

O'REILLY®

Modern System Administration

Building and Maintaining Reliable Systems



Early
Release
RAW &
UNEDITED

Jennifer Davis

Modern System Administration

Building and Maintaining Reliable Systems

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Jennifer Davis with Chris Devers and Tabitha Sable

Modern System Administration

by Jennifer Davis with Chris Devers and Tabitha Sable

Copyright © 2020 Jennifer Davis. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North,
Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://oreilly.com>). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

Acquisitions Editor: John Devins

Editor: Virginia Wilson

Production Editor: Katherine Tozer

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: O'Reilly Media

August 2020: First Edition

Revision History for the First Edition

- 2019-09-24: First Release
- 2020-01-10: Second Release
- 2020-04-09: Third Release
- 2020-06-26: Fourth Release

- 2020-09-14: Fifth Release
- 2020-12-11: Sixth Release
- 2021-03-05: Seventh Release
- 2021-06-30: Eighth Release

See <http://oreilly.com/catalog/errata.csp?isbn=9781492055211> for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Modern System Administration*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-492-05514-3

Part I. Foundations

Chapter 1. Introduction

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the Introduction of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at vwilson@oreilly.com.

While the underlying concepts like managing capacity and security have remained the same, system administration has changed over the last couple of decades. Early administration required in-depth knowledge of services running on individual systems. Books on system administration focused on specific services on the systems from printing to DNS. The first conference dedicated to system administration, LISA, described large scale as sites for over 100 users.

Now, operations engineers are faced with an ever-growing list of technologies and third-party services to learn about and leverage as they build and administer systems and services that have thousands to millions of users. Software development is moving fast, and sysadmins need to move as quickly to accommodate and deliver value.

I wrote this book for all the experienced system administrators, IT professionals, support engineers, and other operation engineers who are looking for a map to understanding the landscape of contemporary operation tools, technologies and practices. This book may also be useful to

developers, testers, and anyone who wants to level up their operability skills.

In this book, I examine the modernization of system administration and how collaboration, automation, and system evolution change the fundamentals of operations. This book is not a “how-to” reference, as there are many quality reference materials to dig into specific topics. Where relevant, I recommend materials to level your skills in that area. I provide examples to guide a deeper understanding of the essential concepts that individuals need to understand, evaluate and execute on their work.

The focus is on tools and technologies in broad use currently, but progress is rapid with new tools and technologies coming into use all the time. These new tools may supplant today’s favorite tools with little notice. Don’t worry about learning the wrong tools; study the underlying concepts. Apply these concepts to evaluate and adopt tools as they become available.

At its core, modern system administration is about assessing and regulating risk to the business. It encompasses changes in how sysadmins collaborate with development and testing, deploy and configure services, and scale in production due to increased complexity of infrastructure and data generation.

Principles

The first part of the book focuses on the number of technical practices. These include:

- Version Control is a practice that enables the organization, coordination, and management of objects. It’s the foundation of automating software development and delivery with continuous integration and continuous deployment.
- Local Development Environment is a practice of standardizing on a set of tools and technologies to reduce challenges to

collaboration and leverage work that has been done to set up an environment. It empowers teams to choose tools intentionally.

- Testing is a practice of getting explicit feedback about the impact of change. It's another critical part of automation and continuous integration and continuous deployment.
- Security is the practice of protecting hardware, software, networks, and data from harm, theft, or unauthorized access.

You can't be the lone sysadmin anymore known for saying "no." The nature of the work may start at understanding operating systems, but it spans across understanding services across different platforms while working in collaboration with other teams within the organization and potentially external to your team. You must adopt tools and practices from across the organization to better perform your job.

You need to be comfortable with using the terminal and graphical interfaces. Just about every tool I'll cover has some aspect of command line usage. Being able to explore and use the tools helps you understand when problems arise with the automation. When you have to debug the automation, you need to know whether it's the tool or your use of the tool.

You can't ignore version control. For years, DORA's annual State of DevOps report has reported that the use of version control highly correlates to high IT performers.¹ Version control is fundamental to collaboration with other parts of the organization whether you're writing code to set up local development and test environments or deploying applications in a consistent and repeatable manner. Version control is also critical for managing your documentation whether it's README's embedded in a project repository, or as a separate project that spans content for the organization. You administer tests of the code you write, as well as the infrastructure that you build within version control.

You build and maintain virtual images and containers for use locally as well as within the cloud. All of this requires some understanding of how to read,

debug, and in some cases write code in a particular language. Depending on the environment, Ruby, Python, or Go may be in use.

NOTE

While I include some code snippets in various languages, this book cannot cover the multitude of information that's important to learn a specific language. While you can (and should) specialize in a specific language, don't limit yourself to a single language as languages do have different strengths. Early Linux administration focused on bash or Perl scripts. Now individuals may additionally use Go, Python, or Rust. Folks who limit their ability to adopt other languages will hinder their employability as new tools evolve.

Whether you are collaborating on a project with development, or just within your role-specific Operations team, you need to define and build development environments to replicate the work quickly that others have done. You then can make small changes to projects — whether they are infrastructure code, deployment scripts, or database changes — before committing code to version control and having it tested.

Modernization of Compute, Network and Storage

The second part of the book examines the contemporary landscape, or general conditions under consideration to lay a foundation for choosing the right options in alignment with requirements now and how to make changes as conditions evolve.

Compute

Virtualization technology set the stage for cloud computing, and containers further transformed the application infrastructure landscape. Serverless computing allows individuals to focus on application build and run in exchange for paying a hosted service for maintaining the server infrastructure as needed.

- Virtualization
- Containers
- Serverless
- Compute

Network

There are more than 20 billion connected devices as of 2021. This includes business and manufacturing robotics, cars, smart home devices, healthcare devices, security systems, phones, and computers. The more devices that need to communicate, the more network bandwidth is needed to enable connection between devices.

Storage

Storage choices have evolved and while storage is a commodity the data stored is not. The choices made about the data - how and where it's stored - impact what can be done with the data.

Infrastructure Management

The third part of this book covers managing infrastructure. Systems administration practices that work well when managing isolated systems are generally not transferable to cloud environments. Storage and networking are fundamentally different in the cloud, changing how you architect reliable systems and plan to remediate disasters.

For example, network tuning that you might handcraft with `tcp` testing between nodes in your data centers is no longer applicable when your cloud provider limits your network capacity. Instead, balance the abilities gained from administering networks in the data center along with in-depth knowledge about the cloud providers limits to build out reliable systems in the cloud.

In addition to version control, you need to build reusable, versioned artifacts from source. This will include building and configuring a continuous integration and continuous delivery pipeline. Automation of your infrastructure reduces the cost of creating and maintaining environments, reduces the risk of single points of critical knowledge, and simplifies the testing and upgrading of environments.

Scaling Production Readiness

The fourth part of the book covers the different practices and processes that enable scaling system administration. As a company grows, monitoring and observability, capacity planning, log management and analysis, security and compliance, on-call and incident management are critical areas to maintain, monitor and manage risk to the organization.

The landscape of user expectations and reporting has changed with services such as Facebook, Twitter, and Yelp providing areas for individuals to report their dissatisfaction. To maintain the trust of your users (and potential users), in addition to improvements to how you manage and analyze your logs, you need to update security and compliance tools and processes. You also need to establish a robust incident response to issues when we discover them (or worse when our users find them).

Detailed systems monitoring adds application insights, deeper observability, and tracing. In the past, system administration focused more on system metrics, but as you scale to larger and more complex environments, system metrics are less helpful and in some cases not available. Individual systems are less critical as you focus on the quality of the application and the impact on your users.

Capacity planning goes beyond spreadsheets that examine hardware projections and network bandwidth utilization. With cloud computing, you don't have the long lead times between analysis of need and delivery of infrastructure. You may not spend time performing traditional tasks such as ordering hardware, and "racking and stacking" of hardware in a data center.

Instance availability is near instantaneous, and you don't need to pay for idle systems anymore.

Whether containerized microservices, serverless, or monolithic applications, log management, and analysis needs have become more complex. The matrix of possible events and how to provide additional context to your testing, debugging, and utilization of services is critical to the functioning of the business.

The system administrator role is a critical role that encompasses a wide range of ever-evolving skills. Throughout this book, I share the fundamental skills to support architecting robust highly scalable services. I'll focus on the tools and technologies to integrate into your work so that you can be a more effective systems administrator.

A Role by any Other Name

I have experienced a dissonance over the last ten years over the role "sysadmin". There is so much confusion about what a sysadmin is. Is a sysadmin an operator? Is a sysadmin the person with root? There have been an explosion in terms and titles as people try to divorce themselves from the past. When someone said to me "I'm not a sysadmin, I'm an infrastructure engineer", I realized that it's not just me feeling this.

To keep current with the tides of change within the industry, organizations have taken to retitling their system administration postings to devops engineer or site reliability engineer (SRE). Sometimes this is a change in name only with the original sysadmin roles and responsibilities remaining the same. Other times these new titles encompass an entirely new role with similar responsibilities. Often it's an amalgamation of old and new positions within operations, testing, and development. Let's talk a little about the differences in these role titles and set some common context around them.

DevOps

In 2009 at the O'Reilly Velocity Santa Clara conference, John Allspaw and Paul Hammond co-presented “10+ deploys per day: Dev and Ops Cooperation at Flickr”. When a development team is incentivized to get features delivered to production, and the operations team is incentivized to ensure that the platform is stable, these two teams have competing goals that increase friction. Hammond and Allspaw shared how it was possible to take advantage of small opportunities to work together to create substantial cultural change. The cultural changes helped them to get to 10+ deploys per day.

In attendance for that talk, Andrew Clay Shafer, co-founder of Puppet Labs tweeted out:

*Don't just say 'no', you aren't respecting other people's problems...
#velocityconf #devops #workingtogether*

—Andrew Clay Shafer (@littleidea)

Having almost connected with Shafer at an Agile conference over the topic of Agile Operations, Patrick Debois was watching Shafer's tweets and lamented not being able to attend in person. An idea was planted, and Debois organized the first devopsdays in Ghent. Later Debois wrote “And remember it's all about putting the fun back in IT”² in a post-write up of that first devopsday event. So much time has passed since that first event, and devopsdays has grown in locations³, to over 70 events in 2019 with new events started by local organizers every year.

But what is devops? It's very much a folk model that gets defined differently depending on the individual, team, or organization. There is something about devops that differentiates practitioners from nonpractitioners as evidenced by the scientific data backed analysis performed by Dr. Nicole Forsgren in the DORA Accelerate DevOps Report.⁴

At its essence, I see devops as a way of thinking and working. It is a framework for sharing stories and developing empathy, enabling people and teams to practice their crafts in effective and lasting ways. It is part of the

cultural weave of values, norms, knowledge, technology, tools, and practices that shape how we work and why.⁵

Many people think about devops as specific tools like Docker or Kubernetes, or practices like continuous deployment and continuous integration. What makes tools and practices “devops” is how they are used, not the tools or practices directly.

Site Reliability Engineering (SRE)

In 2003 at Google, Ben Treynor was tasked with leading a team of software engineers to run a production environment. Treynor described SRE as “what happens when a software engineer is tasked with what used to be called operations.”

Over time SRE was a term bandied about by different organizations as a way to describe operations folks dedicated to specific business objectives around a product or service separate from more generalized operations teams and IT.⁶ In 2016, some Google SREs shared the Google specific version of SRE based on the practices, technology, and tools in use within the organization in the Site Reliability Engineering book ⁷. In 2018, they followed it up with a companion book “The Site Reliability Workbook” to share more examples of putting the principles and practices to work.

So what is SRE? Site Reliability Engineering is an *engineering* discipline that helps an organization achieve the *appropriate* levels of *reliability* in their systems, services, and products.

Let’s break this down into its components starting with *reliability*.

Reliability is literally in the name “Site Reliability Engineer” so it makes sense. Reliability is a measurement of how well a system is performing. But what does that really mean? It is defined differently depending on the type of service or product that is being built. Reliability can be availability, latency, throughput, durability, or whatever else your customer may be evaluating to determine that the system is “ok”.

Being an engineering discipline means that we approach our work from an analytical perspective to design, build, and monitor our solutions while considering the implications to safety, human factors, government regulations, practicality and cost.⁸

One of the strong evolution points from regular system administration work was the measurement of impact on humans. This work has been described as toil due to the work being repetitive and manual. Google SRE implemented a cap of 50% toil work, redirecting this work to development teams and management including on-call responsibilities when the toil exceeded the cap.⁹

By measuring the quality of work and changing who does the work, it changes some fundamental dynamics between ops and dev teams. Everyone becomes invested in improving the reliability of the product rather than a single team having to carry the brunt of all the support work of trying to keep a system or service running. SRE teams are empowered to help reduce the overall toil.

RESOURCES FOR EXPLORING SRE

Learn more about Google SRE from the [Site Reliability Engineering](#) and [The Site Reliability Workbook](#) books.

Read [Alice Goldfuss's "How to Get into SRE"](#) and [Molly Struve's "What It Means To Be A Site Reliability Engineer"](#) blog posts.

How do Devops and SRE Differ?

While devops and SRE arose around the same time, devops is more focused on culture change (that happens to impact technology and tools) while SRE is very focused on changing the mode of Operations in general.

With SRE, there is often an expectation that engineers are also software engineers with operability skills. With DevOps Engineers, there is often an assumption that engineers are strong in at least one modern language as well as have expertise in continuous integration and deployment.

System Administrator

While devops and SRE have been around for approximately ten years, the role of system administrator (sysadmin) has been around for much longer. Whether you manage one or hundreds or thousands of systems, if you have elevated privileges on the system you are a sysadmin. Many definitions strive to define system administration in terms of the tasks involved, or in what work the individual does often because the role is not well defined and often takes on an outsized responsibility of everything that no one else wants to do.

Many describe system administration as the digital janitor role. While the janitor role in an organization is absolutely a critical role, it's a disservice to both roles to equate the two. It minimizes the roles and responsibilities of each.

A sysadmin is someone who is responsible for building, configuring, and maintaining reliable systems where systems can be specific tools, applications, or services. While everyone within the organization should care about uptime, performance, and security, the perspective that the sysadmin takes is focused on these measurements within the constraints of the organization or team's budget and the specific needs of the tool, application, or service consumer.

NOTE

I don't recommend the use of devops engineer as a role. Devops is a cultural movement. This doesn't stop organizations from using *devops* to describe a set of tasks and job responsibilities that have eclipsed the role sysadmin.

I've spent a fair amount of time reading job requirement listings, and talking to other folks in the industry about devops engineers. There is no single definition of what a devops engineer does in industry (sometimes not even within the same organization!).

While engineers with "devops" in their title may earn higher salaries than ones with "system administrator"¹⁰, this reinforces the adoption of the title regardless of the lack of a cohesive set of roles and responsibilities that translate across organizations.

Having said that, "devops engineer" is in use. I will try to provide methods to derive additional context to help individuals understand how to evaluate roles with the title in comparison to their current role.

Finding Your Next Opportunity

One of the reasons you might have picked up this book, is that you've been within your position for awhile, and you're looking to your next opportunity. How do you identify positions that would be good for your skills and experiences and desired growth? Across organizations, different roles mean different things, so it's not as straightforward as just substituting a new title and doing a search. Often it seems the person writing a job posting isn't doing the job being described, as the postings will occasionally include a mishmash of technology and tools.

A danger to avoid is thinking that somehow there is some inherent hierarchy implied by the different roles even as some folks in industry or even within an organization assume this. Names only have as much power as we give them. While responsibilities are changing and we need to add and update our skills, this isn't a reflection of individuals or the roles that they have now.

There is a wide range of potential titles. Don't limit yourself by the role title itself, and don't limit your search to just "sysadmin" or even "sre" and

“devops”. From “IT Operations” to “Cloud Engineer” the variety of potential roles are diverse.

Before you even examine jobs, think about the skills you have. As a primer, think about what technical stacks are you familiar with? How familiar are you with the various technologies described in this book? Think about where you want to grow. Write all of this down.

As you review job reqs, as you note skills that you don’t have that you’d like to have write those down. Compare your skill evaluation with the job requirements and work towards improving those areas. Even if you don’t have experience in these areas, during interviews if you are able to clearly talk about where you are compared to where you want to be for those skills it goes a long way to showing your pursuit of continuous learning (which is a desirable skill).

PREPARING QUESTIONS PRIOR TO THE INTERVIEW

Logan McDonald, a Site Reliability Engineer at BuzzFeed, shares some questions to ask during an interview in this blog post [Questions I ask in SRE interviews](#). While she specifically targets the SRE interview, these are helpful questions for any kind of operations position to help qualify the direction and responsibility for the position.

Today, sysadmins can be devops engineers or site reliability engineers or neither. Many SRE skills overlap with sysadmin skills. It can be frustrating with years of experience as a sysadmin to see a lack of opportunities with the role *sysadmin*. If examined, often the roles advertised as SRE or devops engineer have very similar skills and expectations of individuals. Identify your strengths, and compare them with jobs requirements from positions that sound interesting. Map out your path and work on those skills.

1 DORA’s annual State of DevOps report: <https://devops-research.com/research.html>

2 <http://bit.ly/debois-devopsdays>

3 DevOpsDays Events: <https://www.devopsdays.org/events/>

- 4 DORA Accelerate DevOps Report: [*https://devops-research.com/research.html*](https://devops-research.com/research.html)
- 5 Effective DevOps, Davis, and Daniels
- 6 The Many Shapes of Site Reliability Engineering: [*https://medium.com/slalom-engineering/the-many-shapes-of-site-reliability-engineering-468359866517*](https://medium.com/slalom-engineering/the-many-shapes-of-site-reliability-engineering-468359866517)
- 7 Site Reliability Engineering book: [*https://landing.google.com/sre/books/*](https://landing.google.com/sre/books/)
- 8 Wikipedia: [*https://en.wikipedia.org/wiki/List_of_engineering_branches*](https://en.wikipedia.org/wiki/List_of_engineering_branches)
- 9 Stephen Thorne Site Reliability Engineer at Google, “Tenets of SRE”:
[*https://medium.com/@jerub/tenets-of-sre-8af6238ae8a8*](https://medium.com/@jerub/tenets-of-sre-8af6238ae8a8)
- 10 2015 DevOps Salary Report from Puppet: [*http://bit.ly/2015-devops-salary*](http://bit.ly/2015-devops-salary)

Chapter 2. Infrastructure Strategy

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 2nd chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at vwilson@oreilly.com.

Infrastructure strategy is the plan of action for infrastructure within the organization to conduct business. Often people think of the infrastructure strategy as minimizing cost to maximize profitability. That isn’t a strategy although it may be a desirable outcome. In this chapter, I cover 3 underlying components of infrastructure strategy:

- Infrastructure Lifecycles
- Infrastructure Stacks
- Infrastructure as Code

Understanding Infrastructure Lifecycle

The infrastructure lifecycle informs the planning of your strategy. It helps you to plot a set of actions in alignment to business objectives. Let’s

examine the physical and cloud asset lifecycle and understand the challenges that can hinder your plan.

Lifecycle of Physical Hardware

To maximize the benefits of managed hardware, you can think about the physical assets requiring specific practices as they pass through each phase of the hardware lifecycle.

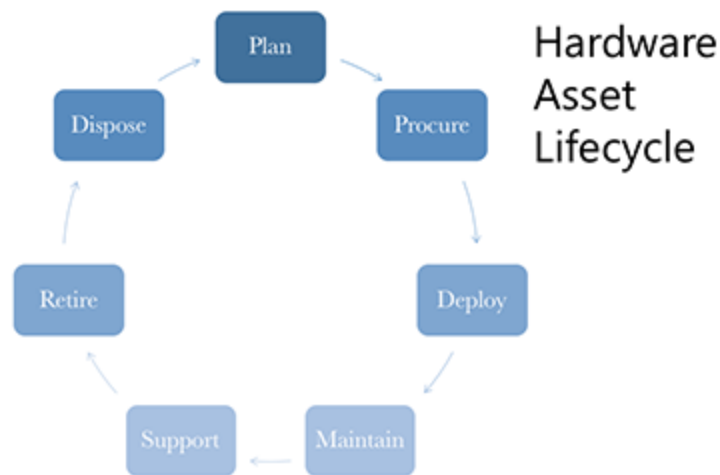


Figure 2-1. Hardware Asset Lifecycle

The phases of the hardware asset lifecycle include:

1. Plan

You plan hardware purchases taking into account space and available space along with hardware that is currently owned. A couple of grand per server builds up quite quickly. Planning also needs to factor in the cost of cooling and power as well.

2. Procure

Once you identify the set of hardware that you need to procure, you determine whether you are buying or leasing it based on obtaining quotes from vendors and aligning to the plan. Building strong relationships with vendors in servers, storage, and

networking helps you get the best prices on hardware as well as any necessary support.

3. Deploy

Once equipment arrives, you need to verify the systems arrive as specced. A different team may be responsible for the physical deployment into the racks, or it may be part of your job responsibilities.

You install the required operating system and necessary updates. You may perform some amount of burn-in to verify that the system behaves as expected and that there is no component performance differences.

Finally, you install and deploy necessary software and services to make the system live.

4. Maintain

You update operating system and upgrade any hardware as necessary to support the required services.

5. Support

You monitor the hardware for issues and repair based on any expectations of services. This may mean coordinating support or physically swapping in new hardware as necessary.

6. Retire

You identify when the hardware is no longer needed and de-provision running systems. This may be a long process to identify any access to the system.

Sometimes, new hardware is being brought into service to replace older hardware and that's somewhat easier to swap in depending on the software architecture.

7. Dispose

Once you have retired software from the system and removed it from service (and if it is no longer useful within your organization in any other capacity), you have to dispose of the hardware. In addition to ensuring that no sensitive data remains on the system, you may need to be aware of specific laws and regulations around disposal.

When planning hardware requirements, it's common to think about a 3-5 year lifespan for non-specialized hardware. In part, this is due to advancements in the physical technology that improves the cost of running servers. It is also due to advancements in the system software, where older hardware might not support current operating systems.

With specialized hardware like storage appliances, the lifecycle changes slightly in that the costs can range from the 10s of thousands to close to a million dollars. On top of that, maintenance and support are separate costs and longer-term investments.

Lifecycle of Cloud Services

To maximize the benefits of the cloud, organizations still need to consider the lifecycle of assets. Physical racking and stacking and the physical security of the hardware are handled by the service provider. You also eliminate the need to maintain and dispose of physical systems, but every other phase is still present in some form.

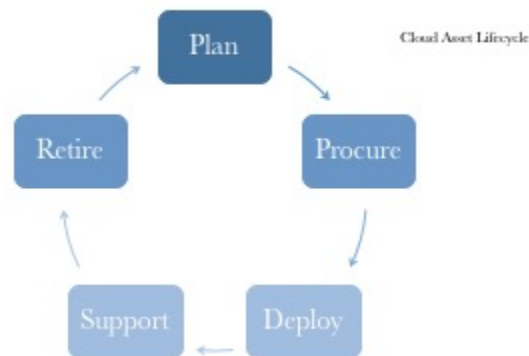


Figure 2-2. Cloud Asset Lifecycle

1. Plan

You focus on identifying specific cloud services to use (for example, specific machine types or reserving capacity versus on-demand) and budget forecasting.

2. Procure

Instead of having to plan for expenditures all at once, you set budgets per individual or team to align spending and leverage purchasing power across the organization. You build relationships with different cloud providers, and identify compatible services that align with business requirements.

3. Deploy

Instead of physically deploying servers, you write infrastructure code to provision, verify, and deploy necessary cloud resources programmatically.

4. Support

Through careful monitoring of systems in use, you identify areas for cost savings.

You assess, monitor, and repair security vulnerabilities in the software and underlying layers depending on the service in use.

You also may be the central contact with the service provider to coordinate support.

5. Retire

Rather than worrying about physical hosts for 3-5 years and maximizing their value, sysadmins make sure instances only live as long as needed eliminating cloud resources that are running and providing no value-add work. Policies can be put in place that shut down and de-provision resources that are no longer in use.

Challenges to Planning Infrastructure Strategy

There are challenges to applying an infrastructure strategy to manage your assets throughout their lifecycle.

Operation engineering teams are often understaffed which leads to insufficient time spent on quality practices on managing hardware effectively. This could mean hardware arriving and delayed deployment or lack of retiring aging systems in a timely manner.

Moving 100% to the cloud may ease some of the stress on operation engineering teams allowing more time to focus on the different practices involved in managing infrastructure.

A hybrid environment where part of the infrastructure is on-premise and part is managed by a cloud provider adds additional complexity. This might be acceptable if there is not in-house knowledge for managing necessary services.

Another challenge is the lack of investment or availability in quality tools. Often spreadsheets are used to design datacenters (including cooling and power), manage vendor relationships and inventory (from the physical hardware itself to the cabling organization). This can hinder collaboration, communication, and knowledge transfer throughout the organization.

With the ease of quickly provisioning resources, visualization of resources in use is critical to prevent costly mistakes.

Infrastructure Stacks

Infrastructure stacks inform patterns in solving common problems. It helps you to reduce complexity by reducing the number of technologies in use.

One of the original web service stacks is the LAMP(Linux, Apache, MySQL, PHP/Perl/Python) stack. The LAMP stack identified a specific set of technologies in use quickly.

More than just quick mnemonic to describe a set of technology, you have to be proficient at many different stacks and architecture patterns to build out the necessary infrastructure required to support various software.

You also need to understand which stacks apply to your problems. For sites leveraging JAM(JavaScript,API, Markup)stack, you still need to collaborate with developers to ensure that the workflow for change minimizes the risk to the end-user while providing value. Deploying bad files that are cached by a CDN can be easily remedied if folks are monitoring for problems and rectifying as needed.

Infrastructure as Code

You have a growing set of options when it comes to building out your infrastructure from virtualization in on-prem data centers, to cloud instances, to containers, and now serverless. You need processes that help you to alleviate the risks while also allowing you to move quickly.

Infrastructure as Code(IaC) provides

- deployment automation,
- consistency in environments where you want consistency and visible customization,
- repeatability with code and separate from an instance of deployment, and
- reusable components.

These benefits are valuable as they

- increase your speed at deploying the same infrastructure,
- reduce the risk in deploying by eliminating errors introduced through manual deploys,
- increase the visibility across the organization to what is getting deployed and how.

It comes with a fixed cost as it takes time to automate what you do manually. You have to think about what it is you are doing, how you are

doing it, and all the corner cases that you take care of when you're driving the provisioning, configurations, and deploys manually.

IaC is the mechanism of treating your infrastructure in the same way that you treat your code, adopting practices that have helped improve quality and visibility. This includes storing infrastructure definitions in version control, testing these definitions, continuous integration (CI), and continuous delivery/deployment (CD).

NOTE

Often IaC is conflated with Infrastructure as a Service(IaaS), but these are two different concepts. IaC can be used with on-prem hardware and cloud instances, while IaaS is a service offered by a cloud provider.

CFEngine, Puppet, Chef Infra, SaltStack, and Red Hat Ansible are all examples of software that have evolved from a need to automate infrastructure configuration to eliminate configuration drift. They each have slightly different features and formats for how to define infrastructure as code.

NOTE

Often the focus in choosing an infrastructure automation solution is based on the underlying language, which is not as useful when examining the functionality that an organization needs.

Each platform provides mechanisms, for example, to install web servers and define configurations for the underlying operating systems. There is also a lot of community shared solutions backing the different options, for instance, the **Chef Supermarket** and **Puppet Forge** can help reduce the time needed to automate a component within a stack. Because the configurations are defined in text files, project repos can be stored in version control, and change can be managed through standard code control processes.

NOTE

One big difference between on-prem hardware that you've physically provisioned and cloud infrastructure is the programmatic nature of provisioning with the cloud. While there are options like OpenStack software that allows us to leverage our physical hardware and provide this abstraction, there is an absolute limit based on the hardware we have previously purchased and deployed into our datacenter.

In the cloud, you are bound by your budget. This is good when it comes to releasing infrastructure that you no longer need, but problematic if you have the tools to track and limit spending.

You need a way to provision infrastructure in the cloud. Service providers generally provide tools to do this programmatically. Amazon Web Services provides AWS CloudFormation, and Microsoft Azure provides Azure Resource Manager templates. There are also shared solutions to help folks get started with these options, for example, [Azure Quickstart Templates](#).

Templates allow you to deploy resources and have confidence that the process is consistent and repeatable. For infrastructure that is mutable, it's also recommended to leverage configuration management tools like Chef Infra or Puppet to keep the instances consistent.

Wrapping Up

After reading this chapter, you should be able to:

- Define your organization's infrastructure lifecycle
- Define infrastructure stack and identify the first web stack service
- Define Infrastructure as Code
- Explain the benefits of Infrastructure as Code

Part II. Principles

Chapter 3. Version Control

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 3rd chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at vwilson@oreilly.com.

Version control is a technology used to commit, compare, merge, and restore past revisions of objects. Many people used to believe that version control was for developers, not system administrators.

Version control systems like Concurrent Versions System (CVS) were heavy weight. You could backup configuration files through copying over the original and then modifying quickly this was good enough for administering systems. Managing backups through files is not sustainable, especially when you need the rest of your team to understand why and how to do the work as well. It also makes it harder for the future you that may have done a great number of things since making a change.

Version control is the foundation of collaborating with others on your team and within the organization. It is also one of the key technical practices tied to high performing organizations according to the 2018 Accelerate: State of the DevOps Report.¹, yet it’s still not a skill that has been embraced by non-developers. All scripts, configuration and definition files should be maintained and versioned within version control.

NOTE

Git, subversion, and mercurial are all acceptable choices for version control. In this book, I'll focus on examples using git as a specific implementation of distributed version control.

There are many options for free or paid hosted git repositories including GitHub, GitLab, AWS CodeCommit, Microsoft Azure Repos, and Atlassian Bitbucket. This is by no means an exhaustive list, and the basic feature parity is pretty similar. Within your organization, one solution may work better for you. To illustrate some of the important concepts and practices of shared version control, I'll primarily use examples from GitHub as a hosted git platform.

While I focus on git from the command line and **Visual Studio Code** throughout this book, there are many graphical interfaces available including some incorporated into text editors. I'll use the command line to explain the concepts but please use the tool that is most effective for you whether it's a specific graphical user interface or the command line.

Fundamentals of Git

You generally start to work on a project for some specific feature or to fulfill a particular request. Maybe you need to add or update a configuration, or write a new script to validate a production service.

With git, you manage change over time to the configuration or script. You may have existing content to manage or start from an empty project directory.

The following example starts from an empty project directory.

TIP

If you're new to git, I recommend learning in a scratch directory. It's helpful to build context of the tool without worrying how it's going to potentially impact something important. After **installing git**, you can replicate these steps.

First, you need to **configure git** if you've never installed and configured it. The basic configuration must include your identity as this is baked into every operation. Replace your name, and email with your own. This can be

done via the command line using the `git config` command or by creating a `.gitconfig` file in your home directory.

```
[user]
  name = Jennifer Davis
  email = jennifer@modernoperations.org
```

Next, create a new directory. From the command line, there isn't going to be anything that automatically tracks your project. You have to explicitly tell git that you want to turn this into a project to track. You do this by using the *init* option.

```
$ mkdir ms-git
$ cd ms-git
$ git init
Initialized empty Git repository in /Users/sigje/wd/ms-git/.git/
```

With this `init`, this directory also becomes a git repository. You haven't connected to an external repository though. If your local system gets corrupted and becomes inaccessible, you don't have backups of this directory anywhere else and you'd lose the repo.

When you make new files or directories, or edit files, that isn't tracked automatically. The only thing that has changed by issuing `git init` has been the creation of the *.git* directory.

ORGANIZING OUR PROJECTS IN VERSION CONTROL: MONO-REPO VERSUS MULTI-REPO

In this example, I'm revealing the fundamentals of git step by step to focus on the concepts of version control. If you are moving an existing project or starting a new project using version control, think about the organization of the project.

There is no one right way to do project organization when it comes to choosing between one project per repo (multi-repo) or all projects within a single repo (mono-repo). Each method includes a set of trade-offs.

One trade-off is code organization. With multi-repos, you agree to one project per repo, but there is no holistic definition of what a project entails. Some projects line up well to the project definition but for other work that might not be so clear-cut.

For example, think about this scenario: where would a single helper script for configuring a laptop reside?

It could be in its own repository, grouped with other random helper scripts, or grouped with all workstation related scripts.

How do individuals find this helper script or identify whether it exists already? In a mono-repo, there is a limited set of locations that the code can be found because everything is in one repo. With multi-repos, someone would have to know which repos to search.

A second trade-off is dependency management. With a mono-repo, you can lock your dependencies down to specific versions which can be helpful when your projects need to have the same version of software. Yet, locking dependencies for software to a single version can be problematic if your projects require different versions of software.

A third trade-off is control especially when separate functional teams need to collaborate on different projects and want to have different ways of working on the mono-repo. Work preferences can cause

personal conflict between the different groups causing problems in code reviews, and merging code.

This is not a comprehensive list of trade-offs. Your team will have to decide whether a mono-repo or multi-repo is more beneficial and should include specific reasons why one method is preferred over the other.

To examine the project directory, use the *tree* command (if available on your operating system).

```
$ tree -a
.
├── .git
│   ├── HEAD
│   ├── config
│   ├── description
│   ├── hooks
│   │   ├── applypatch-msg.sample
│   │   ├── commit-msg.sample
│   │   ├── fsmonitor-watchman.sample
│   │   ├── post-update.sample
│   │   ├── pre-applypatch.sample
│   │   ├── pre-commit.sample
│   │   ├── pre-push.sample
│   │   ├── pre-rebase.sample
│   │   ├── pre-receive.sample
│   │   ├── prepare-commit-msg.sample
│   │   └── update.sample
│   ├── info
│   │   └── exclude
│   ├── objects
│   │   ├── info
│   │   └── pack
│   └── refs
│       ├── heads
│       └── tags
```

There are a lot of files under the *.git/hooks* directory that have the name *sample*. These files are **examples of hooks**. Hooks are custom scripts that can be configured to run based on certain actions being taken. The

examples are mostly Perl scripts but any executable script can be used as a hook. It's safe to remove these.

Create a file named test.md in the project directory.

```
$ tree -a
.
├── .git
│   ├── HEAD
│   ├── config
│   ├── description
│   ├── hooks
│   ├── info
│   │   └── exclude
│   ├── objects
│   │   ├── info
│   │   └── pack
│   └── refs
│       ├── heads
│       └── tags
└── test.md
```

After creating the test.md file, you won't see any changes within the .git directory. This is because you haven't signaled in any way that this is a file to be monitored and stored.

TIP

You have to be explicit about what you want to save to a git repository. New files and directories aren't automatically monitored by git when created.

```
$ git status
```

```
On branch master
```

```
No commits yet
```

```
Untracked files:
```

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

```
test.md
```

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

With *git status*, you receive verification that this file is *Untracked*. Git helpfully gives you some guidance about what comes next. Let's step back and examine the different states of work that files within your project can have.

Untracked is exactly what it sounds like, files that are not currently being tracked in the git repo. There may be certain files or file types that you intentionally don't want to have included in the repository.

Adding the names of files to a special file named *.gitignore* will ensure that these files don't get added to the repository accidentally, and you will stop getting information about them in the status output.

TIP

You don't want to track unencrypted keys, temporary files, and local configurations. Add these to the *.gitignore* to prevent their inclusion.

As an example of where you might use this, on a macOS system, you could add *_Store* to a *.gitignore* file so that the git repository doesn't get polluted with unnecessary *.DS_Store* files that don't add value to your project.

NOTE

It's a good practice after creating a project to create a *.gitignore* and populate it with any file names that you don't want to have included in the repository. Getting into this practice early will help prevent sensitive information getting checked in. Often, within the context of your job there may be standard configurations that can be ignored.

For example, let's look at a *.gitignore* from an Open Source project. Within cookbooks in the sous-chef organization, adopting or creating a new cookbook within the organization uses a similar *.gitignore*. Ruby, yard, chef, and vagrant artifacts are all configurations that are not desirable to be included within the git repo.

For tracked work there are three states in local git operations: modified, staged, and committed.

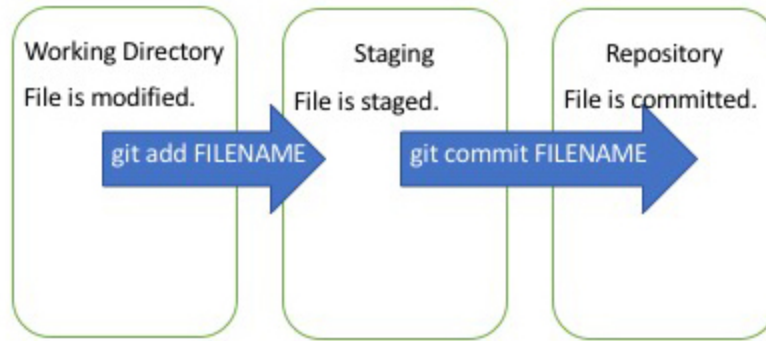


Figure 3-1. Visualizing Git Workspace

1. Modify your project and the work is in a *modified* state.
2. Use *git add* and the work is in a *staged* state.
3. Use *git commit* and the work is in a *committed* state.

When you make edits to your projects in preparation to add them to a git repository, you bundle a set of edits into something called a commit. A *commit* is a collection of actions comprising the total number of changes made to files under version control. Git doesn't make copies of the changes or keep a changeset like many other version control system. Instead, git takes a snapshot of all the files at that moment and stores a reference to the snapshot. If a file hasn't changed, Git doesn't store the file again. Git links to the previous identical file it has already stored.

```
$ git add .gitignore test.md
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

    new file:   .gitignore
    new file:   test.md
```

Using *git add* to the two files tracks them. Git helpfully tells you how to unstage the work, as well as letting you know that the changes haven't been committed yet. You can keep making changes to the files if you want to, but

after making additional changes you would need to stage those changes again with `git add`.

```
$ tree -a
.
├── .git
│   ├── objects
│   │   ├── 27
│   │   │   └── 3c1a9ffdc201dbca7d19b275bf1cbc88aaa4cb
│   │   └── 5d
│   │       └── 2758a8c8d19aece876ae3efa9501f9e4dc1877
│   └── .gitignore
└── test.md
```

Now your `.git` directory has changed. There are 2 files within the `.git/objects` directory. The directory name under objects is the first two characters of the SHA-1 hash of the object. The file name is the rest of the SHA-1 hash of the object that are stored in git.

NOTE

You'll notice that your SHA-1 hashes are different than the ones listed in the above figure because you are committing your work at a different time and with potentially different content in your `test.md` file.

The SHA-1 hash is computed based on the data, the commit message, the time when this work is committed, and a few more details. This is how git maintains integrity. Changes will be reflected in the computed hash.

They are compressed and encrypted so you can't look at them with utilities like `cat`, but you can use `git cat-file` to examine these objects.

```
$ git cat-file -p 273c1a9ffdc201dbca7d19b275bf1cbc88aaa4cb
This is a test
```

With `git cat-file`, you can look at the objects that git stores. With the `-p` flag, you print out the content. Pass the SHA-1 hash which is a combination of the directory name and file name from within the `.git/objects` directory.

Commit changes using *git commit*. Your default editor opens up unless you have specified a specific editor within *gitconfig*.

```
Test Commit and initial ignore file.
```

```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# On branch master
#
# Initial commit
#
# Changes to be committed:
#       new file:   .gitignore
#       new file:   test.md
```

By not specifying a message to be included at the command line, the default editor is opened up with some comments that let you know what's happening with this commit. You will see something similar showing a set of changes bundled in a single initial commit. Add a short description of the purpose of the commit.

Saving the commit message will return with a confirmation if this is a successful commit.

```
[master (root-commit) e363c9f] Test Commit and initial ignore file.
2 files changed, 2 insertions(+)
create mode 100644 .gitignore
create mode 100644 test.md
```

After completing the commit, you can verify that the repository has no work waiting to be committed with *git status*.

```
$ git status
On branch master
nothing to commit, working tree clean
```

After the successful commit, you can go back and examine the *.git* directory within the project. There are a few more updates to the objects directory as well as to the logs and refs directory.

```
$ tree -a
```

```
.
├── .git
│   ├── COMMIT_EDITMSG
│   ├── HEAD
│   ├── config
│   ├── description
│   ├── hooks
│   ├── index
│   ├── info
│   │   └── exclude
│   ├── logs
│   │   ├── HEAD
│   │   └── refs
│   │       └── heads
│   │           └── master
│   ├── objects
│   │   ├── 27
│   │   │   └── 3c1a9ffdc201dbca7d19b275bf1cbc88aaa4cb
│   │   ├── 3d
│   │   │   └── f5341648a47f3487bdf569adef990807e34dc6
│   │   ├── 5d
│   │   │   └── 2758a8c8d19aece876ae3efa9501f9e4dc1877
│   │   ├── e3
│   │   │   └── 63c9f8a8695d06f8f848fdb8852c0d8db3d7b
│   │   ├── info
│   │   └── pack
│   └── refs
│       ├── heads
│       │   └── master
│       └── tags
├── .gitignore
└── test.md
```

```
16 directories, 15 files
```

Branching

In the output of the status command, there is a line of output *On branch master*. A branch is a movable pointer to a commit that moves as you add commits to the branch. The default or base branch name by default is called ‘master’ and points to the last merged commit on *master*. These examples illustrate working on the default or master branch for these commits.

Working with Remote Git Repositories

You can also work with an existing project on your local system. Cloning in git is making an exact copy of a repository.

You clone a project to your local system with the *clone* subcommand:

```
$ git clone git@github.com:iennae/ms-git.git
```

When you clone the project from a remote repository, you make an additional copy of the repository. Any changes you make to the local repository do not automatically sync to the remote repository. Additionally, any changes made to the remote repository don't automatically get updated to your local copy.

This is critical to understand about git in general. If something seems to be going wrong with your local repo, stay calm. You haven't broken the remote repository.

This time, rather than working directly on the master branch, work on a short-lived branch named after an issue. That way changes are associated with the feature or issue when submitted back to the shared git repository.

When you create local branches until you push them to remote repositories they are just local to your repository. This makes branches super speedy to create. It also means you can commit changes to your local repository offline as needed.

Create a named branch with the *-b* flag to the *checkout* subcommand. In this example, I show creating a branch named `issue_1`.

```
$ git checkout -b issue_1
```

This creates a new branch with a pointer to the current location in the code.

NOTE

You can also create a branch with `git branch NEWBRANCH`. To work on the new branch, you follow the git branch with a `git checkout NEWBRANCH`.

After updating a contributing document in the project, check the status of the git repository, add the new file, and commit the changes.

```
$ git status
On branch issue_1
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    contributing.md

nothing added to commit but untracked files present (use "git add" to track)
$ git add contributing.md
$ git commit
[issue_1 d6fd139] Add Contributing Doc.
 1 file changed, 6 insertions(+)
 create mode 100644 contributing.md
```

This updates the branch to point to this latest commit object. You could continue making changes in this branch related to this issue prior to pushing the work back.

If you forget to include a change within a commit, rather than having multiple commit messages, you can update a commit with the flag `--amend`.

```
$ git commit --amend
```

This will launch the editor so that the commit log message can be updated. It's possible to add a flag of `--no-edit` if you're fine with the commit log message.

Once you are ready to share back to the remote repository, there are a variety of different workflows that can be adopted. It's important to identify the workflow in use for an established team prior to just following these instructions.

If the remote repository is one that you have privileges to write to, you can push changes back to the shared git repository on the `issue_1` branch.

```
$ git push origin issue_1
Counting objects: 3, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 627 bytes | 627.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
remote:
remote: Create a pull request for 'issue_1' on GitHub by visiting:
remote:   https://github.com/iennae/ms-git/pull/new/issue_1
remote:
To github.com:iennae/ms-git.git
* [new branch]      issue_1 -> issue_1
```

When updating a short-lived feature branch, any configured tests run automatically on the remote repository. Once they pass successfully, you can initiate a pull request. A pull request, commonly referred to as PR, is a mechanism signaling that a contribution is ready to be reviewed and integrated into the main branch. Depending on the processes within the organization or project, the form that a PR takes may vary. Once the PR has been merged successfully, changes will be merged into the master branch in the remote repository.

Resolving Conflicts

When working with others, it's entirely possible to run into conflicts. Conflicts can occur when git can't figure out what to do when two different commits have changed the same file. To fix the problem, you need to resolve the conflict and tell git what to do.

For simple conflicts, just pull the changes from the default branch on the remote repo and fix the marked sections. After testing and verifying locally, stage and commit the changes locally then follow the standard merging process in use within the project.

Sometimes conflicts can get quite complicated. A recommended practice is to commit often and share work back by pushing back to the shared git repository regularly. This way everyone can see what work is in progress and it doesn't diverge too much before getting integrated back into the default branch. It's more challenging to delete a change or remove the commit once it's been pushed back to the shared repository, but worth the extra communication and coordination with the team.

Sometimes during a code review, you might realize that only part of a pull request is usable. It's always an option to make someone redo their pull request, but it's also possible to select specific commits, or *cherry-pick* from the pull request. Cherry-picking allows you to take what is helpful or necessary to get something working. The recommended practice is to bundle different changes into separate commits within a single pull request so that work can be cherry-picked if needed. For example, if you are implementing the test and the feature together, separate the test and the feature into different commits. This way, if the test is a valid test and does the work, but the feature implementation isn't quite right, the test can be cherry-picked into the default branch even if the rest of the code can't.

Often sysadmins are working on multiple tasks, and they may be regularly interrupted by urgent requests. It's ok because git will let's a sysadmin work on different things at the same time without forcing them to include incomplete work.

- If there are any uncommitted changes, stash them with git stash.
- Start from the default branch so that it is the parent for the new feature branch to complete the emergency work. It's important that the emergency work isn't blocked by any work in progress on a different branch.
- Pull changes from the remote shared repository to the local default branch so that it's up-to-date to minimize any potential conflicts.
- Create a hotfix branch.

- Make changes, test, stage, commit, push, and PR with the hotfix branch following the same process as regular changes.
- Bring back any stashed work with `git stash list`, and `git stash apply`.

Fixing Your Local Repository

While it's always possible to start over with a new clone of your repository, there are other options to help fix your local repository.

If the local repository gets into an undesirable state, and you want to reset the working state to the last commit or a specific commit you can use the *reset* submodule.

```
$ git reset FILE
```

This replaces `FILE` in the current working directory with the last committed version. You can specify a specific SHA-1 hash or a branch to reset to.

You can reset the entire project including uncommitted work with the *--hard* flag:

```
git reset --hard SHA
```

Any work in the history of the repository after the specified SHA will be lost.

Advancing Collaboration with Version Control

A core element of effective modern operational practice is collaboration. You may collaborate by pairing over code or asynchronously through code reviews. There are a few ways to level up your collaboration practices with version control.

- **Credit all collaborators** when pairing on code. Crediting folks ensures that every author gets attribution in the PR as well as updating their contribution graph on GitHub. At the end of the commit message add two blank lines followed by a “Co-authored-by:” trailer and the collaborator’s name and email address. This should be the configuration that the collaborator uses in their git config for *user.name* and *user.email*.

Commit Message describing what the set of changes do.

Co-authored-by: Name <email@example.com>

- **Have at least one reviewer** other than yourself before merging code. Separate reviewers helps us to build context within the team to ensure that more than one person understands the intent and impact of change. You can set up required reviewers in the GitHub interface which will block merges without an admin override.
- **Write quality commit messages** explaining the context for the changes within the commit. Examining the commit tells you what changes; the message should have more information including the why and how. Quality commit messages help give you context when you are reviewing changes as well as when you need to revisit an issue months later.

A benefit of git as a version control as mentioned before is fast local branching. When you work on projects you can do so in isolation without impacting any of the work that your team members may have in progress. This helps minimize conflict resolution that may arise with version control that relies on centralized locking.

Collaboration is not just a point in time activity. Adopting these practices improves how your team works together now, and helps reduce the risks of future edits to the project as code and configurations move or are rewritten.

Wrapping Up

In this chapter, you looked at git, a distributed version control system. I discussed the fundamentals, branching, and recommended practices of using git. I also examined version control collaboration in practice with git and GitHub.

¹ DORA's annual State of DevOps report: <https://devops-research.com/research.html>

Chapter 4. Local Development Environments

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 4th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at vwilson@oreilly.com.

You might wonder why talking about a development environment belongs in a book about contemporary system administration. While you may not do a lot of coding day-to-day for a product, there are a fair amount of standard tools that are essential for working with teams.

A local development environment is the set of tools and technologies that reduces challenges to collaboration, saving time and effort rather than working in silos to replicate work to set up an environment.

The goal isn’t to impact an individual’s productivity by limiting the tools that they need or want. One part of your local development environment is going to be familiar tools and technologies set up in a more consistent automated fashion. Your team or organization may define the other part of your local development environment to help folks onboard and contribute to projects more quickly.

Setting up a local development environment includes choosing a text editor, installing or updating core languages, and installing and configuring applications.

NOTE

One thing I won't cover in this chapter explicitly is the whole host of available tools and utilities available to you.

For example, many sysadmin books have whole chapters dedicated to learning how to use the command line, built-ins, and shell coding. I had to make tough choices about what to include with a focus on how to help folks uplevel skills that they have. Based on the availability of existing high quality materials on these topics, I didn't include them in this book. It's incredibly important to have comfort with the command-line and shell scripting.

Many interviews for system administrators will include questions to gauge your familiarity with the command-line and tools available.

If these are areas you need to level up your skills, a few resources I recommend include:

- Classic Shell Scripting by Nelson H.F. Beebe and Arnold Robbins
- [Learning the bash Shell by Cameron Newham](#)

Choosing an Editor

You're probably familiar with text editors. Having the right text editor for specific work can reduce the overhead needed to understand the multitude of contexts required, whether it's developing infrastructure as code or writing tests to verify configurations. The work that a sysadmin needs to do with an editor includes developing scripts, codifying infrastructure, writing documentation, composing tests, and reading code.

You can love your editor, but you may need to learn to like another one that helps you write code and tests that integrate well with version control and a linter while collaborating with others on your team. For example, using something like Visual Studio Code (VS Code), Eclipse, or Sublime, which have several extensions that can customize and ease processes, may be of great benefit. I'll use VS Code in the examples within this chapter.

NOTE

Be open-minded about trying and adopting new tools. It's possible that your editor has all the features that you want and need out of it, especially if you've customized it. That's great. For folks who don't have that familiarity, building and learning that context from scratch along with all the unique mechanisms to operate an editor like vi or emacs may not be the best use of time, especially when there is so much to learn to be an effective system administrator.

Let's dig into some of the benefits to look for in a text editor. The base editor should be usable without customization. Customization adds specificity to what you're working with. Extensions that are widely tested and used add context for work you need to do, whether it's working with your cloud provider or your version control repository.

Minimizing required mouse usage

Being able to keep your hands on the keyboard (rather than moving to use a mouse) helps to maintain focus and flow of work. Key bindings help to quickly do particular tasks like open a new file, open up the terminal within the browser, save a file, and open up multiple windows for side by side editing. Key bindings are a feature of popular Unix editors like vim and emacs, but the combinations might not be what folks expect. Standard key bindings across applications on an operating system mean that using the application feels more intuitive.

Splitting the screen up vertically and horizontally is helpful when you want to compare two files or have both files open for context. Sometimes, it's nice to be able to have the same file open multiple times, especially when it's a large file. Then, it's possible to look at the bottom and top of the file at the same time.

Untitled-1

Dockerfile x

README.md

1

FROM node:latest

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

version: '2'

services:

db:

image: mongo:latest

api:

build: .

image: mern-crud

command: nodemon server

ports:

- "3000:3000"

environment:

- "DB=mongodb://db/mern-crud"

- "CORS=1"

- "DEBUG=express:*

volumes:

- ./usr/src/app

- /usr/src/app/node_modules

depends_on:

- db

web:

image: mern-crud

command: npm start --prefix react-sr

ports:

- "4200:4200"

PROBLEMS

OUTPUT

DEBUG CONSOLE

TERMINAL

Filter. Eg: text, **/*.ts, !**/node_modules/**

No problems have been detected in the workspace so far.

Figure 4-1. VS Code Side by Side Editing

Here I have two files open at the same time: a Dockerfile and docker-compose.yml file. I can scroll down each individually to make sure I've got the right ports configured.

There is an integrated terminal within VS Code, so you can bounce between editing content within the editor and running commands in the shell.

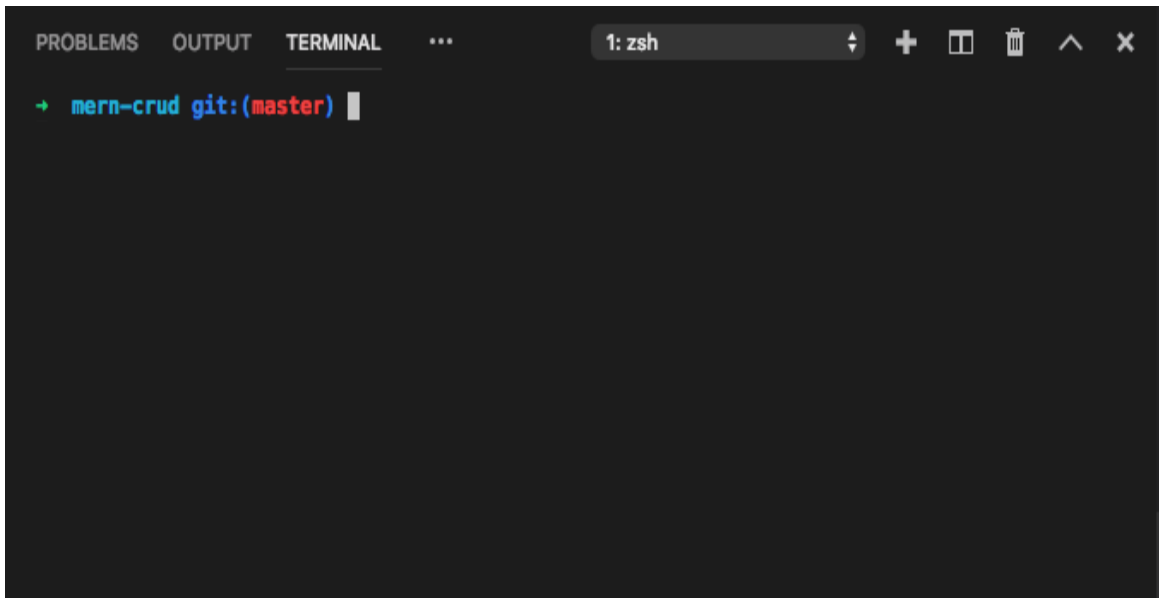


Figure 4-2. VS Code Integrated Terminal

You can open additional terminals. When you split the terminal, you can run multiple commands at the same time.

Integrated Static Code Analysis

You can speed up development and reduce potential issues by adding static code analysis extensions for the languages in use.

For example, you can install shellcheck and the shellcheck extension. Then, you can see problems as you write shell code.

```
(sum=2+2)
echo $sum
```

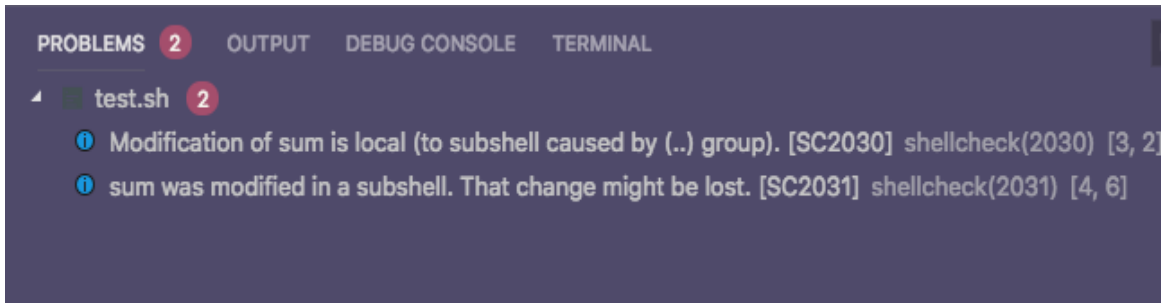


Figure 4-3. VS Code shellcheck lint errors

Running shellcheck as the code is written shows any problems so fixes can be implemented when the code is most fresh in mind.

```
((sum=2+2))  
echo $sum
```

There are linters for all kinds of files that you might work with from infrastructure as code configurations to specific languages. Within the editor, you can customize the options for how the lint runs, allowing you to run linting as you type code or after you save.

Easing editing through auto completion

IntelliSense is a code completion tool that improves the editing experience by providing educated guesses about what you are trying to do. As you type, options will pop up with suggestions for autocompletion. Some languages have better completions automatically; you can add extensions to improve others.

For example, you can add the *Docker* extension to the VS Code editor, which eases the creation of a Dockerfile, the text file that contains the build instruction for a docker container. It has a list of snippets that correspond to valid *Dockerfile* commands so you can quickly build out a new Dockerfile. For an existing Dockerfile, you can hover over the Docker command and get a detailed description of what it's doing.

```
1 FROM node:latest
2 Set the baseImage to use for subsequent instructions. FROM
3 must be the first instruction in a Dockerfile .
4 FROM baseImage
5 FROM baseImage:tag
6 FROM baseImage@digest
7 Online documentation
8 RUN npm install
9
10
11 COPY react-src/package.json /usr/src/app/react-src
12 RUN npm install
13
14 COPY . /usr/src/app
15
16 EXPOSE 3000 4200
17
```

Figure 4-4. Hovering over Docker Command in VS Code

Indenting code to match team conventions

Rather than debate whether spaces or tabs are more readable, control the text indentation, whether it's spaces or tabs or the specific count of white space.



Figure 4-5. VS Code Spacing of Current File

In VS Code, this is visible in the lower panel along with other conventions about the file type. From the Command Palette, you can change the spacing from tabs to spaces.

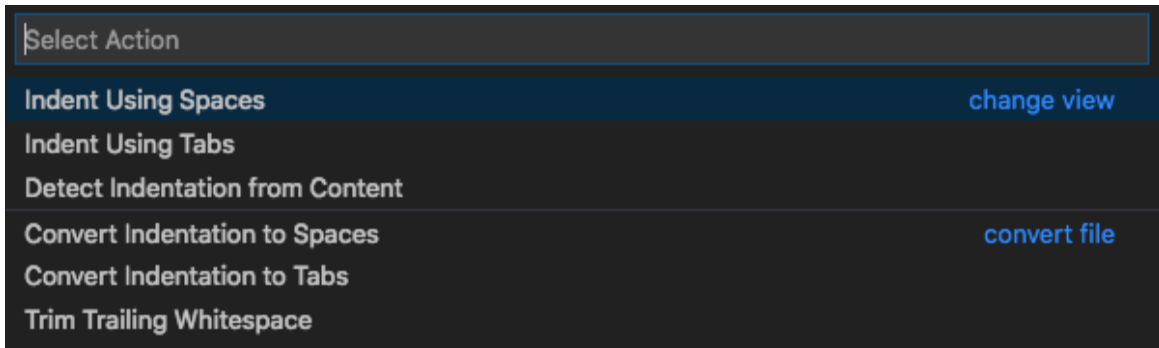


Figure 4-6. Changing Spacing in VS Code

You can convert the amount of spacing currently in use within a file to conform to new requirements as well.

Collaborating while editing

One of the most compelling features is the ability to collaborate with VS Code. Each participant maintains their customized environment with separate cursors while collaborating with the **Live Share** extension. Pairing building infracode doesn't require individuals to be sitting in the same space, and everyone experiences editing with their preferred style.

Integrating workflow with git

As you work on a project, it's helpful to see the changes that you've made and whether you've committed those changes. As mentioned in an earlier chapter, it's critical to use version control.

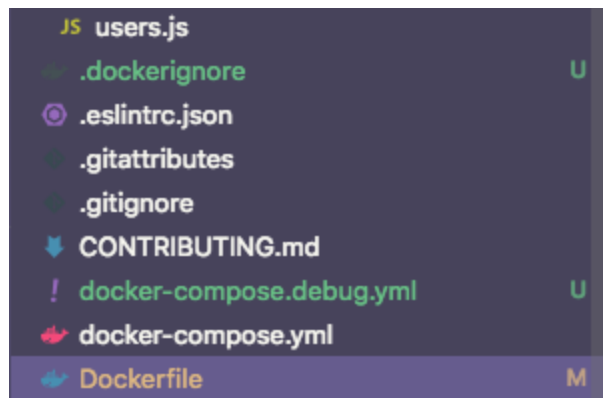


Figure 4-7. VS Code Explorer View of Modified Files

Here the *M* icon shows that there are modified files. The *U* icon shows files that are updated and unmerged.

Extending the development environment

A VS Code extension that came out in Sprint 2019 was the **Remote Extension**. This extension allows individuals to use a remote system for “local” development with familiar local features.

So new team members can spin up a new working development environment in a repeatable fashion without modifying anything on their new system.

You also can leverage more powerful and ad-hoc systems in a remote environment to spin up complex environments that don’t slow down your laptop or desktop system.

NOTE

For more awesome extensions, check out the [Curated List of Visual Studio Code Packages and Resources](#).

Selecting Languages to Install

While you might not spend time developing applications, honing development skills in shell code and at least one language is essential.

If you can work comfortably from the command line and read and write scripts, you will experience a whole host of benefits, like being better able to collaborate with your team to build the functionality that improves productivity of the team as a whole.

Automating toil work —from building faster ways to open JIRA tickets with pre-populated meta information to scanning compute instances for systems out of compliance with required versions —frees up the team’s time to focus on areas that require human thought and creativity.

Bash is a reasonable choice in most environments as it's available on Linux as well as current versions of Windows. That said, when shell scripts start getting longer than about 50 lines, they start becoming hard to manage and understand. When scripts are hard for folks to understand, they become part of fragility within the team that no one wants to touch or maintain.

How do you choose a specific language to invest time and energy in?

Languages like Python, Ruby, and Go become much more useful to write utilities. Depending on what is already in use within the team, that's where you should spend energy on leveling up your skills.

Additionally, it can be beneficial to learn how to read whatever language(s) your development team uses. For example, being able to read Node.js is helpful when collaborating with software engineers on Serverless Functions. When something isn't working as expected, it can be helpful to see whether the "as expected" is the code or tests of that code.

NOTE

Sometimes, your operating system will include a version of the language. Often this is an outdated version, and you'll need to update to leverage the latest features of the language. You could update to the latest version via a package manager like Homebrew on Mac OS. Changing the system included language isn't recommended practice. Instead, install the desired version separately and set execution paths appropriately to prefer the later version. Explicit external installation will help prevent any system instability due to modifying software that the system might be using. It also helps eliminate undefined dependencies in environments.

There are different reasons to choose a particular language. Languages may be chosen based on a team member's experience. They may be selected based on features of other desired software. Sometimes software is chosen based on it using a particular language. For example, some folks choose their infrastructure automation based on whether it's Ruby or Python because of existing skills within the team.

It's ok to choose languages and technologies based on existing skills on the team. Recognize the reasons why your team chose one tool or language

over another (and document those choices). Once a tool is chosen and implemented, it's hard to change cleanly.

For example, organizations trying to migrate from one infrastructure automation tool to another, often end up with both tools in use rather than a clean migration. Multiple tools with overlapping concerns adds confusion and complexity to the environment.

There is no one right language to learn as a system administrator. Be careful to balance your experience and comfort with a specific language with the features and the rest of the team's skills.

Installing and Configuring Applications

Beyond your editor and a specific set of languages, there are common applications and configurations to improve your experience. Here are a few:

- The Silver Searcher

The Silver Searcher, or *Ag* for short, levels up searching through code repositories. *Ag* is fast and ignores file patterns from *.gitignore*. It can be integrated with editors as well. When debugging errors or other “needles in the haystack” of code, it can be super helpful to search for a specific string to understand how it's called.

- *bash-completion*

Modern shells provide command completion. This allows you to start typing the beginning of a command hit TAB and see potential completions. *bash-completion* extends this feature and allows you to add additional completion features. For example, this could be used to prepopulate resources you need access to. Extensions are shareable across the team.

- *cURL*

Curl is a command-line tool and library to transfer data. You can use it to verify whether you can connect to a URL, which is one of the first

validations when checking a web service, for example. You can also use it to send or retrieve data from a URL.

NOTE

Some people object to using `curl bash` or `curl | bash`. This pattern comes from application install instructions that include a reference to *curl URL | bash* to install the application. The problem isn't a reflection on curl the application, but on trusting a random URL and running it randomly.

- Docker

Docker provides a mechanism to create isolated environments called containers. A Dockerfile encapsulates the OS, environment files, and application requirements. You can add a Dockerfile to a project and commit it to version control.

With Docker installed and access to a Dockerfile, then onboarding a new collaborator to a project can be as straightforward as running *docker run* to get a working test environment up. This test environment would even match more closely to a production environment if running production on containers.

- hub

If you are using git as version control and GitHub as the project repository, hub extends git functionality that helps with GitHub tasks from the command-line.

- jq

Jq is a lightweight and flexible command-line JSON processor. Combined with cURL, you can process JSON output from the command-line.

- Postman

Postman is a visual tool used in API testing. It's really useful for exploring APIs, especially when there are complex headers or when working with

JSON.

Requests can be bundled up into collections that you can export and share.

An example of where Postman can be really useful is when you need to build a health check for a service accessed through an API.

- shellcheck

ShellCheck is a utility that shows problems in bash and sh shell scripts. It can identify common mistakes and misused commands. You can ignore specific checks if they are not checks your team wants running against your code with a configuration file.

- tree

Tree is a utility that lists contents of a directory in a tree-like format. It can be helpful to visualize the structure of a file system, especially for documentation.

Wrapping Up

In this chapter, I focused on the broader nature of setting up a repeatable and shareable local development environment. I started with some guidance on identifying the quality features of a text editor. From there, I identified programming languages to install or upgrade as part of the environment. I finished up talking about installing and configuring applications. With a little effort, it's possible to bake this into a personal repository in version control to quickly get productive alongside any specific team or organization scripts and projects.

Chapter 5. Testing

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 5th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at vwilson@oreilly.com.

We’re human. We make mistakes. When organizations punish people for these mistakes, it can feel senseless as outcomes from this punishment increase errors. Fear of repercussions increases the risk of errors and slows down releases. Tests help us change how to talk about the work and reduce that fear. Rather than expecting individuals to be perfect all the time and work from a place of fear, as a team, we build safety nets that help us work from confidence.

Often overlooked or misunderstood, testing is a critical set of work for system administrators. Testing helps you deliver a working product, eliminates single points of knowledge, and increases confidence that problems won’t easily make it to end-users.

There is also an expectation for every engineer to have fundamental knowledge about all areas of the product lifecycle. For companies to adopt continuous testing, you must do your part in applying testing principles and practices in your scope of work.

Testing is a broad subject, so in this chapter, I focus on general testing principles. By the end of this chapter, you'll know the difference between linting, unit, integration, end-to-end, and exploratory testing and how to assess a project's overall testing strategy.

Later chapters will address specific types of testing work that sysadmins do, including:

- Infrastructure Testing,
- Testing in production,
- Failover and capacity testing, and
- Security and compliance testing.

Why should Sysadmins Write Tests?

Often we find ourselves all agreeing that tests are necessary “but”

- Tests are expensive to write,
- Tests are hard to write,
- I'm not a/the tester.

So with these challenges, why should you write tests?

- Write tests to increase your team's confidence in your code.
- Write tests to speed up delivery of working tools and infrastructure with increased confidence.
- Write tests to eliminate yourself as a single point of failure in your organization so you can tackle new projects.

Differentiating the Types of Testing

Software engineers or testers implement different types of tests for projects. It benefits you to know what tests are completed against a service or product to understand the quality level of the product and identify how costly it may be to maintain and support. Some tests may have tools or patterns in use that can help you to eliminate manual processes.

Having a solid foundation in testing is crucial to help you write more effective tools and infrastructure code. Understanding the different types of testing, including their benefits and drawbacks, help you create the appropriate level of testing.

NOTE

There is no exact definition of these test types, so depending on the team, there may be slightly different test implementations. Just as I recommend folks on a team come together with a common understanding, I'm defining these test types to help you read the next chapter and use testing in practice.

Check out this example of how some teams at [Google reframed how they categorized tests with sizes instead](#).

Linting

Linters are a testing tool for static analysis of code to discover problems with patterns or style conventions. Linting can help identify issues with code early. It can uncover logic errors that could lead to security vulnerabilities. Linting is distinct from just formatting changes to code because it analyzes how code runs, not just its appearance.

TIP

There's no tests to write with linting so you don't have to learn an extra language. Adopt linting as an initial additional testing practice in a build pipeline to level up the quality of code.

Note that folks should never make sweeping changes without talking to the team. There can be undocumented reasons why a team has adopted a particular style or convention.

There are three primary reasons to adopt linting in your development workflow: bug discovery, increased readability, and decreased variability in the code written.

- **Bug discovery** helps identify quick problems that could impact the underlying logic of code. If you discover a problem while you are writing the code, it's easier to fix. Because the code is fresh in your mind, you still have the clarity about what you intended to write. Additionally, for new functionality, it's less likely that others will have dependencies on the code that you are writing.
- **Increasing readability** helps to understand the code more quickly. Code context ages quickly; remembering what code was intended to do can be difficult. Debugging hard to read old code is even more difficult, which then makes it harder to maintain, fix, and extend functionality in the code.
- **Decreasing variability** in code helps a team to come to a set of common standards and practices for a project. It helps a team ensure cohesiveness of code. Encoded style prevents arguments over team conventions, also known as bikeshedding.

You can add linting plugins for many languages to your editor for near-instantaneous feedback about potentially problematic code. This allows you to fix potential issues as they arise instead of after putting together your code into a commit and submitting a pull request for review.

You can also configure the linter to ignore rules or modify the defaults of the rules that the team doesn't want to adopt with a configuration file.

For example, your team may want to enable longer length lines in a ruby project. Rubocop, the ruby language linter is configured with a rubocop.yml file that is stored in version control with the source code of the project.

Example 5-1.

Metrics/LineLength:

Max: 99

While individuals may have preferences about whether to use 2 or 4 spaces, or whether to use tabs instead of spaces within their code, common practice within a project can be identified and resolved within the editor. This helps make the project more readable as well as conform to any language requirements. The editor Visual Studio Code with the appropriate plugins automatically highlights problematic issues.

Unit Tests

Unit tests are small, quick tests that verify whether a piece of code works as expected. They do not run against an actual instance of code that is running. This makes them super helpful for quick evaluation of code correctness because they are fast (generally taking less than a second to run). With unit tests, you aren't checking code on real instances, so you don't receive insight into issues that are due to connectivity or dependency issues between components.

Unit tests are generally the foundation of a testing strategy for a project as they're fast to run, less vulnerable to being flakey or noisy, and isolate where failures occur. They help answer questions about

- design,
- regressions in behavior,
- assumptions about the intent in code, and
- readiness to add new functionality.

It's essential when unit testing your code that you aren't checking the software that you are using, just the code that you are writing. For example, with Chef code that describes infrastructure to build, don't write tests that test whether Chef works correctly (unless working on the Chef code itself). Instead, write tests that describe the desired outcomes and check the code.

Examples of a unit in infrastructure code might be a managed file, directory, or compute instance. The unit test to verify the example units of infrastructure code describes the file, directory, or compute instance

requirements including any specific attributes. The unit test describes the expected behavior.

Integration Tests

Integration tests check the behavior of multiple objects as they work together. The specific behavior of integration tests can vary depending on how a team views “multiple objects.” It can be as narrow as 2 “units” working together, or as broad as different, more significant components working together. this doesn’t test each component of the project; it gives insight into the behavior of the software at a broader scope.

Failing integration tests aren’t precise in problem determination.

In general, an integration test should run in minutes. This is due to their being increased complexity in setting up potential infrastructure dependencies as well as other services and software.

End-to-End Tests

End-to-end tests check if the flow of behavior of an application functions as expected from start to finish. An end-to-end test tests all the application and services that were defined by the infrastructure code on the system and how they worked together.

Three reasons end-to-end testing should be minimal in their implementation are that they are sensitive to minor changes in interfaces, and take significant time to run, write and maintain.

End-to-end testing can be very brittle or weak in response to changes. End-to-end tests fail and builds break when interfaces at any point in a stack change.

Documentation about the interfaces can be wrong or out-of-date. The implementation may not function as intended or change unexpectedly.

End-to-end testing increases our confidence that the complete system, with all of its dependencies are functioning as designed.

Additionally, end-to-end test failure is not isolated and deterministic. Test failure may be due to dependent service failure. End-to-end tests checking specific function output require more frequent changes to the test code.

For example, a test environment located in an availability zone on Amazon with network issues may have intermittent failures. The more flakey the tests, the less likely individuals will spend effort maintaining those tests, which leads to lower quality in the testing suite.

End-to-end tests can also take a long time to implement. These tests require the entire product to be built and deployed before the tests can be run. Add on the test suite, and they can be quite lengthy to complete.

Even with these challenges, end-to-end tests are a critical piece of a testing strategy. They simulate a real user interacting with the system. Modern software can be comprised of many interconnected subsystems or services that are being built by a different team inside or outside of an organization. Organizations rely on these externally built systems rather than expending resources into building them in house(which incidentally has even higher risk). System administrators often manage the seams where different systems built by different people are connecting.

Examining the Shape of Testing Strategy

We can examine the tests that exist for a project and qualify the strategy as a shape based on the number of tests. This informs us of potential gaps where additional testing is needed or tests that need to be eliminated.

One of the recommendations in software development is that the quantity of testing should look very much like a pyramid. **Mike Cohn described the Test Automation Pyramid** in his 2009 book *Succeeding with Agile*.

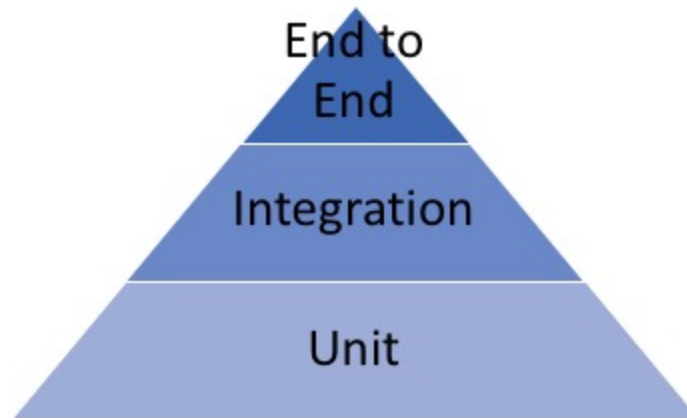


Figure 5-1. Recommended Test Strategy with Shape of Pyramid

Over time the pyramid has been modified and updated, but the general premise remains. The pyramid codifies what is needed when it comes to testing, and stresses the importance of the different types of tests while recognizing that tests have different implementation times and costs. Over time, this pyramid has been adopted and shared widely with the occasional minor modifications. In the pyramid model, approximately 70% of the volume is taken up by unit tests, 20% for integration tests, and 10% for end-to-end.

A good rule is to push tests as far down the stack as possible. The lower in the stack it is, the faster that it will run, and the faster it will provide feedback about the quality and risk of the software. Unit tests are closer to the code testing specific functions, where end-to-end is closer to the end-user experience, hence the pyramid shape based on how much attention and time we are spending writing the particular type of tests.

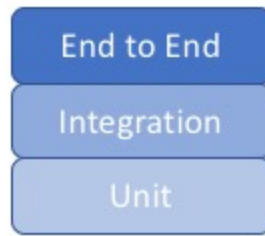


Figure 5-2. Test Strategy with Shape of Square

If a project's tests resemble a square meaning, there are tests equally at every level, that may be an indication that there are overlapping testing concerns. In other words, there is testing of the same things at different levels. This may mean longer test times and delayed delivery into production.

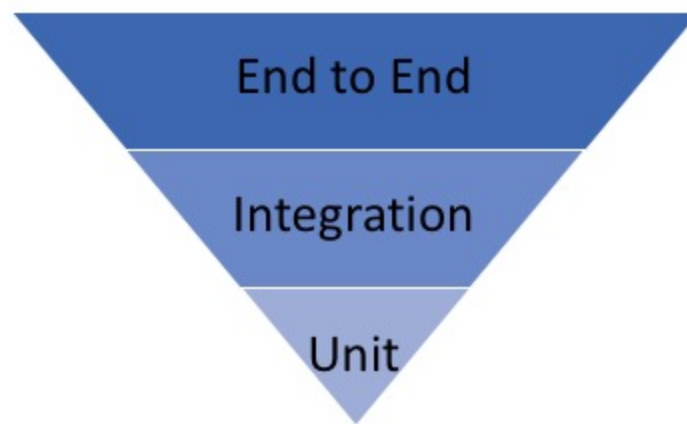


Figure 5-3. Test Strategy with Shape of Inverted Pyramid

If a project's tests resemble an inverted pyramid meaning there are more end to end tests, and fewer unit tests, that may be an indication that there is insufficient coverage at the unit and integration test level. This may mean longer test times, and delayed code integration as it will take longer to verify that code works as expected. Increasing the unit test coverage will increase the confidence of changes in code and reduce the time it takes to merge code leading to fewer conflicts!

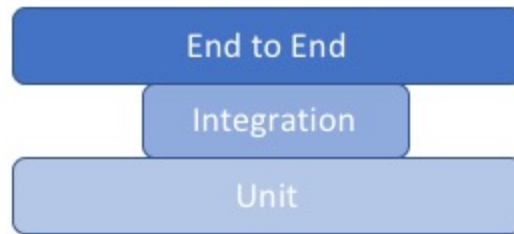


Figure 5-4. Test Strategy with Shape of Hourglass

If a project's tests resemble an hourglass meaning there are more end to end tests compared to integration tests, that may be an indication that there is insufficient integration coverage, or that there are more end-to-end tests than are needed. Remember that end to end tests are more brittle so they will require more care and maintenance with changes.

Having an hourglass or inverted pyramid will also indicate the potential that more time is spent on maintenance of tests rather than developing new features.

Understanding these shapes of testing strategies can help you understand how much invisible work is being passed on to the system administration team to support a project.

That said, infrastructure code testing does not always follow these patterns. Specifically, infrastructure configuration code testing does not benefit from unit tests except when checking for different paths of configuration. For example, a unit test is beneficial when there are differences in platform requirements due to supporting different operating systems. It can also be beneficial where there are differences in environments between development, testing and production or making sure that production API keys don't get deployed in development and testing.

Remember, push tests as far down the stack as possible. With infrastructure, due to its nature, integration testing might be as far down as it makes sense.

Existing Sysadmin Testing Activities

Have you ever run through installing a set of software on a non-production or non-live system, watching to see how the system responded and whether there were any gotchas that could be user impacting?

This process is a type of testing that sysadmins get good at without thinking about it as an “official” type of testing. This manual testing approach is known as *exploratory testing*. The goal with exploratory testing is to help discover the unknown by experimenting with the system looking at areas of the system that may need more subjective analysis as to whether they are in a good state. In [Exploratory Testing Explained](#), James Bach defined exploratory testing as “simultaneous learning, test design, and test execution.”

Sysadmins can level up exploratory testing skills by adopting more rigor in their analysis. This means defining testing objectives with short feedback loops. Working with software engineers and testers, sysadmins can help shape the testing so that the team eliminates some of the manual testing.

NOTE

It's helpful to have new team members explore products and processes as part of their onboarding. They can bring unbiased insight to level up quality to correct problems with the product and processes and clear up misunderstanding and inconsistencies that may already exist within the team.

When Tests Fail

You've run a test and it passed. Time to bundle up the test and add it to your automation suite! Generally, passing tests tell you that the application works as expected. Passing tests tell you that you haven't found a problem - yet.

To really assess and understand how to adopt tests into automation, you need to understand how tests fail. Failing tests tell you more than “found a problem with your code”. Examining why tests fail and the different kind of

feedback you are getting allows you to plan a roadmap and automate responses as possible.

I will share the four types of issues that you may discover when testing. You need to think about these as you create test automation. Automation without the ability to act on the feedback you get back from the tests, just adds work which detracts from the value you could be bringing to your customers. You can plan how to assess the different outcomes of tests and implement controls around what can be automated and what needs human intervention.

When I think about test failures, there are 4 main types to plan for:

- Environmental problems (most likely)
- Flawed test logic
- Changing assumptions
- Code defects (least likely)

For an established project and code, defects are often blamed for test failures, but really they are at the bottom of the stack when thinking about “why did this test fail”. You should look for code defects, but you also need to make sure that things that are harder to discover and identify are ruled out so that you don’t spend time editing and changing code.

Environment Problem

Environmental problems can be super frustrating because so much can go wrong especially in the larger end to end tests that are testing between different services. This is one of the reasons that having sufficient unit test coverage is important as unit tests are not as vulnerable to environment problems. There are many issues with environments that can arise including:

- The testing environment doesn’t match the production environment in scale or function.

- Maybe elements of functionality are costly for example monitoring agents that shouldn't have an impact but do.
- No local testing environment setup due to lack of understanding that it's possible to have a local testing environment.
- Dependencies aren't locked down and vary in environments.
- Third party CI/CD services having failures.

These are just a few examples of environmental problems that can cause tests to fail.

Problems with shared test environments can lead to folks insisting that no testing environment is needed, and instead to test directly in production with feature flags and canary testing. Feature flags make features available to a subset of users in a controlled manner or to turn off a feature if necessary. Canary testing allows you to provide a subset of users access to a feature or product to determine if the quality of the release is ok, and if so continue deployment. If the users report issues then you can migrate them back to the standard experience.

There is no way to replicate a test environment that matches production. So feature flags and canary testing in production are crucial ways to improve feedback and reduce the risk of mass changes to production.

They don't replace the need for fast and early feedback to developers on an ongoing basis from the test environment. When you have long lead times for feedback (i.e. waiting for deployment into production) you end up losing context of the work you are doing. In this situation, it's the difference between minutes of time to potentially days, which adds up over time.

Additionally, shared test environments are critical environments for creating a place for experimentation and exploratory testing. Shared test environments are often seen as an expense without a proper evaluation on return on investment.

One way to minimize what folks might consider a waste of resources is to monitor and manage the creation and decommissioning of test environments

through infrastructure automation. Test environments are available when needed and are consistent and repeatable so that engineers needing to test can get access when they need them. This minimizes the cost of idle systems and wasting engineering time with engineers queueing up to use a single test environment.

Sometimes environmental conditions are completely outside of your control. For example when third party CI/CD services have failures like GitHub, Travis, or CircleCI being down. Outsourcing these services has short to medium term value in terms of not having individuals specialize in ensuring these work as needed locally. Third party services will have failures. It's guaranteed. You have to plan for mitigations. How will you work on code while they are down? How will you test work? How will you deliver value to your customers? What if there is data loss? These are risks you need to factor in. If for example the tests aren't warning of failures, this isn't a guarantee that everything is ok. It could be that the system has had a failure and it no longer thinks that the project is being managed. This happened in the Spring of 2018 when GitHub changed an interface that applications used with an upgrade to their API which led to systems like CircleCI no longer having project state.

Flawed Test Logic

Sometimes the problem that you discover is due to the way you have interpreted the requirements or the way that a customer expressed their needs. The code has been written to do the right thing, but tests are failing. This generally exposes some issue with collaboration or communication. There could be missing, unclear, or inconsistent information. Worse, you may not discover the problem if you aren't testing the right thing. When you do detect failures that can be attributed to flawed test logic then you need to modify the test and review the processes in place that lead to the missing context and address them.

Products evolve over time. Sometimes specifications change from the initial design meetings to developer implementation, tests that at one point were valid can now cause failures.

So if a test failure is due to flawed test logic, fix the test and also assess where the problem occurred:

- Did the initial discussions that were held not include required people?
- Did the requirements gathering not align the testing acceptance criteria with the customer requirements?
- Did feedback not make it back to discussions and design when the implementation changed?

Depending on your development pipeline and the different gates that you have for software to progress, there are different areas where communication and collaboration can fail.

Assumptions Changed

Sometimes you can make assumptions about how something happens and it can kind of be right some of the times, other times not so much but it's not visible until the underlying circumstances change. Maybe some innocuous change in when tests run all of sudden shows that some tests are failing that don't align with any changed code.

This can also be visible when the order of operations of a particular task is changed. This is especially visible with database changes.

Failures that occur due to changed underlying assumptions, are exposing areas where assumptions were made. These were fragilities in the code or tests that were previously not exposed.

Automated tests need to be deterministic, so uncovering hidden assumptions and making them explicit will help eliminate flapping tests. It also might be an area where instead of doing an end to end test, there could be room for tests that are closer to the components themselves so that interfaces changing don't cause failures.

Code Defects

Code defects are listed last on my list because you have created your tests to look for code defects. So when you are assessing the test failure it can be easy to focus first on code defects rather than looking at anything else. Instead look at your environment conditions, consider flawed test logic, and if any assumptions have changed before you dive into addressing code defects.

When you discover a problem with code (really really discover a problem with code and have been able to verify whether it's repeatable, and how often it happens), you need to take care in describing the problem so that the appropriate actions can be taken. Describe the problem in clear specific terms including information about the specific context of what happens and how with the steps to replicate it. Any meta information about the version of the software and infrastructure should be included with the information around context. If the problem was discovered manually, make sure to write a test that will discover it automatically.

Once you write up the defect report, you need to track it and make sure that someone is assigned the responsibility of resolving the problem.

Work with the team to prioritize the report based on what else is in the queue. Once the problem is fixed, verify that it's really fixed revisiting any boundary conditions that might need to be changed and then close the defect report.

If the priority of the bug is not high enough to get worked on or assigned a responsible owner, examine whether the report should stay open.

Long term bug reports remaining open just provide a contextual load on the team.

This isn't to say that all reports should be closed or assigned immediately, but make sure to advocate for problems that are major enough to ensure that they do get worked on.

Failures in Test Strategy

While not literal failures that show up against builds, there are often subtle issues that arise. One metric that is useful to measure is signal to noise ratios in the testing framework, how often is a failed build due to one of the four different test failures and assigning a quality of signal to the tests. Some of the subtle issues that can be uncovered by monitoring and measuring our builds can show you problems in your test maintenance or processes.

Flaky Tests

A flaky test is a test that is nondeterministic. It may pass or fail with the same configuration. These are generally more problematic with more complex tests like integration and end to end tests. When we identify tests of these type, it's important to refactor or eliminate the test.

Some common reasons that a test may be flaky:

- Caches - Does the application rely on cached data? There are many different types of caches, and many ways that a cache can cause problems with testing. For example with a web cache, files that are critical for the rendering of a web page may be cached on the web servers, on edge services like Fastly, or locally within the browser. If the data is stale in the cache at any of these levels, this can cause tests to be nondeterministic.
- Test instance lifecycle - What are the policies with the setup and teardown of the testing instance? If the environment is reused or multi-tenanted it's possible that tests aren't valid or return inconsistent results. This is one of the reasons it's important to do regular hygiene on test environments as well as simplifying the setup and cleanup of all test instances.
- Configuration inconsistencies - When environments are not consistent, or the testing situations don't match up to real world production experiences this can cause problems. For example, if something is time dependent and one environment is syncing to an

NTP server and the other is not there may be conflicts in how the test responds especially in special conditions like during a daylight savings change.

- Infrastructure failures - Sometimes the test itself isn't flaky but it's exposing underlying problems with infrastructure. There should be some kind of monitoring to expose these problems as they occur rather than causing wasted time debugging problems that don't exist.
- Third party services - Your organization will rely on more and more third party services to ensure it focuses on the areas that you compete. When problems occur with those services it can cause problems with your integration and end to end tests. It's important to isolate those challenges and make sure you can pinpoint where problems are occurring much like with your infrastructure failures.

Test environment must be monitored and maintained. Infrastructure automation will help codify the specifications easing the cost of maintenance.

Continuous Integration framework needs to be documented and automated. The build system contains meta information critical to the alignment of test strategy and code.

You have been building critical experiential data in your builds as the older builds logs have system knowledge about experiences and strategies tried before. You might want to consider how you store, retain and process those insights.

Wrapping Up

Chapter 6. Security

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 6th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at vwilson@oreilly.com.

Security is the practice of protecting hardware, software, networks, and data from harm, theft, or unauthorized access. The ultimate purpose of security is to reduce the risk to people. Security is often viewed as being at odds with desirable features and user convenience, which can exacerbate implementation resistance.

It’s impossible to release or manage a perfectly secure application or service when dependencies like underlying libraries, operating systems, and network protocols have security issues. Whether you are building software or deploying open-source or commercial software, plan a layered strategy to minimize vulnerabilities and reduce an attacker’s opportunity to exploit them.

Security incidents are not a matter of if but when. They impact companies financially and reduce users’ trust. The risks may be to networks, physical or virtual machines, applications, or the data being stored and processed. When the pager goes off, you don’t want to discover compromised systems, data corruption, or defaced websites. How do you increase the security of

your systems and services? Tackle security like other difficult problems. Break up the large task of “security” into smaller achievable pieces of work that the team iterates on. Allow feedback and learning to inform and modify the team’s practice of working in collaboration with software and security engineers.

You can’t have perfect security, but you can collaborate with other parts of the organization to establish acceptable levels of security. The amount of security work that every organization needs to do can not be distilled and assigned to one team, especially as the attacks evolve and become more costly and complex. In this chapter, I focus on sharing general security principles that apply to sysadmins so that you can collaborate. By the end of this chapter, you should be able to define security, explain threat modeling, and have a few methods for communicating security value during architecture planning.

Collaboration in Security

With cloud services there is a shared responsibility for security. The more operational burden you hand off to the cloud provider, the more levels of security are taken care of for you. For example, a cloud provider that manages the physical hardware is doing more than just purchasing a server and connecting it to the network for access. They are managing the physical access to that server.

For any service provider your organization uses, ask about and understand their security posture. It isn’t helpful when someone leverages a vulnerability of your provider to tell your customers, “It was our provider’s fault.” You still lose money and the trust of your customers. At minimum find out how the provider handles notifications and the appropriate path of escalation for the discovery of security events.

Whether you use infrastructure, platform, or software-as-a-service from a provider, you are still responsible for some parts of security. Often there is an assumption that security is taken care of for you when you use cloud services, but your organization must configure account and access

management, specify and configure endpoints, and manage data. For example, it doesn't matter if your cloud provider encrypts all data on disk if you configure it to have world wide public access.

NOTE

Different roles exist within security. Just because your organization has a "security team," it doesn't mean that they own all security responsibilities. This also doesn't mean you should do that work without recognition, especially if you are the one sysadmin managing and maintaining the systems. That's a path to burnout. Instead, surface necessary work so that your team can assess and prioritize as necessary.

Borrow the Attacker Lens

Taking a different perspective of the systems you manage can help you to improve security for your managed systems.

Threat modeling is a process by which you identify, prioritize, and document potential threats to your organization's assets (physical hardware, software, data) to help you build more secure systems. Assets are not always well understood or recognized, especially when you haven't designed or deployed the system or service yet. Sometimes the threat modeling process can help you identify explicit data that increases risk to your organization without providing sufficient value, and therefore you shouldn't be storing it.

EXAMPLES OF DIFFERENT DATA ASSETS

Some of the many types of data that your company may collect includes personally identifiable information (PII), personal data, payment card information, and credentials.

- The National Institute of Standards and Technology (NIST) defines **personally identifiable information (PII)** as information that can identify an individual or that is linked or linkable to an individual. An example of PII is an individual's social security number.
- The European Commission defines **personal data** as any information that can directly or indirectly identify a living individual. An example of personal data is a home address.

PII is mainly used within the USA, while personal data is associated with the EU data privacy law; the General Data Protection Regulation(GDPR).

- Payment card information is data found on an individual's payment cards which includes credit and debit cards.
- User credentials are how your site verifies that an individual is who they say they are.
- Examining your data can help qualify your liability based on privacy and data retention laws and regulations.

Next, consider the different vectors of attack also known as the attack surfaces. Attack surfaces are all the potential entry points of intrusion for each asset specific to your organization. For example, look at the vulnerabilities of any endpoints, database connections, and network transports.

NOTE

There are a variety of different threat modeling tools available to help surface and examine problems that might exist in your systems. If there isn't one in use within your environment, it might be a helpful area to understand vulnerabilities and areas for improvement. There is no one right way or tool, rather instigating the necessary discussions is beneficial.

- [NIST Common Vulnerability Scoring System Calculator](#)
- [Microsoft's Threat Modeling Tool](#)
- [Process for Attack Simulation and Threat Analysis \(PASTA\)](#)
- [OWASP Threat Modeling Control Cheat Sheet](#)

Ask yourself these questions:

- **Who are your attackers?** Attackers can be anyone. They may be internal or external from your organization. Based on the statistics coming from thousands of security incidents analyzed in the yearly [Verizon Data Breach Investigations Report \(DBIR\)](#), most attacks are external. There are the occasional internal rogue system administrators, but by and large internal security issues stem from system configuration errors or publishing private data publicly. In the next chapter, I'll cover some tools and technologies that will help reduce the number of errors that result in internal security incidents.
- **What are their motivations and objectives?** Attackers have different motivations and objectives for their activities. Financial gain motivates most attackers. Espionage and nation-state attacks are a growing threat with numerous¹ breaches occurring to gain intelligence and influence politics. Other motivations include for amusement, personal beliefs, ideology about a particular subject, or a grudge against your organization.

Examples of the objectives from obtaining different types of data include:

- With access to PII or personal data, attackers can apply for credit cards or sell information to marketing firms who specialize in spam campaigns.
- With access to payment card information, attackers can spend money fraudulently.
- With access to user credentials, attackers gain access to all the resources and services granted to the individual that can span multiple sites based on reuse of credentials.
- **What kind of resources do they have to attack?** The attacker's resources include time, money, infrastructure resources and skills. Tools are evolving that reduce the knowledge required for an individual attacker to obtain their target assets (and ultimate financial gain). While you can't necessarily prevent every single attack, you can make them more expensive.
- **What are their opportunities to attack?** Opportunities are the windows of access to a particular asset. When a vulnerability or flaw in software is discovered and released, there is a window of time to exploit that vulnerability on unpatched systems and services. Successful mitigation requires awareness of necessary patching and adequate time and authority to complete the work.

In some cases, there may be assets outside of your responsibility that attackers leverage to get into production systems. Minimize these opportunities by tracking all assets and patching operating systems and software promptly.

TIP

Check out Ian Coldwater's talk from KubeCon + CloudNativeCon 2019 [Hello From the Other Side: Dispatches From a Kubernetes Attacker](#) for more on what you can learn by borrowing the attacker lens.

Check out the yearly [Verizon Data Breach Investigations Report \(DBIR\)](#) which provides in-depth analysis of thousands of security incidents and breaches, and provides insight into evolving security trends.

Design for Security Operability

Layer your strategies to reduce risk to services and applications, thereby limiting the attacker's opportunity and the scope of damage of a potential breach. This approach is known as defense in depth. Layering defenses means that if one defense fails, the blast radius of compromise may be contained.

For example, build defenses at the edges of your networks with firewalls and configure subnets to limit network traffic from approved networks. Locally on systems, lock down elevated privilege accounts. Additionally, recognize that 100% secure software is impossible, and assume zero trust. Zero trust means having no implicit trust in any services, systems, or networks even if you are leveraging cloud-native services.

It's important to participate in the early architecture and design process with an operability mindset, especially around security, to provide early feedback to reduce the overall development time required. Case in point — I had joined a relatively new team that was building a multi-tenanted service for an internal audience. I reviewed the architecture and realized that the code relied on having no MySQL root password. With hundreds of backend MySQL servers planned for this service, large numbers of unsecured services worried me.

Some of the potential attack vectors I thought about included:

- A misconfigured subnet could make these servers directly accessible to the broader internet.
- Malicious attackers that breached systems on the internal network could easily compromise unsecured systems.

Working with the security engineering team, I managed to get the work prioritized to repair this design defect. Identifying the issue before deployment to production felt great. However, there was the avoidable development cost to fix if implemented collaboratively to start.

Often, decision makers don't invite sysadmins into design meetings. It's important to foster and build relationships with the individuals designing and building the software. This allows you to provide early feedback that will reduce friction for change that comes later in the process.

One way to collaboratively uncover security requirements and prioritize work is to use the CIA triad model. This model provides a way to establish a common context and align values for feature work. CIA stands for Confidentiality, Integrity, and Availability.

- **Confidentiality** is the set of rules that limits access to information to only the people who should have it.
- **Integrity** is that assurance that information is true and correct to its original purpose, and that it can only be modified by those who should be able to.
- **Availability** is the reliable access to information and resources for the individuals who need it, when they need it.

In the case of the root password for the MySQL issue I described above, anyone with access would have been able to log in to the database management system and look at and edit any available data stored. A database breach is a confidentiality compromise. The modification of data by a non-authorized agent is an integrity compromise. Sysadmins can flag CIA issues as part of the acceptance criteria. Having intentional conversations about the design and tracking those conversations helps

inform the decisions that the development and product teams make. This also adds a way of incorporating operability stories into work and prioritizing them appropriately. For web applications and web services, the **Open Web Application Security Project (OWASP)** provides a set of requirements and controls for designing, developing, and testing called the **Application Security Verification Standard(ASVS)**.

NOTE

If you are finding it challenging to get executive support for your efforts to design and implement quality continuous integration and deployment mechanisms, reducing the impact of security vulnerabilities is an excellent use case.

Qualifying Issues

No matter how much effort the team takes to examine software and services from the attackers perspective and designing systems to incorporate a security mindset, there will still be security issues. Some issues may be discovered with your company's software, other times the problem will be with software that you are using either directly or indirectly. Vulnerabilities in publicly released software packages are tracked with **Common Vulnerabilities and Exposures (CVE) Identifiers**. When quantifying the cost and potential impact, it's helpful to categorize them. One strategy is labeling an issue as a bug or a flaw.

Implementation bugs are problems in implementation that lead to a system operating in an unintended way. Implementation bugs can sometimes cause serious security vulnerabilities, for example, heartbleed². Heartbleed was a vulnerability in OpenSSL that allowed malicious folks to eavesdrop on presumed secure communications, steal data directly from services and users, and impersonate those users and services.

Design flaws are issues where the system is operating exactly as intended, and the problem is with the design or specification itself. Design flaws can be super costly to repair, especially if other tools are building on or

depending on the implementation as-is. Sometimes flaws are too expensive to change, and they carry specific warnings about use.

While you don't want to have metrics that incentivize behaviors that push for discovering flaws and bugs over other types of sysadmin work, it is crucial to surface the work that is in progress, especially when a compromise or security incident has been prevented.

TIP

Check out these examples of implementation bugs:

- MS17-010/EternalBlue
- CVE-2016-5195/Dirty CoW

Check out these examples of design flaws:

- **Meltdown**
- KRACK (WPA2 key reinstallation)

Wrapping Up

After reading this chapter, you should be able to:

- Define shared responsibility
- Describe defense in depth
- Explain the process of threat modeling
- Define the CIA triad
- Differentiate between design flaws and implementation bugs

¹ <https://blogs.microsoft.com/on-the-issues/2019/07/17/new-cyberthreats-require-new-ways-to-protect-democracy/>

2 Synopsys, Inc. “The Heartbleed Bug.” Heartbleed Bug, heartbleed.com/.

Part III. Principles in Practice

Infrastructure is vast and varied. It's a widely accepted practice to eliminate **snowflake servers** with infrastructure as code. Yet, every organization has its unique methods, which leads to challenges in community solved infrastructure management and needless arguments of the one way to do it.

I've seen a number of tools, techniques, and practices advocated for in my years in the industry to manage infrastructure. Some have weathered time, some have not.

The following chapters are areas of infrastructure management that have arisen based on the tools and collaboration between teams with a focus on automation.

At one job, I discovered that we had 11 different ways of managing configuration and deployment for one service along with multiple external services that were considered the source of truth touching the same metadata about the systems. I hope that by providing some waypoints, you too can navigate out of thorny infrastructure management scenarios.

Chapter 7. Infracode

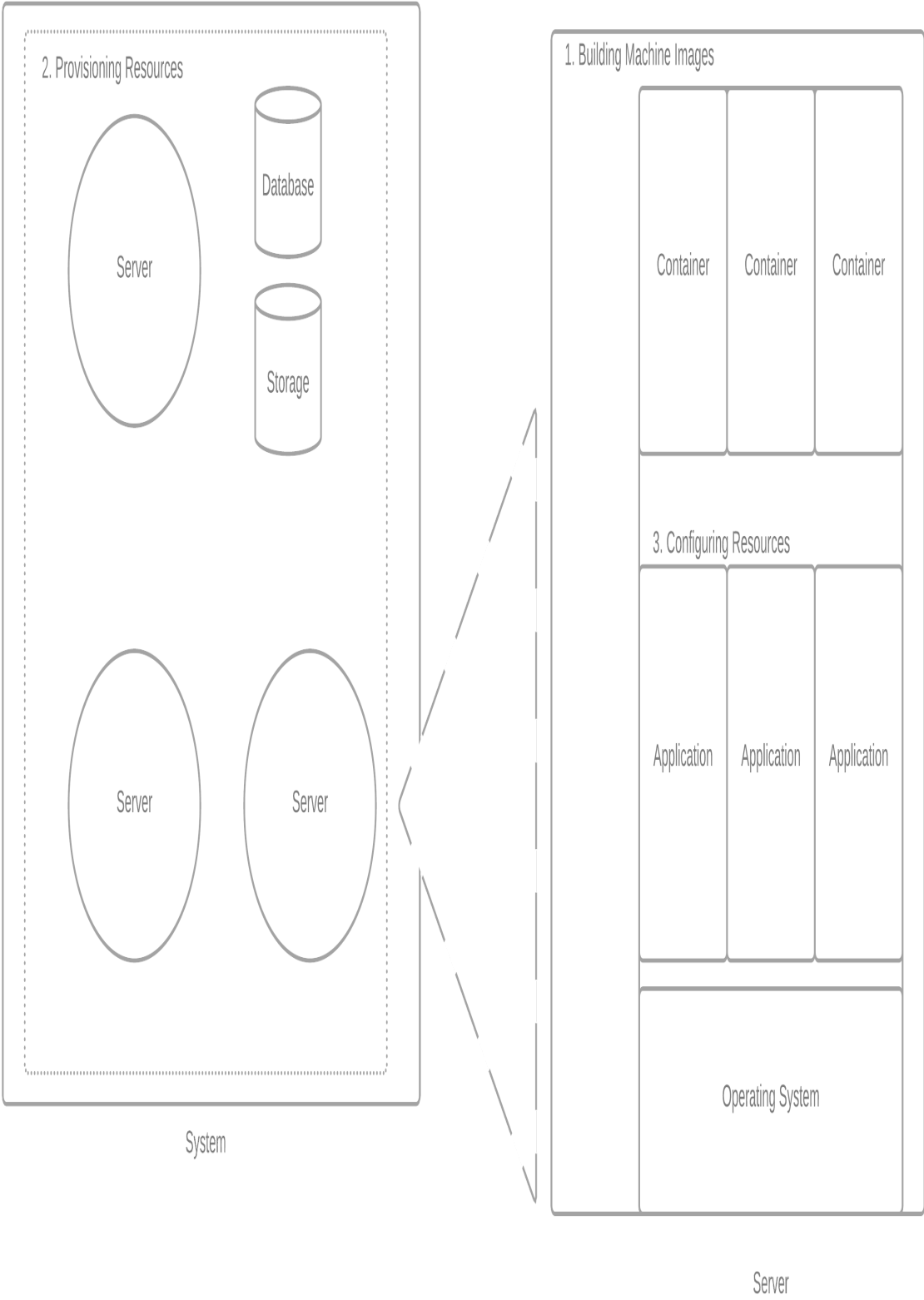
A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 10th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at vwilson@oreilly.com.

Infrastructure code (infracode) is human and machine-readable language to describe hardware, software, and network resources to automate consistent, repeatable, and transparent management of resources.



Think about the infrastructure that you are managing. You may have physical hardware, or separate compute instances with a variety of dependent services. Each compute entity will have an operating system, and may have several containers or virtual machine. Networking connects different entities, often with access control lists or policies that allow or restrict communication. Now, think about how you describe your infrastructure as code. You could look from the forest level of the broad set of instances, applications, and services, over their whole lifecycle from provisioning to removal from service. Or, you could start at a specific tree and think about a single compute instance and exactly how it's configured, including its operating system and any software configurations.

Before automating compute infrastructure provisioning, configuration, and deployment, you need to have a clear vision of the project's purpose and believe that it's worth the effort through the value it delivers. This vision needs to be in alignment with any stakeholders that may have other priorities.

You might adopt infracode to implement:

- **Consistency:** You deploy and configure systems in a uniform way which has been tested and documented.

Consistency can increase productivity and efficiency of the team.

- **Scalability:** Infracode streamlines the provisioning and deprovisioning process, allowing you to activate and deactivate fleets of systems as required, with minimal effort. This can take the form of easy manual scale-up and scale down, fully automated cloud-native management, or any combination, allowing the system to dynamically respond to peaks and troughs in demand while also enabling humans the authority to govern the operation of the automation system.

Scalability can increase revenue, add product differentiation, reduce always-on infrastructure costs, and increase user satisfaction.

- **Empowerment:** You define layers of responsibility to allow different teams to have autonomy over their resource governance. You define how to share responsibility between infrastructure, security and application teams, enabling self-service within negotiated boundaries and maintaining overall visibility.

Empowerment can decrease the friction of deploying new products while keeping spend within acceptable boundaries leading to increased revenue and differentiation in product development.

- **Accountability:** Tracking infracode changes with version control, you have a history of changes to systems and an audit trail so anyone can answer questions about systems created.

Accountability can decrease costs as you can deprovision systems that no longer should be in use.

- **Enculturation:** Version control changelogs facilitate onboarding new team members to show how you do things.

Enculturation can increase productivity and efficiency.

- **Experimentation:** Infracode can allow people to spin up test environments easily, try out new technologies, and quickly push them to production when such experiments are successful.

Experimentation can increase revenue and help the team focus on market differentiation.

In this chapter, I outline three infrastructure code approaches with example code to show you how to put each into practice:

1. Building machine images
2. Provisioning infrastructure resources
3. Configuring infrastructure resources

My goal in selecting the examples for the three approaches is to model tool evaluation to help you identify core components and see sample

implementations. After the examples, I guide you on starting to understand your team's infrastructure automation needs and how to assess different infracode tools.

Be mindful that technological change is much like a biological ecosystem, with various habitats, niches, and species. In the case of technology, some new tool comes along and fulfills a need, and the community adopts the practices, if not the technology. This morphs patterns of collaboration and communication, and other technology platforms change to mirror the community's new needs. I hope to show you general patterns here, but books reflect a point in time, and you may find newer tools and technology. Look at the documentation for the specific version of your chosen tool for up to date recommended practices..

It can be challenging to find consistent terminology for infracode tools and services. In some cases, there is a difference in the name the community uses to talk about particular software and how the company markets that same software. For example, sometimes enterprise versus open source versions have different functionality, or a product is rebranded or acquired by another company. The community may refer to the software by an earlier or open source name, rather than the company's name. Similarly, cloud providers often offer fundamentally similar services with different names and subtle differences in capabilities. Trying to map one-to-one functionality between providers, especially with infracode, can be very frustrating because syntax and abstractions varies widely.

It is common to categorize infracode tools as imperative or declarative. With imperative infracode, you specify the procedure for achieving a task. With declarative infracode, you describe the desired end state and the tool handles the implementation. In practice, tools that confine sysadmins to either of these extremes end up being difficult to work with. For example, a declarative framework might work for most common deployments while being too limited to express what has to happen for particular scenarios. An imperative framework might provide better expressiveness for such edge cases, but too cumbersome for the common boilerplate scenarios where you just want to deploy a standard image with only a couple of minor tweaks

through custom variables. The infracode tools that find widespread adoption tend to balance the declarative-imperative axis, providing straightforward and flexible ways to implement many deployment pipelines. Instead of focusing on whether a tool is imperative or declarative, I focus on the 3 approaches: building machine images, provisioning infrastructure and configuring infrastructure.

Building Machine Images

A key task for system administrators has been deploying computers, but what this entails has evolved. There are a spectrum of environments where your code may run, ranging from a physical machine with a dedicated operating system and installed applications, to a virtualized environment that replicates a physical system, to a containerized application that runs atop a host system. Because these technologies reuse many of the same concepts, tool developers tend to reuse terminology, but this can create confusion when you need to be specific about the level of abstraction you're using. For purposes of this discussion, I'm going to refer to "machines" and "machine images", with the understanding that in practice this has a spectrum of meanings, from physical systems to application containers.

Early in my career, I sped up delivery of new physical systems by cloning from a hardware disk and then updating the configuration of the operating system and applications. It was a very manual process, but faster than building the physical machine, installing the operating system via CDs, and then updating over the internet. This pattern was known as building from a "golden image": a perfect, known good mold from which you could create more systems. Workflows today conceptually descend from this approach, where a machine image — which could be a virtual machine disk image, a container image, or something else — can serve much the same purpose as golden images. You use machine images can to automate system builds, including hardening the operating system to reduce vulnerabilities and pre-

installing any necessary and common tooling. Thus, you provision your compute resources from a more secure and robust base.

Examples of tools that build machine images include:

- Packer for multi-platform machine images
- EC2 Image Builder for Amazon Machine Images
- Docker for docker images

Building with Packer

Let's explore creating a machine image using **Packer** as it is cross-platform and open-source. Packer creates machine images from a single configuration file. This can help you build similar images in a repeatable and consistent fashion for different platforms. The configuration file is called a template. Packer templates are written in JSON with three main parts: variables, builders, and provisioners.

- **Variables** parameterize templates to separate secrets and environment-specific data from the infrastructure code. This helps reduce errors where a specific value may be duplicated multiple times and provides a way to override a configuration at build time. By parameterizing secrets in your templates, you can prevent them from getting checked into your source code.
- **Builders** interface with providers to create machines and generate images. You can select an existing builder, use one from the community, or write your own. Commonly-used builders include Amazon EC2, Azure, Docker, and VirtualBox.
- **Provisioners** customize installation and configuration code after the image is booted.

TIP

Lint your Packer configuration files to validate syntax and clean up formatting issues. `jsonlint` is one common CLI JSON parser and linter.

Here's an example of running `jsonlint` at the command-line on a Packer template that has a missing `,`. The validator identifies that the document doesn't parse as valid JSON.

```
$ jsonlint packer_sample_template.json
Error: Parse error on line 7:
...": "amazon-ebs"      "access_key": "{{use
-----^
Expecting 'EOF', '}', ':', ',', ']', got 'STRING'
```

Look at the Testing chapter for more information about linting and testing recommendations for `infracode`.

With these introductory Packer concepts, you can create a template that would build a base AWS EC2 machine image (AMI). In combination with a continuous integration and build platform, you can create a process to build and update AMIs to use the most up-to-date operating systems, packages and softwares in a repeatable manner. While this template is specific to building AWS machine images, a similar template would build images for other platforms.

In this first block, you pass in AWS specific user variables to keep secret keys out of the template. There are multiple ways to define these and keep secrets completely out of source control. The access and secret key are empty, with the expectation that the values will be defined when Packer runs.

```
{
  "variables": {
    "aws_access_key": "",
    "aws_secret_key": "",
    "aws_account_id": ""
  },
}
```

Next, the builders block interfaces with the specified providers. In this example, it's the **amazon-ebs builder** for interfacing with AWS and building AMIs based off of an existing source AMI.

```
"builders": [{  
  "type": "amazon-ebs",
```

This passes in the user variables that were defined in the earlier *variables* section block.

```
  "access_key": "{{user `aws_access_key`}}",  
  "secret_key": "{{user `aws_secret_key`}}",
```

In the next few lines (below), *region* specifies the region to launch the EC2 instance used to create the AMI.

With this builder type, Packer starts from a base image which is specified using *source_ami* or via a filter using *source_ami_filter*. Using the filter enables you to find the most recently published AMI from a vendor rather than hard coding to a specific image, or to pull from a specific source by defining the *owner* parameter. In this example, you're pulling the latest Ubuntu 20.04 AMI. Your filter must only return one image as a source or this fails.

```
  "region": "us-east-1",  
  "source_ami_filter": {  
    "filters": {  
      "virtualization-type": "hvm",  
      "name": "ubuntu/images/*ubuntu-focal-20.04-amd64-server-*",  
      "root-device-type": "ebs"  
    },  
    "owners": ["{{user `aws_account_id`}}"],  
    "most_recent": true  
  },
```

The **instance_type** specifies the instance type for the image. The **ssh_username** is specific to the AMI default username and will vary. In this example, it's an Ubuntu image that uses ubuntu as the username.

The **ami_name** specifies the AMI name that will appear when you manage AMIs in the AWS console or via APIs. The AMI name must be unique. The function **timestamp** returns the current Unix timestamp in Coordinated Universal Time (UTC) which can help create a unique AMI name. The function **clean_ami_name** removes invalid characters from the AMI name.

```
"instance_type": "t2.micro",
"ssh_username": "ubuntu",
"ami_name": "example-{{timestamp | clean_ami_name}}"
}},
```

The final block in this example is the provisioner block that customizes the image. You specify the type of provisioner with the type parameter. In this case, the provisioner type is the *shell* provisioner which allows you to execute scripts and commands on the image being built. This example uses the Ubuntu specific command-line tool **apt-get** to update the software list and then apply updates and patches.

```
"provisioners": [{
  "type": "shell",
  "inline": [
    "sudo apt-get update",
    "sudo apt-get upgrade -y"
  ]
}]
}
```

There are more customizations for the amazon-ecs builder including security groups, subnet IDs, tags, and IAM instance profiles that can be configured and baked into an AMI image which makes this an extremely powerful tool to create a base machine image that can create everything required to set up repeatable and consistent instances. Additionally, **other builders** are available.

To summarize what happens from the example above when Packer is run:

1. Builders connect with the provider and launch an instance of a base image. Base images are generally vendor supplied images set up

for a first-run as if it's just been installed. In the above example, the builder would connect with AWS and create an instance of the type specified based on the source AMI. It creates a new AMI from the instance and saves it to Amazon.

2. Provisioners customize the instance. You use the shell provisioner in this example, but there are a wide range of different **provisioners** available including InSpec, Puppet Server, and Windows Shell.

In more complex situations, you could configure post-processors to perform any additional tasks after the image has been created, for example to compress and upload artifacts from previous steps.

TIP

Learn more about Packer:

- [HashiCorp Packer website that includes links to more through guides and documentation](#)
- [The Packer Book by James Turnbull](#).

Building With Docker

Let's explore building with **Docker**. You describe an image from a configuration file called a **Dockerfile**. Dockerfiles are plain text files that contain an ordered list of commands. Docker **images** are read-only templates with the instructions for creating a container. You can build images or pull them directly from a registry, an artifact repository that stores images.

Just as with any artifact your environment depends on, it may be beneficial to host verified and tested images rather than depending on an external service for artifacts. This eliminates one possible type of failure and a host of security concerns, at the cost of managing these artifacts within the organization (which brings its own set of failure scenarios and security concerns).

- Docker images have **tags**. Tags are names that refer to a specific snapshot of an image. If a tag is not referenced with the image within the Dockerfile, the tag *latest* is used by default.

Always use a specific tag associated with an image. Latest can move, so if you test and deploy against latest you might not have the same image between test and production.

- Docker **base image** have no parent images. These are generally OS specific images like debian, alpine, or centos.
- Docker **child images** build on top of base images to add functionality.
- A Docker **container** is a runnable instance of an image. You can start, stop, move, or delete a container.

Docker is not a virtual machine manager. If you treat it like VirtualBox or VMWare Workstation, it might feel similar starting up an instance with *docker run* but activities like your system sleeping or rebooting might have unintended consequences especially if you are doing local development within docker. What happens when your system sleeps or reboots? Containers fundamentally behave differently than virtualization software.

Let's use this example of a minimal Dockerfile to review some of the syntax of this configuration file.

Let's break the Dockerfile down line by line.

```
FROM alpine:latest
```

This instruction specifies to use the alpine minimal base image from the default registry, Docker Hub at the latest tag.

```
RUN apk --no-cache add apache2-ssl apache2-utils ca-certificates httpd
```

This instruction uses the Alpine package manager *apk* to install packages `apache2-ssl`, `apache2-utils`, `ca-certificates` and `htop`.

```
ENTRYPOINT [ "ab" ]
```

With the `ENTRYPOINT` instruction, you set the image's main command allowing **ab**, the Apache Bench command to be run as though the container was the command.

After you build the image from the Dockerfile, you can spin up a container. In this instance, after building, you can run Apache Bench directly on your host without installing directly on to your system.

Dockerfiles can get a lot more complex with a number of other commands. There are a number of **recommended practices** for writing Dockerfiles.

You may want to start your infrastructure as code journey with building machine images in `infracode` if you need to:

- Ensure systems have a common updated base image
- Install a set of common tools or utilities on all systems
- Use images that are built internally with known provenance of every software package on the system

Provisioning Infrastructure Resources

Provisioning cloud resources through `infracode` allows you to

1. Specify the virtual machines, containers, networks, and other API enabled infrastructure needed based on your architecture decisions,
2. Connect the individual infrastructure components into stacks,
3. Install and configure components, and
4. Deploy as a unit.

Examples of tools that provision infrastructure resources include:

- HashiCorp Terraform,
- Pulumi,
- AWS CloudFormation,
- Azure Resource Manager, and
- Google Cloud Deployment Manager.

Provisioning frameworks that can deploy to multiple platforms, such as Pulumi and Terraform, are particularly useful.

Provisioning with Terraform

Let's look at an example of provisioning cloud infrastructure using Terraform. Terraform is open-source software from HashiCorp that can provision, manage, and version infrastructure, including physical and virtual machines, networks, containers, and other infrastructure services.

Terraform configuration is written in **HashiCorp Configuration language (HCL)**. Building blocks for Terraform are providers, resources, and the configuration file that collects the set of desired policy.

- A *Terraform provider* is the interface to a specific set of APIs that exposes a collection of resources. There are a variety of providers and if necessary you can also write your own.
- A *Terraform resource* is a component of infrastructure available in a provider. Resources will vary by provider, even for similar services.

Infracode to manage a particular technology differs across providers due to service abstraction implementation and the interfaces available to use them. Let's look at an example of the difference between AWS and Microsoft

Azure's DNS resources. These code fragments define a single *www.example.com* DNS A record with AWS Route 53 and Azure DNS:

Notice that the Azure provider has a resource per record type, where the AWS provider has a single resource with a *type* parameter to specify the different records. After the creation of resources, Terraform stores the current state of the configuration in a file or some external location, according to your preference. Sometimes there is critical data that is external to systems that you manage with Terraform but that informs your configuration, for example, identifying and using a specific AWS AMI image within your infracode. Terraform *data* sources allow this data to be queried and used within Terraform.

TIP

Learn more about Terraform:

- [Hashicorp Terraform Documentation](#)
- [The Terraform Book by James Turnbull](#)
- [Terraform: Up & Running by Yevgeniy Brikman](#)
- [Terraform azurerm Provider on GitHub](#)

Often writing valid infracode requires a lot of knowledge to successfully provision and configure infrastructure resources. This is a longer example of writing infracode to provision infrastructure resources so that you can see this in action. In this example, I walk through using Terraform to define DNS as code on Azure that sets up Fastmail as the mail server for the domain and hosting on GitHub pages. I chose this example as it's low cost and meaningful to many who need to manage a domain and configure DNS for a web and email provider.

| Service | Approximate Cost (\$) |
|---------------------------------------|-----------------------|
| Azure DNS | <\$1/month |
| Fastmail Standard for a custom domain | \$5/month |
| GitHub Pages | Free |

WARNING

If a zone already exists within your cloud provider, be super careful making changes. Terraform will destroy pre-existing resources for some kinds of modifications.

Specify the **Azure Provider** to configure infrastructure in Microsoft Azure using the Azure Resource Manager API.

```
provider "azurerm" {
}
```

Specify the Resource Group. If the Resource Group doesn't exist as defined, it will get created. This creates a bucket for the rest of the DNS configuration resources.

```
resource "azurerm_resource_group" "example" {
  name = "ExampleResourceGroup" # Replace with your Resource Group Name
  location = "westus2"
}
```

Specify the DNS zone replacing *example.com* with your domain. *Example* within the `resource_group_name` should match the parameter in the resource group. Azure hosts zones on Azure's name servers.

```
resource "azurerm_dns_zone" "example" {
  name = "example.com" # Replace with your domain
  resource_group_name = "${azurerm_resource_group.example.name}"
}
```

Specify the necessary Canonical Name (CNAME) for the **GitHub hosted page**. Replace username with your username or organization. The record block within the azurerm provider for the *azurerm_dns_cname_record* is required. It's the target of the CNAME record.

```
resource "azurerm_dns_cname_record" "www" {
  name = "www"
  zone_name = "${azurerm_dns_zone.example.name}"
  resource_group_name = "${azurerm_resource_group.rg.name}"
  ttl = 300
  record = "username.github.io"
}
```

Specify the appropriate configuration so that Fastmail can sign email to verify validity of the email. This helps prevent the classification of email as spam when sent from Fastmail on your behalf.

```
resource "azurerm_dns_cname_record" "mesmtpl" {
  name = "mesmtpl._domainkey"
  zone_name = "${azurerm_dns_zone.example.name}"
  resource_group_name = "${azurerm_resource_group.rg.name}"
  ttl = 3600
  record = "mesmtpl.example.com.dkim.fmhosted.com"
}
```

```
resource "azurerm_dns_cname_record" "fm1" {
  name = "fm1._domainkey"
  zone_name = "${azurerm_dns_zone.example.name}"
  resource_group_name = "${azurerm_resource_group.rg.name}"
  ttl = 3600
  record = "fm1.example.com.dkim.fmhosted.com"
}
```

```
resource "azurerm_dns_cname_record" "fm2" {
  name = "fm2._domainkey"
  zone_name = "${azurerm_dns_zone.example.name}"
  resource_group_name = "${azurerm_resource_group.rg.name}"
  ttl = 3600
}
```

```

    record = "fm2.example.com.dkim.fmhosted.com"
  }

  resource "azurerm_dns_cname_record" "fm3" {
    name = "fm3._domainkey"
    zone_name = "${azurerm_dns_zone.example.name}"
    resource_group_name = "${azurerm_resource_group.rg.name}"
    ttl = 3600
    record = "fm3.example.com.dkim.fmhosted.com"
  }

```

If you don't specify a name within the MX record request, Azure configures the root of the DNS zone. This example configures email to be received at the domain *@example.com*. The record block within the azurerm provider for the *"azurerm_dns_mx_record"* has 2 required parameters: preference and exchange. Lower preference values take priority over higher preference values. The exchange parameter specifies a mail server that is responsible for the domain.

```

resource "azurerm_dns_mx_record" "example" {
  zone_name = "${azurerm_dns_zone.example.name}"
  resource_group_name = "${azurerm_resource_group.rg.name}"
  ttl = 3600

  record {
    preference = 10
    exchange = "in1-smtp.messagingengine.com"
  }

  record {
    preference = 20
    exchange = "in2-smtp.messagingengine.com"
  }
}

```

This example configures the * MX record to receive an email at all subdomain addresses.

```

resource "azurerm_dns_mx_record" "starexample" {
  name = "*"
  zone_name = "${azurerm_dns_zone.example.name}"
}

```

```

resource_group_name = "${azurerm_resource_group.rg.name}"
ttl = 3600

record {
  preference = 10
  exchange = "in1-smtp.messagingengine.com"
}

record {
  preference = 20
  exchange = "in2-smtp.messagingengine.com"
}

}

```

This example configures a TXT Sender Policy Framework (SPF) record to identify that the Fastmail mail servers are allowed to send email on behalf of your domain. The record block within the azurerm provider for the *“azurerm_dns_txt_record”* has one required parameter, value. Multiple records can be defined.

```

resource "azurerm_dns_txt_record" "spf" {
  name = "@"
  zone_name = "${azurerm_dns_zone.example.name}"
  resource_group_name = "${azurerm_resource_group.rg.name}"
  ttl = 3600

  record {
    value = "v=spf1 include:spf.messagingengine.com ?all"
  }
}

```

While this plan creates only a few records, it requires knowledge about:

- Azure Resource Manager and Resource Groups,
- DNS record types,
- Terraform Azure provider,
- SMTP requirements for mail delivery (DKIM and SPF records), and

- specific Fastmail configurations.

Often infracode obfuscates the underlying “how does this work.” Humans work with these systems and must understand more than just “terraform apply.” When problems occur, you need to know where to debug.

For example, without the SPF and DKIM records, mail delivery to most providers could be disrupted, and mail from your domain might not be delivered. Checking for valid Terraform syntax won’t prevent operability mishaps in the code. Redeploying the infrastructure code won’t catch the missing configurations.

These tools also illustrate that not all infrastructure platforms have APIs to leverage with code and automation. Fastmail doesn’t have an available API to automate this kind of configuration. You may find that you can’t define all of your resources as infracode.

You may want to start your infrastructure as code journey with provisioning infracode if you have or need:

- Systems that are already partially using provisioning
- Multi-cloud support
- Multi-tier applications
- Repeatable environments for example a testing environment that is a smaller clone of the production environment

Configuring Infrastructure Resources

Configuring infrastructure resources through infracode allows you to handle software and service configuration once hardware infrastructure is available.

Examples of tools that configure infrastructure resources include:

- **CFEngine**,

- Puppet,
- Chef Infra,
- Salt, and
- Red Hat Ansible.

Configuring with Chef

Let's look at an example of configuring infrastructure resources using Chef Infra. Chef Infra is open-source software for defining configuration policy.

NOTE

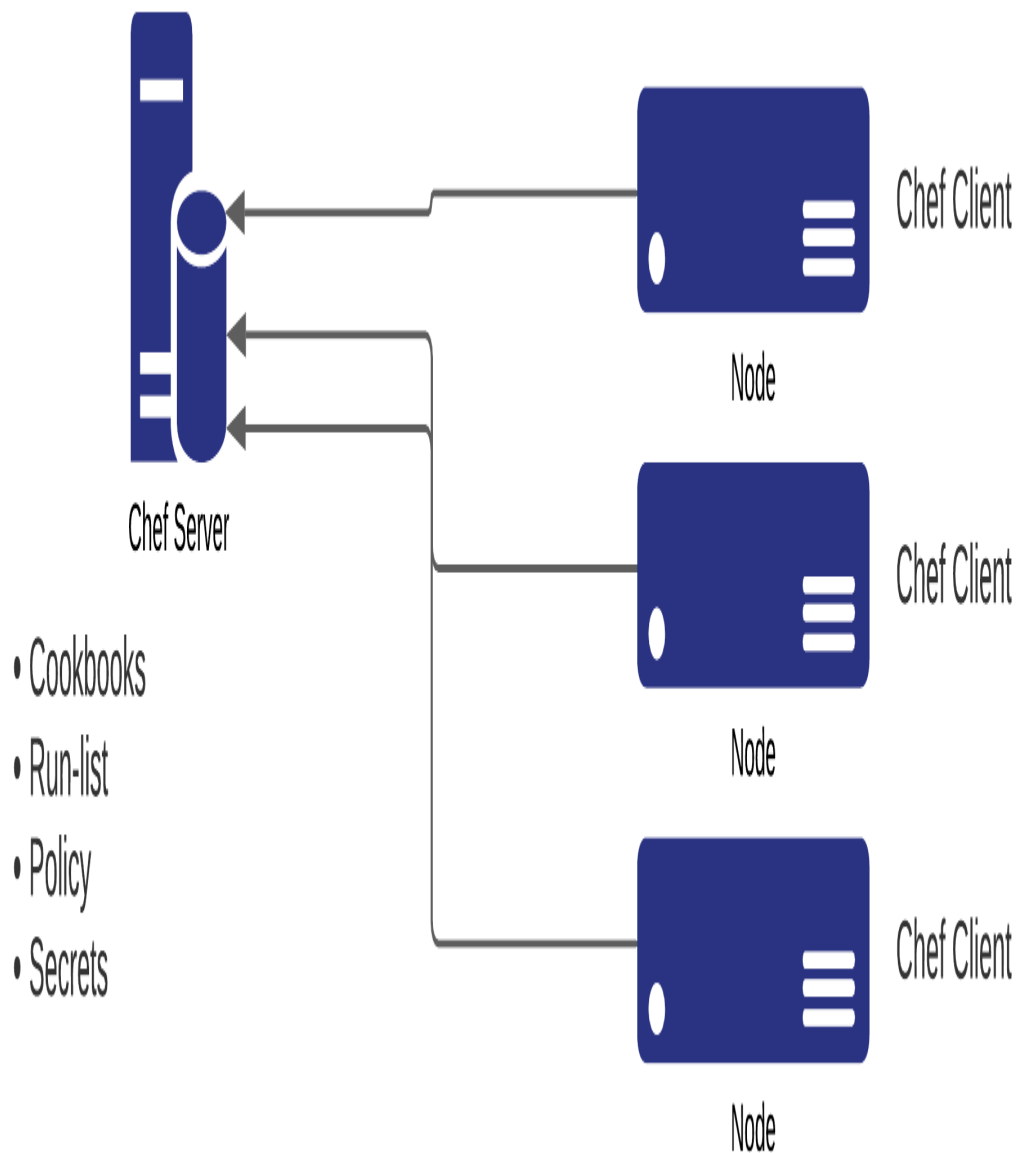
As of April 2019, use of current versions of Chef in a commercial capacity requires payment. **Cinc** is a community edition built by folks in the Chef community to provide a free distribution of Chef.

Chef Infra is written using a Ruby DSL specific to Chef. Because it's a Ruby DSL, anything that you can do with Ruby, you can also do with Chef. Building blocks for Chef Infra are cookbooks, recipes, and resources.

- *Cookbooks* are a collection of policy code that includes everything about a specific function you need. In general, you want to have small cookbooks that focus on a single purpose so that it's easy to figure out where to find the code when it needs to be modified or removed.
- *Recipes* are text files that define the algorithm to do a specific task or set of tasks. Chef Infra processes the resources in the order that they appear in the recipe.
- *Resources* map out to a specific piece of infrastructure you manage in a recipe.
- *Custom resources* are components that can be written and included in a cookbook and used like other Chef provided resources.

- *Nodes* are any device-physical, virtual, network, or cloud resources
 - that is managed by Chef Infra.

Sysadmins make changes to infracode, check it into git, and push policy information to the Chef Infra Server.



When hosts are managed by Chef Infra, the hosts check-in with the Chef Server to get their defined state and correct configuration drift if it has occurred. This means that if someone makes a change directly to the host for

components that Chef manages, the host will revert back to the version as defined by the code.

Let's look at an example of creating a chef recipe to install the Datadog agent to collect event information from a node. Datadog is a monitoring service used to monitor servers, databases, tools, and services. There are a variety of available plugins to collect and visualize information on the Datadog service.

In this recipe, it sets up the Datadog repository for an Ubuntu system and installs the `datadog-agent` package using 2 resources `apt_repository` and `apt_package`. You could also set up the configuration with your API key and start up the datadog agent as a service.

When writing Chef code, the `infracode` describing configuration files on the system are written to the system exactly the way you specify. If there is an error in a template, Chef won't catch the error with the content (unless it's failing Ruby or Chef code).

With the datadog cookbook, if the install recipe got complex enough, you could turn the recipe into a custom resource. This would be reusable by anyone who included a dependency to the cookbook.

One frequently used tool to test out Chef code is Test Kitchen. Test Kitchen is an application that integrates with different cloud providers and virtualization technologies. It can leverage whatever `infracode` definitions in use in an environment in combination with a configuration file that defines the set of test suites to run with those applications.

With a valid project name and region in this kitchen configuration file bundled with the project, any team members can quickly set up the infrastructure and project environment to start iterating on developing Chef `infracode` using the Test Kitchen tool.

TIP

Learn more about Chef Infra:

- [Chef Learn site](#)
- [Chef Software on Azure documentation](#)
- [Zero-to-Deploy with Chef on GCP](#)
- [Chef Automate on AWS](#)
- [Test Kitchen](#)

Getting Started with Infracode

Now that you have seen three different infracode perspectives - building machine images, provisioning infrastructure resources, and configuring infrastructure resources - let's look at infrastructure code in practice within your organization.

The use of infrastructure as code is often seen as critical to contemporary system administration practices. In an organization with little to no current IaC practices, this is a large technical change that also requires process change and potential skill updates to be successfully adopted. If you're not creating and managing infrastructure with code now, it can feel overwhelming to identify a place to start. In an organization with current IaC practices, it may be difficult to try to understand how to use the systems let alone make improvements.

1. Ask what improvements and value will be obtained through the desired infracode improvements. Does everyone on the team and the various stakeholders have a shared vision and believe that the project's purpose is worth the effort?

While a maturity model can be inspiring through its prescription of IaC, this can derail a successful transformational project. Dig deeper as the what and why informs implementation requirements.

Maybe you have inconsistencies in your compute infrastructure that cause problems when code is deployed to production. Maybe deploying dependency applications takes too much time. Think about your current challenges. Sometimes the challenges can readily lend themselves to infracode solutions, such as “golden images” which take too long to build or are inconsistent over time. In other cases the link may be less clear.

I’ve seen infracode projects used to improve time-to-deploy for development environments in a large software company. Infracode can help a geographically distributed infrastructure team collaborate asynchronously by turning real-time system configuration tasks into scheduled code changes at optimal times. Infracode can streamline onboarding, making it easier to accept part-time assistance for specific projects, while also facilitating cooperation among different teams. Whatever the situation, identification of the challenge is key; infracode is not an end unto itself.

2. Narrow the scope of what you can achieve based in a certain amount of time.
3. Select the best tool that can be used to solve the problem. I’ve shared three common approaches which can help you narrow your research to possible tools for your problems. Depending on the technologies you want to manage via code, there may or may not be existing tools with the necessary features; this may lead you to reconsider your underlying technology choices too! Identifying which tools have the necessary features is only the beginning of the process of choosing an infracode tool.

If infracode isn’t in use now as with all decisions about technologies, languages, and frameworks, you must consider how the tool will fit into the social situation of your team. It could mean that you choose a tool because it uses a programming language that several people already know. An existing positive

relationship with a vendor or reseller may sway you toward a particular tool.

Be sure to include the overall cost of using and supporting tools into your decision. Some teams do better collaborating within a worldwide community of fellow-users as made possible with open source technologies, while other teams benefit more from access to commercial training and support. Neither of these options is inherently better, but choosing one that goes against your team culture will add to the complexity of successful adoption.

Selecting and implementing an infrastructure as code platform has long-lasting impacts for the team, if not the entire organization. It's difficult to retire technology that's still in use - difficult, but possible. The field is evolving quickly, and some tools may lock you into using a specific vendor's toolset which may not be an acceptable tradeoff.

4. Put the chosen tool into practice. This process varies depending on if the tool is being used from the first day of adoption of a new technology (green-field deployment) or is being brought in to solve struggles in an existing environment (brown-field deployment).

In a green-field deployment scenario, try to use the selected tooling for all the workflows where it is relevant. This will encourage the adoption of infracode habits from the beginning, avoiding annoying and time-consuming re-learning later on. It will also highlight cultural incompatibility or workflow issues right away, allowing you to solve them or re-scope the project before things get out of control.

In a brown-field deployment scenario, try to adopt the tool into the existing workflows gradually and in a prioritized manner. Your chosen tool can probably be used for nearly everything you do—but you can't do it all at once. Try to focus on one area for improvement at a time, such as getting all SSH configuration under the control of a tool like Puppet or Chef before moving on to

managing mail or webserver configuration. Avoid spending long periods of time in the in-between space where a particular aspect of your infrastructure is configured sometimes via infracode and sometimes manually. It's confusing, forces everyone to know both methods, and can lead to your project stalling out. Find success in one small situation and apply the positive and negative lessons to the next situation. You'll do better and better as time goes on.

Be wary of taking on too complex of a project at the beginning, or trying to force one tool to fix everything for you. For example, if you have numerous UNIX platforms but mostly Linux, you will probably be able to write simpler infracode and build a history of wins by focusing on solving Linux challenges. Then, add in the other platforms when their workflows feel similar to the ones already being managed on Linux. It may be better overall to get all your platforms under the control of one tool, or to use specialized tools for each. Experience will guide you to the best choice for your needs.

- Don't sacrifice collaboration.
 - Implement single points of authority over elements of infrastructure. If multiple tools are updating the same resources, conflicts in updates will cause pain, frustration and needless paging.
5. Long-term success of an infracode project requires considering the workflows the tool will encourage, and how those workflows will change the dynamics of your team. Once you've fully rolled-out an infracode tool, that tool becomes the way relevant system changes will be made in the future. This means that everyone on the team needs to understand the tool well enough to use it in their day-to-day work. If the infracode project feels like it belongs only to a subset of the team, those people will become a bottleneck when the rest of the team comes to them asking to make changes they used to be able to do by themselves. This leads to frustration on both

sides. Be sure to build adoption by asking the whole team for their input and showing targeted demos that make real day-to-day struggles easier.

6. Measure the value

7. Identify the necessary skills required to be successful

By following these guidelines in an iterative loop, you will be able to create an infracode journey that is customized to your organization or team's needs, technology, strengths, and weaknesses. Always remember that the purpose of infracode is to enable you to manage your infrastructure more easily and collaborate more effectively.

If you are evaluating an open source solution based on cost, you should also evaluate the hidden costs of supporting the software. It can be hard to evaluate the costs without participating actively in the community.

Often your infracode solution is a multi-prong one that accommodates the complexity of your infrastructure. This is *ok*. It's perfectly reasonable to adopt Packer to build machine images, Terraform for immutable ephemeral containers in the cloud, and Terraform with Chef for longer lived instances. The important thing is to come up with a cohesive approach that weaves such tools together in a viable way.

Trying to force one solution to solve everything causes a lot of headaches, and even drives some to create yet another inframanagement language.

Before you invest time in a workaround, remember:

Start small. Pick a particular piece of infrastructure to assess and implement infrastructure as code practices. Check to see how the flow of implementation works with any checks and balances your environment has.

Even the most experienced among us needs some level of training to bootstrap the successful adoption of technology.

If folks haven't used version control before, that's the first skill that everyone needs along with accounts on whatever version control system is in use.

For example, to prepare engineers for Chef Infra collaboration, this might be a combination of:

- Ruby,
- Chef Infra, and
- Chef Infra implementation within your organization.

To prepare engineers for Terraform collaboration, they may need a combination of training in:

- HashiCorp Configuration Language (HCL), and
- Terraform, and
- Terraform implementation within your organization.

TIP

Learn more about Infrastructure as Code as a practice from the updated [Infrastructure as Code](#) book by Kief Morris.

Wrapping Up

The purpose of infracode is to enable you to manage your infrastructure collaboratively as a team in a consistent, reliable and repeatable way. Current widely used infracode tools generally focus on one of three main use-cases: building machine images, provisioning infrastructure resources, and configuring existing infrastructure. To help you identify where to get started, I've shared broad guidelines to make timely choices. By following these guidelines in an iterative loop, you will be able to create an infracode journey that is customized to your organization or team's needs, technology, strengths, and weaknesses.

Chapter 8. Testing in Practice

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 11th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at vwilson@oreilly.com.

We write tests to build confidence in our tools and infrastructure code. This is helpful when collaborating on code as this can help eliminate some of the fear of making a change to the systems. The goals of testing is to help you assess risk, respond to and recover from problems quickly, and improve your delivery processes.

In this chapter, I revisit linting, unit and integration testing in practice. You’ll get real examples, and learn a bit more about infrastructure and writing tests for infrastructure.

Writing Unit Tests for Infracode

A challenge of writing unit tests for infracode is that it can be very easy to test the infrastructure platform in use rather than your code. Think about the test and whether it’s verifying the code as written, or testing that the infrastructure as code software is working. Unless it’s an in-house developed system, trust that the software does what it is supposed to do.

Even if you are working with an in-house developed configuration system, test that platform in its git project separately from your infrastructure code project.

With infracode unit tests, there generally is a specific package that maps out to testing the platform you are using. For example, Chef has Chfspec, and puppet has rspec-puppet.

Let's dig into infracode unit testing with Chef. Remember, the key is to focus on broad principles and concepts. While you may not use Chef in your environment, you can apply the testing concepts and practices regardless of tools.

NOTE

You may want to review the Infrastructure and Infrastructure in Practice chapters before proceeding in this chapter. Find more detailed information about Chef in those chapters.

Writing Unit Tests with Chfspec

Chef recipes can get complicated when you have specific customizations, for example, different operating systems, compute instances, or the environment that the system exists in test or production.

Valuable unit tests are going to test those inputs that change how the recipe runs so that you can have deterministic outputs.

Chfspec is the testing framework for unit tests with Chef. It will test the recipes and custom resources in the context of a simulated Chef run. It is an extension of **RSpec**, and once installed, is run using the *rspec* command.

NOTE

Chfspec is packaged as part of the **Chef Workstation**. Chef Workstation is a set of tools and utilities to facilitate developing infracode within the Chef ecosystem.

Defining RSpec Fundamentals

To use Chefspec effectively, it helps to understand RSpec fundamentals. RSpec is a testing tool for Ruby programmers. It facilitates writing acceptance and unit tests that are closer to English. When you write Chef code, you are writing Ruby, so tools that help Ruby programmers will help you with your Chef code as well.

The basic structure of RSpec uses *describe* and *it*.

Use *describe* to signal a collection of tests. In RSpec, this collection of tests is called an *example group*.

Use *it* to define one test. In RSpec, this test is called an *example*.

If you wanted to bake some cookies and wanted to test your cookie making process, you might think of this process as “Baking cookies requires me to gather all the essential cookie ingredients, preheat the oven to the specified temperature, measure the ingredients, combine the ingredients following a recipe, and bake the cookies for a specific amount of time.”

If you translated that into RSpec terminology that would look like this:

Example 8-1.

```
describe 'Baking cookies' do
  it 'gathers ingredients'
  it 'preheats the oven'
  it 'measures the ingredients'
  it 'combines ingredients in order'
  it 'bakes the cookies'
end
```

In this example, you describe the collection of tests that need to be grouped to test “Baking cookies.” Within this collection, you have a test per expectation to ensure that you’re successful in baking cookies.

If you wanted to do further grouping, you add additional describes within the block, or use the RSpec keyword *context*, which is an alias for *describe*. Context and *describe* have no semantic code-level difference. The intent is to provide a mechanism to heighten understanding for humans. So in this

example, you could add context for when baking chocolate chip cookies versus peanut butter cookies.

TIP

Learn more about testing with RSpec from the [Effective Testing with RSpec 3](#) book by Myron Marston and Ian Dees.

Chefspec extends RSpec providing additional matchers to common Chef resources.

Writing Unit Tests for Datadog Install Recipe

In the following examples, I'm going to demonstrate how I'd think through setting up tests for a new datadog cookbook.

The software versions used in this example:

- RSpec 3.8

NOTE

It is not required to have a complete Chef working environment to write or run unit tests on Chef cookbooks.

The first thing I do when I'm working on creating a new cookbook is to check to make sure that it doesn't already exist in some form whether it's something that I can use directly or copy examples from. For this scenario, I'm going to pretend that there isn't a cookbook I can snag any content from, but in practice, there are often community cookbooks for many common scenarios.

The next thing I do is review the installation instructions for the version of the software that I'm planning on installing. For this example, looking at

the installation instructions for the Datadog agent helps me think about what a successful install looks like.

Installing the Datadog agent on a Debian instance requires me to:

1. Set up apt.
2. Configure the Datadog deb repo and import the key.
3. Install the *datadog-agent* package.
4. Copy a sample config file into place.
5. Update the config file with my API key.
6. Start the *datadog-agent* service.

Good unit tests for a recipe in this cookbook will focus on just the portions of Datadog that I'm writing. This first step of setting up apt should be handled by other Chef configuration code if I'm already managing with Chef.

Since the first Datadog specific thing that I need to do is configure the repo, that will be the first unit test that I write.

Example 8-2. Initial Datadog agent install_spec.rb Unit Test

```
require 'chefspect'

describe 'datadog_agent::install' do
  platform 'debian'
  describe 'adds the datadog repository with key' do
    it { is_expected.to add_apt_repository('datadog') }
  end
end
```

I create an initial `install_spec.rb` text file that will contain the unit tests.

I start with a *require* statement so that the `chefspect` gem is included. This is Ruby syntax that makes sure that the `chefspect` library is loaded.

Next, the *describe* keyword is RSpec syntax. I'm creating a block of code that says that I'm describing the `datadog_agent` cookbook install recipe.

Within this block, I'm defining the *platform* as *debian*. This is how I signal Chfspec about a specific platform. If I had different versions of Debian, or different operating systems entirely, I would have different blocks at this level specifying the different platforms under observation.

With the next *describe* keyword, I'm creating another block to say that I'm adding the *datadog* repository.

Within this block, I'm using the *it* RSpec keyword to signal an "Example." With it, I'm specifying the specific behavior I'm expecting.

With the *expect* RSpec keyword, I define an "Expectation." If I "read" this Expectation, it says, "It is expected that our code will add an apt repository *datadog*."

TIP

Using the *chef* command provided with Chef Workstation to generate cookbooks and recipes will auto-provision the unit test file you need associated with the recipe in the right place within your project.

For example, if I run *chef generate recipe install* within the *datadog_agent* cookbook, it creates a new file *spec/unit/recipes/install_spec.rb* for me that I can then update with my unit tests.

To run the tests from the command line, I execute `$ chef exec rspec spec/unit/recipes/install_spec.rb --color`.

Example 8-3. Initial RSpec Output

Failures:

```
1) datadog_agent::install adds the datadog repository with key should add
apt_repository "datadog"
  Failure/Error: it { is_expected.to add_apt_repository('datadog') }

    expected "apt_repository[datadog]" with action :add to be in Chef run.
Other apt_repository resources:
```

```
# ./spec/unit/recipes/install_spec.rb:12:in `block (3 levels) in <top
(required)>'
```

Finished in 0.43668 seconds (files took 4.94 seconds to load)

1 example, 1 failure

By specifying `--color`, I will see the failure in red, the red in my red, green, refactor cycle.

Example 8-4. After Adding the `apt_repository` Resource

```
$ chef exec rspec spec/unit/recipes/install_spec.rb --color
.
```

Finished in 0.45954 seconds (files took 5.29 seconds to load)

1 example, 0 failures

After writing the Chef infracode, I can then re-run my test and see that my test passes. I would then go through the rest of the list of requirements adding a test example for each of the expectations that I have of my code making sure that I'm covering the happy path. I would also add relevant negative tests.

TIP

In larger projects, for example a cookbook with many recipes, it is common to create a `spec_helper.rb` file which would include `require chefspec` and any other common setup tasks. Then in the spec file, we could add `require 'spec_helper'`.

Using the chef CLI to generate recipes will automatically set up this `spec_helper.rb` and add `require 'spec_helper'` to the newly created file.

This helps eliminate duplication across multiple files and is a practice that can be replicated with other testing tools.

WHEN SHOULD YOU WRITE INFRACODE UNIT TESTS?

Generally, very simple infracode doesn't require unit tests! It's only when starting to use more complex patterns (like custom resources in Chef) that it becomes really critical to have unit tests. Remember to assess the value of the tests because there are inherent costs to maintenance. Crufty tests can inhibit folks from collaborating!

Writing Integration Tests for Infracode

Integration tests for infracode can be narrow or broad depending on how many components we are testing against. At the team level, decisions can be made about the test structure. Many times, automating testing infrastructure at all is a big step!

Let's dig into infracode integration testing with Chef. As with unit testing, I'll introduce a few tools and key concepts from those tools and then run through an example.

Test Kitchen is an application that integrates with different cloud providers and virtualization technologies. It can leverage whatever infracode definitions in use in an environment in combination with a configuration file that defines the set of test suites to run with those applications. With a configuration file bundled with a project, it allows for team members to quickly set up the infrastructure and project environment to start testing and developing.

I'll be using Test Kitchen to spin up an instance so that I can do integration testing. I go into more detail in the Infrastructure in Practice chapter, for now I want to focus on the testing tools themselves.

NOTE

Test Kitchen is packaged as part of the [Chef Workstation](#). It is also distributed as a Ruby gem and can be installed using the standard Ruby gem mechanisms.

Writing Integration Tests for Datadog Install Recipe

The software versions I'm using in this example:

- Chef InSpec 3.9.3

Depending on how the team does integration testing, the scope of what falls into integration tests varies. An example of this is whether you would test the integration with the Datadog service and verify that setup actually

works correctly or if you mock out connecting to the service and assume that everything will just work in different environments.

To write metrics to the Datadog service, you must use a valid API key. Since Datadog pricing is based on instance count as of this writing, it's important to remove any test instances after verification.

With integration testing, examine the expectations and operation of the software a bit more. The main components of the Datadog agent include:

- collector - runs checks and collects metrics,
- forwarder - sends data to the Datadog service,
- APM agent - collects traces,
- process agent - collects live process information.

NOTE

On Windows systems Datadog system components are named differently.

These components bind on either 3 or 4 ports depending on the operating system.

Chef InSpec is a framework for testing and auditing applications and infrastructure. It can be used with any type of infrastructure platform to verify the outcomes of code on application and infrastructure and not the code itself. Stating this in a different way, the tests that are written for verifying that code works as expected during development can be reused to audit what is currently in production as well.

InSpec organizes tests suites into profiles, which are then categorized into controls.

TIP

InSpec profiles can be shared and used separately from cookbook code for example on [Supermarket](#) or GitHub directly. The [DevSec project](#) is a popular set of profiles that are built and maintained by the community to implement baselines for hardening popular infrastructure and applications.

You have to start somewhere when writing integration tests. InSpec has a number of **resources** to simplify writing tests. The package and service InSpec resources can be used to verify that the datadog-agent package is installed and that the service is configured to run as expected.

This configuration uses the package and service InSpec resources. The syntax is very similar to RSpec as it's inspired by Serverspec, an extension of RSpec. A key difference between Serverspec and InSpec is the format of the resources. InSpec handles a lot of the setup with the resource definition so the test profiles are a little easier to read.

Example 8-5.

```
describe package('datadog-agent') do
  it { should be_installed }
end

describe service('datadog-agent') do
  it { should be_installed }
  it { should be_enabled }
  it { should be_running }
end
```

To verify that the repository is setup, you can use the **apt** resource.

NOTE

Serverspec is another integration and acceptance testing tool for testing infrastructure as code works as expected.

When writing Chef code, the infracode describing configuration files on the system are written to the system exactly the way you specify. If there is an

error in a template, Chef won't catch the error with the content (unless it's failing Ruby or Chef code). One helpful command from the Datadog command line interface *datadog-agent status* verifies the Datadog configuration, checks the health of the agent, and version information. It can be leveraged with the InSpec *command* resource to validate that the configuration specified is valid syntax.

Another interesting validation might be to check that the agent running is the same as the version installed. Maybe during an upgrade it's possible that the older version lingers and is still the active running process.

The examples I have described in this chapter have testing duplication with testing the repository, package installation and service setup. It's important to recognize this so your teams can choose what gets tested at the unit and integration levels in your environment.

In many ways, the shape of infrastructure as code testing does not and should not follow the standard pyramid shape. Unit tests are more useful in testing complexity in the code and infrastructure code isn't always complex.

Linting Chef Code with Rubocop and Foodcritic

There are 2 useful linters for checking Chef code. First, Rubocop is a Ruby linter that can be useful for identifying issues as Chef cookbooks are Ruby. Second, **Foodcritic** is a Chef linter that is specifically useful for checking Chef cookbook usage.

NOTE

Rubocop and Foodcritic are packaged as part of **Chef Workstation**. They are also distributed as Ruby gems and can be installed using the standard Ruby gem mechanisms.

Issues in Rubocop are known as cops. Cops are categorized into classes of offenses. Different classes, including Style, Layout, Naming, Lint, Metrics, Performance, and Security, can help in a variety of ways, including finding ambiguities or errors in the code, measuring properties of the code, offering

replacements for slower Ruby idioms, and catching known security vulnerabilities.

Rubocop can be customized with a `.rubocop.yml` file to choose which concerns to comply with or ignore. For example if we want to customize the line length to 100 characters, we would have a line within `.rubocop.yml` to configure it:

Example 8-6.

Metrics/LineLength:

Max: 100

Centralize Ruby style guide across an organization with a `.rubocop.yml` that can then be referenced within a project's local `.rubocop.yml` with the `inherit_from` directive.

TIP

Learn more about Rubocop from the [Rubocop doc website](#).

Foodcritic analyzes Chef code looking for issues with portability, potential run-time failures, and anti-patterns. Instead of cops, Foodcritic describes items as rules. Each rule has tags associated with it. It's possible to restrict checking based on a particular tag or a specific rule.

Foodcritic can be extended with organization-specific rules (or **adopt other's shared rules**). Foodcritic can also be customized with a `.foodcritic` file to ignore rules that shouldn't be applied.

TIP

Learn more about Foodcritic from the [project website](#).

The software versions I'm using in this section:

- Rubocop 0.55.0
- Foodcritic 15.1.0

TIP

Due to the nature of linters and evolution of recommended practices, linter versions can be especially sensitive. If one person has one version of lint software on their system and someone else has a different version they can have competing changes that influence how they write code causing needless conflicts when trying to work on the same project.

Cookstyle is one way to help with versioning conflicts with Rubocop when working with Chef code. It pins to a specific version of Rubocop, and has a set of specific conventions preconfigured for cookbook development. Cookstyle is included in the Chef Workstation.

In this example, various types of issues have been found in the install recipe by the ruby linter Rubocop.

Example 8-7.

```
$ chef exec rubocop recipes/install.rb
Inspecting 1 file
C
```

Offenses:

```
recipes/install.rb:8:1: C: Layout/IndentationWidth: Use 2 (not 4) spaces for
indentation.
    keyserver keyserver
    ^^^^^
recipes/install.rb:15:4: C: Layout/TrailingBlankLines: Final newline missing.
end
```

1 file Inspected, 2 offenses detected

Both of these errors are issues with layout. With the first issue, Ruby should have 2 spaces for indentation, but the code has 4. In the second reported issue, Ruby files should have a final newline. The message provided with the error message give guidance towards how to fix the issues.

Example 8-8.

```
$ chef exec foodcritic .
Checking 3 files
X..
FC008: Generated cookbook metadata needs updating: ./metadata.rb:2
FC008: Generated cookbook metadata needs updating: ./metadata.rb:3
FC064: Ensure issues_url is set in metadata: ./metadata.rb:1
FC065: Ensure source_url is set in metadata: ./metadata.rb:1
FC067: Ensure at least one platform supported in metadata: ./metadata.rb:1
FC078: Ensure cookbook shared under an OSI-approved open source license:
./metadata.rb:1
FC093: Generated README text needs updating: ./README.md:1
```

Foodcritic output is not the same as Rubocop, because Rubocop is checking against Ruby standards while Foodcritic is checking against Chef recommended practices. You can look at the [project website](#) and drill down to the specific reported issue to find examples of how to resolve the issue that has been reported.

For example, FC067 has an issue due to the fact that the cookbook's metadata file isn't updated to provide information about what platforms are supported by the cookbook.

TIP

Since I'm using the Chef Workstation bundled versions of software to show these examples, I'm running `chef exec SOFTWARE` which runs arbitrary software within the context of the Chef Workstation environment, for example setting up the `PATH` environment variable and the `GEM_HOME` and `GEM_PATH` Ruby environment variables.

When Foodcritic or Rubocop return a violation, this doesn't automatically mean that the code needs to be changed. It's important to examine the issues and identify whether they are real problems or areas where customizations to the lint configuration file need to be made.

Wrapping Up

In this chapter, I showed different ways you can use testing in practice beyond exploratory testing to automate and make work more predictable,

repeatable, and collaborative.

Chapter 9. Security and Infracode

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 12th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at vwilson@oreilly.com.

Defense in depth tells you to apply security practices at different layers in order to deter harm to your infrastructure. The security mindset improves the reliability, robustness, and general operability of your applications, tools, and services. Infrastructure as Code practices in modern system administration presents an opportunity to apply the security mindset in a scalable way. In this chapter, let’s examine some common example areas (identity and access, secrets, compute and network infrastructure) to consider when writing your infracode.

Managing Identity and Access

Depending on the length of time you have been administering systems, the operating systems in your environment and whether you have started using

hosted services there may be a variety of different ways you've managed users and access including:

- Synchronizing `/etc/passwd` and preventing duplicate user ids,
- Managing LDAP, kerberos, or Active Directory,
- Managing identities in an `htpasswd` file,
- Running sql scripts to add users and grant roles to MySQL databases

These types of processes play an important role in contemporary system administration. Additionally, these processes have been augmented with new tools and technologies that facilitate automation, transparency and compliance.

Some of these processes may still remain as valid methods of managing user access. There are even more possible processes and technologies now to facilitate automation, transparency and compliance.

How should you control access to your system?

Identity and access management is how you configure roles and privileges for users, groups, and services and the underlying technology and processes that support the allocation and revocation of privilege.

There are three core elements of identity and access management:

- **Authentication** - A user is who they say they are.
- **Authorization** - A user has the privilege to do the requested action.
- **Activity Logging** - The recording of user actions via logging.

In addition to any in-house solutions you manage, external services implement identity and access management in their own way with different terminology and concepts. This can even apply to specific services offered by a given provider (e.g. compute instance versus database authentication

and authorization). You'll need to read the specific documentation associated with the service you plan to use to understand exactly how authentication, authorization, and logging is done.

Examples of service providers and their identity services include:

- Amazon AWS Identity and Access Management (IAM)
- Google GCP Cloud Identity and Identity and Access Management
- Microsoft Azure Active Directory

If you are starting a new position, migrating to a different cloud, or using a new web service, don't assume that identity implementations are the same. Due to differences between services and providers, you can accidentally weaken your system with misconfigurations.

Examples of how modern infrastructure identity practices change include:

- Instead of a single factor of authentication such as a password to log into a system, you might require multi-factor authentication(MFA) which requires multiple pieces of evidence to verify that the individual is who they say they are; usually with something they know like a password or PIN and something they have like a security token or card.
- Instead of synchronizing and centralizing /etc/passwd across many UNIX systems or binding them to an LDAP directory, you might rely on configuration infracode to ensure users have accounts only on the systems they need.

Identity and access management can get really complex. For example, in a hybrid scenario where you manage identities with a corporate user directory external from your service provider, you might have to manage trust relationships and federation between different services. This allows you to share authentication methods across services so that users can use existing credentials.

Complexity is also increased by the need for identity and access management in different domains, such as corporate identities within an organization, service identities to enable communication between applications, and consumer identities to access customer facing services.

Most likely, the set of tools you use for identity and access management for the variety of services you manage is more complex than it used to be. Leveraging infracode allows you to have consistent, repeatable, and testable configurations. You'll also need clear processes, especially around on-boarding and off-boarding of employees, to configure anything that doesn't integrate with automation.

Creating more developer friendly ways to manage provisioning of resources also creates the need for additional guardrails and audits. For example, one of the most common access misconfigurations with object storage services like AWS S3 is to configure full anonymous access to a bucket or allowing anyone to read or write to the bucket. How does this happen? Many how-to guides illustrate the concepts behind services by having developers immediately open up access to make it easy to focus on learning the service and don't explain exactly what those configurations do. These patterns then get copied into live environments and create vulnerabilities. Providing example infracode snippets that reflect best practices can help make it easier for others and keep settings uniform across your organization.

You may need to audit for issues in your environment and educate other engineers within your organization to use specific technology. For example, you may wish to ensure everyone has MFA enabled for their accounts. You might set up automation that regularly scans for accounts missing MFA and notifies the account holder to remediate by adding MFA or deactivate the account.

You can leverage your infracode tools of choice to track, audit, and modify corporate and service identities to your systems, as part of your provisioning process. This ensures the settings you encode are applied uniformly, and when your needs change, the tooling makes pushing the change out easier.

Who should have access to your system?

Once you figure out how you control access to your variety of systems, then it's a matter of figuring out who should have access to your system.

When reviewing application or service documentation, you can often find guidance about expectations on running the systems including what accounts are needed and any associated permissions.

Other areas to identify include:

- Are elevated privileges required for individual or service accounts?
- Should there be time boundaries around access?
- Do users who have logged in require a different experience from a casual anonymous user?

You can minimize the scope of possible harm to your system by applying the principles of least privilege and segregation of duties when granting access to your systems. This ensures that a user or component only has access and authority to what they need, versus having root or Administrator accounts. Putting this in another way, if an account is compromised then the harm that can be done to the system is limited to components of the system that the account has access or authority over with those credentials.

Additionally, I examine what application programming interfaces (APIs) are available. Often, this is seen as the realm of developers but these are often the critical vectors of attack to your systems. Most modern web applications expose APIs to users in some form and many cloud providers have an API gateway service to configure and manage access to data and other backend services.

You can examine what application programming interfaces (APIs) are available. Often, this is seen as the realm of developers, but these are often the critical vectors of attack to your systems. Most modern web applications expose APIs to users in some form; check what your service is exposing and that it's intentional. In hosted services, the provider's API gateway is

used to configure and manage access to systems, data and other backend services.

IAM and logging is analogous to the door locks, security cameras, and other physical controls of an on-premises datacenter or server closet and . Infracode is a practical necessity to ensure these “doors” remain appropriately “locked”.

Managing Secrets

Engineers want to get work done as quickly as possible with the least amount of barriers, sometimes trusting the privacy of applications that don't have any notion of privacy or accidentally adding them to source control. Often you have incomplete visibility of what risks you have from exposed secrets, as there may be secrets embedded in code and different services require different processes

Secrets are subject to a bootstrapping problem: If I need to get access to a particular resource, how do I do it? If I need a password, how do I get that password? Early in my career, I remember being handed a carefully written sticky note and informed that it was critical to memorize the password and then destroy the note. Resetting root and Administrator passwords when anyone left the team, while also ensuring everyone remaining who needed access had access, was problematic.

In contemporary environments, you also need to keep more than host passwords secret from people who shouldn't have access to them. Secrets include passwords, mTLS certificates, bearer tokens, and API keys. Using infracode to establish best practices around secret management can help you increase adoption and track your progress. Infracode also introduces new challenges for secret management, as the infracode tools require access to the secrets. Let's examine the tools and concerns that help to manage secrets.

Password Managers and Secret Management Software

Sometimes secrets need to be accessed or used by humans, sometimes by automated processes, and sometimes both. These access patterns dictate what type of interface is best, and so secret management software is usually tailored mainly for one use or the other.

When the primary concern is interactive use by humans, secret management software is usually called a password manager or privileged access management application. Using a password manager, you can generate and store strong, unique passwords. This helps prevent reusing passwords across sites, and enables sharing secrets across the team without resorting to insecure methods like writing them down or sending them over collaboration services or email. Some well-known password managers include:

- 1Password
- Lastpass
- KeePass
- BitWarden
- pass

Secret management software for use by other applications is a key-value database with authentication and auditing features. Vendors add value to their secret management solution by integrating with different software ecosystems or supporting specific usage patterns.

Examples of secret management integrated with IaC include:

- Chef Infra with encrypted data bags and Chef Vault
- Puppet and the Hiera eyaml extension
- Ansible Vault
- Salt Stack with Pillar

These IaC platforms allow you to store secrets encrypted that is decrypted at run time on the configuration of your compute infrastructure.

Service provider-specific methods for storing secrets include:

- Amazon AWS Secrets Manager
- Google GCP Secret Manager
- Microsoft Azure Key Vault

Stand alone secret management tools that you can leverage within your code and configuration infracode include:

- Keywhiz
- Knox
- Confidant
- Hashicorp Vault

The primary purpose of a secret management platform is to allow you to decouple storage of secrets from the code or configuration that consumes the secrets. Besides the ability to support that decoupling, you should evaluate secret management software for other concerns, including:

- **Centralization** - all secrets are stored in one place reducing the risk of leaking secrets via storing it in the code or forgetting about their existence
- **Revocation** - marking a secret invalid and no longer trusted
- **Rotation** - updating credentials for an identity. This may include versioning of the secret allowing for progressive rollout of a new secret so that you don't create brittle interdependencies between secrets and applications.
- **Isolation** - ability to assign secrets to individuals or roles, so that the least amount of privilege is granted as needed. A single application doesn't need full access to all project secrets.

- **Inventory** - visibility of secrets being stored (separate from access of secret data itself) to eliminate secret sprawl.
- **Storage** - visibility and configuration of how and where secrets are stored and replicated.
- **Auditing** - interactions with secrets are logged and monitored.
- **Encryption** - secrets are encrypted at rest and during transit. Secrets shouldn't be written to disk or transmitted over networks in clear text.
- **Generation** - creation of new secrets.
- **Integration support** - usability with other services and ability to integrate with your own software.
- **Reliability** - secret access needs to be reliable. If the secret store is down, how do specific services and systems work?

Defending Secrets and Monitoring Usage

Monitoring access to and usage of credentials and other secrets is an important layer of your defense-in-depth strategy. Secrets can leak in many ways, so it's important to have mechanisms in place to detect and respond when that happens. Some ways that secrets get leaked include command history, debug logs, and the use of environment variables. Environment variables deserve special attention because they are available to the process and secrets there may be exposed through a process listing with no audit logs to trace exposure.

In 2020, **rogue activity** was detected within the Ubiquiti network and traced back to the misuse of an IT administrator's credentials that had been inside Lastpass. Lack of logging made it impossible to track what had been done by malicious attackers while they had access to the systems. Even if you assume that anyone that has access to your system should have access to all secrets at any time, think about the risk from third party services that ingest logs that may contain the secret in plain text. Consider the journey of a

secret that is logged during a problem; for example, it may be ingested by Splunk, included in a PagerDuty alert, and sent through email and text messaging.

You want to know what systems are available (and should be!) as well as be able to detect the use of credentials in unexpected ways (from different source IPs or at different times). Many applications and services provide account anomaly detection to enable you to see this unexpected behavior. This is a great opportunity to collaborate with your security team, if you have one.

To identify the breadth and depth of compromise, you need a comprehensive and clear data management strategy for audit logs. This includes separation of privileges so that administrative activities on systems are separate from administrative activities on the audit logs.

In traditional environments, you had to worry about managing user access. Now, you need to worry about service access as well. Tools and techniques have evolved, yet secret management is still problematic especially for machine to machine communication. Often you have incomplete visibility of what risks you have from exposed secrets, as there may be secrets embedded in code and different services require different processes. Access logs from secret management software can help with this problem: services that access secrets will have a certain pattern, which can help make anomalous access more visible. Also, you can audit which services or applications don't use the chosen secret management software; this may indicate places where secrets are accessed in a risky way. Infracode can help close those gaps.

Securing Compute Infrastructure

The efforts you must undertake to secure your compute infrastructure will depend on what types of services you use. For example, the cost of using managed services includes the service provider owning the responsibility of securing the infrastructure underlying those services. For virtual machines and containers that you choose to run, the service provider only provides

the physical security and operating environment (hypervisor or container host) your workload is running on. Infracode can make it easier to secure the parts of the stack you remain responsible for and ensure its use.

Operating systems and applications often default to open configurations prioritizing ease of use over security. For services that require operating system and application management, you can reduce the exposed attack surface by securing the configuration. This is a common compliance requirement under many regulations and standards, including the Payment Card Industry Data Security Standard (PCI-DSS), ISO 27001, and the United States' Sarbanes-Oxley Act (SOX) and Federal Information Security Management Act (FISMA). Some resources that provide guidelines include:

- The Center for Internet Security (CIS) [implementation guides](#).
- [The Security Technical Implementation Guides \(STIGs\)](#)

These peer-reviewed standards are available for a wide variety of operating systems, popular applications, and network devices. They are filled with detailed instructions for tightening all sorts of security-related settings, some of which may not be appropriate for your situation. Review standards and implement recommendations that make sense for your industry and environment.

Another key part of managing the security of compute infrastructure is patching the operating system(OS), installed packages, and applications. Patching can be difficult due to application dependencies on specific versions of OS or other packages, unsustainable deployment practices, or fear of compatibility and stability problems. Infracode can help to address all these concerns. If an application or package has specific requirements, the requirements can be reflected in the infracode. The automated, repeatable nature of infracode encourages frequent deployment and can enable testing of patches for critical systems. Automated testing can be implemented to test different versions of dependencies to expose the risk to patching and provide peace of mind to proceed with patching as needed.

You need to update a containerized application the same way you would need to update it if the application was running directly on a server. Most container images include a significant number of OS packages that will require periodic updates. You can use infracode to build new, patched container images, test, and deploy them.

The twelve-factor app recommends to *explicitly declare and isolate dependencies* which eliminates the implicit dependence on system-wide packages. By including a manifest with specific versions of applications, you can reproduce builds reliably without impacting the underlying operating system. Additionally, it provides a path to test builds with new versions by updating the manifest rather than relying on available upgrades from your operating system vendor. If you isolate dependencies, remember that in addition to OS patching, you need to plan to keep your dependency manifest up to date as well, which includes rebuilding, testing, and redeploying your application.

Managing Networking

Network controls provide defense in depth for networked services. If an attacker is unable to communicate with a service, then they can't attack that service directly regardless of vulnerabilities or misconfigurations it may have. This basic insight led to the development of the classic two- or three-ring network layout. Sysadmins would create a trusted core to contain most of an organization's systems and configure firewalls to limit incoming access to that core from outer, less-trusted network zones. In this model, publicly-accessible systems such as web servers would go in the outermost zone, often called the demilitarized zone (DMZ).

This has been described as “candy bar network security”: crunchy on the outside, chewy on the inside. The idea is that attention is focused on the perimeter and assuming that anyone accessing internal resources is doing what they need to and needs a minimal friction experience.

The shortcomings of the classic two- or three-ring trust-based network become apparent when that network isolation starts being used as the

primary defense for insecure systems or protocols. An attacker who is able to gain access to one system in the trusted core then enters a playground of insecure systems.

To combat the shortcomings of “candy bar security”, the industry has moved towards a zero trust architecture model. The key principles of zero trust is :

- no implicit trust is granted between entities based on their location
- required authentication and authorization
- protection is oriented around resources rather than network segments

In other words, each authorized and authenticated entity on the network (such as a server or a person’s workstation) can communicate only with the services allowed based on established policies.

Infracode is a key enabler for moving toward a zero-trust network. Zero-trust ideas can be built into your network no matter what tech you use. In an on-premise, hardware-based environment, infracode allows the adoption of much finer-grained network segmentation along whatever boundaries make sense for your needs. Software-Defined Networking products take this flexibility a step further, being designed specifically to adapt their configuration quickly and easily as you add and remove servers and services. In the cloud, infracode makes it easier for you to integrate features such as AWS’s Security Groups with the rest of your provisioning workflow. When new systems are being added, consider what services they need to communicate with, and restrict network communication to only those services. The initial effort of mapping these network dependencies is rewarded later by an easier to understand architecture with data flows explicitly documented in the infracode.

The dynamic nature of containerized and serverless workloads presents further challenges and opportunities for network segmentation. Most products and services have built-in or add-on features to enable zero-trust-style networking integrated with the workload orchestration. For example,

Network Policies in Kubernetes can target specific pods according to the familiar selectors admins and developers use for everything else. If you want to utilize Network Policies in Kubernetes, it's important to make sure your chosen Kubernetes network plugin supports the features required to achieve your network security goals.

Learn more about zero trust:

- Forrester Research “No More Chewy Centers”
- Google BeyondCorp project
- [NIST SP 800-207](#) provides a formal reference

Recommendations for your Security Infracode

If your organization has little to no IaC practices currently, start with understanding the infracode practices in use or planned. Integrate security into your initial plans or add them to your overall strategy.

1. Verify who has the access to run automation and infracode. Make sure that this privilege is limited to only what is necessary to perform those tasks and isolated from modification of the logging of those tasks.
2. Generate and store credentials safely.
3. Don't reuse user or service credentials. With identity and access management, it's possible to generate and revoke the credentials to be used as needed.
4. Check that provisioning infracode only grants the necessary privileges required to users and resources (e.g. virtual machines).

5. Check for resource configurations that can strengthen the integrity of the resources you are using.

For example, with this Terraform snippet you enable **uniform bucket-level access** and provide the key used to encrypt objects in a Google Cloud Storage bucket:

6. Add static code analysis to scan your infracode for security misconfigurations or missing best practices.

For example, **checkov** is an open source tool to scan infracode. Running a scan on the Cloud Storage bucket terraform example above returns the following:

```
terraform scan results:
```

```
Passed checks: 2, Failed checks: 0, Skipped checks: 0
```

```
Check: CKV_GCP_5: "Ensure Google storage bucket have encryption enabled"
```

```
    PASSED for resource: google_storage_bucket.static-assets
```

```
    File: /gcp_bucket.tf:1-7
```

```
    Guide: https://docs.bridgecrew.io/docs/bc\_gcp\_gcs\_1
```

```
Check: CKV_GCP_29: "Ensure that Cloud Storage buckets have uniform bucket-level access enabled"
```

```
    PASSED for resource: google_storage_bucket.static-assets
```

```
    File: /gcp_bucket.tf:1-7
```

```
    Guide: https://docs.bridgecrew.io/docs/bc\_gcp\_gcs\_2
```

7. Scan your version control repositories for secrets. For example, **gitleaks** is an open source tool to detect hardcoded secrets within git repos. Hosted source control services like **GitHub** have started providing secret scanning services that alert repository admins and organization owners about potential leaks.

Part IV. Scaling Production Readiness

Chapter 10. Monitoring Theory

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 14th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at vwilson@oreilly.com.

Monitoring is the process of measuring, collecting, storing, exploring, and visualizing data from infrastructure (including hardware, software, and human processes). Monitoring helps you answer the “when” and “why” questions of your work, and it informs business decisions that support humans working in a sustainable manner(e.g., hiring so that your sysadmins are not constantly working at full capacity).

In this chapter, I focus on broad monitoring theory with the goal of providing you a framework to identify effective monitoring strategies. I will differentiate monitoring from observability, and explain the elements and steps of the monitoring process and how they work together. Understanding these mechanics at a high level will help you prioritize the different desirable outcomes monitoring makes possible, decide how and what you monitor, and increase visibility into your workflow, systems and teams, regardless of the tools you choose.

Why Monitor?

There are many reasons to monitor. All of them involve increasing visibility into your systems. Visibility brings attention to weaknesses, fragility or risk, and helps you make better decisions. Some examples of achieving this visibility include:

- Problem discovery: You want to identify problems and know when and how those problems have been resolved (e.g., monitoring latencies of web requests and identifying when slow MySQL queries are impacting customers).
- Process improvement: You want to identify areas where your processes can be improved to make sure that your team is not overworked, increase accuracy and speed of task resolution, automate toil work, and improve overall efficacy (e.g., monitoring work queues to identify impact on the team).
- Risk management: You want to identify, evaluate, and prioritize potential problems (e.g., monitoring deployments of software, and adjusting automation or processes to reduce the frequency and severity of surprises).
- Baseline behaviors: You want to identify how the system behaves with normal traffic (e.g., monitoring data over a longer period to see your service trends to analyze different events that occur like holidays, weekends, and predictable news events like elections and sports events).
- Budget setting: You want to identify, evaluate, and prioritize infrastructure investment and enforce accountability related to spending (e.g., monitoring infrastructure spend to identify areas where different solutions may be more cost effective or set up constraints that enable engineers to test out new solutions without worrying about a surprise bill).

- Capacity management: You want to build sustainable capacity based on business demand. (e.g., monitoring infrastructure to identify when reserved instances will save money over ad-hoc instances).

Monitoring is so much more than implementing a single tool; it's identifying what you're trying to learn and desirable outcomes, and then assessing available tools and implementing practices that will best help you get there. Thinking about why you are monitoring, and establishing specific monitoring objectives encourages critical thinking around your business context so that you avoid copying specific vendor-implied monitoring practices into your organization that aren't a good fit for your goals.

BE YOUR OWN AUTHORITY

A lot of practitioners tell us why and what to monitor, but I'm here to tell you that you are the best authority on your environment. Imagine for example that you are running a web service for your company. While it might be the same software in use at other organizations, the specifics about the web service vary between those organizations. You know your specific risks based on failures in different parts of the service as well as the different individuals that are responsible within your organization from development to support. All of these variables affect what needs to be monitored and the specific actions that need to be taken to derive the most business value while supporting the humans that run the software.

How Monitoring and Observability Differ?

Rudolf E. Kálmán introduced the concept of observability for linear dynamic systems in the 70s. Observability is a measure of how well you can see inside a system under observation with just the outputs. A system, in this case, is the collection of interrelated objects that are treated as a

whole to model behavior. For example, a system may be a single host, container, or an entire distributed service.

Observability is not monitoring, and monitoring is not observability. Observability is a property of a system; monitoring is a multi-step process of observing a system. Often, individuals think of monitoring as dashboards and production alerts. Framing monitoring in this manner leads people to define monitoring as a subset of observability. The problem with this definition then becomes: What do you call the other activities that you need to monitor?

You end up with overlapping terminology to cover all the potential use cases while also increasing the potential for misunderstanding. Monitoring has always been a broad process with a variety of different practices across organizations.

In some ways, it's a lot easier to think about the "unobservability" of a system. Imagine for a moment that your customers experience a problem that your dashboards and alerts don't identify or explain. If your underlying data doesn't help you explain why and how the problem occurred, that indicates a lack of observability.

You can monitor the observability of your systems by assessing the variety of problems that occur, how often you are able to answer questions with existing data, and how often the final assessment of why a problem occurred is "I don't know."

NOTE

Terms are constantly evolving across teams, organizations, and the industry. Conflict arises in the monitoring community of practice over these terms signalling that there is a lack of shared context. For example, monitoring and observability and whether observability is a subset or superset of monitoring. Often these conflicts arise and build over vendors trying to be the perfect solution and in the process reusing words to mean different things.

Take time to build the shared context within the team around your use of monitoring terms. Then as you assess different vendor's monitoring offerings you will be better prepared to compare implementations and choose solutions that map to the way that your team works and thinks about monitoring.

Monitoring Building Blocks

To better communicate the process of monitoring, let's define some critical terms: events, monitors, metrics, logs and tracing.

Events

An event is a thing that happens, a fact that can be tracked. An event may be system, application, or service specific. Events occur regardless of whether they are being monitored.

Examples of events include:

- CPU utilization at a certain time
- The execution of specific code
- A sysadmin terminates an instance

Monitors

A monitor is a tool that defines and captures events of interest. They can be fixed or flexible.

Fixed monitors are specific functional checks against known issues that can't be customized by individuals at run-time. Examples of fixed monitors

include event logs and CPU or memory gauges.

Flexible monitors can be changed ad-hoc. Tracing is an example of a flexible monitor that captures and records events. For instance, on a Linux system, you can run `strace` on a process to capture all the system calls made by that process. Flexible monitors are often used in diagnosing issues.

Monitors can be narrow or broad. Narrow monitors might define an event as a single instruction like a log that is triggered. Broad monitors might define an event as an aggregate of instructions, for example, a single web request that results in many system activities.

Monitors can be event-driven or sampled periodically. Event-driven monitors execute when the event occurs and aggregate over the reporting period. Periodic sampling monitors execute at a specific interval of time, collecting a statistically significant number of events.

Data: Metrics, Logs, and Tracing

Monitors collect data about configured events into three main types: metrics, logs, and tracing. They are collected from systems, devices, applications, and networks. You may be able to apply filters to limit the data collected or to sample in a way that represents the whole.

Most metrics are time stamped numeric values represented as a counter or gauge.

- A gauge is a value that reflects a point in time. A gauge doesn't tell you anything about the previously measured values.
- A counter is a cumulative value that reflects events since a point in the past. When a counter reaches its upper or lower limit, it may roll over. Counters may be measured per time interval, and reset at the time interval. Counters may also be reset upon certain system events (such as reboots), or upon request. Counters that are measured per time interval and reset will not tell you anything about previously measured values.

Let's look at this difference between a gauge and a counter. A car's speedometer tells you how slow or fast you are driving. You use that information to guide your immediate actions by knowing whether you are traveling within posted speed limits. The car's odometer tells you how far you have gone. You use that information to guide preventative services like tire rotation and oil changes.

NOTE

Monitoring platforms may provide different metric types and implementations of these types may vary. Look carefully at the metric types, as the implementation will affect how the data about your events are collected and stored. Data reduced or aggregated too early may provide insufficient information for debugging purposes. Data that isn't reduced may lead to a flood of traffic that can impact network performance and the quality of service.

Logs are append-only records of events. Generally, logs are unstructured; the file format does not provide context or meaning to fields. Within a log, there is no implied relationship between records. Configuration changes of applications may alter which fields are displayed, affecting any scripts created to parse logs. Logs provide a lot more information than metrics but are more expensive to capture and store. Analyzing logs requires more specific tool customization.

Structured logs are structured in a key-value format that makes it easier for computers to process. Application configuration changes may affect which fields are displayed but won't impact existing scripts to parse logs. Event logs are structured logs that monitor broad events.

Tracing is a specialized form of logging to record a rich set of event data. Examples of tools that provide tracing include strace and tcpdump.

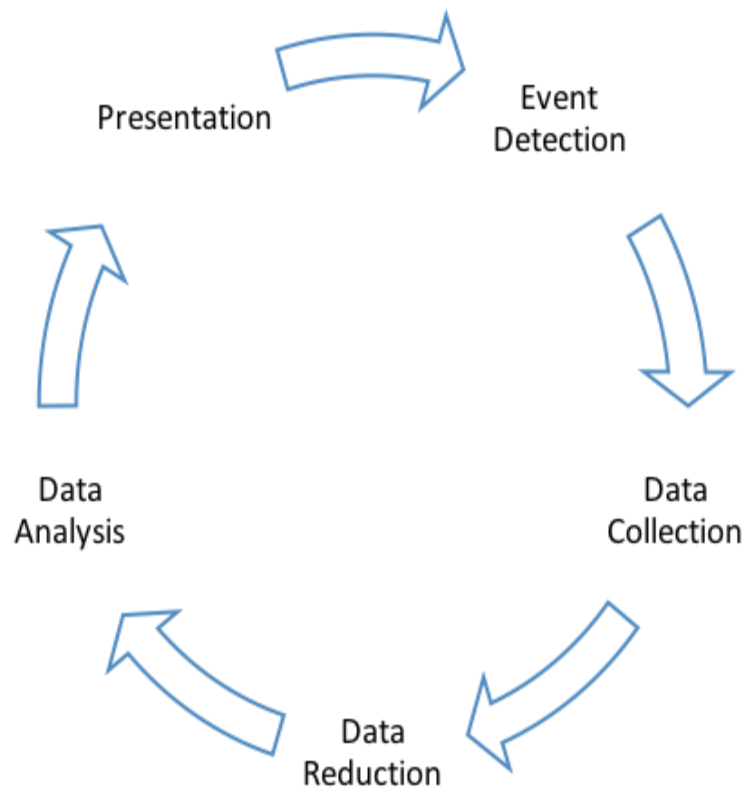
Distributed tracing is a specialized form of tracing that instruments an application to provide rich logs and metrics across different systems to connect contextual data across systems.

There are tradeoffs to consider between choosing metrics, logs, or traces. Metrics allow limited context to be associated with the data which

minimizes the amount of resources required to store. Logs allow you to associate more context to the data you collect. Traces have the highest amount of context and require the most resources to store.

What does Monitoring look like?

The monitoring process includes a set of sequential steps: event detection, data collection, data reduction, data analysis, and presentation.



Let's look at these steps individually.

Event Detection

The first step in the monitoring process is event detection; events trigger monitors. Some monitors track the absence of expected events.

Data Collection

The second step in the monitoring process is data collection when monitors collect data.

Monitored data can be collected by:

- the monitored system pushing the data to the central monitoring server on a schedule or based on an event,
- the monitoring system signaling the server to push the data, or
- the monitoring system pulling data via a health check.

NOTE

Depending on the size of your environment and what you are measuring, a central server pulling data can create a scaling issue. This is one factor to consider when evaluating platform options if you have a larger environment.

The method of collection may create an observer effect; imagine the impact of a time-based collection strategy where every monitor checks at midnight. This frequency of monitoring can cause CPU or disk resource exhaustion, which increases latency and leads to unnecessary alerting.

The method of collection may change what you monitor and how you monitor it. For example, metrics are generally event-driven and aggregated over a period to compress data.

NOTE

If you have metrics that represent people, make sure you protect their privacy and obtain their consent in the collection of their data. With personal data and PII, you may have additional rules and regulations to follow, so when possible, avoid infringing user privacy by not tracking it in the first place.

Additionally, don't assume permanent consent, especially if you change the context or method of data collection. An example where you might need to think about this is telemetry data collected and logged from an individual's use of an application.

Data Reduction

In the third step, your monitoring platform aggregates and reduces the data. While this may happen to some degree at collection time, often it makes more sense to perform separately especially with distributed data.

Your monitoring agents collect data from many different sources. Your monitoring platform may aggregate, edit, sort, or compress the data down to its essential parts.

For metrics, sometimes the older data is aggregated for storage purposes while also providing some historical accounting to show differences against baselines. Older is contextual and could be weeks, months, or years.

For example, if you are monitoring request counts, you might not need 6 months of 5 minute interval data. Instead the count data could be aggregated so you have a baseline to compare against, but with reduced resolution and no ability to examine the original 5 minute intervals from 6 months ago.

Utilization over time of some metrics may be less useful. Storing metrics costs money, so aggregation is a balance of cost and usefulness.

Data Analysis

In the fourth step, you analyse the data to discover useful information about business and direct action.

During this analysis, you identify a set of service level indicators (SLIs) that help you measure the reliability of your system.

There are a few different ways to monitor for reliability including: availability, latency, throughput, and durability.

- Availability measures whether a system is operational and can perform the service as expected.
- Latency measures the time it takes to perform an action.
- Throughput measures the number of requests passing through the system.
- Durability measures long term data protection; that the stored data doesn't degrade or get corrupted.

Once you have SLIs, you can identify the achievable and appropriate levels of reliability through setting service level objectives (SLOs). Because it is very difficult (and costly) to provide better reliability than what you depend on from external service providers, you must factor in those dependencies when setting your targets. Don't forget to factor in network and DNS.

Data Presentation

The fifth step in the monitoring process is the presentation of information. To transform data into information, you create visualizations. You collect charts into dashboards that cover areas of known bottlenecks and elevated risk. You create other ad-hoc visualizations to explore available data.

You may create charts based on real-time off-line data. For example, alerts should be as close to real-time data as possible to limit the impact of problems. Quarterly capacity planning for a Hadoop cluster may be the aggregation of various data sources and processed off-line.

Dashboards aggregate a set of visualizations to communicate information. The specific dashboards you create depend on what you envision people

doing. People could be making a one-time strategic decision, determining day to day operational direction, or reviewing the system weekly or monthly to establish tactical direction.

These dashboards are products that drive action. Outcomes and information should feed back into the various team and organizational processes.

Monitoring for Sustainable Work

The monitoring process is iterative. Monitoring provides information to help you analyze what is happening and the supporting evidence to educate the team and drive changes. Sometimes these changes are to the compute infrastructure, other times it's to the human processes. People are part of the systems that you manage from development to support in production.

For example, I have been in environments where the average work load for the Ops team meant that we each worked at our full capacity. If any one took time off whether planned or unplanned this led to extra stress on the system which led to increased mistakes in resolving incidents, and frustration with one another in the team. Monitoring helped us to establish that we needed additional people on the team based on our expected work load. This gave us extra capacity when everyone was available, but reduced friction when people needed time off.

Chapter 11. Presenting Information

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 15th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at vwilson@oreilly.com.

Stories are the fundamental way that humans organize and make sense of information. Stories provide structure and purpose to data. Effective system administrators recognize the power of a good narrative and use different mediums to share messages effectively. They organize their information and communicate beyond text to tell a story with images, photos, graphs, charts, audio, and even video. So often when mentoring other sysadmins trying to make change occur in their organization, I find myself sharing some of these key concepts about data organization and presentation.

Show five clever people the same data and they’ll come up with ten interpretations of what it means. You can’t assume that others will draw the same conclusions that you do, unless you put in the effort to craft a narrative that will lead and influence people. One well known example of compelling narratives in our industry is The Phoenix Project from Gene Kim and company. They’ve influenced many practitioners sharing the

three ways of devops through the story of Brent, the sysadmin that everyone views as a bottle neck. In this chapter, I share the skill of distilling information to convey meaning, and drive desired action through influencing people regardless of authority.

Know your audience

In the movies, the protagonist often has the ability to determine the next right step based off of a single query or dashboard that integrates all the necessary data. They can show the output and get support for their endeavors. In the real world, there is no single pane of glass possible that can provide this context and support. Additionally, you are competing for attention and acknowledgement that you have the supporting data for your conclusions.

People need insights into information that is relevant to their responsibilities, which can range from “nuts & bolts” details of the operation of a specific system, to “birds-eye” overviews of the activity in an overall environment. No single graphic or dashboard can aggregate information in a way that is useful for everyone. It is necessary to tailor each graph and dashboard to narrowly focus on the needs of a specific audience.

Presenting people with relevant, accurate, and timely information helps them carry out their duties effectively. If individuals aren't taking expected actions, this may be because they've been provided with information that is stale, vague, or inapplicable to them. If teams are overfocused on short-term speed and execution at the expense of long-term strategy, this could be an indication of a broken feedback loop.

TIP

When the team isn't taking time to reflect on how their work aligns with the organization's goals, make sure that this is not a reflection of the environment and broken feedback loops. It will be really hard to have desired impacts no matter how you modify your message in these cases.

I distinctly remember sitting in yet another meeting as a coworker tried to convey the importance of the work he was doing. He read sentences directly off the slides describing extremely boring maintenance work that talked of saving money that had already been spent. The large numbers from his measurements didn't alleviate the boredom or compel me to want to participate in the additional toil to achieve his project goals.

This experience reminds me of a Mark Twain quote: *Often, the surest way to convey misinformation is to tell the strict truth.* It's not enough to give people the cold facts and trust that they'll then be inspired to act in the way you want them to act; you have to demonstrate why those facts are compelling, how they relate to larger goals, and then create an emotional connection so people want to help your cause.

There are key questions to reflect on when you share your information to help you connect with your audience:

- Who are you communicating to?
- What is important to them?
- What do you want them to know or do?
- What do they already know?
- What is their preferred method of consuming information?
- How does your data make your point?

For example, your CTO may have many reports and need high level information distilled into scorecards. Leadership funds initiatives so you're wasting their time by going into the minutiae of your decisions. However, our peers need to be inspired and may want to explore the underlying data in order to give their support to get a project done within a timely manner.

Choosing your channel

Once you've reflected on the questions about your audience think about what you want them to do. Then decide if verbal or written communication is best — this will depend on your objective and type of message.

Verbal communication mostly happens in real time and gives you the opportunity to convey feeling along with facts. It's most useful when there is a component of emotion or sensitivity you want to communicate or if you need immediate feedback.

TIPS FOR SPEAKING

The more you present information through public speaking, the better you will get at it. Beyond practice, there are a few tips that I've learned over the years that may help your level up your speaking.

- Breathe

Especially if you are nervous, you may find yourself breathing faster or holding your breath. This comes across in your speech and affects your pace. It may even be helpful to add cues to your notes to remind yourself to breathe. Leverage pauses for emphasis and to self-check on your breathing. Pause for laughter.

- Vocabulary

Speech needs to sound more like conversation and use clear and natural words especially for technical talks. The environment of the room and the listener's experience and knowledge will all affect how they parse and understand what you are saying. While there is no escaping using technical words, avoid jargon and acronyms.

- Pitch

Modulate your voice to create inflections to drive interest in your message. Practice this on different words to see how it changes the message. When you find the right fit, make notations to your presentation.

- Pace

The right pace for your talk varies depending on your audience. You may find yourself in the moment uncovering that some of your assumptions are incorrect. In general, for simple straightforward topics, it's ok to speed up the pace. For more complex topics, you want to slow down. When you have a mixed audience of beginners and experts this is where you can enter the dreaded middle ground of expectations where

beginners may feel you went through the material too fast, while experts may feel you went through too slow. Be thoughtful and consistent in your delivery as to who your audience is and you'll satisfy at least half of your audience.

- Authenticity

Match your expression to your words. Your body language and expressions convey information. Smiling can convey energy and engagement with your topic. If your message and manner don't match, it conveys a dissonance that is generally interpreted as dishonest. For example, when someone says "I'm so excited to share.." in a dull and disinterested voice, do you believe them?

Finally, in person presentations are very different from virtual. When presenting to people, there can be an energy feedback loop that you tap into as you respond to the audience responding to your content. In front of the camera, it can feel draining. You can level up speaking to a camera by creating a virtual audience through setting up a side channel with live supporters who you can speak to rather than just a camera.

Most of the time written communication is asynchronous whether it's through proposals, design documentation, code, or reviews. For some communication like chat and messaging it can be either real time or asynchronous. Written communication is a better choice when you want to focus on facts and have less urgency about getting a response. For more complex messages, it may be more meaningful.

Either of these communication methods can benefit from visualizations to complement the words that you use. The specific visualizations you choose are influenced by the type of information that you are sharing and stories you want to leverage. Regardless, both methods require time and effort to get right. You have to reflect on your purpose and ideas before you can convey your message effectively.

Choose your story type

You can use stories to reflect on the past to explain what happened and to look forward to provide direction now. Each type of story reveals information in a slightly different way, and choosing an effective story to present information drives your reader's reaction toward your desired outcome. Some example story types include:

- Factoid

Factoids distill data to interesting data points, highlighting the most common trends, or the noteworthy outliers. An interesting story may drive interest in exploring the rest of the data.

An example of a factoid is the total number of community members using a specific technology, or unique visitors to a website. Factoids are commonly used in dashboards for website stats or product newsletters.

- Interaction

Interactions show relationships between different data sets. Positive correlations between data sets move together: when one set moves up or down, the other trends in the same direction. Negatively correlated sets move in contrast to each other, with one moving down when the other moves up. Identifying a positive or negative relationship is useful, but doesn't explain why data sets move together. Be mindful that correlations may be spurious, where the connection is just a coincidence. An effective story shows the correlation and establishes that the data is meaningfully linked.¹

An example of showing an interaction is having a graph showing MySQL query times and end to end request latency to better observe whether the performance is related to the workload, or if an increase in end-to-end latency is due to a problem in database configuration that has become a bottleneck.

- Change

Change stories are a way to describe how something changes over time. You can use change stories in capacity management and problem detection.

An example of showing change is having a graph that shows the growth of your current used capacity as it approaches the total capacity of your configured system over time. It can show the velocity (change in use from one point of time to another) and acceleration (slope between the lines) to provide how urgent it is to plan or increase capacity.

- Comparison

Comparison stories are a way to show the impact of data that tell different stories. An example of a comparison is showing the different performance characteristics between rolling out a managed relational database from a service provider versus a self managed MySQL instance in a scorecard. It could aggregate important metrics like cost (including the cost of in-house support), performance, scalability, and reliability.

- Personal

Personal stories connect to real-world experience. An example of showing a personal story is an incident summary that contextualizes technical issues with the experiences and choices that individuals made based on their understanding.

Presenting Data in Action

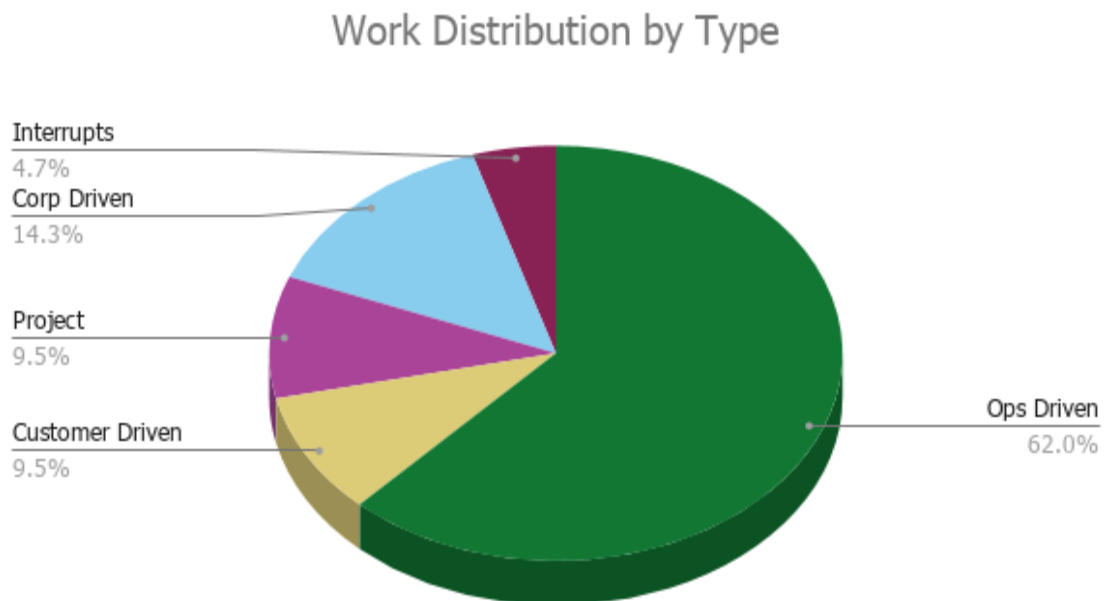
Let me share a couple of scenarios from my own career where presenting data to teams has been useful.

Charts are Worth A Thousand Words.

It was the dreaded quarterly planning time where the team assessed the previous quarter and committed to work in the next. I was new to the team, and I had few expectations. My co-workers expressed frustration because “they never had time to work on team projects to resolve technical debt because of customer interruptions”.

An undisclosed motivation for joining the team was that I had heard that there were challenges with visibility into the work queue and that requests were often delayed or incomplete with no notice. The manager had sought me out explicitly to bring engineering excellence and follow-through execution to the team.

After the planning meeting, I figured out what data to collect around the goals. I worked with the team to categorize the work based on incoming requests and operational debt. I wrote some perl code to query the internal bug API and based on the classification of requests created a few different dashboards to visualize the work. In the next retrospective, I presented a chart like this:



This chart showed that contrary to assumptions, the majority of completed work was driven by the team and not our customers. I could have written up a report, but this simple graphic was easily understood and combined with access to the underlying data, influenced changes in how we prioritized work as a team and led to further improvements for customers in visibility into the work.

Telling the Same Story With a Different Audience

The bigger the impact, the more you need to customize the stories that you tell.

The announcement came out that a number of colocation facilities were to be closed in a few months to cut costs. This meant that our massively distributed database needed to shrink quickly while minimizing impact of latency and otherwise availability to our customers. I needed to think through what actions we could take as a team to limit how normal day to day actions like upgrades to software and onboarding new customers were impacted.

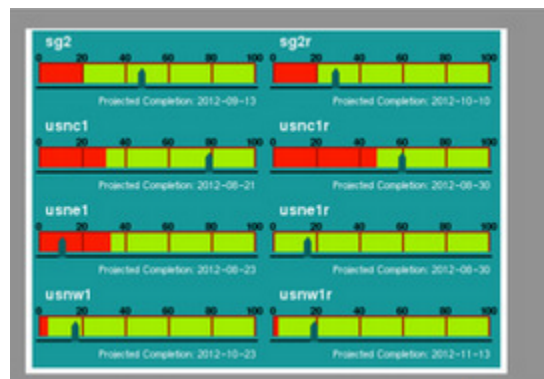
Based on the different timelines for each colocation, I could aggregate where each customer had data and what the best configuration would be to minimize latency impacts in addition to new projects and capacity constraints of the overall system.

I spun up a plan of migrations that balanced out speed, performance, and capacity. I wrote some perl to query the different APIs and javascript to visualize the information.

For the team, I created a table that allowed them to see tasks in progress (P), next prioritized tasks, and work that was complete©. This allowed the ops team to quickly identify whether requests to change a specific table required stopping a task or waiting until the task was complete. Non-impacted table changes could be completed as needed. Additionally, for a region that had work in progress, extra care needed to be taken with upgrades, potentially pausing the migration of data or redirecting customer traffic to the next colo to minimize disruption.

| | REGION1 | REGION2 | REGION3 | REGION4 | REGION5 | REGION6 |
|-------|---------|---------|---------|---------|---------|---------|
| TASK1 | C | C | C | C | C | C |
| TASK2 | - | C | - | P | C | C |
| TASK3 | C | C | C | C | P | C |
| TASK4 | C | - | - | C | C | P |

The table of work in progress showed at a glance which regions were complete “C” and which were in progress. For a region that was complete, upgrades, compute and table deployments could be done with minimal coordination with the individuals working on migrations.



My manager didn’t need to know all the specifics. He just needed to know what work was in progress, were we blocked, and would we finish on time.

For him, I created a set of gauges that showed how far we were and our projected completion for each co-location facility. The projected completion date adjusted each day based on the flow of work. The expected work displayed as a red bar within the gauge based on the planned completion date.

Since this was a long-running project, it provided management with the necessary information required to re-prioritize any work and assign additional interrupt work as they could immediately see the impact. They could then communicate progress to all stakeholders for any other projects.

| | REGION1 | REGION2 | REGION3 | REGION4 | REGION5 |
|--------|---------------|---------------|---------------|---------------|---------------|
| TABLE1 | READY FOR USE | READY FOR USE | May 1 | May 15 | READY FOR USE |
| TABLE2 | | | READY FOR USE | READY FOR USE | |
| TABLE3 | READY FOR USE | READY FOR USE | May 15 | READY FOR USE | |
| TABLE4 | READY FOR USE | READY FOR USE | READY FOR USE | READY FOR USE | |

Finally, every customer had their own set of tables. I provided visualizations that let them know exactly where their data was located, which tables would be updated and when they could expect that the tables would be ready in the new colocation facilities.

The Key Takeaway

Having these different visualizations reduced the number of support and status requests allowing individuals from the team to focus on the work.

Adapt your message based on what your audience needs. Everyone doesn't need all the data collected. Focus your message on the information that matters to the individuals.

Be clear with what data is missing and impacts what individuals can learn from the data that is collected.

Know your visuals

The greatest value of a picture is when it forces us to notice what we never expected to see.

—John W. Tukey

In the previous two scenarios I showed a few ways to visualize data, but there are so many more different visualizations to choose from to transform your data into compelling stories. You can also use design principles to help your audience see what you want them to see.

Visual Cues

Visual cues can help you to display information that others can process without conscious thought. The four basic visual cues are color, form, movement, and spatial position.

1. **Color** You can imply relationships between two different metrics or points in time by varying the hue. You can imply quantity or strength by varying the saturation. You can adjust the temperature, or the perceived warmth or coolness of a color to focus attention. Warmer colors tend to advance into the foreground while cooler colors fade into the background. Be mindful that color should be used to enhance the conveyance of information but that new information shouldn't be expressed solely through the use of different colors.
2. **Form** You can change length, width, orientation, size, and shape.
3. **Movement** Flicker and motion can call attention to specific areas of importance but can be distracting or annoying. You can also imply motion through the other visual properties rather than using motion directly.
4. **Position** You can use a 2-D position and spatial grouping.

Sometimes cues are not appropriate, if they mislead or hinder your audience's interpretation of your visualizations. For example, don't use different sized circles for categorical data if the magnitude difference of the categories aren't important.

TIP

Learn more about design principles from Robin Williams' The Non-Designer's Design Book.

Chart types

You can use different charts to visualize data. Some examples include:

- **Data Tables** organize data into rows and columns.

Tables can be a valuable tool to:

- Plan such as itemizing a list of requirements for a proposal, brainstorming quarterly projects and elaborating on details that apply to each identified element such as proposer or length of time.
- Document for example to lay out a list of options or provide comparisons between different tools and services.
- Define top lists where you want to provide a quick periodic review for tactical direction. Examples include top pages or sources for websites.
- Explore large sets to filter, display data and drill down into individual queries.

Tables can be an overwhelming way to present a large volume of data, so it is a good habit to complement tables with other visualisations that can draw attention to trends, outliers, and other patterns in the raw table data. Take a look at this example:

*T
a
b
l
e*

*l
l
-
l
.
A
m
a
z
o
n*

*D
y
n
a
m
o
D
B*

*T
h
r
o
u
g
h*

*p
u
t
L
i
m
i
t
s
i
n

T
a
b
l
e

f
o
r
m
a
t
a*

| | On-Demand | Provisioned |
|--|--|--|
| Per table | 40K read request units and 40K write request units | 40K read request units and 40K write request units |
| Per account | Not applicable | 80K read capacity units and 80K write capacity units |
| Minimum throughput for any table or global secondary index | Not applicable | one read capacity unit and one write capacity unit |

a “Service, Account, and Table Quotas in Amazon DynamoDB”, Amazon, last modified December 15, 2020,
<https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Limits.html#default-limits-throughput-capacity-modes>

Here, the table format is used in documentation to illustrate a comparison between the on-demand and provisioned Amazon DynamoDB throughput limits. The format works because there isn’t a lot of data, and it’s clear what is different.

| Timestamp <small>UTC</small> | cache_status | client_ip_hash | content_type | geo_c: |
|------------------------------|--------------|--|--|--------|
| 2021-01-18 21:15:25 | HIT | 0a8b1069f217aed3f6f3307136f2e0a59d351ded15f339428af10cd011fb0a52 | binary/octet-stream | cambr: |
| 2021-01-18 21:15:25 | HIT | dbdb4a3a06785feaa5e29dd1bf2466bb187ff3f967ddfe0e6027a5e5c92bf78e | application/octet-stream | ashbu |
| 2021-01-18 21:15:25 | HIT | 4dbc8a020963bf0553e087fea64e85c875b4cdf1156e38141c27d67379629a40 | application/octet-stream; charset=utf-8 | louis |
| 2021-01-18 21:15:25 | HIT | 769a0fe4eb80fa5e2685ea60ac6a061ef8717eaa543fe17e3029113ba7c8dd12 | text/plain; charset=utf-8 | ashbu |
| 2021-01-18 21:15:25 | HIT | cbf0ab785c36706715ff57c25996a39327f49bf2add13d78a99030070f45ebef | application/octet-stream; charset=utf-8 | dubli |
| 2021-01-18 21:15:25 | HIT | afca926e07a90d729928645dbfb566dda787599478a6b248c4c1ea506750c1c1 | binary/octet-stream | london |
| 2021-01-18 21:15:25 | HIT | 61cba6bf0089bcd8273eb5ca78245f636171ba83948cb30795b17130ed2aa962 | application/octet-stream; charset=utf-8 | san j |
| 2021-01-18 21:15:25 | HIT | 794e7164d2f423d2a10173106e2b9c87601c2bea9b91fb78a1209c81881fdca5 | text/html | pavas |

Figure 11-1. Rubygems.org Raw Data in Table format²

In this example from the [Honeycomb play with live Rubygems.org data playground](#), a customized table visualization applies visual cues to the raw event logs in the data table. Rows have alternating colors to make it easier to read the table.

- **Bar charts**

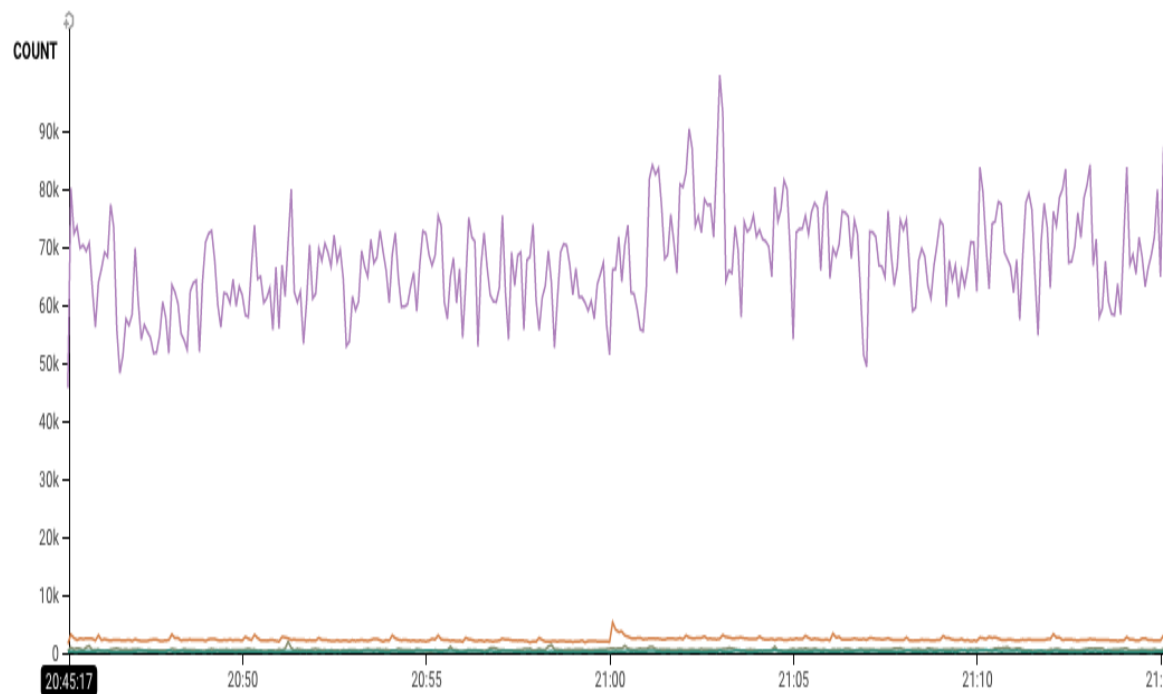
Bar charts are useful for quantified categories of data that you want to compare when you have more than 2 or 3 categories. When you have long category names, use a horizontal bar chart.





I've used them as ways to visualize system audits across multiple co-los to see the number of nodes running out of date operating systems.

- **Line charts**

Line charts plot changes in value and show patterns over time or relationships between two variables. Additional lines can be added to the chart to show trends between series. These are often the go-to for showing time-based trends, as well as differences between series.

Often the vertical axis will represent a statistic like the count, sum or average of a measured attribute across a dataset. On the horizontal axis a continuous interval like time is used.



| | cache_status | COUNT |
|---|--------------|------------|
|  | HIT | 24,365,134 |
|  | ERROR | 847,957 |
|  | MISS | 206,781 |
|  | PASS | 96,952 |

elapsed query time: 668.322622ms rows examined: 41,000,000 nodes reporting: 100%

Figure 11-2. Rubygems.org results in line chart format³

In this example from the [Honeycomb play with live Rubygems.org data playground](#), the raw data from the table before is visualized showing the cache hits, misses, errors, and passes over time.

- **Area charts**

Area charts are based on line charts and show quantitative data over time. Stacked area charts are useful to show part of the whole or cumulative values.

- **Heat maps**

Heat maps show data patterns through shading or color. One of the challenges of these kinds of graphs is making sure that the color schemes are accessible and don't create artificial gradients. Heat maps can also be problematic when there isn't a discernible pattern that can hinder comprehension.

- **Flame graphs**

Flame graphs are a way to visualize profiled software and are helpful in debugging problems of resource exhaustion.

ADDITIONAL RESOURCES FOR CHART VISUALIZATIONS

Learn more about the visualization of information from Edward Tufte's books [The Visual Display of Quantitative Information](#), [Envisioning Information](#), and [Visual Explanations](#).

Learn about other charts from [AnyChart's "Chart Type: Chartopedia."](#)

Learn more about [Flame graphs](#) from the inventor Brendan Gregg.

Recommended Visualization Practices

In presenting information you control the narrative and provide a way to interpret the data. Contemporary tools allow us to explore the data available to us and interact, verify, or provide alternative narratives to explain what is happening.

Imagine you manage a cluster of load-balanced web servers. You might have a line chart of total errors with a different color line per server. Multiple lines can be visibly noisy but quickly show outliers in error types.

You might also have a graph per server that shows different shapes per error type. Different shapes show at a glance when a particular server was serving more errors, and whether the errors are associated with a particular type of error.

Apply these recommended practices when presenting visualizations:

- Distill to your key points. Don't rely on text alone. Choose the right visualizations to support your key points.
- Use consistent colors in a dashboard with multiple charts and within a chart. Color directs focus. Lower the saturation for supporting or less important data. Limit the number of different colors in use.
- Graphs should always have labeled axes and a legend. Eliminate duplicate information within the graph, though. For example, if you are using bar charts and have labeled the categories, then a legend isn't useful.
- Include references to the sources of data. If something looks off about the chart, people can go back to the data to verify and dig deeper if needed.
- Design for the format. For presentations, lots of words will be hard to read and might obscure the most important message. For an on-call dashboard, more detail that provides clear and specific steps to take will be appreciated for those 2am pages.
- When visualizing a specific dataset, point out key observations using annotations and highlighting.
- Construct dashboards in a way that charts can explain each step of discovery. This is especially helpful if you need to rely on those dashboard for middle of the night on-call support.

TIP

See different visualizations of one dataset and how they change the message with [Nathan Yau's "One Dataset, Visualized 25 Ways" on Flowing Data](#).

-
- 1 “Beware Spurious Correlations,” Harvard Business Review, June 2015,
<https://hbr.org/2015/06/beware-spurious-correlations>
 - 2 “Honeycomb’s Play with Live Rubygems.org” Honeycomb, honeycomb.io/play.
 - 3 “Honeycomb’s Play with Live Rubygems.org” Honeycomb, honeycomb.io/play.

Chapter 12. Building Resilient On-Call Teams

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 19th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at vwilson@oreilly.com.

The most critical responsibility of the production environment is on-call and the management of events that impact customers. As an organization scales, often on-call and incident management become a team’s breaking point because not enough investment is put into the production experience early enough in the software development lifecycle. It may not make sense to spend money on operational costs when the service isn’t in operation. Yet, over time, this mode of spending becomes ingrained behavior. Burdened with the reactive work of responding to pages, you don’t have time or energy to repair underlying infrastructure, software, or service problems effectively. Even when you are not on-call, you may find yourself trying to focus on project work while also executing on preventing larger impacts or preparing yourself to be interrupted by an incident.

In this chapter, I examine on-call processes that redistribute the burden of production support from single system administrators to the different

individuals and teams that are responsible for the quality and success of the product or service to create more resilient on-call teams.

What Is On-Call?

On-call is a temporary rotating role assignment that may include being reachable outside of normal business hours (e.g. evenings, weekends, and holidays) to answer requests for support and handle discovered alerts.

When you are on-call, you are one of the people responsible for this work that comes in for a specific length of time. Depending on the size and how distributed the team is, on-call rotations may consist of 8 to 24 hour shifts for one to two weeks.

Historically, sysadmins were assigned a pager and an on-call rotation. On-call work varied widely from failed application services to power outages. Often on-call and interrupt-driven work tended to merge into a single work queue.

Misalignment in severity and priority assessment exacerbated the complexity of on-call. Requestors failed to assess the severity, the pain level of a problem. Severity set to high led to time spent on low impact issues; too low of an assessment increased customer impact.

System administration teams had difficulty assigning priority whether due to having all interrupts assessed at a high priority, or failing to rank incoming issues and modify existing issues.

Many contributing factors lead to unsustainable on-call practices that often push individuals into jobs without growth. Many sysadmin jobs are still associated with this reactive nature of work rather than focusing on more proactive engineering work.

Modern teams improve their resilience through adding developers to the on-call rotation and updating on-call processes.

Adding Developers to the On-call Rotation

In environments where the systems being managed are built in-house, the developers of the software serve in primary on-call roles. Some individuals are unhappy with this change because they view on-call as being something *less* and that people with *less* skills should do on-call. This is shortsighted as on-call provides direct insight about the value of the software in use and commitments made to customers.

The people who build the product have the most understanding and context about how the product should work, but often it's the system administrators who have built up the knowledge of how the product actually works in a production environment. Having developers respond to first level alerts ensures that the people building the product have insight into the most painful parts of the system.

Additionally, working on-call informs developers on how to prioritize work when refactoring, creating something new or removing functionality. Requests into the operations team queue from development teams will have better severity assessments based on this information.

Some organizations embraced adding developers to on-call rotations and started to advocate for the elimination of operations teams. A “no ops” team can be problematic because many developers don't have operability skills. This isn't a problem with the individuals; there is always a tradeoff in skill specialization.

While devs may be on-call for a specific component, sysadmins are usually the ones with knowledge about all of the different systems interoperating. Operations teams play a part in on-call, whether through primary rotations or providing second or third-tier support. Beyond first-tier support, system administrators assist product integration with other dependencies or standard operability like backups and recovery, for example.

Sysadmins are often the ones with administrator visibility across system resources with cloud providers and third party services and have insight in to the ways that production systems can vary from instances that other engineers may develop and test their features.

Updating On-Call Processes

High-stress on-call without relief is harmful to mental health and can lead to anxiety, depression, burnout, as well as other issues. Also, individuals may ruminate about production in unproductive ways shifting focus away from the creativity required to work on complex projects and develop plans.

Teams that apply continuous learning to on-call can help lower the risks of this harm to the individual as well as improve the resilience of the team to respond to outages. If on-call is sustainable to the engineers responding, then everyone benefits from having the mental space to incubate ideas because that worry about larger impacts or the fear of interrupts is removed. Update the on-call process through:

- Monitoring the on-call experience. Lots of pages and long-running events both tire out on-call engineers. When folks are tired, they are more likely to make mistakes. Monitoring the on-call experience improves the general health of individuals and the team as a whole by ensuring tired folks don't stay on-call and encourages a culture of support and care.
- Embracing the whole-team approach and encouraging folks to reach out for help to increase the overall resilience of the team. Planned escalations reduce the stress of the unknown.
- Monitoring and maintaining alerts. Noisy alerts are frustrating and reduce vigilance, increasing the risk of mistakes. If you make use of a service level objective handbook, make sure that documentation is updated to reflect changes to the service level indicators - including why the change is being made.
- Establishing incident protocols. Not every event is something that needs to be measured. Not every measured event should be an alarm that pages individuals. Not every page is an incident. Establishing explicit, clear protocols helps reduce fatigue and promotes alert maintenance. It also provides people with a process to follow for those business-critical pages at 2 am.

Monitor the On-Call Experience

First and most important monitor the work in progress, even if you're the solo on-call engineer. Ideally, on-call work should come into a work queue that is shared by all engineers. It's hard to see the impact of a specific change without establishing baselines and measuring the change.

Here are a few questions to think about and consider monitoring in your environment:

- How often does an alert page?
- How often is it actionable? Does the alert self resolve?
- When was the alert last updated?
- When was documentation last updated?
- What is the impact of the failed system? Does it need to alert outside of hours?
- How much coverage is available? If an individual is paged out and resolving an issue, who takes the next page?
- How often does the person on-call get diverted from normal life activities including: sleep, meals, and showers?
- How often are family gatherings and obligations interrupted?
There are many activities that can't be rescheduled and are critical to healthy relationships.

Rather than just focusing on system time to recovery and time to discovery, these metrics help to classify and direct improvement in the on-call experience. During production meetings, talk about these metrics and ensure that necessary actions items are logged to improve observed trends.

Conduct quarterly retrospectives to reflect on progress and update paging schedule and escalation policies based on feedback and potential improvements.

Adopt a Whole Team(s) Approach

There is no way to know everything. Consider a whole team approach to maximize the comprehension of complex environments and minimize customer impact when perfect systems are inaccessible and costly. Ask for help, and talk about issues as they arise. Encouraging a question culture leads to increased shared understanding as assumptions about work aren't made in isolation.

It can be hard to change a team dynamic where folks hold assumptions about on-call work being the responsibility of a single individual who must be held accountable. Incentives and rewards need to align with the importance of the customer-impacting environment. Hold the reliability and robustness of the live system in as high esteem as feature work.

TIP

Check out this [case study on changing the on-call practice at LinkedIn](#).

On-Call in Practice

With a whole team approach, what do the responsibilities of on-call as an individual look like to you?

Preparing for On-Call

During the weeks leading up to your first on-call shift at a new job, make sure that you know about all the systems you're responsible for and the escalation path. Whether a formal process or not, participate with other on-call engineers, also known as shadowing on-call. Shadowing allows you to see tools and processes in use, examples of how to respond and interact with the team, and assess the cost of the on-call experience to the individual.

Make sure that your laptop and phone are charged and up-to-date with software requirements, and that you can access the services you need to from home, and your favorite coffee shop (or soccer field, bike path, or other area you may visit during your on-call shift).

Configure your phone and other devices in your alerting service. Services have different escalation policy customizations, so make sure to enable more than just email.

Everyone has their preferences to how they like to get informed about events, and teams have a specific expected response time. I am super focused on alerts when I'm on-call, so my preference is to minimize distraction and enable redundancy. For an expected response time of 15 minutes, I prefer email and SMS immediately followed by a phone call if I haven't responded in 10 minutes. This configuration gives me 10 minutes to respond to the SMS before I get another alert reducing potential duplicate alerts.

It's important to take in to account the requirements of the on-call rotation and response time, and your way of working. Find the balance of being responsive while not getting frustrated by noisy notifications.

Expense additional charging cables for all your devices to help eliminate the dreaded "did I leave that cable." At a minimum, pack an extra laptop and phone power cord in your on-call bag. Battery packs can enable remote access via phone or tablets by providing additional power.

While you may not make and receive phone calls regularly, be prepared to have voice or video conferences during your on-call rotation with a handsfree headset.

A Mobile HotSpot or Wi-Fi tethering devices is critical to support sustainable on-call rotations. While many phones have tethering capabilities now, these devices have better reception. It also provides redundancy if the phone signal is weak and allows the phone to be used to further alert on other issues or dial into conferences as needed.

Another advantage of a separate cellular data device may be increasing the diversity of connection options — if your phone has service from one provider and the device gets service from another, you're more likely to have access to a viable signal, even if the location you're in happens to be a dead spot for one provider.

A local team can share devices and hand off as needed. Pre-paid and no service contract devices are quite affordable, so this is still affordable for distributed teams.

One Week Out

The week before your on-call shift, make sure that you **notify your project teams and update associated project tickets to call out that you're going on-call**. On-call work is critical and shouldn't be considered a distraction from "real work." By updating the project tracking system with information about your upcoming on-call, you are minimizing the unplanned stress folks might have about specific work. Hopefully, proactive updates also reduce the project work interrupting on-call. If there are critical time-bound tasks, delegate the work.

If possible, **send test alerts** to confirm that you're enabled to receive alerts. Even if you have checked for past rotations, confirm configuration changes haven't eliminated your notifications.

Plan your snacks and meals ahead of time. Self-care is especially critical during an on-call shift. When and how often you'll get paged is an unknown. While you can estimate what will happen based on past performance, it's not a guarantee. For the things that you can plan, this will help eliminate additional stressors when cascading failures occur.

Plan for any additional coverage. Do you have a long commute or a regularly scheduled doctor's appointment? Do you need to drop your kids off to daycare or attend a soccer game? Do you need to take your pet to the vet? Talk to the secondary or ideally another engineer that can provide coverage. Configure these overrides in advance. On-call rotations need to factor in the real demands of personal life responsibilities and be flexible. A team that already practices this will be more able to handle short term additional demands from outages.

Connect with the rest of the on-call team Ideally, there is a secondary, other escalation points of contact, and an incident manager. The point of this step is to give you additional confirmation that everyone is ready for your participation in on-call.

TIP

While you don't have a responsibility to reach out to everyone on-call, doing so helps build and sustain the meaningful connections for successful, minimum drama rotations. There have been a few times where I've discovered that folks had planned a vacation, and this helped prevent holes in coverage.

It's also helpful when the on-call team is a virtual team comprised of folks from different roles who may not have an awareness of the different skills that the individuals bring to the on-call rotation.

Connect to specialized engineers. While there might not be an official on-call if there are single points of responsibility within the organization, it's essential to have contact details for your security, network, or database engineer. If they are not part of the on-call rotation, identify under what conditions they should be notified as an escalation point of contact.

Talk to your family or roommates about upcoming on-call. Set the expectations around what an event looks like and the expectations they may have of you. Set boundaries around acceptable behaviors (e.g. no hosting parties on your on-call weekends)

Preparatory work is necessary for going on-call. Make sure that time-allocated for the week doesn't focus on a project's progress to the detriment of on-call preparation.

Night Before

Verify that your notification device is charged and not silenced or in do-not-disturb mode. Get enough sleep; restedness is a crucial component to being able to sustain alertness to a changing environment. If you're fatigued going into an on-call rotation, it will hinder your effectiveness at sustained attention.

Your On-Call Rotation

Throughout your on-call rotation, the overall process may vary based on your teams' expectations, but a general approach includes:

- Receive Alert(s)
 - Acknowledge the alert(s)
 - Triage
 - Fix
- Improve On-Call Experience
 - Documentation
 - Monitoring
 - Assessing normal

When you receive an alert, the first action is to acknowledge the page. An acknowledgment lets folks know that you have received the alert and will help minimize further interrupts for the same issue.

Next, triage or assess the severity and urgency of the problem and based on these factors, route the alert to the appropriate action.

Finally, fix the problem that is being alerted. Fixing includes adjusting a noisy alert that pages with no expected action.

Assess your on-call readiness. High impact lengthy incidents and numerous frequent alerts are both concerning. It may be better for you, and the team to hand off primary on-call to someone while you take a break.

Assessing on-call readiness needs to be formalized within the team's processes. A few examples of what that would look like:

- If a team member gets a page after standard working hours that takes over an hour to resolve, then the team member automatically is granted that additional time to come in the next morning.
- If an incident takes more than 8 hours to resolve, then the team member automatically gets the next day off.

Having explicit policies helps increase team resilience as individuals are more willing to be a member of the on-call rotation which helps build the layers of redundancy required so that people can cover for each other when someone needs to take a break.

During the typical on-call workday, when not receiving an alert, the focus is on improving the on-call experience (versus working on project work). Workday tasks could be improving documentation or monitoring, or in learning more about what “normal” behavior looks like in your systems. Sometimes in the process of examining the live system, you'll discover something that is impacting and requires fixing. Make sure that these discoveries are documented (in the work queue as well as the on-call handbook) and alerts configured.

Week After

There should be an official handoff meeting that informs incoming engineers about the past week's issues. This handoff meeting marks the

point when you are off-call.

A few suggestions for a template for this weekly review include:

- Time period
- Individuals who made up the on-call team for the time period
- Incidents and relevant links to more information about those incidents
- Open Incidents
- Resolved Incidents
- Incidents that were not captured by alerts
- Manual work
- Opportunities for automation and improvement
- Open questions; While there may no longer be an impact to consumers, there might still be unanswered questions.
- Call outs for specific items that went well, and what needs improvement.

Breathe, and you're not done doing the on-call work. While the events are fresh, revisit the issues you filed. This is the best time to have those creative ephiphanies to improve what you just experienced. Update necessary documentation, clean up any noisy alerts (which includes reducing the severity of alerts as appropriate) and record any project related work required for long term improvements. For any incidents, add relevant information to the incident report.

TIP

One way to help continuously improve on-call alerts is to have a regular alert review with your team to talk through the impacts and values of the alerts.

Wrapping Up

As environments become more complex, and we recognize the diversity of needs individuals have, on-call work must become more sustainable to promote resilient teams. Companies who build sustainable on-call will have a competitive advantage over those who don't.

Scheduling right now is often performed by humans which can lead to perceived unfair, unbalanced schedules. Being able to plan and add individual needs proactively can promote better, more flexible schedules that work with humans.

TIP

For more resources about on-call, check out:

- [Crafting sustainable on-call rotations by Ryn Daniels](#)
- [The On-Call Handbook by Alice Goldfuss and contributors](#)
- The Notifications chapter in *The Art of Monitoring* by James Turnbull

Chapter 13. Managing Incidents

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 20th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at vwilson@oreilly.com.

As much as an operations team may strive to reduce risk, failure will happen. What happens when a problem in your infrastructure turns into a more significant issue that impacts your customers?

Early system administration practices focused on metrics like mean time between failures (MTBF), mean time to failure (MTTF), mean time to detection (MTTD), and mean time to recovery (MTTR). These metrics were useful when reading hardware specifications to schedule optimum proactive replacements to avoid outages. These metrics are much less valuable when it comes to modern cloud-centric systems, because their focus on predicting hardware failure trends no longer apply, now that the focus has shifted from physical servers to virtualized compute.

Additionally, averaging response times for different times of failures isn’t providing useful information. Instead of these metrics, measure how the team is encouraging continuous collaborative learning through the narrative story within incident reports.

In this chapter, I'll examine the specific collaborative processes of responding to an incident, identifying issues through the post-incident review, and repairing or refactoring identified issues.

What is an Incident?

For some environments, an incident may be anything that pages the on-call engineer. In other environments, incidents are security breaches. In this book, an incident is an exception to a live site, service, or software application that has an impact.

Let's break this down into the components starting with exception. Exceptions occur when the system doesn't behave in an expected way. Exceptions can be bugs in the code, failures in underlying systems (like DNS or the network), or misunderstanding in the project planning that led to a different implementation.

A live site, service, or application is something that is in use by clients or customers. In many cases, this is the production environment for a site or service but also includes applications installed on devices.

Impact is the qualitative effect that the exception has on the clients or customers. Sometimes, this impact may be visible externally. Other times, a decision needs to be made about whether to disclose the incident or not. Incidents may have varying degrees of external impact or be near misses that the customer has not yet observed.

Some examples of public incidents:

- In July 2020, an expired server certificate and a data outage prevented the California Reportable Disease Information Exchange from accepting COVID lab results from external partners leading to discrepancies and under reporting of case information.
- In October 2019, PG&E had a website outage due to a high volume of traffic from individuals looking up information related to the planned power shutoff.

- In October 2019, Docker experienced an incident where the Docker Hub registry was down. Any organization that relied on directly pulling images from the registry would have experienced issues that relied on these images being available. Organizations that cached docker images or hosted their own registry would have minimized their impact.
- In May 2019, **Slack** started a deploy of a feature that prevented some customers from connecting to and using Slack. For organizations that were impacted, this was a complete outage.

What is Incident Management?

Incident management is the process of responding to an incident to reduce damage, costs, and recovery time, identifying code or process issues and repairing issues to prevent repeat incidents. The common shared basic principles for each step in incident management is clearly defined roles and responsibilities, and continuous collaborative learning.

TEAM RED FLAGS

Every position has different aspects; some are enjoyable, and some are not. But beyond the typical highs and lows of the job are some warnings signs that the position itself has limited growth opportunities. A few of these signs are:

- Lack of transparency around failure,
- Blame and fear culture where folks are afraid to talk about mistakes,
- Repetitive incidents without improvement or long term correction.

There are other problematic issues, but these are especially harmful as they hinder learning, disrupt trust and relationship building, and promote burnout which can compound the impact of incidents.

Roles and Responsibilities

Incident response teams vary across organizations. Your incident response team may have different names for specific roles and more or less differentiation. Significant functions that should exist in some capacity (whether they have these names or not) are the incident commander, subject matter expert, liaison, and note taker.

- The incident commander (IC) is responsible for driving an incident to resolution and the post-incident meeting. During an incident, there is always a single acting incident commander. The responsibility may be passed from one individual to another throughout the resolution of the incident.
- The subject matter expert (SME) is the on-call engineer or the designated owner for a particular part of the service. There may be

a number of subject matter experts required to resolve a specific incident.

- The liaison is responsible for communicating internally and externally about the status of a current incident. There may be multiple liaisons for handling the different messaging internally and externally for a specific incident.
- The note taker takes notes, filling in details about the important actions and followups that occur during the incident. This might be done through the use of software that responds to special commands or chatbot. These notes are critical for providing the context for the narrative that will drive learning for the incident.

Pre-emptive Planning

- **Set up and document communication channels.**

During an incident, a team shouldn't be trying to figure out the process for how everyone will communicate. Plan for a distributed team where individuals might not be in the same place or even time-zone.

Some teams create an #oncall channel in Slack where all incidents are discussed. When an incident is identified, a new channel is created in the format of #incident_NUMBER. This keeps the #oncall channel usable for other problems that may occur without impacting the discussion that needs to occur to resolve an ongoing incident.

- **Train communication.** Being explicit about the expectations around communication during an incident reduces mistakes and time to resolution.
- **Create templates for retrospectives.** Templates for the incident meeting reports set expectations and standards for these reports.
- **Maintain documentation.** On-call and incident handling documentation should be reviewed and updated regularly. Stale

documentation that doesn't reflect the processes in use hinder organizational learning as well as frustrate engineers. This includes specific processes that might be defined for how to handle an alert that signals a problem and any disaster recovery plans.

Handling the Incident

As an on-call responder, when an alert first comes in and is triaged, you may assess it as an incident. You're the acting incident commander until you've explicitly handed off the responsibility to someone else.

The more severe the incident, the more important to have different people handling the fundamental roles of incident commander, subject matter experts, and liasons. It's important to separate these different roles so that the subject matter experts can focus their attention on identifying and repairing the problem.

Incident response cycles through the following processes:



Figure 13-1. Incident Response Lifecycle

- **Assess** The IC assesses the incident through the observed symptoms, scope of the problem, and potential risks based on the symptoms.
- **Act**
 1. **Identify possible actions and associated risks.**

2. **Make a decision.** The IC says the decision out loud if on a call and in channel if on a chat platform.
3. **Obtain consensus.** The IC asks whether there are strong objections to the decision. They should adjust actions based on feedback, but ultimately the IC makes the final determination.
4. **Delegate stabilization actions.** The IC delegates the stabilization actions. Assignments must be clear and specific with explicit timing information.

Individuals involved in handling the incident may be unable to do newly identified actions because they are working on a previously identified stabilization action or lack sufficient knowledge or experience.

Assignments should be adjusted based on feedback and required timelines. This could be a good time for an individual to learn how to complete a particular action with guidance. If there isn't sufficient time or too many tasks to complete, the IC should assign the task to an experienced available contributor. Depending on the severity of the incident, this may require pulling people on to the incident response team to complete the required tasks in a timely fashion.

- **Inform.** Depending on the size of the team handling an incident, the IC may name an explicit liaison to handle updates. Liaisons shouldn't be the subject matter experts who are actively investigating and repairing as shifting contexts can exacerbate stress and increase mistakes. When the live site is in a degraded state, clear, timely communication to customers requires skill. Poorly worded explanations can cause more problems than the actual outage.

1. **The IC has the liaison(s) send regular updates** to the team, customers, and executives. The frequency and content of the communications will vary by audience. Updates should include what is happening and the steps taken.
2. As an individual's expertise is no longer needed, the IC **reduces the scope of the incident**. The IC informs the incident response team who is still required to resolve the incident and encourages folks who are no longer needed to take a break.

- **Verify.**

1. The IC checks that the subject matter experts completed stabilization actions.
2. The IC checks the outcomes of those actions. If there is a continued impact, they repeat this action loop starting from accessing the incident.

TIP

Having participated in way too many incidents, I've seen pizza as the designated emergency meal way too often. As a tasty treat, pizza is great. As a "fuel the brain and body for sustained stress," not so much. Incident teams should have a plan for obtaining inclusive snacks and meals. For distributed teams, it's critical to have contact information for the on-call engineers and delivery options or having a policy that grants a stipend for folks to expense a meal.

People forget to eat when focused on repair and recovery. This exacerbates the fatigue that comes from sustained attention to a specific problem. Team leadership should make part of the assessment process a check-in with the humans that are part of the system.

Post-incident meeting

The post-incident meeting is a critical part of continuous learning in an organization. The IC schedules this meeting shortly after the resolution of

the incident giving individuals the time to prepare. The requirements for this meeting include:

- **Collate a record of the incident**

Once an incident has occurred, collate information from all the participants. The goal of this is not to place blame, but to uncover what happened and drive conversations during the post-incident meeting.

One way to help prevent blame is to make sure that the focus is on what happened and what people decided to do based on that information rather than trying to talk about what should have happened, or could have been done. State what happened, without side commentary and second-guessing.

The should of and could of's can derail learning about what was done. This doesn't mean don't acknowledge mistakes. Mistakes need to be talked about and understood. It might be a case where something can be improved about the system or someone may not understand the importance and impact of their actions.

- **Share objectives of the post-incident meeting.**

Everyone heading to this meeting should have shared objectives to help align efforts. A post-incident meeting without shared objectives is often worse than no meeting at all. If there are misaligned incentives or individuals are not getting recognized for the value they bring to the process, this can lead to heroics or dismissal of the whole process.

Objectives shouldn't reflect an idealistic "perfect" world. For example, there is no way to prevent all incidents from ever occurring so having an objective to eliminate incidents isn't reasonable.

Instead, aim not to repeat the same incident in the same way. Another helpful objective might include identifying areas where information about why something occurred isn't understood clearly and where single individuals knew specific information that wasn't known to the entire team.

In other words, the outcome of this meeting should increase knowledge and identify areas of focus.

- **Preparation for the meeting**

Everyone should review the record of the incident and add information that might be missing including areas where they might have been confused or uncertain about next steps.

- **Create artifacts.**

I've often seen a single artifact as the outcome of an incident and the following post-mortem. This artifact felt very much like a way to direct blame. This helps to ingrain fear as a driving focus and hinders collaboration.

A lot of data is generated, many graphs are examined, and many people may have been involved in getting the service back into a healthy state. Sift through all this information and compose the necessary artifacts.

A CEO who needs to respond to an interview, customers reading about the impact, and internal team members all need different artifacts.

EFFECTIVE TEAM INCIDENT REPORTS

It's essential to have a team incident report per incident. These artifacts are for the team to help to spread knowledge and prevent stagnation where the team as a whole doesn't know specific knowledge obtained via the incident. Store these artifacts in a central place. Depending on the organization, these artifacts may be useful to other teams.

Each artifact may have slightly different content based on the nature of the incident. The reader of the team incident report is the individuals on the team, so these reports can be longer and more detailed than external or executive briefings.

An example template for a team incident report:

- Title
- Date
- Author(s)
- Summary of the incident
- Incident participants and their role(s)
- Impact
- Timeline
- Include graphs and logs that help support the facts described in the timeline.
- Lessons learned about what went well, and what needs improvement.
- Action Items - These should include who, what, type of action, and when. Others outside of the incident response team might think of additional action items after reviewing the narrative.

While one person should have the responsibility of being the note taker during the meeting and initial author of the report, everyone on the incident response team should review and update the team report ahead of the post-incident review.

TIP

Learn more about Post-Incident Reviews from [<https://victorops.com/ebooks/oreilly-post-incident-review/>](Post-Incident Reviews) by Jason Hand.

Practice Failure

Incidents will happen. Teams should exercise the process for handling incidents testing various scenarios that might occur. Much like testing in development, practiced failure is a very different experience than a live incident, but it does have value. It can help expose gaps in documentation and process that are better understood before you have to respond to an event at 2 am or deal with differences in skill gaps.

About the Author

Jennifer Davis is an experienced operations engineer, international speaker, and author. Her books include *Modern System Administration*, *Effective DevOps*, and *Collaborating in DevOps Culture*. Jennifer has worked with a variety of companies, from startups to large enterprises, improving operability practices and encouraging sustainable work.

About the Contributors

Tabitha Sable has been a hacker and sysadmin since the turn of the century. She loves to build tools and make friends, and puts those skills to work coordinating the efforts of infrastructure, security, and product teams. She currently serves Kubernetes as co-chair of SIG Security and an associate member of the Product Security Committee. Outside of work, she can often be found organizing or competing in Capture the Flag contests and loves “pretty much anything with wheels.”

Chris Devers has spent the last twenty years helping people get the most out of computers, so that they can spend their time on more important things, helping development teams focus their efforts on delivering software that solves real problems for real people. He lives in Somerville, Massachusetts with his wife, sons, and cat, and would usually rather be taking photos and bike rides.

1. I. Foundations

2. 1. Introduction

- a. Principles
- b. Modernization of Compute, Network and Storage
 - i. Compute
 - ii. Network
 - iii. Storage
- c. Infrastructure Management
- d. Scaling Production Readiness
- e. A Role by any Other Name
 - i. DevOps
 - ii. Site Reliability Engineering (SRE)
 - iii. How do Devops and SRE Differ?
 - iv. System Administrator
- f. Finding Your Next Opportunity

3. 2. Infrastructure Strategy

- a. Understanding Infrastructure Lifecycle
 - i. Lifecycle of Physical Hardware
 - ii. Lifecycle of Cloud Services
 - iii. Challenges to Planning Infrastructure Strategy
- b. Infrastructure Stacks
- c. Infrastructure as Code

- d. Wrapping Up

- 4. II. Principles

- 5. 3. Version Control

- a. Fundamentals of Git

- i. Branching

- b. Working with Remote Git Repositories

- c. Resolving Conflicts

- d. Fixing Your Local Repository

- e. Advancing Collaboration with Version Control

- f. Wrapping Up

- 6. 4. Local Development Environments

- a. Choosing an Editor

- i. Minimizing required mouse usage

- ii. Integrated Static Code Analysis

- iii. Easing editing through auto completion

- iv. Indenting code to match team conventions

- v. Collaborating while editing

- vi. Integrating workflow with git

- vii. Extending the development environment

- b. Selecting Languages to Install

- c. Installing and Configuring Applications

- d. Wrapping Up

7. 5. Testing

- a. Why should Sysadmins Write Tests?
- b. Differentiating the Types of Testing
 - i. Linting
 - ii. Unit Tests
 - iii. Integration Tests
 - iv. End-to-End Tests
- c. Examining the Shape of Testing Strategy
- d. Existing Sysadmin Testing Activities
- e. When Tests Fail
 - i. Environment Problem
 - ii. Flawed Test Logic
 - iii. Assumptions Changed
 - iv. Code Defects
 - v. Failures in Test Strategy
- f. Flaky Tests
- g. Wrapping Up

8. 6. Security

- a. Collaboration in Security
- b. Borrow the Attacker Lens
- c. Design for Security Operability
 - i. Qualifying Issues

- d. Wrapping Up

- 9. III. Principles in Practice

- 10. 7. Infracode

- a. Building Machine Images

- i. Building with Packer

- ii. Building With Docker

- b. Provisioning Infrastructure Resources

- c. Provisioning with Terraform

- d. Configuring Infrastructure Resources

- i. Configuring with Chef

- e. Getting Started with Infracode

- f. Wrapping Up

- 11. 8. Testing in Practice

- a. Writing Unit Tests for Infracode

- i. Writing Unit Tests with Chfspec

- ii. Writing Unit Tests for Datadog Install Recipe

- b. Writing Integration Tests for Infracode

- i. Writing Integration Tests for Datadog Install Recipe

- ii. Linting Chef Code with Rubocop and Foodcritic

- c. Wrapping Up

- 12. 9. Security and Infracode

- a. Managing Identity and Access
 - i. How should you control access to your system?
 - ii. Who should have access to your system?
- b. Managing Secrets
 - i. Password Managers and Secret Management Software
 - ii. Defending Secrets and Monitoring Usage
- c. Securing Compute Infrastructure
- d. Managing Networking
- e. Recommendations for your Security Infracode

13. IV. Scaling Production Readiness

14. 10. Monitoring Theory

- a. Why Monitor?
- b. How Monitoring and Observability Differ?
- c. Monitoring Building Blocks
 - i. Events
 - ii. Monitors
 - iii. Data: Metrics, Logs, and Tracing
- d. What does Monitoring look like?
 - i. Event Detection
 - ii. Data Collection
 - iii. Data Reduction

iv. Data Analysis

v. Data Presentation

e. Monitoring for Sustainable Work

15. 11. Presenting Information

a. Know your audience

b. Choosing your channel

c. Choose your story type

d. Presenting Data in Action

i. Charts are Worth A Thousand Words.

ii. Telling the Same Story With a Different Audience

iii. The Key Takeaway

e. Know your visuals

i. Visual Cues

ii. Chart types

f. Recommended Visualization Practices

16. 12. Building Resilient On-Call Teams

a. What Is On-Call?

i. Adding Developers to the On-call Rotation

ii. Updating On-Call Processes

b. Monitor the On-Call Experience

c. Adopt a Whole Team(s) Approach

- d. On-Call in Practice

- i. Preparing for On-Call

- e. Wrapping Up

- 17. 13. Managing Incidents

- a. What is an Incident?

- b. What is Incident Management?

- i. Roles and Responsibilities

- c. Pre-emptive Planning

- d. Handling the Incident

- e. Post-incident meeting

- f. Practice Failure