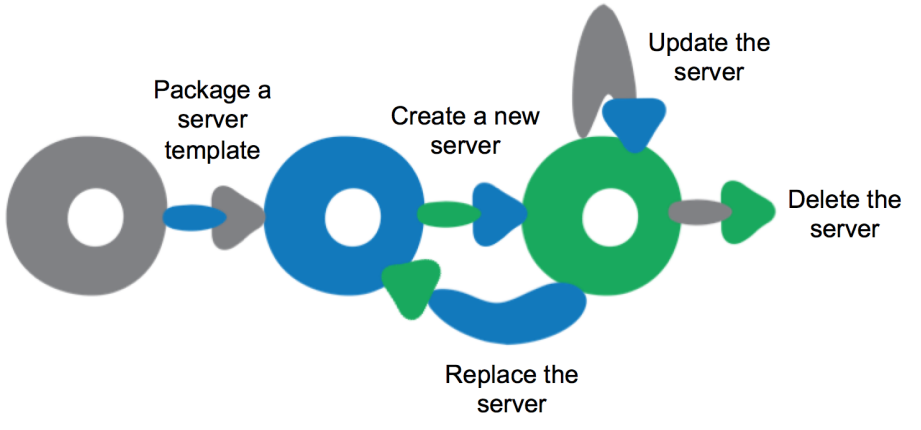


Figure 2-1. A server's lifecycle

This section will explore the tools involved in this lifecycle. There are several functions, some of which apply to more than one lifecycle phase. The functions discussed in this section are creating servers, configuring servers, packaging templates, and running commands on servers.

Tools for creating servers

A new server is created using the dynamic infrastructure platform, as described in (to come). This is normally done starting with a server template, which may be provided by the infrastructure platform, OS vendors, third parties, or custom-built in-house (as will be discussed shortly).

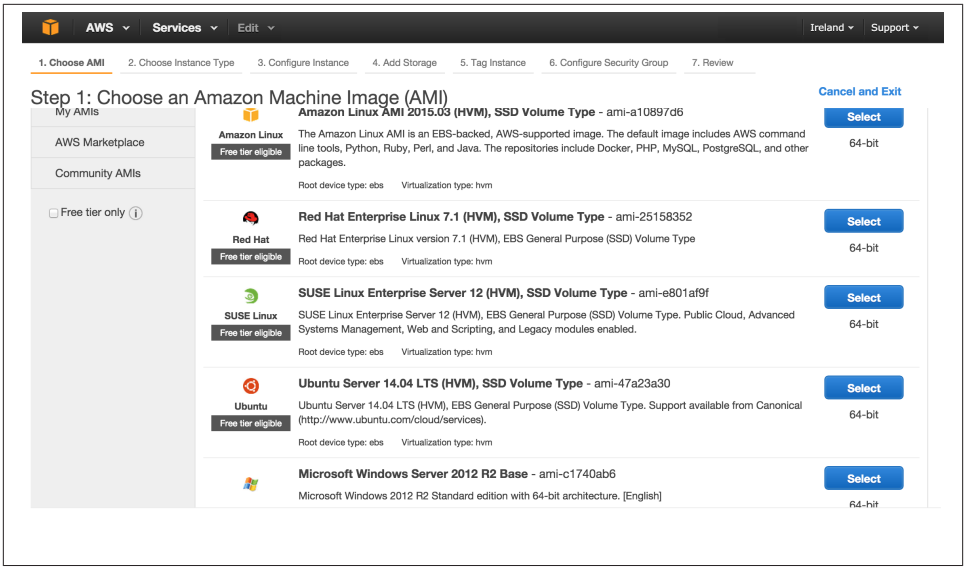


Figure 2-2. AWS web console for creating a new server

Most infrastructure platforms allow servers to be created interactively with a UI, as in the screenshot above. But of course, we'd prefer to have most servers created automatically. There are a few common situations in which new servers are created automatically:

- An infrastructure team member decides to create a standard type of server to address a need, for example replacing a failing DNS server.
- A user wants to create a new instance of a standard service, for example a bug tracking service. They trigger this using a self-service portal, which builds an application server configured to run the bug tracking software.
- A hardware issue causes a web server VM to crash, so the monitoring service triggers a new VM to be created to replace it.
- User traffic increases to a level where the current capacity of the application servers is not enough, so additional application servers are automatically created to take up the additional load.
- A developer creates a new build of application software. The CI software (e.g. Jenkins or GoCD) automatically creates an application server in a test environment, so it can run automated tests.

In some cases, the tooling involved may use the infrastructure platform's API to trigger the creation of new servers. The tool will be configured with the details needed, such as

the server template to use. In other cases, the team will write custom scripts, either using a CLI tool provided by the infrastructure platform, or else APIs.

An increasingly popular method for managing server creation is using an infrastructure definition tool, such as Terraform, CloudFormation, or Heat. These tools define a set of related infrastructure elements, for example the servers and networking for an application deployment cluster. The tools then use the infrastructure platform's API to ensure the correct servers have been created. See "Declarative infrastructure provisioning tools" on page 47 in Chapter 3 for more details on these tools.

Tools for configuring servers

Ansible, Chef, CFengine, Puppet, and Saltstack are examples of tools specifically designed for configuring servers with an infrastructure as code approach. They use externalized definition files with a DSL designed for server configuration.

Many server configuration tools use agents installed on each server. These run periodically, pulling the latest configuration definitions from a central repository and applying them to the server. This is the case for both Chef and Puppet. Other tools work with a push model, with a central server that triggers updates on servers. Ansible uses this model, using ssh keys to connect to server. This has the advantage of not requiring managed servers to have configuration agents installed on them, but arguably sacrifices security. Chapter 7 discusses these models in more detail.

Security tradeoffs with automated server configuration models

A centralized system that controls how all of your servers are configured creates a wonderful opportunity for evil-doers. Push-based configuration opens ports on your servers, which an attacker can potentially use to connect. An attacker might impersonation the configuration master, and feed the target server configuration definitions that will open the server up for malicious use. Or it might simply allow an attacker to execute arbitrary commands. Cryptographic keys are normally used to prevent this, but this requires robust key management.

A pull model simplifies security, as mentioned in the section describing that model. But of course there are still opportunities for evil. The attack vector in this case is wherever the client pulls its configuration definitions from. If an attacker can compromise the repository of definitions, then they can gain full control of the managed servers.

In any case, the VCS used to store scripts and definitions is part of your infrastructure's attack surface, and so must be part of your security strategy.

Security concerns with infrastructure as code are discussed in more detail in (to come).

Server configuration tools have toolchains beyond the basic server configuration. Most have repository servers to manage configuration definitions, for example Chef Server, Puppetmaster, and Ansible Tower. These may have additional functionality, providing configuration registries, CMDBs, and dashboards. Chapter 3 discusses infrastructure orchestration concerns. Arguably, choosing a vendor which provides an all-in-one ecosystem of tools simplifies things for an infrastructure team. However, it's useful if elements of the ecosystem can be swapped out for different tools, so the team can choose the best pieces that fit their needs.

Tools for packaging server templates

In many cases, new servers can be built using off the shelf server templates. Infrastructure platforms, such as IaaS clouds, often provide server templates for common operating systems. Many also offer libraries of templates built by vendors and third parties, who may provide servers that have been pre-installed and configured for particular purposes, such as application servers.

But it's common for an infrastructure team to build its own server templates. These can be pre-configured with the team's preferred tools, software, and configuration. Packaging common elements onto a template saves time when creating new servers. Some teams take this further, by creating server templates for particular roles such as web

servers and application servers. "Provisioning in the template vs. server creation" on page 74 discusses tradeoffs and patterns around baking server elements into templates versus adding them when creating servers.

One of the key tradeoffs is that, as more elements are managed by packaging them into server templates, the templates need to be updated more often. This then requires more sophisticated processes and tooling for building and managing templates.

Netflix pioneered approaches for building server templates with everything pre-packaged. They open sourced the tool they created for building AMI templates an AWS, Aminator[4]

Aminator is fairly specific to Netflix's needs, limited to building CentOS/Redhat servers for the AWS cloud (at least as of this writing). Hashicorp has released the open source Packer tool which supports a variety of operating systems as well as different cloud and virtualization platforms. Packer defines server templates using a file format that is designed following the principles of infrastructure as code.

Different patterns and practices for building server templates using these kinds of tools are covered in detail in Chapter 6.

Server change management models

Dynamic infrastructure and containerization are leading people to experiment with radical approaches to configuration management. There are several different models for managing changes to servers, some traditional, some new and controversial.

- **Ad-hoc** change management makes changes to servers only when a specific change is needed. This was the traditional approach before the automated server configuration tools became mainstream, and is still the most commonly used approach. It is vulnerable to configuration drift, snowflakes, and all of the evils described in Chapter 1.
- **Configuration synchronization** repeatedly applies configuration definitions to servers, for example by running puppet or chef agents on an hourly schedule. This ensures that any changes to parts of the system managed by these definitions are kept in line. Configuration synchronization is the mainstream approach for infrastructure as code, and most server configuration tools are designed with this approach in mind. Its main limitation is that many areas of a server are left unmanaged, leaving them vulnerable to configuration drift.
- **Immutable infrastructure** makes configuration changes by completely replacing servers. Changes are made by building new server templates, and then rebuilding relevant servers. This can increase predictability, since there is little variance be-

4. Netflix described their approach to using AMI templates in this blog post.

tween servers as tested, and servers in production. It requires sophistication in server template management.

- **Containerized services** packages applications and services in lightweight containers (as popularized by Docker). This reduces coupling between server configuration and the things that run on the servers. So host servers tend to be very simple, with a lower rate of change. One of the other change management models still needs to be applied to these hosts, but their implementation becomes much simpler and easier to maintain. Most effort and attention goes into packaging, testing, distributing, and orchestrating the services and applications, but this follows something similar to the immutable infrastructure model, which again is simpler than managing the configuration of full-blow virtual machines and servers.

These models are discussed in more detail in Part II of this book, particularly in Chapter 7.

Tools for running commands on servers

Tools for running commands remotely across multiple machines can be helpful to teams managing many servers. Remote command execution tools such as mcollective, fabric, and capistrano can be used for ad-hoc tasks like investigating and fixing problems, or they can be scripted to automate routine activities.

Some people refer to this kind of tool as "ssh-in-a-loop". Many of them do use ssh to connect to target machines, so this isn't completely inaccurate. But they typically have more advanced features as well, to make it easier to script them, define groupings of servers to run commands on, or to integrate with other tools.

Although it is useful to be able to run ad-hoc commands interactively across servers, this should only be done for exceptional situations. Manually running a remote command tool to make changes to servers isn't reproducible, so isn't a good practice for infrastructure as code.

If someone finds themselves using tools interactively for a particular type of task multiple times, they should consider how to automate it. The ideal is to put it into a configuration definition if appropriate. Tasks which don't make sense to run unattended can be scripted in the language offered by the team's preferred remote command tool.

The danger of using scripting languages with these tools is that over time they can grow into a complicated mess. Their scripting languages are designed for fairly small scripts, and lack features to help manage larger codebases in a clean way, such as re-usable, shareable modules. Server configuration tools are designed to support larger codebases, so tend to be more appropriate.

General scripting languages

The usefulness of specialized tools for server configuration doesn't mean there's no role for general purpose scripting languages. Every infrastructure team I've known needs to write custom scripts. There are always ad-hoc tasks which need a bit of logic, like a script that scrapes information from different servers to find out which ones need a particular patch. There are little tools and utilities to make life easier, as well as plugins and extensions to standard tools.

It's important for an infrastructure team to build up and continuously improve their skills with scripting. Learn new languages, learn better techniques, learn new libraries and frameworks. Not only does this help you make tools for yourselves, it also enables you to dig into the code of open source tools to understand how they work, fix bugs, and make improvements you can contribute to improve the tools for everyone.

Many teams tend to focus on a particular language, even standardizing on one for building tools. There's a balance to make with this. On the one hand, it's good to build up deep expertise in a language, and to ensure that everyone on the team can understand, maintain, and improve in-house tools and scripts.

But on the other hand, broadening your expertise means you have more options to apply to a given problem - one language may be better at working with text and data, another may have a more powerful library for working with your cloud vendor API. A more polyglot[5] team is able to work more deeply with a wider variety of tools.

I prefer to have a balance, having one or two "go-to" languages, but being open to experiment with new ones[6]. The pitfall to avoid is having many tools in scripts written in languages that nobody really understands. When these underpin core services in the infrastructure, they become snowflake scripts that everyone is afraid to touch, in which case it's time to rewrite them in one of the team's core languages.

It should go without saying by now that all of the scripts, tools, utilities, and libraries the team develops should go somewhere into VCS. As time goes on, the team may need to come up with a structure to organize these scripts to make it easy to find and maintain them.

5. Neal Ford coined the term polyglot programming. See this interview with Neal for more about it.

6. I like they way John Allspaw of Etsy put it in an interview, "We want to prefer a small number of well-known tools". Although Allspaw is specifically talking about databases at that point of the interview, he is describing the Etsy team's approach to diversity versus standardization of tools and technologies.

Conclusion

The intention of this chapter was to understand several different high level models for managing individual servers, and how these models relate to the types of tooling available. Hopefully it will help you consider how your team could go about provisioning and configuring servers. However, before selecting specific tools, it would be a good idea to be familiar with the patterns in Part II of this book. Those chapters provide more detail on specific patterns and practices for provisioning servers, building server templates, and updating running servers.

The next chapter will look at the bigger picture of the infrastructure, exploring the types of tools that are needed to run the infrastructure as a whole.

CHAPTER 3
Infrastructure orchestration services

In the previous two chapters, we have discussed the capabilities that we need to automatically manage the basic elements of an infrastructure ((to come)), and those we need to provision and configure individual servers (Chapter 2). This chapter focuses on the capabilities we need in order to run the infrastructure as a whole. This includes managing the infrastructure itself, coordinating the different elements of the infrastructure, and providing core services consumed by applications and services running on the infrastructure.

Examples of infrastructure orchestration services include monitoring, service discovery, centralized configuration management, and distributed process management.

Clearly, well-run infrastructure services are important for a well-run infrastructure. Building and managing these services using the principles and practices of infrastructure as code gives the usual benefits:

- Anything can be rebuilt quickly
- Routine requests are fulfilled quickly, with little effort
- Complex changes can be made easily and safely
- Everything is kept consistent and up to date
- The team is able to spend its time and attention on high value work
- Infrastructure services are provided with a self-service model

Infrastructure orchestration services are the foundation of your infrastructure. So how well your infrastructure operations delivers these benefits depends on how you design and implement these services.

Criteria for infrastructure services

In order to achieve these outcomes, we need to design and build our infrastructure services following the principles of infrastructure as code, which include:

- Reproducibility
- Consistency
- Disposability
- Continuous service
- Self-testing
- Self-documenting
- Small changes
- Versioned
- Self-service

The previous chapters have discussed guidelines for selecting tools that fall into line with these principles, in the context of virtualization platforms and server configuration tools. Many of these apply to infrastructure orchestration services as well.

Externalized configuration

As with server configuration tools, any tool used to build infrastructure services should preferably have configuration which can be managed externally from the tool itself. This allows us to store it in a VCS and use standard tools, scripts, and practices from source code management and text file processing. We can organize our configuration so that it can be modularized and shared, we can trigger actions like testing, and we can promote it between environments for safe change management.

Closed-box tools which hide their configuration behind a GUI and possibly an API risk becoming snowflake systems. A fragile snowflake system doesn't make a good foundation for an anti-fragile infrastructure.

The previous chapter discusses this topic extensively in "Prefer definitions over scripts" on page 31. As discussed there, it may be possible to hack an infrastructure as code model around a closed-box tool if the configuration can be exported and imported between instances, but this is more difficult to maintain.

Coping with dynamic infrastructure

An area where many older infrastructure management products struggle is coping gracefully with a dynamic infrastructure. They were designed in the iron age of infrastructure, when the set of servers was generally static. In the cloud era, servers are

continuously being added and removed, often by unattended processes. A tool that needs someone to point and click every change into a GUI interface is a serious bottleneck.

Here are some useful criteria for considering tooling for dynamic infrastructure:

- Ability to gracefully handle infrastructure elements being added and removed, including entire environments
- Support aggregating and viewing historical data across devices, including those which have been removed or replaced by different devices
- Ability to make changes automatically in response to events, without human intervention
- Licensing that is flexible enough to handle devices constantly being added and removed, and that makes financial sense

Many infrastructure services, such as monitoring, need to know the state of servers or other elements of the infrastructure. It should be possible to add, update, and remove infrastructure elements from these services automatically, for example through a RESTful API.

Licensing concerns

Licensing can make dynamic infrastructure difficult with some products. Some examples of licensing approaches that work poorly include:

- A manual process to add a new device to the license
- Pricing with a long minimum commitment (e.g. needing to pay for a full month to add a server that will only be used for a few hours)
- Heavyweight purchasing process to increase capacity

Ideally, licensing wouldn't be fixed on the number of devices involved. Where it is, it is preferable to have dynamic usage licensing at a small granularity, for example one hour. Discounts for longer commitments can help manage budgets. For example, paying a fixed 1-year contract for a certain base level of usage, with additional charges per hour-use needed over that.

Pitfall - tight coupling of infrastructure

One pitfall to watch out for with infrastructure orchestration services is tight coupling. It is important to ensure that a change can be made to any part of the infrastructure without requiring widespread changes to other parts. Coupling between parts of the system increases the scope and risk of changes, which leads to making changes less often, and with more fear.

It takes vigilance and good design to prevent tight coupling from becoming an issue. Pay attention to the signs that parts of your infrastructure are becoming a bottleneck for changes, and aggressively redesign them to reduce friction.

Typical infrastructure orchestration services

The rest of this chapter will discuss key infrastructure orchestration services.

Infrastructure provisioning

Infrastructure provisioning tools are used to create, modify, and destroy elements of an infrastructure. Some provisioning tools are transactional, while others are declarative.

A transactional tool carries out commands on elements in the infrastructure. Create a server, change the minimum size of a server pool, destroy a load balancer. A declarative tool changes the state of an infrastructure to match a definition. There should be three servers, serverA, serverB, and serverC, behind LoadBalancerX.

Transactional infrastructure provisioning tools

Transactional infrastructure provisioning tools aren't concerned with the state of the infrastructure, either the current or desired state. They might have commands to report back on the current state - list all of the instances in my account with the tag "Role=appserver". But they normally rely on the infrastructure platform to provide this information.

Most infrastructure management platforms provide command line tools that can be used to manage infrastructure in a transactional way, like the AWS CLI and Rackspace's rumm. These typically wrap the platform's API. If you want to standardize a process that requires several actions, you can write a shell script.

On one team I worked with we wrote a command-line tool to standardize the way we provisioned server instances. We defined a set of roles for servers, like "webserver", "appserver", and "dbnode". We also had a set of environments to provision servers into. The characteristics of the server roles were defined in one configuration file roles.yml,

and settings for the environments went into another, environments.yml, which we checked into our VCS.

With this tool, we could create a server by specifying only the role and environment:

Building a new server with a custom provisioning tool.

```
# spin new appserver qa
```

This met our requirement for a repeatable, transparent process. The decisions of how to create the server - how much RAM to allocate, what OS to install, what subnet to assign it to - were all captured in a transparent way, rather than needing us to remember them. These commands were simple enough to use in scripts and tools that trigger actions, so it they worked well as part of automated processes.

If we needed to change something, like adding more RAM to our database nodes, we would edit the configuration files or scripts, commit them, and then apply them to each environment in our pipeline.

More details on implementing this kind of script can be found in "Pattern: Wrapping server creation options into a script" on page 80.

Declarative infrastructure provisioning tools

The way declarative infrastructure provisioning tools work is similar to what was described for server configuration tools in Chapter 2. Someone writes a configuration definition that describe how the infrastructure should look, using a DSL. When the tool is run with the definition, it makes whatever changes are needed to bring the infrastructure in line with the definition.

As with server configuration tools, this process is idempotent, so running the tool multiples times using the same definition should keep things in the same state.

The example below is part of a Terraform configuration that makes sure a single server exists, and belongs to a security group that allows ssh access from any address.

Example Terraform configuration file.

```
resource "aws_security_group" "default" {
    name = "terraform_example"

    ingress {
        from_port = 22
        to_port = 22
        protocol = "tcp"
        cidr_blocks = ["0.0.0.0/0"]
    }
}

resource "aws_instance" "base" {
  instance_type = "t2.micro"
```

```
  ami = "${var.ami_id}"
  security_groups = ["${aws_security_group.default.name}"]
}
```

Declarative infrastructure provisioning tools need a reliable understanding of the current state of infrastructure, as it relates to the managed infrastructure.

Examples of this kind of tool include AWS CloudFormation, OpenStack's Heat, and Hashicorp's Terraform.

Service discovery

Elements of an infrastructure need to be able to discover one another. A reverse proxy needs to know which web server to send traffic to. An application needs to know how to connect to its database. A backup server needs to know which servers and drives to back up.

A dynamic infrastructure needs discovery mechanisms that are automatically updated as servers and devices are added and removed, and as services are shifted between servers. These updates should happen without human intervention, as needed. For example, if a web server is added to a load balancing pool to cope with increased traffic, the load balancer configuration should be updated without anyone needing to think about it.

A few popular discovery mechanisms are listed in the table below:

Table 3-1. Common service discovery mechanisms

Mechanism	How	Comments
Fixed IP addresses	Decide on fixed IP addresses for services, for example the monitoring server is `192.168.1.5`.	Doesn't work with hosting platforms that assign addresses automatically. Complicates replacement of a server, especially with zero downtime replacement strategies (see (to come)). Doesn't work with resource pools that can vary in size, such as a pool of web servers. Difficult to change, not very flexible. Not recommended.
hostfile entries	Use automated configuration to ensure servers have /etc/hosts files (or equivalent) which map resource server names to their current IP addresses.	Ensuring hostfiles are updated automatically as services change is a complicated way to solve a problem that DNS solved long ago.

Mechanism	How	Comments
DNS	Use DDNS (Dynamic DNS, Domain Name System) servers to map service names to their current IP address.	On this plus side, it's a mature, well-supported solution to the problem. However, some organizations don't enable dynamic updates (the first "D" in "DDNS"), which is needed for a dynamically changing infrastructure. While DNS can support pools of resources, it doesn't give the level of control needed for advanced zero downtime replacement strategies. Doesn't support tagging or annotations that can be used to dynamically build configuration, for example to update the configuration for a load balancer with the current set of active web servers in a pool.
Configuration registry	Centralized registry for data about infrastructure elements and services (see "Configuration registry" on page 49)	Not directly useful for network level routing (e.g. what you would use DNS for), but it works well for dynamically generating configuration and generally supplying more detailed information about resources in the infrastructure.

In practice, network level name lookup (i.e. DNS) is essential for most infrastructures. Many teams will find the need to add a configuration registry to support more sophisticated integration of services, resources, and applications, so both of these would live side by side.

Configuration registry

A configuration registry is a directory of information about servers and other infrastructure elements. It provides a place where scripts, configuration definitions, applications, and services can find out about other elements of the infrastructure.

As an example, when a new server is provisioned we can put information in the registry indicating what applications are running on it. Our monitoring system can use this information to add the relevant monitoring checks for those applications. When the server is destroyed, the registry entry is updated to show the server has been retired, and the monitoring system's configuration is then updated to remove those checks.

Using a registry can keep the various scripts and services which need to know about and update details of the infrastructure loosely coupled. The registry acts as a Single Source of Truth" (SSOT).

There are may configuration registry products and tools available, including Zookeeper, Consul, and Etcd. Many configuration management tool vendors provide their own configuration registry, for example Chef Server, PuppetDB, and Ansible Tower. These products are designed to integrate easily with the configuration tool itself, and often with other elements such as a dashboard.

The important characteristic a configuration registry needs to be useful with a dynamic infrastructure is the ability to programmatically add, update, and remove entries from the registry.

Using configuration from a central registry

A configuration registry can be used to provide input to configuration definitions. A definition file may be written to use variables which can be replaced for a given server, depending on the server's purpose, environment (QA, Staging, Production, etc.), or other factors.

For example, a team running VMs in several data centers may want to configure monitoring agent software on each VM to connect to a monitoring server running in the same data center. The team is running Chef, so they add these attributes to the chef server:

```
default['monitoring']['servers']['sydney'] = '10.0.4.20'
default['monitoring']['servers']['dublin'] = '10.2.4.20'
```

When a given VM is created, it is given a registry field called *data_center*, which is set to *dublin* or *sydney*.

When the chef-client runs on a VM, it runs the following recipe to configure the monitoring agent:

```
my_datacenter = node['data_center']
template '/etc/monitoring/agent.conf' do
  owner 'root'
  group 'root'
  mode 0644
  variables(
    :server_ip => node['monitoring']['servers'][my_datacenter]
  )
end
```

The chef recipe retrieves values from the chef server configuration registry with the `node['attribute_name']` syntax. In this case, after putting the name of the data center into the variable `my_datacenter`, that variable is then used to retrieve the monitoring server's IP address for that data center. This address is then passed to the template (not shown here) used to create the monitoring agent configuration file.

A heavily-used registry has pitfalls. It may not always be clear which scripts and systems depend on a particular registry entry type or format, so when the tool that owns it decides to change, remove, or restructure it, things may break unexpectedly.

But when I applied the change, I found that someone else had written a configuration script to build monitoring checks based on the original structure. We didn't discover this for several days, since the monitoring configuration script didn't complain, it just stopped updating the monitoring check when servers changed.

This problem can be mitigated to some extend with good design and communication, but also through automated testing. Some variation of Consumer Driven Contract testing, as described in (to come) could have helped here. People who write scripts that make use of the registry can drop simple tests to be run that raise an alert when registry structures and formats they rely on have changed.

Sharing configuration information between nodes

Configuration management software can not only find configuration settings from a central registry, it can also add and update this information so it can be used for other parts of the infrastructure. This can be especially useful for handling dynamic infrastructure.

The example above for configuring monitoring agents used fixed IP addresses for the monitoring servers. But if the monitoring servers themselves are managed following the principles of infrastructure as code, then servers may be removed and added at any time. When this happens, the IP address needs to be changed in the agent configuration for all of the VMs using that server.

One way of handling this would be for the chef recipe that configures a monitoring server to update the chef-server registry with the new IP address for the server. The next time the monitoring agent recipe is run on each VM, it will update the configuration file with the new IP address found in the registry.

This example explains how a the concept of using a shared configuration registry in a dynamic infrastructure, but it isn't actually a very good solution for this particular problem. This solution requires all of the agents to be reconfigured when the server is replaced, which is not ideal. A better solution would be to have a fixed IP address for each monitoring server, for example using a load balancer VIP, and reassign that IP address to the new monitoring server. This way, the IP address never needs to be changed on the agents.

Lightweight configuration registry approaches

Some teams find that they can get the benefits of a centralized configuration registry without the complexity of a running a registry service. A registry service comes with a cost, in terms of having to set up and maintain server processes which are effectively a single point of failure for the entire infrastructure. The registry may need to scale to support thousands of nodes. It needs to be reliable and available, possibly across data centers and regions. This in turn may require keeping data synchronized and consistent across multiple nodes.

While these are solvable problems, alternative approaches may simplify things and make the system more reliable. A popular approach is to store configuration values in a reliable file store, for example in Amazon S3 buckets. Values can be stored in a simple file format, such as yaml or json files, which can be retrieved and updated by local server configuration tools like puppet apply or chef-client, or by wrapper scripts that run the configuration tool after pulling down the relevant files.

A VCS can be used along with this kind of file-store based approach for versioning and history. Some organizations use a VCS directly as a shared configuration registry.

Another variation of this is packaging configuration settings into system packages, such as a .deb or .rpm file, and pushing them to an internal apt or yum repository. The settings can then be pulled to local servers using the normal package management tools.

The choice of approach is often down to the priorities and preferences of the team managing the infrastructure.

Is a configuration registry a CMDB?

A CMDB (Configuration Management Database) is a concept pre-dating the rise of automated, dynamic infrastructure. A CMDB is a database of IT assets, referred to as configuration items (CI), and relationships between those assets. It is many ways similar to a configuration registry - they're both databases listing the stuff in an infrastructure.

But CMDB's and configuration registries are used to address two different core problems. Where they can be used to handle overlapping concerns, they do it in very different ways. For these reasons, it's worth discussing them as separate concerns.

A configuration registry is designed for automation tools to share data about things in the infrastructure, so they can dynamically modify their configuration based on the current state of things. So it needs a programmable API, and the infrastructure team should use it in a way that guarantees it is always an accurate representation of the state of the infrastructure.

CMDB's were originally created to track IT assets. What hardware, devices, and software licenses do we own, where is it now, and what are we using it for? In my distant youth, I built a CMDB with a spreadsheet, and later moved it into a Microsoft Access database.

We managed the data by hand, and this was fine because it was the Iron Age, when everything was hardware, and things didn't change very often.

So these are the two different directions that configuration registries and CMDB's come from - sharing data to support automation, and recording information about assets.

But in practice, CMDB's products, especially those sold by commercial vendors, do more than just track assets. They also discover and track details about the software and configuration of things in an infrastructure, to make sure that everything is consistent and up to date. They even use automation to do this.

An advanced CMDB can continuously scan your network, discover new and unknown devices, and automatically add them to its database. It can log into servers, or have agents installed on them, so it can inventory everything on every server. It will flag issues like software that needs to be patched, user accounts that should not be installed, and out of date configuration files.

So CMDB's aim to address the same concerns as infrastructure as code, and they even use automation to do it. However, their approach is fundamentally different.

The CMDB audit and fix anti-pattern

The CMDB approach to ensuring infrastructure is consistent and compliant is reactive. It assumes infrastructure elements may be provisioned incorrectly, and then reports on what needs to be corrected. This makes sense for teams who manage infrastructure with manually-driven processes. It assumes that servers will be routinely provisioned in inconsistent ways, or changes made that make them inconsistent, and establishes a process to identify the inconsistencies.

The problem is that resolving these inconsistencies adds a constant stream of work for the team. Every time a server is provisioned or changed, new work items are added to fix them. This is obviously wasteful and tedious.

The infrastructure as code approach to CMDB

The alternative, the infrastructure as code way, is to ensure that all servers are provisioned consistently to start with. Team members should not be making ad-hoc decisions when they provision a new server, or make a change to one. Everything should be driven through the automation. If a server needs to be built differently, then the automation should be changed to capture the difference.

This doesn't address 100% of the concerns addressed by a CMDB, although it does simplify what a CMDB needs to do. Here are some guidelines for handling CMDB concerns when managing an infrastructure as code:

- If you need to track assets, consider using a separate database for this, and have your automation update it. This should be kept very simple.
- Make sure everything is built by automation, and so is recorded correctly and accurately.
- Your automation should accurately record and report your use of commercial software licenses in the configuration registroy. A process should report on license usage, and alert when you are out of compliance, or when you have too many unused licenses.
- Use scanning to find and report on things not built and configured by your automation.

For example, you can have scripts that use your infrastructure provider API to list all resources (all servers, storage, network configurations, etc.), and compare this with your configuration registry. This can catch errors in your configuration (e.g. things not being added correctly to the registry), as well as things being done outside the proper channels.

Monitoring - Alerting, metrics, and logging

Monitoring is a broad topic, and a label often applied to different things. It might mean alerts and notifications that get people out of bed at night to fix problems, or dashboards with metrics and graphs. One person says "monitoring" with system level resources like CPU utilization in mind, another is talking about business-focused KPI's like conversion rates.

Monitoring is about making information visible. This may be information about applications, services, infrastructure, or users. It may be about state, or it can be about events. The information may be made available for interactive searching and analysis, presented on a dashboard, or pushed to someone's phone.

One way to look at monitoring as an infrastructure capability is as events. Events come from many sources, including applications, server operating systems, devices, monitoring agents, and various services and tools throughout an infrastructure. They may also come from third party services, such as a hosted analytics or monitoring service.

> My esteemed colleague Peter Gillard Moss prefers to think about monitoring rather than logging. That is, design monitoring around state rather than streams of events. I don't have any real disagreement with this, and there are some interesting insights and

ideas that come out of a state-oriented view of monitoring. But I've just found it easier to discuss monitoring architecture as a flow of events.

Events may be pushed by the source system or application over the network, written to log files, or pulled by agents using an API such as SNMP, JMX, WMI, or a custom protocol such as a custom JSON format provided by a REST endpoint.

An event might be a measurement of a system resources such as RAM, an infrastructure change like a new server being created, or a user transaction within a service.

There are different possible destinations for an event, including things like triggering alerts or being added to a database for analysis or reporting. Many events may be used for multiple purposes in the end.

The best way to approach the design of an monitoring system is to think about how it will be used. As with any software system, you can identify different users and their use cases and design your system to meet their various needs. Some examples of typical users are systems administrators, developers, and product managers. Typical use cases for each of these users are:

- Be notified when there is a problem that needs fixing.
- View key metrics.
- Search and analyze events to diagnose the cause of a problem.

The specific data that each user is interested will tend to vary. Product managers will want to know about user activity, transactions, and other business and user related events. Systems administrators need to know what's happening at the infrastructure level. Developers will want to know about the applications they write and support.

In practice, in a highly functioning organization most users will be interested in at least some data not directly within their scope. Developers and systems administrators should each fully understand what's going on in the others' world, since they are closely related. Both should care about what's happening at the business and commercial level, since it indicates how well their systems and applications are doing their job.

Alerting - tell me when something is wrong

Monitoring is a cornerstone of a reliable infrastructure. As mentioned in Chapter 1, the secret ingredient of an anti-fragile, self-healing infrastructure is people, and monitoring lets people know when there is a problem that needs fixing.

In an infrastructure where the servers and other elements underpinning a service come and go, and where they routinely shrink and grow, the monitoring system needs to be

designed to focus on what's important. End-user service is what really matters, so checking that everything is working correctly is essential.

Active checks may do things like log into a service and carry out key transactions or user journeys. They prove that the end to end systems are working correctly and giving the correct results to users.

Deductive monitoring tracks activity and metrics, alerting when behavior strays outside normal boundaries. If transaction numbers drop, it may indicate a variety of problems, including failures of the system, or even problems with critical externally managed services like DNS or the CDN. Unusual spikes are also worth noticing, since they may be a sign of an attack such as a DDOS attack, or a bug that causes an application to go haywire.

More thought is needed to properly monitor the elements underlying the service - it's important to ensure that alerts are relevant. An all-too-common situation is a monitoring system that continually spews a stream trivial alerts. The team learns to ignore alerts, which guarantees that any alert of a real problem will be ignored as well, defeating the purpose of the monitoring system.

The team should create categories for potential alerts, based on the actions that should be taken when they occur. Some alerts, especially those which indicate the service is no longer working for users, or is likely to stop working soon, should get someone out of bed to fix the issue immediately. These alerts should be very few, and very easy to recognize.

Some alerts may need human attention, but can wait until working hours. There are other conditions which, although they might indicate a problem, really only need to be looked at if they become unusually frequent. For these, consider recording the condition as a metric, making sure the metric is visible (for example on a dashboard or information radiator), and setting alerts when the metric goes outside of expected bounds. For example, it may be fine that the web server pool grows and shrinks throughout the day, but if it fluctuates wildly there may be a problem, or at least a need to tune it better.

What is an information radiator?

An information radiator, or a communal dashboard, is a highly visible display of information put up in a team space so everyone can easily see key status information. Agile software development teams typically use these to show the status of the build in the CI and CD pipeline. Ops teams tend to show the status of services. Cross-functional product teams tend to show both of these, plus indicators of key business metrics.

Information radiators will be ignored unless they show actionable information in a concise way. A useful rule of thumb is that the radiator should make it immediately obvious which of two courses of action to take: 1) carry on with routine work, or 2)

shout, stop what you're doing, and take some action to make the radiator show that you can carry on with routine work. Loads of tiny line graphs showing memory usage and CPU utilization don't do this. A block that goes red when the customer checkout process stops working does.

Dynamic infrastructures give some good examples of events which may or may not need alerts. Servers come and go, added and removed automatically based on demand or for routine updates. It's not always easy or even possible to configure tools that trigger changes to infrastructure to properly notify monitoring systems about changes, which means the monitoring system can't tell the difference between a server that disappears because of a failure and a server that has been automatically removed because it's no longer needed.

And even the failure of servers or other infrastructure elements may not be worth getting someone out of bed, if the systems deliver uninterrupted service to end users and the infrastructure recovers on its own. Ideally it should be brought to someone's attention the next day, so they can check things over and make sure there are no underlying problems, and that there really were no user service issues such as lost transactions.

Even when an event doesn't warrant an alert, it is useful to record metrics and make sure unusual fluctuations are made visible to the team. Organizations such as Netflix with cloud-based estates numbering in hundreds of servers find that a certain percentage of instances fail during the course of a day. Their infrastructure replaces them without any drama, but having a chart that shows the failures makes the team aware if the number becomes unusually high. It sometimes happens that some part of a cloud provider's infrastructure develops a problem, in which case the team may want to manually shift resources around to cope better.

Metrics - Collect and analyze data

Another side of monitoring is collecting data so it can be analyzed, and so interesting trends can be made visible on dashboards or information radiators. This is another area where dynamic infrastructure makes things tricky for older tools. It can be difficult to see trends of memory usage for an application server if the server has been replaced four times in the past week. The monitoring data for those five servers may not be easy to link together and show as a single effective server.

A good monitoring solution will use tagging or similar functionality that allows you to build graphs (and alerts) based on dynamic aggregates of resources. For example, you should be able to have a graph of CPU utilization across a fluctuating pool of application servers. Plotting this against a count of the number of servers should show how auto-scaling helped to manage load.

I've run into some monitoring tools which don't cope properly with servers that have been removed. They insist on treating the server as **critical**, cluttering dashboards with red status indicators, unless the server is removed from the monitoring system, in which case the data is no longer available for looking at historical trends. If you run into a monitoring tool that does this, insist the vendor update it for the modern era, or replace it.

Log aggregation and analysis

IT infrastructures are awash with log files. Systems and their core service and utilities generate logs, applications write their own log files, even network devices can log events to syslog servers. One of the side effects of server sprawl is that it can be hard to even know where to look to dig into problems, and dynamically managed servers make the problem even worse.

The common solution to this is centralized log aggregation, sending logs from individual servers and devices to a service that stores them all. Tools can be added to make it easy to search the aggregated logs, build dashboards based on the activity in them, and raise alerts when bad things appear in them.

While it's no doubt useful to make all of this data available for investigation, it's probably wise to think about those events which need to be used for alerting and for making key metrics visible, and consider a more structured method for managing them. Log files have a tendency to be a messy combination of message strings that are (maybe) meaningful to humans, but difficult for machines to reliably parse, and dumps produced by machines that aren't necessarily meaningful to humans, and probably also not very easy for machines to parse (stack traces spring to mind). Again, these are useful to muck around in when troubleshooting an issue, but they're less useful if you want to reliably detect and report problems or status. Applications and systems should be written and configured to generate more structured messages for automated consumption.

Routing events

So there are various sources of events, and various things they're used for. A single event could be used in multiple ways; added to aggregated logs for reporting and analysis, appended to a metrics database for trending, and triggering an alert if it's out of bounds. The monitoring system may not even be a one-way, directed graph - many metrics databases can be configured to create alerts based on the value of a metric, aggregated metrics, or even patterns. For example, the system could be configured to send an alert if the system records CPU utilization exceeding 80% at least three times in a row within a period of two minutes.

There are tools which aim to handle routing of events, offering a rules-based system for deciding when an event should go to a metrics database, alerting system, other destinations, or simply be dropped. An example of this is Riemann, which has a sophisticated language for filtering, aggregating, and routing events.

Storage orchestration

(to come) discussed the storage services provided by an infrastructure management platform ((to come)). As mentioned there, these are typically provided as block storage devices, which can be mounted on server instances as if they are local disk drives, and block storage services, which are accessed using network protocols.

Infrastructure teams often find the need to share storage more directly between server instances. One way to do this is using file sharing network protocols like NFS, SMB/CIFS. A server instance can mount local or block storage and make it available for other servers to mount and use. These file server technologies have long been used in pre-virtualized environments, but care should be taken before jumping into implementing them on a virtualized or cloud-based infrastructure.

Sharing a file system over a network when the "local" file system is itself mounted over the network may be adding unnecessary overhead and complications. Using a traditional network file system on cloud servers also creates extra administrative complexity and overhead. These file server technologies don't necessarily cope well with file server nodes that are routinely added and removed, which makes continuity challenging.

Distributed and/or clustered file service tools such as GlusterFS, HDFS, or Ceph are designed in a way that can make them more suitable for use in a dynamic infrastructure. They ensure data availability across multiple server nodes, and can usually handle nodes being added and removed more gracefully. However, don't make assumptions about how well this will work, be sure to test not only for performance and latency, but also for the impact of changes to the cluster and for failure scenarios.

Before leaping into the specific technologies and tools to use, think through the use cases. I've seen at least one team implement a complex and fragile glusterfs cluster simply to achieve fault tolerance - to make sure the data from a server would be available for a failover server to pick up if the first server fails. It is generally simpler and more reliable to make direct use of services built into the platform, such as block storage replication.

Distributed process management

Part of managing services is deciding where processes should run. An infrastructure may have dozens or hundreds of servers available to run a particular process on. A dynamic infrastructure can create even more servers if needed. There are a few different

models for assigning processes to servers, which may vary depending on whether the process is a long running process or a job.

A long running service process is a daemon in Unix/Linux terminology, or a Windows Service in the Windows world. It runs perpetually, even when it's idly waiting for work to do. An example is a web server, which has one or more processes running, listening on a network port for requests to handle.

A job has a finite lifespan. Something triggers it, maybe on a schedule like a cron job, or else fired up by some service process. The job carries out its work and then terminates, perhaps having made the output of its work available for other jobs or processes to pick up. An example is a CI build job that compiles and tests application source code.

Managing distributed server processes

The classic way to assign a service process to a server is to make it part of the server's role. The definition of the server's role includes the service processes that should run on it. This is a simple model. If you need more web server processes, you build more web servers. If you need to remove some, you destroy the servers. This model does require your servers to be specifically to run a certain set of processes.

Containerization offers a different model for managing server processes (see (to come)). Processes are packaged so that they can be run on servers which haven't been specifically built for the purpose. So you have a set of generic container host servers, and you can deploy and run a containerized service on any one of them. This makes assigning processes to servers more flexible, and simplifies how you manage servers. However, it adds complexity to how you start, track, and manage processes.

So it makes sense to use containers to decouple services from servers when you need more fluidity. This comes in handy if you need to add and remove services frequently, or if you need to balance the use of server resources more finely. Deploying and removing a container instance is much faster than creating or destroying a server instance. And the cost of running a full server instance - even a small one - for each service process can become unwieldy. Sharing a smaller number of larger server instances between processes is usually more economical, and containers can make it much easier to do.

Managing distributed jobs

As mentioned above, a distributed job is a process that runs for a finite period of time across one or more servers. Many services, especially infrastructure services, manage special purpose jobs with their own systems of servers and agents, for example monitoring, server configuration updates, and CI build and deployment jobs. However, it can be useful to have more general purpose job management tools and systems.

I've seen teams use CI tools like Jenkins and TeamCity to manage general purpose infrastructure management tasks. It's convenient and easy to configure jobs in these

tools which run on a schedule or in response to events, and assign them to agents running on various parts of the infrastructure.

Remote command execution tools, such as mcollective, capistrano, fabric, and others can run commands across multiple servers. These are typically used for ad-hoc tasks, such as investigating and fixing problems, but can also be scripted and scheduled. These tools are discussed in a bit more detail in "Tools for running commands on servers" on page 40.

Container orchestration

As described earlier, containerization can be used to decouple processes from the servers they run on. This creates flexibility in assigning processes to servers, but that flexibility adds complexity. When starting a new containerized process, someone needs to decide where it can and should be run. We need to be able to find out where a given process is currently running. We may need to establish networking connections between containers, assign storage, and identify when a process has failed or doesn't have enough resources.

The complexity of managing large numbers of containerized processes needs are driving the rise of container orchestration tooling. As of early 2015 the landscape of available tools is evolving rapidly. Different tools take different approaches, and focus on different aspects of container management. Examples include Fleet, Docker Swarm, Kubernetes, and Mesos.

Software deployment

Most of the software installed on servers managed through infrastructure as code is put there through the definitions used in the provisioning process. However, it's common for software developed in-house to have more complex deployment requirements. They may need to install components on different servers, make changes to database schemas, and alter web server configuration, all while taking components down and back up in a specific order.

In the worst cases, these complicated software deployment processes are done manually, and are difficult to automate. There are a few reasons why this happens:

- The software has been manually deployed from its start. As a system grows over time without the use of automation, retrofitting automation is extremely difficult without significant refactoring or even re-architecting.
- The installation process for different releases of the software involves doing different things. You might call these "snowflake releases". A smell that suggests you have snowflake releases is when comprehensive release notes need to be written for each release.

- Releases don't happen very often, which contributes to each release having it's own special one-off process. Which makes it difficult and expensive to release frequently. This is the automation fear spiral, again!
- Environments are not consistent, so deployment to each one is a custom effort. Deployment to an environment needs special knowledge of how to tweak the software, its configuration, and the environment's configuration, to make everything mesh. This is a variation of the snowflake release.

Many teams do manage to automate complex deployment processes that require orchestration across multiple servers and infrastructure elements. Remote command execution tools, covered in the next section, can be useful for automating these deployments.

However, the best approach is to architect, design, and implement software and its supporting infrastructure to keep deployment simple. Close collaboration between software developers and infrastructure developers is important to keep the seams between them smooth. As a former colleague of mine, Ben Butler-Cole once said, "Build systems, not software".

Continuous Delivery

It's critical that, throughout development, software is continuously deployed to test environments that accurately reflect their final production deployment environment. This deployment should be done in the exact same way in every environment, with all of the same restrictions applying to the test environments that will apply to production.

Doing these continuous test deployments ensures that the deployment process is well-proven. If any changes are made to the application that will break deployment to production, they will first break deployment to the test environment. This means they can be immediately fixed. Following this practice rigorously makes deployment to production a trivial exercise.

The twelve-month Continuous Delivery project

I worked with a large team at a major bank on a project that took twelve months to release into production. We insisted on frequently deploying to a "controlled" environment, with production constraints, several times a weeks. We created an automated process that was agreed to be suitable for production, including needing a human to manually enter a password to confirm it was suitable.

Many people on the team resisted this rigorous process. They thought it would be fine to deploy to less restricted environments, where developers had full root access and could make whatever configuration changes they needed to get their builds working.

They found it frustrating that some of us insisted on stopping and fixing failed deployments to the controlled environment - surely we could worry about that later?

But we got the deployments working smoothly. When the release phase came, the bank's release management teams created a six-week project plan for getting the "code complete" build ready for production release. The first step of this plan called for the production support team to spend two weeks getting the software installed on the pre-production environment, which would identify the work and changes needed to make the software ready for production use.

The support team got the software deployed and working in pre-production in one day, rather than two weeks. They decided that the work to prepare the production environment would also be less than one day. They were delighted - they saved over five weeks of work. Even better, fixes and improvements demanded by the business after go-live could be whipped through the release process, and everyone was confident the software was rock-solid.

We'll discuss supporting the development, release, and operations of in-house software in more detail in (to come). Implementing Continuous Delivery for infrastructure is discussed in "Continuous Delivery (CD)" on page 118. The book Continuous Delivery, by Jez Humble and David Farley, is the canonical resource on the subject.

Packaging software

The ideal way to deploy in-house software is to do it the same way you deploy any other software. Package builds using a native packaging format for your system that you use to install external software, for example .rpm files, .deb files, Nuget packages for Windows. These have standard mechanisms for pre- and post-installation scripting, version management, and distribution using repository systems like apt-get and yum.

Language packaging formats like Ruby gems can also be suitable, as long as they can handle everything needed to install and configure the software. This should include things like making sure the software is installed as a system process, creating directories, and using the correct user accounts and permissions.

Other language packaging formats, like Java .jar and .war files, don't do these on their own, so they need other tooling. I've seen teams package these up inside .deb or .rpm files, which include scripts that carry out installation activities, including deploying .war files into an application server.

When an application is built into a system package and made available on a repository, deployment is simply a part of the configuration definition for the server. The following Chef recipe snippet assumes that the version to be deployed has been set in the configuration registry. On the project that this example came from, we do this from our change management pipeline. The deployment stage in our GoCD server runs a script that sets

the version in the Chef server. This recipe then uses this to install the package from our yum repository.

Example of installing software with Chef.

```
package "usermanager" do
  version node[:app][:version]
  action :install
end
```

Using packages this way can be challenging when there needs to be some kind of orchestration, especially if we have a pool of servers. For example, if you need to run a script to make database schema changes, this script can run as part of the package installation. But if the package is installed on multiple servers in a pool, then the schema update may run multiple times, perhaps concurrently, which can have bad results. Database locking can help - the first script locks the database, and the others, finding it locked, skip the schema update.

Deploying microservices

Microservice architecture[1] lends itself particularly well to safe, simple, and reliable software deployments in larger, more complex environments. As it happens, using a microservice architecture also requires mature infrastructure management. So microservices and infrastructure as code are complementary approaches, each supporting the other, and together providing resilient, responsive, and adaptable services to organizations.

Microservice architecture makes deploying changes to a complex system simpler and more reliable because it emphasizes loose coupling between the deployable components. Each microservice should be tolerant of upstream and downstream dependencies. The microservice should gracefully handle situations where its dependencies are an unexpected version, down, or giving errors.

Graceful handling might mean that the microservice itself returns an error to its consumers or end users. But it doesn't crash, corrupt data, or produce incorrect results. Once the problem with its dependency is resolved, the microservice resumes working correctly, without needing someone to restart, reconfigure, or redeploy it.

On the flipside, a microservices architecture requires sophisticated infrastructure management. Organizations using microservices tend to have numerous deployable services, which are built and deployed frequently. Amazon's ecommerce site is implemented

1. For more on Microservices, see Martin Fowler and James Lewis' series of posts, and Sam Newman's excellent book Building Microservices.

with over 200 microservices[2]. They typically have dozens or hundreds deployments to production a day.

Even smaller organizations with fewer than a dozen microservices will struggle to maintain and deploy these on a static infrastructure. A microservices architecture not only needs dynamic environments with rigorously consistent configuration, it also needs to be able to roll out changes and improvements to servers easily and safely. Manually driven change management processes don't cut it.

Conclusion

This chapter has given examples of some of the services that may be needed in managing a dynamic infrastructure, and how they can be implemented and used with infrastructure as code. There are of course many other services that and concerns that could be necessary for different teams and their infrastructures. Hopefully the examples here make it easy to think about how services you may need can fit into this approach.

Now that we've looked at the foundations and principles of infrastructure as code, Part II of this book will delve into more detailed patterns and practices for implementing it, including how to provision and maintain servers.

2. For more on Amazon's use of microservices, see the article Amazon Architecture on highscalability.com

CHAPTER 4

Patterns for provisioning servers

Provisioning a server is the process of making it ready to use. Typical provisioning tasks include allocating hardware, creating a server instance, partitioning disks, loading an operating system, installing software, configuring the server, setting up networking, and registering the server with services like DNS which are needed to make it fully usable.

These activities may be done at different points of a server's lifecycle. Most simply, they can all be done when creating each new physical server or virtual machine instance. But for many provisioning activities it may be more efficient to do them up front, saving them on a server template, and then use that template to create servers. This way, rather than repeating the same activity every time a new server is created, it can be done only once. This makes creating new servers faster, and can help avoid inconsistencies.

Provisioning is not only done for new servers. Sometimes an existing server is re-provisioned, changing its role from one to another. Or a large server may be used for multiple roles, and occasionally have a new role added. For example, many IT Ops teams run multiple infrastructure services like DNS, DHCP, file servers, mail servers, etc. together on shared servers, rather than having a dedicated server for each role. This was a much more usual thing to do before virtualization and automation made it easy to manage a larger number of smaller servers. In this kind of environment, it's common to shift certain services between existing machines as needed to balance workloads.

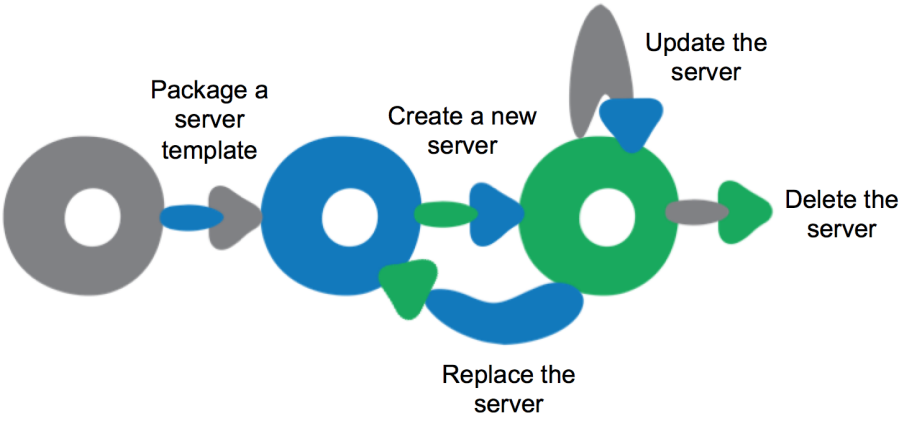
This chapter explores the types of things that happen as part of provisioning, and different patterns for provisioning.

A server's life

It can be helpful to think about the lifecycle of a server as having several phases:

- Package a server template
- Create a new server

- Update a server
- Replace a server
- Delete a server



Package a server template

As mentioned above, it's often useful to create a template, a base image with common elements already provisioned, that can be used to create servers. The process of creating a template may start with an operating system image, such as an ISO file for an OS setup disc, or it may start with a previously created template. A server instance is created, changes are made to it, and then it is saved to a snapshot of some kind.

Some infrastructure management platforms have direct support for templates, such as Amazon's AMIs and VMWare's Templates. Others, such as Rackspace cloud, don't have explicit server templates, but the functionality used to create snapshots for backups can easily be used for templating.

This process may be manual, or may be automated, possibly using configuration management tooling to prepare the template before saving it. Templates may be very simple, with little more than the base operating system, or they may be heavier, with everything needed for a particular server role already installed. The most minimal templating approach is to directly use an OS installation ISO as a template, creating each new server instance by booting and running the OS setup.

Tradeoffs of different approaches are discussed later in this chapter, and ways to implement template packaging are covered in Chapter 6.

Create a new server

Creating a new server involves allocating resources such as CPU, memory, and disk space, and instantiating a virtual machine or hardware server. This phase also usually involves configuring networking, including assigning an IP address and placing the server in the networking architecture, for example adding it to a subnet. The server may also be added to infrastructure orchestration services such as DNS, a configuration registry, or monitoring (see Chapter 3).

Software and configuration may be installed at this point, particularly those needed for a given role, if they're not installed in the template. In many cases, updates may be run to apply system updates and security patches released after the server template used to create the server was built.

Again, different approaches to handling provisioning activities in templates versus server creation time are discussed below, as well as in the following chapters.

Update a server

A good server creation process ensures new servers are consistent when they are created. However, this consistency doesn't last. Changes may happen on servers over their lifetime, which may cause them to diverge. And the contents of server templates, and activities that happen on server creation, tend to be continuously updated and improved by the infrastructure team, which means that servers created at different times are likely to be different.

Automated configuration tools such as Ansible, Cfengine, Chef, and Puppet were created to deal with this problem. Chapter 7 goes into this subject in detail.

Replace a server

Some changes may require completely replacing a server, rather than simply updating an existing one. For example, it may not be possible to upgrade a runnign server to a major new version of an OS. Even for less dramatic changes, it may be easier to completely rebuild a server rather than changing it, especially when automation can make rebuilding quicker and more reliable than selectively upgrading parts of it.

Some change management strategies involve replacing servers rather than modifying them. A new server can be built and tested before bringing it online. It is then swapped out for the existing server, which is taken offline but held in reserve for some period, giving a quick rollback option in case something goes wrong with the new server.

Replacing servers seamlessly is a key to continuity strategies, ensuring major changes to infrastructure can happen without interrupting service or losing data. This topic is discussed in more detail in (to come) and in the (to come).

Replacing a server usually requires updating infrastructure services such as DNS and monitoring. These tools should handle the replacement of a server gracefully, without interrupting service, and without losing the connection of data for services that run across the lifespan of multiple servers. Chapter 3 covers this in more detail.

Delete a server

Destroying a server may be simple, but as with creating and replacing servers data may need to be retained, and infrastructure services need to be updated. It's generally useful to make sure that information about deleted servers are retained after a server is deleted. This is clearly true for logs and metrics, although some monitoring systems aren't good about keeping historical data once a server is removed from the list of things to monitor.

Other events in a server's life

The lifecycle phases are the major things that happen to a server, but there are some other interesting events that occur, which have implications for provisioning servers.

Recover from failure

Cloud infrastructure is not necessarily reliable. Some providers, including AWS, have an explicit policy that server instances may be terminated without warning, for example if the underlying hardware needs to be replaced. Even providers with stronger availability guarantees have hardware failures that affect hosted systems.

As a rule, a team that has responsibility for managing the lifecycle of server instances needs to take responsibility for continuity of operation. Infrastructure as code can be used to make services more reliable than the individual system components they run on. The "replacement" phase of a server's lifecyle, as described above, is about replacing servers while maintaining data and service continuity. If this can be done easily and routinely, a bit of extra work can ensure it happens automatically in the face of failure. This is discussed in more detail in (to come).

Resize a server pool

Being able to easily add and remove servers from a pool, such as load balanced web servers, is a benefit of being able to build servers repeatably and consistently. Some infrastructure management platforms and orchestration tools offer functionality to do this automatically. This can be done in response to changes in demand, for example adding web servers when traffic increases and removing them when it decreases, or on a schedule to match predictable patterns of demand, for example shutting down services overnight.

For many services it's useful being able to easily add and remove capacity even if it's not automated, for example if someone can use a UI to quickly add and remove servers.

Create a new environment

For some services, an IT department can create and manage multiple independent instances of the service, for different users or purposes. For example, different teams may want to have their own instance of a ticketing application like Jira. Or a service instance may be created to test an application, especially one developed in-house; for example a QA environment, performance testing environment, etc.

For many organizations, these tend to be created and managed inconsistently, with problems similar to those we've discussed for infrastructure in general - configuration drift, snowflakes, etc. Testing environments in particular tend to be managed statically, and although they are intended to accurately reflect production environments, often become different enough that it causes quality and release management problems.

Infrastructure as code practices can be applied at the service level, to ensure that new application deployments are easily and consistently created, and that changes are applied consistently.

This then opens the possibility of creating application instances on demand. Departments can easily create a standard application instance for their own use, with minimal or no time needed from IT. Developers can create individual instances as needed, tearing them down when finished. Multiple QA instances can be created, so different work streams can be tested without having to schedule time on a shared environment, and torn down when not in use to save costs.

Reconfigure hardware resources

Many virtualization and cloud platforms offer an API to changing a server's hardware specifications, such as RAM or CPU. I've rarely seen situations where it's been useful to automate this, it's usually easier to rebuild a server after tweaking the configuration that manages server resources. - depends on the level of automation. With good, reliable automation, it's true that it's easier to rebuild. Changing an existing server tends to be more appealing for servers that aren't easy to automate. If you find that the idea of rebuilding a server is scary enough that you prefer to manually reconfigure its resources, this is probably a smell that you need to work on making it easier to reproduce the server.

What goes onto a server

It's useful to think about the various things that go onto a server, and where they come from.

This chapter describes some different models for categorizing the things that go onto a server, but it's important to remember that these kinds of categorizations are never going to be definitive. They're fine as far as they are useful to map out approaches to managing servers. If you find yourself obsessing over the "right" category for a particular thing from your infrastructure, take a step back. It probably doesn't matter.

Types of things on a server

One way of thinking of the stuff on a server is as software, configuration, and data. This is useful for understanding how configuration management tools should treat a particular file or set of files.

Table 4-1. Things found on a server

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

The key difference between configuration and data is whether we want automation tools to automatically manage the contents of the file. So even though some infrastructure tools do care about what's in system logfiles, they're generally treated as data files. When building a new server, we don't expect our provisioning tools to create logfiles and populate them with specific data.

Data is treated as a black box by configuration management. Data that we care about needs to be managed so that it survives what happens to a server, as discussed in (to come). Provisioning a server may involve making sure that the appropriate data is made

available, for example mounting a disk volume or configuring database software so that it replicates data from other instances in a cluster.

Quite a lot of system and application configuration will be treated as data by automated configuration management. An application may have configuration files that specify network ports and other things which the configuration system will define. But it may also have things like user accounts and preferences which get stored in other files. Automated configuration will often treat these as data, since their contents aren't defined by configuration definitions.

Where things come from

A given server instance has many different elements, which tend to come from difference sources.

Table 4-2. Where things on a server come from

Base Operating System	Typically comes originally from an installation image – CD, DVD, or ISO. Often there are optional system components that may be chosen during the OS setup process.
System package repositories	Most modern Linux distributions support automatically downloading packages from a centralized repository. For example, RHN/Yum repositories of RPM packages for Red Hat-based distributions, apt repositories of .deb packages for Debian-based distributions. Repositories may be hosted or mirrored internally, and/or public repositories may be used. You may add packages not available in the main public repositories.
Language and other platform repositories	Many languages these days have library formats and repositories to make them available. For example, Ruby Gems, Java Maven repositories, Python PyPi, Perl CPAN, NodeJS NPM, etc. The principle is the same as the system package repositories.
Third party packages	Things not available in public repositories, e.g. commercial software. May be managed by putting them into internal, private repos.
In-house software packages	Again, may be put into internal repos. In many cases these are handled separately, e.g. deployment tooling pushes a build onto servers.

It's often best to install third party and in-house packages from an internal repository, so they can be managed and tracked consistently. Some IT Ops teams like to use version control repositories like Git and Perforce to manage these, although some tools are better than others at storing large binary files.

Some teams install these packages manually. In some cases this can be pragmatic, especially in the short term while automation is still being implemented. However it obviously doesn't support the principles discussed in this book, such as repeatability.

Another option for distributing packages to servers is within the definition files of a configuration tool. Puppet, Chef, and Ansible all allow files to be bundled with modules, cookbooks, and playbooks. Although this is more repeatable than manual installation, they tend to make configuration definitions unwieldy, so this is really only suitable for a stopgap until a private artifact repository of some sort can be installed.

Server roles

Different servers will need different things installed on them depending on what they will be used for, which is where the concept of roles comes in. Puppet calls these classes, but the idea is the same. In many models for server roles, a particular server can have multiple roles.

One pattern is to define *Fine Grained Roles*, and combine these to compose particular servers. For example, one server might be composed of the roles TomcatServer, MonitoringAgent, BackupAgent, and DevelopmentServer, with each of these roles defininging software and/or configuration to be applied.

Another pattern is to have a *Role Inheritance Hierarchy*. The base role would have the software and configuration common to all servers, such as a monitoring agent, common user accounts, and common configuration like DNS and NTP server settings. Other roles would add more things on top of this, possibly at several levels.

It can still be useful to have servers with multiple roles even with the role inheritance pattern. For example, although production deployments may have separate web, app, and db servers, for development and some test cases it can be pragmatic to comine these onto a single server.

Provisioning in the template vs. server creation

A natural question is which configuration elements and software packages should be provisioned in the server template, and which ones should be added when a new server is created. Different teams will have different approaches to this.

Provisioning at creation time

One end of the spectrum is minimizing what's on the template and doing most of the provisioning work when a new server is created. New servers always get the latest changes, including including system patches, software package versions, and configuration options.

Managing templates is simplified. There will be few templates, probably only one for each hardware / OS combination used in the infrastructure. The templates don't need to be updated very often, since there is little on them that changes, and in any cases changes can be made when servers are provisioned.

This is often appropriate when the team isn't able to invest in sophisticated template management tooling. Many teams manage and update templates manually, since it doesn't happen very often. Most of the tooling effort goes into configuration management, which runs the same processes to provision new servers that are run regularly to keep servers up to date.

Keeping template minimal makes sense when there is a lot of variation in what may be installed on a server. For example, for an organization where people can create servers by self-service, choosing from a menu of options of what to install, it makes sense that the provisioning happens more dynamically when the server is created, rather than trying to maintain a large library of templates.

The main drawback of doing most of the provisioning work every time a new server is created is that it takes longer to create servers. The same work is done every time a new server is created, which can feel wasteful. For infrastructures where automatically creating new servers is a key part of disaster recovery, scaling, and/or deployment, a heavyweight server creation process means these things take more time. Larger scale disaster recovery scenarios can be especially painful, if provisioning new servers requires infrastructure services which are themselves being recovered.

An important consideration when using heavier server creation is network usage, especially where software packages and other server elements are downloaded from external repositories. It may pay to maintain local mirrors or caches of these repositories to make provisioning quicker, more reliable, and to save on bandwidth costs.

The JEOS approach - Just Enough Operating System - pares the system image down to the bare essentials. This is mainly intended for building virtual appliances and embedded systems, but with a highly automated infrastructure with short lived servers, it's a good idea to consider how far you can strip the OS down. This has benefits for speed, security, and resource usage.

Provisioning in the template

At the other end of the provisioning spectrum is putting nearly everything into the server template. Building new servers then becomes very quick and simple, just a matter of selecting a template, and handling instance-specific configuration like setting the hostname. This can be useful for infrastructures where new instances need to be spun up very quickly, for example in order to automatically scale to handle rapidly shifting load patterns.

Doing all of the significant provisioning in the template, and disallowing changes to anything other than runtime data once a server is created, is the key idea of immutable servers (see "Immutable server flow" on page 110).

Doing more provisioning in the template requires more mature processes and tools to manage templating. Even teams which don't have very many templates need to build new templates quite often, every time there is a change to software or configuration on the template. It's not unusual to bake new templates at least weekly, and some teams have new templates being packaged and rolled out several times a day.

As of this writing, building templates is becoming easier thanks to tools like packer and Aminator.

Balancing provisioning across template and creation

Although the immutable infrastructure approach is increasing the number of teams moving towards the "heavy template" end of the spectrum, most teams handle provisioning across both phases, template packaging and server creation. There are a few considerations and techniques to think about.

The cycle time for making a change that involves updating a template is longer, from making the change, testing that it works, and rolling it out. Changes that happen often, or changes that are experimental - trying out a new software package, for example - are easier to do when it can be done at server creation time. When provisioning new servers uses the same configuration management that is used for updating servers, this can make experimenting with new changes even faster, since it may not need building a new server.

So one useful pattern is to run the same configuration management update tooling when packaging a new template, when creating new servers, and to update existing servers. When you want to make a change, put it into the configuration definitions so that it's quickly applied to new and existing servers, without building new server templates.

You can then build new versions of your server templates periodically. This rolls up the changes which have been applied at creation time into the template. As time goes by without refreshing the server templates this way, the server creation process becomes slower and slower, and the templates become less relevant.

Server provisioning in larger organizations

I've run into some organizations which struggle to fit automated server provisioning into their processes. This is often because they have accumulated cumbersome structures and governance requirements over time, which have been designed assuming servers are provisioned manually.

For example, one large financial institution, when asked to create two virtual machines for development and testing from their standard catalog, needed 6 weeks to do so. There were 8 different teams within the organization that needed to handle the VMs on their way to us, including change management, OS installation, account creation, message queue client team, DBAs (to install the database client software), Java team, security audit, and quality control.

After all of these people did their thing, using two different, very expensive enterprise software configuration systems, an enterprise virtualization management system, and request management and tracking systems, we received two VMs with slightly different configuration, on one of which the application failed to start because the user account had been configured incorrectly.

Larger organizations, especially those working under strict regulations like those found in finance, government, and those handling financial transactions and/or personal data, tend to have a number of stakeholders with different requirements that need to be taken into account when provisioning infrastructure.

The best way to handle this kind of environment is to engage the various stakeholders in defining these requirements and designing automated provisioning and configuration to enforce them. Rather than insisting each stakeholder have a hands-on role in carrying out repetitive provisioning activities manually, have them help create the scripts that carry out these actions. Even better, have them help define automated tests and monitoring to validate compliance. Doing this collaboratively with the various stakeholders is far more reliable and effective than mandating intrusive manual inspections.

Conclusion

This chapter has described approaches and patterns for provisioning servers. The core theme has been what to provision, and in what parts of the server lifecycle. The next two chapters will go into specific patterns and practices for creating servers (Chapter 5), and for building and managing server templates (Chapter 6).

CHAPTER 5

Patterns for creating servers

The previous chapter discussed provisioning in the context of the overall lifecycle of a server, which includes phases for building templates and creating servers. This chapter focuses on the server creation phase, including patterns, practices, and anti-patterns.

A good server creation process aligns with the general principles of infrastructure as code, leading to servers that are created easily, consistently, repeatably, and auditably. The process for creating servers should be self-documenting, with the knowledge and decisions captured transparently in the scripts and definition files used to execute the process, rather than in peoples heads' or even in wiki pages or documentation which quickly become out of date.

The server creation process

Most infrastructure management platforms, whether they're a public cloud like AWS or a virtualization product like VMWare, can create servers either from a user interface like a web UI or admin application, or by using a programmable API.

Once a new server image has been launched, there are typically changes made on the server itself, which may include installing and configuring software as well as system updates and configuration. Orchestration activities integrate the new server into network and infrastructure services such as DNS.

Launching new servers

The source for launching a new server instance might be:

- Cloning an existing server,
- Instantiating from a snapshot that was saved earlier from a running server,
- Building from a specially prepared server template such as an EC2 AMI,

- Or booting directly from an OS installation image such as an ISO file from an OS vendor.

Antipattern: Manually creating servers

The most straightforward way to create a new server is to use an interactive UI or command line tool to specify each of the options needed. This is usually the first thing I do when learning how to use a new cloud or virtualization platform. Even when I've been using a platform for a while and have more sophisticated scripting options available, it's still sometimes useful to use the basic tools to whip up a new VM to try out something new, or to help with debugging a problem by testing different things about how the creation process works.

The following example uses the AWS command line tool to create a new server:

```
$ aws ec2 run-instances
 --image-id ami-12345678
 --region eu-west-1
 --instance-type t1.micro
 --key-name MyKeyPair
```

But manually creating servers is not a sustainable way to routinely create servers.

Manually creating a server relies on someone deciding, for each new server, which options to choose. This is clearly not repeatable, at least not easily and reliably. Even if the same person creates another server later on, they may not remember exactly which options they chose the first time, and a different person is even more likely to configure the server at least a little differently than the first person did. It may be possible to refer back to an existing server to figure out the settings that were used, but this takes time and effort and can still lead to errors.

So manually building servers leads almost immediately to configuration drift and snowflake servers. Server creation is not traceable, versionable, testable, and is certainly not self-documenting.

Pattern: Wrapping server creation options into a script

A script is a simple way to capture the process for creating new servers so that it's easily repeatable. Check the script into a VCS to make it transparent, auditable, reversible, and to open the door to making it testable.

The aws command from the previous example could easily be saved in a shell script and committed to version control, taking a first step into infrastructure as code.

Avoid scripts that need to be edited every time.
I've seen more than one experienced systems administrator new to infrastructure as code write a script that creates a server, but needs to be edited every time its run to set parameters. This doesn't really automate the server creation process, but just helps make part of a manual process a bit easier.

For all but the simplest situations, a script to create new servers will need to use different options for different situations. The goal should be that, even though the person using the script will need to know what they want, they shouldn't need to remember the details of how to make it happen. These details, and the logic of how to assemble the various options to give the user what they want, should be captured in the script.

For example, a user may want to be able to create a server in one of several different environments - QA, Staging, or Production. Each environment may use a different subnet, but the user shouldn't need to know which subnet to assign the new servrer to. Instead, the user should pass the environment name to the script, which works out which subnet to use. In this case, the user may not even need to be aware that different subnets are involved.

Here is a simple example of a Ruby script for our example situation.

```
#!/usr/bin/env ruby

require 'aws-sdk'

# Get the environment from a command line argument
abort("No environment was specified. Usage: create-server <qa|stage|prod>") unless ARGV.size > 0
environment = ARGV.first

# Decide which subnet to assign the server to
subnet_id = case environment
when 'qa'
  'subnet-12345678'
when 'stage'
  'subnet-abcdabcd'
when 'stage'
  'subnet-a1b2c3d4'
else
  abort("Unknown environment '#{environment}'. Usage: create-server <qa|stage|prod>")
end

# Create the new server in the appropriate subnet
ec2 = Aws::EC2::Client.new(region: 'eu-west-1')
resp = ec2.run_instances(
  image_id: 'ami-12345678',
  min_count: 1,
  max_count: 1,
  key_name: 'MyKeyPair',
```

```
    instance_type: 't1.micro',
    subnet_id: subnet_id
  )
```

You can see how this could be extended to set other options for a new server, such as creating servers in different regions. We could also add other inputs to use for setting options in addition to the environment. For example, we could have roles like web server, app server, and database server, and use these to give the server a different AMI to build from, and a different instance type which has memory and CPU tuned for how the server will be used.

As the logic of how a server may be created becomes more complicated, good software engineering practices become important. The simple script could evolve to use modules and classes to keep the code well organized, and might have a configuration file to define which environments and roles are available, and which options to use for each.

Ideally, a team shouldn't need to invest the time and energy to create and maintain a complicated server creation toolset. As discussed in "Declarative infrastructure provisioning tools" on page 47, declarative tools which provision servers as well as the rest of the infrastructure are growing in popularity and maturity.

Antipattern: Creating servers by cloning a running server

My team and I loved cloning servers when we started using virtualization. Nearly every new server we needed was a variation of one we already had running, so cloning was a simple way to get a server already configured with everything it needs. But as our infrastructure grew, our team realized our habit of cloning servers was one of the reasons we were suffering from server sprawl and configuration drift.

We ended up with servers that were wildly different from one another, even when they had the same role. Whatever differences we had on our web servers were made worse every time we picked one of them - usually the one we thought was most up to date - to create a new server and tweak its configuration.

We tried declaring a particular server to be the master for all cloning, and tried to keep it in the ideal state, with all the latest updates. But of course, whenever we updated the master servers, the servers that we'd created before weren't updated to match.

A server that has been cloned from a running server is not reproducible. You can't create a third server from the same starting point, because both the original and the new server have moved on - they've been in use, so various things on them will have changed.

Cloned servers also suffer because they have runtime data from the original server.

I recall a colleague who was debugging an unreliable Tomcat server, and spent several days working on a fix based on strange messages he'd found in the server's logfiles. The fix didn't work. Then he realized that the strange messages hadn't actually been logged

on the server he was trying to fix. That server had been cloned from another server, and the messages had come from the original, so he had been wasting several days chasing a red herring.

Pattern: Creating new servers from a template

Cloning running servers isn't a good way to build servers consistently, but making a snapshot of a server and using that as a template can be. They key difference is that the template is static. If two new servers are built from a given template, they will start out identical even if the second one is built a week after the first, aside from whatever creation-time provisioning is done to them. A server cloned from a running instance a week later than another server cloned from the same instance will already be different, since it will have whatever runtime changes happened to the original server during that week.

Of course, even servers created from the same template won't stay consistent. As they run differences will creep in, so we still need strategies to update servers as discussed in Chapter 7.

Configuring new servers

After launching a new server, but before putting it into use, it usually needs to have changes made to it. These may include installing system updates and setting system configuration options, installing and configuring software, and/or seeding data onto the server. (Note that this generally isn't necessary when you're using the immutable server pattern discussed in "Immutable server flow" on page 110).

Assuming the infrastructure team is using server configuration tooling of the types discussed in Chapter 2, it usually makes sense to just run the tool on each new server. You could have a specific set of configuration definitions used for building new servers, but it's a better idea to have a single set that is applied at creation time, and they reapplied continuously to ensure consistency. This also avoids overlap and conflicts between two different sets of configuration definitions.

See "Patterns and practices for continuous synchronization" on page 103 for a discussion of the pushing and pulling models for applying server configuration - the same issue is relevant for using these tools when building servers.

One risk with the push model is if the implementation needs login details baked into the server template, which might be exploitable by attackers. This can be mitigated by having a one-off authentication secret created when the server is launched, available only to the service controlling the creation process, and discarded when creation is finished.

For example, the AWS EC2 API makes it easy for a server creation script to create and assign a unique SSH key to each new server instance as it is created. Rackspace cloud assigns a strong, random root password to new instances, returning it in the API call used to create th instance. The configuration definitions applied when the server comes up can lock the system down, disabling keys, passwords, or accounts as relevant.

Pulling changes onto new servers

Rather than having configuration applied using the push model, a newly launched server can pull its configuration. This involves having a setup script baked into the server template which carries out the steps to configure the server.

If you are building your own server templates, you can put everything you need into the template. For example, you can have your configuration agent already installed, as well as the script that runs on startup to carry out the configuration steps. The agent downloads the current configuration definitions from the network, so the new server will be fully up to date and consistent.

This model can even be used if you're not building your own server templates. Most Linux distributions include cloud-init, which provides support for bootstrapping new VM instances on most virtualization and cloud platforms. When creating a new server instance through the infrastructure platform's API, you will pass some data that cloud-init then uses. This can be a bit of shell script which bootstraps the configuration process.

Smoke testing new servers

If servers are automatically created, they can be automatically tested. It's too common to create a new server, hand it over, and only discover there was something wrong with it once people are trying to use it.

In an ideal world, automation means every new server is perfect. Since we don't live in an ideal world, we have to use automation scripts written by human beings, so mistakes are common. Automation scripts are also maintained by human beings, so even after a script has spun up many perfectly good servers, we can never assume the next one will be fine.

So it's good sense to sanity check each new server when its created. Fortunately, if we can automatically trigger the creation of new servers, we can very easily automatically trigger tests to check them afterwards.

Automated server smoke testing scripts can check the basic things we expect for all of our servers, things specific to the server's role, and general compliance. For example:

- Is the server running and accessible?

- Is the monitoring agent running?
- Has the server appeared in DNS, monitoring, and other network services?
- Are the necessary services (web, app, database, etc.) running?
- Are required user accounts in place?
- Are there any ports open that shouldn't be?
- Are any user accounts enabled that shouldn't be?

Smoketests should be automatically triggered. If we rely on a person to manually run the script, it will tend to be skipped, and become less useful over time. This is where using a CI or CD server like Jenkins or GoCD to create servers helps, because it can automatically run these kinds of tests every time.

Smoketests could be integrated with monitoring systems. Most of the checks that would go into a smoketest would work great as routine monitoring checks, so the smoketest could just verify that the new server appears in the monitoring system, and that all of its checks are green.

Conclusion

This chapter has described patterns and practices for creating new servers. The next chapter discusses building and managing server templates that can be used to create servers from.

CHAPTER 6

Patterns for managing server templates

This chapter builds on the previous chapters on provisioning and server creation to discusses ways to manage server templates.

Using server templates can help ensure new servers can be consistently and repeatably built. However, it does require having processes and systems for managing the templates. Templates need to be built, and then updated to incorporate updates and improvements. The process and tooling for doing this, as with those for creating servers, should support the principles of infrastructure as code. Templates should be repeatably built, consistent, auditable, and use self-documenting and self-testing processes.

Can't someone else do it?

The simplest practice for managing templates is to let someone else do it. Many OS vendors and cloud services provide pre-packaged server templates. These tend to have a base OS installation, plus default configurations, tools, and user accounts to make servers ready to use with the hosting provider. For example, the server template may be configured to use local DNS servers and OS package repositories, and may have utilities installed for finding metadata about the hosting environment.

Using stock templates offloads the hassle and complexity of managing your own templates. This can be especially helpful in the early days of setting up and running a new infrastructure. Over time, the team may find the need to push provisioning activities from server creation into the template, and so decide to move to managing their own templates to gain more control.

It is important to understand where the stock template comes from, and what user accounts, software, and security configuration are in place. Make sure the template doesn't have security issues such as well known passwords or ssh keys; even if they are only in place temporarily, attackers can exploit these brief windows of opportunity.

When using stock server templates, be sure to follow changes and updates to them, so your servers are not left with unpatched bugs or vulnerabilities.

The rest of the patterns and practices in this section assume you will be baking your own server templates.

The processs of building a server template

The process for building a template tends to look like this:

- Launch the starting image
- Create a new server from the image
- Customize the server
- Package (bake) the server into a template

In order to build a server template, you need a starting image. This may be a raw OS setup image, a stock template from a vendor, or a previously created server template. A few options are discussed in the next section.

A new server instance is created from the starting image, and then this server is configured, updated, and otherwise massaged to have everything a new server should have when it is first created. This may include things like applying system updates, installing common software packages like a monitoring agent, and standard configurations like DNS and NTP servers. It should also include hardening the server by removing or disabling unnecessary software, accounts and services, and applying security policies like local firewall rules.

Once the server is ready, it is baked, saved into a format that can be used to create new servers. With some infrastructure management platforms this is the same snapshot process used to back up running server instances. Other platforms have a special format or flag to mark the snapshot as a template, such as Amazon's AMIs, or VMWare's server templates.

Principle: Build templates repeatably
The infrastructure as code principles of repeability and consistency lead to the idea that server templates themselves should be created in a way that is repeatable, so that the servers created from them are consistent. The following patterns and practices aim to help with this.

Launching an origin image for templating

Server templates are built from an origin image. This section covers patterns and practices for this.

Practice: Build templates from clean servers

Templates should be created from a clean server, one that has never been used for any other purpose. This way the only history and runtime data on a new server is left over from the process of setting up the template, not from any kind of production use.

Pattern: Baking templates from scratch - using an OS installation image

One approach is to create a new server directly from an OS installation image, like a the OS installation ISO from the vendor. This gives a clean, consistent starting point for building templates. This tends to be an option for virtualization platforms, but most IaaS cloud platforms either don't provide a way to build servers directly from ISO's, or else doing it is difficult. In these cases, the most viable option is building from a stock template.

Pattern: Building templates from a stock template

As discussed earlier, many infrastructure management platforms, especially IaaS clouds, provide stock server templates that you can use to build your own servers and templates. These images may be provided by the platform's vendor, by the OS vendor, or even by third parties or user communities. Amazon AWS has the AMI marketplace with server templates built and supported by a variety of vendors.

A stock templates can be used directly to create servers, or it can be used as the basis for creating your own customized templates. Most stock templates are tailored to the infrastructure platform. For example, they may have pre-installed toolsets for integrating with functionality and services of the hosting platform. You can then add your own organization's customizations onto this, for example adding monitoring agents, DNS configuration, etc.

You can also take the opportunity to improve aspects of the templates to suit your organizations needs. Stock templates are usually intended as a starting point for multiple uses, so they might have installed software packages that you don't need. You can strip them down in order to make them more secure, to reduce their resource footprint, and/or to make them smaller so spinning up new servers is faster.

Updating server templates

You will need to update templates from time to time, in order to make improvements, and to apply updates to the system and software installed on them. This section discusses ways to do this.

Patterns for updating templates: Reheating or building fresh templates

One approach for updating templates is to use the previous version of the template as a starting image, so you only need to make the latest changes - reheating the template. The other approach is to rebuild a fresh template from the same starting point as the previous version, for example the OS installation image or the stock template.

Rebuilding fresh templates makes sense when you have an automated process for customizing and updating the server before baking it into a template, as discussed below. It may make automation simpler, since the starting point of the server before running the updates doesn't change.

Reheating templates is a straightforward way to handle updates when you don't have this well-automated. The steps to update may be simple, for example running the update commands for the different packaging systems used on your servers (yum, gem, etc.), then baking the new template.

In either case, you should consider the process for refreshing the original image used for the templates. As new OS updates come out, it can be good to build new templates from the latest installation image or stock template, rather than running upgrades on a template from the older version. This helps keep templates clean and consistent, otherwise they may tend to accumulate cruft.

Practice: Template versioning

It's important to call out that when we talk about updating a template, what we really mean is creating a new template image to be used instead of the old one. The old template image should not be overwritten. The reasons for this are similar to the reasons for using version control for files - the ability to troubleshoot and roll back if we find problems with changes, and visibility of what changes were made and when. We should make sure there is a way, for any running server, to trace back the version of the server template used to create the server.

Some infrastructure platforms have some features that can support versioning, but I haven't seen one that implements it fully for you. So teams need to implement at least part of the versioning themselves. Tagging features can help, otherwise template naming conventions may be used.

For example, I like to name templates with the format `${TEMPLATE_ROLE}-${YYYYMMDDHHSS}`. The `${TEMPLATE_ROLE}` is the name of the template type, which might be something like "base", "centos65", "dbserver" or something else, depending on what kinds of templates our team uses. The datestamp string gives us a unique version number each time we build a new version of that particular template. When creating a new server, the creation script will know the template type to use, usually based on the server role,

and will either select the latest version number, or else choose the appropriate version number based on which template has been promoted to the relevant stage of the pipeline (see (to come) for more on promoting templates.)

If your infrastructure management platforms has tagging functionality, you can use this instead of overloading the template name to include version numbers. You can add tags to templates for the template role and template version when you bake them. Server provisioning scripts can then refer to these tags to select the right template to use.

It should be possible to trace a running server instance back to the original template version used to build it. This can be used to debug problems, and may also be useful for cleanup routines that ensure servers are kept up to date. Some infrastructure platforms, including EC2, directly support tracing instances to their original image. Otherwise, if the platform has tagging support then a tag could be used. If neither of this is an option, you could add fields to servers in your configuration registry (assuming you're using one), or in some other centrally stored location, log, etc.

While it could also be handy to store this in a file somewhere on the server itself, this won't be helpful if the server crashes or becomes inaccessible, at which point you may really want to trace the details of the server's origin.

Cleaning old templates

It's usually necessary to clean out older template versions, to save storage space. Teams that update templates frequently, for example those using immutable servers, will find that this consumes massive amounts of storage if they don't prune them.

A rule of thumb is to keep the templates that were used to create the servers currently running in your infrastructure, to ensure you can rapidly reproduce the server in a pinch.

Retaining templates for legal reasons

Some organizations have a legal requirement to be able to reproduce, or at least prove the provenance, of their infrastructure, for some number of years later. Because they use so much storage, archiving server templates may be a difficult and expensive way to meet this requirement. If your templates are built automatically it may be a viable alternative to archive the scripts and definitions used to build the templates, as well as reference to the original third party images used to build from.

Check with appropriate auditors and/or experts. Often, the kind of traceability and regularity that comes with infrastructure as code is far better for compliance regimes than what most IT organizations consider "best practice".

Building templates for roles

We've discussed the use of server roles in "Server roles" on page 74. It may be useful to create a separate server template for each role, with the relevant software and configuration baked into it. This needs more sophisticated (i.e. automated) process for building and managing templates, but makes creating servers faster and simpler.

Different server templates may be needed for reasons other than the functional role of a server.Different operating systems and distributions will each need their own server template, as will significant versions. For example, a team could have separate server templates for CentOS 6.5, CentOS 7.0, Windows Server 2008 R2 and Windows Server 2012 R2.

In other cases, it could make sense to have server templates tuned for different purposes. Database server nodes could be built from one template that has been tuned for high performance file access, while web servers may be tuned for network I/O throughput.

Pattern: Layering templates

Teams which have a large number of role-based templates may consider having them build on one another. For example, a base server template may have the optimizations, configuration, and software common to all servers in the infrastructure. The base template would be used as the starting image to build more specific role-based templates for web servers, app servers, etc. This fits well, conceptually at least, with a change management pipeline (see (to come)), since changes to the base image can then ripple out to automatically build the role-specific templates.

But although this appeals to our techie-minded love of structuring and ordering things, in practice it isn't always useful. In fact it can actually make the end to end process of making a change and getting the usable template longer.

If you analyze the process of building a server template, it's often the case that spinning up the server, and then baking it into a template, take much longer than actually making changes to the server before baking it. If this is true for your team's process, then building two templates will taken much longer, no matter how much faster the server updating step takes.

However, layering can be useful when the base template is updated much less often than the role templates. For example, a team using the immutable server pattern may bake each new build of their application into a new template, which can mean building templates many times a day when the application is being actively developed. Building these templates from a base template that already has all of the heavy changes on it can speed up the application build and test cycle.

This is especially helpful when the base template can be built in a way that makes building role templates much quicker, for example by stripping it down to a minimal system so that it boots and bakes very quickly.

Practice: Sharing base scripts for templates

One of the motivations for layering is to get consistency when various role templates share common base elements. However, an alternative to layering that is often more efficient is to simply share common scripts or configuration definitions that are used to customize a server before it is baked into a template.

For example, base scripts may install common utilities, a monitoring agent, and common configuration settings. Each role template is built directly from the same stock server image, running the base scripts as well as role-specific scripts.

This is likely to be useful even when layering templates. It will be useful to put some server elements on certain role templates, but not all of them. For example, some server roles may need a JVM to run application servers and certain utilities, but java may not be needed across all servers.

Practice: Caching to optimize template customization

One of the most common causes for the step of customizing a server before baking it into a template is downloading software packages and updates. This can usually be optimized by making the software available closer to where templates are built, for example making it available on a file server, cache, or repository mirror. This may include OS installation images, system and language package repositories (apt, yum, gem, maven, etc.), and even more custom software and static data files.

As with any performance optimization, take the time to measure the different parts of the process before spending time and effort to optimize it, not to mention adding complexity to your infrastructure.

Automating server template management

For teams with simple needs, managing server templates manually may be sufficient. It's important that processes are repeatable and consistent, but a very simple process that isn't done very often can achieve this without automation. This can be done by using stock templates and doing most of the customization and updating at server creation time, investing effort in automating that part of the process.

That said, tooling for automating server templates is emerging and maturing, so that it's becoming easier to move away from manual processes. Even for simple templating this has benefits, not least of which is bringing template management into a change

management pipeline, so they can be tested and rolled out across an infrastructure in a controlled way.

Automating the template building process

It's pretty straightforward to automate the end to end process of building a template, from spinning up a server, applying updates, and then baking the template and making it available for use. "Tools for packaging server templates" on page 38 in Chapter 5 described tooling that can be used for building templates.

Automatically customizing servers before baking

Tools like Packer that automate the end to end template building process typically allow you to trigger scripts or configuration management tools to run on the server to customize it before baking it into a template.

For teams which use a server configuration tool, it often makes sense to use the same tool during template packaging. They'll often define a subset of configuration definitions that should be run to customize server templates. This then speeds up the process of running the server configuration tool again when creating a new server, because much of the common work has already been done.

On the other hand, the things that need to be configured during template building are often straightforward enough that simple shell or batch scripts can be used. Teams using immutable servers, who don't run a server configuration tool to update running servers, often find they can build server templates with them as well.

Full-blown configuration management tools are intended for a more complex use case: continuously synchronizing configuration on running servers whose state may have changed. Configuration changes made when building a template are one-off, with a well-known and controlled starting state - a clean stock template or freshly installed OS.

Pattern: Automatically testing server templates

Automatically building templates leads naturally to automatically testing them. Every time a new template is built, the automation tooling can spin up a new VM from it, run tests to make sure it's correct and compliant, and then mark the template as ready for use. Scripts that create new servers can then automatically select the newest template that has passed testing.

As with any automated testing, it's important to keep test suites pared down to those which actually add value, by preventing common errors and keeping confidence in the automation. Automatically testing server templates falls into the category of testing that may be more work than it's worth. It's best not to implement template testing until the

team finds that they have issues which would be usefully solved this way. Designing an effective testing regime is the topic of the testing chapter (Chapter 9).

Conclusion

The past three chapters have focused on aspects of server provisioning. The next chapter moves onto ways of keeping servers updated.

CHAPTER 7

Patterns for updating and changing servers

Dynamic infrastructure makes it easy to create new servers, but keeping them up to date once they've been created is harder. This combination often leads to trouble, in the shape of a sprawling estate of inconsistent servers. As we've seen in earlier chapters, inconsistent servers are difficult to automate, so configuration drift leads to an unmanageable spaghetti infrastructure.

So a processes for managing changes to servers is essential to a well-managed infrastructure. An effective change management process ensures that any new change is rolled out to all relevant existing servers, as well as being applied to newly created servers. All servers should be up to date with the latest approved packages, patches, and configuration.

Changes to servers should not be allowed outside the automated process. Unmanaged changes lead to configuration drift and make it difficult to quickly and reliably reproduce a given server. If changes are routinely made by bypassing the automation, then this is a sign that the processes need to be improved, so that they are the easiest and most natural way for team members to work.

Goals for the server change process:

- Changes are rolled out to all of the relevant existing servers
- Servers which are meant to be similar are not allowed to drift into inconsistency
- The automated process is the easiest, most natural way for team members to make changes

Our process for updating servers should be effortless, so that it can scale as the number of servers grows. Making changes to a server should be a completely unattended process. A person may initiate a change, for example by committing a change to a configuration definition. Someone may also manually approve a change before it is applied to certain

parts of the infrastructure. But the change should roll out to servers without someone turning a handle.

Once a well-oiled server change process is in place, changes become fire and forget. Commit a change, and rely on tests and monitoring to alert the team if a change fails or causes an issue. Problems can be caught and stopped before they're applied to important systems, and even those that slip through can be rapidly rolled back or corrected.

Characteristics of an effective server change process:

- Changes are effortless, regardless of the number of servers affected
- Unattended application of changes
- Errors with a change are made visible quickly

This chapter drills into the main server change management models that were introduced in Chapter 2 ("Server change management models" on page 39), with some patterns and practices to give ideas for how to implement them.

Models for server change management

This chapter discusses four models for making changes to a server's configuration. These models were touched on in Chapter 2, but it's worth going into a bit more detail now, so that we can explore patterns and practices around these in the rest of the chapter.

Ad-hoc change management

The traditional approach has been to leave servers alone unless a specific change is required. Even with newer configuration management tools like Chef, Puppet, Ansible, etc., many teams still only run them when they have a particular change to make.

As discussed throughout this book, this tends to leave servers inconsistently configured, which in turn makes it difficult to use automation comprehensively. That is, in order to run an automated process across many servers, the state of those servers needs to be generally consistent to start with.

Configuration synchronization

With continuous synchronization, configuration definitions are regularly applied and re-applied to servers on an unattended schedule. This ensures that any changes made to parts of the system covered by those definitions are brought back into line, keeping consistency. This, in turn, ensures that the automation can be reliably run.

Configuration synchronization is a good way to keep the discipline of a well-automated infrastructure. The shorter the time between configuration runs, the more quickly issues

with the configuration definitions are found. The more quickly issues are found, the more quickly the team can fix them.

Most server configuration tools are designed with this approach in mind.

However, writing and maintaining configuration definitions has overhead, and there is a limit to how much of a server's surface area can be reasonably managed by definitions. This leaves other areas of a server's configuration vulnerable to changes outside the tooling, and therefore configuration drift.

Immutable infrastructure

Immutable infrastructure is the practice of making configuration changes by building new infrastructure elements, rather than making a change to the element that is already in use[1]. This ensures that any change is tested before being put into production, whereas changes made to running infrastructure could have unexpected effects.

The configuration of a server is baked into the server template. So the contents of a server are predictable - whatever is on the base image, and whatever configuration definitions and/or scripts that make changes when building the template. The only areas of a server which aren't predictable, and tested, are those with runtime states and data.

Immutable servers are still vulnerable to configuration drift, since their configuration could be modified after they've been provisioned. However, the practice is normally combined with keeping the lifespan of servers short - the Phoenix Server pattern. So servers are rebuilt as frequently as every day, leaving little opportunity for unmanaged changes. Another approach to this issue is to make those parts of a server's file systems which should not change at runtime read-only.

Immutable servers aren't really immutable

Using the term "immutable" to describe this pattern can be misleading. "Immutable" means that a thing can't be changed, so a truly immutable server would be useless. As soon as a server boots, its runtime state changes - processes run, entries are written to logfiles, and application data is added, updated, and removed.

It's more useful to think of the term immutable as applying to the server's configuration, rather than to the server as a whole. This creates a clear line between configuration and data. It forces teams to explicitly define which elements of a server they will manage deterministically, as configuration, and which elements will be treated as data.

1. The term immutable servers was coined by my former colleague Ben Butler-Cole when he, Peter Gillard-Moss, and others implemented this for ThoughtWorks Mingle SaaS.

Containerized services

The increasing popularity of standardized methods for packaging, distributing, and orchestrating lightweight containers creates opportunities to simplify server configuration management. With this model, each application or service that runs on a server is packaged into a container along with all of its dependencies. Changes to an application is made by building and deploying a new version of the container. This is the immutable infrastructure concept, applied at the application level.

The servers that host containers can be greatly simplified. They can be stripped down to a minimal system, with only the software and configuration needed to run containers. These hosts could be managed using configuration synchronization or the immutable server model, but in either case, because they are kept simple, their management should be simpler than for more frequently changing servers.

As of this writing, very few organizations have converted their infrastructure as radically as this. Most are using containers for a minority of their applications and services, typically ones that change often, such as software developed in-house. And infrastructure teams are finding that at least some core services work better running directly on the host rather than inside containers. But assuming containerization continues to mature, and becomes a standard way to package applications for distribution, then this could become a dominant model for infrastructure management.

General patterns and practices

Applying configuration during provisioning

The model used for server change management affects the provisioning process as well. Teams which use the configuration synchronization model typically use the same tooling and definitions for provisioning, running Ansible, Chef, Puppet, etc. when a new server is created. This ensures that the latest configuration definitions have been applied. In some cases, the same tool may be used when building server templates, perhaps applying a common subset of the definitions.

So a server management process might use a configuration management tool at several stages.

- Packaging the template - Apply a subset of configuration definitions that are common to all server roles built from the particular template.
- Creating the server - The configuration tool is run, applying all of the definitions for the given server's role. This prepares the server for its specified role, and also applies any changes that were made to the common definitions since the server template was packaged. This avoids the need to update server templates for every change that applies to it. But since the majority of common configurations and

software packages will have been baked into the template, creation should be quicker than it would be if they needed to be installed now.

- Updating the server - The configuration definitions are regularly applied, to keep the server's configuration in line and up to date.

Practice: Minimized server templates

The less you have on your servers to start with, the less you need to manage afterwards. This is also a good idea for security, performance, stability, and troubleshooting. To use this practice, ensure that server templates are stripped down, with all unnecessary packages, user accounts, configuration files, etc. removed. This is a good opportunity for general hardening routines, configuring the server to lock down user accounts, services, ports, etc.

This can be done by building templates from a stock image and running scripts to remove unwanted elements. This assumes the team knows everything to be removed. If a new stock image is used, for example for a new release of the OS distribution, then work is needed to make update the strip-down scripts to remove anything new that's appeared, and to drop scripting that removes elements no longer in the new stock image.

Alternatively, the template can be built by starting with a (Just Enough Operating System) OS distribution, which is already bare bones[2]https://developer.ubuntu.com/en/snappy/, RedHat's [Atomic]http://www.projectatomic.io/, and [CoreOS]https://coreos.com/]. The templating process would selectively add in only those things specifically needed.

Practice: Replace servers when the server template changes

When a server template is updated and new servers are created from it, the new servers may be different from existing servers created from previous versions of the template. For example, the template may be changed to use a new version of the OS distribution, which is likely to involve changes to a number of libraries and other server elements not directly managed by configuration definitions.

If servers are kept in use for a long time - months for example - and templates are updated several times, a team can end up with a fair bit of configuration drift that continuous synchronization won't resolve.

So it's a good practice to replace running servers whenever the template used to build them is updated. This should be done progressively rather than in a disruptive big bang. For example, a new template can be applied to test environments, then promoted to

2. Many Linux vendors are producing JEOS, or minimalist Linux distributions, such as [Snappy Ubuntu

more sensitive environments. See (to come) for more on this. Also, members of a server cluster can be replaced sequentially, to avoid disrupting service. See (to come) for ideas on this.

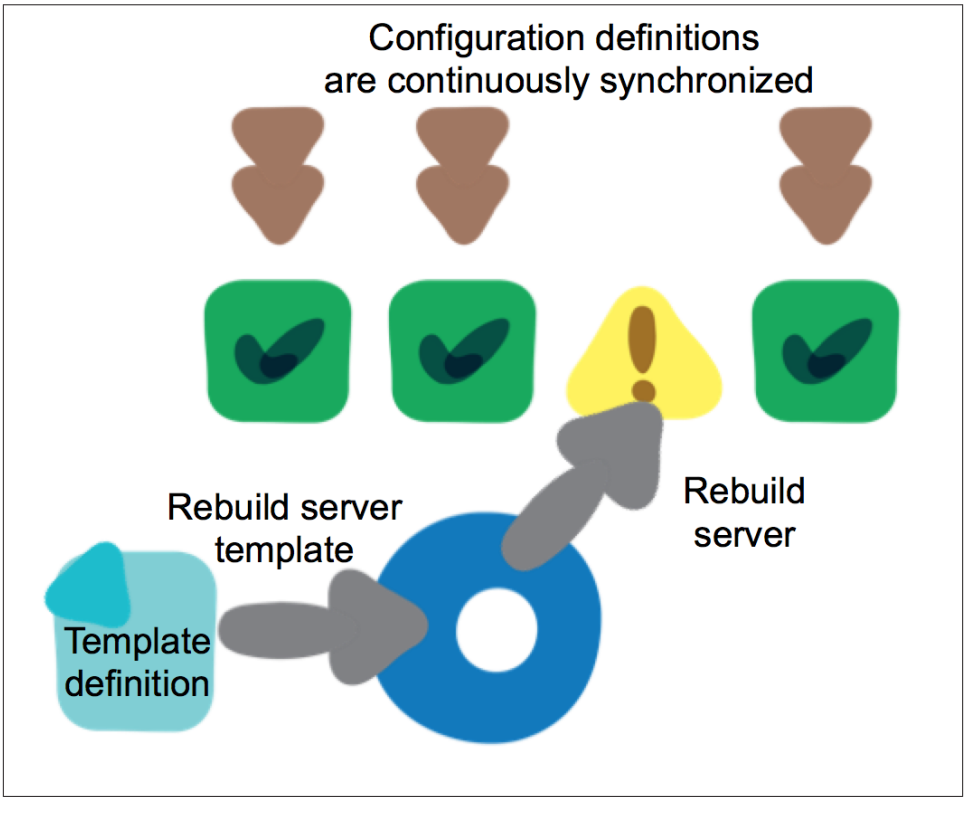


Figure 7-1. Server templates with continuous synchronization

Pattern: Phoenix servers

Some organizations have found that routinely replacing servers, even when the server template hasn't been updated, is a good way to combat configuration drift. In effect, this achieves the goal of having 100% of your machine's surface area automatically managed. Any changes that happen to parts of the server not managed by configuration definitions that are continuously synchronized will be reset when the server is rebuilt from the server template.

This is implemented by setting a maximum lifespan for all servers, and having a process that runs on a schedule to rebuild servers older than this limit. This process should ensure that service is not interrupted, for example by using zero-downtime replacement patterns (see (to come)).

Patterns and practices for continuous synchronization

Push versus pull

Continuous synchronization involves regularly running a process to apply the current configuration definitions to a given server. This process can work in one of two different ways. With the push model, a central process controls the schedule, connecting to individual servers to send and apply the configuration. With the pull model, a process on the server itself runs, downloading and applying the latest configuration definitions.

Pushing to synchronize

Many server configuration tools have a master server which pushes configuration to the servers it manages. Push configuration may need a client running on each managed server, which receives commands from the server on a network port or message bus. Other systems use remote command protocols like ssh to connect and execute configuration commands, in which case client software may not be needed. In either case, it's a central server which decides when it's time to synchronize, and orchestrates the process.

Ansible, for example, connects over ssh, copies scripts onto the target server and executes them. The only requirement for a machine to be managed by Ansible is an ssh daemon and the Python interpreter.

An advantage of the push approach is that it gives centralized control over when and where to apply configuration changes. This can help if changes need to be orchestrated across an infrastructure in a certain order, or with certain timings. Ideally we should design our systems with loose coupling that avoids the need for this type of orchestration, but in many cases it's difficult to avoid.

Pulling to synchronize

Pull configuration uses an agent installed on the managed server to schedule and apply changes. It could be a scheduled process using cron, or else a service that runs its own scheduler. The client checks a master or some central repository to download the latest configuration definitions, and then applies them.

The pull model has a simpler security model than the push model. With the push model, each managed server needs to expose a way for the master to connect and tell it how to configure itself, which opens up a vector for a malicious attack. Most tools implement an authentication system, often using certificates, to secure this, but of course this requires managing keys in a secure way. Ansible's use of ssh lets it take advantage of a widely used authentication and encryption implementation rather than inventing its own, but key management is still an issue.

With a pull system, there is still a need to ensure the security of the central repository that holds the configuration definitions, but this is often simpler to secure. Managed servers don't need to make any ports or login accounts available for an automated system to connect.

Depending on the design of the configuration tool, a pull-based system may be more scalable than a push-based system. A push system needs the master to open connections to the systems it manages, which can become a bottleneck with infrastructures that scale to thousands of servers. Setting up clusters or pools of agents can help a push model scale. But a pull model can be designed to scale with fewer resources, and with less complexity.

Masterless configuration management

Many teams find it effective to scrap the idea of a centralized server, even when the configuration tool they're using supports it. The main reasons for doing this are to improve stability, uptime, and scalability.

The approach is to run the configuration management tool in offline mode (e.g. chef-solo rather than chef-client), using copies of configuration definitions that have been downloaded to the managed server's local disk.

Doing this removes the master server as a potential point of failure. The definitions still need to be downloaded, but they can be hosted as static files. This can scale very well with minimal resources - object storage services (see (to come)) work very well for this.

Another way I've seen teams implement the masterless pattern is by bundling the definitions into a system package like an .rpm or .deb file, and hosting it in their private package repository. They run a script from cron which executes `yum update` or `apt-get update` to download and unpack the latest definitions before running the configuration tool.

Continuous synchronization flow

With continuous synchronization, configuration definitions are applied and repeatedly re-applied to a server. Figure 7-2 shows this process. Once a configuration definition has been applied to a server, each time the configuration process runs no more changes are made to the server. When the definition is modified and made available for the server, the change is made on the server the next time the configuration process runs.

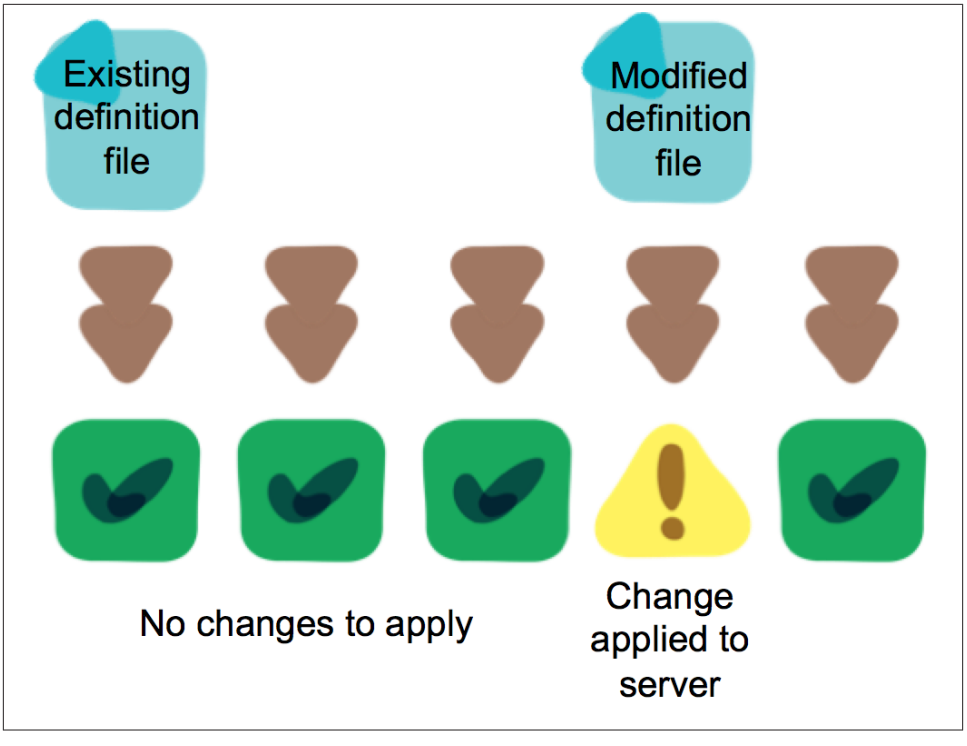


Figure 7-2. Making a change with configuration synchronization

This process also ensures that changes made outside of the automation are reverted, as shown in Figure 7-3. When someone makes a manual change on the server that conflicts with a configuration definition, the next time the configuration process runs it re-applies the definition, reverting the manual change.

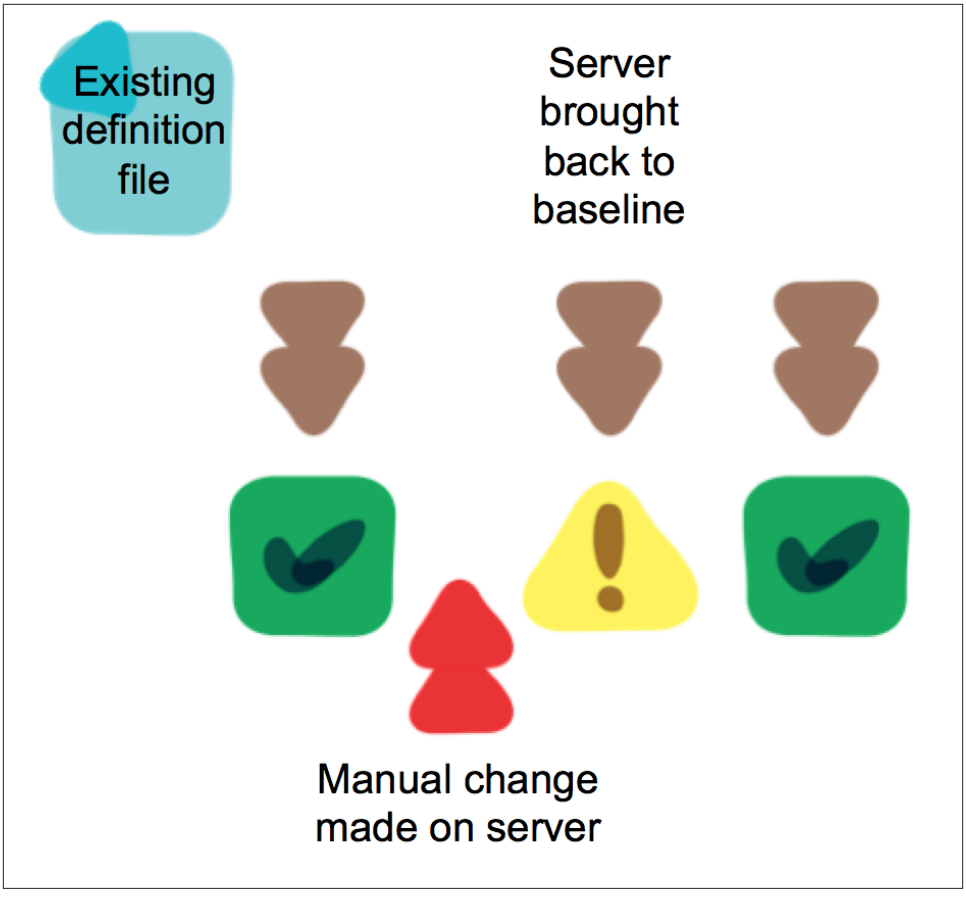


Figure 7-3. Reverting a manual change with configuration synchronization

The unconfigured country

When using continuous synchronization, it's important to be aware that the majority of a server will not be managed by the configuration definitions that are applied on every run. As shown in Figure 7-4, configuration definition files can only cover a fraction of the surface area of a server's configuration.

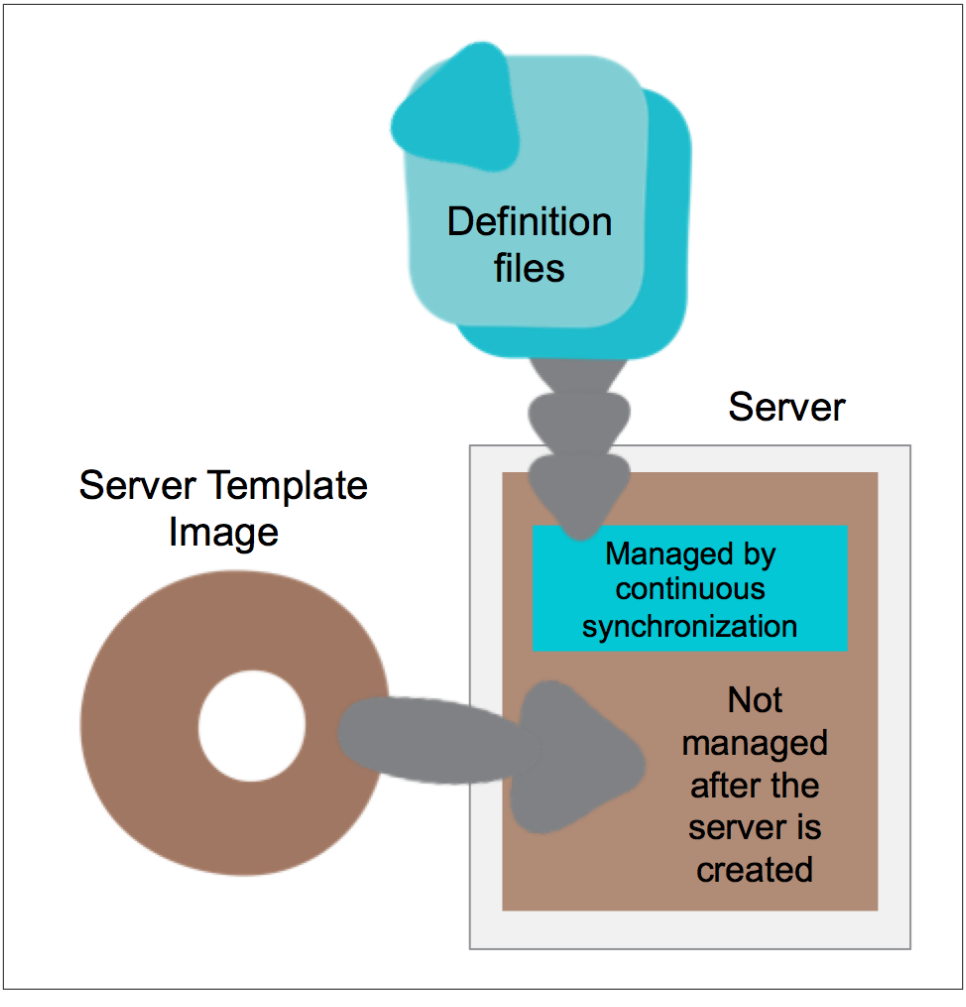


Figure 7-4. Sources of things on a server

The majority of a server's configuration actually comes from the server template - AMI, OS installation disc, etc. - used to create the server. Once the server is up and running, this is essentially unmanaged. This is often not a problem. The fact that we haven't written a configuration definition to manage a particular part of the server probably means we're happy with what's on the server template, and that we don't need to make changes to it.

These unmanaged areas do leave the door open to configuration drift. If someone finds the need to change the configuration in an unmanaged part of a server, they may be tempted to make it manually rather than going through the hassle of writing a configuration definition. This then means that particular change isn't captured in a way that

ensures it will always be made where it's needed. It can also create a snowflake server, where some application only works on the server that someone tweaked, but nobody remembers why.

Why unmanaged areas are unavoidable

An obvious solution to this issue is to make sure we have configuration definitions that cover everything on the system. But this simply isn't realistic with the current generation of configuration tools. There are two major obstacles. One is that there are too many things on a server than can be realistically managed with current tools. The other is that current tools can't prevent unwanted things from being added to a system.

Firstly, consider how much stuff would need to be managed. As discussed in the sidebar "How many things are on a Linux server" on page 108, even using a pared down Linux server image leaves a huge number of files. Writing and maintaining configuration definitions to explicitly manage every one of these is a ridiculous amount of work.

How many things are on a Linux server

I built some fresh VMs to see how many files are on a typical server, before adding applications and services. I did this with CentOS 6.6, using the "CentOS-6.6-x86_64-minimal.iso", and then with CoreOS version 607.0.0, to see how a JEOS distribution compares.

Running `find /etc -type f | wc -l` shows the configuration files on the server. My CentOS VM had 712 files, while CoreOS had only 58.

I then ran a similar find command across root folders that tend to include files managed by configuration definitions and/or packages, but which don't normally have transient or data files. The folders I included in this were: `/bin`, `/etc`, `/lib`, `/lib64`, `/opt`, `/sbin`, `/selinux`, and `/usr`.

My CentOS VM had 38,496 files, and my CoreOS VM had 4,891.

So while a JEOS server can dramatically reduce the number of files that might need to be managed, there are still quite a few.

Even if you declared everything that exists on a given server, maybe by taking a snapshot of it, there are still the things that we *don't* want on it. This is a literally infinite set of things we'd have to write definitions to exclude. "On web server machines, ensure there is no installation of mysql, no oracle, no websphere application server, ….".

We could handle this with a white-listing approach, deleting everything on the system that isn't explicitly declared. But there are large parts of the system which have transient files and data files that are necessary for the server to work properly, but can't be pre-

dicted ahead of time. We would need to make sure that data files, logs, temp files, etc. are all protected. Our current server configuration tools simply don't support this.

This isn't to say that this can't or won't be made much easier than it is with current tools. And most teams get along just fine without an obsessive level of control over every element of their servers.

Patterns and practices for immutable servers

Once an immutable server is created from its server template, its configuration files, system packages, and software are not modified on the running server instance. Any changes to these things are made by creating a new server instance. However, servers are vulnerable to configuration drift if left running for long periods, so teams using immutable servers tend to keep their lifespan short, following the phoenix pattern described earlier in this chapter.

Teams using immutable servers need more mature server template management, because of the need to build new templates quickly and often. The practices described in Chapter 6 are important to help with this.

Of course, immutable servers aren't truly immutable. Aside from transitory runtime state, there is often a need to maintain application and system data, even as servers come and go. (to come) explores ways to manage this kind of data.

Server image as artifact

One of the key drivers for adopting immutable servers is strong testing. Once a server template is created and tested, there is a smaller chance for untested changes and variation to happen on a production server. Runtime state and data will change, but system libraries, configuration files, and other server configuration won't be updated or revised at runtime, as can happen with continuous synchronization. Change management pipelines for server templates (as described in (to come)) can help to ensure the quality of a template.

Simplifying configuration management tooling with immutable servers

Many teams which embrace the immutable model find that they can simplify the tooling they use to configure servers when preparing a server template. Ansible, Chef, Puppet, and similar tools are designed to be run continuously, which adds some complexity to their design and use. They need to be able to apply configuration definitions to servers which may be in different states when the tools run, including running repeatedly when the configuration has already been applied.

But configuration which is always done to a server template can make assumptions about the starting state of the server when they run. For example, a new template may always be built using a base OS installation image. It's common for teams to use a series of fairly simple shell scripts to configure server templates, which reduces their tooling and maintenance overhead.

Immutable server flow

These diagrams show the typical flow for making a change with immutable servers. The existing servers have been built from the server template image generated from the first version of the server template definition file, e.g. a packer template file.

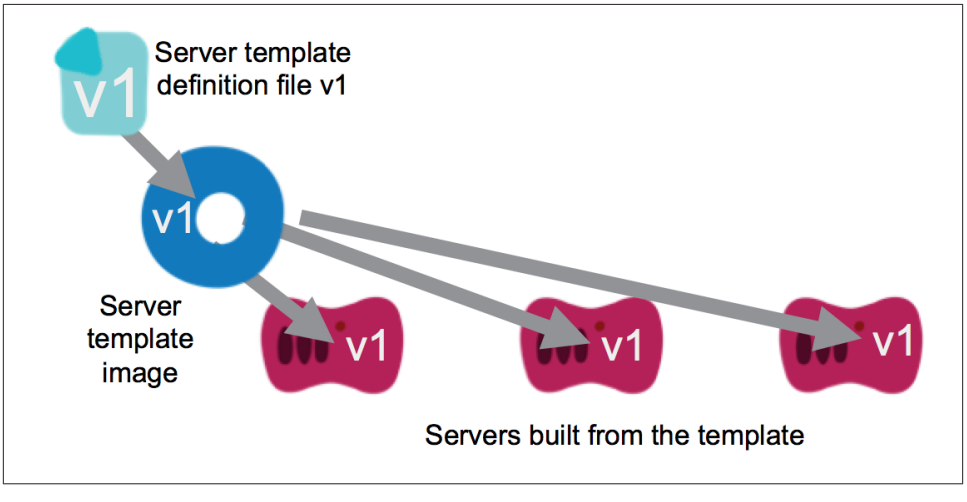


Figure 7-5. Immutable server flow - part 1

A change is needed, so the template definition file is changed, and then used to package a new version of the server template image. This image is then used to build new servers to replace the existing servers.

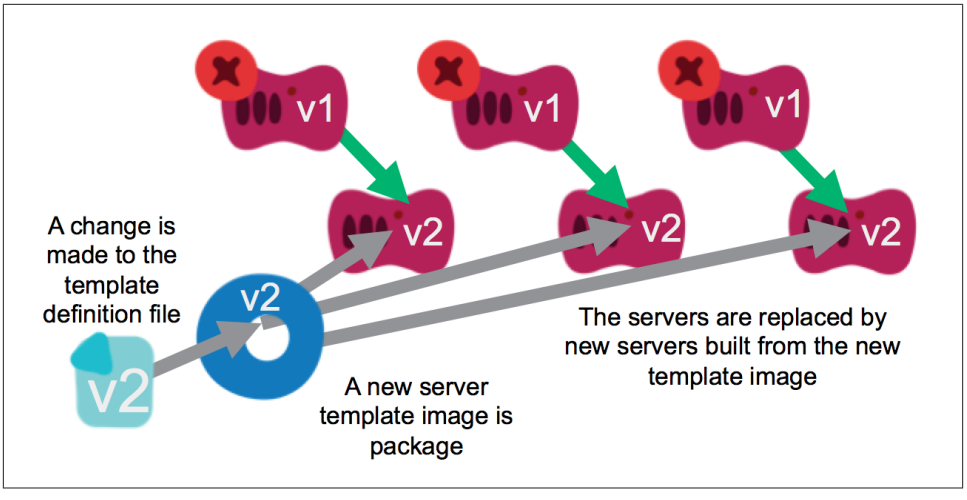


Figure 7-6. Immutable server flow - part 2

The process for replacing a running server with a new one will ideally be done without interrupting services provided by the server. Patterns for doing this are discussed in (to come) ((to come).)

Creation-time provisioning with immutable servers

The early practices for immutable servers involved putting everything onto a server template. But some teams have found that for certain types of changes, the time needed to build a new template makes the turnaround time too long. An emerging practice is to put nearly everything into the server template, but to have a few things that can be done when the server is created.

Setting configuration options is an obvious use for this. Most dynamic infrastructure platforms allow parameters to be passed to a server when it is created, which a tool like cloud-init can use to set options. This can be helpful to pass configuration for a specific instance of a server type. For example, a server may be passed an environment id and the endpoint of a configuration registry, which applications on the server can then use to look up relevant configuration settings.

But even more complex things can be done at creation time, such as provisioning software. For example, many teams provision microservices in this way. They create a server template configured to run a generic microservice application, assuming the application follows conventions. At boot time, the new server is passed the URL for an application artifact. A cloud-init script runs to download and run the artifact.

Example of a team's microservice application standards

A number of teams I've worked with have used the pattern of standardized application deployments. They typically develop multiple microservice applications, and want to be able to add and change these applications without needing to change the infrastructure. They do this by having conventions around how an application is packaged, deployed, and managed by the infrastructure. The infrastructure can then deploy and run any application developed according to these standards.

These are some of the standards agreed by one team:

Applications are packaged as a tarball (.tar.gz file), including certain files at root level: - A jar file named SERVICEID-VERSION.jar, where VERSION follows a format matching the regular expression (\d+)\.(\d+)\.(d+) - A configuration file named SERVICEID-config.yml - A shell script named SERVICEID-setup.sh

To deploy the application, the server provisioning script will download the artifact from a URI passed in the environment variable SERVICE_ARTEFACT_URI. It will then extract the contents of the artifact to the directory /app/SERVICEID, which will be owned by a user account named SERVICEID. It will also create a file in this same directory named SERVICEID-env.yml, which contains some configuration information specific to the environment.

The provisioning script also runs the SERVICEID-setup.sh script, which can carry out any initialization activities needed before the application is started. Finally, the provisioning script creates and enabled a service startup script as /etc/init.d/SERVICENAME, which starts the application by running the startup command "java -jar /app/SERVICEID/SERVICEID-VERSION.jar server".

Developers can build an application based on these standards, and know that it will work seamlessly with the existing infrastructure and tooling.

This still follows the immutable server pattern, in that any change to the server's configuration (such as a new version of the microservice) is carried out by building a new server instance. It shortens the turnaround time for changes to a microservice, because it doesn't require building a new server template.

However, this practice arguably weakens the testing benefits from the "pure" immutable model. Ideally, a given combination of server template version + microservice version will have been tested through each stage of a change management pipeline as described in (to come). But there is some risk that the process of installing a microservice, or making other changes, when creating a server will behave slightly differently when done for different servers. This could cause unexpected behavior.

So this practice trades off some of the consistency benefits of baking everything into a template and using it unchanged in every instance, in order to speed up turnaround

times for changes made in this way. In many cases, such as those involving frequent changes, this tradeoff works quite well.

Transactional server updates

Another approach which is has potential is transaction server updates. This involves package management tooling which supports updating all of the packages on a server to a defined set of versions as an atomic transaction. If any of the updates fail, the entire set of updates is rolled back. This ensures that the server's configuration, at least as far as system packages, don't end up in an untested state. The server will either be updated to the new, completely tested state, or reverted to the existing state, which was presumably tested previously.

(Note/TODO: there is a specific Linux distro designed to do this, which I need to reference)

This approach has a lot in common with immutable servers, and also with containers.

Practices for managing configuration definitions

Teams moving to infrastructure as code, especially those using continuous synchronization, find that working with configuration definitions for their toolchain becomes a core activity. Chapter 8 explains a number of software development practices that are useful for infrastructure developers. This section discusses a few practices specifically useful for managing configuration definitions for configuring servers.

Practice: Keep configuration definitions minimal

Code, including configuration definitions, has overhead. The amount of time your team spends writing, maintaining, testing, updating, debugging, and fixing code grows as the amount of code grows. This overhead grows exponentially, since each piece of code has interactions with at least some other code. These interactions may be deliberate - one piece of code may depend on another - or unintentional - two pieces of code happen to affect common parts of the system in a way that can create conflicts.

Teams should strive to minimize the size of their codebase. This doesn't mean writing code that uses the least number of lines as possible, an approach which can actually make code harder to maintain. Rather, it means avoiding the need to write code whenever possible. As discussed below, most of what is configured on a server is undeclared - it comes from the server template used to create a server, rather than being explicitly defined in a configuration definition.

This is a good thing. As a rule, an infrastructure team should only write and maintain configuration definitions to cover areas that really require explicit, continuous config-

uration. This will include parts of the system that need to change quite often, and those which most need to be protected from unwanted, unmanaged changes.

A useful approach to implementing automated configuration is to start small, writing definitions for the things we need most. Over time you'll discover changes that are needed, so you can add definitions for those. You should also keep a constant eye out for definitions that are no longer needed and so can be pruned.

Practice: Organizing definitions

Configuration definitions, as with any code, seems to prefer becoming a large, complicated mess over time. It takes discipline and good habits to avoid this. Once an infrastructure team gets a handle on the basics of writing and using a server configuration tool, they should invest time in learning how to use the tool's features for modularization and re-use.

The goal is to keep individual modules or definition files small, simple, and easy to understand. At the same time, the overall organization needs to be clear, so anyone with a reasonable understanding of the tool can quickly find and understand the sections of definitions which are involved in a given part of the system. Chapter 8 gives more ideas about how to do this.

Practice: Use Test Driven Development (TDD) to drive good design

A common misunderstanding of TDD is that it's just a way to make sure automated tests get written as code is developed. But developers and teams who become effective at TDD find that it's main value is in driving clean, maintainable code design.

The way this works with configuration definitions is ensuring that every module has a set of unit tests which can test that module in isolation. If the tests are written and maintained with good discipline, they make it clear when a module is becoming too large or complicated, because the tests become more difficult to update and to keep working as the definitions are updated. Tests also reveal when a module is too tightly coupled to its dependencies, because this makes it difficult to set up self-contained tests.

Chapter 8 and Chapter 9 go into the software engineering and test automation practices behind this.

Conclusion

This part of the book has described patterns and practices for managing servers, including provisioning, creating, and updating servers, and managing templates. The next part will go into how infrastructure teams can take advantage of software engineering practices, including development workflows, testing, and change management pipelines.

CHAPTER 8

Software engineering practices for infrastructure

Part III of this book focuses on practices from software engineering that can be brought to bear on infrastructure that is itself treated like a software system.

This chapter discusses a few of the basic practices, including the use of version control, Continuous Integration (CI), and managing code quality. The next chapter (Chapter 9) digs into one of the core aspects of software engineering, testing. (to come) explains how to design and implement change management pipelines to automate the testing, integration, and rollout of changes to an infrastructure. This part of the book then closes with (to come), which describes the workflow of an engineer managing infrastructure using these practices.

Developers who are experienced with agile and XP practices like TDD and CI may find the contents of this chapter familiar. There are a few tips that specifically call out differences for infrastructure development, so even if you skim this chapter, you might find these worth noting.

Good software engineering practices produce high quality code and systems. Quality is not just about correctness, it's also about ensuring changes can be made easily and often. People maintaining a system that can be changed quickly and safely can build habits of routinely making fixes and improvements without ceremony, which in turn helps keep the quality up, in a virtuous cycle.

A cornerstone of maintaining high quality systems is fast feedback on changes. When we make a mistake in a change to a configuration definition, we'd like to find out about that mistake sooner rather than later. Ideally we'd like to find out before we apply it to an important system.

When we make a mistake while working on a large project, for example implementing a new service, we'd like to catch it before we do more work on the project. If we don't find the mistake until later, for example, because we don't do comprehensive testing until the project is nearly finished, its impact is much worse. First, it's harder to discover the mistake when it's buried in several weeks or months worth of other changes. It's harder to debug the mistake and uncover the actual problem. We've lost the context we had when we made the original change, because we've been doing so many other things in the meantime. And the mistake is liable to be intertwined with loads of other changes, making it difficult to disentangle and fix.

So the practices described in this chapter are intended to give us fast feedback when working with infrastructure as code.

Continuous Integration (CI)

Continuous Integration is the practice of frequently integrating and testing all changes to a system as they are being developed. This is different from the traditional software development approach of having an integration phase towards the end of the project.

With the integration phase approach, different people and/or teams work on their piece of the project in isolation. The pieces are only brought together and thoroughly tested once they're finished. The idea behind this is that the interfaces between the pieces can be defined well enough that there should be no problem putting them together. This idea is proven wrong again and again[1].

The problem with late integration is that you discover problems late in the process, usually shortly before users are expecting the system or changes to be ready to use. When this happens, you're forced to choose between slipping the release date, or else slapping in a workaround so you can push the thing out the door. Most often we do both of these things, telling ourselves that we'll fix it properly later. Unfortunately, most IT teams have a large backlog of work, and find it difficult to make time to properly fix things.

The integration phase is often cited as "best practice" for IT projects. Its widespread use is reflected in the current state of the art in IT: buggy systems that take up far too much of our time in firefighting and workarounds. Not to mention the prevalence of IT projects that are delivered behind schedule and over budget. Continuous Integration and Continuous Delivery offer a compelling alternative. While they don't guarantee perfect projects, if used correctly they are more effective than an integration phase.

1. The lego integration game is a fun way to demonstrate the differences between late integration and continuous integration.

With Continuous Integration, engineers frequently commit small changes. It's not unusual for people to commit several changes a day, or even a change or two every hour. Every time someone commits a change - to a configuration definition, script, test, or whatever - the CI server pulls the change and runs tests. Ideally this takes less than a minute, or a few minutes at worst.

Who broke the build?

A test failing in CI triggers a "stop the line" situation. Development teams call this a *broken build*, or a *red build*.

Nobody else in the team should commit any changes until the error is fixed, to make it easier to resolve. When commits are added to a broken build, it can be difficult to debug the original cause and untangle it from the following changes.

The person whose commit triggered the failure should prioritize fixing it. If the change isn't fixed quickly, then it should be reverted in VCS, so they CI job can run again and presumable return to a green status.

Build monitors

Teams need immediate visibility of the status of their CI jobs. It's common to use an information radiator (see "What is an information radiator?" on page 56) in the team's work area, and desktop tools that pop up notifications when a build fails. Email isn't very effective for this, since they tend to get filtered and ignored. A broken build needs to be a "red alert" which stops everyone until they know who is working to fix it.

Fixing broken builds immediately is critical for effective CI. Don't fall into the habit of ignoring failing tests. Once this happens, and the build is always red, the value you get from CI is zero. You might as well turn off your CI completely and save the cost of running it.

A related bad habit is disabling or commenting out tests that fail, "for now", just to get things moving. Fixing a failing test should be your immediate priority. Getting a particular change released may be important, but your test suite is what gives you the ability to release effectively. If you allow your tests to degrade, you are undermining your ability to deliver.

Stamp out flaky tests - tests which routinely fail once, then pass when run another time. Randomly failing tests condition the team to ignore red builds - "it's always red, just run it again". Legitimate problems aren't noticed right away, after which you need to sift through the various commits to figure out where the real failure came from.

Flaky builds are a symptom of problems with your system or your tests. Be merciless about tracking down the cause of these failures, and improving your testing techniques to eliminate flakiness. The next chapter has advice on ways to do this.

TDD and self-testing code are essential for CI. Chapter 9 discusses using these practices to develop tests alongside the code and configuration definitions that they test. Together, TDD and CI are a safety net that makes sure you catch mistakes as you make them, rather than being bitten by them weeks later.

Continuous Delivery (CD)

Continuous Delivery takes CI a step further. Whereas CI integrates and tests every commit, CD ensures the entire system is production ready on every commit. Changes are applied to a production-like environment, using the same tooling and processes that will be used to apply them to production. When this is done continuously while working on changes, then rolling them out to production becomes a trivial exercise.

The next few chapters discuss implementation details for testing - automated and manual - as well as for pipelines. A pipeline can be useful to apply tests with increasing levels of complexity. The earlier stages focuses on faster and simpler tests, while later stages replicate more of the constraints and complexities of production.

Production-like environment doesn't mean identical to production. The important point is that the test environment emulates the key aspects of the production environment, including the various complications and complexities. Basically, anything that might go wrong when deploying to production should be emulated sufficiently so that it will go wrong in a test environment.

Test environments should be locked down just like production. Allowing privileges in test that aren't allowed in production only paves the road to failed production deployments. If production constraints make it hard to make changes quickly and often in test, then find solutions that allow you to deploy quickly and often within those constraints.

At the bank I mentioned in my story ("The twelve-month Continuous Delivery project" on page 62), deploying to production required a sensitive password to be manually entered. We created a mechanism so someone could enter the password while triggering the deployment job in the Jenkins console. We automated this process for test stages where the password wasn't needed, but the script and job ran the exact same scripts. This allowed us to emulate the production constraint with a consistent, repeatable process.

This story illustrates that manual approvals can be incorporated into a CD process. It's entirely possible to accommodate governance processes that require human involvement in making changes to sensitive infrastructure. The key is to ensure that the human involvement is limited to reviewing and triggering a change to be applied. The process

that actually applies the change should be automated, and should be exactly the same for each environment it is applied to.

Continuous Delivery versus continuous deployment

One misconception about CD is that it means every change committed is applied to production immediately after passing automated tests. While some organizations using Continuous Delivery do take this continuous deployment approach, most don't. The point of CD is not to apply every change to production immediately, but to ensure that every change *could* be applied.

CD makes the decision of whether and when to apply the change to production a business decision, rather than a technical one. The act of rolling out a change to production is not a disruptive event. It doesn't require the team to stop development work. It doesn't need a project plan, handover documentation, or a maintenance window. It just happens, repeating a process that has been carried out and proven multiple times in testing environments.

For IT Ops teams, Continuous Delivery means that changes to infrastructure are comprehensively validated as they are made. Changes needed by users, such as adding new servers to a production environment, can be made without involvement by the IT Ops team, because they already know exactly what will happen when somebody clicks the button to add a web server to the web server pool.

VCS for infrastructure management

Chapter 1 details the reasons why a Version Control System (VCS) is an essential part of an infrastructure as code regime. Chapter 2 goes into more detail of how to integrate a VCS with configuration management tooling. This section will discuss some software engineering practices around using a VCS.

What to manage in a VCS

Even for developers experienced in using VCS, it's useful to step back and consider what kinds of things an infrastructure team should manage in a VCS.

Put everything in version control that is needed to build and rebuild elements of your infrastructure. Ideally, if your entire infrastructure were to disappear other than the

contents of version control, you could check everything out and run a few commands to rebuild everything, probably pulling in backup data files as needed.

An incomplete list of things to version:

- Scripts and source code for compiled utilities and applications
- Configuration files and templates
- Configuration definitions (Cookbooks, Manifests, Playbooks, etc.)
- Tests

Things that might not need to managed in the VCS include the following. Some of these reference "Types of things on a server" on page 72 in Chapter 4:

- Software artifacts should be stored in a repository, for example a Maven repository for Java artifacts, and apt or yum repository, etc. These repositories should be backed up or have scripts (in VCS) which can rebuild them.
- Software and other artifacts that are built from elements already in the VCS don't need to be added to the VCS themselves, since they can be rebuilt from source.
- Data managed by applications, logfiles, etc. don't belong in VCS. They should be stored, archived, and/or backed up as relevant. (to come) and (to come) cover this in detail.
- Passwords and other security secrets should not be stored in a VCS. There are tools for managing encrypted keys and secrets in an automated infrastructure which should be used instead.

Branching versus CI and CD

Version Control Systems support branching, which allows people to work on different versions of the same code at the same time. This is a very common practice in software development, but is problematic for CI and CD.

Infrastructure teams adopting VCS, especially those working in organizations which use branching in software development, may be tempted to adopt similar branching strategies. This section explains those strategies, and why they're problematic even for software teams trying to use CI and CD.

The reason branching is problematic is because it's a variation of late integration, whose problems are discussed earlier in this chapter. In fact, it adds even more risk because people, and sometimes even teams, are working on the same code. Integrating branches is called merging, and involves working out what areas of the code have been modified by more than one person or team. These are called merge conflicts, and are resolved by manually deciding which changes should be accepted and used in the merged code.

Modern VCS tools like git handle the technical aspects of merging more easily than slightly older tools like Subversion. However, they don't magically fix the logical issues of merging. Even when a merge doesn't include changes to the same lines of code, there is still a risk that the changes may have functional interactions which cause issues.

This isn't to say branching is evil or stupid. Many teams find it an effective way to manage complex software projects, and some, highly mature teams, use very short-lived branches effectively with CI. However, keeping branches for longer than a day or so does conflict with Continuous Integration which, by definition, merges and integrates all code continuously, as it's developed.

The bottom line is, merging branches is a form of integration. The longer you go without merging or integrating code being worked on separately, the higher the chance of an issue. This in turn means more testing, fixing, and re-work.

Branching strategies

There are a few common ways of using branches on software development projects.

Table 8-1. Common branching strategies

Branching Strategy	Description
Feature branching	When a person or small group starts working on a change to the codebase, they can take a branch, so they can work on it in isolation. This way, their work in progress is less likely to break something in production. When the change is complete, the team merges their branch back into trunk.
Release branching	When a new release is deployed to production, a branch is taken to reflect the current version in production. Bugfixes are made on the release branch and merged to trunk. Work on the next release is done on trunk.
Trunk based development	All work is done and committed to trunk. Continuous Integration is used to ensure every commit is fully tested, and a Continuous Delivery Pipeline can be used to ensure changes are only applied to production after being fully validated.

Release branching can be useful for a codebase that has longer development cycles and infrequent releases, because it makes it easier to make emergency fixes to production between releases. It is less applicable to infrastructure teams, which normally make a constant stream of small changes.

Creating a branch may seem appealing for a major infrastructure change which will be disruptive to daily operations. However, doing this only accumulates risk. As more and more work is done in the special branch, the amount of work that will be needed to roll it out also grows. Releasing the big new changes, even after they've been finished on their branch, becomes a major project of its own, with all of the work and risk that comes with a big-bang release.

Approaches for managing major changes are discussed later in this chapter.

Some teams use CI server software with feature branching. When changes are committed to a branch, the CI server builds and tests that branch, so the developers get the benefit of having their code continuously tested. However, they aren't actually integrating continuously, so they will only find out about a conflicting change in someone else's branch after the change is merged.

Teams can take a step towards mitigating this by continuously merging trunk into their feature branches before testing it in CI. This way, if one developer merges a conflicting change into trunk, the other developer will discover the conflict the next time they build and test their branch. However, this discovery still happens later than it would have if the changes were being made directly to trunk rather than on branches.

Trunk based development is the most effective strategy for discovering conflicts between work being done by different developers quickly.

Good coding practices

Over time, the infrastructure codebase grows and can become difficult to keep well-maintained. The same thing happened with software code, and so many of the same principles and practices can be used to make maintaining large infrastructure codebases.

Practice: Clean code

In the past few years there has been a renewed focus on "clean code"[2] and software craftsmanship, which is as relevant to infrastructure coders as to software developers. Many people see a tension between pragmatism - getting things done - and engineering quality - building things right. This is a false dichotomy.

Poor quality software, and infrastructure, is difficult to maintain and improve. Choosing to knock something up quickly, knowing it probably has problems, leads to an unstable system, where problems are difficult to find and fix. Adding or improving functionality on a spaghetti-code system is also hard, typically taking surprisingly long to make what should be a simple change, and creating more errors and instability.

Craftsmanship is about making sure that what you build works right, ensuring loose ends are not left hanging. It means building systems that another professional can quickly and easily understand. When you make a change to a cleanly built system, you are confident that you understand what parts of the system the change will affect.

Clean code and software craftsmanship are not an excuse for over-engineering. The point is not to make things orderly to satisfy our compulsive need for structure. We don't need to built a system that can handle every conceivable future scenario or requirement.

2. See Robert "Uncle Bob" Martin's Clean Code: A Handbook of Agile Software Craftsmanship

Much the opposite. The key to a well-engineered system is simplicity. Build only what you need, then it becomes easier to make sure what you have built is correct. Reorganize code when doing so clearly adds value, for instance when it makes the work you're currently doing easier and safer. Fix "broken windows" when you find them.

Practice: Manage technical debt

Technical debt is a metaphor for problems in our system which we leave unfixed. As with most financial debts, your system charges interest for technical debt. You might have to pay interest in the form of ongoing manual workarounds needed to keep things running. You may pay it as extra time taken to make changes which would be simpler with a cleaner architecture. Or charges may take the form of unreliable or hard to use services for your users.

Software craftsmanship is largely about avoiding technical debt. Make a habit of fixing problems and flaws as you discover them, which is preferably as you make them, rather than falling into the bad habit of thinking "it's good enough for now".

This is a controversial view. Some people dislike technical debt as a metaphor for poorly implemented systems, because it implies a deliberate, responsible decision, like borrowing money to start a business. But it's worth considering that there a different types of debt. Implementing something badly is like taking a payday loan to pay for a vacation - it runs a serious risk of bankrupting you.

Martin Fowler talks about the Technical Debt Quadrant, which distinguishes between deliberate versus inadvertent technical debt, and reckless versus prudent technical debt.

Managing major infrastructure changes

The engineering practices recommended in this book are based on making one small change at a time (see "Principle: Small changes" on page 17). This can be challenging when delivering large, potentially disruptive changes.

For example, how do we completely replace a key piece of infrastructure like a user directory service? It may take weeks or even months to get the new service working and tested. Swapping the old service out for a new one that isn't working properly would cause serious problems for our users and for us.

The key to delivering complex work in an agile way is to break it down into small changes. Each change should be potentially useful, at least enough that someone can try it out and see an effect, even if it's not ready for production use.

I find it useful to think in terms of capabilities. Rather than defining a task like "implement a monitoring check for ssh", I try to define it in terms such as "make sure we'll be notified when sshd is not available on a server". For larger projects, a team can define progress in terms of capabilities.

There are a number of techniques for incrementally building major changes to a production system. One is to make small, non-disruptive changes. Slowly replace the old functionality, bit by bit. For example, in Chapter 7 we discussed implementing automated server configuration incrementally. Choose one element of your servers and write a manifest (or recipe, playbook, etc.) to manage that. Over time, you can add further configuration elements piece by piece.

Another technique is to keep changes hidden from users. In the example above of replacing a user directory service, you can start building the new service and deploy it to production, but keep the old service running as well. You can test services which depend on it selectively. Define a server role that uses the new user directory, and create some of these servers in production that won't be used for critical services, but which can be tested. Select some candidate services which can be migrated to the new directory at first, perhaps ones that are used by the infrastructure team but not by end-users.

The important point is to make sure that any change which will take a while to implement is continuously being tested out, during development.

Conclusion

The focus of this chapter was on techniques for making changes to systems easily and often, so that infrastructure can be continuously, safely, and easily improved. The next chapter moves on to testing practices.

CHAPTER 9
Testing Infrastructure Changes

The purpose of testing is to help us to get our work done quickly. Sadly, in many organizations testing is seen as something that slows work down. This is usually due to treating QA as a separate function from the work of building software or managing infrastructure, rather than integrating it into the work being done.

As with infrastructure management, automating a broken testing process doesn't make it work any better. But automated testing does make it easier for teams to take real ownership of the quality of what they build. They can continuously test what they're working on, which gives them confidence that their work is solid.

This confidence enables a sustainable, fast pace of work that isn't possible with a system that has undiscovered and unresolved problems.

I routinely see QA teams struggling to use tools to automate their testing phase. They aren't able to write enough test scripts to cover as much of the system as they would like. The tests they do have are brittle - each new round of testing takes too much time in fixing and updating the test scripts. They had hoped that automated testing tools would make their job faster and easier, but they just gave them more work. Quite often, the testers fall back to their lists of manual tests just to keep things flowing.

Other teams embed automated testing into their normal development and maintenance processes. The automated tests run continuously, throughout the day, as a part of CI and CD pipelines. They aren't seen as the domain of a separate team, but rather as the responsibility of the people who write code and maintain infrastructure.

However, even some of these teams find it difficult. They aim to have tests cover 100% of their codebase, but maintaining the tests takes over. Every change to the system triggers a cascade of tests that need to be tweaked and massaged in order to get the build green again. Tests may fail intermittently, either because their system integrates with outside systems which are unreliable, or because of non-deterministic behavior in their own system.

But many teams make automated testing work well for them. They may occasionally spend time making improvements to the way the tools work, but they don't get in their way. They tend to use several different testing tools for different purposes, which helps them to keep a balance of tests. They keep the size of their test suite from growing too large. The test suite runs quickly, so people don't spend much time waiting for it to run through. Making a change may require a few tests to be added or updated, but it's not a significant effort.

This chapter aims to help you reach this state.

Sysadmins versus testing

Infrastructure teams tend to find testing a challenge. The typical systems administrator's QA process is: 1) make a change, 2) do some ad-hoc testing (if there's time), 3) keep an eye on it for a little while afterwards. Not many QAs understand infrastructure very well, so it tends to slip under the radar.

Change Advisory Boards (CAB)

Organizations which want a more rigorous process around infrastructure often set up a CAB (Change Advisory Board). This handles the problem of testing infrastructure by having a meeting to review changes before they're made. This can have some value. The person doing the change consults with colleagues, who may have useful experience and advice to offer. Potential conflicts with other changes are raised. People are made aware of the change, so if someone notices an issue they know who to talk to about it.

But too often, a CAB can turn into a bureaucratic exercise. People may be forced to spend time on activities that don't add real value - filling out detailed forms, getting rubber-stamp signoffs, etc. In some cases, the CAB will reject a change request because the boxes haven't been ticked, forcing the person needing the change to wait until the next CAB meeting, which may be a week or several weeks later.

If a CAB process becomes burdensome, then people will find ways to get their work done in spite of it. They may learn how to bypass or shortcut it, for example by turning everything into an emergency fix. In many organizations, people copy and paste change request forms without worrying about the details in it, knowing that the people signing off on them don't understand the details. Or people may just ignore the CAB process entirely.

Improving on the CAB approach

It's important to consider what the value is in the process, and find the most effective ways to achieve that value.

With a CAB, the main goals are typically:

- Ensure planned changes are reviewed, rather than made by someone on their own
- Avoid conflicts with other changes taking place
- Make people aware of the change, in case it impacts them
- Plan for any supporting activities, such as notifying users

There are a number of practices that can help achieve these without becoming a bottleneck. These are just some examples:

- Automatically test changes before they're applied to an important system
- Having a pair of people work on a change together
- Notify people within the relevant technical teams about planned changes, for example on an email list, so they can ask questions and make suggestions
- Sent commit/diff messages to the technical team when changes are committed to VCS
- Require a code review and signoff for highly sensitive parts of the system before a change can be applied to production systems

Whatever practices your team decides to use, there are some principles which can help to keep things effective.

Principle: Prefer to review a working implementation rather than an idea

It's better to review a change that's been implemented and tested, rather than when it's an unproven idea. This can be done safely with a good system for testing and staging changes before applying them to production, for example a change management pipeline (as in (to come)). The implementation can always be fixed, improved, or discarded if necessary.

Principle: There are diminishing returns from adding approvers

Having a second pair of eyes is invaluable. Inviting a broad group to review and comment on an idea or implementation is a great way to bring different viewpoints, experience, and skills into it. But the more people who need to approve a change, the more difficult the process becomes, without much added value.

Principle: Focus on the ability to make changes over the ability to stop them

As mentioned in Chapter 1 ("What good looks like" on page 21), your team should aim to be very good at catching problems and making fixes quickly - MTTR (Mean Time

To Recover). Too many teams believe they should (and can) avoid ever making a mistake. But while this book strongly encourages a strong culture of quality and effective continuous testing processes, it's important to get the cost/benefit balance right. The benefit of a fast time to market may outweigh a risk that has a fairly low cost, especially when the risk can be detected and correctly very quickly.

Principle: Small changes reduce risk

The principle of small changes, as mentioned in Chapter 1 and many times since, supports these other principles. A small, incremental change that demonstrates a larger solution is cheap to put together for discussion and review. A small change has less risk of breaking something, and in any case the risk is simpler to assess. If it does cause a problem, it's easier to track down and fix. This then makes it easier to relax and allow changes to go through with a more streamlined process.

Owning tests

One of the big wins of agile software development was breaking down the silos between developers and QA. Rather than declaring quality to be owned by a separate team, developers and testers share this responsibility. Rather than allocating a large block of time to test the system when it's almost done, agile teams start testing when they start coding.

All validation should be done continuously during development, so there's little or nothing left to do at the end. This includes testing for correctness, as well as exploratory testing, user testing, performance, security, availability, regression, and anything else needed for a given system.

There is still controversy over what the role of a QA engineer or tester should be, even within an agile team. Some teams have decided that, since developers write their own automated tests, there is no need for a separate role. Personally, I find that even in a highly functioning team, QAs bring a valuable perspective, expertise, and a talent for discovering the gaps and holes in what I build.

There are some guidelines for how to managing testing with a team.

People should write tests for what they build

Having a separate person or team write automated tests has several negative effects. The first one is that delivering tested work takes longer. There is delay in handing over the code, then a loop while the tester works out what the code should do, writes the tests, and then tries to understand whether test failures are because they've gotten the test wrong, because of an error in the code, or a problem with the way the work was defined.

If the developer has moved on to another task, this creates a constant stream of interruptions.

If a team does have people who specialize in writing automated tests, they should work with the developers to write the tests. They could pair with the developer for the testing phase of work, so the developer doesn't move on to something else but stays with the work until the tests are written and the code is right. But it's better still if they pair during development, writing and running the tests while writing the code.

The goal should be to help the developers become self-sufficient, able to write automated tests themselves. The specialist may carry on working with the team, reviewing test code, helping to find and improve testing tools, and generally championing good practice. Or they may move on to help other teams.

Everyone should have access to the testing tools

Some automated testing tools have expensive per-user licensing models. This leads to only a few people able to run the tests, because paying for licenses for everyone on the team is just too expensive. The problem with this is that it forces longer feedback loops.

An engineer builds something and hands it over. The tester runs the tests, and reports a failure back to the engineer. The engineer must reproduce and fix the error blindly, without being able to run the failing test themselves. This leads to a few rounds of "fixed it", "still fails", "how about now", "that broke something else", and so on.

There are many powerful test automation tools, with large communities of free and commercial support, which are either free or cheap enough to be used without adding waste and pain to the delivery process.

Defining work for testability

One of the most useful things that QAs have brought to projects I've worked on is helping us to define our work more clearly. Most sysadmins I've worked with share my tendency to jump into implementing something. We think we know enough about what we need to do, and that we can figure the rest out by building something.

While this may work for learning and trying things out, for a concrete task it's helpful to be clear on what outcome you're looking for, why, and how to to know when you've done it.

Story sentence

Agile development has the concept of a "User Story"[1]. This is a small, concrete piece of work that can be clearly defined in terms of its value and outcome. There is a popular format for a "story sentence" to communicate this:

As a (*USER*), I want to (*CAPABILITY*), So that (*VALUE*).

Infrastructure teams who aren't used to this often struggle to define stories in a way that really captures these. Having help a Business Analyst (BA) or QA with agile experience can be helpful to learn the habits and techniques of defining work this way.

It's important to really get to the capability and value here. Some good and bad examples of infrastructure stories are below:

Table 9-1. Examples of infrastructure story sentences

Bad	Good
As the infrastructure architect, I want the nagios memory check to be added, so that an alert will be sent	As a support team member, I want to be alerted when a server is about to run out of memory, so that I can prevent an outage
As a security auditor, I want VMs to be secure, so the system is secure	As the owner of a VM, I want to ensure only strictly required processes are running on my VM, to reduce the attack surface of my VM

In some of these examples we see broad, potentially vague statements like "I want VMs to be secure". Forcing ourselves to be more concrete helps us to break the stories down into smaller, achievable pieces of work. A general story like "make VMs secure" can stay in progress indefinitely, because there is always more that can be done under that heading.

Acceptance criteria

Having a good story sentence helps to make the goal of a piece of work clear. Acceptance criteria are a further level of detail. These lay out ways to verify that we've achieved the goal, which helps the person doing the work to be clear on what's expected, and can help with testing.

The classic format for acceptance criteria is the "given/when/then" format.

Given (*STARTING CONDITION*), When (*ACTION OR EVENT*), Then (*ENDING CONDITION*).

Table 9-2. Examples of acceptance criteria for infrastructure stories

Given VM in any role, When the CPU usage exceeds 95% for more than 5 minutes, Then a monitoring alert is sent

1. Mike Cohn's website has a good introduction to user stories, and his book User Stories Applied is a solid reference.

Given a server role, When a new VM is provisioned with that role, Then only the processes listed in the "core processes" and process defined by the role are running

Acceptance criteria can be a starting point for writing automated tests. However, writing an automated test for every single acceptance criteria for every single story leads to an unmanageable, fragile testing suite, and often ends up with an ice-cream cone test suite, as described later in this chapter.

Story process

In most IT organizations there is a strong temptation to create complex, heavyweight processes to manage workflow. It's common even for those adopting agile methods to accumulate a series of meetings, kickoffs, reviews, showcases, and signoffs. Often, the time spent discussing a small, routine task is much more than the time spent actually doing, checking, and fixing the work.

It's helpful to boil things down to the things that add value. For a piece of work that hasn't already been defined, the activities I find most valuable are:

- Have a clear statement of the work to be done. This can be as simple as a single sentence in some cases, as long as it isn't vague or open-ended.
- Review that statement with whoever has asked for the work, the person who will do it, and someone who will validate it (tester, PM, BA, etc.). Be sure that everyone understands what is not included in the work (what's out of scope) as well as what's in scope. Be sure that everyone agrees how to validate the work will be done.
- Review work in progress, as needed, to make sure we're on the right track.
- Review the work with the stakeholder once it's done to make sure they're satisfied.

Frequent, common tasks can be defined once, and the definition re-used. For example, there's no need to go through a full story process for creating a user account. But it is useful to have the task defined, with acceptance criteria.

It's also worth differentiating routine tasks, like setting up a user account, which can be clearly defined once and re-used. Having acceptance criteria makes it easy to ensure that anyone doing the task knows what to double-check. Of course, this kind of task will ideally be automated, so that the steps are carried out in a consistent, repeatable, transparent process, with criteria automatically validated afterwards.

Writing tests

As I mentioned early, I often meet teams struggling to make automated testing a routine part of their process. Implementing a full suite of automated regression tests for an existing system is extremely challenging.

The secret truth about automated testing is that the most valuable outcome of writing automated tests is not the tests, it's the code they test. The act of writing a test forces us to question and think about the design of the system component that we're testing. This should lead us to improve the design of that component.

So, we write tests to help us to shape the stuff we build. Since writing tests helps us to design and build better systems, clearly we should be doing it while we're designing and building, rather than after we've built something.

It's very difficult to add automated tests to a system that was built without them. Good automated testing requires the ability to isolate each piece of the system and test it separately, as described later when we talk about the test pyramid. This ability requires the system components to have been designed to support this isolation, and the most effective way to ensure this is to write tests as we build the components.

The secret of successful automated testing

The secret to making automated testing an effective part of your process is to be doing it with the right goal in mind. You should see automated testing as a way to support designing, building, and improving your infrastructure, not as a way to guarantee the correctness of your infrastructure.

Test Driven Development (TDD)

Test Driven Development (TDD) is a core practice of eXtreme Programming (XP). The idea is that, since writing tests can improve the design of what we build, we should write the test before we write the code we test. The classic working rhythm is:

1. Write a test
2. Run the test, see that it fails
3. Write the code
4. Run the test, see that it passes
5. Commit the code and test
6. See that the change passes in CI

We write the test first because it forces us to think about the code we're about to write. What does it do, what does it need before it can run, what should happen afterwards? What is the interface for the functionality, what are the inputs and outputs? What errors may happen, and how do we handle them? Once we've got a clear test, writing the code to make the test pass should be easy.

But we actually run the test and make sure it fails before we've written the code. The reason for doing this is to prove that the test is useful. If the test passes before we make the code change, then it doesn't actually test the change we're planning to make.

Take it one test at a time

I once met a team which was struggling with TDD, because they were writing all of the tests for each component of the system before they started building it. The problem with this approach is similar to the problem with overly-detailed, Big Design Up Front (BDUF) specifications. As Helmuth von Moltke said, "No plan survives contact with the enemy".

If you write all the tests before starting to build the component, you'll inevitably need to change the tests once you do start, maybe radically.

This comes back to doing work a small piece at a time. Focus on one operation, write a test, implement the operation, then move on to the next.

Tests are code, too

The tooling for writing and running tests should be treated the same as everything else in the infrastructure. It should be possible to set up testing - agents, software, etc. - in a way that is repeatable, reproducible, transparent, and has all of the other qualities we expect with infrastructure as code. "Principle: Version all the things" on page 18 applies to tests.

Tests should be stored in an externalized format that can be committed to a VCS, rather than hidden inside a proprietary black-box tool. This way, a change to a test can automatically trigger a test run in CI, proving that it works.

Keep test code with the code it tests

Tests should be managed together with the code of the thing they test. This means putting them together in your VCS, and promoting them together through your pipeline until they reach the stage where they're run. This avoids a mismatch of test to code.

For example, if you write some tests to validate a Chef cookbook, they may need to change when there's a change to the cookbook. If the tests are stored separately, you may end up running an older or newer version of the tests against a different version of the cookbook. This leads to confusion and flaky builds, because it's unclear whether there is really an error in the cookbook or just a test mismatch.

Getting the TDD habit

I struggled to adopt TDD, and I know many other people have, too. I started a new project - a Ruby Sinatra application - and decided I would be a good boy and use TDD. I set up rspec and started writing tests. But before long I fell back into my wicked ways, coding and coding without bothering with the tests.

Writing tests slowed me down. I spent too much time fiddling around with the test tools, libraries, mocking, setting up data before running a test, and I struggled to keep my code decoupled enough to avoid my tests becoming messy and awkward to write.

In short, TDD was hard because I wasn't good at it.

When I started working with teams where people were used to TDD, it opened my eyes. For these people, writing tests was second nature. There's a learning curve with a dip - the valley of despair. It takes self-discipline, or working with a group that has the discipline, to force yourself to push through the pain and reach the point where you start actually working faster with TDD than you did before.

People who have got the TDD habit avoid building up messy, dodgy code with hidden problems that trip them up. They don't hesitate to clean up messy or broken bits of code they run across - if they break something, the tests will catch it. This is a virtuous cycle.

Adding tests to existing infrastructure and systems

As we've seen, adding automated tests to a system or infrastructure that was built without them is challenging. The system's design is unlikely to lend itself to layered testing. Components aren't likely to be easy to set up and test in isolation. It may be difficult to automatically manage data and integration points for test scenarios.

Launching a project with the goal of building a comprehensive test suite is unlikely to succeed. The goal of implementing automated testing should be to make test-writing a core part of the team's working habits. The outcome of such a project should be that the team routinely and comfortably writes and updates tests as they build and manage their infrastructure.

So implementing automated testing should be done as a part of the team's normal ongoing work. Each task that the team undertakes, such as adding a new service, fixing a problem, or making an improvement, should involve considering and implementing automated tests. Write tests as the new service is implemented, to validate that the fix for the problem works, and that the improvement does what it should.

These tasks will take longer than usual at first. New tooling may need to be set up, people need to learn how to write tests using the tool, and the part of the system being tested may need to be restructured to enable testing.

The system's design is likely to be the biggest obstacle to adding tests, for the reasons mentioned earlier. Quite often, a team launches automated testing as part of an overall effort to rebuild or re-platform their infrastructure. For example, these days it's common for teams to launch a major effort to migrate from a fairly traditional model of server management to an infrastructure as code model. Often this is part of a project to migrate onto a cloud platform. This is the perfect time to introduce automated testing. If the project is delivered incrementally, then the team can add tests incrementally as well.

The test pyramid

Managing a test suite is challenging. We want our test suite to run quickly, be easy to maintain, and to help us quickly zero in on the cause of failures. The testing pyramid is a concept for thinking about how to balance different types of tests to achieve these goals.

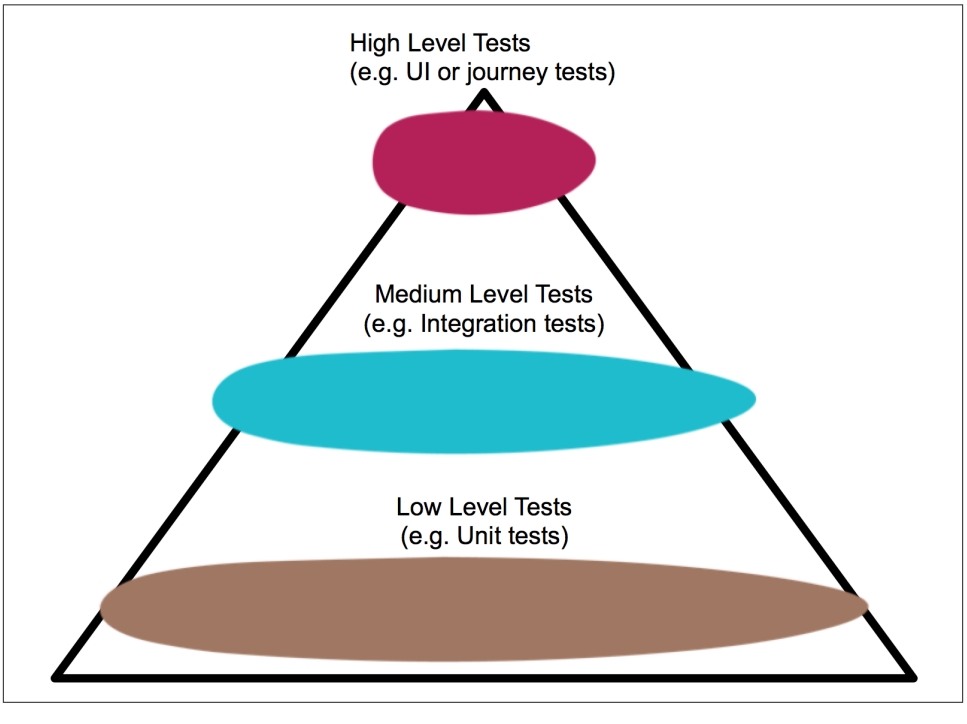


Figure 9-1. The testing pyramid

The pyramid puts tests with a broader scope towards the top, and those with a narrow scope at the bottom. So the lowest level tests validate very small components - unit tests which test a class or very few classes of application code, a Chef recipe, a Puppet manifest, etc. These tests run very quickly, and there will be large number of them. The strong

point of tests at this level is fast, specific feedback. If I make an error in code that causes a unit test to fail, it will run and fail quickly, and I immediately know which part of the code to find the error in.

Tests at the top of the pyramid test large parts of the system. These validate that the various components, systems, and services integrate correctly. For applications, these tests typically interact with the UI, simulating user journeys through the application. For infrastructure, they may test an end to end service request. For example, if our service catalog includes the ability to provision a new instance of Jira for a development team to use, our test could carry out the full provisioning operation and prove that the Jira instance is operational.

These higher level tests take much longer to run, and are more complex than lower level unit tests.

The middle tiers of the test pyramid fall between these, validating the integration of certain components, services, and requests. For example, we may have tests which validate our server provisioning process by building a complete VM and then checking that it passes our criteria for use.

A well-balanced test suite

A common anti-pattern in testing is the ice-cream cone, or inverted pyramid.

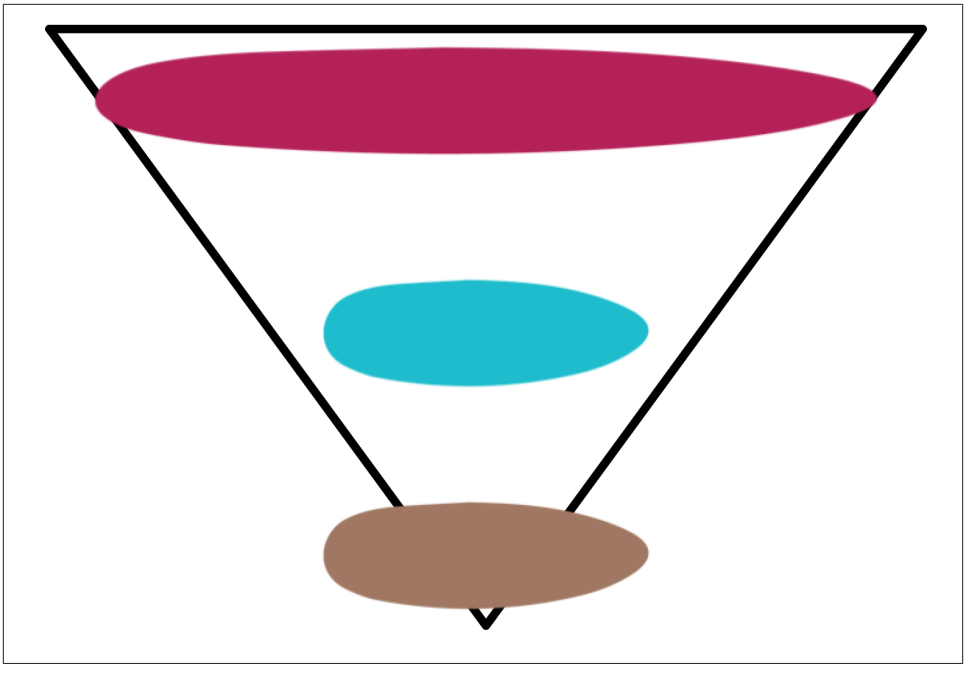


Figure 9-2. The inverted testing pyramid

A top-heavy test suite is difficult to maintain, slow to run, and doesn't pinpoint errors as well as a well-balanced suite.

UI level tests tend to be brittle. One change in the system can break a large number of UI tests, which can be more work to fix than the original change. This leads to the test suite falling behind development, which means it can't be run continuously. UI tests are also slower to run than lower level tests like unit tests, which again can make it impractical to run the full suite frequently. And because UI test cover a broad scope of code and components, when a test fails it may take a while to track down and fix the cause.

This usually comes about when a team puts a UI-based test automation tool at the core of their test automation strategy. This in turn often happens when testing is treated as a separate function from building. Testers who aren't involved in building the system don't have the visibility or involvement with the different layers of the stack. This prevents them from developing lower level tests and incorporating them into the build and change process. For someone who only interacts with the system as a black box, the UI is the easiest way to interact with it.

Practice: Test at the lowest level possible

UI and other high level tests should be kept to a minimum, and should only be run after the lower level tests have run. Many software teams run a small number of end to end journey tests, ensuring they touch the major components of the system and proves that integration works correctly. Specific features and functionality are tested at the component level, or through unit tests.

Whenever a higher level test fails, or a bug is discovered, look for ways to catch reoccurrences at the lowest possible level of testing. This ensure that the error is caught quickly, and that it's very clear exactly where the failure is. If an error can't be detected at the unit test level, move up a layer of the stack and try to catch it there. Essentially, a test at any level above a unit test should be testing an interaction that only happens when the components at that level are integrated.

So if there is a problem in component A, then the test for that component should catch it, rather than an integration test that runs across components A, B, and C. Tests should fail when testing A, B, and C only when they have a found an error in the way those components work together, even when each component is correct in itself.

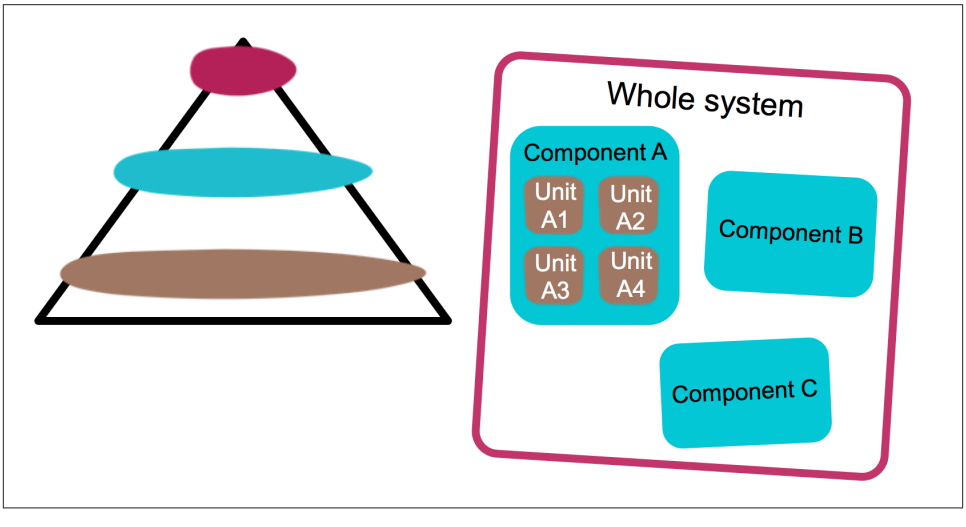


Figure 9-3. The testing pyramid and components

As an example, let's say we discover that an application error happens because of an error in a configuration file written by Chef. Rather than testing whether the error appears in the running application, we should write a test for the Chef recipe that builds the configuration file.

Testing for the error in the UI would need a VM instantiated, maybe a database or other dependencies set up, and the application built and deployed. A change which caused a problem in the application might cause our test to fail along with dozens of others.

Instead, we can have a test that runs whenever the Chef recipe changes. The test can use a tool like chefspec that can emulate what happens when Chef runs a recipe, without needing to apply it to a server.

Example of a Chefspec test.

```
require 'chefspec'

describe 'creating the configuration file for our_app' do
  let(:chef_run) { ChefSpec::Runner.new.converge('our_app_recipe') }

  it 'creates the right file' do
    expect(chef_run).to create_template('/etc/our_app.yml')
  end

  it 'gives the file the right attributes' do
    expect(chef_run).to create_template('/etc/our_app.yml').with(
      user:   'ourapp',
      group:  'ourapp'
    )
  end

  it 'sets the no_crash option'
    expect(chef_run).to render_file('/etc/our_app.yml').with_content('no_crash: true')
  end
end
```

Managing the test suite

Maintaining an automated test suite can be a burden. It gets easier as a team becomes more experienced, and writing and tweaking tests becomes routine. But there is a tendency for the test suite to grow and grow. Writing new tests becomes a habit, so teams need a corresponding habit of trimming tests to keep the suite manageable. We also need to avoid implementing unnecessary tests in the first place.

Practice: Only implement the layers you need

The testing pyramid suggests that we should have tests at every level of integration, for every component. So many teams' initial approach is to select and implement an array of test automation tools to cover all of these. This can quickly over-complicate the test suite.

There is no formula for what types of tests should be used for an infrastructure codebase. It's best to start out with fairly minimal tests, and then introduce new layers and types of testing when there is a clear need.

Anti-pattern: Reflective tests

A pitfall with low level infrastructure tests is writing tests that simply restate the configuration definitions. For example, here is a Chef snippet that creates the configuration file from our earlier test example:

Simple definition to be tested.

```
file '/etc/our_app.yml'
  owner ourapp
  group ourapp
end
```

Now here is a snippet from the earlier Chefspec unit test example:

Test that the definition created the file.

```
describe 'creating the configuration file for our_app' do
  # ...
  it 'gives the file the right attributes' do
    expect(chef_run).to create_template('/etc/our_app.yml').with(
      user:   'ourapp',
      group:  'ourapp'
    )
  end
```

This test only restates the definition. Basically, it's testing whether the Chef developers have correctly implemented the `file` resource implementation, rather than testing what we've written. If you're in the habit of writing this kind of test, you will end up with quite a few of them, and you'll waste a lot of effort editing every piece of configuration twice - once for the definition, once for the test.

As a rule, implement the test when there's some complexity to the logic that you want to validate. For the example of our configuration file, it may be worth writing that simple test if there is some complex logic that means the file may or may not be created.

For example, maybe `our_app` doesn't need a configuration file in most environments, so we only create the file in a few environments where the default configuration values need to be overridden. In this case, we would probably have two unit tests - one that ensures the file is created when it should be, and another that ensures it isn't created when it shouldn't be.

If you find yourself wanting to write tests because of a buggy third party tool, and that tool is open source, consider writing the tests and contributing them to the tool's codebase!

Continuously review testing effectiveness

The most effective automated testing regimes involve continuously reviewing and improving the test suite. From time to time you may need to go through and prune tests, remove layers or groups of tests, add a new layer or type of tests, add, remove, or replace tools, improve the way you manage tests, etc.

Whenever there is a major issue in production or even in testing, consider running a blameless post-mortem. One of the mitigations that should always be considered is adding or changing tests, or even removing tests.

Some signs that you should consider revamping your tests:

- If you spend more time fixing and maintaining certain tests than you save from the problems they catch,
- Do you often find issues in production?
- Do you often find issues that stop important events such as releases?
- Do you spend too much time debugging and tracing failures in high level tests?

Code coverage for infrastructure unit tests

Avoid the temptation to set targets for code coverage for infrastructure unit tests. Because configuration definitions tend to be fairly simple, an infrastructure codebase may not have as many unit tests as a software codebase might. Setting targets for unit test coverage - a much-abused practice in the software development world - will force the team to write and maintain useless test code, which makes maintaining good automated testing much harder.

Testing tools

Frameworks for testing configuration definitions

I've used a number of examples of chefspec tests in this chapter. Similar frameworks are available for most of the popular server configuration automation tools, including rspec-puppet. Saltstack even includes its own libraries to support unit testing in its distribution.

These make it possible to test a subset of definitions without needing to apply them to a running server. Usually, unit tests for configuration definitions are written using the language the configuration management tool is written in. Puppet and Chef are both written in Ruby, so rspec - the Ruby unit testing tool - is popular for unit testing manifests and recipes. Using the same language for low level testing frameworks makes it easier to write test doubles and to implement test setup.

Higher level testing

For higher level tests, the language used doesn't need to match the infrastructure tooling, because tests shouldn't interact with the internals of the tools. General purpose Behavior Driven Development (BDD) and UI testing tools can be used for higher level tests that involve UI's, especially web-based UIs.

For infrastructure, it's particularly useful to validate the state of files and services on a running server. Serverspec is another rspec-based tool, adding a functionality and libraries to connect to servers and validate them.

The following is an example chefspec that validates whether ourapp has been successfully installed on a server and is running.

Serverspec to validate a service.

```
describe service('our_app') do
  it { should be_running }
end
```

We can also use this type of test to validate our networking configuration. The following serverspec is run on a front-end web server, and checks that it is able to connect to the app server port. If someone makes a change to networking that blocks connections on port 8080, this will fail and alert us to the issue.

Serverspec to validate connectivity.

```
describe host('appserver') do
  it { should be_reachable.with( :port => 8080 ) }
end
```

Securely connecting to servers to run tests

Automated tests that need to remotely log into a server to validate it may be a security issue. These tests either need a hard-coded password, or else an ssh key or similar mechanism that authorizes unattended logins.

One approach that mitigates this is to have tests execute on the test server and push their results to a central server. This could be combined with monitoring, so that servers can self-test and trigger an alert if they fail.

Another approach is to use temporary credentials for test server instances. Some cloud platforms randomly generate credentials when creating a new instance, and return them to the script that triggered their creation. Other platforms allow credentials to be defined by the script that creates an instance. So automated tests can create temporary server instances and either generate random credentials or receive the credentials created by the platform. The tests then use the credentials to run the tests, then, when finished, destroy the server instance. The credentials never need to be shared or stored. If they are compromised, they don't give access to any other servers.

Implementing and running tests

Isolating components for testing

In order to effectively test a component, it must be isolated from any dependencies during the test.

As an example, consider testing an nginx web server's configuration[2]. The web server proxies requests to an application server. However, we would like to test the web server configuration without needing to start up an application server, which would need the application deployed to it, which in turn needs a database server, which in turn needs data schema and data. Not only does all of this make it complex to set up a test just to check the nginx configuration, there are many potential sources of error aside from the configuration we're testing.

A solution to this is to use a stub server instead of the application server (see "Test doubles" on page 144). This is a simple process that answers the same port as our application server, and gives responses needed for our tests. This stub could be a simple application which we can deploy just for the test, for example a Ruby Sinatra webapp.

2. This example was inspired by Philip Potter's blog post about testing web server configuration at GDS (UK Government Digital Services).

It could also be another nginx instance, or a simple http server written in the infrastructure team's favorite scripting language.

It's important that the stub server is simple to maintain and use. It only needs to return responses specific to the tests we write. For example, one test can check that requests to `/ourapp/home` returns an HTTP 200 response, so our stub server handles this path. Another test might check that, when the application server returns a 500 error, the nginx server returns the correct error page. So the stub might answer a special path like `/ourapp/500-error` with a 500 error. A third test might check that nginx copes gracefully when the application server is completely down, so this test is run without the stub in place.

A server stub should be quickly started, without only simple requirements from the environment and infrastructure. This means it can be run in complete isolation, for example in a lightweight container, as part of a larger test suite.

Test doubles

Mocks, fakes, and stubs are all types of "test doubles". A test double replaces a dependency needed by a component or service being tested, to simplify testing. These terms tend to be used in different ways by different people, but I've found the definitions used by Gerard Meszaros in his xUnit patterns book to be useful[3].

Refactoring components so they can be isolated

Often times, a particular component can't be easily isolated. Dependencies to other components may be hard-coded, or simply too messy to pull apart. Going back to "Writing tests" on page 131 earlier in this chapter, one of the benefits of writing tests while designing and building systems is that it forces us to improve our designs. This friction point - the component that is difficult to test in isolation - is a symptom of poor design. A well-designed system should have cleanly and loosely coupled components.

So when you run across components that are difficult to isolate, you should fix this design. This may be difficult. Components may need to be completely re-written, libraries, tools, and applications may need to be replaced. As the saying goes, this is a feature, not a bug. In order to make a system testable, it needs a clean design.

3. Martin Fowler's bliki Mocks Aren't Stubs is a useful reference to test doubles, and is where I learned about Gerard Meszaros' book.

There are a number of strategies for restructuring systems. Refactoring[4] is an approach that prioritizes keeping the system fully working throughout the process of restructuring the internal design of a system.

Managing external dependencies

It's common to depend on services not managed by your own team. Infrastructure elements and services like DNS, authentication services, or email servers may be provided by a separate team or an external vendor. These can be a challenge for automated testing, for a number of possible reasons:

- They may not be able to handle the load generated by continuous testing, not to mention performance testing.
- They can have availability issues which affect your own tests. This is especially common when vendors or teams provide test instances of their services.
- There may be costs or request limits which make them impractical to use for continuous testing.

Test doubles can be used to stand in for external services for most testing. You should only integrate with the external services once your own systems and code have been validated. This ensures that if there is a failed test, you know it's either because of an issue with the external service, or in the way that you've integrated with it.

You should ensure that, if an external service does fail, it's very clear that this is the issue. I recall spending over a week with one team, poring over our application and infrastructure code to diagnose an intermittent test failure. It turned out that we were hitting a request limit in our cloud vendor's API. It's frustrating to waste so much time on something that could have been caught much more quickly.

Any integrations with third parties, and even those between your own services, should implement checking and reporting that makes it immediately visible when there is an issue. This should be made visible through monitoring and information radiators for all environments. In many cases, teams implement separate tests and monitoring checks that report on connectivity with upstream services.

Test setup

By now you may be tired of hearing about the importance of consistency, reproducibility, and repeatability. If so, brace yourself: these things are essential for automated testing.

4. Martin Fowler has written about Refactoring, as well as other patterns and techniques for improving system architecture. The Strangler Application is a popular approach I've seen on a number of projects.

Tests that behave inconsistently have no value. So a key part of automated testing is ensuring the consistent setup of environments and data.

For tests which involve setting up infrastructure - building and validating a VM, for example - the infrastructure automation itself lends itself to repeatability and consistency. The challenge comes with state. What does a given test assume about data? What does it assume about configuration that has already been done to the environment?

A general principle of automated testing is that each test should be independent, and should ensure the starting state it needs. It should be possible to run tests in any order, and to run any test by itself, and always get the same results.

So it's not a good idea to write a test that assumes another test has already been run. For example, the example below shows two tests. The first tests the installation of nginx on our web server, the second tests that the home page loads with expected content.

Tests that are too tightly coupled.

```
describe 'install and configure web server' do
  let(:chef_run) { ChefSpec::SoloRunner.converge(nginx_configuration_recipe) }

  it 'installs nginx' do
    expect(chef_run).to install_package('nginx')
  end
end

describe 'home page is working' do
  let(:chef_run) { ChefSpec::SoloRunner.converge(home_page_deployment_recipe) }

  it 'loads correctly' do
    response = Net::HTTP.new('localhost',80).get('/')
    expect(response.body).to include('Welcome to our home page')
  end
end
```

This example looks reasonable at a glance, but if the *home page is working* spec is run on its own, it will fail, because there will be no web server to respond to the request.

We could ensure that the tests always run in the same order, but this will make the test suite overly brittle. If we change the way we install and configure the web server, we may need to make changes to many other tests which make assumptions about what has happened before they run. It's much better to make each test self-contained, as in the example below:

Uncoupled tests.

```
describe 'install and configure web server' do
  let(:chef_run) { ChefSpec::SoloRunner.converge(nginx_configuration_recipe) }

  it 'installs nginx' do
    expect(chef_run).to install_package('nginx')
```

```
  end
end

describe 'home page is working' do
  let(:chef_run) {
    ChefSpec::SoloRunner.converge(nginx_configuration_recipe,
                                  home_page_deployment_recipe)
  }

  it 'loads correctly' do
    response = Net::HTTP.new('localhost',80).get('/')
    expect(response.body).to include('Welcome to our home page')
  end
end
```

In this example, the second spec's dependencies are explicit - you can see at a glance that it depends on the nginx configuration. It's also self-contained - either of these tests can be run on their own, or in any order, with the same result every time.

Managing test data

Some tests rely on data, especially those which test applications or services. As an example, in order to test a monitoring service, a test instance of the monitoring server may be created. Various tests may add and remove alerts to the instance, and emulate situations that trigger alerts. This requires thought and care to ensure tests can be run repeatably in any order.

For example, we may write a test that adds an alert and then verifies it's in the system. If we run this test twice on the same test instance, it may try to add the same alert a second time. Depending on the monitoring service, the attempt to add a duplicate alert may fail. Or, the test may fail because it finds two alerts with the same name. Or the second attempt to add the alert may not actually work, but the validation finds the alert added the first time, so does not tell us about a failure.

So some rules for test data:

- Each test should create the data it needs
- Each test should either clean up its data afterwards, or else create unique data each time it runs
- Tests should never make assumptions about what data does or doesn't exist when it starts

Immutable servers (as described in "Immutable server flow" on page 110) help ensure a clean and consistent test environment. Persistent test environments tend to drift over time, so that they're no longer consistent with production.

Monitoring and testing

Monitoring and automated testing have a lot in common. Both make assertions about the state of an infrastructure and its services, and both alert the team that there is a problem when an assertion fails. Combining or at least integrating these concerns can be very effective. Consider re-using automated tests to validate whether systems are working correctly in production. Some caveats, however:

- Many automated tests have side effects, and/or need special setup, which may be destructive in production.
- Many tests aren't relevant for production monitoring. Monitoring checks for problems which can happen because of changes in operational state. Testing validates whether a change to code is harmful. Once the code is changed and applied to production, re-running a functional test may be pointless.
- Re-using code between tests and monitoring is only useful if it makes life easier. In many cases, trying to bend and twist a testing tool so it can be used for production monitoring may be more effort than it's really worth.

Conclusion

Automated testing is possibly the most challenging aspect of infrastructure as code, while also being the most important for supporting the a reliable and adaptable infrastructure. Teams should build the habits and processes to routinely incorporate testing as a core part of their infrastructure. The next chapter explains how to create change management pipelines to support these habits.