Chapter 1. Challenges and Principles

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

Why Infrastructure as Code?

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

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

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

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

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

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

THE IRON AGE AND THE CLOUD AGE

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

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

What Is Infrastructure as Code?

Infrastructure as code is an approach to infrastructure automation based on practices from software development. It emphasizes consistent, repeatable routines for provisioning and changing systems and their configuration. Changes are made to definitions and then rolled out to systems through unattended processes that include thorough validation.

The premise is that modern tooling can treat infrastructure as if it were software and data. This allows people to apply software development tools such as version control systems (VCS), automated testing libraries, and deployment orchestration to manage infrastructure. It also opens the door to exploit development practices such as test-driven development (TDD), continuous integration (CI), and continuous delivery (CD).

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

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

Goals of Infrastructure as Code

The types of outcomes that many teams and organizations look to achieve through infrastructure as code include:

- IT infrastructure supports and enables change, rather than being an obstacle or a constraint.
- Changes to the system are routine, without drama or stress for users or IT staff.
- IT staff spends their time on valuable things that engage their abilities, not on routine, repetitive tasks.
- Users are able to define, provision, and manage the resources they need, without needing IT staff to do it for them.
- Teams are able to easily and quickly recover from failures, rather than assuming failure can be completely prevented.
- Improvements are made continuously, rather than done through expensive and risky "big bang" projects.
- Solutions to problems are proven through implementing, testing, and measuring them, rather than by discussing them in meetings and documents.

INFRASTRUCTURE AS CODE IS NOT JUST FOR THE CLOUD

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

I use the phrase "dynamic infrastructure" to refer to the ability to create and destroy servers programmatically; Chapter 2 is dedicated to this topic. Cloud does this naturally, and virtualization platforms can be configured to do the same. But even hardware can be automatically provisioned so that it can be used in a fully dynamic fashion. This is sometimes referred to as "bare-metal cloud."

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

Challenges with Dynamic Infrastructure

This section looks at some of the problems teams often see when they adopt dynamic infrastructure and automated configuration tools. These are the problems that infrastructure as code addresses, so understanding them lays the groundwork for the principles and concepts that follow.

Server Sprawl

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

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

This leads to inconsistency across the servers, called *configuration drift*.

Configuration Drift

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

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

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

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

Snowflake Servers

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

Years ago I ran servers for a company that built web applications for clients, most of which were monstrous collections of Perl CGI. (Don't judge us, this was the dot-com era, and 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 that simply wouldn't run on 5.8.

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

We ran this way for a shamefully long time, keeping Perl 5.6 on the staging server and crossing our fingers whenever we deployed to production. We were terrified to touch anything on the production server, afraid to disturb whatever magic made it the only server that could run the client's application.

This situation led us to discover Infrastructures.Org, 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, installing the operating system with the Fully Automated Installation (FAI) tool, configuring the server with CFEngine, and checking everything into our CVS version control system.

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

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

Fragile Infrastructure

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

The solution is to migrate everything in the infrastructure to a reliable, reproducible infrastructure, one step at a time. The *Visible Ops Handbook*[3] outlines an approach for bringing stability and predictability to a difficult infrastructure.

DON'T TOUCH THAT SERVER. DON'T POINT AT IT. DON'T EVEN LOOK AT IT.

There is the possibly apocryphal story of the data center with a server that nobody had the login details for, and nobody was certain what the server did. Someone took the bull by the horns and unplugged the

server from the network. The network failed completely, the cable was plugged back in, and nobody ever touched the server again.

Automation Fear

At an Open Space session on configuration automation at a DevOpsDays conference, I asked the group how many of them were using automation tools like 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.

This is the automation fear spiral, as shown in Figure 1-1, 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.

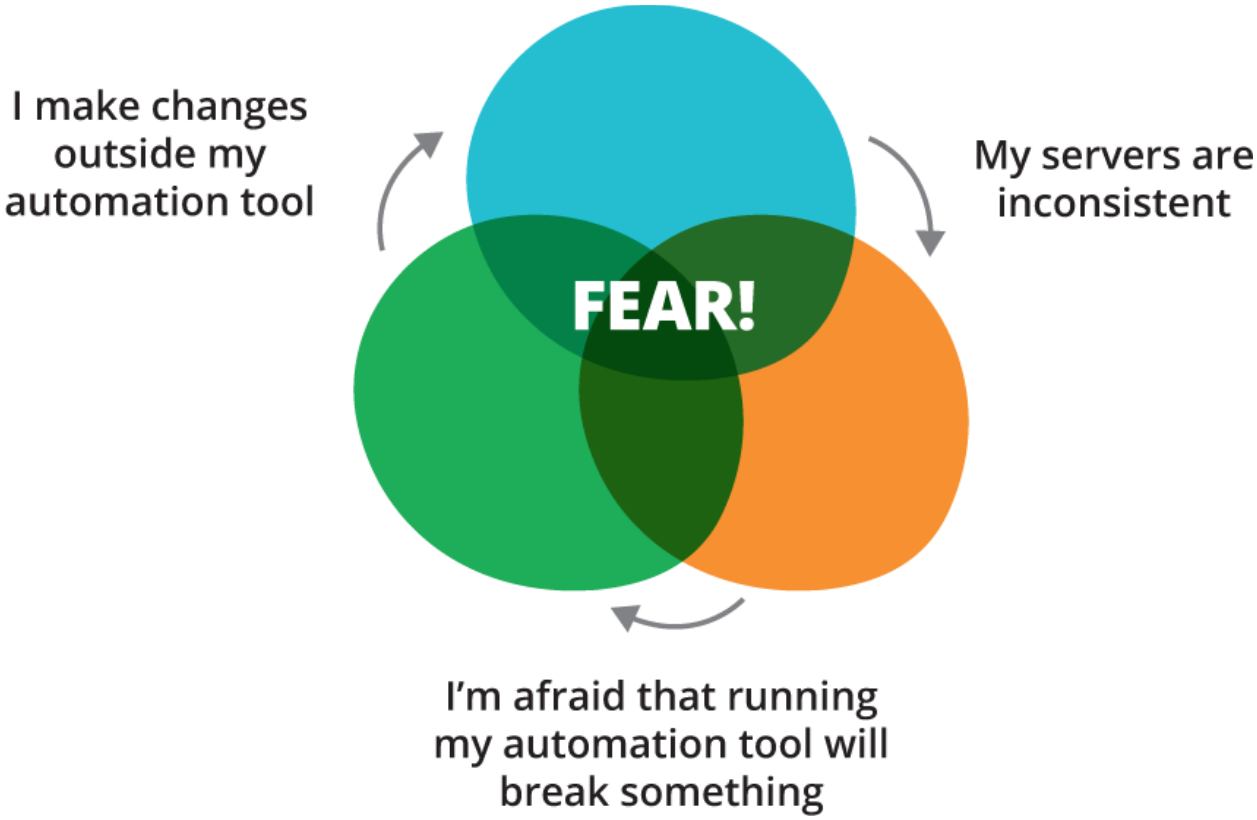

Figure 1-1. The automation fear spiral

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, you would never need to touch an automated infrastructure once you'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, infrastructure decays over time. The folks at Heroku call this *erosion*. Erosion is the idea that problems will creep into a running system over time.

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

- Operating system upgrades, kernel patches, and infrastructure software (e.g., Apache, MySQL, SSH, OpenSSL) updates to fix security vulnerabilities
- The server's disk filling up with logfiles
- One or more of the application's processes crashing or getting stuck, requiring someone to log in and restart them

- Failure of the underlying hardware causing one or more entire servers to go down, taking the application with it

Principles of Infrastructure as Code

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

Systems Can Be Easily Reproduced

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

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

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

Systems Are Disposable

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

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

CATTLE, NOT PETS

A popular expression is to "treat your servers like cattle, not pets."[4] I miss the days of having themes for server names and carefully selecting names for each new server I provisioned. But I don't miss having to manually tweak and massage every server in our estate.

A fundamental difference between the iron age and cloud age is the move from unreliable software, which depends on the hardware to be very reliable, to software that runs reliably

on unreliable hardware.[5] See Chapter 14 for more on how embracing disposable infrastructure can be used to improve service continuity.

THE CASE OF THE DISAPPEARING FILE SERVER

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

Systems Are Consistent

Given two infrastructure elements providing a similar service—for example, two application servers in a cluster—the servers should be nearly identical. Their system software and configuration should be the same, except for those bits of configuration that differentiate them, 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 has 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 (e.g., one of the file servers needs a larger disk partition), there are two ways that keep consistency. One is to change the definition so that all file servers are built with a large enough partition to meet the need. The other is to add a new class, or role, so that there is now an "xl-file-server" with a larger disk than the standard file server. Either type of server can be built repeatedly and consistently.

Being able to build and rebuild consistent infrastructure helps with configuration drift. But clearly, changes that happen after servers are created need to be dealt with. Ensuring consistency for existing infrastructure is the topic of Chapter 8.

Processes Are Repeatable

Building on the reproducibility principle, any action you carry out on your infrastructure should be repeatable. This is an obvious benefit of using scripts and 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 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 filesystem 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 the ability to automate things.

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

Design Is Always Changing

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

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

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

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

Practices

The previous section outlined high-level principles. This section describes some of the general practices of infrastructure as code.

Use Definition Files

The cornerstone practice of infrastructure as code is the use of definition files. A definition specifies infrastructure elements and how they should be configured. The definition file is used as input for a tool that carries out the work to provision and/or configure instances of those elements. Example 1-1 is an example of a definition file for a database server node.

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

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

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

Definition files are managed as text files. They may use a standard format such as JSON, YAML, or XML. Or they may define their own domain-specific language (DSL).[6]

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

Self-Documented Systems and Processes

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

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

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

AUTOMATICALLY GENERATING DOCUMENTATION

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

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

Version All the Things

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

Reasons why VCS is essential for infrastructure management include:

Traceability

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

Rollback

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

Correlation

: When scripts, configuration, artifacts, and everything across the board are in version control and correlated by tags or version numbers, it can be useful for tracing and fixing more complex problems.

Visibility

: Everyone can see when changes are committed to a version control system, which helps situational awareness for the team. Someone may notice that a change has missed something important. If an incident happens, people are aware of recent commits that may have triggered it.

Actionability

: VCSs can automatically trigger actions when a change is committed. This is a key to enabling continuous integration and continuous delivery pipelines.

Chapter 4 explains how VCS works with configuration management tools, and Chapter 10 discusses approaches to managing your infrastructure code and definitions.

Continuously Test Systems and Processes

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

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

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

Chapter 11 explores practices and techniques for implementing testing as part of the system, and particularly how this can be done effectively for infrastructure.

Small Changes Rather Than Batches

When I first got involved in developing IT systems, my instinct was to implement a complete piece of work before putting it live. It made sense to wait until it was "done" before spending the time and effort on testing it, cleaning it up, and generally making it "production ready." The work involved in finishing it up tended to take a lot of time and effort, so why do the work before it's really needed?

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

- It's easier, and less work, to test a small change and make sure it's solid.
- If something goes wrong with a small change, it's easier to find the cause than if something goes wrong with a big batch of changes.
- It's faster to fix or reverse a small change.
- One small problem can delay everything in a large batch of changes from going ahead, even when most of the other changes in the batch are fine.

- Getting fixes and improvements out the door is motivating. Having large batches of unfinished work piling up, going stale, is demotivating.

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

Keep Services Available Continuously

It's important that a service is always able to handle requests, in spite of what might be happening to the infrastructure. If a server disappears, other servers should already be running, and new ones quickly started, so that service is not interrupted. This is nothing new in IT, although virtualization and automation can make it easier.

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

The chapter on continuity (Chapter 14) goes into techniques for keeping service and data continuously available.

Antifragility: Beyond "Robust"

Robust infrastructure is a typical goal in IT, meaning systems will hold up well to shocks such as failures, load spikes, and attacks. However, infrastructure as code lends itself to taking infrastructure beyond robust, becoming antifragile.

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

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

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

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

The Secret Ingredient of Antifragile IT Systems

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

This doesn't fit the idea of automation as a way to run things without humans. Someday it might be possible 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 nontechnology 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.[7]

Conclusion

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 mean time to recover (MTTR) and focuses on ways to improve this. Although mean time between failure (MTBF) may also be tracked, the team does not rely on avoiding failures.[8]
- Team members feel their work is adding measurable value to the organization.

What's Next?

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

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

Dynamic infrastructure platforms
: Used to provide and manage basic infrastructure resources, particularly compute (servers), storage, and networking. These include public and private cloud infrastructure services, virtualization, and automated configure of physical devices. This is the topic of Chapter 2.

Infrastructure definition tools
: Used to manage the allocation and configuration servers, storage, and networking resources. These tools provision and configure infrastructure at a high level. This is the subject of Chapter 3.

Server configuration tools
: These tools deal with the details of servers themselves. This includes software packages, user accounts, and various types of configuration. This group, which is typified by specific tools including CFEngine, Puppet, Chef, and Ansible, are what many people think of first when discussing infrastructure automation and infrastructure as code. These tools are discussed in Chapter 4.

Infrastructure services
: Tools and services that help to manage infrastructure and application services. Topics such as monitoring, distributed process management, and software deployment are the subject of Chapter 5.

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

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

3 First published in 2005, the *Visible Ops Handbook* by Gene Kim, George Spafford, and Kevin Behr (IT Process Institute, Inc.) was 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.

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

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

6 As defined by Martin Fowler and Rebecca Parsons in *Domain-Specific Languages* (Addison-Wesley Professional), "DSLs are small languages, focused on a particular aspect of a software system. You can't build a whole program with a DSL, but you often use multiple DSLs in a system mainly written in a general-purpose language." Their book is a good reference on domain-specific languages, although it's written more for people thinking about implementing one than for people using them.

7 Brian L. Troutwin gave a talk at DevOpsDays Ghent in 2014 titled "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.

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