Chapter 10. Continuous Integration

In this chapter, we are going to change direction a little bit. Up until now, this book has provided details on specific tools and technologies that you can learn, all for the purpose of applying them toward network automation. However, it would be improper to assume that network automation is all about shiny new tools—in fact, that's only one piece of the bigger picture.

This chapter is going to instead focus much more on optimizing the processes around network management and operations. Armed with knowledge of the specific tools and technologies mentioned in previous chapters, you can use this chapter as a guide for using those tools to solve the *real*, challenging problems that network operators at any scale are facing. This chapter will answer questions like:

- How can I use network automation to produce a more stable, more available network?
- How can I help the network move as quickly as the rest of the business demands, without compromising on availability?
- What kind of software or tools can I use to help me implement better processes around my network?

Networking touches *every* other area of IT, and any outages, policy changes, or impediments to efficient process will impact any technology connected to the network. In modern times, these impacts are felt by every other technology discipline. This has caused the rest of IT and the business at large to view the network as something that should "get out of the way" and "just work." These days, the network is called upon to be always accessible, and be more flexible at a more rapid pace than ever before, ensuring it support any service or application the business requires.

The reality is there is no magic bullet here; to accomplish these goals takes discipline, and it requires a disruption of your existing processes and communication silos. It also takes a significant amount of work, learning, and new tools. That work may seem like you're just adding more complexity, but it will pay off in the long run by adding both stability and speed to your network operations processes.

One common underlying theme is the removal of humans from the direct control path of the network. You would be right to be skeptical of this idea, since we've talked about automating humans out of a job for a long time. However, removal of humans from direct control is not the same thing as removing humans entirely. Today, humans maintain direct control over the network by forming a manual, human pipeline for making changes to the network, as illustrated in Figure 10-1.

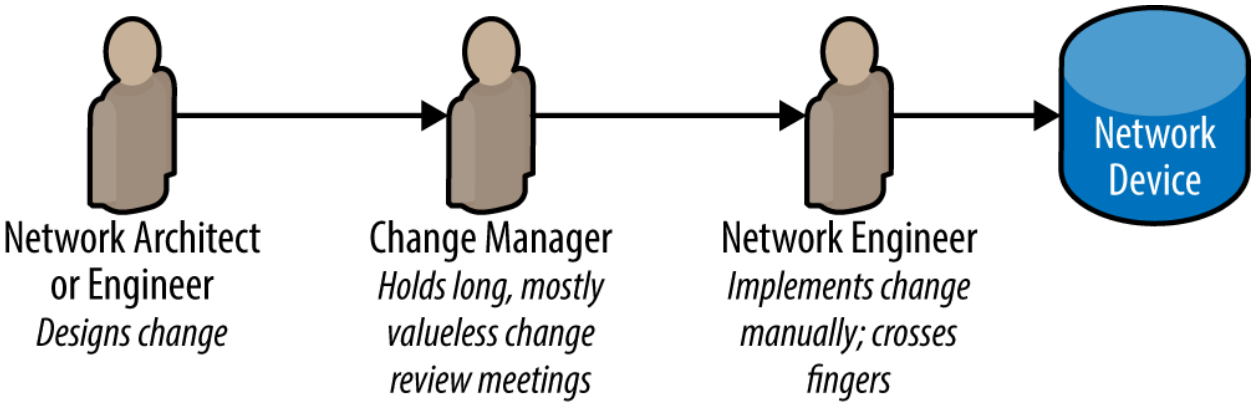

Figure 10-1. Humans in direct path of network

This technique has proven to be slow and arduous, while also not providing much, if any, additional reliability to making changes on the network. This method mostly just gets in the way, while providing the illusion of safety around making changes.

When we talk about removing humans from the direct path, we're talking about Continuous Integration—that is, automating the discrete tasks that should be taking place when we are managing infrastructure change, and freeing technical resources to sit above that pipeline, improving it and making it more efficient (Figure 10-2).

As a result of this fundamental shift toward Continuous Integration, we can actually introduce real protections against human error in network operations instead of the "Change Management Theater" that we've relied on historically.

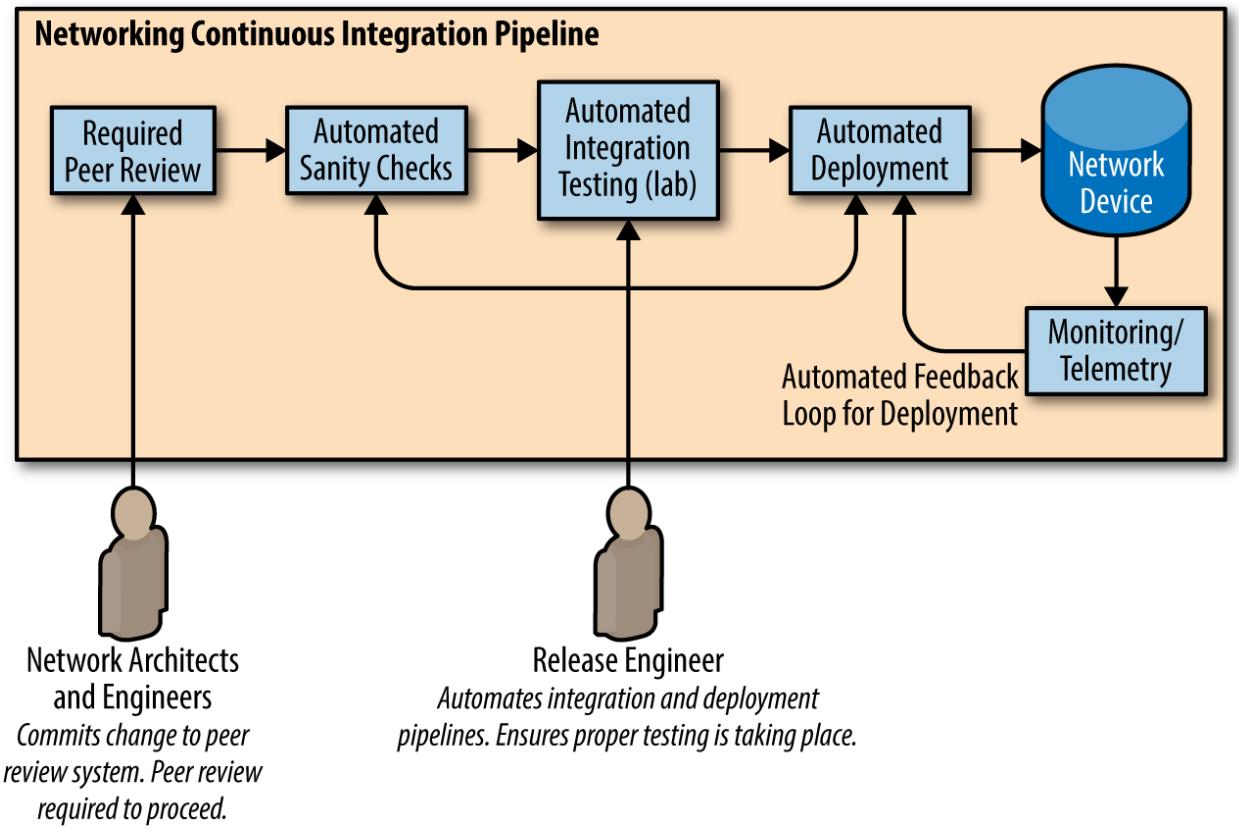

Figure 10-2. Automated change with Continuous Integration

Important Prerequisites

To maximize your success in using the concepts in this chapter, there are a few things to keep in mind.

Simple Is Better

One of the best things you can do to enable your network for network automation has nothing to do with learning to code or using a hot new automation tool—it's all about your network design. Stay away from snowflakes, and strive to deploy network services in a cookie-cutter fashion.

In other words, you may decide you want to deploy network configurations driven by templates. If each of your network devices has a unique configuration with a wide variety of features, it's going to be fairly difficult to build templates for a large group of devices.

NOTE

Network configuration templates were discussed in detail in Chapter 6.

The more thought you put into the consistency of network design, the less work you'll have to do when it comes time to automate the network. Often this means staying away from vendor-specific features, or bypassing embedded features entirely and implementing network services right at the compute layer.

People, Process, and Technology

In the previous chapters, we've discussed a lot of great technologies and tools, but there are a lot more serious challenges facing the network industry today—challenges of process and of working with other IT teams that may not share your primary skill set.

Many of the chapters in this book address specific technologies and tools that you can use to build efficient systems for network automation. There are a multitude of technologies that can be used for purposes of automation—many of which may be new to many network engineers, and it's important to be aware of them. It's also important to improve and change the ways that we communicate with other areas of IT and the business at large—which we discuss in Chapter 11.

In this chapter, however, we're going to discuss some process enhancements that software developers have used for quite some time to improve the way they make changes to applications. The ultimate goal is to make such changes quickly, and push them into production while minimizing the risk of negative impact. There are many important lessons here that can be learned by the network engineering community, especially when considering network automation.

Learn to Code

First, you don't have to be a software developer to leverage the concepts in this chapter. In fact, this chapter primarily exists to convey that message. However, you will likely find that no one tool (or even set of tools) will solve all your problems.

It's likely that you'll have to fill in some gaps in your Continuous Integration journey by writing some sort of custom solution, like a script. Use this as an opportunity to broaden your skill set. As discussed in Chapter 4, Python is a great language to start with, as it's simple enough to learn quickly and robust enough to solve a lot of complex problems.

Introduction to Continuous Integration

Before we dive into how Continuous Integration (CI) is useful within a network automation context, let's first talk about its origins, and how it provides value to software development teams.

First, when we talk about implementing CI, we're looking to accomplish two primary objectives:

Move faster

Be able to respond to the changing needs of the business more quickly.

Improve reliability

Learn from old lessons, and improve quality and stability of the overall system.

Before CI, changes to software were often made in large batches and sometimes it took months for developers to see their features make it into production. This made for incredibly long feedback loops, and if there were any serious issues, or new features/requirements, it took a very long time for issues to be addressed. This inefficiency meant not only that new features took much longer to get developed, but also that software quality suffered.

Naturally, it would be great if developers could simply make changes and push them directly to production, right? It would certainly solve the speed problem—and developers would certainly be able to see the results of their changes more quickly. However, as you might expect, this is incredibly risky. In this model, it's very easy to introduce bugs into production, which could seriously impact the bottom line for many businesses.

Continuous Integration (when combined with Continuous Deployment, which we'll explore later in this chapter) is the best of both worlds. In this model, we're pushing changes to production very quickly—but we're doing so within a context that tests and validates these changes, to be more confident that they're not going to cause problems when they're manifested in production.

In the sections to come, we're going to discuss some of the components of and concepts related to Continuous Integration, and then look at how we can apply these concepts to our network automation journey.

Basics of Continuous Integration

In short, Continuous Integration is all about being able to merge changes to a source code repository at any time. A team of developers, no matter when they're working, can "integrate" changes to some shared repository at any time because there are tools in place that allow the team to know—in an automated fashion—that those changes are not going to break the functionality of the overall system.

You might have heard the term *pipeline* used when discussing CI. This is used because CI is not one particular technology, but usually a suite of different tools and technologies used together to accomplish the goal. Changes to a codebase flow through these tools in some sort of predetermined way, which forms a *CI pipeline*. All changes must go through this pipeline in its entirety before moving on to deployment (Figure 10-3).

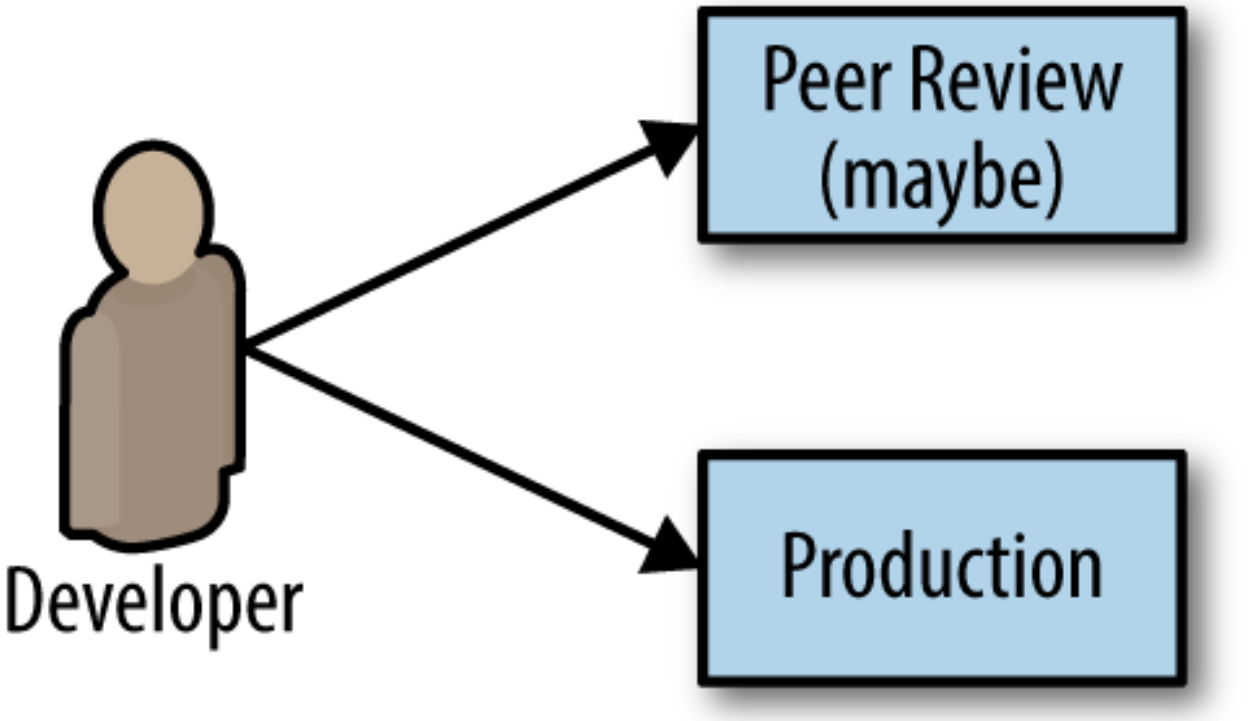

Figure 10-3. Deploying software directly to production

The diagram in Figure 10-3 should look very familiar. Network engineers around the world do this all the time. Maybe they're not deploying software, but what they *are* deploying is a critical change to infrastructure that tends to have a higher "blast radius" than most other infrastructure changes. Logging in to an SSH session to a router to make some config changes is no less risky than editing the source code of an application live in production.

In contrast, Continuous Integration offers only one place where a human can push changes, and that's the CI pipeline—specifically the very first part of the pipeline known as *peer review*. This is the first in a long line of automated steps such as automated testing and sanity checks. In order for changes to make it into production, they *must* go through this pipeline without exception (Figure 10-4).

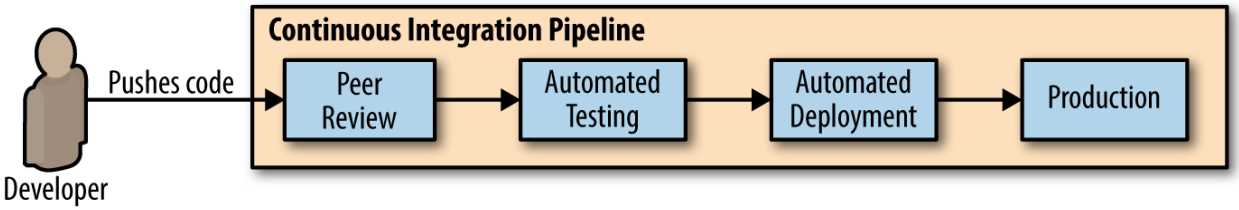

Figure 10-4. Deploying software to a Continuous Integration pipeline

Again, the point of CI is to increase speed while also maintaining (or improving) stability. It does so by creating a single point of entry for making any changes. It becomes nonoptional

to change your infrastructure without subjecting that change to a suite of tests that ensure it undergoes to technical peer review, as well as prevent old mistakes from being repeated.

NOTE

Some organizations hire specialists called *release engineers* to manage the Continuous Integration pipeline. They're skilled with tools like Git, testing tools, build servers, and peer review systems. They maintain the integrity of the pipeline so the developers don't have to. Ultimately, their goal is to automate the process from the laptop to production (thus, "release" engineer).

It's not always easy to get head count for a dedicated release engineer, but if you can, and if your team is large enough to warrant it, it's nice to have.

We've talked about the basics of Continuous Integration, so now let's dive into some of the components and related concepts and technologies you might encounter along the way.

Continuous Delivery

Continuous Delivery (CD) is another term you may have heard as closely related to CI. Continuous Delivery is the idea that the software team is continuously providing software that is able to be deployed into production; they are *delivering* working software in the form of an always-deployable codebase.

NOTE

Continuous Deployment tends to imply that you're always pushing new code to production immediately. The industry has lately been using the term *Continuous Delivery* instead. This term generally means that your code is always in a condition where it *could* be deployed at any time, but doesn't have to be. Your organization may still wish to keep deployments on a set schedule, such as on a nightly or weekly basis.

Continuous Integration is fairly easy to apply to network automation (as we'll see in upcoming sections), but Continuous Delivery requires a bit more thought. The rest of this chapter may blur the lines between CI and CD with respect to network automation, so keep in mind a few things:

- *What* am I deploying?
- *To what/whom* am I deploying it?

These are important questions to address because they determine your delivery model. For instance, some network teams may perform all their automation with in-house Python applications. This is fairly simple since they are essentially a software development shop within the infrastructure team.

On the other hand is the canonical network automation example: provide some kind of configuration artifact (say, some YAML file) into a Git repository, and have the CI/CD pipeline take it through some basic sanity checks before finally calling it with a tool like Ansible, resulting in actual and immediate changes to network devices in production. This may work for some organizations, but this is analogous to a software development team deploying each and every software patch to production immediately—and this is not always desired.

Consider, perhaps, a "staging" environment to which these changes can be continuously delivered, and whenever the business requires that those changes are finally deployed to production, they can be moved from staging, where (hopefully) they've been tested. At the time this chapter was written, many network vendors have heard our demands for providing virtual images of their platforms, so this is much easier to do than it used to be.

NOTE

While all of these virtual appliances work great for testing automation, not all are actually meant to carry production network traffic. Please refer to your vendor's documentation for clarity on this.

You also need to think about rollback procedures. Are you periodically taking the configurations that are in your "production" Git repository and using them to overwrite the current "production" configurations, or at least making comparisons between the two? If you're not, even if you roll back the repository, the production "deployment" of those configurations may not get rolled back. What will be the impact, based on whether you're using Ansible or Puppet, or maybe some custom Python programs, if you roll back the Git repository? You need to own that layer of your software stack and understand how your tools and software will react (if at all) when your production configurations get rolled back.

The truth is, you'll likely have to address the Continuous Delivery question on your own. What works for one organization probably won't work for yours, due to the large number of tools and languages available for solving network automation problems. However, this chapter should at least provide some starting points, and ideas for how to properly deliver changes to your network in an automated fashion.

Test-Driven Development

It's also important to discuss yet another software development paradigm that has seen a growing amount of adoption—Test-Driven Development (TDD).

Let's say you're working as a software developer and you've been tasked with creating a new feature in your software project. Naturally, you might first gather some basic requirements, put together a minimal design, and then move forward with building the feature (Figure 10-5). We'll even say that you're on board with Continuous Integration, so you will then build some unit tests that validate the functionality that you've built.

Unfortunately, it doesn't always happen this way. In reality, building tests after the feature has been built is often difficult to justify, or at the very least, deemed less important than the feature itself.

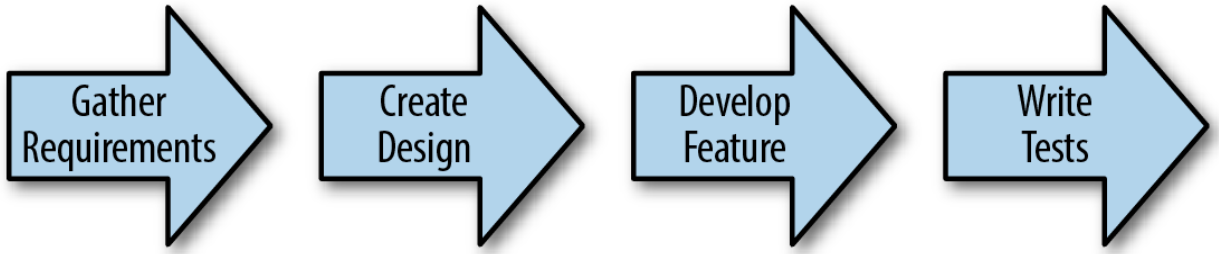

Figure 10-5. Software development life cycle before test-driven development

In practice, this can easily lead to the accumulation of "technical debt." In other words, if you don't build your tests first, there's always a temptation to not build them immediately after you develop the desired feature, or to not build them at all. This inevitably leads to gaps in test coverage, and on large projects, this gap only increases over time.

Test-driven development turns this idea on its head. When using TDD, after going through requirements gathering and putting together a basic design, you would *first* write a test for that feature, before the feature is even implemented (Figure 10-6). Naturally, this means the test would fail, since there's no code to test against. So, the final validation of this feature is to write code that passes that test (or tests).

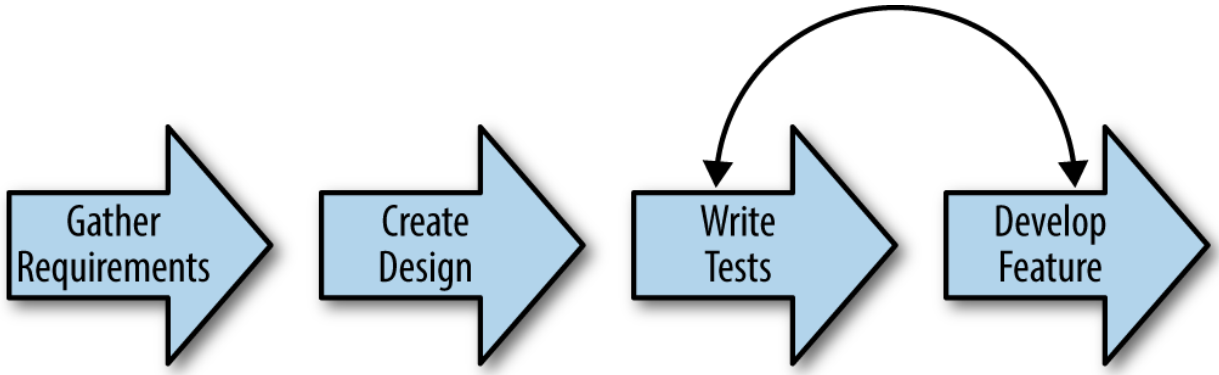

Figure 10-6. Software development life cycle after test-driven development

Why use a test-driven approach? The most immediate benefit is the reduction of technical debt—if the tests are built before the feature, then there's no temptation to let test coverage fall behind while the shiny new features take priority. However, there's also a bit of a conceptual difference. By writing tests first, the developer must have a strong grasp on how their software is being used—since they're writing tests to do just that. This is widely believed to have a positive influence on software quality.

When you apply these concepts to network automation, you begin to realize numerous parallels. The network is as much of a business resource as the applications that flow on

top of it. Therefore, it's important to have adequate testing in place that not only validates any changes made on the network, but also helps warn of any problems ahead of time (capacity planning). Are you gathering detailed statistics about the applications flowing on your network? Not just what the SNMP service on your network devices are telling you—but from the perspective of the applications themselves? It's important as network engineers to learn the lesson that TDD is teaching software developers: the use case matters. User experience matters.

The point of bringing up TDD in this chapter is twofold. We want to remember two things whether leveraging existing network automation tools, or writing our own software:

- We care enough about the quality of our network automation system, our network's uptime, and the positive experience of our users and applications to ensure our system is properly—not minimally—tested.
- Testing is so important that it should be done before you automate a single thing on your network. With the software available today, and the increasingly lower barrier to entry for these tools to nondevelopers, there really is no excuse for not doing this.

NOTE

We'll see a lot more detail about *how* to do this testing in a future section of this chapter, in case those two points seem a little far-fetched. For now, just keep those two points in mind—they are crucial for the success of any network automation effort.

By adopting a methodology similar to TDD, we are not only helping to put the applications first, but we're also building a repeatable process by which we can constantly be *sure* that our network is serving the needs of the application, despite changes to configuration or environment. Later in this chapter we'll discuss some specific tools and technologies that we can use to accomplish these goals.

Why Continuous Integration for Networking?

So far, we've discussed ideas like Continuous Integration as well as test-driven development, and how those concepts have provided value to software development teams. From now on, however, we'll be applying concepts like these exclusively to our network automation journey.

Why are we doing this? What value could Continuous Integration or test-driven development have to network engineers? Remember the goals of Continuous Integration:

Move faster

Be able to respond to the changing needs of the business more quickly

Improve reliability

Learn from old lessons, and improve quality and stability of the overall system

These goals, which have driven results for more stable software and more agile development teams, can also help us to create a *more* reliable network—not less. Automation that compromises on either of these two goals is pointless.

"CI for networking" means a lot of the same things as the canonical software example—creating a single point where changes to network infrastructure are performed, where testing and reviewing those changes is automated and nonoptional.

For a long time, we've thought about and administered our networks as black boxes that happen to be connected to each other, and this mindset isn't very conducive to the practices and concepts implemented in CI. So the first thing to do is start to think about your network as a pool of resources and fluid configurations—a system with ever-changing environments and requirements.

This is the driving idea behind the "infrastructure as code" movement—maintaining the state and configuration of your infrastructure with the same processes developers use to manage source code.

A Continuous Integration Pipeline for Networking

At this point, we've discussed a lot of the high-level concepts and theories behind Continuous Integration, but now it's time to put these concepts into practice. In this section, we'll go through a few practical examples and tools for helping us achieve the goals we've discussed within the context of network automation.

While reading the following examples, keep these tips in mind:

- The tools used in this section are just examples. In every category, there are choices beyond what's presented here. It is encouraged that you evaluate the tools available in each category and determine if they fit your needs.
- This section comes after the previous section for a reason. Implementing these tools without fixing the bad process that has plagued many organizations for years will accomplish nothing.

NOTE

It's also worth noting that these tools can be configured in a variety of ways. Only one approach will be presented in this section, so remember the fundamental concepts here, and adopt the right configuration to realize the same benefits within your organization.

There are five main components to our Continuous Integration pipeline for networking:

- Peer review

- Build automation
- Deployment tools
- Test/dev/staging environment
- Testing tools and test-driven network automation

To illustrate the concepts in this chapter, we'll use a project called Templatizer, which renders Jinja templates into network device configurations based on data found in YAML data files. Many of the examples will center on the Templatizer Git repository hosted on our private Git server.

Figure 10-7 will serve as a useful illustrative example for our Continuous Integration journey.

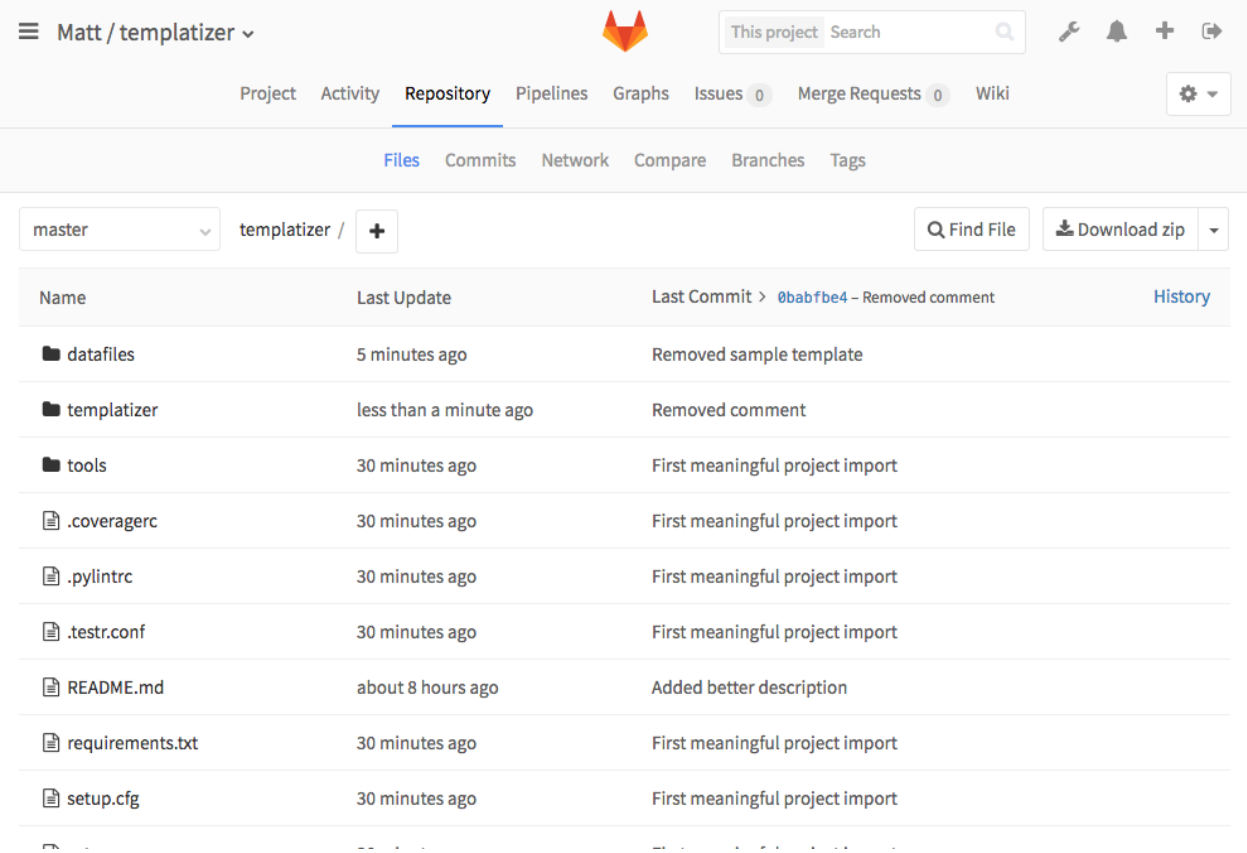

Figure 10-7. Templatizer project

Peer Review

When we talk about peer review in a traditional software sense, we're typically talking about source code for an application. A developer will submit a patch containing some diffs to some source code files, that patch will be posted to the code review system in some way, and a reviewer (or reviewers) will look at the patch and provide comments or approval.

When adapting this portion of the pipeline for network automation, we're not that far off from this example. Several other chapters in this book, like Chapters 5 and 6, have advocated an "infrastructure-as-code" approach with network automation, where any and all relevant configuration information is treated in the same way a developer would treat source code. In our case, instead of Java code, we might have YAML files or Jinja templates. They're all just text files, and we can run automated tests on them just the same.

We're going to be building on the knowledge we gained about version control in Chapter 8 by using Git to not only control the versions of our various configuration artifacts like YAML files, but also leverage the first stage of this pipeline—peer review—to get an additional pair of eyes on our change to make sure we're doing the right thing.

If you've maintained any form of production IT infrastructure, you've likely taken part in Change Approval Board (CAB) meetings. Perhaps you were responsible for filling out a form where you describe the configuration change you want to make, and then attending long conference calls to say a few quick words that were carefully constructed to appease the approvers and get them out of your way. This process has deep roots in modern IT, but it doesn't do much to *actually* minimize risk, or provide transparency between related technical teams. This is the old way of doing things.

When we talk about using Continuous Integration for networking, we start with the idea of peer review, and it might seem similar to what was just described, but there are some fundamental differences. In CI, if you want to make the change, you simply cut a new branch in Git, and *make the change*. By having our configurations performed in a Git repository that is part of a CI pipeline, we don't have to ask for permission before doing the work in a branch, because that work is not actually pushed through production until it has been reviewed and merged to master.

This new model has some very attractive benefits. With respect to peer review, there is now no need to "describe" the change you want to make, and hope you get it right when it comes time to implement—now, the description of the change is the same as the change itself. There is no ambiguity about what you're going to do because it's displayed right in the peer review system being used. In order to put your change into production, the approver(s) will simply merge your working branch into master.

When it comes to code review platforms, we have a few options. Here is a non-exhaustive list:

GitHub

: Very popular SaaS offering for reviewing and displaying source code (enterprise edition also available for a cost).

GitLab

: Community edition is open source and free to download and run behind your firewall. There is also a tiered SaaS offering, as well as a closed-source enterprise edition.

Gerrit

> Open source, complicated, but lots of integrations available, and a popular choice for many open source projects.

All three options leverage Git for the actual version control portion (and Git is therefore the way that you will interface with them when submitting code) but on top of Git, they implement a fairly unique code review workflow. For instance, with GitHub, you can submit additional changes by simply pushing more commits to the same branch, but with Gerrit, the submitter must always work with the same commit (meaning additional changes require the `--amend` flag to be used).

We'll be using GitLab throughout this chapter, primarily because it offers a lot for free, and we don't have to fuss around with setup too much. Know, however, that the other systems may work out better for you.

NOTE

At this point, you should be familiar with not only Jinja templates and YAML files, but also how to work with a Git repository. Our example assumes the Templatizer project has already been cloned to the local filesystem, and we're ready to do some work.

As an illustrative example, we'll add some Jinja templates and YAML files so the Templatizer project is able to create configurations for network device interfaces. We'll start by creating a new Git branch for committing our changes:

```
~$ git checkout -b "add-interface-template"



Switched to a new branch 'add-interface-template'
```

We're on a branch that is only on our machine (we haven't run `git push` yet) and it's on a non-master branch. So we simply make the change. No waiting for approval before we get started—we do the work first, and let the work speak for itself when the time comes for approval.

After we've added the template and YAML file, Git should notify us that two new files are present but untracked:

```
~$ git status



On branch add-interface-template
```

```
Untracked files:

  (use "git add <file>..." to include in what will be committed)



    datafiles/interfaces.yml

    templatizer/templates/interfaces.j2



nothing added to commit but untracked files present (use "git add" to track)
```

We need only make a commit and push to our origin remote, which in this case is the GitLab repository shown earlier:

```
~$ git add datafiles/ templatizer/



~$ git commit -s -m "Added template and datafile for device interfaces"

[add-interface-template 4121bfa] Added template and datafile for device
interfaces

 2 files changed, 10 insertions(+)

 create mode 100644 datafiles/interfaces.yml

 create mode 100644 templatizer/templates/interfaces.j2



~$ git push origin add-interface-template

Counting objects: 7, done.

Delta compression using up to 8 threads.

Compressing objects: 100% (7/7), done.

Writing objects: 100% (7/7), 718 bytes | 0 bytes/s, done.
```

```
Total 7 (delta 2), reused 0 (delta 0)

To http://gitlab/Matt/templatizer.git

 * [new branch]      add-interface-template -> add-interface-template
```

The next step is to log in to our code review system (GitLab) and initiate the step that would kick off a peer review. Every code review system has its own workflow, but ultimately they all accomplish the same thing. For instance, Gerrit uses terminology like *change* and *patchset*, and GitHub uses *pull requests*. In short, these tools are a way of saying, "I have a change, and I'd like it to be merged into the main branch" (usually master).

GitLab uses a concept very similar to GitHub pull requests called *merge requests*. Now that we've pushed our changes to a branch, we can specify in the merge request creation wizard that we'd like to merge the commit we made on "add-interface-template" to the master branch, which is considered "stable" for this project (Figure 10-8).

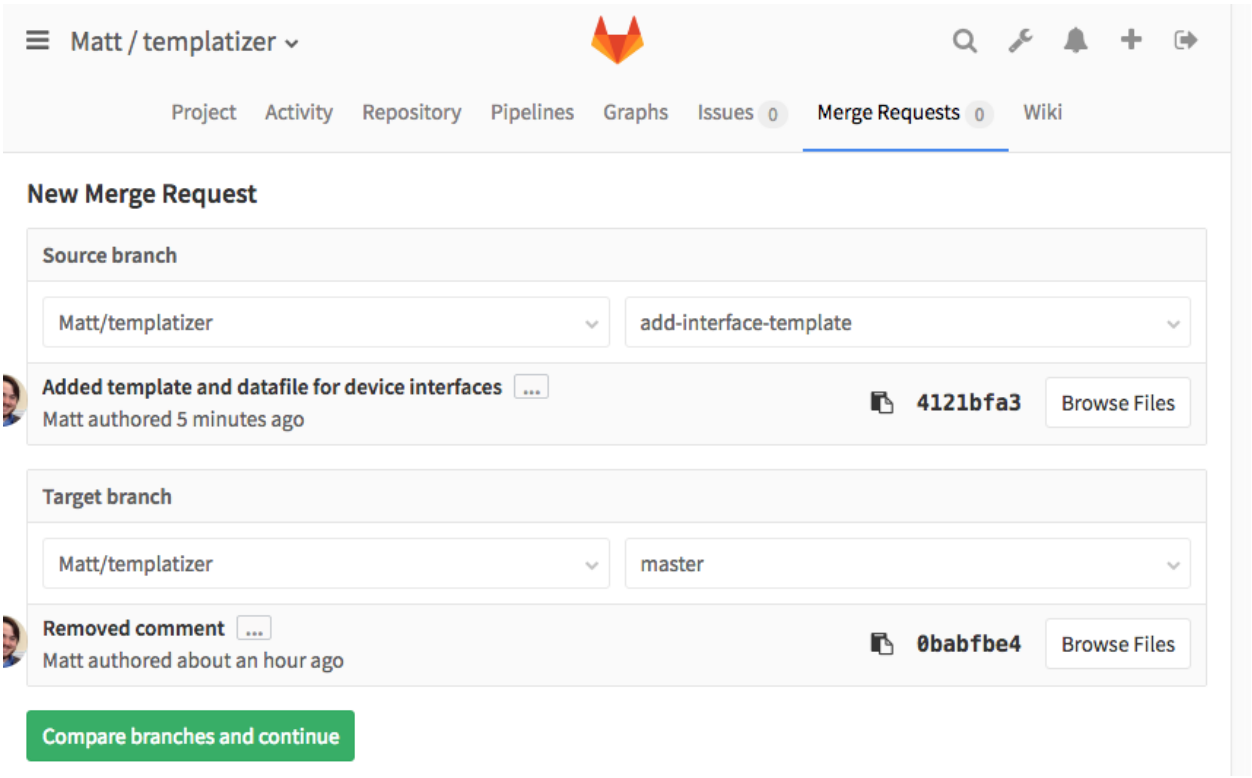

Figure 10-8. Creating a merge request

After we click through to the follow-up confirmation screen, our merge request is created. Keep in mind that this is still just that—a request. There has still been zero impact to the master branch, and as a result, the current "stable" version of the Templatizer project. This is just a proposal that we've made, and will serve as a point of reference for the upcoming peer review.

So the next step is to get our merge request reviewed by someone in authority. This part of the workflow can differ based on the culture of the team, as well as the review platform. Some teams restrict access to the master branch so that only certain senior members can accept merge requests, while other teams use the honor system and ask that each merge request is reviewed by at least one other team member. A common convention is to refrain from merging any changes until someone gives a "+1," which is a way of saying, "From my perspective, this change is ready to be merged." This may happen right away, or a reviewer may have some comments or pointers before they're ready to give their +1.

Our imaginary teammate Fred is on hand to review our Templatizer change, and we can engage him any number of ways. Most code review tools have a way of "adding" a reviewer, which should notify them by email, or you can message them directly. Either way, Figure 10-9 shows what Fred will see when reviewing our change in GitLab.

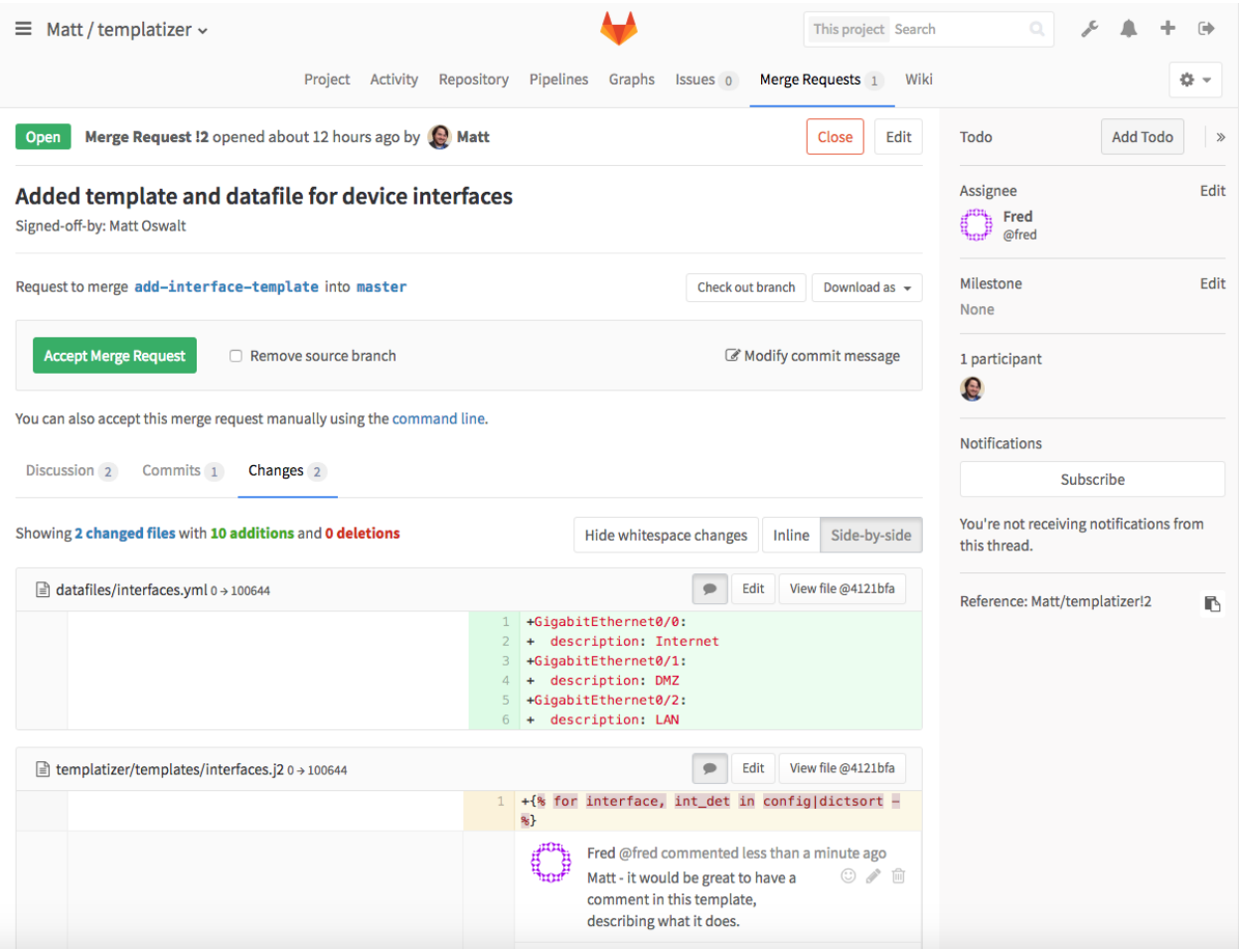

Figure 10-9. Fred providing comments to merge request

As you can see, Fred left a message indicating he felt we should add a comment to our template, explaining how it works. It's not uncommon for a change to go through multiple iterations before being merged, and most platforms have facilities for this. With GitLab, we only need to add another commit to this branch and push to GitLab, and the new commit

will be added to this merge request. Fred can easily see these additional changes, and once he is satisfied that this change is ready to be merged, he can do so.

Figure 10-10 shows us how GitLab can track this entire event stream for anyone that may want to see the status of this change.

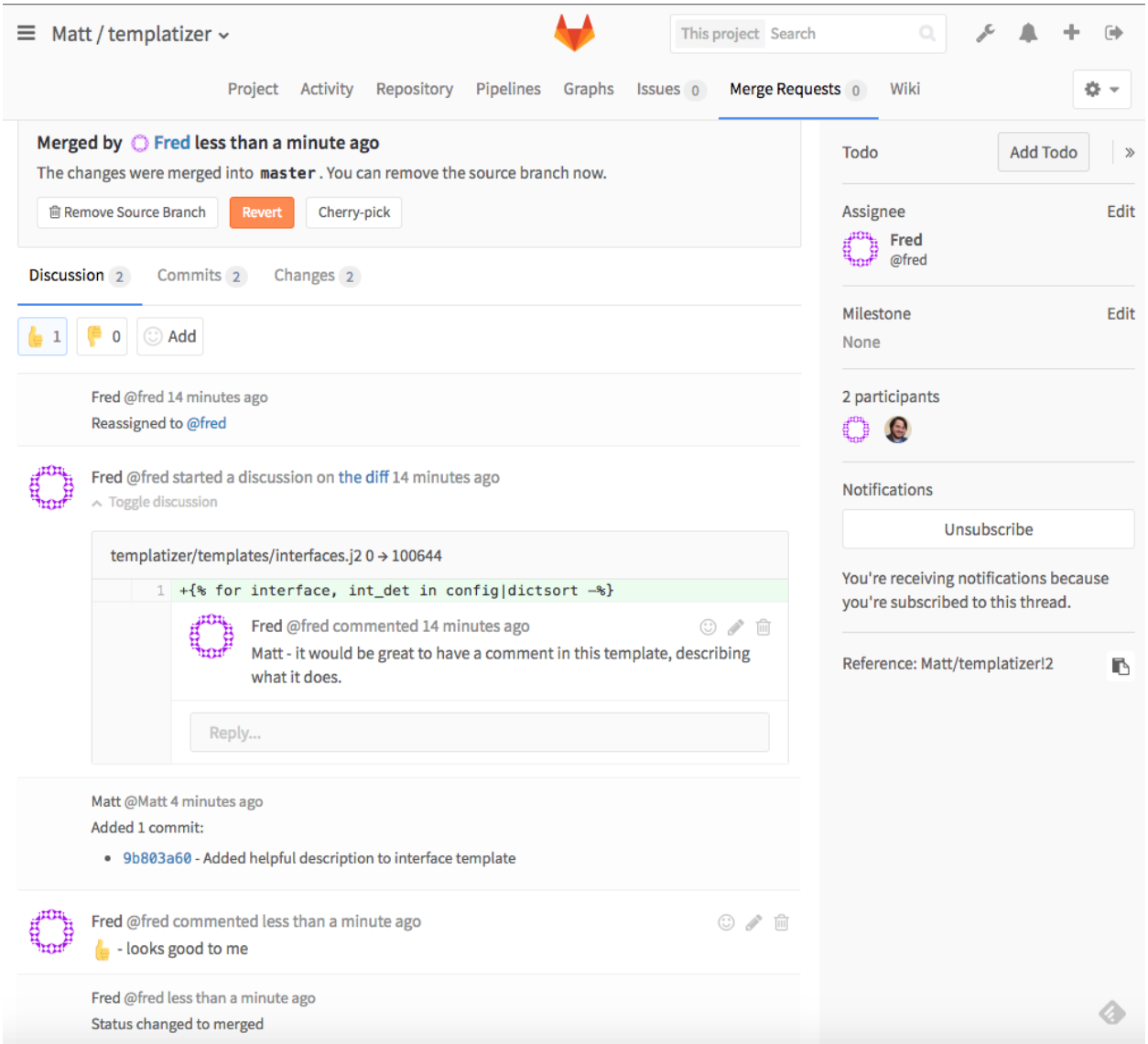

Figure 10-10. Change accepted and merged

Build Automation

Next up is an extremely important topic, *build automation*. This term largely stems from the use of Continuous Integration tooling as a way of automatically compiling or installing software in order to test it. For instance, a program written in C must be compiled before it can be run in a test environment.

We may not necessarily be compiling software in our pipeline, but we can reuse many of the tasks that software developers will want to automatically perform on every proposed change to the repository. For instance, a pipeline for a Python project may perform some static code analysis to ensure the code conforms to Python's style guide, PEP8. In a network automation context, we may only be making changes to YAML files, but we can perform similar checks to automate some of the simple stuff that we don't want human reviewers to deal with, including verifying that the file is in fact valid YAML (ensuring indentation is correct)!

This is the crux of what makes build automation so valuable. Before even bothering a human reviewer, there are a number of things we can automatically do to ensure that the reviewer is providing useful comments:

- Static code analysis (checking for proper syntax and adherence to any style guides)
- Unit testing (unit tests, parsing of data files or templates, etc.)
- Integration testing (does this change break any existing functionality in the whole system?)

With these out of the way, the reviewer can leave comments like "this needs to be more readable," instead of "add a space here." For this reason, these automated steps usually take place immediately when a change is submitted (our merge request from the previous example), and a reviewer is only engaged when these checks pass.

This process saves time for both the submitter and the reviewer, since the submitter gets close to immediate feedback if their change breaks something, and the reviewer knows that if a change passes these basic tests, they won't be wasting their time with simple comments. It also produces repeatable, more stable changes to network automation efforts—when a bug is discovered, it can be added to these automated tests to ensure it doesn't happen again.

As expected, there are a number of platforms that provide this kind of functionality. One very popular choice is Jenkins, which is an open source build server with a multitude of integrations and capabilities. For our example, we're going to stick with GitLab, since it offers all of the features we need for these examples, and they're all bundled into the single piece of software. In addition, much of the actual automation can be done by scripts in the repository itself, keeping the dependence on the actual build server to a minimum, and providing for a lot of transparency for anyone working with the repository.

Let's say we make a minor change to our new *interfaces.yml* file, and Fred reviews it. Everything looks good to him, so he gives a +1 and merges the change to the master branch (Figure 10-11).

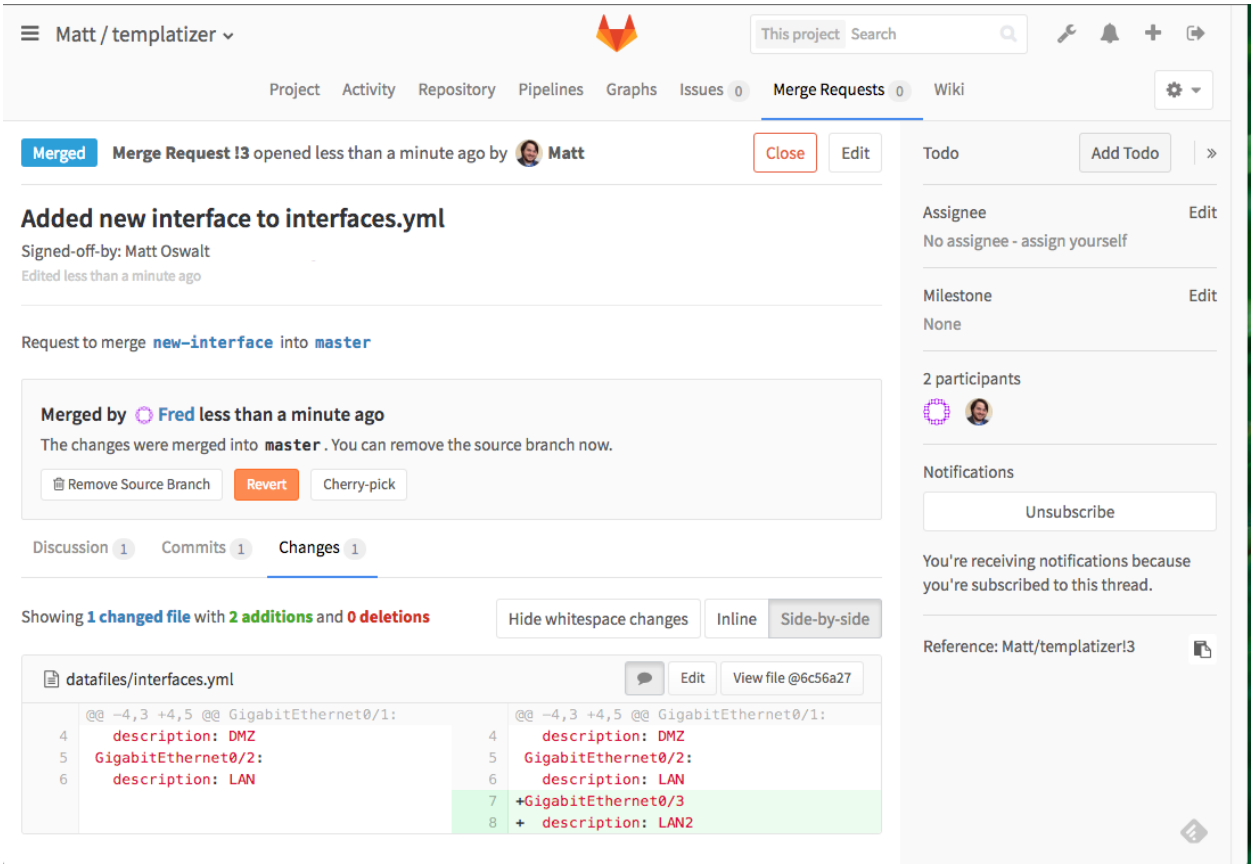

Figure 10-11. Minor change to YAML

However, we have a problem. This change produces invalid YAML, which shows when we try to run our Templatizer program:

```
File "/Users/mierdin/Code/Python/templatizer/lib/python2.7/site-
packages/yaml/scanner.py", line 289,

  in stale_possible_simple_keys "could not found expected ':'",
self.get_mark())

yaml.scanner.ScannerError: while scanning a simple key

  in "datafiles/interfaces.yml", line 7, column 1

could not found expected ':'

  in "datafiles/interfaces.yml", line 8, column 14
```

This was a very minor change, but Fred is still a human being and overlooks typos like this, just as Matt did. In larger changes, where multiple files undergo multiple changes, this can be an even more common occurrence.

On the other hand, it should be trivial to write a script that checks for this error, and provides feedback to our build system. If we can do this, and configure our automated build system to check for this on all future patches, we should avoid this problem in the future:

```
#!/usr/bin/env python

import os
import sys
import yaml

# YAML_DIR is the location of the directory where the YAML files are kept
YAML_DIR = "%s/../datafiles/" % os.path.dirname(os.path.abspath(__file__))

# Let's loop over the YAML files and try to load them
for filename in os.listdir(YAML_DIR):
    yaml_file = "%s%s" % (YAML_DIR, filename)

    if os.path.isfile(yaml_file) and ".yml" in yaml_file:
        try:
            with open(yaml_file) as yamlfile:
                configdata = yaml.load(yamlfile)

        # If there was a problem importing the YAML, we can print
        # an error message, and quit with a non-zero error code
        # (which will trigger our CI system to indicate failure)
        except Exception:
            print("%s failed YAML import" % yaml_file)
            sys.exit(1)

sys.exit(0)
```

NOTE

This is a highly simplified example. There are several libraries that can allow you to do much more detailed validation. Check out `pykwalify` for more detailed YAML validation (not just syntax, but the presence of expected values).

Once we've committed that script to our *tools* directory in the repo, since we're running GitLab we also need to modify the CI configuration file *.gitlab-ci.yml*:

```
test:

  script:

  - cd tools/ && python validate_yaml.py
```

Now, GitLab will run this script every time a change is proposed. Now that this validator script is in place, let's take a look at what Fred sees now, when Matt proposes another change with invalid YAML (Figure 10-12):

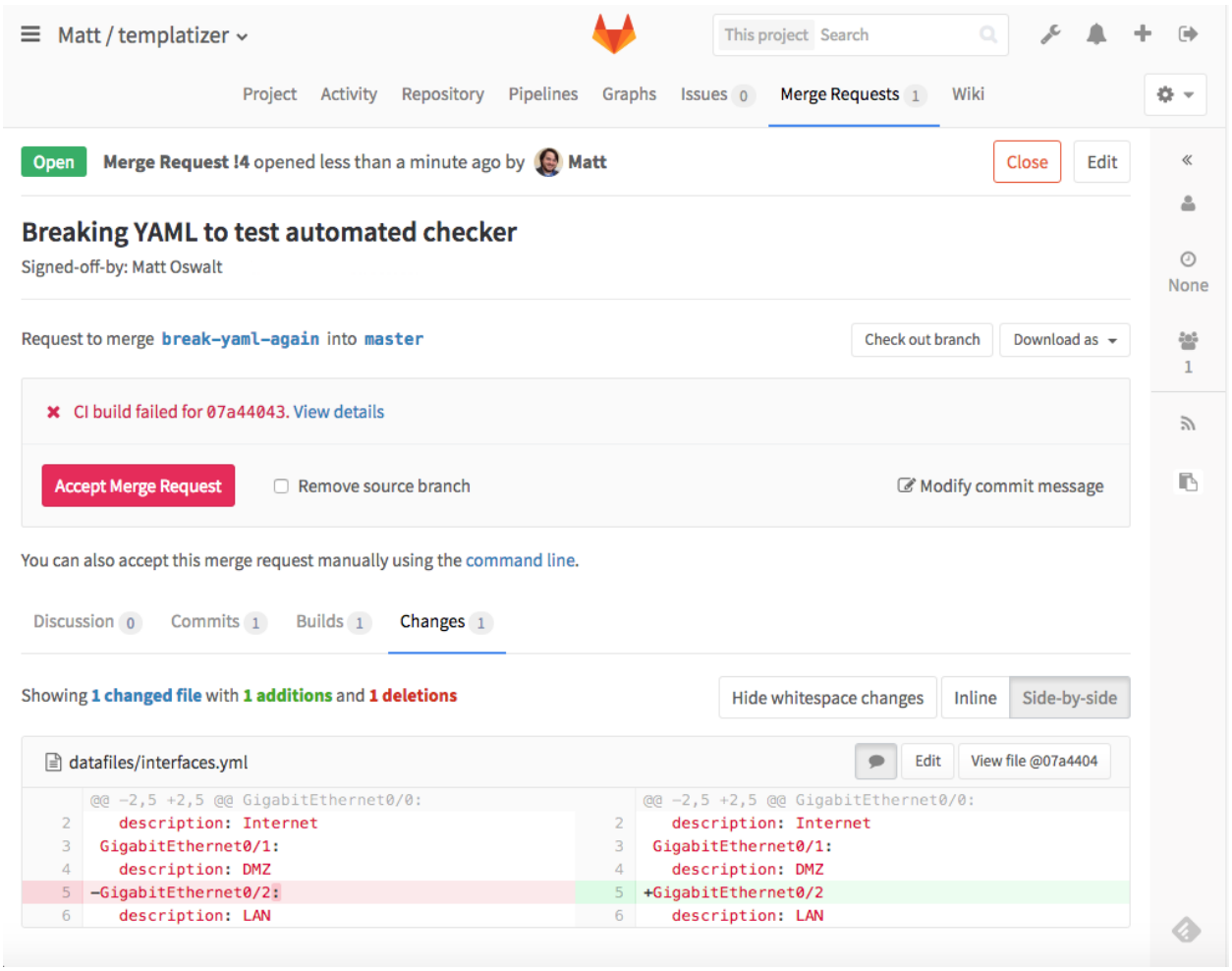

Figure 10-12. CI build failed due to invalid YAML

Both Matt and Fred are able to plainly see that there was a problem during automated testing. They can also click through to see details about what happened, including a full console log that shows the output of the script, indicating which file had the issue (Figure 10-13).

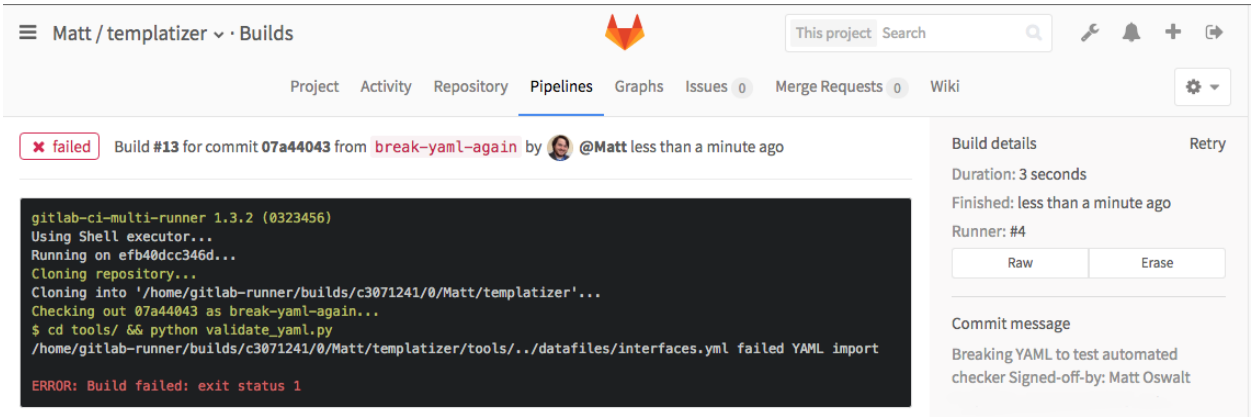

Figure 10-13. YAML validation script output

This is just one example in a multitude of possibilities with respect to automated validation and testing. Templatizer is also a Python project, so we can explore some of the tooling present in that ecosystem to run some Python-specific validation and testing as part of this CI pipeline. For instance, Tox is a popular tool for doing all kinds of automated testing within a Python project. The OpenStack community uses Tox to really simplify the CI process, by summarizing a slew of tasks within a small list of commands:

```
test:

  script:

  - cd tools/ && python validate_yaml.py

  - tox -epep8  # Checks for PEP8 compliance with Python files

  - tox -epy27  # Runs unit tests

  - tox -ecover # Checks for unit test coverage
```

Again, all of these commands must pass without error in order to "pass" the build process. When a reviewer receives a merge request that shows that these checks were passed, they know it's ready for a real review.

This part of the pipeline is crucial and is a great way to keep the workflow efficient, while also helping to ensure that past mistakes are not repeated. Here are some additional ideas that may be useful at this stage in the pipeline—explore each in your own journey toward network automation:

- Unit testing any code (e.g., Python)
- Integration testing to ensure any code is able to interoperate with other projects and APIs
- Syntax and style validation (both source code as well as data formats like YAML)

Test/Dev/Staging Environment

After some basic automated testing of simple things like syntax or style, it's usually desirable to run more "real-world" testing on the changes we make to our repository. In the case of Templatizer, we might want to actually render real configurations using the Jinja templates and YAML files against virtual devices that mimic the real production devices we'd like to eventually target. There are a number of ways to do this, and we'll discuss some of them here.

One interesting solution, if you're just looking to do some small-scale testing on your laptop, is Vagrant. Vagrant is a tool developed by HashiCorp to make management of

virtual machines easier. The configuration for this environment is contained within a file called a Vagrantfile. This is a plain-text file that's easy to place in a Git repository, and useful for ensuring that anyone working on a repository can spin up identical virtual environments. Virtual images can also be automatically downloaded from a predetermined location, reducing the need for users to put together their own images. If configured properly, a user needs only to type `vagrant up` into a shell and the environment will be instantiated automatically.

This has useful implications for software developers (a Vagrantfile can be configured to spin up identical development environments for all developers working on a codebase), but network engineers can also use Vagrant to spin up multi-vendor lab topologies. For instance, Juniper makes their Junos images publicly available to be used with Vagrant.

We can devise a Vagrantfile that constructs a simple three-node topology:

```
Vagrant.configure(2) do |config|

  # NOTE - by the time this book was released, this image was quite old.
  # Please refer to Juniper documentation on the appropriate image to use.
  config.vm.box = "juniper/ffp-12.1X47-D15.4-packetmode"
  config.vm.box_version = "0.2.0"

  config.vm.define "vsrx01" do |vsrx01|
    vsrx01.vm.host_name = "vsrx01"
    vsrx01.vm.network "private_network",
                      ip: "192.168.12.11",
                      virtualbox__intnet: "01-to-02"
    vsrx01.vm.network "private_network",
                      ip: "192.168.31.11",
                      virtualbox__intnet: "03-to-01"
  end

  config.vm.define "vsrx02" do |vsrx02|
    vsrx02.vm.host_name = "vsrx02"
    vsrx02.vm.network "private_network",
                      ip: "192.168.23.12",
                      virtualbox__intnet: "02-to-03"
    vsrx02.vm.network "private_network",
                      ip: "192.168.12.12",
                      virtualbox__intnet: "01-to-02"
  end

  config.vm.define "vsrx03" do |vsrx03|
    vsrx03.vm.host_name = "vsrx03"
    vsrx03.vm.network "private_network",
                      ip: "192.168.31.13",
                      virtualbox__intnet: "03-to-01"
    vsrx03.vm.network "private_network",
                      ip: "192.168.23.13",
                      virtualbox__intnet: "02-to-03"
  end
```

```
end
```

In the directory where this Vagrantfile is located, we need only run `vagrant up`, a Junos image will be downloaded, and three virtual Junos routers will be started and connected as described. We can immediately log in to any one of these devices quite easily:

```
~$ vagrant up vsrx01

Bringing machine 'vsrx01' up with 'virtualbox' provider...

==> vsrx01: Checking if box 'juniper/ffp-12.1X47-D15.4-packetmode' is up to
date...

==> vsrx01: Clearing any previously set forwarded ports...

==> vsrx01: Clearing any previously set network interfaces...

==> vsrx01: Preparing network interfaces based on configuration...

    vsrx01: Adapter 1: nat

    vsrx01: Adapter 2: intnet

    vsrx01: Adapter 3: intnet

==> vsrx01: Forwarding ports...

    vsrx01: 22 (guest) => 2222 (host) (adapter 1)

==> vsrx01: Booting VM...

==> vsrx01: Waiting for machine to boot. This may take a few minutes...

    vsrx01: SSH address: 127.0.0.1:2222

    vsrx01: SSH username: root

    vsrx01: SSH auth method: private key

==> vsrx01: Machine booted and ready!

==> vsrx01: Checking for guest additions in VM...

    vsrx01: No guest additions were detected on the base box for this VM! Guest
```

```
    vsrx01: additions are required for forwarded ports, shared folders, host
only

    vsrx01: networking, and more. If SSH fails on this machine, please install

    vsrx01: the guest additions and repackage the box to continue.

    vsrx01:

    vsrx01: This is not an error message; everything may continue to work
properly,

    vsrx01: in which case you may ignore this message.

==> vsrx01: Setting hostname...

==> vsrx01: Configuring and enabling network interfaces...

==> vsrx01: Machine already provisioned. Run `vagrant provision` or use the `--
provision`

==> vsrx01: flag to force provisioning. Provisioners marked to run always will
still run.



[output omitted for similar output from vsrx02 and vsrx03...]



~$ vagrant ssh vsrx01

--- JUNOS 12.1X47-D15.4 built 2014-11-12 02:13:59 UTC

root@vsrx01% cli

root@vsrx01> show version

Hostname: vsrx01

Model: firefly-perimeter

JUNOS Software Release [12.1X47-D15.4]
```

Provided there are sufficient resources present on your machine to run the topology, and that the images you need are available, there are a multitude of ways this can be used for testing network changes. This is fairly unprecedented—network vendors have historically been very slow to provide free-to-use virtual images of their platforms. However, more and more vendors are starting to do just that.

To bring this back to Continuous Integration and automated testing, this concept could be very useful for validating our changes. For instance, if you made a change to our Templatizer repository that we would like to test before rolling into production, we could render one of these templates and automatically deploy that configuration change to a virtual topology provisioned by Vagrant. Even if you're just looking to test something out on your laptop, Vagrant remains an increasingly useful tool for evaluating network platforms.

It's also possible to run a virtual environment in the public or private cloud. For instance, if your organization is running an OpenStack deployment, many of these virtual network devices can be run as virtual machines in OpenStack. You could even automate their deployment using OpenStack Heat templates. Alternatively, companies like Network to Code provide workflow automation with their On Demand Labs platform for network engineers to leverage public cloud resources to run these virtual topologies. These methods are useful if you want the same topology to be accessible by multiple engineers, all the time.

NOTE

Disclosure: Jason Edelman, one of the authors of this book, is the founder of Network to Code.

Many of these options can be automated using many of the same tools that you would use to automate the "real thing," and it's important that you consistently use the same tooling between your test and production environments, otherwise the testing is pointless. For instance, if your goal is to use a virtual environment to test the deployment of a configuration change using Ansible, construct a virtual topology that mimics your production infrastructure as closely as possible, then run through the same Ansible workflow in both the test environment and production. This gives you greater confidence that if it worked in test, it will work in production.

Test environments are one of the key components that force organizations to seriously consider a dedicated engineer for maintaining them. To do this right, test environments need to be carefully maintained, so that they're not a tremendous bottleneck to the rest of the pipeline, and that they are an adequate simulation of the real network environment.

Deployment Tools

Earlier in the chapter, we discussed the importance of understanding *what* you're deploying in a Continuous Integration/Continuous Delivery pipeline. One reason for this is that it has a big impact on the tools you use to actually deploy the changes you make.

For instance, if you're writing some Python code to automate some tasks around your network, you should consider treating it like a full-fledged software project. Regardless of the size, production code is production code. A small script is as likely to have bugs as a large web application.

In addition to the very important testing and peer review discussed earlier, you may find it useful to explore the delivery mechanisms that software developers are starting to use. If your organization uses cloud platforms like OpenStack, you may be able to leverage the available APIs to automatically deploy your changes at the end of the CI pipeline.

It's also becoming increasingly popular to deploy software in Docker containers. You could instruct your CI pipeline to automatically build a Docker image once a new change is reviewed and merged. This image can be deployed to a Docker Swarm or Kubernetes cluster in production.

On the other hand, sometimes we're not deploying custom software—sometimes our Git repositories are used simply to store configuration artifacts like YAML or Jinja templates. This is common for network automation efforts that use configuration management tools like Ansible to push network device configurations onto the infrastructure. However, while the method of deployment may differ between network engineers and software developers, Continuous Integration plays a vital role (Figure 10-14).

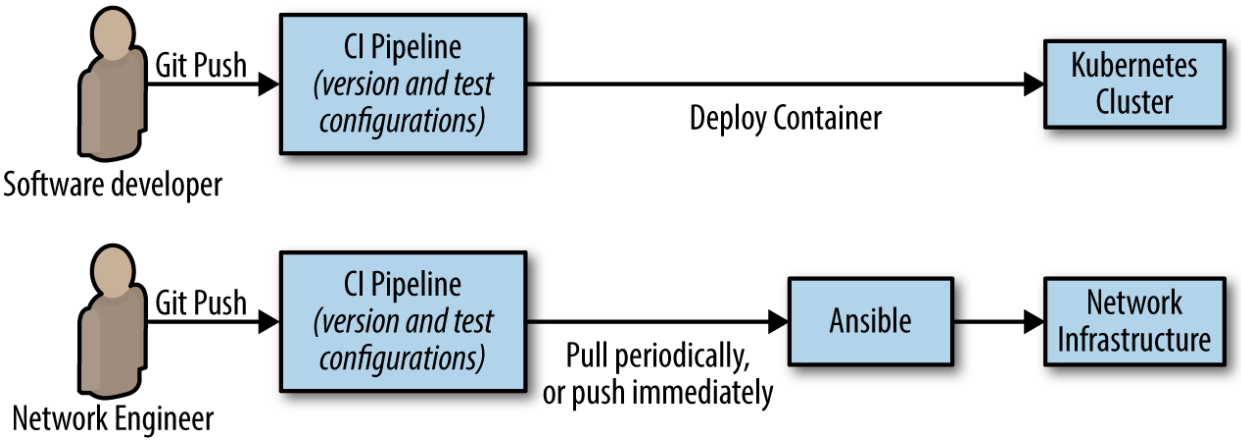

Figure 10-14. A comparison of development and networking CI pipelines

In this case, it's important to understand how these configurations are going to be used in production, as well as how rollbacks will be handled. This is an important idea not only for deciding how Ansible will actually run in production, but also how the configuration templates themselves are constructed. For instance, you might consider running an Ansible playbook to deploy some configuration templates onto a set of network devices every time a new change is merged to the master branch—but what impact will that have on the

configuration? Will the configuration always be overwritten? If so, will that overwrite a crucial part of the configuration that you didn't intend?

Some vendors provide tools to assist with this. For instance, when pushing an XML-based configuration to a Junos device, you can use the `operation` flag with a value of `"replace"` to specify that you want to replace an entire section of configuration. The following example shows a Jinja template for a Junos configuration that uses this option:

```
<configuration>
  <protocols>
      <bgp operation="replace">
          {% for groupname, grouplist in bgp.groups.iteritems() %}
           <group>
               <name>{{ groupname }}</name>
               <type>external</type>
              {% for neighbor in grouplist %}
               <neighbor>
                   <name>{{ neighbor.addr }}</name>
                   <peer-as>{{ neighbor.as }}</peer-as>
              </neighbor>
               {% endfor %}
           </group>
          {% endfor %}
      </bgp>
  </protocols>
</configuration>
```

Unfortunately, not all vendors allow for this, but in this particular case, you could simply overwrite entire sections of configuration for each new patch in the CI pipeline, to ensure that "what it should be" (WISB) always equals "what it really is" (WIRI).

This is another area where there is no silver bullet. The answer to the deployment question depends largely on what you are deploying, and how often. It's best to first settle on a strategy for network automation; decide if you want to invest in some developers and write more formalized software, or if you want to leverage existing open source or commercial tools to deploy simple scripts and templates. This will guide you toward the appropriate deployment model.

Above all, however, deployment should *never* take place until the aforementioned concepts like peer review and automated testing have taken place. A network automation effort that does not prioritize quality and stability above all is doomed to failure.

Testing Tools and Test-Driven Network Automation

Earlier in this chapter, we talked a lot about the influence that test-driven development can have on network automation. In that section, we discussed how important it was to go beyond the traditional network statistics that we use as network engineers and leverage additional tools and metrics more useful for determining application and user experience.

These metrics can be used before and after each change to truly determine the health of the network and its configuration (Figure 10-15).

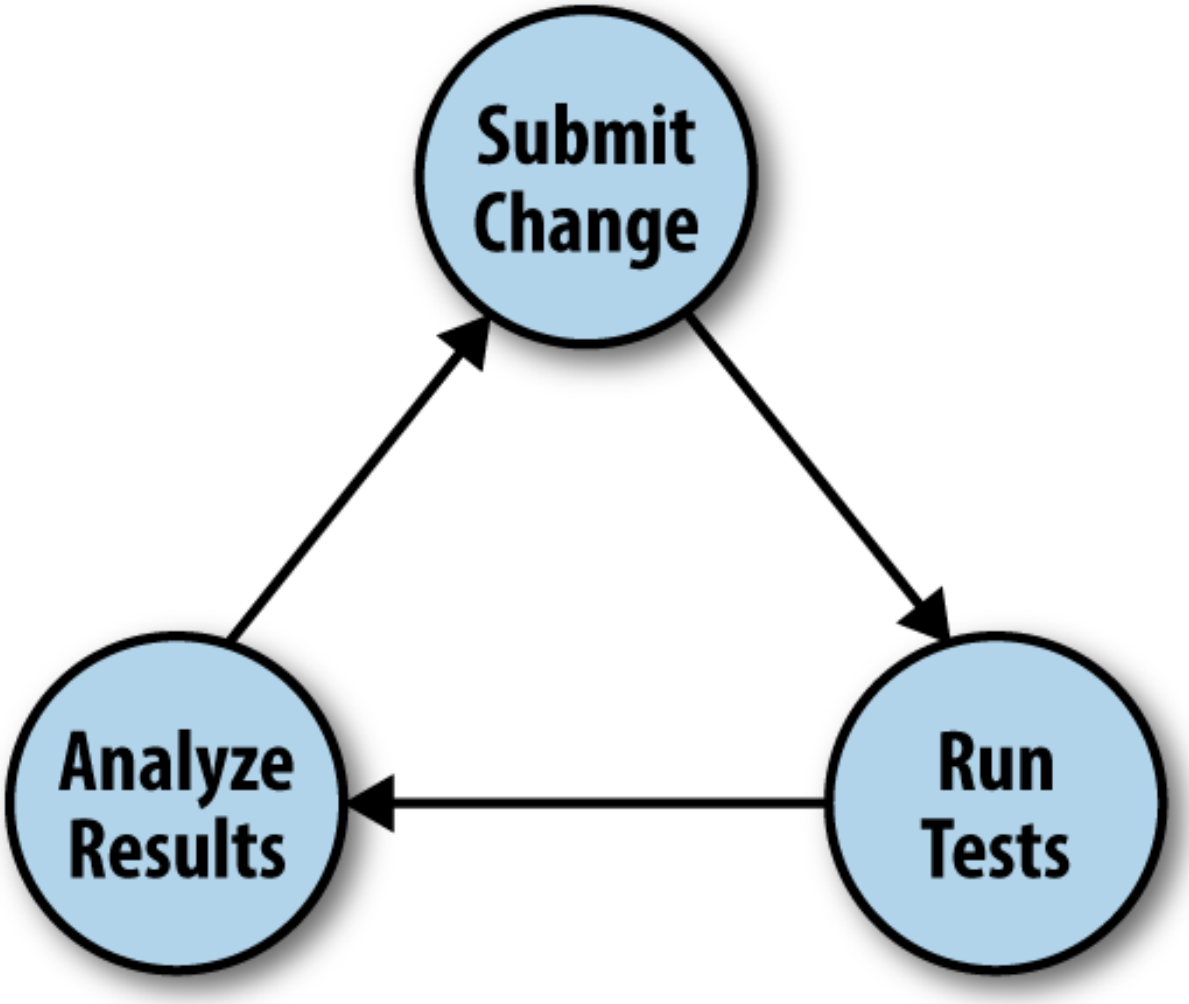

Figure 10-15. Continuously testing automated changes

Unfortunately, after several decades, the tools available for determining application experience or troubleshooting on the network haven't improved or evolved very much. These days, troubleshooting a network problem is typically relegated to one of only a handful of tools like ping, traceroute, iperf, and whatever your network management platform is able to poll via SNMP.

Largely, these tools are insufficient even from a network engineer's perspective, let alone the fact that they provide minimal visibility into application performance, which is why we run networks in the first place. However, thanks to the rise of open source, and offerings like GitHub that make open source software much easier to consume, this is starting to change.

One area that is ripe for improvement, especially within network infrastructure, is the ability to gather detailed telemetry in a flexible and scalable way. Currently, network engineers are limited to what SNMP can provide, which has a few shortcomings. SNMP is a monitoring tool that is not only limited to network infrastructure itself, but even then only exposes a subset of available data points within those devices.

The main problem here is that we're ignoring a lot of really valuable context available outside the network itself. Using frameworks like Intel's Snap, we can constantly and intelligently gather telemetry about infrastructure elements that rely on the network, like application servers, clusters, containers, and more. If we make a change on the network that adversely affects one of these entities, we can see that in the available telemetry, and perhaps automatically roll back those changes based on a set threshold.

An additional, complementary approach is to actively test the network and application infrastructure using tools like ToDD, which provides a mechanism to perform network testing like ping, http, port scanning, and bandwidth testing in a fully distributed manner. ToDD also aggregates reported data in a single JSON document so you can make decisions on the resulting test data, regardless of scale. It's important to test application-level performance (not just "ping") and to also test at a scale comparable to peak real-world activity.

NOTE

The ToDD project was started by one of the authors of this book—Matt Oswalt.

Tools like these can provide additional visibility during failover testing, such as the simulation of a data center failure. Failover testing is an under-appreciated activity when it comes to network infrastructure. Often, it's hard to get the approval to run such a test, and in the rare cases where such approval is obtained, it's even more difficult to determine how the network and the connected applications are performing. Using these and other tools, we can gather a baseline of what "normal" application performance looks like, and by running the same tests after a failover, we can have greater confidence that we have sufficient capacity to keep the business running.

These are just a few examples—the point is that open source software is no longer just an elite club only for software developers. These days, there is no excuse not to at least consider using tools like these to fill in some of the huge gaps in existing monitoring strategies—specifically the lack of application-level visibility into network performance.

Summary

These days, chances are good that your organization has some kind of in-house software development shop. Reach out to those teams and ask about their processes. If they're using Continuous Integration, there's a chance that they'd be willing to let you leverage some of

their existing tooling to accomplish similar goals with network automation. As mentioned previously, a dedicated release engineer can help greatly with management of the pipeline itself.

In this chapter we talked about a lot of process improvements (as well as some tooling to help enforce these processes), but the real linchpin to all of this transformative change is a culture that understands the costs and benefits of this approach. We'll talk a lot more about this in Chapter 11. If you don't have buy-in from the business to make these improvements, they will not last.

It's also important to remember that a big part of CI/CD is continuously learning. Continuously challenge the status quo, and ask yourself if the current model of managing and monitoring your network is *really* sufficient. Application requirements change often, so the answer to this question is often "no." Try to stay plugged in to the application and software development communities so you can get ahead of these requirements and build a pipeline that can respond to these changes quickly.