We are a software company and a community of passionate, purpose-led individuals. We think disruptively to deliver technology to address our clients' toughest challenges, all while seeking to revolutionize the IT industry and create positive social change.
This Preview Edition of *Infrastructure as Code, Chapters 10–12*, is a work in progress. The final book is currently scheduled for release in April 2016 and will be available at [oreilly.com](http://oreilly.com) and other retailers once it is published.
## Table of Contents

1. **Software engineering practices for infrastructure.** ........................................ 7
   - System quality .................................................................................................................. 8
   - Infrastructure quality through code ............................................................................ 8
   - Fast feedback .................................................................................................................. 9
   - Continuous Integration (CI) ............................................................................................ 9
       - Who broke the build? ..................................................................................................... 11
       - CI for infrastructure ................................................................................................... 12
   - VCS for infrastructure management .............................................................................. 13
       - What to manage in a VCS .......................................................................................... 13
       - Continuous Delivery (CD) .......................................................................................... 14
       - Continuous Delivery versus continuous deployment ................................................. 16
   - Code quality .................................................................................................................... 16
       - Practice: Clean code .................................................................................................... 16
       - Practice: Manage technical debt ............................................................................... 17
   - Managing major infrastructure changes ...................................................................... 18
   - Conclusion ....................................................................................................................... 18

2. **Testing infrastructure changes.** .............................................................. 21
   - The agile approach to testing ....................................................................................... 21
       - Automating tests for fast feedback .......................................................................... 22
       - Organically building a test suite .............................................................................. 23
   - Structuring the test suite - the test pyramid ................................................................. 24
       - Avoiding an unbalanced test suite ............................................................................ 25
       - Practice: Test at the lowest level possible ............................................................... 26
       - Practice: Only implement the layers you need .......................................................... 27
       - Practice: Prune the test suite ..................................................................................... 27
       - Practice: Continuously review testing effectiveness ............................................... 27
   - Implementing a balanced test suite .............................................................................. 28
Low-level testing
Mid-level testing
Higher level tests
Testing operational quality
Managing test code
  Practice: Keep test code with the code it tests
  Anti-pattern: Reflective tests
  Techniques to isolate components for testing
  Refactoring components so they can be isolated
  Managing external dependencies
  Test setup
Roles and workflow for testing
  Principle: People should write tests for what they build
  The test-writing habit
  Principle: Everyone should have access to the testing tools
  The value of a Quality Analyst
  Test Driven Development (TDD)
Conclusion

3. Change management pipelines for infrastructure. ................................. 47
  Principles for designing pipelines
    Consistency across stages
    Immediate feedback for every change
    Run automated stages before manual stages
    Get production-like sooner rather than later
  Basic pipeline designs
    The build stage
    Publishing a configuration artifact
    Automated testing stages
    Manual validation stages
    Apply to live
    The rhythm of the pipeline
  Practices for using a pipeline
    Every change is intended to be production ready
    Don’t make changes outside the pipeline
    Stop the line on any failure
  Scaling pipelines to more complex systems
    Pattern: Fan-in pipelines
    Scaling pipelines
    Principle: keep pipelines short
    Principle: Decouple pipelines
    Integration models
Techniques for handling dependencies between components
  Pattern: Library dependency 69
  Pattern: Self-provisioned service instance 71
  Providing pre-release library builds 72
  Providing test instances of a service to consumers 73
  Using test instances of a services 74
Interface contracts and contract testing 76
  Practice: Ensure backwards compatibility of interfaces 76
  Practice: Decouple deploying from releasing 77
  Practice: Version tolerance 77
  Practice: Providing test doubles 78
  Practice: Contract tests 78
  Practice: Automated reference implementation 79
  Practice: Contract validation tests 79
Conclusion 80
The enabling idea of infrastructure as code is that the systems and devices which are used to run software can be treated as if they, themselves, are software. The previous chapters of this book have examined the nuts and bolts of this premise - the underpinning technology, tooling, and some patterns for handling dynamic infrastructure.

This chapter looks at some of the fundamental practices of software engineering, with an eye to how people are adapting them to working with infrastructure. The goal is to understand some specific practices which software development teams have found helpful, and which underpin some of the more advanced topics in later chapters.

Specifically, this chapter will talk about the use of Version Control Systems (VCS) for managing infrastructure code and definitions. This is explained in the context of Continuous Integration (CI) and Continuous Delivery (CD). The practice of continuously testing and fully validating changes to software have transformed software development over the past one or two decades, and promise to do the same for infrastructure change management. They are essential tools for maintaining a high level of code quality, which is too often seen as a luxury, but which can actually have measurable benefits for organizations.

Later chapters will build on the core software engineering practices from this one. Chapter 2 covers testing, in particular how to design, implement, and maintain a well-balanced suite of automated tests. Chapter 3 explains how change management pipelines are used to put software engineering and testing practices into use. Chapter 4 outlines the workflows that people can use to actually work with the tools on a day to day basis to develop, test, and debug automated infrastructure as code. And Chapter 5 closes the book by explaining how teams and organizations can build and run IT operations using infrastructure as code.
Note for experienced software developers

Experienced software developers who are experienced with agile and XP (eXtreme Programming) practices like TDD and CI may find the contents of this chapter familiar. Even so, it should be worth at least skimming the contents, as there are some notes on how these specifically apply to infrastructure development.

System quality

Good software engineering practices produce high quality code and systems. Too often, quality is seen as a simple matter of functional correctness. In reality, high quality is an enabler of change. The true measure of the quality of a system, and its code, is how quickly and safely changes are made to it.

Poor quality systems are difficult to change. A seemingly simple change can take much more time than seems reasonable, and cause far more problems than it should. Even the people who wrote the code have difficulty understanding how a given part of the codebase works when looking at it. Code that is difficult to understand is difficult to change without creating unexpected problems. That simple change may require pulling apart large sections of code, which can drag on and create even more of a mess. Users and managers are left puzzled and frustrated by how ineffective the team seems.

The same is often true with infrastructure, even without automation. Different people have built, changed, updated, optimized, and fixed various parts of the systems involved over time. The whole interrelated web of parts can be precarious, any change to one part can potentially break one or more others.

High quality systems are easier, and safer, to change. Even someone new to the team can understand how any piece of it works when they look into it. The impact of a change is generally clear. Tools and tests are in place which quickly surface problems caused by a change.

Systems like this need only minimal technical documentation. Typically, most members of the team can quickly draw the parts of the system’s architecture that are relevant to a particular conversation. New joiners with relevant technical knowledge are brought up to speed by a combination of conversations, poking through the code, and getting their hands dirty by working on the system.

Infrastructure quality through code

Defining system infrastructure as code and building it with tools doesn’t make the quality any better. At worst, it can complicate things. A mess of inconsistent, poorly maintained definition files and tooling can be combined with ad-hoc manual inter-
ventions and special cases. The result is a fragile infrastructure where running the wrong tool can do catastrophic damage.

What infrastructure as code does is shift the focus of quality to the definitions and tooling systems. It is essential to structure and manage automation so that it has the virtues of quality code - easy to understand, simple to change, with fast feedback from problems. If the quality definitions and tools used to build and change infrastructure is high, then the quality, reliability, and stability of the infrastructure itself should be high.

**Fast feedback**

A cornerstone of high quality systems is fast feedback on changes. When I make a mistake in a change to a configuration definition, I'd like to find out about that mistake as quickly as possible.

The shorter the loop between making a change and being notified it causes a problem, the easier it is to find the cause. If I know I made a mistake in the work I've done over the past few minutes, I can find it quickly. If a problem crops up in a massive set of changes I've spent a few weeks on, there is quite a lot of code to go through.

Ideally I'd like to be notified of my mistake before it's been applied to an important system, which is the goal of CI and CD. At worst, I'd like good monitoring to flag a problem quickly after it's been applied, so my teammates and I can resolve it with as little damage as possible.

**Continuous Integration (CI)**

Continuous Integration is the practice of frequently integrating and testing all changes to a system as they are being developed. CI tools, such as Bamboo, Jenkins, SnapCI, TeamCity, TravisCI, and others, can be used to enable this practice.

However, it's important to note that Continuous Integration is not the practice of using a CI tool, but is rather the practice of frequently integrating all changes. With Continuous Integration, all of the developers on a team commit their changes to the trunk of the codebase. Every time a commit is made, the CI tool builds the codebase and runs an automated test suite.

The benefit of this approach is fast feedback when a change doesn't build correctly, or which causes a test to fail. Because the tests are run on every commit, it's immediately clear which set of changes caused the issue. The more frequently developers commit, the smaller the change set, and the faster problems are surfaced and fixed.

Many development teams use CI tools but don't integrate and test continuously. Running the test suite on a schedule, for example nightly, gives feedback on a slightly slower loop than running it on every commit throughout the day.
Another common alternative to continuously integrating is to split development work into different streams in the codebase, using VCS branching. One or more developers may work on each branch, keeping their changes isolated from other work going on until they have finished their work. At that point, the branch is merged, and any conflicts with other workstreams are resolved.

Although tests may be run automatically on commits to each separate branch using CI, the integrated change sets are only tested together when the branches are merged. Some teams find that this works well for them, generally by keeping branches very short-lived. The longer a branch runs without being integrated and tested together with the other branches, the slower the feedback loop.¹

¹ Paul Hammond has a good presentation on issues with branching.
Figure 1-2. Frequent commits are smaller, and more easily managed

The more frequently changes are committed to the shared trunk, the smaller they are, the fewer conflicts will arise, the easier testing is, and the less there is to fix.

Table 1-1. Common branching strategies

<table>
<thead>
<tr>
<th>Branching Strategy</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Feature branching</td>
<td>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.</td>
</tr>
<tr>
<td>Release branching</td>
<td>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.</td>
</tr>
<tr>
<td>Trunk based development</td>
<td>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.</td>
</tr>
</tbody>
</table>

Who broke the build?

In order for Continuous Integration to be effective, every failure of a build or test in the CI system needs to be addressed immediately. Ignoring, or not noticing, when a build fails allows errors to pile up, and becomes confusing and difficult to untangle later. So a failed run in the CI tool 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 presumably 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 ???) 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 2 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.

CI for infrastructure

For infrastructure as code, CI can be used to continuously test changes made to definition files, scripts, and other tooling and configuration written and maintained for running the infrastructure. Each of these should be managed in a VCS, as described below. Teams should avoid branching, to avoid building up an increasing “debt” of code that will need to be merged and tested. Each commit should trigger some level of testing.

Chapter 2 describes the “Structuring the test suite - the test pyramid” on page 24, which is a way of considering how to organize different layers of testing for the infrastructure and systems.
VCS for infrastructure management

?? reasons why a Version Control System (VCS) is an essential part of an infrastructure as code regime. ?? 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 ?? in ??:

- 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. ?? covers 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.
Continuous Delivery (CD)

Continuous Integration addresses work done on a single codebase. Multiple developers working on the same code continuously merge their work so that issues are immediately surfaced and resolved. Continuous Delivery\(^2\) expands the scope of this continuous integration to the entire system, with all of its components.

The idea behind CD is to ensure that all of the deployable components, systems, and infrastructure are continuously validated to ensure that they are production ready. It is used to address the problems of the “integration phase”.

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.\(^3\)

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.

Carrying out frequent 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 on a project at a large enterprise that took twelve months before the first 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.

\(^2\) Continuous Delivery, by Jez Humble and David Farley, is the canonical resource on the subject.

\(^3\) The lego integration game is a fun way to demonstrate the differences between late integration and continuous integration.
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 organization'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.

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 organization I mentioned in my story (“The twelve-month Continuous Delivery project” on page 14), 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.

**Code quality**

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”⁴ 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.

⁴ See Robert “Uncle Bob” Martin’s *Clean Code: A Handbook of Agile Software Craftsmanship*
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.

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 ???). 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 ??? 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

This chapter reviewed some core software development practices and how they relate to working with infrastructure as code. The underlying theme of these practices is quality. In order to ensure the quality of systems and the code which defines them is to make quality a first order concern.
Teams which prioritize the quality of their systems, by getting continuous feedback and acting on it immediately, create a virtuous cycle. They have the confidence to routinely make the small fixes and tweaks that keep their systems humming smoothly. This gives them more time to spend on the more satisfying, higher order work rather than fighting fires.

The next chapter moves into more specifics on automated testing.
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. There is a common misconception that quality and delivery speed are opposing forces that must be traded off, one against the other. This mindset leads to the idea that automation can speed the delivery process by making the act of testing a system go faster.

These misconceptions - testing as the enemy of delivery speed, and automation as a silver bullet to make testing go faster - lead to expensive, failed test automation initiatives.

The reality is that quality is an enabler of delivery speed. Spending less time finding and fixes bugs creates a fragile system which is difficult to change. It doesn't take long until even trivial changes are difficult and time consuming, as well as risky.

The goal of automated testing is to help teams focus on keeping the quality of their system high, through fast feedback. When combined with a team culture and discipline that prioritizes quality, automated tooling helps to find quality issues quickly so the team can respond and fix them quickly. This in turn keeps the system in a state where changes can be made quickly, easily, and confidently.

So faster delivery speed is a side effect of focusing on quality. And automated test tooling is an aid to keeping quality at the forefront of the team's attention.

**The agile approach to testing**

Many organizations have a process that separates implementation and testing into different phases of work, usually carried out by separate teams. Agile processes encourage teams to integrate testing with implementation, in order to shorten the
feedback loop. Testing takes place continuously, as changes are made. This is done by testers and developers working closely together, combined with automated testing.

![Diagram of change-test-fix feedback loop]

*Figure 2-1. The change-test-fix feedback loop*

So the most useful goal for test automation isn’t to make a test phase run faster, but to enable testing and fixing activities as a core part of the workflow. As someone works on changes to the system, whether it’s application code or infrastructure definitions, they continuously test. They test to make sure their change works as they expect. They test to make sure their change doesn’t break other parts of the system. And they test to make sure they aren’t leaving any loose ends to cause problems later on.

People test so they can fix each problem right away, while they’re still working on their changes. Everything is still fresh in their mind. Because the scope of the changes are very small, the problems are quick to find and fix.

**Automating tests for fast feedback**

Teams whose testing process is based around separate implementation and test phases often attempt to adopt automation by automating their test phase. This is often a project owned by the QA team, which aims to create a comprehensive regression test suite. There are several reasons why these automation efforts tend to be disappointing.
One problem is that automated test suites built by a separate testing team tend to focus on high level testing. This is because tools marketed at test teams tend to be UI-focused, because they are seen as replacing manual testing driven through the UI. The result of this, however, is that the test suite is unbalanced. The test pyramid described later in this chapter explains how a balanced test suite helps.

The key to designing and implementing a well-balanced automated test suite is for the entire team, especially the implementers, to be involved in planning, designing, and implementing it.

Another challenge with big bang test automation initiatives is that they bite off more than they can chew, and struggle to keep up with ongoing development. The system is a constantly moving and changing target. Before the massive test suite is complete, the system has changed. Assuming the test suite can be completed, the system will change again immediately. So tests tend to be constantly broken, and the nirvana of a complete test suite is never achieved.

Aiming for the goal of a complete, finished test suite is doomed to fail. Instead, the goal of an automation initiative should be to embed the habit of continuously writing tests as a part of routine changes and implementation work. So the outcome of an automated testing initiative is not a completed test suite, but a set of working habits and routines.

When automated testing has been successfully adopted by a team, tests are written or updated whenever a change is made to the system. CI and CD regimes run the relevant tests for every change continuously, as discussed in Chapter 1. The team responds immediately to fix failing tests.

**Organically building a test suite**

The way to start an initiative that results in embedding these kinds of testing habits is not to attempt to build a full test suite to cover the existing functionality. Instead, write tests for each new change as it comes up. When a bug is found, write a test that exposes that bug, and then fix it. When a new feature or capability is needed, begin implementing tests as you go, possibly even using TDD as described later in this chapter.

Building the test suite organically as a part of making routine changes forces everyone to learn the habits and skills of sustainable, continuous testing. Again, the outcome to aim for is not a “finished” test suite, but the routine of testing each change. A test suite will emerge from this approach. Interestingly, the test suite that emerges will be focused on the areas of the system that most need tests - the ones which change and/or break the most.
Structuring the test suite - the test pyramid

Managing a test suite is challenging. We want our tests to run quickly, to 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.

The pyramid puts tests with a broader scope towards the top, and those with a narrow scope at the bottom. The lower tiers validate smaller, individual things such as definition files and small scripts. The middle tiers test some of the lower level components together, for example by creating a running server. The highest tiers tests working systems together, for example a service with multiple servers and their surrounding infrastructure.

There are more tests at the lower levels of the pyramid, and fewer at the top. Since the lower level tests are smaller and more focused, they run very quickly. The higher level tests tend to be more involved, taking longer to set up and then run, so they run slower.
Avoiding an unbalanced test suite

A common problem with automated test suites is the ice-cream cone, or inverted pyramid. This happens when there are too many high level tests when compared with lower level tests.

Figure 2-3. 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.

High level tests tend to be brittle. One change in the system can break a large number of 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. Higher level tests are also slower to run than the more focused lower level tests, which makes it impractical to run the full suite frequently. And because higher level tests 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 re-occurrences 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.

As an example, let's say we discover that an application error happens because of an error in a configuration file managed 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 2-1. Example of a Chefspec test*

```ruby
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

(Note that the example above may violate the recommendation later in this chapter, “Anti-pattern: Reflective tests” on page 37.)

**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 code-base. 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.

**Practice: Prune 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: 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.

Implementing a balanced test suite

There is a variety of tooling available to implement automated infrastructure testing. In many cases, tools designed for software testing can be directly adopted and applied to infrastructure. Some of these tools have been extended to add infrastructure-specific functionality. For example, Serverspec extends the RSpec Ruby-based testing tool with features for checking server configuration.

Different tools may be needed for different types of testing at different levels of the pyramid. It’s important to avoid getting hung up on the tooling. Avoid choosing a tool and basing your testing strategy around it. Instead, analyze your systems and components to decide how you need to approach testing them. Then find tools to carry out your approach. As with any part of your infrastructure, assume that you will change parts of your test tooling over time.

The following diagram gives an example of a service to be tested, and gives a rough idea for the levels of testing that may be appropriate.
Our example service has two types of servers, a database server and an application server. Each has a Puppet manifest to configure it, a Packer template to build AMIs for those two server roles, and a CloudFormation definition file that defines the surrounding infrastructure for the service.

- Application server Puppet manifest
- Database server Puppet manifest
- Application server Packer template
- Database server Packer template
- CloudFormation file

These will be used to illustrate an automated testing suite.

**Low-level testing**

Each of the definition file types listed above is managed in a VCS, and is independently testable. When a change is committed, the CI server pulls the relevant files from the VCS and runs some validation on it.

In our example, a change to either one of the Puppet manifests triggers some validation for the manifest:

- Syntax checking
• Static analysis
• Unit testing

All of these tests can be run quickly, because they don’t have complex setup requirements. They don’t require actually creating a server and applying the manifest. They tend to find simple mistakes very quickly.

Without this layer of tests, a simple syntax mistake might not be found until several minutes have been spent creating a virtual machine and fully configuring it. And at that point, tracing a problem down to the specific error can take a while. It’s much nicer to have a quick check that immediately tells you about the error.

Syntax checking

Many tools that use configuration definition files, like Puppet, Packer, and the command line tool for CloudFormation, have a syntax checking mode that can be used to implement this kind of check.

Example 2-2. Running Puppet’s syntax checking

$ puppet parser validate appserver.pp

Static code analysis

Static code analysis tools work by parsing code or definition files without executing them. Lint is the archetypal static analysis tool, originally written to analyze C code. Static code analysis tools are available for a growing number of infrastructure definition file formats, like puppet-lint and Foodcritic for Chef.

Static analysis can be used to check for common errors and bad habits which, while syntactically correct, can lead to bugs, security holes, performance issues, or just code that is difficult to understand.

Many of the things that static analysis tools check for can seem trivial or pedantic. But the most effective infrastructure and development teams tend to be rigorous about good coding hygiene. Forcing yourself to change your coding habits to keep definitions and scripts consistent and clean results in more reliable systems. Those small bits of messy code and trivial bugs accumulate and create unreliable infrastructure and services.

Unit testing

A unit test in software code executes a small subsection of a program, on the order of a one or two classes, to make sure that they run correctly. Most infrastructure tools have some kind of unit testing tool or framework, such as rspec-puppet and Chef-Spec. Saltstack even comes with its own built in unit testing support.
Tools like this allow a particular configuration definition to be run without actually applying it to a server. They usually include functionality to emulate other parts of a system well enough to check that the definition behaves correctly. “Techniques to isolate components for testing” on page 38 discusses some techniques that are useful for ensuring that your definitions, scripts, etc. can be executed independently. This isolation can be challenging. But it is also a key enabler for improving system design through automated testing.

A pitfall with unit testing is that it can lead to reflective testing (as described in “Anti-pattern: Reflective tests” on page 37). Don't fall into the trap of thinking that you must have unit tests covering every line of your configuration definitions. This idea leads to a mass of brittle tests which are time-consuming to maintain, and which have little value for catching problems.

**Mid-level testing**

Once the quicker validation has been run, different pieces can be assembled and tested together. In our example, this might involve building an application server template using the Packer template and the Puppet manifest. The validation process would be to create a server instance using the new template, and then run some tests against it, perhaps using serverspec as in Example 2-3.

![Figure 2-5. Flow for testing our example application server](image)
At this point, each of the pieces involved in building the application server template has been validated on its own. Tests at the middle level should not be proving the correctness of the individual components, but rather should be proving that they work correctly when integrated together.

For example, our Puppet manifest may make assumptions about packages that have been installed by the base operating system. If someone updates the version of the operating system in the template, the packages may change, which may cause problems for the Puppet manifest. This level of testing is the first point where we can catch this kind of error.

Other failures at this stage may come about because we’re now using an actual server instance, rather than having unit testing software emulate the environment the manifest is applied to.

Example 2-3. Serverspec to validate our application runs correctly

```ruby
describe service('our_app') do
  it { should be_running }
end
```

Creating test infrastructure

Tests at this level will need to create test instances of the relevant infrastructure. In our example this is a test server. But other tests may need to test other things. Scripts or definitions which manage network configuration will require the ability to stand up at least emulated pieces of infrastructure.

Ideally this can be done using the same infrastructure platform that the things being tested will actually use. How practical this is will depend on the platform. Cloud style infrastructure allows test tooling to draw from a pool of resources, to spin up servers, networking elements, etc. as needed. For teams using a more limited set of resources, for instance a set of physical hardware dedicated to the project, this requires an investment to ensure there is enough capacity to run this kind of testing.

Test infrastructure should be baselined before each test. If possible, new instances should be provisioned from scratch, using the same tooling and processes that are used to provision production infrastructure. The resources are then disposed of after the test is complete. This requires the infrastructure provisioning process to be quick, so that the testing process doesn’t run overly long.

If testing happens on a fixed set of infrastructure, the instances should be cleared down and reconfigured from scratch. This avoids situations where cruft builds up across test runs, polluting the test environment.
Locally virtualized test infrastructure

In more constrained environments, or just to optimize the time for testing, it’s often useful to run tests on a locally virtualized environment like Vagrant, rather than on the main infrastructure platform. If the CI agents that run tests are built in a way that supports hosting virtual machines, then they can spin up local VMs to validate server configuration.

The important criteria is how useful this is in actually catching problems quickly. In many cases, a Vagrant box running the same OS, built and provisioned using the same tools and definitions, is an accurate enough representation of a VM running on the production infrastructure.

Don’t get hung up on the language

Teams often prefer testing tools written in the same language they are using for their infrastructure management scripts and tooling. This can be sensible when using with the tool for testing involves writing tests directly in the language. It can also be useful when writing test suites often involves writing extensions. For example, knowing (or learning) Ruby is helpful when using RSpec-based tools.

But the implementation language is less important when using tools where tests are defined using a DSL that is more abstracted from it. If the team doesn’t have much exposure to the underlying language, it should be less of a consideration when choosing the tool.

Tooling for managing test infrastructure

There are a variety of tools which make it easier to automatically create server instances, or other infrastructure context, for the purposes of testing. Test Kitchen is designed to make it easy to manage test servers on different infrastructure platforms in order to apply and test configuration. It was written specifically for Chef, but is also used for testing Puppet and other tools.

Higher level tests

The higher levels of the test suite involve testing that multiple elements of the infrastructure work correctly when integrated together. End to end testing for a particular service integrates all of the elements needed to deliver that service. This potentially includes integrating external systems as well (see “Managing external dependencies” on page 40).
The considerations for testing at this level are similar to those for testing at the middle tiers. For example, a test instance of the relevant infrastructure needs to be provisioned, preferably by building it from scratch, or at least reliably setting it to a baseline, before applying the configuration.

For higher level infrastructure, it becomes more difficult to emulate the infrastructure without using the production infrastructure platform. In any case, this is typically the point where accurately reproducing production infrastructure is more important, because this is probably the last stage of testing to be carried out before production.

This level of testing doesn’t tend to need much special tooling that isn’t already used elsewhere. The infrastructure should be provisioned using the same tooling and definitions that are used for the production infrastructure. Testing can often be carried out using a mix of tools as appropriate. The types of tools used for mid-level valida-
tion, like Serverspec, are often appropriate here. Simple scripts using curl can validate that connectivity works correctly.

In some cases, it may be useful to use general purpose Behavior Driven Development (BDD) and UI testing tools like Selenium. This type of testing is particularly useful to test more complex interactions involving a web UI.

Because this tier of testing is the slowest to set up and run, it’s crucial to keep the testing rigorously pruned. Tests at this level should be the minimum needed to find problems that can’t be found through lower level testing.

**Example 2-4. Serverspec to validate connectivity**

```ruby
describe host('dbserver') do
  it { should be_reachable.with( :port => 5432 ) }
end
```

The above example uses a Serverspec that will be executed from the application server. It proves that the database server is reachable from the application server. This will catch any errors with the firewall rule configuration.

**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.
Testing operational quality

People managing projects to develop and deploy software have a bucket of requirements they call Non-Functional Requirements (NFRs), or sometimes Cross-Functional Requirements (CFRs). Performance, availability, and security tend to be swept into this bucket.

These can be described as Operational Qualities. They are things that can’t easily be described using functional terms - take an action, see a result. Operational requirements are only apparent to users and stakeholders when they go wrong. If the system is slow, flaky, or compromised by attackers, people notice.

Automated testing is essential to ensuring operational requirements. Every time a change is made to a system or its infrastructure, it’s important to prove that the change won’t cause operational problems. The team should have targets and thresholds defined which it can write tests against.

Operational testing can take place at multiple tiers of the testing pyramid, although the results at the top tiers are the most important. These tests measure and validate performance, security, and other characteristics of the system when all of the pieces are in place end to end. So this is the level where operational targets are normally set.

However, it can be useful to have operational quality tests at lower levels of the pyramid. Care should be taken to avoid complicating the test suite with low value tests. But when some parts of the system tend to be the weak link in delivering the right level of operational quality, it can be useful to push testing down to that level.

For example, if a particular application tends to be the slowest part of the end to end system, putting application-level tests around it can be useful. Components that the sluggish application integrates with can be mocked or stubbed (as in “Techniques to isolate components for testing” on page 38), so the performance of the application on its own can be measured.

Automated security tests can take the form of static analysis tools that look for common programming errors that lead to vulnerabilities. They can also check for versions of packages or libraries that have known vulnerabilities. There are many tools available for scanning systems and infrastructure for security issues, many of which can be automated and made part of a regular testing suite.

Managing test code

The tooling for writing and running tests should be treated the same as everything else in the infrastructure. Team members need to apply the same time, effort, and discipline to maintaining high quality test code and systems.
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.

??? applies to tests just as it applies to infrastructure definitions. 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.

**Practice: 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.

**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:

**Example 2-5. Simple definition to be tested**

```chef
file '/etc/our_app.yml'
  owner ourapp
  group ourapp
end
```

Now here is a snippet from the earlier Chefspec unit test example:

**Example 2-6. Test that the definition created the file**

```chef
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
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 ourselves. 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.

**Techniques to isolate 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. 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 39). 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. 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 /our_app/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.

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 “The test-writing habit” on page 44 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.

There are a number of strategies for restructuring systems. Refactoring is an approach that prioritizes keeping the system fully working throughout the process of restructuring the internal design of a system.

---

2 Martin Fowler’s bliki Mocks Aren’t Stubs is a useful reference to test doubles, and is where I learned about Gerard Meszaros' book.

3 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.
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.

Improve the tools you use

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!

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. 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.

**Example 2-7. Tests that are too tightly coupled**

```ruby
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:

**Example 2-8. Decoupled tests**

```ruby
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 ???) 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.
Roles and workflow for 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.

On the flip side, not many testers understand infrastructure very well. So as a result, most testing in IT operations tends to be at a fairly high level.

One of the big wins of agile software development was breaking down the silos between developers and testers. Rather than making quality the responsibility of a separate team, developers and testers share ownership. 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.

There is still controversy over what the role of a QA (Quality Analyst) 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.

**Principle: 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.
The test-writing habit

Many teams struggle to make writing tests a routine habit. It’s a lot like exercising, in that it’s unpleasant to do when you aren’t used to it. It’s easy to find excuses to skip doing it, promising yourself you’ll make up for it later. But if you push yourself to do it, over time it becomes more comfortable. Eventually, the habit becomes embedded enough that you feel uncomfortable writing code or configuring systems without writing tests.

The secret truth about TDD and continuous testing is that the goal isn’t the act of testing, or even the correctness of code. Instead, the goal is design. In order to write a test for a piece of code or a system element, that element must be cleanly designed. The need to make code testable shapes the code.

This is why it’s so difficult to add automated testing to a system that was built without it. A system needs to be designed to be testable, and the only way to ensure this is to designed and implement the system in conjunction with implementing its tests. And it so happens that the qualities that make a system testable are qualities that make a system easy to understand and modify. That is, testable code is quality code. Making the code testable makes it clean, simple, focused, and decoupled.

Principle: 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.

The value of a Quality Analyst

A person with deep expertise and enthusiasm for quality assurance is a huge asset to any engineering team. They often act as a champion for testing practices. They help people to think about work that is being planned or discussed.

Team members should consider how to clearly define each task so that it's clear where it begins and ends, and how to know that it's been done correctly. Surprisingly often,
simply having this conversation before starting the work saves an enormous amount of time and wasted effort.

Some quality engineers have expertise with automated testing tools. Their skills are best harnessed by teaching, coaching, and supporting the other team members in using these tools. It’s important to avoid the trap of letting this expert do all of the work of writing the tests. This risks becoming a test silo, and leaves the team disengaged from their own work.

Another strength that many testers bring is a talent for exploratory testing. These people can take any system and immediately discover the gaps and holes.

**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:

- Write a test
- Run the test, see that it fails
- Write the code
- Run the test, see that it passes
- Commit the code and test
- 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 that was struggling to get started with TDD, because they thought they needed to write all of the tests for a component before they started coding the system itself. TDD is not about switching around the phases of a waterfall. It is about writing tests and code together, one small piece at a time.
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.
Previous chapters have discussed Continuous Integration, Continuous Delivery, and automated testing. A change management pipeline pulls these concepts together and puts them into practice. A change management pipeline is a system that orchestrates the progression of a change through a series of stages to prepare and validate it before then applying it to production systems. This extends of the deployment pipeline used for Continuous Delivery\(^1\). An infrastructure team uses a pipeline to automate its process for change management, ensuring changes can be rolled out safely and easily, and in a way that guarantees compliance with governance and quality controls. Continuous Provisioning, applying these ideas to infrastructure.\(^2\) of software to manage changes to infrastructure.

A change management pipeline works by:

- Immediately and thoroughly testing each change to prove whether it is production ready.
- Testing the elements of the system affected by a change progressively. This aligns with the test pyramid discussed in the previous chapter (test-pyramid), an idea which will be illustrated a bit later in this chapter.

---

1 Jez Humble and David Farley explained the idea of a Deployment Pipeline in their book Continuous Delivery, building on experiences from a ThoughtWorks client project. Humble defines a deployment pipeline as an "automated manifestation of your process for getting software from version control into the hands of your users".

2 The folks at Heavy Water have described this nicely.
• Enabling manual validation activities, like exploratory testing, User Acceptance Testing (UAT), and approvals, where appropriate.

• Applying changes to production systems easily, quickly, and with low risk and impact. This happens by making sure that the process for applying the changes is fully automated, and has been thoroughly and accurately tested in upstream environments.

Teams who embrace the pipeline as the way to manage changes to their infrastructure find a number of benefits:

• Their infrastructure management tooling and codebase is always production ready. There is never a situation where extra work is needed - merging, regression testing, “hardening”, etc. - to take work in progress live.

• Delivering changes is nearly painless. Once a change has passed the technical validation stages of the pipeline, they often don’t need technical attention to carry through to production unless there is a problem. There is no need to make technical decisions about how to apply a change, since those decisions have been made, implemented, and tested in earlier stages.

• It’s easier to make changes through the pipeline than any other way. Hacking a change manually - other than to bring up a system that is down - is more work, and scarier, than just pushing it through the pipeline.

• Compliance and governance is easy. The scripts, tools, and configuration for making changes are transparent to reviewers. Logs can be audited to prove what changes were made, when, and by whom. With an automated change management pipeline, a team can prove what process was followed for each and every change. This tends to be stronger than taking peoples’ word that documented manual processes are always followed.

• There can be less bureaucracy around change management. People who might otherwise need to discuss and inspect each change can put their requirements into the automated tooling and tests. They can periodically review the pipeline implementation and logs, and make improvements as needed. Their time and attention goes to the process and tooling, rather than to turning the handle for routine changes.

This chapter explains how to design, build, and use a pipeline for managing changes to infrastructure.

Principles for designing pipelines

There are a few guiding principles for designing an effective change management pipeline.
Consistency across stages

Environments, tooling, processes should be consistent in the essential ways across stages.

For instance, server operating system versions and configuration should be the same across environments. This is easily done by using server templates, and rolling changes to them out across environments using the pipeline.

Whenever there is a failure applying a change to a downstream system, especially production, consider whether upstream systems can be changed to more accurately emulate the conditions which led to the failure. Always be looking for opportunities to catch errors earlier in the pipeline.
Consistent environments does not mean identical environments

One objection to making testing environments consistent with production is the expense. In organizations with hundreds or thousands of servers, or very expensive hardware, it's not practical to duplicate this in each stage of the pipeline, or even in any stage of the pipeline.

However, the point isn't to have the same number or even size of servers in every environment. Here are some guidelines:

- Make sure that the essential characteristics are the same. The OS and version should always be the same. If the production environment uses a more expensive OS variant (“Enterprise” editions, for example), then you either need to bite the bullet and use this in at least one earlier environment, or else make it a priority to ensure you can use on an OS that your organization can afford to use across the board.

- Replicate enough of the production characteristics so that potential issues will be caught. You don’t need to run 50 servers in a load balancing pool for a test environment. But you should at least make sure you are running 2 servers behind a similar load balancer, and exercise it enough to uncover potential problems with state management.

- At least one environment at the early stage should replicate the complexities of production. But it isn’t necessary that every single environment does. For example, the automated deployment test environment described earlier in the simple pipeline may have a load balanced pool of servers, and deploy applications on to separate servers as they are in production. But even later stages like QA and UAT may be able to run all of the applications on a single server. This needs to be carefully considered to ensure you’re exposing all of the potential risks due to the production environment’s architecture.

Often times it’s difficult to replicate essential production characteristics because it involves expensive equipment. If your organization is struggling to justify buying additional devices just for testing, it’s important to take a hard look at its priorities. Having an important system which can’t be properly tested outside of production is unwise. Either spend the money needed for a responsible change management process, or else find solutions that the organization can afford.
Immediate feedback for every change

Make sure the pipeline runs quickly so that you get immediate feedback on each change as it is committed. The pipeline should be the easiest way to get a quick fix through to production. If people feel the need to bypass the pipeline to make a change, it’s a sign that the pipeline needs fundamental design changes.

The pipeline should be triggered to run immediately after every change is committed. This means multiple changes will be continuously running through the pipeline as different team members make different changes.

If the pipeline takes too long for this to be practical, teams may be tempted to run the pipeline, or parts of it, on a periodic schedule - hourly, nightly, etc. Rather than doing this, teams should invest the effort to improve their pipeline until it’s possible for the pipeline to run continuously. This is essential to ensuring fast and continuous feedback on changes.

Reducing the running time of the pipeline may require changing the architecture of the system. See “Pipelines and architecture” on page 69 later in this chapter.

Run automated stages before manual stages

Once you’ve automated the way changes are deployed to a system so it can be done unattended, it becomes cheaper and faster to run a set of automated tests against it than to have skilled humans spend time testing it. So it makes sense to run all of your automated tests first, before handing the system over to humans to spend time on it.

Running all of the automated test stages first makes sure that the basic problems have already been found. When a change has been handed over to humans to test, they can focus their energies on the trickier problems, without getting tripped up by stupid errors.

Get production-like sooner rather than later

Traditional release processes wait to test the software in the most production-like environment until just before the release is applied to production. “Pre-production” or “Staging” is the final check before going live. But errors or problems found at the last minute are the most painful to fix. Fortunately, an automated change management pipeline can turn this around.

An ideal pipeline automatically applies each and every change to the most accurate test environment as it is committed, running automated tests against it. Doing this ensures that the change has been proven to be technically production ready before it is handed over to humans to decide if it’s human-ready.

Doing this may seem counter intuitive. Leaving the comprehensive testing for the last stage makes sense when deploying changes is an expensive, time consuming manual
process. But when deploying changes is painless, then comprehensively testing every change as it's made aligns with the other pipeline design principles: run automated stages before making humans spend time on it, and of give immediate and full feedback on each change as it's made.

---

**DevOops**

I learned the value of managing changes to infrastructure using a pipeline the hard way. I was helping a development team to automate the infrastructure they used for testing and hosting a public-facing web-based application.

The team had a sophisticated CD pipeline that automatically deployed their application to one stage at a time, only deploying to the next stage when the previous stage passed. However, the Chef cookbooks we used to configure the servers were directly applied to all of our servers, in all environments.

We had decided to have chef manage the `/etc/hosts` files on the servers, so it could automatically add and update hostname mappings, in lieu of running a private DNS server.

You can probably see where this is going.

I made a tweak to the cookbook that builds the `/etc/hosts` file and pushed it to our chef-server. Unfortunately, it had a simple error which made the file completely invalid. Once my broken cookbook was applied to our servers, the servers were unable to resolve the name of the chef-server itself. I fixed my mistake and pushed a new version of the cookbook to the chef-server, but none of the servers could get the updated cookbook.

Work ground to a halt for the development team, since all of the environments across the board were broken. Fortunately we were not yet live, so there was no customer impact. But I had to endure the glares of my teammates while I logged into each and every server, one by one, to resolve the issue.

My colleague Chris Bird described this as “DevOops”: The ability to automatically configure many machines at once gives us the ability to automatically break many machines at once.

Ever since this misadventure, I’ve made sure to have changes to my automated configuration automatically applied to a test environment before they’re applied to any environment that people rely on for something important.

---

3. Actually, I was able to use Chef’s knife tool to make the fix across our servers fairly quickly. But I was conscious of how easily this, too, could backfire.
Basic pipeline designs

The diagram below shows the shape of a simple change management pipeline. Each change is progressively tested in automated stages, and then made available for manual testing and signoff.

![Diagram of basic pipeline](image)

*Figure 3-1. Basic pipeline*

The following sections drill into what typically happens in different types of pipeline stages.

**The build stage**

The build stage (sometimes called the CI stage)\(^3\) is the first stage triggered after a change is committed to the VCS. It runs on a build agent, which is a server instance managed by the CI or CD system. This agent will first checkout the relevant file set from the VCS onto the local file system. It then carries out validation on that code, prepares it for use, and then makes it available to be applied to other environments.

Actions that may happen in the build stage may include some of the following:

- Syntax checking and static analysis. See “Low-level testing” on page 29 in Chapter 2 for details on these activities.
- Compiling code, where relevant
- Unit tests
- Local execution tests. For software components and services, it may be appropriate to run and test it, even though it’s not currently in a full deployment environ-

---

\(^3\) The reason the build stage is called the CI stage is that it encapsulates the early approaches to Continuous Integration. These tended to be focused on compiling software code and creating a build artifact. Continuous Delivery expanded the CI concept to deploying and testing software all the way through to production. This meant adding additional stages and linking them together in a pipeline. Many of the tools used to implement CD pipelines were originally designed for the single stage, CI use case.
ment. This will normally use test doubles (as described in “Test doubles” on page 39) rather than integrating with other systems or components, to keep this stage self-contained. In some cases, containerization (such as Docker) may help with this.

- Generating and publishing reports, such as code change reports, test results, generated documentation, and code coverage reports.
- Packaging code into a distributable and deployable format

As with any stage of the pipeline, if one of these activities fails, then the stage should go red, and refuse to progress the changes downstream. The results, such as test reports, should be made easily available either way, so they can be used to diagnose issues.

The output of this stage is that whatever is being tested - a script, library, application, configuration definition, configuration file, etc. - is now in the format that will be needed to apply it to production.

**Publishing a configuration artifact**

At the end of a successful build stage run, the configuration artifact is packaged and published. This means preparing the the materials and making them available for use in later stages of the pipeline. What this involves depends on the nature of the system component. Examples include:

- Building a system or language installation package file (an RPM, .deb, .gem, etc.) and uploading it to a repository,
- Uploading a set of definition files (Puppet module, Ansible playbook, Chef cookbook, etc.) to a configuration repository,
- Instantiating a server template (VM template, server image, AMI, etc.) in an infrastructure platform.

The classic software artifact bundles configurations into a single file, but this isn’t necessarily the case for infrastructure artifacts. Most server configuration tools use a set of files which are grouped and versioned as a unit, such as a Chef cookbook, but they tend not to be packed into a single distributable file. They may be uploaded to a configuration repository which understands their format and versioning. Some teams eschew this functionality and do package the files, either into a system package or general purpose archive format (as mentioned in ??? in ???).

But the important thing is in how the artifact is treated conceptually. A configuration artifact is an atomic, versioned collection of materials to provision and/or configure a system component. An artifact is:
• Atomic, in that a given set of materials are assembled, tested, and applied together as a unit,

• Portable, so that it can be progressed through the pipeline, and different versions can be applied to different environments or instances,

• Versioned, so that it can be reliably and repeatably applied to any environment, and so that any given environment has an unambiguous version of the component,

• Complete, a given artifact should have everything needed to provision or configure the relevant component. It should not assume that previous versions of the component artifacts have been applied before,

• Consistent, applying the artifact to any two component instances should have the same results.

### Table 3-1. Examples of infrastructure artifacts

<table>
<thead>
<tr>
<th>Artifact</th>
<th>Details</th>
<th>How it can be managed as an artifact</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configuration</td>
<td>Cookbooks, Manifests, Playbooks, etc., that are downloaded to local</td>
<td>The manifest files can be bundled into an archive (.tgz, .zip, .rpm, .deb), and uploaded to a file server, web server, object storage service, package repository, etc. The artifact serving is treated as a static object.</td>
</tr>
<tr>
<td>definitions</td>
<td>machines and run in a local mode, rather than being pulled from a master configuration server.</td>
<td></td>
</tr>
<tr>
<td>Masterless</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Configuration</td>
<td>Cookbooks, Manifests, Playbooks, etc., that are uploaded to a configuration master like a Chef Server, Puppet Master, Ansible Tower, etc.</td>
<td>The master server may allow a set of definitions to be tagged with a version. For example, Chef Server has “Environments”, so a set of Cookbook versions can be copied from one Environment to the next. Alternatively, multiple master servers can be used, so each stage of the pipeline has its own master server. Definitions are uploaded to each master in turn as they progress through the pipeline.</td>
</tr>
<tr>
<td>definitions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Master</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Server templates</td>
<td>The build stage uses a server template definition file, such as a Packer template, to build a server image, such as an AMI or VMWare Template. This image is normally managed by the infrastructure platform.</td>
<td>The server image can be used as the artifact. It should be versioned, either in metadata, if possible, or by including a version number in the name of the template. In some cases where server templates can’t be shared between environments, it may be necessary to store the definition file with a version number, so it can be used to build a server image in a new environment further down the pipeline. For example, A different set of VMWare servers may be used between development and production environments.</td>
</tr>
</tbody>
</table>

### Automated testing stages

Once the build stage has passed and the configurations have been published, they can be applied and tested in more realistic environments. The specifics will depend on what is being tested - some of the examples later in this chapter should illustrate this.
Automated testing may take place across multiple pipeline stages. This can be useful when different types of setup is needed for different test cases. For example, one stage might test an application server role on its own. Tests would not use a database (the database could be stubbed or mocked, as per “Test doubles” on page 39), or anything else outside of the application server itself. A following test stage could then test the application server integrated with other infrastructure elements, including the database server.

Multiple stages should involve progressively widening scope of the infrastructure and systems being tested. This follows the test pyramid, as illustrated by Figure 3-2.

Figure 3-2. The relationship between the test pyramid and a pipeline

Progressing the testing scope this way has a few benefits. Firstly, the wider scoped tests normally take longer to run, since there are more things to set up, and the tests tend to take longer to run. So if the earlier tests fail, the pipeline stops and gives feedback about the error more quickly, rather than waiting for the longer tests to run first.

Secondly, it’s easier to find the cause of a failure in the narrower-scoped tests. If a test fails in the broader test suite, there are more elements which may have contributed to the failure. If the stage that tests the application server on its own passes, you have confidence that it is being built correctly. If tests fail at the next stage, with the application server and other elements, then you can guess that the problem is caused by the integration of the components, rather than the components themselves. (Of course, this assumes that all of the other components have themselves been tested, as described in the “Pattern: Fan-in pipelines” on page 61)

Manual validation stages

Once the automated testing stages have all been applied, the configuration set can be made available for manual testing stages. These stages are normally manually triggered, but automatically deployed. That is, a person will go to the pipeline orchestration tool and cause it to run the stage. But the actual deployment will be carried out
automatically, using the same tools and processes that were used to deploy to upstream environments.

Unlike with the automated testing stages, the sequence of manual stages will tend to be based on organizational process. Typically, testing for quality assurance happens in the first instance, with activities for demonstrating and signing off happening afterwards. There may also be stages to deploy to externally visible environments, for example beta testing or closed previews.

Keeping the pipeline moving

Don't allow work to become bottlenecked by sharing environments or stages. Often teams will try to use a test environment for both QA and demonstrations to stakeholders. This usually leads to QA work having to stop while the environment is carefully prepared and then frozen in the leadup to a demonstration. New changes can’t be deployed to the environment, possibly for days at a time, for fear that something will break the system shortly before the demo.

In these cases it’s far better to split the environments. QA work should have at least one environment, depending on how many testers are working simultaneously. Demonstrations should have their own environments. This is a huge benefit of being able to dynamically provision environments. In many cases, environments can be provisioned on demand for a demo or testing exercise, and then destroyed afterwards.

Apply to live

The final stages in a pipeline is normally the production deployment stage. This should be a trivial task, since it carries out exactly the same activities, using exactly the same tools and processes, as have been applied many times in upstream stages. Any significant risk or uncertainty at this stage should be modeled and addressed in upstream stages.

The techniques described in ??? (???) can also help to manage the risks of releasing changes to production systems, and avoiding interrupting services.

The pipeline stage to release to production may involve governance controls. For example, in some situation legal or compliance rules require specific individuals, or multiple individuals, to authorize changes to production systems. The production stage can use authentication and authorization controls to enforce these requirements.
The rhythm of the pipeline

A commit will run through all of the automated stages of a pipeline, assuming it doesn’t fail any of the stages. The manual stages tend to be run less often. The diagram below shows a typical series of commits to a component managed by a pipeline.

![Diagram of pipeline stages](image)

Figure 3-3. Commits reach later stages of the pipeline less often

Several commits (1, 2, and 3) run through the automated stages without being manually tested. When a tester is ready, she triggers the QA stage to apply the latest change (commit 4) to the QA environment.

In the meantime, the team commits a few more changes (5 and 6). The tester may report an issue, and a fix is committed (7) and then pulled onto the QA environment. After this tested, one more change is committed (8), tested by the tester, and then deployed to the Demo environment.

The cycle is repeated a few more times, with commits 9 through 12 all running through the automated stages. Changes 10 and 12 are tested in the QA stage, with change 12 making it through to staging, and finally to production.

The point of this example is that the earlier stages of the pipeline will run more often than the later stages.
Practices for using a pipeline

The previous section described a basic pipeline. The next section will expand this into more complex situations. However, before doing this, it’s useful to consider some key practices for teams working with pipelines.

Every change is intended to be production ready

For every commit, assume the change could be put through to production if it passes the automated checks. Don’t assume you will be able to finish it off, clean it up, etc. So don’t make a change that you would be alarmed to find in production.

This can be an issue because multiple people may be making changes to the same component in the VCS during the same period. One person may commit a change that they know will need some more work before it should go to production. In the meantime, someone else may make a change that they send straight through to production, taking the first person’s unfinished work with it.

The classic technique for managing this situation is to have people work on different branches, and merge each one to trunk as it’s ready. As described in Chapter 1 (“Continuous Integration (CI)” on page 9), there are a number of techniques which can help to avoid the need for branching this way.

Some changes can be structured in a way so that they won’t actually take effect even when rolled out to production. For example, a new script to rotate logs can be committed and tested in a series of incremental changes, but not enabled in cron until it’s complete and fully tested.

Feature toggles[^3] are a technique to do this when make changes to an existing script. The new changes are added with some conditional logic, so that they’re only used when a configuration setting is “toggled”. For software or tools that use a configuration registry, different environments can have the feature toggled on in test environments, and off for production.

Clean up unused toggles

A common problem with using feature toggles and similar techniques is that it complicates the codebase of the system. Over time, this tends to accumulate a large amount of conditional code and configuration options which aren’t needed any more.

It’s a trap to think that removing feature toggles is unnecessary work, or that configuration options should be left in “just in case”. These add complexity, which increase the time to understand the system and debug errors. Defects love to hide in code that we assume isn’t used.

An optional feature that is no longer used, or whose development has been stopped, is technical debt. It should be pruned ruthlessly. Even if you decide later on that you need that code, it’s should be in the history of the VCS. If, in the future, we decide we want to go back and dust it off, we’ve got it in version control.

Don’t make changes outside the pipeline

Occasionally, when a change fails in a downstream stage, it’s tempting to make a fix directly there, perhaps editing the files where they’ve been published in the artifact repository. However, this is a dangerous practice, since the new edit won’t have been tested in the upstream stages, so may add problems that won’t be caught until too late.

Instead, the fix should be made in the source and committed to the VCS. A new run of the pipeline should start, from the beginning, to fully test the fix. The same rule applies when something breaks because of the tooling or the process that applies the change. Fix the tool or process, commit the necessary changes, and run the pipeline through from the start. If your pipeline is fast enough, then this should not be painful.

Stop the line on any failure

If a stage fails, everyone committing changes to components upstream from that stage should stop committing.

This avoids situations where a fix to a later environment causes a problem that would have been caught by tests in earlier stages of the pipeline. It also ensures that the fix is captured and included in future releases. Manual fixes in later environments have a habit of either becoming forgotten, or being added to a checklist of manual steps which erode the effectiveness of the automation.
So stop the line whenever there is a failure. Make sure you know who is working to fix it, and support them if they need it. If you can't come up with a fix right away, then roll back the change to get the pipeline green again.

**Emergency fixes**

Emergency fixes should also be made using the pipeline. If the pipeline isn't fast or reliable enough, then improve it until it is.

In a critical outage, it might be necessary to figure out what the fix is by working directly on a production system. But once the fix is identified, it should be made in the VCS and rolled out through the pipeline. If you neglect to push the fix through properly, most likely your fix will be reverted when a future change, using the unfixed original source, is pushed through the pipeline.

---

**Scaling pipelines to more complex systems**

The pipeline designs we've seen so far in this chapter have been relevant for a single component. In reality, most systems involve multiple components which may be committed and built separately, and are then integrated together. Quite often, a component, such as a service, is composed of several smaller, separately built components. In other cases, the components are deployed and managed independently, but have run-time dependencies that may be useful to test ahead of releases to production.

There are a number of design patterns that can be used to build pipelines for these more complex scenarios. The appropriate pattern depends on the system and components that the pipeline manages.

**Pattern: Fan-in pipelines**

The fan-in pattern is a common one, useful for building a system that is composed of multiple components. Each component starts out with its own pipeline to build and test it in isolation. Then the component pipelines are joined so that the components are tested together. A system with multiple layers of components may have multiple joins.

As an example, let's consider the kinds of environments described in ???, such as ???, reproduced and simplified here.
This environment is composed of several components. These include three infrastructure stacks, each defined in a Terraform definition file: one for the global infrastructure, and one each for web server and application server infrastructure. (The ideas behind this design are laid out in ??? in ???). There are also two different server templates, one for web servers, and one for application servers, each defined in a Packer template.

A pipeline to test this stack starts with feeder branches for each component. Figure 3-5 shows the section of the pipeline that builds the web server parts of the stack. This has two inputs, the first being a Packer template defining the web server template, the second being a Terraform template defining the infrastructure for the web server tier of the environment stack.

Changes to either of these components starts the pipeline for that piece. A change to the Packer template in the VCS kicks off the top branch of the pipeline. The first stage is a build stage, which may do some static analysis of the template file, before packaging the server template. If our example is using AWS, the server template would be an AMI image.
The second stage runs automatically after the build stage. This stage uses the newly created AMI image to build a server instance, and then tests it, perhaps using server-spec to validate the server’s contents, and an HTTP-client based testing tool to check that the web server is running and correctly serving test content. Once the tests have passed, the server instance is destroyed.

Similarly, if the Terraform template file is changed and committed to VCS, then that part of the pipeline runs. This is simply a build stage which may do some syntax checking before pushing the Terraform file to a central location, with a version number.

When either of the feeder branches - the Packer template or the Terraform template - completes successfully, it then runs the joining stage. This uses the Terraform template to provision the infrastructure of a web server tier, using the last AMI image built and tested by the Packer branch.

This is similar to the Packer stage which created and tested a server instance, but this stage also creates the rest of the infrastructure for the web server tier. This includes networking - firewall rules and a load balancer VIP - and mechanisms for automatically managing the sizing of the pool of web servers. Building this stack component in AWS involves a security group, ELB, and Autoscaling group.
This joining stage will then run tests against the web tier it created. This shouldn’t test things which could have been discovered in previous stages. For example, it doesn’t need to check that the web server is installed and working correctly, because this was done in the feeder branch for the web server AMI. It would test routing and firewall rules, to make sure the correct ports are open. It could also test the automated scaling, by generating the conditions that trigger automatically creating and removing servers from the tier. Once the testing is complete, the web tier instance can be destroyed.

![Figure 3-6. Pipeline joining the components of a full service stack](image)

**Practice: Use the last good build from feeder branches**

When a feeder branch runs successfully it triggers the join stage, which uses the new artifact from that branch. For each of the other feeder branches, the join stage should use the artifact from the last successful run of that branch. This ensures that the scope of change that the join stage is testing is limited to just the changes to the one feeder branch, since all of the other inputs are known to have passed the tests.

When changes are made to more than one feeder branch, the join stage should run for each of the branches, rather than being configured to wait until both branches
complete so it can test their changes together. The reason for this is again to limit the scope of changes being tested at once.

As an example, in our example pipeline above someone could commit a change to the Packer template, and before that pipeline runs through someone else could make a change to the Terraform file. Both of the feeder branches would kick off. If the join stage waits until both feeder changes have passed before running, and then fails its tests, then it's unclear which change caused the failure.

On the other hand, if the join stage first runs with the new AMI from the Packer change, using the previous Terraform file, then a failed test is clearly caused by the Packer change. The version of the Terraform file used in the test has already been through the join stage, so is known to be good. But if the Packer change passes the join stage, and the Terraform change then causes it to fail, it's immediately clear where to look for the problem.

**Fan-in and the test pyramid**

The diagram below illustrates that a change management pipeline with fan-in pattern implements the test pyramid model.

![Diagram](image)

*Figure 3-7. A fan-in pipeline is the implementation of a test pyramid*

**Scaling pipelines**

As the size of a system grows, and as the number of people working on it grows, fan-in pipelines create challenges. Joining stages become bottlenecks as the number of inputs grows. More coordination may be needed to avoid conflicts between changes.
to different components, which creates the need to add processes, meetings, and team roles to manage schedules and communications.

Teams should take the pressure to create manual processes and ceremonies for managing changes as a signal that the design of their system, pipeline, and/or organization is not coping with growth. Adding manual processes to work around these limitations avoids addressing the real problems, and adds unnecessary overhead.

In some cases, teams can improve their designs and implementations within the fan-in model, to allow it to handle growth without limiting throughput. In other cases, however, moving away from the fan-in model helps. These alternatives are, arguably, similar to the decision of whether to scale a server up by adding more hardware and optimizing the software it runs, or scale out, by distributing the system across multiple servers.

Measuring pipeline effectiveness: Cycle time

As with any effort to improve a system's performance, the starting point is to measure the existing level of performance. The most effective measurement of a change management pipeline is the cycle time. Cycle time is the time between deciding on the need for a change to seeing that change in production use.

It's important that the starting point for the cycle time is early enough to reflect the full impact of the change management process. The cycle time does not start when a change is committed to trigger the pipeline. The pipeline is only one part of the change management process, and measuring it on its own is likely to suboptimize.

For example, if changes often fail in the pipeline because of conflicts with changes to other components, we could solve this by enforcing a policy that only one change can be committed and run through the pipeline at a time. This would allow us to tune the pipeline implementation so that it runs very quickly. But it would also create a huge backlog of changes waiting to be committed into the pipeline. From the perspective of a user needing a change, the fact that the pipeline completes within 5 minutes 99% of the time doesn't help if 99% of my changes take over a week to get into the pipeline.

This isn't to say that measuring the end to end run time of the pipeline isn't important. However, make sure that improving the full, end to end cycle time of a change from identifying to fulfilling a need is the paramount goal.

Principle: keep pipelines short

The longer the pipeline is, the longer it will take changes to run through. It takes more work to shepherd a change through a longer pipeline, and there are more opportunities for integration errors with other systems. Keeping pipelines as short as possible helps to keep changes running through quickly and smoothly.
Start out with the most minimal pipeline, and see how it works in practice. Resist the temptation to add things into the pipeline that you think you *might* need. Instead, only add things in as you discover real issues that need to be addressed. Always keep an eye out for opportunities to remove things that are no longer relevant.

**Principle: Decouple pipelines**

When separate teams build different components of a system, such as microservices, joining pipeline branches for these components together with the fan-in pattern creates a bottleneck. The teams need to spend more effort on coordinating the way they handle releases, testing, and fixing. This can be fine for a small number of teams, but as the number of teams grows, the overhead grows exponentially.

Decoupling pipelines involves structuring the pipelines so that a change to each component can be released independently. The components may still have dependencies on one another, so they will probably need integration testing. But rather than requiring that the versions of each components released to production are tested and released together, a change to one component may go ahead to production before changes to the second component are released.

Decoupling the release of integrated components is easier said than done. Clearly, simply pushing changes through to production risks breaking things in production. There are a number of techniques and patterns for testing, releasing, and pipeline designs to help. These are discussed in following sections.

**Integration models**

The design and implementation of pipelines for testing how components integrate depends on the relationships between the components, and the relationships between the teams responsible for them. There are several typical situations:

- Single team. One team owns a component or set of components, and is fully responsible for managing changes to those components. In this case, a single pipeline, with fan-in as needed, is often sufficient.

- Group of teams. A group of teams works together on a single system. Different teams own different components or services, which integrate together to form a single system. In this case, a single fan-in pipeline may work up to a point, but as the size of the group and its system grows, decoupling can become necessary.

- Separate teams with high coordination. Each team (which may itself be a group of teams) owns a system, which integrates with systems owned by other teams. A given system may integrate with multiple systems. Each team will have its own pipeline, and manage its releases independently. But they may have a close enough relationship that a given team will adapt its systems and releases to sup-
port the requirements of another team’s processes. This is often seen with different groups within a large company, and with close vendor relationships.

- Separate teams with low coordination. As above, but one of the teams is a vendor with many customers. Their release process is designed to meet the requirements of many teams, with little or no adaptations to the requirements of individual customer teams. “X as a Service” vendors, providing logging, infrastructure, web analytics, etc. tend to use this model.

As with any model, this is a rough approximation. Most teams will have a variety of integration models, as seen in the following diagram.

![Figure 3-8. Integrating multiple systems](image)

Clearly, different approaches are needed to test and releasing changes across integrated components depending on how closely the teams coordinate. As integration testing becomes a bottleneck to the cycle time for changes, it’s worth considering moving towards a different model. The way to decouple component is to decouple teams.
Giving individual teams more autonomy over how they build, test, and release their components empowers them to become more efficient.

**Pipelines and architecture**

The design of your change management pipelines is a manifestation of your system's architecture. Both of these are a manifestation of your team structure. *Conway's Law* describes the relationship between the structure of an organization and their systems:

“Any organization that designs a system (defined more broadly here than just information systems) will inevitably produce a design whose structure is a copy of the organization's communication structure.”

Organizations can take advantage of this to shape their teams, systems, and pipeline to optimize for the outcomes they want. This is sometimes called the Inverse Conway Maneuver. Ensure that the people needed to deliver a given change through to production are all a part of the same team. This may involve restructuring the team, but may also be done by changing the system's design. It can often be achieved by changing the service model, which is the goal of self-service systems.

A key thing to remember is that designs will always evolve. Designing a new system, team structure, and pipeline is not a one-off activity. As you put designs into practice, you will discover new limitations, issues, and opportunities, and will need to continually revise and adapt to meet them.

**Techniques for handling dependencies between components**

There are a number of techniques for a pipeline to ensure that a component works correctly when it integrates with another component not managed by the same pipeline. There are two sides to this problem. Given two integrated components, one provides a service, and the other consumes it. The provider component needs to test that it is providing the service correctly for consumers. And the consumer needs to test that it is consuming the service correctly.

**Pattern: Library dependency**

One way that a component can provide a capability to another component is to work like a library. The consumer pulls a version of the provider and incorporates it into its own artifact, usually in the build stage of a pipeline. A programming library is the obvious example, such as a .jar file for a logging library that is packaged into a .war file for an application.
The important characteristic is that the library component is versioned, and the consumer can choose which version to use. If a newer version of the library is released, the consumer may opt to immediately pull it in, and then run tests on it. However, it has the option to “pin” to an older version of the library.

This gives the consumer team the flexibility to release changes even if they haven’t yet incorporated new, incompatible changes to their provider library. But it creates the risk that important changes, such as security patches, aren’t integrated in a timely way. This is a major source of security vulnerability in IT systems.

For the provider, this pattern gives freedom to release new changes without having to wait for all consumer teams to update their components. But it can result in having many different versions of the component in production, which increases the time and hassle of support.

A common example of this pattern is configuration definition packages, such as Chef Cookbooks, Puppet Modules, and Ansible Playbooks. The provider team can create and publish updates to the module, and consumer teams can then pull and use these.

![Figure 3-9. Decoupled provider and consumer pipelines](image)

In this example, one team provides an Ansible playbook for installing and configuring the nginx web server. Another team owns an application microservice, and uses the Playbook to install nginx on their application server. The team also uses a Jetty Playbook from a third team.

Another example of this pattern is for server templates. One team might create a standard Linux VM template image for the organization. This would have the standard build of a Linux distro, with common configuration and packages for things like shared authentication, network configuration, monitoring, and security hardening.
Other teams can then pull the latest version of this image to build their servers, or perhaps to build role-specific VM images (as in ??? in ??).

### Repository as an artifact

Another way to promote sets of package versions between pipeline stages is on the repository side, using the Repository as Artifact pattern\(^3\). A team using this approach sets up a separate package repository mirror for each environment. The upstream stage updates packages from public repositories and triggers tests. As each pipeline stage passes its tests, the packages are copied to the next stage’s repository mirror. This ensures the consistency of the full set of dependencies.

Repository as an artifact can be implemented with systems like Spacewalk channels, or more simply with things like reposync or even rsync.

### Pattern: Self-provisioned service instance

The library pattern can be adapted for full-blown services. A well-known example of this is the **Relational Database Service** (RDS) offered by AWS. A team can provision complete working database instances for itself, which it can use in a pipeline for a component that uses a database.

As a provider, Amazon releases new database versions, while still making older versions available as “https://aws.amazon.com/rds/previous-generation/[previous generation DB instances]”. This has the same effect as the library pattern, in that the provider can release new versions without waiting for consumer teams to upgrade their own components.

Being a service rather than a library, the provider is able to transparently release minor updates to the service. Amazon can apply security patches to its RDS offering, and new instances created by consumer teams will automatically use the updated version.

The key is for the provider to keep close track of the interface contract, to make sure the service behaves as expected after updates have been applied. Interface contracts are discussed in more detail below (“**Interface contracts and contract testing**” on page 76). An update that changes expected behavior should be offered as a new version of the service, which can be explicitly selected by consumer teams.

This pattern is often useful for infrastructure and platform services such as monitoring, databases, and dashboards. Consumer teams can have their pipelines automati-

---

\(^3\) I learned about the “repository as an artifact” pattern from my colleague Inny So
cally spin up a service instance to test against, and then tear it down again after the testing stage is finished.

**Providing pre-release library builds**

Teams providing libraries often find it’s useful to make pre-release builds available for consumer teams to test against. Betas, snapshots, early releases, etc., give consumers the opportunity to develop and test the use of upcoming features. This gives the provider feedback on their changes, and allows the consumers to be ready to take advantage of the new features as soon as they’re released.

Pre-release builds should be generated by a pipeline stage. This could be an inline stage which produces a pre-release build for every build that passes a certain stage, as in Figure 3-10. Or it could be a fork, where a person selects and publishes a pre-release build, as in Figure 3-11.

![Figure 3-10. Publishing pre-release builds with an inline stage](image)

In the above example, every build that reaches the pre-release stage is published. The build published as a full release at the end of the pipeline will also have been a pre-release build. On the other hand, the build published as a full release build from the forked pipeline below will not necessarily have been previously selected to publish as a pre-release build.
Providing test instances of a service to consumers

Providers of hosted services need to enable consumers to test how their own systems integrate against the provided service. This is useful to consumers for a number of purposes:

- To learn how to correctly develop the integration from their consumer system,
- To test that changes to the consumer system still integrate correctly,
- To test that the consumer system still works after changes to the provider interfaces,
- To test and develop against new consumer functionality before it is released,
- To reproduce production issues for troubleshooting,
- To run demonstrations of the consumer system without affecting production data

The most effective way for a provider to support these is to enable consumers to provision their own instances, as described above ("Pattern: Self-provisioned service instance" on page 71). If consumers can provision and configure instances on-demand, automatically, then they can easily meet their own testing and demonstration needs.

For example, a monitoring service team in a larger organization may make it possible for other teams to spin up and manage their own instances of the monitoring tools.
Consumer teams can use this capability to experiment with new configuration options, without polluting their production monitoring instance. And they can also spin up instances of the monitoring service so they can run reliability tests in their change pipeline, validating that they have the correct monitoring checks to detect and alert on problems.

Providers can supplement test instances by making it possible for consumers to provision pre-release builds of their service. This is the same idea as a pre-release library, but offered as a hosted service.

Data management is a challenge with test instances of a service. It’s important that consumer teams are able to automatically load appropriate data. This could be artificially constructed test data, to enable automated tests. Or it could be a snapshot of production data, useful to reproduce production issues or demonstrate functionality. Service provider teams may need to develop APIs, tooling, or other mechanisms to help consumer teams do this easily.

For services which can’t be self-provisioned, running test instances is more challenging. Configuration and data in a long-lived test instance can become stale over time, and easily drifts so that it is inconsistent with production. This makes testing less accurate. Multi-tenancy test systems are even more likely to become messy. Teams in this situation would do well to move their systems towards single-tenancy, and to provide self-provisioning capabilities to their consumers.

**Using test instances of a service**

Teams consuming a service provided by another team can include stages that integrate with test instances of the provider in their own pipeline.

If the provider offers a test instance with the current version of the service, then each build of the consumer service can be automatically tested against this. This helps to prove the production readiness of the end to end integrated set of systems, from the consumer’s point of view.

Normally, this will occur in the automatically triggered stages. It would run near the end of this part of the pipeline, after the consumer component has been tested against test doubles (as described in “Techniques to isolate components for testing” on page 38 in Chapter 2).
If the provider also deploys pre-release builds of their service to a test instance, then the consumer team can use this as well. Having a pipeline stage that integrates with a pre-release build helps to uncover potential issues with upcoming releases of the provider. This works well in conjunction with version tolerance ("Practice: Version tolerance" on page 77), to ensure that the consumer component will work easily with either the new or old version of the provider. This frees the consumer team from worrying about the provider’s release schedule.

As described above, managing data in test instances can be challenging. Long-running test instances can become polluted with old test data. This complicates automated tests, and can result in test systems which don’t represent the production instances very well.

Automated test suites should set up the test data they need when they run. It may be possible to load snapshots of production data. However, automated tests should not make assumptions about data that they have not explicitly constructed, otherwise they can’t be guaranteed to run consistently.
In some cases it’s possible to run tests against production instances of a third party system. This is obviously fine when the consumer doesn’t alter data in the other system. For example, when you have an external service which provides lookups, such as DNS, there isn’t much call to have a special test server to integrate with. However, this can cause problems if the data being looked up to support an automated test changes in the source system.

Testing while integrating with a production system can also work fine where the data from the consumer test can be easily segregated. For a monitoring service, for example, the test instances of the consumer can be tagged as a separate environment, which is configured as non-critical. This keeps the production system monitoring data clearly segregated from the test system data.

**Interface contracts and contract testing**

A cornerstone of independently deployable components is loose coupling. Ideally, changing one part of the system won’t have any impact on other parts of the system. The way in which one component integrates with another is the interface.

An interface could be a formal API. A server provisioning script may use a monitoring server’s REST interface to add and remove checks for servers. It could be less formal, such as the arguments accepted by a command line tool that is executed in a Bash script. Or it could be a matter of file system locations where a script or tool can expect to find executables, libraries, or configuration files.

One of the most important jobs of a pipeline is to make sure that when a change is made to one (or both!) side of these interfaces, everything still works correctly. If we decouple the pipelines so that the pieces on each side of the interface are released on their own, we run the risk that problems with the integration between these interfaces will only be discovered in production.

Clearly this would be bad. But there are a number of techniques to catch integration errors with independent pipelines.

**Practice: Ensure backwards compatibility of interfaces**

Providers should work hard to avoid making changes that break existing interfaces. Once a release is published and in use, new releases should not change the interfaces used by consumers.

It’s normally easy to add new functionality without breaking interfaces. A command line tool can add a new argument, without changing the behavior of existing arguments.

When improvements are made to the way those existing arguments work, the results should be the same as before. This can be tricky with bugfixes. Ideally, fixing incor-
rect behavior should only make consumers happy, but often what people build on top of something buggy may rely on the incorrect behavior. The provider needs to make a call on this - in some cases, vendors have actually maintained incorrect behavior to avoid breaking their users’ systems.

If there is a need to make drastic changes to an existing interface, it’s often better to create a new interface, leaving the old one in place. The old interface can be deprecated, with warnings to users that they should move to using the new version. This gives consumers time to develop and test their use of the new interface before they switch. Which in turn gives the provider the flexibility to release and iterate on the new functionality without having to wait for all of their consumers to switch.

For example, a team may install a new version of a library or command line tool on a different file system path. User might change environment variables or paths in their scripts in order to use the new version when they’re ready.

**Practice: Decouple deploying from releasing**

Many teams that operate very large scale systems find that testing changes in pre-release environments has low value. It can be difficult to reproduce production conditions. The number, complexity, and versions of components running in production may be large, and constantly changing. The cost to reproduce this can be very high, not just in spending money on infrastructure, but also time and effort.

The biggest cost for an organization where IT systems are both complex and core to their business is the opportunity costs. In order for testing in a pre-release environment to be even moderately viable for catching issues, the pace of change needs to be kept slow. Throttling the pace of IT change can be dangerous for a business that relies on it to remain competitive.

So a common technique is to allow changes to be deployed into production, without necessarily releasing them to end users. New versions of components are put into the production environment and integrated with other components in ways that won’t impact normal operations. This allows them to be tested in true production conditions, before the switch is flipped to put them into active use. They can even be put into use in a drip-feed fashion, measuring their performance and impact before rolling them out to the full user base.

Techniques for doing this are discussed in ??? in ???.

**Practice: Version tolerance**

The flipside of maintaining backwards compatibility for providers is for consumers to ensure version tolerance. A consumer team should ensure that a single build of their system can easily work with different versions of a provider that they integrate with, if it’s a likely situation.
A shell script can check to see which version of an executable is available, and adapt its behavior accordingly. A provisioning script can change the version of the monitoring server's REST API it assumes after querying the API version.

It's ideal to handle this version detecting and switching dynamically. But in some cases the provider doesn't make this possible. In others, a consumer team may want to control which version of the provider is used. For example, they may want to keep using the old version in production until the new version is stabilized, although they still want to be able to test the new version in non-production environments.

Some teams will resort to building different versions of their consumer system or component to integrate with different versions of the provider. But this risks diverging the codebase, which degrades the effectiveness of Continuous Integration (as discussed in “Continuous Integration (CI)” on page 9 in Chapter 1).

Another option is feature toggles, as described earlier (“Every change is intended to be production ready” on page 59). A toggle can be implemented so that the default version of the upstream provider interface to use can be overridden in some environments.

**Practice: Providing test doubles**

Chapter 2 describes using test doubles (“Test doubles” on page 39) to test a component's code without actually integrating with a provider instance. This is particularly useful for components developed by different teams. Early pipelines stages can test the component itself with stubs that emulate the provider's interface. This tends to be quicker and more reliable than integrating with a real provider. Once the component has been validated to work correctly on its own, then it can be deployed to an environment using a real provider instance, to test how they work together.

Providers might consider making stubs or other types of doubles for their systems available for consumers to use in their own testing. Some infrastructure platform API libraries, such as fog.io (mentioned in ??? in ???) include mocks, so that developers writing scripts to use an infrastructure provider can test their code without actually creating infrastructure.

**Practice: Contract tests**

Contract tests are automated tests that checks whether a provider interface behaves as expected. Writing these tests help a provider to think through and clearly define what consumers can expect from their interface - this is an example of how test-driven development (TDD) helps to improve design. The tests themselves can become documentation for consumers to see how to use the system.
Then by running contract tests in the pipeline, the provider team will be alerted if they accidentally make a change that breaks the contract - the expectations consumers have from the interface.

**Practice: Automated reference implementation**

A variation of contract testing is for a provider team to maintain a simple example of a consumer. For example, ??? had an example of a global infrastructure definition file which manages networking elements that can be used by separately defined microservices (???).

The pipeline which tests this global definition can use an example microservice in its testing. A test stage would create an instance of the global infrastructure, and deploy its example microservice into it. Automated tests then exercise the various capabilities provided by the global stack, for example ensuring that traffic is correctly routed to the application.

**Practice: Contract validation tests**

In some cases, it may help teams building and running consumer systems to automatically validate the interfaces of components and systems provided by other teams. The value of this tends to be higher as the two teams involved are less well-connected. The consumer team can check that new versions of the provider meet their own assumptions for how they behave.

When the provider is a hosted service, the consumer may not have visibility of when changes are made to it, so they may want to run these tests more frequently. For example, they might do this whenever they deploy a change to their own system. Consumer teams might even consider putting these tests into monitoring checks.
Provider health checks

When key integration points are managed by other teams, it’s common to find that the availability of the provider is a source of problems for your own system. Unless the provider has a bulletproof track record (few do), and unless it reliably and quickly notifies you when it has issues (not enough do), then it’s worth making sure you have things in place to give you good visibility of their health. Here are a few suggestions:

- Monitoring checks can probe the provider and let you know when it is down, slow, or not returning correct results.
- Pre-deploy checks can check the health of the provider before deploying a change to your own system. Without this, it’s easy to spend a great deal of time investigating deployment problems that you assume were caused by your own changes, when they are actually due to issues with another service.
- In-application checks which detect when a provider is unhealthy. These can create alerts, log errors, generate metrics on reliability, and also enable the application to handle the failure gracefully.

Practice: Consumer driven contract testing (CDC)

A variation on these previous practices is for a provider to run tests provided by consumer teams. These tests are written to formalize the expectations the consumer has for the provider’s interface. The provider runs the tests in a stage of its own pipeline, and fails any build that fails to pass these tests.

A failure of a CDC test tells the provider they need to investigate the nature of the failed expectation. In some cases, the provider will realize they have made an error, and can correct it. In others, they may see that the consumer’s expectations are incorrect, and they can let them know so they can change their own code.

Or the issue may not be as straightforward, so the failed test drives the provider to have a conversation with the consumer team to work out the best solution.

Conclusion

Implementing change management pipelines for infrastructure transforms the way IT systems are managed. When done well, people spend less time and energy on individual changes - planning, designing, scheduling, implementing, testing, validating, and fixing them. Instead, the focus shifts on the processes and tooling that make and test changes.
This transformation helps the team to focus on improving their flow of work. Their goal becomes being able to quickly and reliably make changes to adapt services to the needs and opportunities of the organization. But it's also a major change in the way that people work on their infrastructure. The next chapter takes a look at team workflow with infrastructure as code.