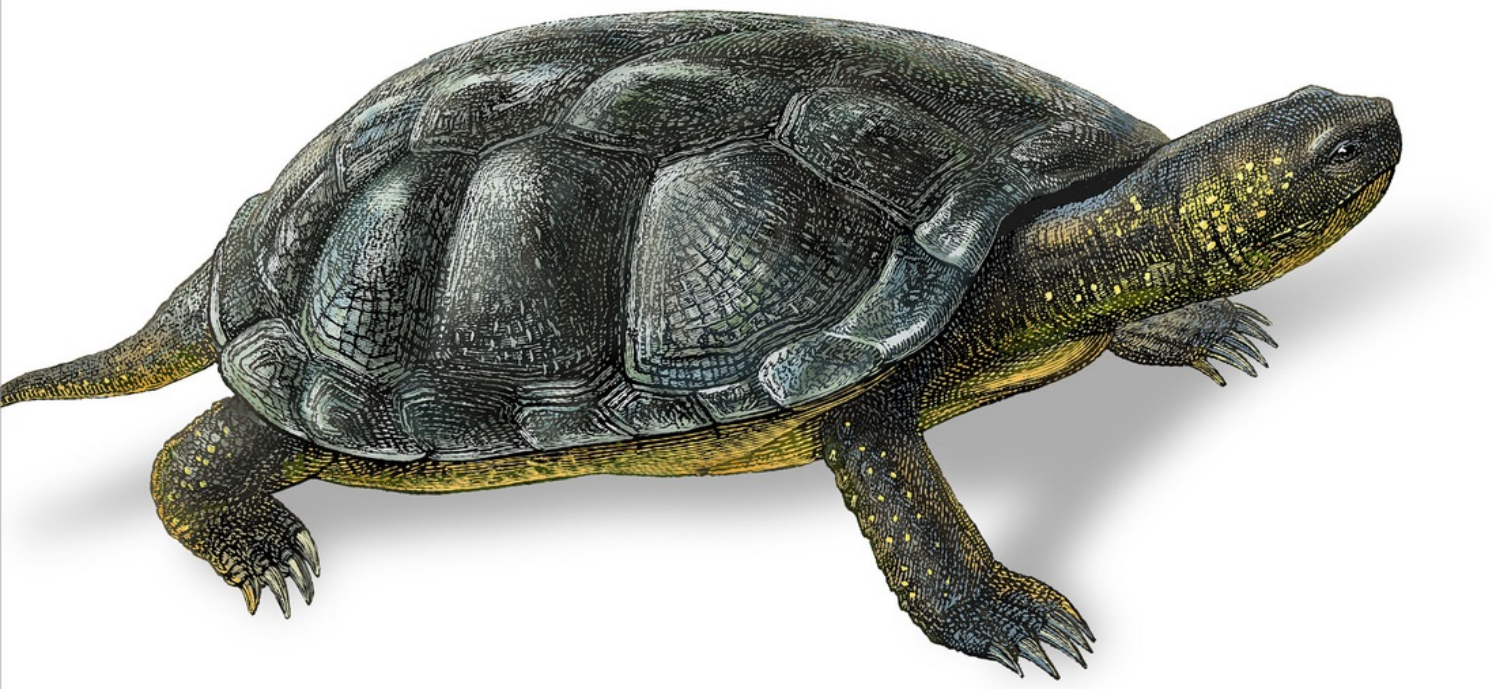
O'REILLY®

Terraform at Scale

Patterns, Practices, and Pitfalls of Enterprise Terraform and OpenTofu Development



Robert Glenn

Praise for *Terraform at Scale*

Terraform at Scale *goes beyond syntax and tutorials to tackle the messy, real-world decisions that determine whether Terraform succeeds across an organization. Robert Glenn turns hard-earned experience into practical, thoughtful guidance on modules, state, automation, policy, and the tradeoffs that truly matter at enterprise scale.*

— Anton Babenko, Terraform influencer and creator of Compliance.tf

Robert Glenn writes with the clarity of someone who has actually lived through Terraform hell, and this book is essential reading for anyone who wants practical advice.

— Diva Dugar, technical PM

Terraform at Scale *looks at the questions and challenges operators run into after they've conquered the Day 1 work of Terraform. Robert's book successfully guides the reader toward a platform that is ready to scale, and does so in a structured, well-thought-out way.*

— Kerim Satirli, author of *Terraform Cookbook*

Terraform at Scale

Patterns, Practices, and Pitfalls of Enterprise Terraform and OpenTofu Development

Robert Glenn

Terraform at Scale

by Robert Glenn

Copyright © 2027 Robert Glenn. All rights reserved.

Published by O'Reilly Media, Inc., 141 Stony Circle, Suite 195, Santa Rosa, CA 95401.

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

Acquisitions Editor: Megan Laddusaw

Development Editor: Jill Leonard

Production Editor: Aleeya Rahman

Copyeditor: Stephanie English

Proofreader: Laura K. Miller

Indexer: WordCo Indexing Services, Inc.

Cover Designer: Susan Brown

Cover Illustrator: José Marzan Jr.

Interior Designer: David Futato

Interior Illustrator: Kate Dullea

October 2026: First Edition

Revision History for the First Edition

- 2026-10-13: First Release

See https://oreilly.com/catalog/errata.csp?isbn=9798341649712 for release details.

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

The views expressed in this work are those of the author and do not represent the publisher's views. While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author 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.

979-8-341-64971-2

[LSI]

Preface

This book is intended to be a field guide focusing on what you might need to do your job now. However, before we dive in, I want to spend a moment to briefly walk you through how we got here.

Not so long ago as to evade memory for many, system administration required physical interaction with individual servers and the use of shell scripts to configure each machine's state. The advent of virtualization and early private clouds improved this process by allowing system administrators (sysadmins) to operate remotely and standardize virtual machine (VM) images, reducing invalid variability.

This evolution was accelerated and democratized by public cloud service providers (CSPs) such as Amazon Web Services (AWS), which commodified compute power: rather than buying computers directly, and thus suffering from the dual challenges of navigating bulk discounting and significant depreciation (both of which require foresight and neither of which are directly proportional to revenue), companies could rent the power they needed. This turned a significant capital expense into an operating expense, which meant much shorter time frames to forecast and costs that were much more related to existing output and thus revenue.

However, despite these advancements, sysadmins still relied on sometimes brittle, largely procedural shell scripts, especially for last-mile provisioning (and sometimes for remediation). Because the full battery of scripts was typically run only at time of provisioning, and by privileged operators, it was difficult to prevent state drift or efficiently manage the multiple duplicate — often not perfectly identical, by design — environments required by agile software development. Although this meant steady, highly revered work for the venerable sysadmins, it was by no means a risk- or stress-free (or steady, in terms of workload) job.

To better address these ever-growing challenges, idempotent infrastructure as code (IaC) was introduced. IaC allows sysadmins to replace the multitude of circuitous, highly dense shell scripts with self-consistent, repeatable configurations that can converge a system to a desired state. This approach is especially valuable in multicloud and hybrid-cloud environments, enabling reliable deployments while reducing the cost of context-switching across disparate platforms.

Consequently, the role of the sysadmin has morphed into the infrastructure software engineer who now writes configurations in Domain-Specific Languages (DSLs) such as HashiCorp Configuration Language (HCL). However, because HCL is heavily interpreted and loosely formatted, organizations often develop distinct "dialects" of code — sometimes more than one and often none significantly more valid than the rest. This makes it difficult to enforce quality and consistency at scale, or to construct definitive best practices that are maximally appropriate to every situation. Ultimately, implementing Terraform and OpenTofu (an open source fork of Terraform v1.5.6, largely operating as a parallel project) requires thoughtful, systemic design choices because infrastructure challenges cannot be solved simply by adding more engineers or automation.

Who Should Read This Book

There are plenty of general books on Terraform and IaC. If you're looking for a single, foundational book on Terraform (or OpenTofu), this is probably not that book. This book assumes you have read one of those other books first, you skipped reading those books and instead took one or more courses on Terraform or IaC, or you've been working on a Terraform or OpenTofu project team for several months or years.

If you're reading this book, it might mean that you're looking for answers that can't be found outside of one-off blog posts or off-hand comments in technical forums (if anywhere), or that you're simply looking for acknowledgement that you're not alone in experiencing a less-than-happy path in your journey. You followed the tutorial but ran into friction once

you went beyond its narrow confines or once you applied it in an existing project that was already full of other components. You came across all the various recommended practices and found them difficult to fully square with each other or your own experience, or you're working within an ecosystem with great gravity that took a different approach altogether and you're seeking a way to navigate these patterns.

Anyone who is managing or operating within multiple root modules, perhaps scattered in distinct repositories, is dropped into a project that is part of a wider, multiroot module or layered ecosystem, or building configuration on top of remote modules — be they developed by a third party or by another internal team — will benefit from this book. Likewise, anyone who is charged with leading a team that follows any of these patterns can leverage the findings presented here. Moreover, if you find yourself relating to any of these positions, many of the scenarios found in this book will seem familiar or reminiscent of situations you've been in. In short, just about anyone who is working in a company with a significant Terraform or OpenTofu footprint will appreciate this book.

Finally, anyone who looks around at their organization's IaC and wonders whether the way it is is the way it should be will find value in this book (spoiler alert: it may be that way for good reasons, or at least not terrible reasons — but the hope is this text grounds the explanation rather than you feeling like you have to accept it on faith alone).

Why I Wrote This Book

As suggested, I wrote this book not as a replacement for other books on Terraform and IaC in general, but to complement those titles. This book is intended to go beyond the introductory text, the tutorial, or the proverbial toy example, extending to the complicated problems and imperfect/uncomfortable scenarios that are so common in real-world infrastructure work. It is meant to address the hard problems many practitioners encounter at some point in their journey, filling the gaps that

one runs into after hitting a wall with a broken state, competing configurations, or shifting corporate priorities.

Frankly, I wrote this book because I wish it had existed 10 years ago when I first began working with Terraform. I wish it was available when I first built out the cloud infrastructure for a corporate client using someone else's opinionated templates or when I created my own templates piloted at a massive multinational corporation and found myself having to make tough decisions as to where resources went and how modules were layered. I also wish it was available when I was dropped into clients who had massive Terraform ecosystems, with layers upon layers and countless modules — some of which were no longer supported, but which were certainly available, muddying the waters of what was appropriate for use — and had to reverse-engineer (or trial-and-error) the acceptable way to incorporate the work of the team I was leading. And I really wish this book had been around to support me in my quest to convince my superiors and stakeholders that things weren't as simple as they expected or wanted them to be.

It's worth addressing the fact that this book was written in the age of generative AI (GenAI) and while the world became enchanted with, and entrenched in, *vibe coding*, where massive initiatives are fully implemented by coding agents powered by language models. I can claim very few of the ideas presented in this book as mine alone. Many of them can be found in Stack Exchange posts, Reddit comments, or Medium articles, meaning that they can be retrieved from the very data the models were trained on. However, allowing these models to consume such sources as the only contexts for these concepts seems illogical if not immoral. Hopefully, no one is fully YOLO (as in, "you only live once") vibe coding their full cloud stacks for a business-to-business (B2B) service or a software as a service (SaaS) application that records Personally Identifiable Information (PII) and takes credit cards, but who would be willing to say such fears are totally unfounded? And moreover, if only as a thought experiment, wouldn't it be nice if you *didn't* have to worry about your infrastructure and whether an AI agent was competent and safe enough to offload all that complexity and

frustration for you? That day doesn't seem too far off, so why not build it on top of a sober, serious, and relatively complete body of concepts, instead of letting it stitch together ideas it extracted from the landfill that is the public internet writ large?

Finally, I wrote this book because I was tired of waiting for someone else to write something like this. I'll be the first to argue that there are bigger names and better choices for a book like this, and I was lucky enough to have a few of them on my editorial team. Hopefully, you find value in this book's descriptions, guides, and examples.

How to Read This Book

Before getting into how to read this book, I'll offer a caveat of how *not* to read this book: it is not meant to be a series of tutorials that you should expect to apply directly; it's not a get-out-of- jail-free card to win arguments with those who disagree with you; and it's certainly not gospel. Rather than applying the examples and guidance exactly, consider them exemplary and, at the risk of trending toward the metaphysical, take what resonates and leave what doesn't. As they say, your mileage will vary. Cloud infrastructure is not a one-size-fits-all pursuit.

Use this book as a reference to validate that "crazy idea" you had, or to reassure your colleagues, bosses, or boss' bosses when they think you're full of it. Use it to reassure yourself when you encounter a P1 issue at 4 p.m. on a Friday before a long weekend. I won't guarantee this book will eliminate all the footguns you might encounter, but I will suggest that, while this text is in no way exhaustive, most of them are thematically discussed here.

Because the examples in this book are intended to go well beyond the simplistic, they can be rather long. It's worth noting that many of them are intentionally incomplete — for example, missing variable or output definitions, provider and backend configuration, or even surrounding resources that get referenced. This is especially true in later chapters, where the focus is less on perfectly written code and more on how to reason through major changes, weird problems, and narrowing constraints.

Intentional omissions are also made for brevity because otherwise the book would stretch another 100 pages with little to no added value. To provide complete examples, I developed a corresponding GitHub repository as an accompanying resource.

This book is largely intended as a resource for both Terraform and OpenTofu technologies (OpenTofu originated as an open source fork of Terraform v1.5.6, when Terraform changed licensing to a proprietary type, and has since largely operated as a parallel project). Rather than spelling both technologies out every time, I have opted to use the shorthand *TF* throughout this book as a catch all for both. Unless one or the other technologies is specified, the example, guidance, or description generally holds for both. This also manifests in examples referencing the CLI, such as `tf apply` instead of `terraform apply` and `tofu apply`, or `tf validate` instead of `terraform validate` and `tofu validate`. As such, for those reading on the online platform, if you intend to follow the examples and guidance pertaining to the CLI in this book by copying and pasting them into your terminal, create an alias for your preferred platform. In Linux or Linux-inspired shells, this can be done in your terminal with alias `tf="terraform"` or `alias tf="tofu"` (these can even be included in a bash run commands, or similar, file such as `~/.bashrc`, so the alias is configured for each new terminal shell session).

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

> Indicates new terms, URLs, email addresses, filenames, and file extensions

`Constant width`

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords

`Constant width bold`

Shows commands or other text that should be typed literally by the user

`Constant width italic`

Shows text that should be replaced with user-supplied values or by values determined by context

> **TIP**
>
> This element signifies a tip or suggestion.

> **NOTE**
>
> This element signifies a general note.

> **WARNING**
>
> This element indicates a warning or caution.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at *https://github.com/robertglenn/terraform-at-scale-book*.

If you have a technical question or a problem using the code examples, please email *support@oreilly.com*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and

documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but generally do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Terraform at Scale* by Robert Glenn (O'Reilly). Copyright 2027 Robert Glenn, 979-8-341-64971-2."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

O'Reilly Online Learning

> **NOTE**
>
> For more than 40 years, O'Reilly Media has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, visit *https://oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.

141 Stony Circle, Suite 195

Santa Rosa, CA 95401

800-889-8969 (in the United States or Canada)

707-827-7019 (international or local)

707-829-0104 (fax)

support@oreilly.com

https://oreilly.com/about/contact.html

We have a web page for this book with additional information. You can access this page at *https://oreil.ly/terraform-at-scale*.

For news and information about our books and courses, visit *https://oreilly.com*.

Find us on LinkedIn: *https://linkedin.com/company/oreilly*.

Watch us on YouTube: *https://youtube.com/oreillymedia*.

Acknowledgements

This book is not a singular accomplishment. Rather, echoing Albert Einstein, Barack Obama, and Leslie Knope (the main character in the TV show *Parks and Recreation*), this was a result of collective effort and standing on the shoulders of giants, both organizations and individuals.

I want to thank John Shiner for pointing me to the Terraform project in the first place back when I worked at the Accenture Liquid Studio in Redwood City, California and Pavol Szorad, who believed me (somehow) when I insisted everything should be built that way.

Thank you, John Forman, who acted as a mentor and advocate for me for several years as I grew as a consulting manager (also at Accenture), and Sven Loberg, the managing director who believed in me enough to bring me along as he developed the Accenture Google Business Group (and who hired me in the first place). I also want to thank Anil Rao, who both challenged my stubbornness and supported my convictions, leading to a highly trusted rapport.

I want to thank Sebastian Stedhil and the OpenTofu Day organizers who gave me a platform from which to preach my somewhat unconventional views from the standpoint of a practitioner, and who always bring together compelling and valuable presentations and discussions on the OpenTofu project and IaC in general.

Thank you to the folks who graciously reviewed this book and provided truly substantial feedback and several corrections and improvements. This includes Anton Babenko — who deserves a special shout out for his *Terraform Weekly* newsletter and Terraform Best Practices website — Kerim Satirli, and Diva Dugar. I can confidently say this is a far better product with the input they provided.

I also want to thank O'Reilly for allowing me to write this book, and my editorial team: Jill Leonard and Corbin Collins. Their guidance turned this into a coherent textbook from a collection of perspectives and examples.

I want to thank my parents for always sharing their pride and support for my various work and side projects, even when I suspect they thought I was stretching myself too thin.

Finally, I want to give a special thanks to my wonderful wife, Diana, for believing in me when I struggled to believe in myself and for putting up with my work-life balance (or lack thereof) during the writing of this book

(and in general, really, since I have yet to actually *achieve* such balance). Her unwavering support kept me going on days I felt like quitting.

With that, let's get into it.

Chapter 1. Bad 'Form!

It happens all the time: an organization adopted Terraform and has been using it, fairly successfully, for some months or years, when suddenly, no one can push a change. The automation jobs that apply Terraform code are either failing or reporting unexpected changes to the configuration of running instances, or both. Pretty much all new features are blocked until the issue is resolved and the jobs are clean and green. You get your team(s) together on multihour long virtual calls to swarm on the issue (despite only two or three engineers really understanding the most complex portions of the codebase, but why should they be punished have all the fun?). Hours turn into days. Meanwhile, other meetings are called (primarily among nontechnical, unfamiliar parties) to discuss why and how and when and why (again) this happened. A week goes by. People have spent nights, weekends, and lunches trying to figure out what's going on. People are fearful of losing their jobs. Then, suddenly, almost miraculously, the issue is resolved by a single line of configuration code that no one can remember being there in the first place ("Oh, yeah, here's the commit where it was removed..."). Or worse, the problem seems to magically resolve itself.

No one wants to be in this scenario. We all want to think we're immune. We overlook the warning signs and even the shouted alarms by that one pesky, annoying guy who keeps bringing up problems until it's far too late: budgets are blown, deadlines are missed, and engineers are burned out.

There has to be a better way.

Dozens of books out there will tell you how to write code for Terraform or OpenTofu. They provide excellent examples, both from canon and the real world, of how to configure a resource, write a module, and store a state file. For the most part, that's where they end: with each subject considered in isolation. There is little definitive guidance on how a full installation across an entire organization should (and should not) be structured or organized.

This book assumes you have tried some of those other books and/or are an active practitioner of Terraform/OpenTofu development. It presents insights and recommendations for enterprise-scale cloud ecosystems, including topics such as efficient delivery, access control, and software deployment environments. Ideally, this book will help you avoid several common missteps and the frustrating quagmires that lie beyond.

The Good Parts

When I got my first internship (which later precipitated my first industry job), I had very little experience actually writing software. Don't get me wrong — I had taken computer science classes and finished assignments that required coded deliverables both in undergrad, where I didn't take them very seriously and didn't do very well as a result, and in grad school, where I excelled in one code-heavy class and (shall we say) *supported* the project work for another. But I generally regarded them with more dread or apathy than enthusiasm.

I wasn't worried. I knew I would be able to figure it out, and the organization to which I was assigned was extremely nurturing, allowing me time to get oriented and contribute at my own pace and ability. They used a web software that I had heard of but never really used before: JavaScript. At that time, I still had bad memories of the projects I had half-cobbled together in college using Java and was worried that this would be a similar language. Luckily, I found it to be quite different. Not exactly easier to understand, but easier to get running (I didn't have to validate using other third-party virtualization software that I would always find a way to misconfigure on my laptop, I just had to have a web browser) and quicker to experiment with. I could just print results or values out into HTML elements to see the outcome of updated logic immediately.

I don't recall if it was recommended to me or if it came up in a search engine result, but I remember reading *JavaScript: The Good Parts* by Douglas Crockford (O'Reilly) very early on in this experience. This book was a game changer. There were a ton of resources that explained, in

excruciating detail, how prototypes were sort of like classes from the object-oriented programming (OOP) paradigm in these ways but weren't like classes in these other ways, how variables were hoisted and how this element should be treated in different contexts, and how functions can be objects, or constants, or just plain old functions. This was the first guidance I came across that went beyond the *hello world*–style tutorials whereby the reader is asked to download pre-built source code, run it in a fully prescribed way, and review cherry-picked notable lines of code that illustrate whatever concept is under review.

In an ocean of material that taught the equivalent of how to spell and fit together the parts of speech, or how to recite well-worn refrains, Crockford had delivered a master class in poetry or journalism: how to actually use the components for successful, maintainable implementations, not just write spaghetti code that happens to work somehow. And perhaps more importantly, he explained which components to actually use.

This book you're reading is an attempt to bring the same to Terraform and OpenTofu projects: beyond *how to do X, Y, and Z*, you'll be presented with *when to do X*, *why to do Y*, and *reasons to avoid Z*. Rather than present concepts in a vacuum, the book assumes the concepts are to be applied in the real world, which undoubtedly involves more than just code (agendas, budgets, skillsets, biases, and so on). The goal of this book is not necessarily to establish or present a set of industry best practices (which it will surely include as much as possible), but it is meant to provide you with practices that will lead toward success and away from frustration, dysfunction, and failure. It's also intended to expose the severe lack of industry best practices; my personal suspicion is that IaC is more of an art than a science and that there are truly few practices that can be considered "best" in all situations. However, this must be explored more by the community, and this book aspires to begin that discussion.

With this context in mind, I'll begin this book with a brief overview of what I would likely include as the primary content included in a theoretical *Terraform: The Good Parts* title.

> **NOTE**
>
> Nearly all the concepts and many of the techniques in this book are applicable in both Terraform and OpenTofu. OpenTofu is an open source fork of a prior version of Terraform and has since become its own solution with its own product roadmap that largely, though not exactly, mimics that of the now source-available Terraform. For convenience, I will refer to them, collectively, as *TF*. This will also be reflected in the CLI rendering, as in `tf plan`, `tf apply`, or `tf import`.

Concepts

From a very high level, IaC seems fairly straightforward and familiar: you know what infrastructure you need, you know how to code, put it together, and *voila,* you have IaC! While this is true, there are innumerable nuances and variations that make the space seem much more like an art than a science. Thus emerges a collection of additional concepts, many of which are unique (or uniquely *significant*) to IaC. Moreover, there are a slew of concepts that are relevant or valuable in other software settings that have either no place, a reduced (if not negative) impact, or severe diminishing returns in the IaC paradigm.

Important concepts unique(ish) to IaC

The following concepts are unique or have unique implications in the world of IaC in general and TF specifically:

Idempotency

> This refers to the property that TF aspires to in which an element of an applied configuration (a cloud resource) is unchanged when subsequent operations such as `tf apply` operations are run against it. This property allows one to run TF operations to detect drifted resource configurations or to rebuild deleted or tainted components without the risk of doing unnecessary things to other resources, such as recreating or duplicating everything that is encoded.

TF plan

As with many other software tools, TF is driven through a well-established CLI. Unlike many other software tools, the primary use of TF, to establish cloud infrastructure, is executed in two steps: first to plan the change and then to apply it. The plan step effectively amounts to a large, complex procedure that runs from start to finish in an ordered fashion, similar to a shell script. The execution plan is essentially *immutable* and does not allow for variation once it's constructed. Although the plan can be automatically accepted (which is decidedly *not* a recommended approach), it's always rendered and describes what TF will attempt to configure upon the execution of the apply step.

> **NOTE**
>
> Two important notes: A `tf plan` command will not ordinarily detect existing resources and may not expose certain violations of a particular CSP limitations, such as a name or ID collision. There are also typically (many) multiple ways to construct HCL configuration to arrive at the exact same execution plan. We'll explore these in detail throughout the book.

Stored "state"

In all software paradigms, there are at least two components: the code and the execution. The *code* is what is written and renders an expected state; the *execution* is what a system (OS, platform, runtime, and so on) executes as it interprets the code. In IaC configurations, there are always three components: the code, the execution, and the (point-in-time) state at execution. When TF executes a `tf apply`, it records the changes it made in a JSON-encoded file called a *state file*. This will be explored further in Chapter 7.

Backend

The state file that TF generates and updates each time a `tf apply` is executed gets stored in a user-designated location (typically a remote storage service such as AWS S3). This is referred to as the *backend*.

Locked state

A follow-on implication of stored state is the concept of a state lock. The concept of a *lock* or *lock file* is nothing new in developing transaction algorithms, but it takes on a greater emphasis in IaC because it helps to prevent concurrent applications of a particular project, which might otherwise introduce race conditions, causing well-known problems.

Provider

The *provider* is the software that TF uses to turn HCL code into a procedure used by another interface, such as API calls consumed by an associated service provider, often a CSP. A provider may be developed by HashiCorp (with support from the relevant service provider), the service provider itself, or members of the community. A provider is often developed as open source software (OSS), albeit typically governed by a single, well-known entity.

> **NOTE**
>
> Typically, I will use CSP when referring to the platform or system that actually hosts the running resources (for example, AWS or Google Cloud Platform [GCP]). The term *provider* will denote the software that TF uses to interact with such a CSP.

Important borrowed concepts

The following concepts are borrowed from the wider software development field but have interesting implications for IaC and TF:

Encapsulation

Encapsulation is an important, and often overlooked, concept in constructing TF projects. All TF configuration is contained in modules that may be intelligently scoped and nested, and even developed in isolation, creating richly complex constructions that may be developed in a highly controlled way, just as in OOP and other paradigms that

implement and promote encapsulation. Constructing a module is as simple as placing *.tf* files in a directory and referencing it either in a TF CLI command or as the `source` meta-argument in another module. This means module development is largely an exercise in effective source control.

Resource management

If you're using OpenTofu or Terraform Enterprise and are responsible for the compute resources that it runs on, the resource management of TF commands becomes especially important. The more components a TF project maintains, the more is stored in the state object that gets constructed in `tf apply` command and read in a subsequent `tf plan` command. Although this may not be especially important to consider in small projects, very large TF states can become cumbersome to traverse to ensure all dependencies are respected and effectively ordered, and may even exhaust your service provider's API call limits (this final point is relevant even in HCP Terraform [formerly Terraform Cloud]). Similarly, the larger the TF state, the wider the blast radius if it becomes problematic. Conversely, having many smaller TF projects may require sophisticated scaling governance or suffer from execution queues as tasks await an available automation agent; this becomes especially relevant when also implementing multistage automation pipelines.

Identity and access management

As with most modern software, proper TF implementations include significant Identity and Access Management (IAM) controls. TF itself should have a digital identity configured in each CSP it services. It's also common and prudent to establish access policies around the source control repositories in which TF is stored. Doing so creates a basic secured supply chain for your infrastructure and can be evolved with typical principles such as more granular policies or identifiers, and so on.

Standards and style

Like many other languages, HCL has a strict syntax. However, unlike a YAML or Python, it is lenient on whitespace, generally allowing for any number of spaces or carriage returns to appear consecutively. This means TF will not enforce any consistency in how it appears. Nor will TF care about the order in which configuration appears, as long as resources are enclosed in a single block. As such, it can be important to follow a consistent standard or style of writing and organizing TF modules and projects. A review of one of the popular Robert C. Martin books, such as *Clean Code* (Pearson, 2009) may serve as an excellent lesson or reminder for how to improve readability and consistency.

Concepts irrelevant to (or diminished for) IaC

The following concepts are less important (or relevant) to IaC in general or TF, specifically:

Entry point/main function

Because of the two-step nature of TF — first converting the HCL you write into the CSP-specific API calls using the provider software as described previously — there really isn't a main function or entry point for the process. When you run a TF command, you target an entire directory (including any subdirectories referenced through modules), not a specific file or function. Instead of requiring you to construct perfectly ordered operations, TF will establish this order as part of a planning phase either directly using the `tf plan` command or as part of a `tf apply` command.

Performance

Although we all want instant gratification, IaC performance may not be the place you want to invest. There are certainly more important aspects of your IaC supply chain than whether it takes 10 seconds or 1,000 seconds (16 minutes and 40 seconds). Get your IAM set (see Chapter 12), implement the ancillary components and services to fully

operate TF programmatically (see Chapters 7, 9, and 10), and apply sensible abstractions (Chapters 3 and 6) before worrying too much about trimming off seconds and minutes to your build. That said, if your jobs are taking dozens of minutes or hours, something *is* seriously wrong. Perhaps you're subject to constraints that are inappropriate for your current use case. For example, don't run your company's environments using constraints you found in a tutorial; that was a tutorial, it wasn't meant for real-world scenarios (see Chapter 11 for more examples). Perhaps your TF project has become bloated or is suffering from inefficient use of TF capabilities (Chapters 7 and 8).

Unit testing

There is no value to unit testing typical TF configuration, at least not in the way it's most commonly used in software development. First, you're mostly not writing in an actual programming language; HCL is a configuration language and gets interpreted as inputs to TF. I imagine (and dearly hope) that HashiCorp (and the OpenTofu folks) are extensively unit-testing its logic before shipping it out to their consumers. Second, unless you're working in one of these organizations as a developer (or in the latter case, are a member of the development community), TF is not your code, so you have no business unit-testing its behavior. Instead, it's more valuable to introduce validation to ensure the HCL you and your team are constructing adheres to your organization's practices and policies. The notable exception here, which you'll explore in later chapters, is constructing tests using the TF testing framework around the results of custom, complex logical operations. However, as you'll see, this should be viewed as secondary to custom validation expressions.

Integration testing

Along with unit testing, there's little value to integration testing TF — again, at least not the same way it's applied to other software development practices. What exactly are you testing? The only way to truly ensure that your TF project will succeed in configuring all the

components the way you have encoded them is to actually run it with the live values; this is possible using the TF testing framework or other third-party tools. However, even using dummy values and building an entire test environment to vet a configuration can hide things like bucket or project name collisions until the real values are attempted. Not to mention the cost of creating such an environment (which is hopefully cleaned up quickly; this is a common feature included in TF testing frameworks). The costs seem to far outweigh the benefit, especially when so many low-cost/high-value initiatives are undoubtedly available.

> **NOTE**
>
> The real offender to engineering effort is test coverage. Hopefully, you're not asked to meet some high percentage of TF code covered by tests, for the reasons listed previously. However, it may be unwise to actively campaign against test coverage because it will likely be a lonely, uphill battle. In nearly all software contexts, the value of testing indeed outweighs its cost (eventually subject to diminishing returns, as most good things in life are). The natural inclination to apply successful patterns to new problems, therefore, will generally lead bright, influential folks to conclude that there is a need to apply such validation techniques. That being said, if you're asked to develop Terratest cases, it might be more worth it to not argue and just go with it. You'll at least be learning (or growing your experience with) the Go programming language.

Features

Along with the important concepts, there are a number of features that everyone should be familiar with, especially to grow a project to an enterprise scale. Many of these features can be overlooked or ignored without necessarily compromising the success of a TF command, but the most successful projects incorporate them and capitalize on the value they each can bring:

Variables

The named *variables* in TF are the inputs to their enclosing module (which, again, is the entire directory in which a set of *.tf* files reside).

This is how your organization-specific information may be abstracted, isolated, and passed to the configuration. Variables have their own configuration block types and include optional fields such as description and type. Variables even support custom validation for values that must adhere to certain internal constraints. Variables are explored in great detail in Chapter 2.

Output values

The point of a `tf apply` operation is the running services maintained by the TF configuration, so these would obviously be the most important output. However, TF also supports output values that may be constructed using custom logic and other values such as computed resource attributes. These may be used to pass information out of a module, which is especially useful to print computed values in a highly readable format in the log output of a successful `tf apply` or in the case of gathering the values generated within a nested module for use elsewhere in the configuration (such as another `module{}` block). Output values are explored in Chapters 4, 6, and 7.

Local values

Along with the named variables used as inputs to a module, TF supports local values that are also referenced by name and are most commonly used to store intermediary logic or to construct complex data types expected by resource or module fields. Effective use of local values is investigated in Chapters 4 and 6.

Data sources

Data sources fetch detailed data for the referenced resource type. They introduce API requests to the particular CSP, and TF will attempt to make these during a planning phase, where possible. They can be especially useful for ensuring the existence of resources that this TF project's resources depend on and that are maintained external to the current TF project — perhaps in another TF project.

Modules

As discussed earlier, *modules* are the primary encapsulation mechanism in TF, providing delineation and boundaries between different collections of HCL configurations. As suggested, a module is simply a filesystem directory that includes *.tf* files. It may be referenced from another module, thereby nesting logic and organizing related resources or targeted by a TF command directly. Modules are covered extensively in Chapters 6, 7, and 8.

Common Misconceptions

TF cannot do everything. It can't even do certain things well that it explicitly supports. Unfortunately, this truth is lost on many seemingly competent professionals expecting TF to solve all of their problems. Most likely, decisions regarding infrastructure aren't made by TF practitioners, but by architects and managers, and can therefore come with expectations and biases with no foundation in reality.

For example, it's not uncommon for an influential individual or group to plan the configuration of a set of components in a new CSP your organization is adopting. They insist that it'll be "just like how it works in [another CSP your organization uses]," and they schedule a series of tasks intended to implement this configuration. After a number of sprints, the configuration is complete and compute resources begin to be launched with your technology department's requisite battery of security software, only to sound all the alarms. You realize that none of your resources are protected or restricted from public access. The individual or group doubles down: "It should have worked, you did something wrong." You show them your work and their specifications, and you send them articles about how the CSP implements network routing by default, and they suddenly become both agreeable and totally forget about how wrong *someone* might have been. For what it's worth, they mumble something about how they "thought TF would have caught this." You brush it off because you just don't have time to get into this again.

This could be avoided if the following common misconceptions about what TF can do were more generally understood:

Monitor your infrastructure and correct deviations from specification

> TF, specifically the executable, will not constantly inspect your running services and report on or take corrective actions for deviations from the codebase. There are tools that do this, and there are certainly benefits to such a solution. For example, the HashiCorp platform products HCP Terraform and Terraform Enterprise (TFE) do support drift detection and enable alerts that may be integrated with other tools.

Know everything about your existing infrastructure

> TF won't seek out other resources either to determine if it will fail due to a name collision or to understand any dependency implications. It will only provision what you tell it to. In a way, this is a gift and a curse; a gift because it allows one to better adhere to principles such as separation of concerns, and a curse because it leads to some of the misconceptions presented here.

Fill (or even report) security gaps

> Because TF does not analyze your entire infrastructure, it cannot assess possible security holes (for example, overexposed resource permissions or overly permissive firewall policies). Interestingly, in this case, the opposite is true in a way: many services that do detect possible security holes will often provide snippets of HCL that would assist in filling the gap. However, the key is still in introducing these strategically, rather than copying them at the bottom of some random *.tf* file.

Roll your encryption keys and other sensitive information

> TF will not keep track of time limitations or other such attributes to which a resource might be subject. There may be a way to construct such logic within TF, in certain cases, but this would be custom logic and may be better served by a different solution (not by IaC). Indeed,

there are plenty of available solutions, both enterprise and open source, that will do this much more appropriately.

Keep you from doing something foolish

Again, TF will provision what you tell it to, so if you tell it to do something foolish, it will. This book is not intended to be a guide for cloud architecture and assumes someone with appropriate expertise is responsible for assuming the role of architect.

Totally abstract away CSP differences

This is because the provider is CSP-specific, and there are differences in how different CSPs approach analogous resources. Some of these are nuanced, such as lagging features and differences in product roadmaps. In other cases, the gaps are wide: see virtual private cloud (VPC) implementations across the big names. In going from one CSP to another (either to migrate or expand in a multicloud approach), it's certainly not as simple as copy/pasting and find/replacing the corresponding resource names. This point is perhaps the most obvious. It's included here more for completeness than novelty.

Technical IaC Debt

TF projects are (and IaC in general is) awash in technical debt. A major part of the problem is a lack of resources that might constitute industry best practices. Even the most highly recommended practices are more suggestive than definitive, leaving a lot of room for variation, nuance, and outright departure. There are certainly emergent practices, some of which are largely self-consistent and relatively complete. However, to my knowledge, all practices have their drawbacks, and it's not uncommon to incorporate multiple such practices or to switch from one approach to another. Moreover, it's overwhelmingly common for projects to start small and simple and to evolve into increasingly complex structures over time. This all naturally results in technical debt. This section presents some

common nightmare scenarios where technical debt arises, suggests opportunities for paying it down, and offers insights into how to estimate the total bill.

Nightmare Scenarios

As presented earlier, TF is really only aware of the resources that it provisioned at the time of a `tf apply`. It does not track changes to those resources that are made by other systems or operators. It also does not provide any additional protection to prevent such updates to the resources it provisions. This can give rise to our first nightmare scenario: a resource (call it resource A) provisioned by TF is modified by some other mechanism (perhaps manually in the browser console or using a CLI by a user with write access permissions); later, an emergency change needs to be made to another resource (resource B) in the project, and when a `tf plan` is executed, it reports that it will revert resource A, undoing the manual change.

Although there are certainly several options (quickly encode the change to resource A; ignore the revert for now and fix resource A later; and so on), they may not all be stomachable at 4 p.m. on a Friday. Suddenly, folks are being asked to stay late at the office (or online) to provide a fix, impacting team morale and confidence. Perhaps the obvious solution is to prevent any process other than TF (manual or otherwise) from making changes to the resources TF maintains. However, unless severe tragedy or an unamusing comedy of errors has befallen your organization, someone has the keys to the proverbial kingdom, and most likely they have delegated access to portions of the kingdom to various ostensibly competent individuals (such as staff engineers or senior site reliability engineering [SRE] specialists) in some way. Thus, without effective and complete tooling and policies, it's purely up to the discipline of these individuals that this scenario may be avoided. Chapter 7 explores such tooling and policies.

> **WARNING**
>
> A common refrain in this book will be some variation of *don't mix TF with other processes*. A corollary (and indeed a somewhat more understanding version) would be *(nearly) anything you do outside of TF to resources maintained by TF is technical debt* because you must address the modification in TF before the project can progress any further. If the only outcome of this book is that it convinces you and your organization to incorporate the proper tooling and policies so that you never need to make updates manually (either with a CLI or in a browser console interface), I would consider it a total success.

As described earlier, TF stores a state file in a user-supplied backend. To prevent a race condition, TF places a state lock file at the backend location to designate that a `tf apply` command is in progress. Another nightmare scenario can arise when a `tf apply` command is forcibly cancelled. In this case, the TF project remains in a locked state. If no one is available who has the permissions to release the lock, the project is effectively disabled, as no changes can be applied until the lock is removed. Chapter 7 introduces techniques, such as the wildcard job, that helps to mitigate this situation.

One of the few well-established practices for which there is perhaps universal consensus is around the use of the `count` meta-argument. Technically, `count` may be used to create a specified number of copies of a particular resource or module. However, as will be detailed in Chapter 3, the most appropriate use of `count` is in combination with a ternary operation that returns either `0` or `1`, thus establishing a way to conditionally include a single such resource or module. The primary reason this is the only acceptable use of `count` is in the case where a resource in the list needs to be removed. Unless the resource to be removed happens to be at the end of the list, there is no (remotely convenient) way to do this. Certainly, with sufficient forethought, and perhaps some clever logic, using `count` in this way can be avoided. However, it's possible to construct other convoluted iteration logic that results in a similar experience. Typically, the only way to really address this kind of problem without recreating resources (a common expectation of product owners and other business-sensitive

roles) is to manipulate the JSON object encoded in the state file using the `tf state` and `tf import` commands, which are covered thoroughly in Chapter 7.

A somewhat more subtle situation comes about when the basics of TF are forgotten or misconceptions arise. For example, it's not uncommon for a complex and seemingly airtight architectural solution to get midway through implementation and suddenly an influential group of highly regarded, well-intentioned individuals throw in a "simplifying" change, mid-sprint ("because we're agile"), to how the networks should be constructed with TF. They send someone to your standup to briefly explain the change and rationale, and before they have to go because they "have to run to another meeting," they drop the bomb: there is to be no downtime. And then they're gone. You try to get more information from them over chat, but somehow, they don't have all the details and would have to include so and so, maybe on a call. You decide to get everyone together on a call, but they each "can't do it today, maybe tomorrow/on Monday/next Wednesday;" they promise to check their calendar and get back to you (or you can attempt to corral everyone's attention in an email thread, another tall order). Your project manager is asking why you can't "just try to do it like they said" in the meantime. You tell them you probably could, if you had the proper access, but your organization's policies restrict you from making the change. You can request access, and try to get the approval expedited, but by now it's Thursday afternoon, and your expectations are not high. Also, you've never done this before, and they're asking you to make risky modifications to how foundational components are configured. Chapter 7 provides both commentary and solutions for this scenario.

There is one scenario that is especially frustrating because it's the result of many hours (and sprints and months, worth) of very deliberate and well-intended work: when conflicts in policies occur and create a gridlocking effect. Although this seems like it should be extremely rare, it's more common than it appears because it can lie dormant for months and years, only to surface well after the initiatives that established the policies have completed, and perhaps even after the implementing team has turned over

or disbanded. As a highly contrived example (only because it should be obvious enough to avoid, not because it is all that uncommon or unnatural), if one policy caps the number of a particular resource in a particular region, and another policy restricts the regions available for provisioning any policy, it's certainly possible that the resource capacity policy eventually gets triggered. These policies may even have been established in different tools and by different teams, compounding the issue's complexity. Technically, this is a fairly simple problem to solve: either update the resource capacity policy to allow for more such resources in the region or update the region restriction policy to allow for additional regions. Bureaucratically, this can take multiple all-hands working sessions to even come to a decision; ideally, this can be completed in a week, but schedules and agendas do exist at companies in the real world and they do create real delays. There is nothing more frustrating than red tape keeping you from a one-line code change.

Perhaps the most (initially) terrifying scenario is when a state file or the entire backend is destroyed. Suddenly, the running resources are all unmanaged, with no way to update, add, or remove resources. What a nightmare! Luckily, there are several ways to reestablish a TF state file/backend, which are explored in detail in Chapter 7. Depending on the type of backend, you may even be able to restore things relatively painlessly. However, the best way to address this situation is to avoid it, typically through access control policies that prevent such a deletion in the first place. Policy-based approaches are discussed in detail in Chapter 12.

Paying It Down

Technical debt is a commonly discussed concept in software. None of these techniques are especially unique to TF/IaC. Many other applications of modern software may be entirely replaced by building a fully functional alternative solution, testing it thoroughly, and swapping it out in a way that is both highly controlled and that minimizes impact — and is also publicized, if there will be downtime, although any downtime should be highly predictable. By contrast, the resources provisioned with TF likely

undergird your wider digital ecosystem, and wholesale replacement of solutions may be infeasible. Therefore, actually paying off technical debt in TF projects may be more attractive than in other software projects.

> **NOTE**
>
> A fairly prevalent attitude (albeit one that is often somewhat privately held) regarding technical debt in many circles is that it is a necessary evil and that paying it down is more trouble than it's worth. Instead of being consistently and proactively addressed, technical debt is typically only confronted after it becomes noticeably problematic and on a case-by-case basis. Rather than frame technical debt as something that can be left as an afterthought in TF projects, it should be viewed as something on the order of an existential threat; mitigation is still appropriate (perhaps more so than a draconian obsession with paying off all technical debt completely), but a consistent, proactive approach to mitigation is recommended.

Some technical debt is indeed low impact; however, often the lowest impact cases are also the most obvious and simplest to address. Because of this, it's strongly recommended to address these as they pop up, rather than ignore them indefinitely. As an example, most resources have several unique (or sufficiently distinguishing) identifiers supplied by the CSP that can be used to reference them, such as name, ID, or a generated URL. One resource, say a subnetwork, may specify another resource, such as a VPC, as a required field in order to properly establish the requisite relationship. Some TF providers will allow these multiple formats for a single field when TF constructs the plan initially, but because the CSP returns a specific format when queried, subsequent plans will report a change to the field (which will effectively result in no actual change to the field). This, in turn, causes TF to report an update to the resource (albeit not always one that might result in the resource being fully recreated) in the output of a `tf plan` command. Typically, this results in a no operation (no-op) `tf apply` procedure (the resource doesn't actually report an update as a result of the `apply`). However, this leads to projects that always report changes in the planning phase — a *never-clean* project — because TF will always try to execute the plan. The solution is to simply use the format that is reported in

the `tf plan` output. Maintaining never-clean builds may hinder or complicate simple validation techniques for detecting drift or confirming updates.

Some technical debt can be addressed over time, with steady, regular updates, either as its own long-form initiative or as a regular process after so many sprints or months. For example, over time, configuration may become repetitive, and it becomes attractive to refactor it into more streamlined HCL code. Perhaps a long list of nearly identical services can be combined into a single `resource{}` block with the `for_each` meta-argument, to reduce the chance of needing to make updates in multiple resources (possibly inviting typos or other typographical errors).

Alternatively, a collection of related resources used in multiple places throughout the collection of codebases might be bundled together into a module that can be developed remotely as a separate project, to be developed in isolation, and referenced in a `module{}` block wherever needed. This can be presented to the various development teams, and a timeline can be set to refactor the existing components into the new approach. In the exceedingly common case that no resources can experience downtime, this may require an experienced and confident hand (likely with an elevated level of access) to reestablish the TF state to reflect the new construction of the code. Such a slow and steady approach can reduce or avoid any headaches or inefficiencies that undoubtedly arise if this is never done, or if it's done well after real problems arise. This latter example is explored in greater detail in Chapter 6.

In the case where repetition or other inefficient patterns do get out of hand, it may be attractive to replace, wholesale, entire swathes of configuration. This may be due to a redesign in how networks are organized (such as to consolidate or democratize control) and by whom they're maintained. This may not even be a great technical change, but the fallout of a fairly bureaucratic series of decisions that, in a way, carry more gravity than a purely technical one. In other words, you're not getting out of this one. This will also most likely come with the proverbial *no downtime* clause, which again means manipulating the TF state.

Rather than initiate a complete overhaul of a large collection of foundational components, it may be most prudent to take an approach inspired by the *strangler pattern* from application modernization circles by which components within the collection are carefully and strategically migrated from one configuration to another. However, unlike breaking down a monolith application into a collection of microservices, you're likely going from one monolithic configuration to another. Instead of slowly chipping away the monolith, creating isolated components from the chunks of extracted code, it's like you're building a new, reimagined monolith, scaffolding out the updated organization of resources, piece by piece, and filling them into the new, initially sparse-looking structure. Such approaches are discussed in Chapters 6 and 7.

The Real Cost of Technical Debt

Technical debt occasionally rears its head as a serious topic of debate, but it seems that folks generally fall into two relatively stable camps, each with little interest in making especially passionate arguments for their perspective: those who believe technical debt is a real problem but can't possibly spare the time to pay it down, and those who believe technical debt is hogwash — a boogeyman that a few ornery staff and senior engineers grumble about from time to time but that never actually arises. The problem with technical debt is it can be a ticking time bomb that always seems like a nothingburger until it explodes, often at the worst time, such as right before or after a major feature release.

Because it's so often such an afterthought, the real cost of addressing the technical debt is rarely considered in its entirety. It's easy to point to the time it takes to schedule, implement, and deploy a fix as the cost of a fix, but it goes beyond this purely technical cost. For starters, any change, no matter how necessary or obvious, must be evaluated by the solutions architects and other stakeholders that are ultimately responsible for how your infrastructure is configured and maintained. If such individuals are doing their jobs properly, this is not a simple, rubber-stamp review. Finally, there's the strictly bureaucratic cost: the time spent in meetings among

stakeholders negotiating the various terms of the changes — the timelines impacted, budgets hit, and teams assigned responsibility.

There's also a cost to not addressing certain technical debt that, in isolation, may seem negligible but that adds up over time and across a large team. For example, excessive repetition may result in additional seconds to find the appropriate place to make an update, or to make all the changes across repeated blocks. Doing this once or a few times may be fine, but if this is the normal experience, where every change is made in a 1,000+ line *.tf* file, it can add up over time. It's also likely to introduce new muscle memory as developers are conditioned to do this repeatedly, which might result in overlooking issues as the changes are made with increasingly less active thought. These oversights may result in more costly problems later.

Finally, there are certain issues that teams may just choose to live with and can go unchecked for months and years. An example is the never-clean projects described earlier. Whether an actual change is necessary, a `tf plan` command would report a change, and this might cause a human or automated operator to follow through with an unnecessary `tf apply` command. This is both a waste of time and compute (which may be billed on-demand, depending on the way it's architected).

Another example is *tightly coupled projects*, whereby two or more TF projects depend on each other, and a change in one incurs a change in the other(s). Each single update may be viewed as a nuisance, but as they add up over months and years, they eat away at budget contingencies and team morale. It's better to spend the time it takes to refactor these projects to be more easily maintainable.

Recommended Tools and Services

There are countless combinations of tools and services that may successfully serve TF projects. Rather than prescribe a specific combination, this section presents the general functions served by various

tools and services. You should choose whatever product you like that serves the roles covered here.

The Basics

These are the basic tools and services recommended for any serious TF project:

Code editor

> This is where you may edit the HCL and other configurations that together make up a TF project.

Terminal/shell

> This is where you will issue any CLI commands such as those for source control management (described next). Early on, this may also include running CSP and TF commands, but as projects progress, this should fall out of favor/be disallowed.

Source control

> As previously mentioned, a TF module is made up of a filesystem directory that includes *.tf* files. These can and should be recorded in such a way that they can be shared across multiple members of a team and referenced in a controlled fashion by automated operations or by TF itself. This is typically addressed using *source control management* (SCM; also referred to as *version control*, *revision control*, or simply *source control*). The most common solution for this is Git, and indeed, despite the message in the section intro, this is likely most recommended, if only for the sake of overwhelming market share: most likely, it's the solution with which your organization's vendors, clients, and personnel are most familiar, so it will be the most interoperable. That said, it's certainly not the only option.

Task automation

As suggested, certain tasks may be performed in terminal shells on engineers' workstations at the beginning of a TF project's development, but it's highly recommended to establish a task automation framework early on. Your experience will depend on whether you're using OpenTofu or Terraform (HCP Terraform or TFE), as expressed in the next subsection.

Cloud service provider

The CSP is the paid service in which the resources you intend to manage with TF will be hosted. Your organization likely has exactly one truly root account per provider. You'll generally need view access permissions to access the CSP, with elevated access to edit resources if you intend to run the `tf apply` command locally. This may be extremely obvious but is included here for completeness.

For OpenTofu Versus HCP Terraform/Terraform Enterprise

One of the main differences between developing in OpenTofu and developing in the Terraform products HCP Terraform and TFE is that the latter products provide a number of additional tools and solutions that you would have to develop on your own to achieve a comparable setup (or not). The following compares certain aspects that are included in Terraform and would have to be otherwise established in OpenTofu:

Task automation

For Terraform, this is provided by the central web interface. For OpenTofu, often, the same platform that's used for managing the delivery and/or deployment of application software may be repurposed to automate TF. In both cases, the platform performing the automation needs to be able to authenticate with the appropriate CSP and be authorized to administrate specific services and resources within the CSP. It would be responsible for systematically running the TF commands, such as `tf plan` and `tf apply`.

Workspace management

In the Terraform products, there's a feature called a *workspace* that is a built-in way for using different input variables with the same TF module (again, established as a filesystem directory) against a common backend. This feature is offered as a way to create similar environments, but in cases of significant variation among environments (if only in terms of breadth or over resource capacities, which may happen between a true development environment and production), there may be more appropriate approaches, even for HCP Terraform/TFE, which is explored in Chapters 7 and 9.

> **NOTE**
>
> Generally, throughout this book, I will refer to a *TF project,* which is what I'll use to denote a TF module (filesystem directory) that is targeted by a TF command, such as `tf plan` or `tf apply`. In most cases and for most purposes, this could be considered synonymous with a workspace. Only in topics in which there are workspace-specific considerations will a distinction be made.

Summary

This chapter covered the following topics:

- A review of what might be considered The Good Parts of Terraform and OpenTofu
- Important concepts that are unique to TF and the wider IaC paradigm
- Concepts that are borrowed from the wider software industry, and their manifestation in TF
- Important features of TF and an overview on how they may be utilized

- Common misconceptions that nonpracticing engineering professionals may have about TF
- Implications of technical debt in TF (and IaC)
- The basic tools and services necessary to operate any TF project
- The main differences between operating an OpenTofu project and a HCP Terraform or Terraform Enterprise project

I hope you find this book useful. I hope it helps you evade perilous situations and correct missteps (perhaps some of which you don't yet realize your team has made). But most of all I hope it begins a dialog to establish patterns for successful operation and organization of TF projects at enterprise scale.

In Chapter 2, you'll get an in-depth analysis of variables in TF. You'll learn how to introduce type safety and input validation, as well as recommended approaches to balance consistency with maintainability.

Chapter 2. Input Variables

Understanding *input variables* is a natural starting point to writing high-quality, highly scalable TF projects. Typically, these variables are introduced to provide abstraction to TF projects, enabling reusability of configuration and easy updates to modifiable resource attributes. However, for projects that are intended to be preserved, they should be introduced early in the project development.

In this chapter, you'll review the various ways variability may be introduced, focusing on the input variable component itself. You'll also gain exposure to the other governance capabilities that input variables provide, such as type constraints and `validation{}` blocks. By the end of this chapter, you'll understand how to create high-quality input variables that optimize the balance between expressiveness and control.

Variability of TF Configuration

As an IaC platform, TF is primarily concerned with taking encoded infrastructure configuration information and applying it in an idempotent fashion such that the result is the same no matter how many times you execute it against the same configuration. It evaluates the contents of the *.tf* and *.tfvars* files (and the provider code it depends on) to construct the execution plan. As presented in Chapter 1, this *execution plan* is an immutable script and cannot be modified once it's constructed. However, as was also covered, there are usually multiple ways to construct the HCL configuration that you write to arrive at the same execution plan.

Even beyond the order that resources or modules appear (which is both an obvious and an irrelevant differentiator, since TF will construct the order of execution in the execution plan anyway, so order was never especially important to the TF platform), there are often multiple default values for any particular resource. This means setting a resource's field to its default

value is functionally equivalent (for TF's purposes) to omitting this field altogether. Additionally, as I also covered in Chapter 1, most TF providers will support multiple constructions of the same resource field, especially to identify a dependent resource (such as a firewall rule referencing the network to which it belongs in various ways).

The following is true about functionally equivalent HCL configurations:

- A field omitted in only one configuration is set to its default value in the other, as depicted in Example 2-1.
- Two fields that are present in each configuration may appear (even in the execution plan) in distinct ways, as depicted in Example 2-2.

Example 2-1. Two distinct HCL configurations that produce equivalent execution plans

```
#------------------------------------------------------------------
-----
# Version 1: relying on the default value set in the variable
definition
#------------------------------------------------------------------
-----
#
////////////////////////////////////////////////////////////////////
////////
# main.tf
#
////////////////////////////////////////////////////////////////////
////////
resource "google_storage_bucket" "tas_2_1_unset_gce" {
  name     = var.bucket_name
  location = var.location_with_default
}

# variables.tf
variable "bucket_name" {
  description = "This is the name of the bucket."
}

variable "location_with_default" {
  description = "The location for the bucket. This has a default
value set to 'US'."
  type        = string
  default     = "US"
```

```
}

# terraform.tfvars
bucket_name = "bucket-terraform-at-scale-2-1-notset"

#-----------------------------------------------------------------
-----
# Version 2: explicitly setting variable to the same value as the
default
#-----------------------------------------------------------------
-----
#
/////////////////////////////////////////////////////////////////
////////
# main.tf
#
/////////////////////////////////////////////////////////////////
////////
resource "google_storage_bucket" "tas_2_1_set_gce" {
  name     = var.bucket_name
  location = var.location_with_default
}

# variables.tf
variable "bucket_name" {
  description = "This is the name of the bucket."
}

variable "location_with_default" {
  description = "The location for the bucket. This has a default
value set to 'US'."
  type        = string
  default     = "US"
}

#
/////////////////////////////////////////////////////////////////
////////
# terraform.tfvars
#
/////////////////////////////////////////////////////////////////
////////
bucket_name           = "bucket-terraform-at-scale-2-1-set"
location_with_default = "US"     # Setting the variable to its
default value
```

Example 2-2. Two distinct HCL configurations with different execution plans that produce equivalent results

```
#------------------------------------------------------------------
-----
# Version 1
#------------------------------------------------------------------
-----
#
////////////////////////////////////////////////////////////////////
////////
# main.tf
#
////////////////////////////////////////////////////////////////////
////////
resource "google_compute_network" "tas_2_2_vpc_v1" {
  name = "vpc-terraform-at-scale-2-2-v1"
}

resource "google_compute_firewall" "tas_2_2_fw_v1" {
  name    = "fw-terraform-at-scale-2-2-v1"
  network = google_compute_network.tas-2-2-vpc-v1.name

  allow {
    protocol = "icmp"
  }
}

#------------------------------------------------------------------
-----
# Version 2
#------------------------------------------------------------------
-----
#
////////////////////////////////////////////////////////////////////
////////
# main.tf
#
////////////////////////////////////////////////////////////////////
////////
resource "google_compute_network" "tas_2_2_vpc_v2" {
  name = "vpc-terraform-at-scale-2-2-v2"
}

resource "google_compute_firewall" "tas_2_2_fw_v2" {
  name    = "fw-terraform-at-scale-2-2-v2"
  network = google_compute_network.tas-2-2-vpc-v2.self_link
```

```
  allow {
    protocol = "icmp"
  }
}
```

Because of this, functionally equivalent HCL configurations may be drastically different in how they're constructed and maintained. It's therefore up to those developing and maintaining TF projects to agree on conventions, constructions, and style — consider using a style guide such as the Terraform style guide.

Variables in TF

Another major TF component is the input variable. This allows the values assigned to resource fields, for example, to be abstracted away. They may then be gathered together and set, collectively, in a special file type (those with the *.tfvars* or *.tfvars.json* file extension) or passed into any TF command that includes a planning phase. This means the same module (a collection of *.tf* files in a filesystem directory) may produce different outcomes (including different execution plans) with different values set in a *.tfvars* file, on the command line, and so on. They also enable TF modules to be reused and referenced from other modules to follow principles such as single responsibility principle (SRP), don't repeat yourself (DRY) principle, principle of least privilege, etc.

The input variable is therefore an important component to any high-quality TF project. However, input variables are more than just instruments of abstraction: they also introduce useful governance and validation qualities that will be introduced in "Type Constructors: Composite Variable Construction" and "Validation of Variables".

Parts of a Variable Definition

Here are the possible fields of a defined input variable. All of these fields are optional:

Variable description

The `description` field is what it sounds like: it describes the input variable. According to the Terraform documentation, this should generally be phrased from the perspective of an end user. It's up to you what goes into this, but it's a best practice to include a description for every variable, and it's recommended to be consistent with the format across all variable descriptions.

Type specification

The `type` field defines what values may be assigned to the input variable. TF will complain if the `type` field is set and the value assigned to the variable is not expressible as that type (it does automatically convert primitive types such as `'true'` to `true` in the case of a `bool` type, `123` to `'123'` for a `string` type, and so on). Type specification is also especially important for module developers because it improves the experience of a module's consumers by expressing the information it expects.

Default value

A variable's `default` value will be used in the case that the variable is not set in the targeted *.tfvars* file(s) or on the command line or passed in from another module. It must conform to the `type` field, if it is set. Default values are best used for extremely prevalent cases (such as a preferred region, machine type, resource size, etc.) to reduce excessive repetition while still allowing for customization.

Validation nested-block

TF introduces the ability to include further logic to determine whether a value that is supplied to a variable is appropriately constructed, beyond just the value's type, by providing the `validation{}` subblock (also referred to as nested block). A variable's `validation{}` subblock has two required fields: `condition` and `error_message`. The former is a logical expression that must produce a Boolean value; a value is only accepted if this evaluates to true. The latter is an

expression that must evaluate to a string value and will be printed to the command line as part of an error message from TF.

Ephemeral value

By default, variable values are preserved in the TF state object. The values of variables marked with the `ephemeral` field set to `true` will not be preserved in the TF state object. This is available in only HCP Terraform and TFE and is perhaps most appropriate for values that are intended for informative purposes (in that they are not used to create a resource or assigned to some field of a resource) and are computed based on environmental factors (which may not be reproducible). This way, if the variable is subsequently computed and yields a different value, it won't be reported as a drift away from the established infrastructure configuration as preserved in the TF state object. You'll find more on the TF state object in Chapter 7.

Sensitive value

By default, variable values may be printed to the CLI output. Similar to the `ephemeral` field, values of variables marked with the `sensitive` field set to `true` will not be printed to the CLI output. This does not mean that these values are protected in any other meaningful way; other measures must be taken to protect them, such as secret management, access control, and encryption mechanisms.

Nullable value

By default, any variable will allow a `null` value to be assigned to it. Variables that are marked with the `nullable` field set to `false` will require any value set to the variable to have a value of the variable's type (assuming the `type` field is also set), such as the type's unity element.

The minimum configuration for all `variable{}` blocks should include `description` and `type` values. The use of the `validation` field can be especially useful for tightly controlled or

standardized options and formats, but implementing it for every field (perhaps as a written requirement or encoded policy) may err on the side of overengineering, as we'll explore in "Validation of Variables". Setting `default` fields should be relegated to only the necessary situations: those where a variable may be effectively optional (and is set to the type's unity element) or where a very typical or standard value for the variable is available. Similarly, `ephemeral` and `sensitive` fields should be used only where necessary, as most values should be preserved and/or printable in log outputs. The use of `nullable` may be employed to strengthen the variable's type (where made explicit) by restricting a `null` value to be set to the variable.

Setting Input Variables

In terms of how values should be set on variables, it's really a case-by-case basis. Often, the most mature and successful configurations include a mix of the following approaches:

The .tfvars *files*

These are only applicable to the root module (the TF project, targeted by a TF command) and are best for configurations that may change and whose value should be controlled by the module's development team.

Environment variables

These are also only applicable to the root module. They're best for variables that define team-specific, platform-level configuration (such as hosting and account information like a region, subnetwork, or project ID), especially those that change rarely, if ever, and are primarily the responsibility of a team other than the module's developers. To set a TF variable with an environment variable, prefix the variable name with `TF_VAR_` as shown in Example 2-3.

CLI input

It's possible to set variables with a command-line argument (such as for use with the `tf plan` command) with the `-var` flag. This is likely most appropriate for custom automation platform configurations (such as for an OpenTofu project) to override values set in *.tfvars* or environment variables such as in a self-service configuration or for testing a module without making an update in source control.

The `default` *value*

As described previously, `default` values are most valuable (pun unavoidable) in situations where a specific value is preferred or rarely needs to deviate from a typical value. Moreover, these should generally *not* be used in cases where the variable is known to be especially *varied* (as in, many values are possible and no one value has a majority share) or at all *volatile* (as in changing over time, such as a modifiable resource value like a list of network tags). In such cases, the default value is not only a case of overengineering but also opens the module's less astute consumers to the risk of misconfiguration by overlooking a variable that is really necessary to set.

Module fields

This is only applicable to modules that are referenced by other modules (that is, referenced by the `source` field of a `module{}` block) and is the only way to pass information into the nested module from the outer module.

Workspace variables (HCP Terraform and TFE only)

These may be of type Terraform or Environment. The former is like setting the value with the command-line argument, and the latter can also be used like the Environment variables, with the `TF_VAR_` prefix (though it may be best served for other use cases). Additionally, a workspace variable may be directly set as sensitive, which would supersede the setting (or lack thereof) in the `variable{}` block.

Variable sets (HCP Terraform and TFE only)

These are useful for configuration that is shared across multiple workspaces, such as organization-level information. These can also be marked sensitive directly in the HCP Terraform or TFE interface, which is useful for configuring variables that require sensitive information.

Example 2-3. Various examples of setting variable values

```
#
/////////////////////////////////////////////////////////////////
////////
# main.tf
#
/////////////////////////////////////////////////////////////////
////////
module "module_with_fields" {
  source = "third-party-module-tas-2-3"

  module_string_field = "module-string-value"
}

#
/////////////////////////////////////////////////////////////////
////////
# variables.tf
#
/////////////////////////////////////////////////////////////////
////////
variable "set_by_default" {
  description = "This is set by default"
  type        = string
  default     = "default-string-value"
}

variable "set_in_tfvars" {
  description = "This is set in tfvars"
  type        = string
}

variable "set_on_cli" {
  description = "This is set on the command line with the -var
flag"
  type        = string
}
```

```
#
**********************************************************************
********

#
////////////////////////////////////////////////////////////////////
////////
# terraform.tfvars
#
////////////////////////////////////////////////////////////////////
////////
set_in_tfvars = "tfvars-string-value"
```

A typical configuration may include something like the following:

- Project-specific *.tfvars* file with configurations that the development team needs to have control over, possibly referenced by a command-line argument
- Individual environment variables, `-var` CLI arguments, or workspace variables for components that the infrastructure operations team need to specify, such as the targeted deployment environment
- Variable defaults set in remote modules (corresponding to `module{}` block fields), possibly also curated by the infrastructure operations team

Variable Types

As briefly described previously, setting an explicit type on an input variable is a useful way to improve the experience of developing for TF. Instead of guessing how an input variable should be constructed and getting something working through trial and error (or analysis of the dependency chain to the underlying logic and resources), a module's consumers and developers can see what the type should be and construct the input variable values accordingly. Although this is not exactly the most pressing concern for primitive types such as `string` or `bool`, it's especially valuable for

complex types that may be created through type constraint expressions. In cases where the type isn't known or isn't yet consistent, the `any` type is available as a placeholder.

Primitive Types in TF

The *primitive types* in TF are the fairly standard type constructs. Although many of them will be automatically converted between each other (where feasible), it's always best to use the appropriate type and declare an input variable's type field explicitly. Most often, the variables that are being constructed will map to variables for a resource definition or nested module. Assuming the provider or module being leveraged is of high quality, it will express the types for its own variables or fields. These types should be reflected as much as possible in the variables you develop.

The following are the primitives in TF:

`string`

> TF will accept any `string` value. It doesn't distinguish between different string encodings, and supports even very long strings. The empty string, `''`, is regularly used as a unity element for a `string value`. A `number` or `bool` that is assigned to a `string` field or variable will be automatically converted to the string representation of the value.

`number`

> TF numbers can be integers or rational numbers. The zero integer, `0`, is the typical unity element for numbers. A string encoded integer or rational number that is assigned to a number field or variable will be automatically converted to the number representation of that value.

`bool`

> TF `bool` values represent the outcome of Boolean logic, that is, `true` or `false`. The strings `'true'` and `'false'` that are assigned to a

`bool` field or variable will be automatically converted to the corresponding Boolean value.

Explicit references to primitive types are generally found only in `variable{}` blocks, as the value of the `type` field (or a part thereof), as shown in Example 2-4.

Example 2-4. Setting the `type` field of input variables

```
variable "string_typed_variable" {
  description = "a variable with the type field set to string"
  type        = string
}

variable "number_typed_variable" {
  description = "a variable with the type field set to number"
  type        = number
}
```

The unity elements are commonly used as part of the logic to establish conditional elements such as a resource. For example, a `string` type variable could be given the default value of `''` and in the resource definition, a `count` meta-argument is paired with a conditional expression that tests the input variable against this unity element to determine whether it should be applied, as in Example 2-5.

Example 2-5. Leveraging the unity element in a count's conditional expression

```
# Directly using a number
variable "conditional_control_variable_s3" {
  description = "set this to 1 to create the conditional S3
resource"
  type        = number
  default     = 0
}

resource "aws_s3_bucket" "conditional_s3_bucket" {
  count = var.conditional_control_variable_s3

  bucket = "conditional-s3-bucket-example"
}

# Using a ternary expression
```

```
variable "conditional_name_variable_ec2" {
  description = "set this to any valid string to create the EC2
resource"
  type        = string
  default     = ""
}

resource "aws_instance" "conditional_ec2_instance" {
  count = var.conditional_name_variable_ec2 == "" ? 0 : 1

  ami           = var.default_ami
  instance_type = "t2.micro"

  tags = {
    Name = var.conditional_name_variable_ec2
  }
}
```

Type Constructors: Composite Variable Construction

Although it's technically possible to encode all input variables as one of the primitive types, it may not be feasible for a number of reasons. For one, it may be too confusing and repetitive to list each and every field as a separate input variable, especially in cases where the same `resource{}` block is configured over and over with different field values. It may also be impractical if the variables of a nested module expect a more complex variable type.

Type constructors are used in variable type constraint definitions for input variables that are not of type `string`, `number`, or `bool`. The type constructors are:

`list()`

> The `list()` type constructor can be used with any other single type to create a uniform list of values of that type. This is especially useful for constructing input variables that may be iterated over in a `for`/`in` expression.

`set`

This is similar to the `list()` type constructor, but the values within the grouping must all be unique.

`map`

The `map()` type constructor may also be used with any other type, and the resulting variable will expect a string value as the key element of the variable and a value of the supplied type as the value element of the variable. This is also useful in `for`/`in` expressions, especially in conjunction with the `for_each` meta-argument, to ensure unique, nonsequential keys for the resulting state object.

`object()`

The `object()` type constructor is similar to the `map()` type constructor, except you explicitly express the names of all the key elements and you may set the type of the value elements to whatever type you wish (including other, nested composite variables), instead of all value elements being of a uniform type. These can generally be found in situations where multiple related pieces of data are bundled together, such as the fields for a compute disk, for example, block size (`number`) and disk type (`string`), and other fields.

`tuple()`

The `tuple()` type constructor may be used with any number of other types to create a collection similar to a list or set, but with a static length and with differentiated data types. The values in a tuple type are accessed similar to a list or set type, by index.

NOTE

The `tuple` type is generally not worth using over other types, such as `object`, except where the data to be assigned to the value of an input variable is provided by a highly structured automated system that marshals data into this format. In this case, it might be considered too much additional overhead to re-marshal the data into a new format.

Defining these is strongly preferred to leveraging `any` type or not including a type at all, but `object()` constructions in particular can get complex, especially when multiple type constructors are implemented in a nested fashion, such as in Example 2-3. This becomes especially acute when it becomes necessary to include a default value (Example 2-6).

Example 2-6. Complex variables and example default values using nested type constructors

```
variable "nested_object_variable" {
  description = "this variable may be overly complex"
  type        = object({
    string_key     = string
    number_map_key = map(number)
    object_key     = object({
      number_sub_key      = number
      string_list_sub_key = list(string)
      object_sub_key      = object({
        boolean_sub_sub_key    = bool
        string_set_sub_sub_key = set(string)
      })
    })
  })
  default     = {
    string_key     = ""
    number_map_key = {}
    object_key     = {
      number_sub_key      = 0
      string_list_sub_key = []
      object_sub_key      = {
        boolean_sub_sub_key    = false
        string_set_sub_sub_key = []
      }
    }
  }
}
```

Generally, the presence of these functions in `variable{}` block definitions should follow the structure of the module code that operates on the input variable: for example, an expression assigned to the `count` or `for_each` meta-argument requires a `list`, `set`, or `map` type on which to operate. Constructing a module's input variables to align directly to these expressions, rather than requiring inline logic or the use of local values, can

simplify the module's own complexity. However, if the input variable is too complex, it puts the burden on the module's operator (either through a directly issued TF command or by virtue of being a referenced submodule) to establish such a value. If this burden is too high or otherwise disagreeable, the module's input variable in question should be refactored such that the complex value may be constructed in either a local value (preferred) or in an inline expression, as shown in Example 2-7.

Example 2-7. Before and after refactoring an overly complex variable construct

```
#-----------------------------------------------------------------
-----
# Before refactor
#-----------------------------------------------------------------
-----
#
////////////////////////////////////////////////////////////////
////////
# main.tf
#
////////////////////////////////////////////////////////////////
////////
module "module_with_complex_variable" {
  source = "third-party-module-tas-2-4"

  complex_module_field = var.complex_module_variable
}

#
****************************************************************
********

#
////////////////////////////////////////////////////////////////
////////
# variables.tf
#
////////////////////////////////////////////////////////////////
////////
variable "complex_module_variable" {
  description = "This variable might be too complex"
  type = object({
    string_key     = string
    number_map_key = map(number)
```

```
    object_key     = object({
      number_sub_key     = number
      stringList_sub_key = list(string)
    })
  })
}

#-----------------------------------------------------------------------
-----
# After refactor
#-----------------------------------------------------------------------
-----
#
////////////////////////////////////////////////////////////////////////
////////
# main.tf
#
////////////////////////////////////////////////////////////////////////
////////
locals {
  complex_module_local = {
    string_key     = var.string_key_var
    number_map_key = var.number_map_key_var
    object_key     = {
      number_sub_key      = var.number_sub_key_var
      string_list_sub_key = var.string_list_sub_key_var
    }
  }
}

module "module_with_complex_local_value" {
  source = "third-party-module-tas-2-4"

  complex_module_field = local.complex_module_local
}

#
**********************************************************************
********

#
////////////////////////////////////////////////////////////////////////
////////
# variables.tf
#
////////////////////////////////////////////////////////////////////////
////////
variable "string_key_var" {
```

```
  description = "String for the string_key"
  type        = string
}

variable "number_map_key_var" {
  description = "number map for the number_map_key"
  type        = map(number)
}

variable "number_sub_key_var" {
  description = "number for the number_sub_key"
  type        = number
}

variable "string_list_sub_key_var" {
  description = "string list for the string_list_sub_key"
  type        = list(string)
}
```

The alternative approach would be to expose the submodule's overly complex variable as an input variable for the surrounding module (the module that references the submodule), too. While this appears to merely shift the issue, rather than address it, this may be appropriate as long as it doesn't introduce repeated information. If the same variable may be exposed all the way to the root module, without introducing repetition of information, it may result in an overall more maintainable project than if excessive data manipulation is encoded in HCL (Example 2-8).

Example 2-8. Exposing an overly complex variable construct to the root module

```
#-----------------------------------------------------------------
-----
# Root module
#-----------------------------------------------------------------
-----
#
////////////////////////////////////////////////////////////////
////////
# main.tf
#
////////////////////////////////////////////////////////////////
////////
module "module_with_complex_variable" {
  source = "../submodule"
```

```
  complex_module_field = var.complex_module_variable
}

#
*****************************************************************
********

#
/////////////////////////////////////////////////////////////////
////////
# variables.tf
#
/////////////////////////////////////////////////////////////////
////////
variable "complex_module_variable" {
  description = "This variable might be too complex"
  type        = object({
    string_key     = string
    number_map_key = map(number)
    object_key     = object({
      number_sub_key     = number
      stringList_sub_key = list(string)
    })
  })
}

#---------------------------------------------------------------
-----
# Nested module
#---------------------------------------------------------------
-----
#
/////////////////////////////////////////////////////////////////
////////
# variables.tf
#
/////////////////////////////////////////////////////////////////
////////
variable "complex_module_field" {
  description = "This variable might be too complex"
  type        = object({
    string_key     = string
    number_map_key = map(number)
    object_key     = object({
      number_sub_key     = number
      stringList_sub_key = list(string)
    })
```

```
  })
}
```

The any Type

In addition to the primitive types and those created through type constructors, TF also supports an `any` type. This is effectively the same as omitting the type altogether. However, it may be an appropriate alternative when a type is a required component but the type's format isn't fully developed or stable.

The `any` type may be referenced in the same way a primitive type would be and can therefore be present within a larger type construct, as depicted in Example 2-9. However, it's not recommended to nest an `any` type within a larger type construct except as an intermediate step to developing a fully specified type.

Example 2-9. Using the `any` *type within a type constructor (to be avoided)*

```
variable "object_variable_nested_any" {
  description = <<-EOT
    including the any type within a larger object variable obscures
appropriate
    values
  EOT
  type = object({
    stringKey      = string
    keyWithAnyType = any # can be set to any primitive or JSON-like
value
  })
}
variable "map_any_variable" {
  description = <<-EOT
    including the any type within a map variable requires all
values to be the
    same (valid) type, but otherwise no other constraints
  EOT
  type = map(any) # the values must still have a consistent type
}
```

Using the type() Function to Determine Type

The `type()` function — which is available only when using the `tf console` shell — can be used to determine the shape of a particular value. This way an input variable with the `any` type (or no type field set) may, for example, be printed to a command line in its full type specificity, as depicted in Example 2-10. This can then be copied to the clipboard and pasted directly into the input variable's type field (possibly generalizing the collection type, if appropriate), thereby fully specifying the variable's construction.

Example 2-10. Using the `type()` *function to ascertain the fully specified type of a value*

```
#
////////////////////////////////////////////////////////////////////
////////
# variables.tf
#
////////////////////////////////////////////////////////////////////
////////
variable "any_variable" {
  description = "This variable accepts any value"
  type        = any
}

#
********************************************************************
********

#
////////////////////////////////////////////////////////////////////
////////
# terraform.tfvars
#
////////////////////////////////////////////////////////////////////
////////
any_variable = [
  {
    key_one = "valueOne1"
    key_two = 1231
  },
  {
    key_one = "valueOne2"
    key_two = 1232
  }
]
```

Validation of Variables

In theory (and very much in practice, one hopes), you have good software delivery controls around your TF code, including legitimate (as in, not just rubber-stamped) code reviews. Furthermore, there would (again, in theory) be well-defined policies, perhaps as a combination of written directives published internally for reference by engineers and codified constraints enforced (if possible) within the CSP. In a perfect world, such directives would be fully respected and such constraints would be exhaustive.

However, in the real world, vulnerabilities get overlooked and new ones emerge often enough to require a certain amount of overhead. Implementing a validation expression for certain input variables may be one component of a *defense-in-depth* strategy by imposing conditions that go beyond the type or shape of the variable's value: to furthermore ensure they fall within a specific range or set. This also provides a level of self-documentation to the variable because it can be read by the engineers to learn what values are appropriate. The `validation` field, while appropriate in many scenarios, is most effective in the case of a module produced such that a third party would consume it (a remote module referenced using a `module{}` block's `source` field).

It's also worth noting that the variable's validation is checked at the planning phase, rather than the apply phase, so that out-of-policy values may be caught before any new (and potentially erroneous) configuration is established, which may have to be subsequently rolled back. If a `tf plan` command is used as part of a pull or merge request, this also means issues may be caught before the errant code is committed to a branch designated for the execution of a `tf apply`, potentially saving hours of unnecessary build time (not to mention the headaches of reverting commits) over the course of a year.

> **TIP**
>
> A defense-in-depth strategy or posture is important for any organization leveraging cloud services. In most cases, a combination of cloud resources and third-party tools is necessary to cover all common vulnerabilities. Once all such vulnerabilities are covered by some combination of tools and resources, it may still be possible and even advisable to implement additional controls that effectively overlap in function. For example, you may implement firewall policies, service perimeters, and forwarding rules that all restrict inbound network traffic from certain geographical regions or IP ranges. However, there are diminishing returns (left as an exercise for the reader) to implementing too many such measures. For one, there's the risk that a particular tool or service has a zero-day vulnerability lurking beneath its surface, so layering on additional components, ad infinitum, will end up like Zeno's arrow, never reaching its target of perfect coverage (this would also obviously not be especially cost effective). Additionally, there's always the risk that a measure was too strict, preventing legitimate business practices, and additional policies, exceptions, or conditions must be introduced to now allow the prohibited behavior.

Custom Conditions

The `validation{}` block of a variable requires a `condition` field that may be assigned any valid logical expression that results in a Boolean value (`true` or `false`). Typically, this would include some evaluation of the value assigned to the variable in which this `validation{}` block is configured. However, there is really no restriction on what the expression can reference, and thus it can also leverage other available values. This involves multiple variables that may be useful for applying more complex conditional expressions, as depicted in Example 2-11, but it should be used sparingly.

Example 2-11. Validation condition expression referencing other variables

```
variable "special_s3_type" {
  description = "controls how the dependent_s3_bucket is validated"
  type        = bool
  default     = false
}

variable "dependent_s3_bucket" {
  description = "has validation constraints based on another
variable"
  type        = string
```

```
  # the bucket should start with "spc-" or with "crp-", depending
on the
  # Boolean special_s3_type variable
  validation = {
    condition     = (var.special_s3_type ?
      (startswith(var.dependent_s3_bucket, "spc-"))
      : (startswith(var.dependent_s3_bucket, `"crp-")))
    ...
  }
}
```

More common, and more widely applicable, is an expression that focuses exclusively on the variable to which the expression is assigned. For example, it would be appropriate to include a condition on a `string` type variable that restricts its value by length or requires it to include a specific substring, perhaps by matching it against a regular expression. This is common for variables that are assigned to certain resource fields used for identification purposes, such as names or tags, since certain characters and excessively long strings may be disallowed by the CSP. It's also useful for communicating and enforcing standards that your organization develops and imposes, such as requiring a name to be comprised of an ordered set of corporate-internal code words representing business units, localities, technical domains, environments, and application names, similar to Example 2-12, or establishing a minimum and maximum number of nodes in a managed Kubernetes cluster, such as in Example 2-13.

Example 2-12. Enforcing string standards through the `validation{}` *block's* `condition` *field*

```
variable "s3_bucket" {
  description = "validation encodes CSP constraints and corporate
standards"
  type        = string

  # The first 4 constraints come from S3 specifications.
  # The latter 5 constraints are theoretical and impose additional
rules
  # such as disallowed characters, a prefix, and required
substrings
  validation {
    condition = ((length(var.s3_bucket) > 2 &&
```

```
length(var.s3_bucket) < 64)
      && (length(regexall("^[a-z0-9-]+$", var.s3_bucket)) > 0)
      && (length(regexall("^[a-z0-9]", var.s3_bucket)) > 0)
      && (length(regexall("[a-z0-9]$", var.s3_bucket)) > 0)
      # some CSP constraints above, corporate constraints below
      && (length(regexall("\.", var.s3_bucket)) == 0)
      && (length(regexall("^(pub|prv)(-[a-z0-9]+)*$",
var.s3_bucket)) > 0)
      && (length(regexall("^[a-z0-9]+-(na|eu|ap)(-[a-z0-9]+)*$",
var.s3_bucket)) > 0)
      && (length(regexall("-", var.s3_bucket)) > 1)
      && (length(regex("^[a-z0-9]+", var.s3_bucket)) > 2)
      ...
    )
  }
}
```

Example 2-13. Enforcing topology standards through the `validation{}` *block's* `condition` *field*

```
variable "eks_cluster" {
  description = "EKS cluster configuration variable"
  type        = object({
    cluster_name   = string
    nodes_per_zone = number
  })

  validation {
    condition = (var.eks_cluster.nodes_per_zone > 1
      && var.eks_cluster.nodes_per_zone < 6)
    error_message = "nodes_per_zone must be between 2 and 5,
inclusive"
  }
}
```

Finally, for variables with a composite type, the `validation{}` block's `condition` field may be used to check multiple subcomponents and aspects of the variable by stitching multiple expressions together with logical *ANDs* (`&&`). An example is a variable that encodes a list of configuration packaged together and used for the fields of a `module{}` block to construct a collection of firewall rules, and may be constructed using the `list()` and `map()` constructors, the latter of whose values are of the `string` type. The `condition` value of such a variable may test

the direction and type fields against their allowed values (allowing only `"ingress"`/`"egress"` and `"allow"`/`"deny"`, respectively), whereas the range value(s) may have a more complex string-matching expression that tests against a regular expression (regex) or substring, such as in Example 2-14.

Example 2-14. Validation condition for a composite type variable

```
variable "aws_network_acl_list" {
  description = "Access Control List configurations"
  type        = list(map(string))

  validation = {
    condition = (length(var.aws_network_acl_list) > 0 ?
      (var.aws_network_acl_list[0]["direction"] == "ingress"
         || var.aws_network_acl_list[0]["direction"] == "egress")
      && (var.aws_network_acl_list[0]["action"] == "allow"
         || var.aws_network_acl_list[0]["action"] == "deny")
      : true)
    error_message = join(" ", [
      "Each item in the aws_network_acl_list",
      "should have the 'direction' and 'action' keys"
    ])
  }
}
```

Similarly, for a variable that is constructed using the `object()` constructor, the various variable subcomponents may each have a unique subexpression that tests its validity. This may become especially important the more complex a variable's construction gets, to ensure all the values within the variable are constructed appropriately as early as possible. For example, if the firewall configuration described previously is embedded as part of a larger, more holistic network module, the example variable construct may manifest as a subfield of the variable, as specified in the variable's type constructor. The validity can be based on the aggregate validation across all fields, however complex each field is, as shown in Example 2-15.

Example 2-15. Validation of a complex composite variable that uses the `object()` *constructor*

```
variable "ntwrk_info" {
  description = "Contains several configurations for a VPC"
  type = object({
    vpc_region = string
    acl_list        = list(map(string))
    ...
  })
  validation = {
    condition = ((contains(["us-east-2", "us-west-2"],
var.ntwrk_info.vpc_region))
      && (length(var.ntwrk_info.acl_list) > 0 ?
          (var.ntwrk_info.acl_list[0]["direction"] == "ingress"
            || var.ntwrk_info.acl_list[0]["direction"] == "egress")
          && (var.ntwrk_info.acl_list[0]["action"] == "allow"
            || var.ntwrk_info.acl_list[0]["action"] == "deny"))
      : true)
    error_message = join(" ", [
      "The region must be 'us-east-2' or 'us-west-2' and each item
in the",
      "acl_list should have the 'direction' and 'action' keys"
    ])
  }
}
```

Error Messages

The variable `condition{}` block's `error_message` field is also a required field and allows custom error messages to be printed to the CLI output if a validation fails. Generally, this should be expressive enough to explain both why the value failed the variable's validation condition and what the variable expects for its component values, as shown in Example 2-16. This is especially true for modules that may be used by third parties, since it's possible to construct intermediate TF modules that obscure (either intentionally or not) their inner workings, including any references to such third-party modules, as shown in Example 2-17. In such cases, the third-party module would at least express how the intermediate module is failing, enabling a more detailed troubleshooting experience.

Example 2-16. Error message that expresses what the validation condition expects

```
variable "special_s3_type" {
  description = "controls how the dependent_s3_bucket is validated"
```

```
  type        = bool
  default     = false
}

variable "dependent_s3_bucket" {
  description = "has validation constraints based on another
variable"
  type        = string

  # the bucket should start with "spc-" or with "crp-", depending
on the
  # Boolean special_s3_type variable
  validation {
    condition     = (var.special_s3_type ?
      (startswith(var.dependent_s3_bucket, "spc-"))
      : (startswith(var.dependent_s3_bucket, "crp-")))
    error_message = join(" ", [
      "since special_s3_type is ${var.special_s3_type},
dependent_s3_bucket must",
      "begin with \"${var.special_s3_type ? "spc-" : "crp-"}\""
  ])
  }
}
```

Example 2-17. Error messages due to failed condition within a nested third-party submodule

```
#-----------------------------------------------------------------
-----
# Root module
#-----------------------------------------------------------------
-----
#
////////////////////////////////////////////////////////////////
////////
# main.tf
#
////////////////////////////////////////////////////////////////
////////
module "tas_2_7_intermediate_module_module" {
  source = "../intermediate-module-module"

  # Both will produce validation errors in the third-party module
  string_key  = "camelCaseString"
  number_list = [42, -2, 0, 13]
}

#-----------------------------------------------------------------
```

```
-----
# Intermediate module
#-----------------------------------------------------------------
-----
#
/////////////////////////////////////////////////////////////////
////////
# main.tf
#
/////////////////////////////////////////////////////////////////
////////
module "tas_2_7_third_party_module" {
  source = "../third-party-module"

  module_field = {
    string_key    = var.string_key
    digit_map_key = { for val in var.number_list : "item-${val}" =>
val }
  }
}

#
*****************************************************************
********

#
/////////////////////////////////////////////////////////////////
////////
# variables.tf
#
/////////////////////////////////////////////////////////////////
////////
variable "string_key" {
  description = "This is a string to be used in the third-party
submodule"
  type        = string
}

variable "number_list" {
  description = "A list of numbers to be used in the third-party
submodule"
  type        = list(number)
}

#-----------------------------------------------------------------
-----
# Third-party module
#-----------------------------------------------------------------
```

```
-----
#
////////////////////////////////////////////////////////////////////
////////
# variables.tf
#
////////////////////////////////////////////////////////////////////
////////
variable "module_field" {
  description = "This variable is developed by a third party"
  type = object({
    string_key    = string
    digit_map_key = map(number)
  })
  validation {
    condition = (regexall("^[a-z]+$", var.module_field.string_key)
      && alltrue([
        for val in values(var.module_field.digit_map_key)
        : parseint(val, 10) > 0
      ])
    )
    error_message = join(" ", [
      "string_key must be lowercase letters and the values",
      "of the digit_map_key must all be positive digital integers"]
  }
}
```

If a module is intended for internal consumption, it may be efficient for a `validation{}` block's `error_message` expressions to print out a link to an internal web page that provides greater detail on the constraints established for the variable. This can be a good way to reduce repetition in the case where the same validation constraints are used in multiple places (although, the constraint expression itself would still need to be replicated, and thus maintained, in each place). Similarly, for constraints that are publicly available, such as character length limits on a resource's name or ID field, a link to the relevant public documentation, as shown in Example 2-18, would be considered a good practice.

Example 2-18. Including a link to public documentation corresponding to a variable's constraints

```
variable "s3_bucket" {
  description = "bucket with CSP constraints and link to the
documentation"
```

```
  type        = string

  validation {
    condition     = (length(var.s3_bucket) > 2 &&
length(var.s3_bucket) < 64)
      && (length(regexall("^[a-z0-9-]+$", var.s3_bucket)) > 0)
      && (length(regexall("^[a-z0-9]", var.s3_bucket)) > 0)
      && (length(regexall("[a-z0-9]$", var.s3_bucket)) > 0)
      ...
    error_message = join(" ", [
      "Must follow AWS bucket naming rules. See",
      join("/", [
        "https://docs.aws.amazon.com/AmazonS3/latest",
        "userguide/bucketnamingrules.html"
      ]),
      "for more details."
    ]
  }
}
```

Effective Use of the validation{} Block

As described, the `validation{}` block can be a great way to enhance your defense-in-depth posture and to provide valuable information to the consumers of your module. Also, as presented, it's likely a worthwhile component for any highly restricted or specific values leveraged in a module that's intended for third-party consumption. This way, the module's consumers will be alerted to any well-known constraints as soon as possible.

Similarly, for restrictions that are defined by the organization as part of a deliberate policy, standard format, or support posture, it should be considered best practice to establish corresponding `validation{}` blocks. Ideally, this would be mandated by the very policy or standard to which it adheres (just as a web page for the policy or standard may be linked to in the corresponding `error_message`, as described previously), thereby tying the accountability of such validation to the team responsible for defining the policy/standard.

The `validation{}` block does introduce its own level of complexity and can become overly cumbersome if employed thoughtlessly and with a

draconian stance. Similar to the commentary regarding the `any` type, requiring a `validation{}` block for every variable may not be the most efficient approach during the early stages of a variable's development, especially in cases where the constraints are not fully established or well known. Instead of implementing `condition` and `error_message` expressions that are overly general, or overly nonexhaustive, it may be more prudent and pragmatic to implement validations as part of a quality specification strategy, thereby encoding validation as issues come up. This way, while issues may make it all the way to the application phase (requiring TF to perform a rollback operation), appropriate measures may be encoded in the `validation{}` block to prevent the same issue from getting that far again.

Moreover, the expression assigned to the `validation{}` block's `condition` field is not something that may be easily debugged if it's incomplete or insufficient to accurately validate an input variable. In cases of variables with excessively complex composite types, such as those built with the `object()` constructor, the complexity of the constraint expression compounds with each value in the variable. This would also expand the `error_message` field, as the variable could fail validation in a number of different ways. In these cases, it may be most effective to take an approach that loosely aligns with the Pareto Principle, with the goal of catching most of the problems (and certainly, the most common or costly problems), rather than fully mitigating all of them.

When constraints are not well defined, and thus a complete constraint may be constructed only by happy accident, trial and error (or, as my high school math teacher would say, "brute force and ignorance") may be the only way to discover the relevant limitations. This is where the flexibility of TF modules can become irksome, as modules may be built from other modules and low quality of any one module, however nested or removed, may be ruinous for an entire project.

Things to Avoid

Variables in TF are optional components, but they can become valuable tools if included effectively. Unfortunately, they can also be established in ways that limit, corrupt, or outright negate their effectiveness. The following practices should be considered anti-patterns and bad practice.

Empty variable{} Blocks

The `variable{}` block actually has no required fields. It can therefore be constructed as an empty block, as shown in Example 2-19. Unfortunately, this might be the most common use of variables, and it's also clearly the most lazy.

Example 2-19. Depiction of an empty `variable{}` *block*

```
variable "empty_variable" {}
```

Certainly, at a very early stage of a variable's development (that is, when it's first encoded into a module), it may be fine and natural to establish an empty `variable{}` block as a placeholder. However, it's just as certainly a bad practice to include this into configurations that support your organization's workloads. There is no good reason to maintain a variable that doesn't specify an explicit type and doesn't include a sufficient description.

Overusing the any Type

As previously stated, using the `any` type is effectively the same as omitting an input variable's `type` altogether. There are very few (good) reasons to include the `any` type in mature codebases. Even where a `type` is required by an agreed upon (or handed-down) policy, using the `any` type becomes a workaround that undermines the policy, anyway.

> **TIP**
>
> If this is the situation you find yourself in, hopefully you have some other policy that is raising a noticeable warning (that is, one not buried in textual log output) for when the `any` type appears in the codebase. Ideally, setting a variable's type to something definitive should be included as part of the normal development process, either as a final step or as a follow-up task. If it isn't, it should, at the very least, be treated like technical debt, and paying the debt down may be included as part of typically scheduled work (it should not be left ignored for months).

During early stages of a module's development, it's normal, and may even be highly appropriate, to leverage the `any` type in variable definitions (especially in environments that require a type to pass validations). This may be due to a need for fluidity in a complex variable's shape while fields are iteratively updated to account for the module's needs. However, leaving `any` as the type, especially in configuration supporting the infrastructure running your production workloads, is risky to say the least. Once a structure is solidified for the variable, define it using the more specific type functions. Any modifications thereafter will be all the better tracked, expressing very granular changes to the shape of the variable.

You certainly should not use the `any` type to hide, ignore, or relax the complexity of your input variables, whether it be from the root module (the one to which a `tf plan` command is targeted) or a nested module. This might be an attractive option in the case where the data being supplied to the input variables is generated by some software process or compiled from environmental data (either of which might be corrupted in some way), and you cannot control how the data is constructed. However, it's better to prepare your input data outside of the current module in a deterministic fashion before supplying it as values to input variables so that it's in the correct shape for consumption by the module.

Overusing Composite Variable Types

Speaking of constructing proper variable types, it's best to avoid overuse of composite variables. Unless there's an expectation that a particular variable

will be leveraged in some collection, it's often unnecessary to package several values into the same variable. If, for example, you're configuring several firewall rules, all with unique name, range(s), and description values passed in as inputs, then it likely makes sense to package the fields into a single variable, with a well-defined type (using one of the type constraint functions).

However, if you're setting up a unique configuration for the compute resources of a special security appliance in this module, it's unwise to package all such necessary resource fields into a single variable. Constructing such a complex composite type for a variable vastly reduces clarity, as seen in Example 2-20.

Example 2-20. Overusing composite variable constructs

```
variable "special_ec2_info" {
  description = "All the information necessary for the special
instance."
  type        = object({
    ami               = string
    instance_type     = string
    subnet_id         = string
    core_count        = number
    threads_per_core = number
    tags              = map(string)
    ...
  })
}

resource "aws_instance" "special_ec2_instance" {
  ami           = var.special_ec2_info.ami
  instance_type = var.special_ec2_info.instance_type
  subnet_id     = var.special_ec2_info.subnet_id
  tags          = var.special_ec2_info.tags
  ...

  cpu_options {
    core_count       = var.special_ec2_info.core_count
    threads_per_core = var.special_ec2_info.threads_per_core
  }
  ...
}
```

If you're leveraging a nested module, it may seem natural to package all of the module's variables into a composite variable. This may indeed be appropriate if the nested module itself will be used in a collection (that is, leveraging the `for_each` meta-argument) or conditional (leveraging the `count` meta-argument) fashion. It also may be appropriate in cases where the nested module's variables are fairly simple (such as all primitive types) and flat, or when multiple nested modules are used that have similar input variables (such as data about a disk attachment or a firewall rule that could be used in different contexts for different modules and must be sufficiently distinct). Otherwise, it's typically best to mirror the nested module's variables as variables in the outer module.

Mixing Essential and Nonessential Data in Composite Variables

Speaking of composite variables, it is, as discussed, a natural instinct to package multiple values together that are intended for use as module field values. Especially for fields that could logically be grouped together, or for a collection of such modules (either of which suggests a strong relationship among the group), it becomes attractive to bundle the values into a single, JSON-like object that may be deconstructed together as a unit. However, this can become messy if the module variables that these object values represent have a `default` value. This is because the `default` value effectively renders the module's variable as optional, since omitting a value from a referenced module's field will use the `default` value. Including a corresponding value for such a module field in a composite variable that supports the module and itself has a `default` value necessitates reflecting the variable's `default` value in the composite variable's `default` value (Example 2-21). If the module's variable default changes, it will require the corresponding changes in any such variables supporting it.

Example 2-21. Composite variable supporting module with optional variable fields

```
#------------------------------------------------------------
-----
```

```
# Root module
#------------------------------------------------------------------
-----
#
////////////////////////////////////////////////////////////////
////////
# main.tf
#
////////////////////////////////////////////////////////////////
////////
module "tas_2_8_submodule" {
  source = "../submodule"

  submodule_string_key  = var.submodule_variable.string_val
  submodule_number_list = var.submodule_variable.number_list_val
}

#
****************************************************************
********

#
////////////////////////////////////////////////////////////////
////////
# variables.tf
#
////////////////////////////////////////////////////////////////
////////
variable "submodule_variable" {
  description = "value"
  type = object({
    string_val      = string
    number_list_val = list(number)
  })
  default = {
    string_val      = "whatShouldThisBe?"
    number_list_val = []
  }
}

#------------------------------------------------------------------
-----
# Nested module
#------------------------------------------------------------------
-----
#
////////////////////////////////////////////////////////////////
////////
```

```
# variables.tf
#
////////////////////////////////////////////////////////////////////
////////
variable "submodule_string_key" {
  description = "This is a string variable"
  type        = string
}

variable "submodule_number_list" {
  description = "This is a list of numbers (optional)"
  type        = list(number)
  default     = []
}
```

Similarly, it also requires any `validation{}` block `condition` fields provided in a third-party module to be reflected in the referencing context. When condition expressions are already complex, it will further complicate the expression used by any composite variables in the referencing context intended to leverage this module. Because these expressions are already difficult to troubleshoot, evaluating multiple complex expressions at once only complicates the circumstance.

Misusing default Values

As suggested previously, the use of `default` variable values should be reserved for cases where a different value is unlikely in most situations. This is perhaps the most appropriate use of the `default` value (indeed it's the closest to the spirit of the field), however, it may also be appropriate in cases where the meta-arguments `count` or `for_each` reference a variable directly to ensure at least the corresponding type's unity element.

Because of its nature, it may seem reasonable to leverage `default` values in cases where you would normally want to hardcode a value. However, because these are still variables, the value can be overwritten by anyone using the module (either as a root project module or a remote module referenced in a `module{}` block), so there is no protection for the value. Any attempt to protect these `default` values with additional overhead (such as *git hooks* that warn a user before committing code, or even that

prevent such commits altogether) risks becoming an arms race as new hardcoded variables are included. A better approach might be to leverage the local value TF feature, as discussed in detail in Chapter 4.

Summary

In this chapter, you learned the following about Terraform input variables:

- Their component elements and how they're established
- The various ways values are set to input variables
- Variable types and how they're designated and constructed
- The `any` type and how to check a value's type
- How to validate the values assigned to variables before the apply phase
- How to provide detailed feedback in instances where a value assigned to a variable does not match specification
- How to balance a composite variable's compactness with the complexity of its validation condition

You've now seen several key considerations for properly establishing input variables in Terraform code for various needs. You also understand how the features may be used (or not) to best include flexibility without overengineering or introducing excessive overhead. You've encountered several examples of well-designed `variable{}` blocks and a few examples of constructions that were less than high quality.

In Chapter 3, you'll venture through the wild world of TF logic: functions, expressions, operations, and more. You'll see what logical constructs to use, where to use them, and how to avoid unnecessary headaches from haphazard logical statements.

Chapter 3. Effective Use of Logic

TF is a platform that abstracts the APIs of service providers and other platforms. Although the HCL, the language in which TF code is written, is primarily a configuration language, there are logic expressions available in HCL allowing developers to introduce inferences and abstractions, improving the flexibility (or general accessibility) of a solution, without introducing repetitive code or requiring deep technical knowledge on behalf of a module's consumer. However, there is a well-established tendency to see all the world's problems as a collection of nails once one has mastered the hammer, and this can lead a team into a quagmire of configuration as all complexity is shoved into HCL, regardless of the means. At the same time, a draconian adherence to a set of traditional, sacrosanct HCL expressions can force teams into developing unreasonably large and repetitive codebases or into splitting a team's overall state into a vast wilderness of TF state trees.

In this chapter, you'll get an overview of the logical operations and expressions available in HCL, effective use thereof, and other ways to incorporate programmatic logic and functionality into TF solutions. By the end of the chapter, you'll be comfortable with the safe application of the most useful HCL expressions and operations, as well as the effective use of ancillary or third-party components to ensure a balance among the logic internal to, operating upon, and entirely external to the TF codebase. You'll also be able to evaluate the risks associated with particular applications of specific logic and to determine how to improve risky portions of your codebase.

Logic and Terraform

HCL supports a wide variety of common expressions and operators. However, as mentioned, HCL is primarily a configuration language meant for establishing, tracking, and configuring cloud and platform resources and for creating templates thereof. Because of this, there are certain logic elements in HCL that may manifest or behave in unexpected ways.

For example, the functions in HCL are operated in a functional programming style, which is esoteric and, if constructed haphazardly, can be overly convoluted. This is especially acute given that the majority of engineers who find themselves writing TF modules are often much more comfortable with an object-oriented (Java, JavaScript), procedural (C), or scripted (bash/PowerShell) paradigm.

A common approach to this is to actually retreat: to remove as much logic from the HCL code as possible and implement any logic in ancillary operations (leveraging the paradigm of choice). This leads to additional problems as logic is peppered throughout pipeline configurations and Dockerfiles, or implemented by a complex web of manually driven procedures.

Terraform Functions and Expressions

One of the most significant features of the functional programming paradigm is how it stores the results of a calculation: specifically that it doesn't, by default. Rather than requiring the result of every logical statement to have some way to reference it, statements can be combined such that the execution result of one is immediately treated as the input value of another and otherwise forgotten. This means the inner statement's results are not preserved beyond the execution of the outer statement — in theory, reducing its impact on memory utilization. However, this benefit to memory utilization comes at the expense of code readability and reasoning.

That being said, because HCL is also a configuration language, it must preserve certain values as they are assigned to the `resource{}` and `module{}` block argument fields. It also introduces other internal objects, such as the local object defined by the `locals{}` block, which can

preserve arbitrary values, as needed. The general recommendation is to preserve the results of certain functions within this local object in order to reference them later on.

FUNCTIONAL PROGRAMMING

If you're familiar with functional languages like those in the Lisp family, Haskell, and others, it shouldn't be too foreign a concept to reason through function calls in Terraform. If you're more familiar with the much more limited approach of many object-oriented languages to *lambda functions* (whereby pure functions are passed as arguments to specialized methods of collection types that iterate over the collection, sequentially calling said function on each iteration), the nesting behavior of inline function chaining in the functional programming paradigm may be both awkward and difficult to reason through.

The general guidance is to externalize as much of the more complex logic as possible and thus reduce or minimize the use of HCL functions. However, if you do find yourself working in a TF codebase that heavily leverages functions for calculating values assigned to resource arguments, it's recommended to pick up a book on the subject. (There are some famous ones that are worth reading even simply for pleasure, such as *The Little Schemer* by Daniel P. Friedman and Matthias Felleisen (MIT Press).

Automated Logic Outside of TF

TF will most likely be embedded in your automation tool's DSL, even if it amounts to YAML and free text bash scripts. This unlocks the breadth of tools (or lack thereof) available to the execution context. In many popular platforms, these execution contexts are already containerized and even allow for custom container images to be leveraged for a job's execution. This opens a world of possibilities to incorporate tools other than TF to perform operations on inputs and outputs.

> **NOTE**
>
> This will get messy if you're leveraging inline `sed`/`awk` commands or pipes in conjunction with the TF commands within the context of a shell.

Instead, consider using a friendlier scripting language such as Python or JavaScript if you have a lot of fancy logic to expand, mask, or enrich any inputs or interpret any outputs with code. Such code can be tested and stepped through with a debugger in ways that TF cannot. This logic can be executed beforehand, generating or preconfiguring a TF project's input variable assignments, or afterward, interpreting and translating the output values of a `tf apply` command. It can even be expanded to drive the execution of TF commands using the Cloud Development Kit (CDK) and related tooling, which, with a serviceable UI, could achieve a fully self-service and/or event-driven solution, reducing the need for fully custom automation engineering.

Manual Processes

Before a rich service portfolio can be constructed, there will very likely be countless manual procedures required to operate the various projects that make up a full enterprise TF ecosystem. These might involve managing TF project dependencies (such as copying the output values of one module and pasting it into the input variable assignments of another), calculating input values or interpreting output values, or remediating issues and problematic incidents.

Manual processes have their place, especially in early exploratory work, before robust operation software has been developed. Especially when the provider has convenient configuration wizards that walk a user through configuration, manual configuration can be useful to learn what all components a typical solution may encompass, allowing you to then render those components in HCL code. The challenge here is to ensure you're capturing all the (sometimes many) components that are also configured in the background when certain resources are created this way — for example,

the boot disk that conveniently gets created along with a compute resource — and are tracked as unique and distinct resources in most cloud providers. System access logs can be very helpful in diagnosing this.

Because of the interconnectedness of cloud resources, and because it rarely makes sense to put all configuration of your organization's cloud infrastructure in the same project (for example, repository and TF state), there will surely be dependencies across projects. This will sometimes include *cascading changes* (a change in one project precipitates a required change in another project) and these cascading changes may be time-consuming and complex to fully automate away. In the end, certain manual processes, due to project dependencies will likely stay manual. More on project dependencies, in Chapter 7.

The main problem with manual processes is they can be highly error prone. One typo or out-of-order option in the CLI command, and suddenly you're getting drastically different results than you expect. The cloud provider consoles are rather effective at putting up guardrails, keeping users from making overtly inappropriate or outright disallowed operations and configurations, but just because you can do something doesn't mean you should. Almost worse than getting an error when you know what you're doing is getting a working solution through blindly making selections in a configuration wizard. Unless you have a photographic memory (or permission to read the system access logs and are prepared to recreate the steps), it may be cavalier to believe that a repeat occurrence of successful operation is any more than coincidental.

The other problem is manual processes scale poorly, more or less linearly up to muscle memory–related speed improvements, and arguably are the worst out of most available methods. Depending on the provider, adding a permission to 100 groups one by one in a cloud provider's web console takes 100 times longer than it takes to configure one group. Provisioning cloud services in the command line may be much faster, especially if you're skillful with shell scripting, but if you aren't hardcoding the values and instead are building some kind of templating system and, say, storing it in a shared location for reuse, you're basically doing simplified infrastructure as

code. Why reinvent the wheel? If you need to do something once or rarely (and you don't have a wildcard job as described in Chapter 7), it may be appropriate to do it manually, but if you're doing it often and repetitively, it only makes sense to encode the procedure in some way.

> **NOTE**
>
> There may be corporate or legal policies that dictate certain manual operations. For example, it's typical in some industries to require that releases to production be initiated by a designated release manager, so that there is assumed responsibility and liability by someone with ostensibly sufficient experience and discernment. In such cases, you can integrate countless validation and reporting measures over endless efforts to make it foolproof, thereby effectively reducing the level of acumen such a role requires to interpreting a dashboard and pressing one of two buttons: accept or reject. However, doing so undermines the spirit of the policy it's intended to support, effectively to the same degree as automating it (with perfectly implemented best practices) would. Sometimes the only way to implement a fully automated solution is to first modify the applicable policies. That this is easier said than done may not be lost on anyone.

As your systems mature, you should begin to encode most, if not all, manual processes into programmatic procedures that may be triggered at will or automatically in response to a particular event. Deciding which procedures to encode first is most likely neither trivial nor deterministic. Most likely, there will be manual procedures that are considered computationally simple to perform — some that arise more commonly than others, and some that are more time consuming (requiring many person-hours) or time sensitive (risk of issue directly related to time taken to perform) than others. Certainly, any procedures that fall into all three of these categories (regular, protracted, and tedious) would be primary candidates to encode, followed by those in the intersection of any two. Beyond that, it's up to you and your organization to decide the relative importance of each of these dimensions and to determine the order in which manual operations are codified. Do you close out all procedures in a general order along a particular dimension, or do you take a tiered approach across all dimensions, closing out, for example, the easiest tier across all before moving to the next easiest tier? Your answers will be unique to your

organization, its policies, and the availability (and saturation) of technical skills.

The Messy Middle

When TF is first adopted, there may only be one or a few teams working with it, meaning that variation in structure and convention is also likely to remain low due to the collective nature of development. As operations advance, however, it's highly probable that many more teams will become responsible for various parts of the overall codebase, inviting much more variation. While it may seem simple enough to document or templatize successful patterns and keep the IaC masterminds centralized, producing perfect modules that enable more and more capabilities in a safe and effective way, it's very likely that pioneering techniques or competencies will be born from the teams developing the most complex solutions, likely far abstracted from such a central unit of command.

As you've seen, and as with any software development, there are numerous (if not innumerable) ways to accomplish a particular goal. Often a balance between optimized code and shipped code is asked of us (after all, the perfect is the enemy of the good), with a heavy emphasis on the former from folks with influence on staffing considerations. But excessive variety is confusing and can even lead to misuse, especially with similar (but distinct and with unique behavior/implications) TF expressions or features; for example, misunderstanding the usage of `count` and extending it beyond the `{0|1}` conditional expression format, or mixing up the effective use of local values and variable default values. There's nothing wrong with getting something working, but over time it's important to establish sensible standards and patterns for incorporating logic into your TF projects and to find appropriate ways to refactor deviations wherever they have cropped up and identify and address them before they become canon.

The way you go about refactoring depends on personal preference (or is mandated by policy); you and I may approach refactoring a particular bit of code in different, and more or less equally valid ways (assuming our results

are equivalent, which isn't always guaranteed). The means is not typically the problem; the problem is usually actually getting around to refactoring. Refactoring working code (even with a few bugs) is not typically a priority, as the benefits are hard (and the costs are fairly straightforward) to measure; however, no one wants to wade through a mire of spaghetti code during a widely impactful, high-severity incident. Clean-up sprints, improve-what-you-touch policies, and refactor bounties can be effective ways to pay down technical debt like deviations from established standards. Whatever the method, it needs to be understood and agreed upon by everyone in the relevant department, with material commitment by all responsible parties, so that other agendas don't crop up and derail such efforts.

Things to Avoid

In your pursuit of working code, you may introduce certain risky behavior that degrades maintainability or even breaks the idempotent nature of TF. This section is a review of certain techniques that are commonly encountered. Typically, when these are introduced, it's done with a noble intent but with a lack of awareness or care. One or a few of these may be found in any project, even ones that directly support workloads representing a great deal of your company's revenue generation. Most likely, even canonically problematic practices, such as using `count` outside of conditional logic, won't require massive efforts to fix (although some fixes may require a great deal of care and courage). The problem is their potential to congregate and become patterns that can hide unusual and mysterious issues, possibly with indeterminate behavior. Let's look into some common issues that are easy to avoid if you know what to look for.

Inline logic

TF supports the use of logic directly as the assignment for a field, with the logic evaluated at runtime as a computed value that must adhere to (or may be interpreted as) the accepted data type of the field. This is especially common for using string interpolation to construct unique values, such as for name, description, or resource tag fields. As we'll see in "String

Interpolation and Directives", string interpolation can be a great way to reduce input variable footprint by reusing common parts, but doing this inline, especially excessively, is likely to result in mystifying puzzles of text.

Worse is the overuse of inline logic on, for example, collections. It's common, and perhaps acceptable, to see inline logic for `for_each` expressions, but outside of this, including `for`/`in` expressions in other inline contexts should be considered bad practice. These expressions are already hard to read, and they're not easy to debug, especially if multiple variables are referenced in the same expression.

Instead of inline logic, a better practice is to place logic in the local values. Although these can't necessarily be evaluated any better than inline statements, they can be decomposed and identified for clear indication of how or where to reference. You'll see more of this in "Expressions".

> **NOTE**
>
> The order of evaluation of the TF state overall can depend on resources referenced in logical expressions. Evaluation of an expression during a `tf apply` is delayed until all of its referenced resources are computed. This is explored further in Chapter 7.

Implicit type conversion

Whereas string interpolation may be appropriate in many cases to construct new strings from smaller substrings, relying upon implicit type conversion is generally regarded as sloppy at best and potentially issue-prone at worst. An example of this is using a string value in a numeric field, such as isolating a portion of an ID that reads `myinstance_m32_d120`, and extracting the values `32` and `120` to use directly in the fields that determine the sizes of memory and disk, respectively, of a compute resource.

Aside from the maintainability and cosmetic aspects, the main problem with interpreted type conversion is when it doesn't actually evaluate to the proper type, such as when a string that should contain a decimal number actually

includes alphanumeric characters. These types of issues may not be caught by TF during a `tf plan`, directly, and may only become obvious when the plan fails to apply. A much better approach is to use the built-in type-conversion functions. We'll explore those in the section "Effective Use of Type Conversion and Safety Functions".

Expressions

TF supports many recognizable expressions, including logical operators, conditional expressions, and iterators. Many of these behave in the very way you would expect from any programming language. Others have quirky nuances, especially in syntax, that beg additional attention to fully comprehend. This section reviews the most common expressions and their safe (and in some cases, where illustrative, unsafe) practice.

Arithmetic and Logical Operators

The logical operators available in TF generally follow the order of arithmetic operations, followed by inequalities/equalities, and finally Boolean operators. Although these operators generally work the way you might expect, not all of them may be wholly appropriate to include in the majority of TF configurations. This section goes into detail on a few choice selections. Here are the operators in the order they're evaluated, together with general observations about their use:

Element negation

> The symbols `!` and `-` (multiplication by -1) should be reserved for flipping an output value read from one resource or module that is to be passed into another `resource{}` or `module{}` block.

Magnitude and modulo

> The `*`, `/`, and `%` (modulo operator) symbols are useful for converting between magnitudes of values or for the uniform spreading of resources.

Addition/subtraction

The symbols + and − (subtraction) should in most cases be used sparingly, only to account for computed or well-defined offsets.

Inequality checks

The symbols >, >=, <, and <= are to be used wherever they're necessary in logic expressions.

Equality checks

The symbols == and `!=` are also to be used wherever they're necessary in logic expressions.

Logical and

The symbol `&&` is also to be used wherever it's necessary in logic expressions.

Logical or

The symbol `||` is also to be used wherever it's necessary in logic expressions.

In addition to negating the output values of, for example, remote modules, it may also be appropriate to introduce negation operators as part of a structure or style standard. For example, in cases where a ternary conditional expression is used to test a Boolean input variable in the context of a `count` statement to optionally include a `resource{}` block, a standard may be imposed to always have a consistent order of the values `1` and `0` (which indicate the resource should or shouldn't be created, respectively), possibly forcing a negated value if the resource is to be created only in the case the variable is false (Example 3-1).

Example 3-1. Creating a resource only when a Boolean value is false

```
resource "google_compute_firewall" "http" {
  name    = var.http_firewall_name
```

```
  network = var.vpc_network

  allow {
    protocol = "tcp"
    ports    = ["80"]
  }
}

resource "google_compute_firewall" "https" {
  count = !(var.http_only) ? 1 : 0

  name    = var.https_firewall_name
  network = var.vpc_network

  allow {
    protocol = "tcp"
    ports    = ["443"]
  }
}
```

The multiplication and division operators are helpful for converting between magnitudes (or other well-defined scaling constants) of fields, such as megabyte to gigabyte, and so on. As mentioned, the modulo operator may be useful for uniformly spreading components across a set of regions, subnets, and more. You can see this in Example 3-2.

Example 3-2. Uniformly spreading compute instances across three regions

```
data "google_compute_regions" "available" {
}

locals {
  three_supported_regions =
slice(data.google_compute_regions.available.names, 0, 3)
}

resource "google_compute_instance" "instances" {
  for_each = { for idx, vm in var.vm_instance :
    vm.name => merge(vm, { index = idx }) }

  name         = each.value.name
  machine_type = each.value.machine_type
  zone         = local.three_supported_regions[each.value.index %
3]
  …
}
```

The for Expression

As alluded to in the opening section, certain logic in HCL may be expressed with less familiar patterns. One such element is the `for` expression. While easily identified, and generally understandable in simple, isolated instances, combining multiple such expressions can be especially difficult to read and reason through.

As its name suggests, the `for` expression (along with its companion reserved word, `in`) is the HCL equivalent of the traditional *for loop*. However, the `for` expression is only the iterator for manipulating the configuration metadata (for example, variables; resource, data, and module outputs; local values; and so forth), not the configuration elements such as a `resource{}` or `module{}` block (iteration over these elements is implemented using the `for_each` and `count` meta-arguments). The `for` expression is used to dissect and marshal data for the configuration elements to consume as field and attribute values.

The `for` expression can also iterate over any collection type. However, there are differences in how elements are referenced from the collection in the iteration statement, depending on the type of collection iterated over. All collections in a `for` expression expose up to two temporary symbols to use as the current element's reference in the expression's predicate. For all collection types, if a single symbol is used, it will always contain the value of the current element. Also, for all types, the inclusion of a second symbol causes it to be the symbol associated with the current element's value, with the first symbol repurposed to expose the index (`list`, `tuple`, and `set` type collections) or the current element's key (`map` and `object` type collections).

> **TIP**
>
> It's highly recommended to establish and publish internal guidelines for use of these temporary symbols, both when one or two temporary symbols should be used (to promote consistency and avoid confusion or mistakes) and how they're to be named (since this is up to the code author). Names for map types, for example, can be as simple as `k` and `v`, or so expressive as to describe what the key represents and the entity type or purpose of the value.

In addition to the uncommon syntax already presented, the `for` expression also requires a special bracket syntax surrounding the statement to denote the expected output type of either a list (surrounded by `[]`) or a map (surrounded by `{}`). The former expects the result of each iteration to be some primitive or custom type and places them in memory as a list. The latter requires a mapping from some primitive type to any other type (primitive or custom).

Though they may often be thought of as interchangeable (and indeed either syntax can be easily modified to produce the other result type), there are distinct use cases for the `list` and `map` versions of the `for` expression. Certainly, a particular collection type may be expected as the input type of a resource or module field, for example, but there are plenty of other uses that are often subjective, and the choice seems arbitrary. Nonetheless, there are good, objective reasons for selecting one over the other: it's recommended to use `for` expressions to output `map` collections when constructing the input for `for_each` statements and to output `list` collections when constructing the value of module outputs. The former ensures a consistent order in which resulting resources are constructed (this is explored further in Chapter 4), whereas the latter often provides a more readable (and theoretically more compressed) format of the data.

> **NOTE**
>
> As suggested, the `for` expression and the `for_each` meta-argument are commonly found together. This may lead to a (perhaps unconscious) belief that the two must be packaged together — a kneejerk reaction that "a `for` expression must be part of a `for_each` statement, and vice versa." However, also as suggested, they address separate components within TF and are thus distinct, with one having no requirement to include the other.
>
> As evidence of this, Example 3-3 depicts valid statements for both `for` and `for_each` in the same resource, but leveraged exclusive from each other.

Example 3-3. Using `for` and `for_each` in an exclusive situation

```
variable "config_map" {
  type = map(object({
    firewall_name = string
    network_name  = string
  }))
}

variable "port_data" {
  type = list(object({
    protocol = string
    port     = string
  }))
}

resource "google_compute_firewall" "fws" {
  for_each = var.config_map

  name    = each.value.firewall_name
  network = each.value.network_name

  allow {
    protocol = "tcp"
    ports    = [
      for item in var.port_data : item.port
      if item.protocol == "tcp"
    ]
  }
}
```

Although it may be very common to find `for` expressions directly inline within, for example, a `resource{}` block as the value of a `for_each`

statement, it's perhaps cleanest to put them all in a `locals{}` block. This way, the results of the expressions can be given meaningful names that describe their contents and intended use.

It's also recommended not to nest `for` expressions, but to instead break into individual lines. This is also established through the use of locals: you can create a named local variable for each `for` expression. The result of each expression could then be used in outputs for better traceability when desired. This is much easier to maintain than wrapping up all the looping logic into one compound expression that's both hard to reason through and impossible to step through the inner workings.

The if Clause

We've already seen examples of how TF/HCL implements common programming language functionality in unfamiliar ways. In particular, conditional expressions, as we'll see in the next section, are handled using the ternary (`?`, `:`) syntax. While there's an `if` keyword/reserved word, it's not meant to be used, as it is in other languages, to bifurcate the path of instructions, creating distinct sequences of actions. Instead, an `if` clause may be included as the final part of the predicate/result of a `for` expression to filter the resulting collection: when iterating over an input collection that contains differentiated elements, it's often the case that only some of the elements are relevant; the `if` clause filters the output to only those elements for which the `if` clause resolves to `true`.

This might be used in the case of a list of VMs, for example, that optionally have an attached disk, where the module creates both instances and disks: the module can first create a collection of disk resources for only those VMs that need them, and these can be optionally referenced later when creating the VMs, as shown in Example 3-4.

Example 3-4. Basic use of the `if` *clause*

```
resource "google_compute_disk" "attached_disks" {
  for_each = {
    for key, vm in var.vm_inputs : key => vm
    if vm.has_attached_disk
```

```
  }
  …
}
```

Depending on the complexity of the information that is being iterated over, there may be multiple ways you'd like to filter it. To extend the preceding example, there may be some other subset of the VMs that, for example, require a second network interface. This second collection may intersect with the first, but generally there's no relationship between the two. As such, filtering the collections would include distinct `if` clauses. Again, it's recommended to include this logic in locals blocks instead of placing them inline as much as possible. This allows you to use suggestive names to advertise their uniqueness and intended use, as depicted in Example 3-5.

Example 3-5. Using different `if` clauses to filter the same collection in different ways

```
locals {
  vms_with_attached_disks = {
    for key, vm in var.vm_inputs : key => vm
    if vm.has_attached_disk
  }
  vms_needing_second_nic = {
    for key, vm in var.vm_inputs : key => vm
    if vm.has_second_nic
  }
}
```

Ternary Conditional Expressions

As mentioned earlier, conditional expressions are primarily included in the form of the ternary conditional expression. This is the same ternary expression found in many other programming languages, and it behaves in effectively the same way, with the same general format as in Example 3-6.

Example 3-6. General format of ternary conditional expressions in HCL

```
BOOLEAN_STATEMENT ? IF_TRUE_RESULT : IF_FALSE_RESULT
```

Just as in other programming languages, HCL conditionals may be used in a number of different scenarios, to various degrees of usefulness. One of the most effective uses is perhaps also obvious: to test a Boolean value, and to

conditionally produce one of two distinct values. This is the typical construction used in other programming languages. Ternary conditional expressions are also useful in conjunction with the `count` meta-argument to construct conditional configuration blocks (such as a `resource{}`, `data{}`, or `module{}` block, or even `dynamic{}` blocks).

Another fairly common use of the ternary conditional expression is to provide a guaranteed default value (often the identity element of the type: such as `0`, `''`, `[]`, `{}`, and so on) in the case where an expression might otherwise result in error. For example, suppose you're setting the value of a field that expects a number type and want to base this value on the length of a list value that is produced as part of an output value read from another resource or module, but this output value may not exist for whatever reason (see Example 3-7). In this way, we can first test for a null value and provide a result of `0`, avoiding an error.

Example 3-7. Using ternary conditional expressions to provide protection against nulls

```
numberField = module.remote_module.complex_output.possibly_null ?
  length(module.remote_module.complex_output.possibly_null) : 0
```

This latter construct is effective for many cases, but its excessive use may expose an unwillingness to introduce more appropriate validation patterns or provide better governance around nullable/empty values. Earlier versions of TF didn't include all the patterns it has today, such as variable validation and nullable fields, the `try()` and `can()` functions, or the `optional` modifier. Unfortunately, a significant number of companies seem to have not progressed beyond these earlier versions (either literally or developmentally/culturally), and although this pattern is rather syntactically expensive and unsightly, it still appears regularly.

String Interpolation and Directives

Generating field values is some of the most common logic found in TF configuration. The most effective (and least problematic) field generation is string building via interpolation or map building over lists of input or

output. In the former scenario, it's introduced to simplify the input fields of component blocks (for example, resources or modules), allowing for a greater number of fields to be constructed from fewer variables to be set by operators. This might be common for an organization that requires standardization around naming conventions and resource tagging (for example, for cost attribution and/or as the subject of a network configuration) such as a repeated application or business code name that must appear as part of the standardized name of several managed resources, in their network and resource tags, or in the descriptions thereof.

String interpolation is great for constructing longer, more complex strings from smaller ones. This is useful for several common resource fields such as resource descriptions, custom names and IDs, and custom outputs. Perhaps the most acceptable use of string interpolation is to clean up or directly manipulate the outputs of resources or third-party remote modules, as it's not always immediately possible to address otherwise. Because these can become difficult to read, especially when, for example, multiple variables are combined with hardcoded text to construct a single input value, it may be most appropriate to construct these in the locals block.

If inline string interpolation is allowed, it's recommended to establish a standard for use. For example, it may be appropriate to allow these in resource ID and name fields that are part of a block that includes the `for_each` meta-argument, instead of requiring these strings to be fully constructed in the locals block (as part of a `for`/`in` expression there), as shown in Example 3-8.

Example 3-8. Using inline string interpolation for certain fields as part of a standard

```
variable "vms_map" {
  type = map(object({
    app_id       = string
    zone         = string
    type         = string
    machine_type = string
    …
  }))
}
```

```
resource "google_compute_instance" "vms_collection" {
  for_each = var.vms_map

  name         =
"vm-${each.value.app_id}-${each.value.machine_type}"
  description  = "${each.value.type} vm for app
${each.value.app_id}"
  zone         = each.value.zone
  machine_type = each.value.machine_type
  …
}
```

The alternative to this would be to fully compute all the fields within the list in the `locals{}` block, before passing it to the `for_each`. This is arguably more complex, but it does lend itself better to more intricate constructions of, for example, string interpolation without compromising the readability of the `resource{}` block itself (Example 3-9).

Example 3-9. Including all string interpolation in `locals{}` *block to keep configuration tidy*

```
#
/////////////////////////////////////////////////////////////////////
////////
# locals.tf
#
/////////////////////////////////////////////////////////////////////
////////
locals {
  augmented_vms_map = {
    for vm in var.vms_map : vm => merge(
      vm,
      {
        name = join("-", [
          "vm",
          "${vm.app_id}",
          "${slice(vm.machine_type, 0, 2)}",
          "${slice(vm.zone, 0, 6)}"
        ])
        description = "Dedicated ${vm.machine_type} vm for app
${vm.app_id}"
      }
    )
  }
}
```

```
#
********************************************************************
********

#
////////////////////////////////////////////////////////////////////
////////
# main.tf
#
////////////////////////////////////////////////////////////////////
////////
resource "google_compute_instance" "vms_collection" {
  for_each = local.augmented_vms_map

  name         = each.value.name
  description  = each.value.description
  zone         = each.value.zone
  machine_type = each.value.machine_type

  network_interface {
    network = each.value.network
  }

  boot_disk {
    initialize_params {
      image = each.value.image
    }
  }
}

#
********************************************************************
********

#
////////////////////////////////////////////////////////////////////
////////
# variables.tf
#
////////////////////////////////////////////////////////////////////
////////
variable "vms_map" {
  description = "value"
  type        = map(object({
    app_id       = string
    zone         = string
    machine_type = string
```

```
    network      = string
    image        = string
  }))
}
```

String directives are more complex templating formats, allowing for conditional logic to be embedded directly into a string's construction. They can be especially useful in custom, log-oriented outputs for resources or modules with optional or binary features (or in lengthy resource descriptions of the same), thereby allowing only the variable portion to be computed and avoiding any repetition of the common portions.

Directives should be used sparingly, given there are often other, more readable ways to implement conditional logic, such as by using the ternary expression. They may be most useful the more similar the resulting strings of sufficient length can be. For example, short, very different resulting strings, such as `'yes'` and `'no'` should be implemented using the typical ternary expression rather than a string directive, as shown in Example 3-10. Even for very long, highly similar resulting strings, it will often be clearer to compute the variable portion as a separate local value and include it with string interpolation.

Example 3-10. Depiction of a string directive and an alternative rendering using only interpolation

```
locals {
  high           = "better than 50/50"
  low            = "worse than 50/50"
  with_directive = join(" ", [
    "The answer is",
    "%{ if var.random > 0.5 }${ local.high }%{ else }${ local.low
}%{ endif }"
  ])
}
# Note the length of the template, making it difficult to
comprehend

locals {
  high_or_low  = var.random > 0.5 ? "better than 50/50" : "worse
than 50/50"
  no_directive =  "The answer is ${local.high_or_low}"
}
# This is simpler and more readable in most cases
```

There is the ability to introduce whitespace stripping with directives using the ~ marker, which might not be well handled by another method. There may also be sound rationale to introduce directives or the Heredoc syntax in the case of *workflow-driven* IaC automation (in which a ticketing system passes in the parameters that were input into a web form by a user or other automated system). However, I would strongly recommend cleansing the textual variable inputs before feeding them to Terraform with a more commonly used tool (as described in the section "Automated Logic Outside of TF", in this case, for example, `sed`/`awk`, Jinja, and so on).

There is support for multiline strings using a Heredoc syntax. This would certainly be an improvement over including newline characters in some other string construction, such as string interpolation. Heredoc is often included as inline instructions, such as shell scripts, to be executed by a remote process or resource. However, multiline shell scripts are probably better served by a remotely stored script file fetched before the TF command is executed and referenced in the code via the `path.module`, `path.root`, `path.cwd`, or `terraform.workspace` values.

Splat Expressions

Splat expressions are intended as shorthand for certain common collection expression constructions that would otherwise require the construction longform `for`/`in` expressions to, for example, extract a specific field from each item in a collection of TF components (such as a list of resources or data sources), as shown in Example 3-11.

Example 3-11. Splat expression and equivalent `for` expression

```
resource "google_compute_instance" "gce_vms" {
  for_each = var.vm_configs
  …
}

…

output "vm_boot_disk_sources_splat" {
  value = google_compute_instance.gce_vms[*].boot_disk.source
}
```

```
# Clean, single line of code; requires some familiarity

…

output "vm_boot_disk_sources_for" {
  value = [
    for vm in google_compute_instance.gce_vms : vm.boot_disk.source
  ]
}
# Somewhat longer to write due to extra syntax; possibly more
accessible
```

Splat expressions are quite appropriate in the definition of output values intended for direct consumption by other modules or systems. A single expression can map a list of created resources to a list of their IDs, for example, which may be necessary as an input to a dependent module. For instance, one module creates a collection of subnets, and another module creates a collection of managed Kubernetes clusters, one per subnet. The latter module could be constructed to create a cluster attached to a supplied subnet name, with a `for_each` expression used to loop over them; rather than also needing to loop through the list of resources to construct the list of IDs, create a new output for the former with the list of IDs printed out. See Example 3-12.

Example 3-12. Using the splat operator in the output of a module, which may be used to assign the input argument for another module

```
#-----------------------------------------------------------------
----------
# Root module
#-----------------------------------------------------------------
----------
#
///////////////////////////////////////////////////////////////////
////////
# main.tf
#
///////////////////////////////////////////////////////////////////
////////
module "network" {
  source = "../network"

  vpc_name = "vpc-tas-3-2"
  regions  = [ "us-central1", "us-east1", "us-west1" ]
```

```
}

module "gke" {
  for_each = module.network.subnet_names
  source   = "../cluster"

  subnet_name = each.value
}

#-------------------------------------------------------------------
----------
# Nested module
#-------------------------------------------------------------------
----------
#
/////////////////////////////////////////////////////////////////////
////////
# outputs.tf
#
/////////////////////////////////////////////////////////////////////
////////
output "subnet_names" {
  value = google_compute_subnetwork.snets[*].name
}
```

Alternatively, splat expressions may be included in output definitions intended for convenient interpretation and reference by human users. This could be useful for recording lists of specific resource information (such as the list of compute resource instance IDs that was created by the script) into a spreadsheet or other document for reporting or tracking purposes.

> **NOTE**
>
> The splat operator has no way of filtering out elements. In any such situation, an `if` clause as part of a `for` expression is likely to be the best solution, as discussed previously.

Functional Terraform

As mentioned in the introductory sections, the functions in TF follow the functional programming paradigm, which may be somewhat unfamiliar if

you come from a background in systems administration or are trained in OOP. For someone most comfortable in a shell environment, each function can be thought of as a terminal/shell command, with support for inline subcommands that get evaluated first and provide their results as, for example, values assigned to an option of the outer command.

If you are more familiar with the object-oriented paradigm, especially those that support *method chaining*, allowing the results of one method to be passed to the input of a subsequent method, each function can be thought of as a method, and instead of chaining, the functions are nested such that the results of an inner function become the value used by the outer function.

This section discusses effective use of several HCL functions, including where each may be most appropriately found.

Effective Use of String Functions

HCL has a number of familiar functions that operate on strings, including to match according to various criteria, create substrings, and operate on the constituent characters.

The `substr()` function is commonly used in conjunction with string interpolations to ensure that strings assigned to specific resource fields will adhere to maximum length guidelines imposed by a particular platform or policy. Ideally, strings are constructed in a fully deterministic way. This may be achievable for any such interpolation strictly using root module inputs, assuming variable validation `condition` expressions are implemented (as shown in Chapter 2): variables simply get set to right-sized string's component elements, so there's no chance for values that are too long. This requires a well-defined standard and significant encoded overhead, the alternative to which is extremely disciplined teams and individuals. However, when constructing input strings based on the output values of other resources and modules, the inclusion of `substr()` may be appropriate, as shown in Example 3-13.

Example 3-13. Using `length()` *and* `substr()` *to cap the length of interpolated strings*

```
locals {
  first_item   = substr(data.google_compute_machine_types.example ,
0, 8)
  second_item  =
substr(data.google_compute_zones.available.names[0], 0, 10)
  final_string = "vm-${local.first_item}-${local.second_item}"
}
```

The `strcontains()`, `startswith()`, and `endswith()` functions produce a Boolean result based on the presence of a substring. This has a clear, but limited, benefit and may often be better served through some other attribute; a contrived example might be to detect the presence of a specific codename as part of a resource's name attribute that is meant to indicate its purpose. A better option would be to use a more self-contained indicator such as a resource tag that can encode this information in a more directly applicable fashion.

The related `regex()` and `regexall()` functions each return a list of matching substrings. These allow for richer matching but are unlikely to be directly actionable, requiring additional expressions to make use of the outputs. This may be better served by abstracting the logic with externalized tooling and passing in additional input variables, or by again leveraging a separate, more specific resource attribute or module output.

CONSISTENT REGEX

Between the two, `regexall()` is far more consistent and less error-prone, always resulting in a list, even if no matches are found (in this case, it returns an empty list, `[]`). By contrast, the `regex()` function actually returns a `list` as one of its possible output types, in addition to a `string` (when only a single result is found) and a `map` (when provided named capture groups as part of the pattern). This former behavior (possibly returning a single string) requires additional logic to properly handle the output of the `regex()` function, further supporting the use of an alternative method, where available. The latter behavior (passing in named capture groups as part of the `regex` pattern) provides highly expressive pattern matching, but this is also supported by the `regexall()` function (which would produce a list of maps).

However, a great deal of skepticism should be applied to any use of the `regex()` or `regexall()` functions, as there are very likely better ways to make the decisions or construct the values in question.

Ideally, the strings the code is operating on are clean enough to use directly. The behavior of the `chomp()`, `trim()`, and various `trim*()` functions (used to remove newlines, whitespace, and arbitrary characters from the beginning or end of a string, for example) in HCL are cumbersome to test as a unit and likely do not perform any better than other common tools external to TF. Then again, there's something to be said for defense in depth (or cleanliness in depth?), so even the use of external tools to clean up text input does not necessarily preclude the use of these functions. The point is, other tools may be more reliable than TF for cleaning up strings (especially if only one tool is to be relied upon).

It's probably best to avoid complex string manipulation like nesting `split()`, `replace()`, `join()`, and so forth, except perhaps in cases in which one must manipulate output values to make proper use of them. Because TF functions are purely functional (rather than, for example, object-oriented), chaining commands can make for less readable code,

making it difficult to reason through. Instead, include these functions in (possibly multiple) local values to allow for expressive names.

Effective Use of Collection Functions

HCL has several functions that operate on collection types. Many of these behave more or less the way their analogous functions/methods would in other programming languages, so the behavior itself is not especially noteworthy. Instead, it's the rationale behind their inclusion that is worth considering. Some functions are very useful in many situations, especially as projects become more and more complex. Others are only useful in specific corner cases. Still others should be considered ill-advised, unless there is truly no other solution available.

The most useful collection functions are those used to combine, decompose, or otherwise reorganize a collection or set of collections, possibly casting it from one collection type to another. These are often used to format various values so they may be properly consumed by a particular resource or module field. For example, the `concat()` and `merge()` functions can be used to combine list-like collections and map-like collections, respectively, which may be useful for combining configuration from multiple sources: for example, combining a list of configuration from an outer module with a list of required configuration in an inner module's local values, as shown in Example 3-14.

Example 3-14. Using `concat()` *to combine two lists*

```
list_1   = [0, 1, 2, 3]
list_2   = ["a", "b", "c"]
big_list = concat(list_1, list_2, ["inline", "list"])
# produces [0, 1, 2, 3, "a", "b", "c", "inline", "list"]
```

The latter function, `merge()`, has the additional effect of allowing the overriding, or masking, of values by ordering them such that the most accurate values appear last in the list. Example 3-15 shows an example of this. A similar effect can be achieved for lists by wrapping the result of `concat()` with a call to the `distinct()` function, which will produce a

`set` type, which is treated exactly like a `list` type except that it must have distinct elements.

Example 3-15. Using `merge()` *to override required field defaults*

```
locals {
  vm_config_default = {"vm_type": "n1-standard-1", "disk_size_gb":
20}
  vm_config_final   = concat(local.vm_config_default,
var.vm_override_map)
}
```

Another useful set of functions includes the `keys()` and `values()` functions, which each produce a list of the respective elements in a collection, each in lexicographical order of the keys providing consistency in order between the two. For the opposite direction, combining list-like types to produce a collection type, HCL includes the `ziplist()` function, which takes a pair of equal-length lists, one of keys (technically it's a *set*, for keys) and one of values, and matches them up in the order in which they appear. These functions can be especially useful for succinctly converting a map into a list and vice versa, for various purposes, including to shape output values for better interpretation or reorganize the collection for use by a resource or remote module. The related function `matchkeys()` may be used along with the `keys()` and `values()` functions to, for example, create a shorthand way of generating a subset of the values of a map object. But there may be more accessible ways to accomplish the same using a `for` expression with an `if` clause (Example 3-16).

Example 3-16. Using `matchkeys()` *to filter a supplied map's keys and the equivalent* `for` *expression rendering*

```
locals {
  filter_keys     = [ "app_name" , "region" ]
  vals            = values(var.config_map)
  keys            = keys(var.config_map)
  using_matchkeys = matchkeys(local.vals, local.keys,
local.filter_keys)
}
# Arguably less syntax; perhaps requires elevated proficiency

locals {
  filter_keys = [ "app_name" , "region" ]
```

```
  using_for   = [
    for ikey, ival in var.config_map : ival
    if contains(local.filter_keys, ikey)
  ]
}
# More syntax; perhaps easier to comprehend
```

Another group of functions that may be used to reorganize data specifically consume or produce set type collections and operate like their corresponding mathematical concepts in set theory. In very specific cases, these might be able to greatly simplify certain logic, especially when collecting and organizing the output values of other modules. The `setintersection()` and `setunion()` functions behave in the way their names suggest, with the former producing a set of the common elements between multiple input lists and the latter producing a set including all elements found in any of the input lists. The `setsubtract()` function takes two sets and performs the *relative complement* of them, which is a fancy way to say that it removes any element from the first set that's also found in the second set. As shown on the TF documentation, the `setunion()` and `setsubtract()` functions can be combined to return the *symmetric difference* of two sets, which are the elements that appear in one, but not in both, sets. The `setproduct()` function takes multiple sets and computes their *Cartesian product*, with every value from each set combined with every value from every other set, as shown in Example 3-17. Again, this would certainly reduce the logic necessary to accomplish this another way, but this should appear in only very limited scenarios, such as where one set of resources must be created for every combination of two other sets of resources, like subnets for a set of VPCs and a set of available regions.

Example 3-17. Using the `setproduct()` *function to compute the Cartesian product of three input lists*

```
locals {
  vpc_names      = ["vpc-dev", "vpc-test", "vpc-prod"]
  subnet_regions = ["us-west1", "us-east1", "us-central1"]
  zones          = ["trust", "non-trust"]
  to_create      = setproduct(local.vpc_names,
local.subnet_regions, local.zones)
```

```
}
# to_create produces the following:
# [
#     ["vpc-dev", "us-west1", "trust"],
#     ["vpc-dev", "us-west1", "non-trust"],
#     ["vpc-dev", "us-east1", "trust"],
#     ["vpc-dev", "us-east1", "non-trust"],
#     ["vpc-dev", "us-central1", "trust"],
#     ["vpc-dev", "us-central1", "non-trust"],
#     ["vpc-test", "us-west1", "trust"],
#     ["vpc-test", "us-west1", "non-trust"],
#     ["vpc-test", "us-east1", "trust"],
#     ["vpc-test", "us-east1", "non-trust"],
#     ["vpc-test", "us-central1", "trust"],
#     ["vpc-test", "us-central1", "non-trust"],
#     ["vpc-prod", "us-west1", "trust"],
#     ["vpc-prod", "us-west1", "non-trust"],
#     ["vpc-prod", "us-east1", "trust"],
#     ["vpc-prod", "us-east1", "non-trust"],
#     ["vpc-prod", "us-central1", "trust"],
#     ["vpc-prod", "us-central1", "non-trust"]
# ]
```

The `transpose()` function flips a specific type of map, one whose values are lists of strings, producing another map with lists of strings as its values, such that all the elements of the original lists become a unique set of keys for the produced map, and the original keys appear as strings in the new value lists under each of the keys that they originally had in their associated list. While this obviously has fairly niche usefulness, it does come in handy in certain use cases. For example, suppose you use a third-party remote module to iteratively generate network components, such as VPCs and subnetworks, and suppose this module includes the list regions in which it creates subnets as an output value. Assuming there is variation to these regions, you might find it valuable to transpose the values and keys to get a map such that the keys are regions, and the values are the list of VPCs that support that region, as shown in Example 3-18.

Example 3-18. Using the `transpose()` *function to flip a map's keys with its values*

```
locals {
  regions_per_vpc = { for vpc in module.networks : vpc.name =>
vpc.regions }
```

```
  vpcs_by_region  = transpose(local.regions_per_vpc)
}
# regions_per_vpc:
# {
#   "vpc-123": ["us-central1", "us-west1"],
#   "vpc-456": ["us-central1", "us-east1"],
#   "vpc-789": ["us-east1", "us-west1"]
# }
#
# vpcs_by_region:
# {
#   "us-central1": ["vpc-123", "vpc-456"],
#   "us-east1": ["vpc-456", "vpc-789"],
#   "us-west1": ["vpc-123", "vpc-789"]
# }
```

Certain additional functions may help clean up the result of another function or a module's output values. For example, the `compact()` function takes a list of strings and returns a copy of the list with the empty strings and nulls removed. This might be useful in combination with a splat expression or other iteration result to, for example, filter down to the nonempty entries of an optional attribute. This might be part of an organization standard as opposed to, for example, using a `for` expression with an `if` clause, as described earlier in the section "The if Clause". The `flatten()` function can clean up a nested list, for example, generated by some other function. This may be useful for collecting information about similar resources that are grouped in different ways (region, workload, and so on), creating a single list.

The `sort()` function takes a list of strings and returns a copy of the list sorted lexicographically. This can help clean up the result of a `for` expression or another collection function. The `reverse()` function will produce a copy of a list in the reverse index order, which may be used in conjunction with `sort()` to produce a list of strings sorted in reverse-lexicographical order. These would likely be most useful for output values of root modules intended for easy interpretation by human operators or read by other services.

Yet another set of functions retrieve individual items from a collection. The `element()` and `lookup()` functions may provide somewhat safer

alternatives to the bracket notation method of selecting elements in, for instance, a list or map, respectively. The `element()` function has the additional feature of "wrapping around" if the index requested exceeds the length of the list, providing a modulus indexing mechanism. This could be useful for uniformly spreading resources across a set of regions, projects, subnets, or other resources, for example. The `lookup()` function includes a default value that can protect against the case where a particular key isn't found in a map. This shouldn't be necessary in most cases, and may be most useful in dealing with inconsistencies in other modules' output values. The `index()` function takes a list and a value and returns the first index in the list in which the value appears; however, this will produce an error if the value is not found. It's also highly unlikely that the utility this provides cannot be done in a more appropriate fashion or context.

There are several functions that test the values of a collection. For example, `alltrue()` returns `true` only if all the elements are `true` or `"true"` or if the collection is empty, and `anytrue()` returns `true` if any of the elements are `true` or `"true"` and `false` otherwise or if the collection is empty. Another similar function, `contains()`, returns `true` if a supplied list, tuple, or set contains the second argument as an element. These functions may be most useful in conjunction with other functions and expressions to construct aggregate indicators to be exposed as a module's output values.

There are plenty of other functions with seriously suspect usefulness. For example, the `range()` function can create a series of numbers; this has limited applicability, most of which can be solved in a more direct way, and which, if truly required, could probably be generated in a more prescriptive way. The purpose of the `chunklist()`, `coalesce()`, `coalescelist()`, `one()`, and `slice()` functions are all likely better served by externalized logic, or by constructing the overall project logic in a more consistent and deterministic way. The most obvious exception to this is when reading from service provider-managed data sources such as available cloud regions or availability zones; using the `slice()` function enables a limit to the number of elements.

> **NOTE**
>
> The recurring theme here is that there is certainly a great deal of functionality supported in TF/HCL, but not all of it is strictly necessary, especially when other platforms, services, and software are available.
>
> That's not to say the use of the latter collection of functions mentioned is necessarily dangerous or problematic, and indeed there may be no other valid way to accomplish a certain task in very corner cases (and there are sure to be corner cases in almost every sufficiently complex TF ecosystem).
>
> The point is more that any use of such functions should be scrutinized for alternative methods, and heavily documented and spread in awareness, before accepting its inclusion into your infrastructure project.

The length() Function

Like many other languages, HCL exposes a `length()` function that can be used on strings or collection types. This behaves how you would expect: for strings, it returns the number of characters; for collections, it returns the number of value entries (which can be thought of as the number of indexes for lists or keys for maps and objects).

For strings, the `length()` function should be largely unnecessary in most cases, outside of variable validation `condition` expressions. As shown in Chapter 2, the `length()` function may be used to enforce the length of an individual input variable, which may be necessary for ensuring that strings are properly sized. Similarly, as mentioned in "Effective Use of String Functions", `substr()` can be used to ensure a maximum limit to a string's length. However, `length()` can be used to enforce a *minimum* limit on string length, even outside of `condition` expressions, if this is a necessary constraint that you cannot guarantee you will meet another way (such as if the string is coming from a resource's attribute or a third-party module's output value).

For collections, `length()` may be appropriate for the purpose of constructing aggregate metrics exposed as a module's `output` values. For instance, if your module constructs a variable number of resources, it may be helpful to include the latest number of provisioned resources, according

to the current codebase/TF state, as an output, both as a convenience for any consuming modules or platforms and to present a convenient way to interpret the results as a human operator (assuming the values are output by the root module, which may require them to be passed from one module's output value to another; Chapter 6 explores this further). An example of this is shown in Example 3-19.

Example 3-19. Using the `length()` *function in an* `output{}` *block for convenient indicators*

```
output "vms_count" {
  description = "Count of vms created by this module"
  value       = length(google_compute_instance.vms_list)
}
```

Introducing more complex logic based on length is perhaps ill-advised, avoidable, and awkward (it also seems to be fairly rare). A somewhat contrived (and arguably inappropriate) example of this may be to trigger a quota increase request when a certain threshold of inputs or resources is reached. This may be an effective way to enhance flexibility, especially in the case where a codebase is shared among multiple teams (perhaps using unique *.tfvars* files, as discussed in detail in Chapter 7). However, there are other ways to detect and react to this (for example, as a subsequent step in the automation job or as a separate workload that gets triggered and asynchronously approved), which may be more traceable and have a greater degree of governance.

Effective Use of Encoding and Hash/Crypto Functions

HCL includes a number of encoding, hashing, and cryptographic functions. While these may have similar behavior or produce similar results, they should be used only for their intended purpose according to the specific use case.

There are several encoding and decoding functions available in HCL, including functions to decode/encode into base64, properly encoded URLs, and to decode/encode various file encodings (such as JSON, YAML, and CSV). The best use of the `base64encode()`, `base64zip()`, and

`textencodebase64()` functions may be to compress certain information (perhaps for archival purpose) or to generate certain hash IDs that are based on strings.

> **NOTE**
>
> It's always important to note that encoding is not encryption. Nothing that is encoded into base64 is secure in any way.

The hash functions such as `uuid` and `uuid5` may be best used for coming up with hashes or IDs that are unique and not based on anything like file contents. There are also several other hashing functions available in HCL, including various `sha*` and `md5` functions, and the `bcrypt()` function. These might be used as a file's signature and computed to validate a remotely fetched file.

Finally, there is the cryptographic function, `rsadecrypt()`, which will decrypt an RSA-encrypted ciphertext and return the corresponding cleartext. Unfortunately, there is no corresponding function for encrypting a string. Any encryption should be done outside of TF (which suggests that any decryption should be done external to TF, as well).

Effective Use of Filesystem Functions

There are two types of filesystem functions in HCL: those that actually access the filesystem, such as `file()`, `fileexists()`, and `templatefile()`, and those that simply help format filepath-related strings, such as `dirname()`, `pathexpand()`, and `basename()`. The former operate on files that exist (and are available) on the same local filesystem as what TF is running against.

> **NOTE**
>
> The functions that access the filesystem require the files on which they operate to exist on the filesystem in which TF is executed "before [TF] takes any actions," according to the documentation.

Filesystem functions might be used to pass shell scripts to compute resources for startup/restart functionality. Ideally, any such scripts are assigned in some kind of template or configured in an image, rather than initiated separately on individual resources.

ALTERNATIVES TO FILESYSTEM FUNCTIONS

There are probably more appropriate mechanisms for most of the utility provided by these filesystem functions. As suggested, generating instance templates or machine images, for example, is typically more fitting than configuring individual compute instances; however, there are plenty of tools and services that are specifically designed to manage such components.

Additionally, generating such templates or machine images probably has a different lifecycle than the compute instances that leverage them, so managing them in another tool or mechanism is unlikely to cause technical friction.

Another example of where the filesystem functions might be considered, but other solutions are likely to be more suitable, is in managing updates to resources in a secret manager. Sensitive data (ostensibly any data stored as secrets) should never be stored anywhere it isn't strictly needed (including the environment from which TF commands are executed), just like it should never appear in log output, so there should never be a sensitive file from which to read. As Chapter 7 explains, a one-off operation such as this is generally outside the purview of TF anyway, as it violates the idempotent nature of TF and can cause undesirable behavior if necessary files are missing at the time a plan is applied.

Effective Use of Type Conversion and Safety Functions

As you've seen, HCL implements implicit type conversion and will do its best to interpret a field's value as the field's configured type. However, this may have unexpected behavior if, for example, the value is coming from a third-party service or tool (or a junior team member) that includes characters not supported by the field's specified type so that there's no obvious conversion available. As such, HCL additionally includes a number of functions for testing and casting between types.

Perhaps the most basic such function is the `type()` function. This returns the type of the input, from the HCL/TF perspective, and can be used to test for a value's type before it's passed along to the resource or module for which it is intended. In most cases, this should be unnecessary: the values assigned to a module's variables or that come out of a remote module's outputs should be well known and generally should not change. As you'll see in Chapter 6, rather than changing the type of a module's variable or output value, it's better to introduce a new variable or output with the desired type, potentially formally deprecating the variable or output if it's no longer needed or supported.

In cases where you cannot control the type of a value, such as output values of a third party, it may indeed be appropriate (or at least most pragmatic) to cast one value to another. For converting from a number or a Boolean to a string value, HCL exposes the `tostring()` function. While this adds syntax, it's more than sugar as it advertises the intent to convert type. It also will throw an error if it receives anything other than a primitive type value (one of type `string`, `number`, or `bool`) or `null`, which means it provides better validation than simple type conversion.

> **NOTE**
>
> It's also possible to use the string interpolation method for converting from numeric or Boolean values to strings. However, `tostring()` is arguably a far more expressive and readable method, and it's consistent with other recommended conversion approaches.

Likewise, for converting from a string to a numeric value, HCL provides the similar `tonumber()` function. Just as before, this will throw an error if it receives a value that isn't a string-encoded decimal, `null`, or itself a number. Another option is to use the `parseint()` function, which also takes a string and converts it to a numeric value, but be aware that this function also requires the base to be passed in (for example, 10 for

decimal). Because of this, it can also process other number systems like binary and hexadecimal.

For converting from a string to a Boolean, the `tobool()` function accepts only the exact strings `"true"` and `"false"`. If this is found within a longer string, it may require additional logic to first isolate, and then cast to lowercase, the specific portion. Unfortunately, sometimes Boolean values are output as a string by the cloud service provider (or the TF provider that operates it), making this explicit type conversion a good way to identify and handle such behavior.

> **TIP**
>
> Note that none of these methods work for converting between Boolean and numeric values. The safest way to convert between a Boolean and the numeric representation is to use a ternary expression: for example, `numType = boolVal ? 1 : 0` and `boolType = numVal == 1 ? 1 : 0`.

HCL also exposes *type conversion* functions for collection types: `tolist()`, `tomap()`, or `toset()`. The former two are really only intended to fully marshal the data to a consistent type. They inherently use implicit type conversion to make the values all of the same type. The latter function, `toset()` (in addition to implicitly normalizing the values as it does with other collection types), takes a list-like value and coalesces duplicate values. Neither `tolist()` nor `toset()` preserves order. The latter may be most useful in everyday use as a way to prepare the output from a remote module to, for instance, pass the resulting value to a `for_each` meta-attribute statement, as shown in Example 3-20.

Example 3-20. Cleaning up remote module output with `toset()`

```
resource "google_storage_bucket" "regional_log_bucket" {
  for_each = toset(module.network.subnets[*].region)
  location = upper(each.value)
  …
}
```

A final type conversion function is `ephemeralasnull()`, which will force an ephemeral value to be null. Frankly, this — and the `ephemeral` keyword in general, which should be used for only temporary elements that are not intended to be recorded in the TF state, which conflicts with the idempotent nature of TF — should have limited use in most cases. Chapter 4 explores this further.

Additional type-related functions are `can()` and `try()`, the former of which was reviewed in Chapter 2. They operate in much the same way but have different use cases: they both take expressions, other functions, or any other logical phrase that may be problematic (for example, that throws an error). However, `can()` returns a Boolean value, with `true` indicating no errors would be thrown; `try()` allows for a default return value if the expression would throw an error.

The `can()` function is one of the rare places where the TF documentation provides solid guidance: it "is intended only for simple tests in variable validation rules." Outside of this, the docs recommend that you "prefer to use [the `try()` function]". This is solid advice, especially given how useful the `try()` function is.

One of the best uses of the `try()` function is for filling in missing data (or incompatible signal data, such as `"NULL"`, or something supplied by a platform, service, or another tool) with default type values (for example, `""` for string, `0` for numbers, `[]` for lists, `{}` for maps, and so forth), which may be necessary for consumption by a TF or CSP process.

The `try()` function can also be used to handle a value that "might be provided in two different forms," but this should not be necessary in most cases. If you have any influence over how the values are constructed (such as if they're developed by an internal team or you're able to set them yourself as a variable assignment), you should do your best to ensure the values are a single, specific value. If the value comes from a third-party remote module, hopefully the third-party entity is following good module development practices (you'll see more of this in Chapter 6).

A last piece of solid advice provided by the TF docs is that although complex expressions may successfully be evaluated, they "recommend using [`try()`] only with simple attribute references and [in conjunction with] type conversion functions" such as `tostring()`, `tobool()`, and so on. Instead of placing a more complex expression within the `try()` function, place the `try()` function around only the specific portion of the expression, as shown in Example 3-21.

Example 3-21. Comparing unsafe and safe uses of `try()`

```
try(tostring(local.raw_value.name), null)
# This might not always be safe

tostring(try(local.raw_value.name, null))
# This is much safer
```

Effective Use of Sensitive Value Functions

As you saw in Chapter 2, variables may be designated as sensitive to prevent TF from printing its value in the log output of TF commands like `plan` or `apply` (there's a similar designation for outputs, explored in Chapter 4). Additionally, HCL provides a collection of functions that work with sensitive values, including a `sensitive()` function to designate a particular value in the same way, a `nonsensitive()` function to remove the designation, and an `issensitive()` function to test if a value has the sensitive designation. Also, as referenced in Chapter 2, ideally you aren't generating sensitive values in HCL. Moreover, alternative methods to passing in sensitive information as variables should be considered, such as only leveraging platform and infrastructure components that you can configure to read sensitive data directly from a secret management system. However, this may not always be feasible if more appropriate tools and services are not yet available.

The TF documentation for `sensitive()` recommends to prefer the variable and output arguments when possible. Indeed, setting a value to `sensitive()` after it's been initialized gives little guarantee to it being treated with sensitivity before the function call. If you want to suppress any

chance a value gets printed to a console output, any consuming modules must provide a variable marked sensitive, and any module producing any such values in outputs must likewise mark them as sensitive. As such, the `sensitive()` function should ideally never be necessary. However, it may be the best you can do if the developers of a third-party remote module are slow (or loath) to introduce the sensitive argument (and it's determined to be too costly, or legally disadvantageous, to fork — and implement and maintain — yourself).

The opposite function, `nonsensitive()`, is arguably more useful even assuming all modules effectively account for the sensitive argument on all relevant variables and outputs. Indeed, as mentioned in the TF docs, indiscriminate use of the `nonsensitive()` function suggests a severe misapplication, possibly exposing sensitive data in plain-text log output. However, as presented in the TF documentation for the `nonsensitive()` function, it may be used in situations where "you've derived a new value from a sensitive value in a way that eliminates the sensitive portions of the value." Valid (yet contrived) examples might include decomposing a string that only has a sensitive value as a component element, and the resulting substring is both nonsensitive and valuable to have printed, such as in the output of `tf apply` or `tf plan` commands.

Finally, the `issensitive()` function validates whether a value is designated as being sensitive, returning the Boolean `true` if so and `false` if not. While the use of this function has no chance of exposing any sensitive data directly, the motives for including it should be explored. As discussed in detail, any generation of sensitive data should be done outside of TF if possible, and otherwise, the sensitive nature of any values should be established in the variable or output value definition blocks. Perhaps the only valid (yet also highly contrived) case where this might be seen is where a remote module is both volatile in how variables and output values are established and restrictive in the way it implements exposed versions. This way, it may not be guaranteed that a particular value is known to be treated with sensitivity within the module. While this sounds excessively unlikely, it may be one of those problems born out of an illusion of success

and safety and that you don't realize you have until it's too late (the list of such possible problems is infinite). Along with the other sensitivity functions, use the `issensitive()` function sparingly and with great scrutiny.

Effective Use of Other Functions (Numeric, IP Network, and Date/Time)

The numeric functions `abs()`, `ceil()`, `floor()`, `log()`, `pow()`, and `signum()` are all effective in handling their respective mathematical operations. These may be appropriate for converting between one representation, or magnitude, of a value, passing it from the output value of one module or resource into the attributes of another. However, any such calculations may be better computed in a mechanism external to TF.

The collection-related numeric functions, `max()` and `min()`, also are effective in handling their respective operations and take a variable number of number arguments. As suggested by the TF documentation, a list of numbers may be evaluated with `max()` or `min()` by expanding the list with the `...` operator. Once again, any such comparisons may be better computed externally.

The `parseint()` function produces an integer representation of the supplied input string that is encoded in a specified number base (between 2 and 62 inclusive). This can be useful for reformatting — for example, a hexadecimal or binary value that is gathered from a resource attribute or module output and needed as an argument for another `resource{}`/`module{}` block.

The `cidrhost()` function returns an IP address within a supplied Classless Inter-Domain Routing (CIDR) encoded as a string according to an offset value corresponding to the host's index within the network. This might be used to prescribe a configured host's IP, especially in cases where the CSP or platform exposed by the TF provider would otherwise assign IP addresses in a way that, for example, violates your organization's policies. The `cidrnetmask()` function converts a range of IP addresses encoded

in CIDR format into the mask address format. This would likely be most useful for converting a CIDR exposed as, say, an output value by one module and passed into another `module{}` or `resource{}` block that requires the mask address format.

The related `cidrsubnet()` and `cidrsubnets()` functions both take a CIDR block as input and compute a specific subrange (or set of ranges) within the block according to other supplied inputs. This might be used to automatically break down a large CIDR block (perhaps assigned to, or designated for, an entire VPC, collection of projects, or super-region) and spreading the resulting subranges among a known number of subnets. Most likely this should be calculated and tracked externally by a network architect or team, and CIDR ranges explicitly passed in (perhaps read from a structured file, like a CSV). However, in cases of replicating highly standardized network topologies or infrastructure reselling, the `cidrsubnet()` and `cidrsubnets()` functions may be highly effective for constructing networks with uniform cardinality and size designed to be assigned in a factory pattern.

The `formatdate()` function is useful for formatting third-party, system-generated, or CSP-provided timestamp values, converting them into different time formats. This might be especially useful for passing a timestamp output from one system to another system that expects a different format. It also may be used to effectively format a timestamp exposed in a root module that's intended to be interpreted by human operators.

The time functions exposed by HCL are `plantimestamp()`, `timeadd()`, `timecmp()`, and `timestamp()`. The last function, `timestamp()`, will generate a timestamp value at the instant it is encountered in the evaluation of an `apply` action (it cannot be predictably evaluated in a `tf plan`). You can use it in conjunction with the functions `timeadd()` (which adds a duration value to a timestamp in RFC 3339 format) and `timecmp()` (which compares two such timestamps formatted à la RFC 3339) in the context of a `postcondition`'s `condition`

expression within a `lifecycle{}` block to evaluate, for example, when a policy or certificate's expiration date is impending (or passed). Because the function returns a different value each time it's invoked, it's generally advised to also include the `ignore_changes` meta-argument to keep it from affecting a resource's lifecycle. Similar to `timestamp()`, the `plantimestamp()` function is intended to evaluate the time during a plan activity and may be computed as part of the expression assigned to an `assert`'s `condition` argument within a `check{}` block to likewise evaluate an expiration date or other impending deadline, for instance. However (as will be asserted again elsewhere in this book), due to the system's goal of idempotence, it's generally recommended to handle the management of security certificates, temporary access policies, and other time-limited data (especially sensitive such data), outside of TF.

Summary

You now know what operators, expressions, and functions are available in TF, which ones to leverage in various scenarios, and which should be avoided (or eventually replaced) in normal HCL code construction for TF development. You can also recognize which operations are appropriate to encode into your TF projects and which should be externalized using other tools, services, or custom software. You've seen several examples of potentially problematic configuration with corresponding examples of safer, more effective approaches.

In this chapter, you learned the following about the various operators, expressions, and functions:

- General guidance on when, where, and why to implement logic in TF projects
- Which concepts or constructs to avoid in TF development
- The order of mathematical operators and their relative usefulness

- Expressions for iterating through collection values and filtering out unnecessary elements
- Implementing conditional logic using the ternary expression
- How to implement string interpolation and when to use directives for implementing effective string templates
- The many various functions available in HCL and their valid and successful use

In Chapter 4, you'll review additional built-in fields such as outputs, local values, and meta-arguments. You'll see how to use these to develop high-quality, well-organized code that enhances clarity and reduces repetition. This will complete the overview of TF configuration in terms of how to write the code.

Chapter 4. Built-In Fields: Reference Attributes, Local Values, Meta-Arguments, and Outputs

In Chapter 2, you saw the various ways to construct and assign input variables in TF. In this chapter, we'll build on this premise, and you'll see the other elements that may hold a value in TF, how to effectively construct them, and how they may be leveraged strategically to enhance flexibility, readability, and maintainability of TF projects. You'll also learn the meta-arguments and the `dynamic{}` block, which are both used to further augment flexibility and, more importantly, how to introduce them tactically.

Resource Fields and Reference Attributes

Similar to input variables for modules, the *resource fields* are the input elements for a `resource{}` block. They're defined by the provider and generally map to parameters expected by the corresponding API that's exposed by the CSP. It's common for most resource fields to be optional, with default values (that are sometimes `null`) in many cases. For example, a general purpose storage bucket may support dozens of first-order resource fields, but require only two, as shown in Example 4-1.

Example 4-1. Using only the required resource fields to establish a Google Cloud Storage bucket

```
resource "google_storage_bucket" "simplest" {
    name     = "simplest-bucket-configuration"
    location = "US"
}
```

In addition to resource fields, TF supports another resource component, the *reference attribute*, which will be assigned informational data as values by the CSP. These could be considered the output values and may be read from the resource by referencing its name (or object path) following a dot notation format, as depicted in Example 4-2.

Example 4-2. Referencing a resource's reference attribute

```
resource "google_storage_bucket" "referencing" {
    name     = "referencing-bucket-configuration"
    location = "US"

    logging {
        log_bucket = google_storage_bucket.simplest.name
    }
}
```

The reference attributes will include entries for each of the resource fields and will typically also include several values that are assigned by the CSP, such as a computed, CSP-specific ID or some other distinguishing information. In some cases, the CSP (or perhaps the provider) may even assign a recalculated value from the resource field, if there is a preferred, or more specific, format. For example, if a location field such as a specific region or availability zone is not provided for a network-attached resource, it may be assigned a default location by the CSP, and a supplied network name may be augmented with this same location.

TF works best when any dependencies among the resources under a given TF state are all established. This is due to the hierarchical nature of certain components. For example, a typical compute instance resource must identify a network, such as a VPC, to connect to via a network interface. It may be possible to identify the VPC by name, rather than a more specific identifier. If the compute instance does not leverage the VPC's `name` reference attribute but instead assigns to its `network` resource field the same input variable that the VPC uses for its own `name` resource field, as shown in Example 4-3, TF may produce an error if it tries to create the compute instance resource first. To communicate the correct order to TF, you should use the VPC's `name` reference attribute, also depicted in

Example 4-3. Proper establishment of dependencies is explored more in "Effective Use of depends_on Meta-Argument".

Example 4-3. Before and after refactoring: replacing shared input variables with reference attributes to prevent dependency errors

```
#------------------------------------------------------------------
----------
# Before refactor
#------------------------------------------------------------------
----------
#
////////////////////////////////////////////////////////////////////
////////
# main.tf
#
////////////////////////////////////////////////////////////////////
////////
resource "google_compute_network" "vpc" {
  name = var.network_name
}

resource "google_compute_instance" "instance" {
  name         = var.instance_name
  machine_type = var.machine_type
  zone         = var.instance_zone

  boot_disk {
    initialize_params {
      image = var.boot_image
    }
  }

  network_interface {
    network = var.network_name

    access_config {
      // Ephemeral public IP
    }
  }

  service_account {
    email  = google_service_account.default.email
    scopes = ["cloud-platform"]
  }
}
```

```
#
********************************************************************
********

#
////////////////////////////////////////////////////////////////////
////////
# variables.tf
#
////////////////////////////////////////////////////////////////////
////////
variable "network_name" {
  type = string
}

variable "instance_name" {
  type = string
}

variable "machine_type" {
  type    = string
  default = "n2-standard-2"
}

variable "instance_zone" {
  type    = string
  default = "us-central1-a"
}

variable "boot_image" {
  type    = string
  default = "debian-cloud/debian-12"
}

#
********************************************************************
********

#
////////////////////////////////////////////////////////////////////
////////
# terraform.tfvars
#
////////////////////////////////////////////////////////////////////
////////
network_name  = "vpc-tas-4-1"
instance_name = "gce-tas-4-1"
```

```
#-----------------------------------------------------------------
----------
# After refactor
#-----------------------------------------------------------------
----------
#
/////////////////////////////////////////////////////////////////
////////
# main.tf
#
/////////////////////////////////////////////////////////////////
////////
resource "google_compute_network" "vpc" {
  name = var.network_name
}

resource "google_compute_instance" "instance" {
  name         = var.instance_name
  machine_type = var.machine_type
  zone         = var.instance_zone

  boot_disk {
    initialize_params {
      image = var.boot_image
    }
  }

  network_interface {
    network = google_compute_network.vpc.name

    access_config {
      // Ephemeral public IP
    }
  }

  service_account {
    email  = google_service_account.default.email
    scopes = ["cloud-platform"]
  }
}

#
*****************************************************************
********

#
/////////////////////////////////////////////////////////////////
////////
```

```
# variables.tf
#
////////////////////////////////////////////////////////////////////////
////////
variable "network_name" {
  type = string
}

variable "instance_name" {
  type = string
}

variable "machine_type" {
  type    = string
  default = "n2-standard-2"
}

variable "instance_zone" {
  type    = string
  default = "us-central1-a"
}

variable "boot_image" {
  type    = string
  default = "debian-cloud/debian-12"
}

#
**********************************************************************
********

#
////////////////////////////////////////////////////////////////////////
////////
# terraform.tfvars
#
////////////////////////////////////////////////////////////////////////
////////
network_name  = "vpc-tas-4-1"
instance_name = "gce-tas-4-1"
```

Output Values

Similar to resource attributes, TF also allows customized information to be passed out of a module through its support of output values. *Output values*

are defined in an `output{}` block and may be referenced from the `module` object it belongs to or, in the case of the root module, be printed to the CLI. They are used to organize and expose the information related to the resources created and configured by TF.

Technically, output values are optional components of a TF module. In many cases, output values are not strictly necessary, anywhere, to the construction of highly effective, fully consistent, and predictable TF configurations. It's perhaps because of this optional nature that custom module output values might be one of the most underutilized features in all of TF. However, constructing outputs sensibly can lead to stronger dependencies (where desired and appropriate) between resources.

Moreover, constructing output values early (perhaps as part of a standard process for TF development) is advised, as their evolution (or stability) can be used to help expose unexpected changes. Finally, even in the event of establishing output values that go unutilized, the benefit of visibility and understanding will likely far outweigh any truly problematic overengineering.

To put a finer point on it: you definitely want to implement outputs.

The output{} Block

The `output{}` block has only a single required field, the aptly named `value` field. The remaining fields are optional, but similar to the `variable{}` block, you should also define a `description` field. Here's an overview of each field:

`value` *field*

> This is the only field required by the `output{}` block. It's generally either the result of an expression or a reference to a resource's attribute, local value, or output of a submodule.

`description` *field*

Similar to the `variable{}` block's `description` field, the value of this field is a string that should ostensibly render the purpose of the `output{}` block's value. As with variables, the `output{}` block's `description` field is strongly recommended to be a typical (if not mandatory) element.

`sensitive` *field*

The `sensitive` field is also similar to the version found in an input variable. It keeps outputs from being printed to the CLI, which would be especially useful for any sensitive values generated by the TF project, directly or indirectly. It does *not* encrypt or otherwise obfuscate an output's value.

`ephemeral` *field*

The `ephemeral` field is reminiscent of its counterpart field for input variables. It, too, will prevent the output's value from being preserved in the TF state object. This might be useful for values that are purely intended to be passed from a submodule up to the scope of a referencing module, or for values that are useful to print to the CLI but are duplicate or otherwise extraneous for TF to remember.

`depends_on` *field*

The `depends_on` field has no analog in the `variable{}` block. However, it works the same as the `depends_on` field of a `resource{}` or `module{}` block. It may be assigned only a reference to a resource, a module, or a list thereof. As it suggests, it creates an explicit dependency on whatever element is assigned as the value. Technically, this is a meta-argument (more on this in "Effective Use of depends_on Meta-Argument").

`precondition` *field*

The `precondition` field has an analog in the `variable{}` block: the variable's `validation` field. Similarly, the output value's

`validation{}` block has two required fields: `condition` and `error_message`. The former is a logical expression that must produce a Boolean value; a value is accepted only if this evaluates to `true`. The latter is an expression that must evaluate to a string value and will be printed to the command line as part of an error message from TF.

Outputs must contain a value and should contain a description. Beyond these, the remaining fields are case dependent. Example 4-4 depicts the typical renderings of several common output constructions.

Example 4-4. Several common output constructions

```
output "simple_reference_output" {
    description = "Name of the simplest configuration bucket"
    value       = google_storage_bucket.simplest.name
}

output "forin_expression_output" {
    description = "Several bucket names using for/in"
    value       = [for b in google_storage_bucket.multiple_buckets
: b.name]
}

output "splat_reference_output" {
    description = "Several bucket names using Splat"
    value       = google_storage_bucket.multiple_buckets[*].name
}
```

While it's technically possible to generate sensitive values with TF, there may be better, safer ways to instantiate or update them. A better way may be to integrate the platform or system running TF with a secret manager, letting that system generate and maintain the sensitive value.

One case where generating sensitive values is wholly appropriate is when they're intended to be consumed by the platform running TF, such as a key that it uses to secure or identify itself in some way. In such cases, the `sensitive` field will keep the key from being read by anyone who happens across the log output.

Strategic Use of Outputs

Without explicit definitions, a module has no outputs. Every module should have outputs defined. Even small and experimental modules should include a basic collection of output values, if only to organize some of the information already expressed by the output of the plan phase. Ideally, outputs should be included in a dedicated file, typically named *outputs.tf*, and grouped by the resource or module from which it draws its information. It may also be worth it to maintain an order of the grouped `output{}` blocks matching the order in which they appear in the module.

For referenced modules (those intended to be referenced as the source of a `module{}` block), `output{}` blocks allow for information about the module's components to be organized in meaningful ways. They can even expose the resources (or nested `module{}` blocks), which allows all their reference attributes to become available in the output value. These may then be read by the calling/outside module, either to further expose the information (in that outer module's outputs) or to pass this information to other configurations, thereby establishing an implicit dependency similar to the reference attribute described in "Resource Fields and Reference Attributes". Examples of exposing data from a module are depicted in Example 4-5 (descriptions omitted for brevity).

Example 4-5. Examples of exposing different data from a module

```
output "full_compute_resource_reference" {
    value = google_compute_instance.module_compute
}

output "compute_resource_name" {
    value = google_compute_instance.module_compute.name
}

output "full_submodule_reference" {
    value = module.submodule
}
```

> **NOTE**
>
> As explored in Chapter 5 on remote module development, a module shouldn't mix `resource{}` and `module{}` blocks. The example is only for illustrative purposes.

For *root modules* (those that are the direct subject of a TF command such as `tf plan`), outputs are printed in the command line. This may be useful in cases where another system is working downstream from TF itself and requires information that's available in the TF project. The output values may be constructed and formatted so that these systems can more readily interpret the information TF captures about the running resources. In these cases, it may be prudent to include some basic validation in the `precondition{}` nested block to ensure the information is available and well formatted. Validation is discussed in the next section.

> **TIP**
>
> For root modules, it's always recommended to construct meaningful outputs that are intended to be interpreted by either human operators or downstream services or platforms. This may require constructing new values by reorganizing or decomposing the information so that they can be printed in the most meaningful and actionable fashion.

Validating Output Values

In most cases, it's not strictly necessary to include a `precondition{}` nested block for every output value. However, because output values do not have a direct mechanism for enforcing their value's type like input variables do, the `precondition{}` nested block may be used to do this. Hopefully, between the `validation{}` blocks of your input variables and the output types defined by the resource provider, you may find `precondition{}` nested blocks to be unnecessary. However, in cases where the output value is constructed from different resource attributes, it may be the best way to enforce some semblance of type safety, especially if

the elements used to construct the value have dissimilar types, as shown in Example 4-6.

Example 4-6. Checking for the output value's type in the `precondition{}` *nested block*

```
# resource entity types truncated for readability
output "precondition_output" {
    value = "VPC_${network.vpc.name}_${length(firewall.fw)}"

    precondition {
        condition = (startswith(network.vpc.name, "corp-vpc-")
                            && length(firewall.fw) < 3)
        error_message = "VPC starts with 'corp-vpc-'; at least 3
firewalls."
    }
}
```

Another scenario that may call for leveraging the `precondition{}` nested block is with the expectation that a downstream system, as described in the previous section, will read an output and needs it to adhere to a specific format. The `precondition{}` nested block can be used to ensure the output will contain the necessary information or pattern the downstream system expects, as shown in Example 4-7.

Example 4-7. Matching an output's `precondition{}` *nested block to a downstream system's format*

```
output "system_format" {
    value = upper(module.submodule.special_instance_name)

    precondition {
      # Disallowing a special sequence that would be misinterpreted
      condition     =
!startswith(module.submodule.special_instance_name, "DEL")
      error_message = "The submodule created a disallowed instance
name"
    }
}
```

Output validation may also be useful as part of a post-condition check for certain fragile resources that may not always get configured fully or with the appropriate values. For example, a resource that has a *last mile configuration* script applied to it as part of its initialization may run into

issues that go undetected by TF. In these cases, it's possible to construct a `precondition{}` nested block to check the effect of this procedure.

Local Values

Unlike a scripting language like bash or Python, or any other software language with a strong procedural orientation, HCL configurations are primarily confined to value assignment with support for logical expressions as the *predicate* (that is, the value portion that appears after a = or `:` in a field, variable, or other value assignment). Frequently, this manifests as logic embedded into the `resource{}` or `module{}` block as inline expressions. Over time, as more and more flexibility and standardization is introduced, these blocks can become dense jungles of operators, functions, and meta-arguments, reducing clarity and maintainability.

The locals{} Block and the local Object

As with any human-generated software project, your TF code should be organized and structured to enhance maintainability. This prevents obfuscating key logic or the proven-to-be-convenient misapplication of techniques. Instead of chaining and jumbling multiple nested expressions, abstract logical expressions into named values that may be composed, where necessary, and referenced where needed.

Enter the `locals{}` block. This block establishes the `local` object, which may be used to assign arbitrary key-value pairs of module-scoped named values, called *local values*. You can assign the result of an expression to a local value and reference this elsewhere in the module, such as a `resource{}` block. See Example 4-8.

Example 4-8. Establishing a local value and referencing it in a resource definition

```
locals {
    name_value = "tas-4-7-network-name"
}
```

```
resource "google_compute_network" "vpc" {
    name = local.name_value
}
```

It's recommended and common to have one `locals{}` block per module. If all component blocks are in the *main.tf* — and the `locals{}` block isn't too many lines — these can be too, ideally at the top of the file. However, it's typically most appropriate to put the `locals{}` block in its own *locals.tf* file. Name your locals well; consider *self-documenting code*, where the names tell what relationship or purpose the generated variable addresses. As discussed previously, the `locals{}` block is the most appropriate place for complex calculations or constructions, including restructuring resource attributes or the outputs of third-party remote modules, as shown in Example 4-9.

Example 4-9. Using local values to restructure the output values of third-party modules

```
locals {
    outputs_map  = { for el in module.remote.element_list : el.name
=> el }
    output_names = [ for el in module.remote.element_list : el.name
]
}
```

> **TIP**
>
> The `locals{}` block may also be the best place to hardcode nonvariable values that should be referenced in a `resource{}` or `module{}` block. This is especially effective in modules that are intended to be consumed by third parties (including other internal teams that use the modules in a downstream project), to ensure a required field is set to a specific value.

Organizing Local Values

In terms of organization within a `locals{}` block, there may be a number of valid options. TF doesn't impose any order on local values, and local values can reference each other, which can result in effectively random order, depending on how and when values are introduced. For example, if

new values are always added to the bottom of the `locals{}` block, unrelated values may be found shuffled together, without an obvious pattern or order. Such projects become just as opaque and confusing as those that omit the `locals{}` block altogether and leave all expressions inline with the resource configurations.

Instead, develop a style or formatting guide for constructing local values. For example, it may be appropriate to group related local values together and order them within a group such that downstream values (those that reference another local value) appear after their references, as depicted in Example 4-10. These clusters act as markers for related configuration values. If desired, they may also be prefaced with a comment to describe which resource, module, or so on, the local values apply.

Example 4-10. Example of grouping and ordering multiple local values

```
locals {
    # Subnet values
    first_10_regions =
slice(data.google_compute_zones.avlble.names, 0, 10)

    subnets = { for reg in local.first_10_regions :
        reg => { name: "snet-${reg}", region: reg }
    }

    # Compute values
    vm_nics        = [ for snet in local.subnets : snet.name ]
    vm_name        = "vm-${local.first_10_regions[0]}"
    vm_description = "Description for ${local.vm_name}"
}
```

The preceding example may be great for small projects or those that heavily use module nesting because the list of values might not extend much beyond a screen or so of configuration code. However, as `locals{}` blocks get filled with more and more values, it may be appropriate to change the order the values appear in the block, again, dictated by a well-defined style guide. One solution might be to elevate all the local values that are referenced by resources, modules, and so on (those that aren't merely used as a step in the calculation and are referenced only by other local values), to the top of the `locals{}` block and then group the

calculations in reverse order, as shown in Example 4-11, making it easy to follow the logic from resource to component expressions.

Example 4-11. Elevating the values referenced by `resource{}` *blocks to the top*

```
locals {
  # Used by resources
  subnets         = { for reg in local.first_10_regions :
    reg => { name: "snet-${reg}", region: reg }
  }
  vm_nics         = [ for snet in local.subnets : snet.name ]
  vm_name         =
"vm-${local.first_10_regions[0]}-${local.image_type}"
  vm_description = "Description for ${local.vm_name}"
  ...

  #Used by other locals
  first_10_regions = slice(data.google_compute_zones.avlble.names,
0, 10)
  image_type       =
data.google_compute_images.avlble.images[0].self_link
  ...
}
```

Clever or misapplied logic (that which is leveraged for something other than its express intent, as a "hack") in `locals{}` blocks should be used sparingly. There's little benefit to preserving configuration that's difficult to comprehend, regardless of its effectiveness. It's highly recommended to decompose any complex logic into component expressions and statements that may be individually referenced in a "self-documenting code" fashion with thoughtfully named local values. The values in the `locals{}` block will describe how even very complex calculations are determined by unwinding the compositions into their component parts, following the chain back to the most basic calculation elements.

Meta-Arguments

Meta-arguments appear alongside the fields of modules, resources, data sources, or dynamic blocks, but they have special, unique properties that

affect the behavior of the surrounding element, depending on the value assigned to them.

The primary meta-arguments include the following:

`for_each`

Perhaps one of the most useful and common of the meta-arguments, `for_each` enables iteration at a block level.

`count`

The `count` meta-argument is best used as a — rather clunky, but reliable — way to implement optional or conditionally invoked blocks.

`depends_on`

Using `depends_on` will establish an explicit dependency between elements and may be used to ensure the referenced elements are configured before the one that contains the `depends_on` phrase.

`lifecycle` *nested block*

The `lifecycle` nested block may be used to establish custom lifecycle rules — creation and destruction — for resources.

`provider`

The `provider` meta-argument is also specific to resources and data sources and provides a way to select between different `provider{}` block configurations or versions thereof.

`providers`

The `providers` meta-argument is specific to modules and describes which providers are available in the module.

In the next section, you'll get a deeper view of each of these and guidance on how to best implement them.

Effective Use of for_each, count Meta-Arguments

The `for_each` meta-argument takes a collection as its assigned value, and for each element of the collection, it will configure a copy of the block in which it appears — in the resource, module, and so on — according to the rest of the configured fields. The elements in the collection may be referenced using a special `each` object, which exposes the fields `key` and `value`. For map-like types assigned to `for_each`, the `each` object's `key` and `value` fields are assigned the corresponding key and value from the iterated element in the map. For list-like types (really, they must be set-like), the `each` object's `key` and `value` fields will match and both be assigned the iterated list element itself.

PRIMITIVE ELEMENTS FOR SET VALUES

For the list/set-like types, the elements within the set must be of a primitive type. This is due to how the resource (or module, etc.) is identified within the TF state (which will be discussed in great detail in Chapter 6). In brief, the resource is assigned a unique address that is determined by the chain of TF elements — that is, the modules that were traversed to get to the resource and the resource itself. For elements that are modified with the `for_each` meta-argument, the address includes the same phrase as whatever is assigned to the `each.key` element: the key itself for a map and the value itself for a set (this also explains the requirement of a set, rather than the more general list: the values must be unique so the addresses of their configured TF element is guaranteed to be distinct). Similarly, for `count` (as we'll review in further detail, below), the TF state address of a configured element will include the index of the count's iteration.

You've already seen how `for_each` can be used in tandem with a `for` expression to iteratively configure a collection of resources. As discussed previously, these two components, while related conceptually and often found together, are distinct and may be used independently. Moreover, even

when a `for` expression is implemented specifically to create a value to be assigned to `for_each`, this `for` expression may best be served in a `locals{}` block and assigned as a reference.

Often, the collection you'd like to iterate over is a list (set) of map-like items. This is especially common when reading the output of another TF element that has the `for_each` meta-argument applied to it, as seen in Example 4-12.

Example 4-12. Incorrectly referencing the output of an iterated `resource{}` block

```
resource "google_compute_instance" "gce_nodes" {
    for_each = var.node_names
    …
}
…
resource "google_dns_record_set" "vm_a_record" {
    for_each =
google_compute_instance.gce_nodes[*].network_interface
    # INVALID - returns a list of non-primitives; disallowed for
for_each
    # resulting format: [
    #     { network_ip: string, access_config: list, ... },
    #     { network_ip: string, access_config: list, ... },
    #     ...
    # ]
```

To get the result you want, use a `for` expression to generate a map with a unique identifier as each of the keys (see Example 4-13). This also ensures the resulting information stored in the TF state is recorded in a predictable and deterministic (and therefore more robust) fashion, as the keys will be evaluated in lexicographical order.

Example 4-13. Generating a map from a list using the `for` expression

```
resource "google_compute_instance" "gce_nodes" {
    for_each = var.node_names
    …
}
…
locals {
    vm_nics_map = { for vm in google_compute_instance.gce_nodes :
        vm.instance_id => vm.network_interface
```

```
    }
    # resulting format: {
    #     instance-id-1: { network_ip: string, access_config: list,
... },
    #     instance-id-2: { network_ip: string, access_config: list,
... },
    #     ...
    # }
}
…
resource "google_dns_record_set" "vm_a_record" {
    for_each = local.vm_nics_map
    # VALID - each key is unique, includes all necessary
information
```

> **NOTE**
>
> It's not recommended that the `locals{}` block appear among the `resource{}` blocks, as depicted in the example. This is for illustrative purposes only.

Using a `for_each` to configure a large or variable set of the same component (e.g., `resource{}` or `module{}`) block is certainly superior to trying to duplicate full component blocks, especially as you manage dozens or hundreds of components this way. However, if you know you will only ever have a small (two or three), static number of resources — for example, an active/passive setup for a particular tool to stay available during outages or help recover from disaster — it may not be worth the loss of clarity and extra syntax to implement using `for_each`. Also, to move from one paradigm to another may be any of the following undesirable qualities: destructive, in the sense that resources may get recreated, or cumbersome, by using the `tf state rm` and `tf import` commands to align the TF state object with the resulting shape expressed by the new rendering of the code, as shown in Chapter 7. This requires extra care, especially for any components or appliances that are part of your security infrastructure, foundational in nature (that is, destroying it would cause, or require, the destruction of other resources that rely upon it for their existence), or that otherwise require extra care.

Similar to the `for_each` meta-argument, the `count` meta-argument is used to configure a variable number of resources (or modules, etc.) in an iterative fashion. It also exposes a special `count` object that includes an `index` field, for referencing the current iteration. However, rather than a collection, `count` takes a number as its assigned value and will configure that number of copies of the block. While this seems innocent enough, it's well established that the `count` meta-argument can become problematic, especially for resources that may at some point need to be permanently deleted, not recreated. For example, suppose you're tasked to create a set of 10 subnets with prescribed region placement. Example 4-14 shows the difference of using `count` instead of `for_each`.

Example 4-14. Comparing `count` *with* `for_each` *for creating 10 subnets*

```
resource "google_compute_subnetwork" "snets_count" {
    count  = 10
    region = var.subnet_regions[count.index]
    …
}

resource "google_compute_subnetwork" "snets_for_each" {
    for_each = var.subnet_regions
    region   = each.value
    …
}
```

Each of the constructs presented in this example will result in 10 identically configured pairs of subnets. However, if the regions are ever updated and the new set of regions has a different lexicographical order than before, the `count` construction will recreate all reordered elements; the `for_each` version will simply remove the unneeded resource and create a new resource for the updated region entry.

> **TIP**
>
> As shown in Example 4-14, `count` may result in inconsistent or especially nondeterministic behavior. This is because `count` necessarily uses consecutive whole numbers — 0, 1, 2, and so on — as its indices for the resulting resource set, and there's no way to remove arbitrary elements. Only elements from the end of the set may be removed, by reducing the `count`'s value. It's therefore recommended to avoid the use of `count` for configuring multiple copies of a particular `resource{}` or `module{}` block and to only use the `for_each` meta-argument construct.

However, there's another common use of the `count` meta-argument that may be much more appropriate to introduce: as a way to optionally or conditionally configure a `resource{}` or `module{}` block. In this way, the only valid values allowed to be assigned to `count` are `1` and `0`, typically evaluated using the ternary conditional expression and indicating that the resource should or should not be configured, respectively. Example 4-15 shows this construction to optionally configure a DNS entry as part of a remote network module. The result will be either the presence of a list with a single element or nothing, so the output of a resource conditioned like this requires the use of the list index notation. Referencing this output will also typically require the use of another ternary or the splat expression, spreading complexity into other parts of the codebase.

Example 4-15. Optionally creating a DNS entry as part of a remote network module

```
#-----------------------------------------------------------------
----------
# Root module
#-----------------------------------------------------------------
----------
#
///////////////////////////////////////////////////////////////////
////////
# main.tf
#
///////////////////////////////////////////////////////////////////
////////
module "remote" {
  source = "remote-module-tas-4"
```

```
  dns_domain = var.domain
}

#-----------------------------------------------------------------
----------
# Remote module
#-----------------------------------------------------------------
----------
#
/////////////////////////////////////////////////////////////////
////////
# main.tf
#
/////////////////////////////////////////////////////////////////
////////
resource "google_dns_managed_zone" "special" {
  count = var.dns_domain != "" ? 1 : 0

  name     = "special-zone"
  dns_name = "special.${var.dns_domain}."
}

resource "google_dns_record_set" "a_record" {
  count = var.dns_domain != "" ? 1 : 0

  name         = var.dns_domain
  type         = "A"
  managed_zone = google_dns_managed_zone.special.name
}

#
*****************************************************************
********

#
/////////////////////////////////////////////////////////////////
////////
# outputs.tf
#
/////////////////////////////////////////////////////////////////
////////
output "dns_record_name" {
  value = (length(google_dns_record_set.a_record) > 0
  ? google_dns_record_set.a_record[0].name : "")
}
```

Moving from a component block that uses `count` to one that uses `for_each` — or one with neither — is generally destructive. This may be mitigated using `tf import`, as discussed in Chapter 7.

> **TIP**
>
> It's best to use `count` only to construct conditional blocks. As shown, assigning any value to `count` other than `0` or `1` can introduce unwanted behavior. Also, avoid using `count` even for its conditional capabilities in *root modules* (those that are the direct subject of a TF command, such as `tf plan`), especially those that service a single TF state (more complex arrangements are explored in detail in Chapter 7).

Effective Use of depends_on Meta-Argument

Dependencies between resources within a particular TF state may be established implicitly through the reference resource attributes or explicitly with the `depends_on` meta-argument. Perhaps counterintuitively, the implicit formation is the preferred option.

As its name suggests, the `depends_on` meta-argument creates a unidirectional dependency between two TF entities in cases where such dependencies cannot be directly inferred by TF. This might be necessary to ensure that resources are created in a specific order so that creating or configuring the dependent resource doesn't fail. It also serves to prompt TF for lifecycle-related tasks, such as when a resource that must be recreated has other resources it depends on; TF will do its best to plan out an ordered set of operations that account for the dependency's destruction and reestablishment.

The `depends_on` meta-argument should be included only where expressly needed. In many cases, there are better solutions, such as refactoring how the various modules are constructed and organized, as depicted in Example 4-16. Moreover, including `depends_on` where it isn't strictly necessary may incur unnecessarily complex plan evaluations and can even cause evaluation cycles, which will result in an error.

Example 4-16. Refactoring to remove a `depends_on` *phrase in favor of an implicit dependency*

```
#-----------------------------------------------------------------
----------
# Before refactor
#-----------------------------------------------------------------
----------
#
/////////////////////////////////////////////////////////////////
////////
# main.tf
#
/////////////////////////////////////////////////////////////////
////////
resource "google_compute_network" "vpc" {
  name = var.network_name
}

resource "google_compute_instance" "instance" {
  name         = var.instance_name
  machine_type = var.machine_type
  zone         = var.instance_zone

  boot_disk {
    initialize_params {
      image = var.boot_image
    }
  }

  network_interface {
    network = var.network_name

    access_config {
      // Ephemeral public IP
    }
  }

  service_account {
    email  = google_service_account.default.email
    scopes = ["cloud-platform"]
  }

  depends_on = [google_compute_network.vpc]
}

#-----------------------------------------------------------------
```

```
----------
# After refactor
#-----------------------------------------------------------------
----------
#
/////////////////////////////////////////////////////////////////
////////
# main.tf
#
/////////////////////////////////////////////////////////////////
////////
resource "google_compute_network" "vpc" {
  name = var.network_name
}

resource "google_compute_instance" "instance" {
  name         = var.instance_name
  machine_type = var.machine_type
  zone         = var.instance_zone

  boot_disk {
    initialize_params {
      image = var.boot_image
    }
  }

  network_interface {
    network = google_compute_network.vpc.name

    access_config {
      // Ephemeral public IP
    }
  }

  service_account {
    email  = google_service_account.default.email
    scopes = ["cloud-platform"]
  }
}
```

The lifecycle Meta-Argument and Custom Conditionals

As your TF systems work to maintain the resources you encode, it will create, destroy, or update resources, in that order, depending on the changes made to the code or the running resources. Furthermore, for resources that

TF determines will require an update, it will segment the changes into those it can do in place and those that will require the resource to be destroyed and then recreated, running all of the in-place updates first. In many cases, TF projects can be organized and constructed such that the basic lifecycle rules will sufficiently support most resources. However, there are likely to be special cases — such as a specialized networking or security software appliance — that would be enhanced or reinforced by implementing specific lifecycle controls, such as preventing its careless destruction or recreating a duplicate before it's destroyed. In such cases, TF supports a `lifecycle` meta-argument that manifests as a nested block and supports rules such as `prevent_destroy` and `create_before_destroy`.

Several of the `lifecycle{}` nested block's fields are available only for `resource{}` blocks. These correspond to rules for how and when to create or destroy a resource. The following are the lifecycle rules specific to resources and general rules for when they are most appropriate:

The `create_before_destroy` *rule*

> This rule is useful for resources that cannot have downtime and are designed to not clash if they were to be duplicated, as encoded in TF. This is not appropriate for configurations that rely on specific identifying or determining attributes that cannot be assigned to multiple resources in the given CSP. For example, a compute resource that acts as part of a collection of stateless servers load balanced based on a network tag or IP range, so that a new copy of the instance would be entered into the pool without harm, but a recreate across the collection would likely cause an undesirable disruption in service.

The `prevent_destroy` *rule*

> In cases where an errant destroy might cause a great deal of damage or severely expose sensitive material, the `prevent_destroy` rule can be used to prevent TF from allowing a resource to be destroyed either directly or via the recreate mechanism. When TF gets to the operation, it will error out. Ideally, you're scouring the results of a planning phase

to ensure that the changes TF is reporting are what you expect it to do. In which case, this lifecycle rule should in theory be unnecessary and their inclusion may be more symbolic than truly strategic.

The `ignore_changes` *rule*

Using this rule allows you to restrict TF from updating resources too aggressively. This may be especially useful for resources that would need to be recreated if the specified fields are modified. For example, there may be a system external to TF that's allowed to modify certain attributes of a compute resource and that need to be preserved, despite it changing the resource's state according to TF.

The `replace_triggered_by` *rule*

The near opposite rule, `replace_triggered_by`, causes TF to force the resource's recreation in the event that any of the specified resource fields are modified — this is in addition to whatever other resource fields would trigger a replace if modified, by default. This may be useful in cases where you consider the resource to be corrupted in whatever way if the specified field is changed, or as part of a deployment strategy that treats resources as immutable and favors recreating them to updating them.

Generally speaking, the resource-specific lifecycle rules should be used sparingly and considered on a case-by-case basis. They should be motivated by a business or technical justification and should have corresponding policy statements published in an appropriate source of truth for such policies. For example, a decision to introduce the `replace_triggered_by` rule in a blanket fashion to enforce immutability should have significant real estate in a Technical Design Document (TDD), or similar document, and be available in your corporate wiki site.

In addition to establishing custom resources lifecycle rules, the `lifecycle{}` nested block allows you to include resource-level

validation through the `lifecycle{}` nested block's own `precondition{}` and `postcondition{}` nested blocks. These work much like the validation support in input variables and output values, including the same required fields: `condition` and `error_message`. The interesting thing about these expressions is that they're evaluated before and after the resource gets created or updated, respectively. As such, the `precondition{}` nested block is an opportunity to ensure any variables or data sources are available and appropriate before attempting to create the resource. Moreover, the `postcondition{}` nested block allows for validation of the created resource to ensure it's configured properly. This is especially useful for attributes that aren't directly managed by TF and are instead established by a third-party system, possibly the CSP itself, by a shell script, or other such system. Review "Validation of Variables" for more information on how the `condition` and `error_message` fields work.

Unlike most of the other meta-arguments, `precondition{}` and `postcondition{}` nested blocks are available in the `lifecycle{}` nested blocks of `data{}` and `ephemeral{}` blocks. One thing that's unique to the `postcondition{}` nested block is support for a convenient `self` object that represents the resource entity (or data source or ephemeral entity), and may be used to check the resource's reference attributes, as shown in Example 4-17.

Example 4-17. Using the special `self` *object within a* `postcondition{}` *nested block's* `condition` *field*

```
resource "google_compute_instance" "instances" {

    ...

    lifecycle {
        postcondition {
            condition = (self.network_interface.network !=
"default"
                && length(self.scratch_disk) < 2)
            error_message = "No default network; at most 1 scratch
disk"
        }
```

```
    }
}
```

The final field of the `lifecycle{}` nested block is the `destroy` field, which is only allowed to appear in `removed{}` blocks. The `removed{}` block itself is the `resource{}` block, but with the keyword `resource` replaced by the keyword `removed` — leaving the remainder of the block, including the resource type and name, intact. If the `destroy` field is set to `true` in the `lifecycle{}` nested block of the `removed{}` block when it's first created — that is, the first time a `tf apply` operation is undertaken after the `resource{}` block gets changed to a `removed{}` block — it will also destroy the resource. This may prove to be useful, for example, in the case of an audit practice that has such resource configurations preserved for a certain period of time or indefinitely. If the `destroy` field is instead set to `false` in the `lifecycle{}` nested block of the `removed{}` block when it's first created, TF will remove the resource from the TF state *only*. Thus, the resource itself does not get destroyed, it simply becomes unmanaged by TF and will no longer be included in operations such as `tf plan` or `tf apply`. This may be useful in cases where certain resources are initially generated by one team and later gifted to another, perhaps as part of a controlled adoption practice for certain experimental managed services.

> **NOTE**
>
> Once a `removed{}` block is successfully applied, the `destroy` field may not be modified. This explains the belabored refrain of *when it is first created*. In the case where it's initially `true`, the resource is destroyed (and removed from the TF state as an automatic result). In the case where it's initially `false`, the resource becomes unknown by TF and thus may not be managed in any way, unless it's first reimported into the TF state. Finally, omitting the `lifecycle{}` nested block and/or the `destroy` field is equivalent to setting the `destroy` field to `true`.

The provider Meta-Argument

The `provider` meta-argument is used in conjunction with the `provider` block: whereas the latter establishes, for example, the CSP, platform, or version of the software development kit (SDK), the former meta-argument selects between these by alias. This might be used to construct modules that support multiple CSPs, as shown in Example 4-18, or different versions of a particular provider plug-in.

Example 4-18. Supporting multiple CSPs and different versions of a particular provider

```
#
////////////////////////////////////////////////////////////////////
////////
# providers.tf
#
////////////////////////////////////////////////////////////////////
////////
terraform {
  required_providers {
    google = {
      source  = "hashicorp/google"
      version = ">= 7.4.0"
    }
    aws = {
      source  = "hashicorp/aws"
      version = ">= 6.14.0"
    }
  }
}

provider "google" {
  project = "same-project-id"
  region  = "us-west2"
}

provider "aws" {
  region = "us-west-2"
}

#
********************************************************************
********

#
////////////////////////////////////////////////////////////////////
////////
```

```
# main.tf
#
////////////////////////////////////////////////////////////////////
////////
resource "google_compute_network" "gcp_vpc" {
  provider = google

  name = var.network_name
}

resource "aws_vpc" "aws_vpc" {
  provider = aws

  cidr_block = var.cidr_block

  tags = {
    Name = var.network_name
  }
}
```

> **TIP**
>
> Consider remote modules that support a single CSP (although it may be appropriate for a single remote module to support multiple versions of a specific provider, as described next). Modules that themselves support multiple CSPs are likely to be specific to your needs and may be more appropriate to be stored locally, rather than maintained as remote modules.

A common use case is to leverage a beta (or even alpha) version of a particular provider plug-in to build a preview or newly released feature or service not yet included in the generally available provider. A *default provider* (one without an alias configured) may be established, and alternate providers are constructed so they can be referenced in the resources that require them.

Using the `provider` meta-argument may seem strictly necessary in certain cases, and indeed it may be the only way for a single TF state to manage multiple clouds. However, in many cases it may be more appropriate to simply manage these different CSP resources in different TF states (and perhaps even projects/repositories) to maintain a separation of

concerns between the two. After all, a typical goal of implementing a multicloud infrastructure is to take advantage of the enhanced availability and recovery, which is somewhat thwarted if the resources are combined together into the same TF state.

The dynamic Block

Some resource configurations are actually expressed as nested blocks. Moreover, some such nested blocks may be repeated to establish multiple distinct configurations. In such cases, it may be desirable to use a set of configurations to enable a variable number of blocks. This is enabled in TF with the `dynamic{}` block. This special block may be used to iteratively create such nested blocks. The `dynamic{}` nested blocks require special syntax for identifying the specific configuration nested block and for setting the contents assigned within a particular iteration. They are also used along with a `for_each` meta-argument for repeatable, nested blocks or the `count` meta-argument for conditionally included singular nested blocks.

As an example, common repeatable nested blocks for a compute resource are disk attachments and network interfaces. To create a module that could support a variety of different valid configurations, `dynamic{}` blocks can be introduced for each such nested block as shown in Example 4-19.

Example 4-19. Rendering repeated nested blocks as `dynamic{}` *blocks*

```
#
////////////////////////////////////////////////////////////////////
////////
# terraform.tfvars
#
////////////////////////////////////////////////////////////////////
////////
disk_configs = {
    "dsk-primary": { disk_name: "dsk-primary", device_name: "dev-
primary" },
    "dsk-backup": { disk_name: "dsk-backup", device_name: "dev-
backup" }
}
nic_configs = {
    "snet-dev": { subnet_name: "snet-dev", public_access: true },
```

```
    "snet-admin": { subnet_name: "snet-admin", public_access: false
}
}

#
****************************************************************
********

#
////////////////////////////////////////////////////////////////
////////
# main.tf
#
////////////////////////////////////////////////////////////////
////////
resource "google_compute_instance" "gce_vm" {
    …
    dynamic "attached_disk" {
        for_each = var.disk_configs

        content {
            source      = attached_disk.value.disk_link
            device_name = attached_disk.value.disk_device_name
        }
    }

    dynamic "network_interface" {
        for_each = var.nic_configs

        content {
            subnetwork = network_interface.value.subnet_name

            dynamic "access_config" {
                count = access_config.value.public_access ? 1 : 0

                content {
                    # Intentionally left blank to generate
ephemeral IP
                }
            }
        }
    }
}
```

As shown in the example, the `dynamic{}` block can be nested, if necessary, and can thus establish extremely complicated encodings. Use

them only where it's truly useful, and develop them with effective guardrails, for example, through the use of input variable `validation{}` blocks.

> **NOTE**
>
> As is shown in Example 4-19, each `dynamic{}` block creates a block-scoped object that shares a name with the dynamic block itself, scoped to the `content{}` nested block of each `dynamic{}` block. This means the block-named object is related to the outer block's `for_each` and may be used in the nested `dynamic{}` block's `for_each` expression to dig deeper into a particular configuration — for example, a complex input variable such as one composed of nested `map()` and `object()` constructors.

The `dynamic{}` nested block is commonly found in third-party remote modules to enhance flexibility. These may be the most appropriate places for them to appear. While this flexibility may seem to make it worthwhile to include `dynamic{}` blocks anywhere it might be useful — that is, any place where a nested configuration block is repeated — it's often unnecessary. They should be included sparingly and judiciously, and never included to future-proof (i.e., overengineer) a module. A small, static number of a particular repeated nested block may best be served with individual such blocks.

The `dynamic{}` nested block may also be appropriate to include within `resource{}` blocks that are subjected to a `for_each` statement, especially if there's variation in the relevant nested configuration blocks among the collection of individual resources it's meant to configure–for example, a set of firewall rules that may or may not support multiple protocols, as depicted in Example 4-20.

Example 4-20. Using `dynamic` nested block to configure set of firewall rules with variable protocols

```
resource "google_compute_firewall" "internal_allow_firewalls" {
    for_each = var.firewall_configs
```

```
    name    = each.value.firewall_name
    network = google_compute_network.internal_network.name

    dynamic "allow" {
        for_each = each.value.allow_configs

        content {
            protocol = each.value.protocol
            ports    = length(each.value.ports) > 0 ?
each.value.ports : null
        }
    }
}
```

> **NOTE**
>
> It's worth noting that the previous example requires the input variables to be constructed as maps. Typically, such configuration would be established as a set of similar configuration objects, and the common exercise is to first use the `for/in` expression with curly braces to turn the set into a map for use by the `for_each` statement.

Clean TF

One of the biggest challenges in writing and organizing TF configuration is that there's almost unlimited variability in how sufficiently complex infrastructure may be encoded. The most effective way to account for this is to establish and enforce consistent standards for how blocks and values may be constructed and where they may appear. Rather than focusing on shipping only working code, standardize the format and structure of your TF configuration.

Regardless of how your application workloads are developed, your TF should be developed in a highly deliberate and forward-thinking way. It's also worth remembering that regardless of whether it's designated for a "development" or "integration" environment, all of your infrastructure must be considered production-ready from your perspective and for your purposes. It could be considered an anti-pattern to implement IaC development in a typical agile methodology such as scrum, XP, or Scaled

Agile Framework (SAFe), as we've shown how it explicitly violates the agile manifesto's tenets to prioritize working code and speed to market. A methodology that has many similar components and is highly appropriate for IaC development is Kanban.

Your infrastructure is meant to establish and secure the various environments needed for application development. The robustness of your infrastructure, and of the code that maintains it, is what enables high-velocity agile application teams to move fast and break things (in a way that doesn't expose you to financial or legal jeopardy).

Organizing Field Values

Because of the interpreted nature of HCL configuration, TF doesn't have strong rules in terms of where resource fields need to be placed within a block. It also has fairly loose expectations around whitespace. As a result, it's totally up to the configuration's authors to determine and enforce any standards or styles.

A typical and serviceable order of statements for any TF block, an example of which is in Example 4-21, is as follows:

The `for_each` *or* `count` *meta-argument (if needed)*

> If either of these are introduced, turning the resource into a list of resources, it should typically be found at the top, thereby signifying the iterative or conditional nature of the configuration.

The `provider` *meta-argument (if needed)*

> If different providers are leveraged, either to support multiple CSPs or to support different API versions thereof, they should typically be communicated near the top of the block for clarity.

All one-line resource fields

> These should be directly under the iteration or conditional and provider specifications and will be the primary configuration elements of the

resource. Order may be largely considered on a case-by-case, or resource type-by-resource type, basis, but it's common to elevate identifying (such as name or description) and placement (such as CSP region) fields toward the top of the resources fields. It's also common to place required fields above optional fields, as determined by the provider.

All multiline resource fields

These could be any collection type such as `list`, `map`, or `object`.

Any plain nested blocks

After the single-line and multiline resource fields come the resource-specific nested block configurations that are not subject to a `dynamic{}` wrapper. These would typically have their own fields and would generally follow the same order as depicted here for the top-level blocks.

Any `dynamic{}` *nested blocks*

The next section would include the nested blocks configured within a `dynamic{}` nested block.

The `depends_on` *meta-argument (if needed)*

The `depends_on` meta-argument can be placed below the typical resource-specific configuration, as it's used to establish dependencies between different resources within a TF project, rather than configure anything specific to the resource that contains it.

The `lifecycle{}` *nested block (if needed)*

Finally, because lifecycle rules are primarily the purview of the operators of the TF system, the `lifecycle{}` nested block may be placed at the bottom, wherever it's needed. This way it does not separate, for example, any of a `resource{}` block's configuration but is treated as a boundary term to affect the operation of the resource.

Example 4-21. A typical organization of fields in a `resource{}` *block*

```
resource "google_compute_instance" "instances" {
    for_each = local.instance_map
    provider = google-beta

    name         = each.value.name
    machine_type = each.value.machine_type
    zone         = var.instance_zone
    tags         = each.value.network_tags

    labels = {
        created_by = "terraform"
        bu_code    = "amer-eng"
    }

    boot_disk {
        initialize_params {
            image = each.value.image
        }
    }

    service_account {
        email  = google_service_account.default.email
        scopes = ["cloud-platform"]
    }

    dynamic "network_interface" {
        for_each = each.value.network_names

        content {
            network = each.value

            access_config {
                // Ephemeral public IP
            }
        }
    }

    dynamic "scratch_disk" {
        for_each = each.value.scratch_disk ?
[each.value.scratch_disk] : []

        content {
            interface = "NVME"
        }
    }
```

```
    lifecycle {
        create_before_destroy = true
    }
}
```

In terms of whitespace, there may be less consensus and fewer defined camps regarding how it should be handled. Certainly, one common pattern, as shown previously, is to add a blank line between the various sections, such as between the initial meta-arguments and the single-line fields, as well as surrounding the nested blocks. Another common pattern — one that is used heavily in the examples in this book — is to align the predicates of resource field expressions (especially among the single-line fields) such that all the equals signs (=) align. The command `tf fmt` will do this for you automatically.

These formatting guidelines will ensure configuration blocks are accessible by anyone who should come across them. Without any such guidelines, sufficiently complex configuration can become overly confusing with fields interspersed among nested blocks, potentially hiding certain configuration, especially if whitespace is not well formatted. There may be a good reason to modify or refine this approach, for example, to elevate a `lifecycle{}` block or `depends_on` instruction, but this should not be done haphazardly. Your organization should have published guidelines and should allow exceptions only in well-documented and justified cases.

Self-Documenting Elements

As you've seen elsewhere in this chapter and book, it's a good practice to take a *self-documenting* approach to TF projects, as presented by Robert C. Martin in his books, most prominently, *Clean Code* (Pearson, 2008). This means using the TF element names you give to `resource{}` blocks, `output{}` blocks, and so on, to describe its purpose, rather than relying on comments, code words, or tribal knowledge, all of which may become corrupted or experience lossiness over time.

HashiCorp's official recommendation is to use *snake-cased* element names instead of camel-cased or kebab-cased, which means lowercase strings

separated by an underscore character. This does somewhat limit the expressiveness that might be desired for certain values that represent the result of especially complex computations. Of course, if the names of values and entities become overly lengthy, they too begin to pollute the `resource{}` block real estate, so it's truly important to balance expressiveness with brevity.

In addition to expressive naming conventions, and as presented previously in "Local Values", arbitrary logical expressions may be assigned to local values that allow them to be decomposed into single operations or simpler combinations thereof. This can be especially important for keeping the `resource{}` blocks clean from dense, convoluted logic, which are the most important elements in a TF project to understand.

One convention that may be introduced for certain elements, such as local values, is to put something of an alias value. This way, you may establish arbitrarily expressive local values, and rather than reference these directly, you establish another, much more compactly named local value that you reference instead, as depicted in Example 4-22.

Example 4-22. Setting an alias local value for an uncommonly long local value name

```
locals {
  map_from_item_list  = { for item in var.items : item.name => item
}
  inverted_output_map = { for key, val in instance.prod.map : val
=> key }
  merged_inverted_map = merge(local.map_from_item_list,
local.inverted_output_map)
  final_map           = local.merged_inverted_map
}
```

Summary

In this chapter, you learned about many of the various value-holding components within TF, including the following:

- The relationship between resource fields and reference attributes

- How to construct output values and leverage them strategically
- Successful use of the `locals{}` block and how to organize local values
- The `for_each` and `count` meta-arguments that may be used to respectively iterate over, or conditionally create, configuration elements and manifests as a list of resources in the TF state
- The `depends_on` meta-argument and potential ways to refactor to strengthen dependency relationships
- The `lifecycle{}` meta-argument nested block that can control TF lifecycle events for a resource, data source, or ephemeral entity that contains it
- The `precondition{}` and `postcondition{}` blocks that may be used to validate resource before and after they are created or modified, respectively
- A typical way to organize the fields of configuration blocks
- Techniques for introducing self-documenting features to TF configuration elements

You have seen all the ways values may be assigned in TF projects. You know that not all features are necessary, or even appropriate, for all entities all the time. You've seen examples of how to refactor code that is convoluted, clumsy, or overly cryptic into code that is self-descriptive and easily accessible.

In the next chapter, you'll get a deep dive into how to construct modules in TF that are well suited for use as submodules. These may be local to the repo or stored in a remote artifact registry and may be referenced by their location in a `module{}` block's `source` field. You'll learn several techniques for module development that may be used to repurpose, combine, or otherwise mold submodules to your needs.

Chapter 5. Developing Modules

In software development, a common point of discussion is around scopes and contexts. They shape how we organize, protect, and reuse code — reducing ambiguity and minimizing friction during development, execution, and maintenance.

Up to this point, we've focused on two primary scopes in TF: expressions and blocks. Many times, these are indeed the important scopes to consider. However, there's another major scope: the module.

In TF, everything is encapsulated within a module, so at all times, you're developing a module in some fashion. It could be oriented around a number of different aspects of a software project, including the cloud services that host and service your application stack as well as the applications themselves.

In this chapter, you'll see different types of modules and how they may be used strategically to create effective TF projects that follow the shape of your organization. You'll learn how to develop reusable modules for use by other members of your organization or even the general public. Finally, you'll get an overview of useful practices that, with thoughtful application, may be combined into a successful codebase — and practices to avoid or be especially judicious about.

Types of Modules

As stated, everything in a TF project is encapsulated within a module. This has implications both where certain information is utilized and how information may be abstracted or protected.

Here is an overview of the various types of modules and their general properties:

Root module

One that is the immediate target of a particular TF command, such as `tf init` or `tf plan`. These may include `module{}` blocks that would introduce nested modules as child elements in the state object, as described in greater detail in Chapter 6.

Nested module

Referenced by another module's `module{}` block's `source` field and provides a way to bundle multiple resources together into a single, repeatable solution.

Nesting module

Has one or more `module{}` blocks referencing other modules. Also typically referred to as an *outer module* or *referencing module* when discussing it in reference to the nested module itself.

Leaf module

Extending the tree metaphor, a leaf module has no `module{}` block (but ostensibly does have at least one `resource{}` block). Also may be considered a *terminating module*, as it terminates the module chain from root to resources.

Mixed module

Has both `resource{}` and `module{}` blocks. If used to include one or a small number of `resource{}` blocks into a module that is primarily used as a nesting module, this might be referred to as a *bolt-on module*. These should generally be avoided, as we'll see in the section "Things to Avoid".

Unitary module

This is a module with only one `resource{}` or `module{}` block defined. These should be used sparingly and are discussed in the section

"Additional Considerations". This might also be referred as a *wrapper module* for those with a single `module{}` block or as a *singleton module* for those with a single `resource{}` block.

Remote module

A nested module whose source code is managed in a different repository than the one that references it using the `module{}` block's `source` field.

Local module

A nested module whose source code is managed in the same repository as the one that references it using the `module{}` block's `source` field.

Logic module

Doesn't record any resources to a TF state. A logic module includes only `variable{}` blocks, `output{}` blocks, and an optional `locals{}` block, and is intended to abstract logic expressions away from resource definitions.

Many of these features are not mutually exclusive, and it's typical for a module to be some combination of the types described. However, it's worth discussing how to approach modules for the cases that are diametrically opposed. In the following subsections, you'll get a deeper sense of when to use which type of module.

> **NOTE**
>
> The HashiCorp standard module structure documentation uses the term *root module* to refer to the module that appears at the root of the module's repository. This text will specify this distinction in cases where it arises. In all other cases, the term will specifically refer to the definition in the preceding list, the target of a TF command, and thus the *root* of the TF state. Typically, the root module (as it appears in this text) will indeed appear at the root of some repository. However, in the case of nested modules sourced from remote locations, these will include their own root directories, and these modules will appear along branches of the TF state. In Chapter 6, you'll find greater detail about TF state, including its tree-like structure.

Root Modules

As described, a *root module* is a module that is the target of a TF command and can be thought of as the originating module, from which any other module in use must be referenced (perhaps indirectly). These modules are the only ones that may read their input values from *.tfvars* files, environment variables, command-line settings, or other such mechanisms. These are also the modules that define the backend location for a particular TF state.

Aside from this, there's nothing especially unique about root modules. In fact, root modules may still be leveraged as a nested module in the typical way: by referencing a valid location for the module in the `source` field of another (outer) module's `module{}` block. This may even be leveraged as part of the validation steps for a particular module that is intended to be consumed by a third party (that is, a remote module): create an instance of the module's resources by passing in test values, for example, using a temporary *.tfvars* file to set the input variable values.

> **TIP**
>
> When using the technique described, using a temporary *.tfvars* (or other mechanism) to validate a remote module, it may be worth mentioning `**/*.tfvars` in your *.gitignore* file so that these are not accidentally pushed to the repository. They will be ignored when referenced by outer modules, but this keeps the repository clean from extraneous files. Alternatively, using individual `-var` arguments with the TF CLI would also avoid preserving any such unnecessary content.

It's also worth pointing out that if a root module (M1) — which would necessarily have a stored TF state — is referenced by another module's (M2) `module{}` block as a nested module, the M1's state will *not* be considered or updated by a `tf plan` command that is applied to M2. What's more, it will not necessarily detect any potentially problematic resource attribute collisions until an apply phase. If you choose to introduce this dual-purpose module use — both as a root module and as a nested module — it is advised to tie this decision to a business need or to well-documented business justification. As it is with most things in TF and the wider IaC practice, context is everything: if you're not using the module as a root module only for the purposes of validating its functionality, as described previously, and instead using it as a truly standalone project (along with the nested use), it may be more appropriate to use a wrapper module that wraps this standalone version as the root module — a *wrapper root module*, if you will. This, of course, turns this dual-purpose module into a purely nested module, which frankly may be a more justifiable approach (and certainly a more flexible one, with little, if any, downside).

Leaf Modules Versus Nesting Modules

Nesting modules, those that include `module{}` blocks, behave as organization agents in a TF state, responsible for passing information, including input variables and output values, to, from, and in between its component nested modules. Because of this deliberately disjointed behavior, a nesting module may not clearly reflect the resources that are established by its underlying nested modules. This incurs a level of risk

because the nesting module's developers must maintain a high degree of awareness with respect to these nested modules. Similarly, it's the responsibility of a nesting module's developers to expose the outputs of its nested modules in the output values of the nesting module.

Opposite to nesting modules are *leaf modules*, those that lack any `module{}` blocks, instead only leveraging `resource{}` and `data{}` blocks to configure services in the corresponding service provider or system. These modules are explicit in the resources they manage. It's the responsibility of these module types to expose all useful and relevant information of the resources it manages through output values.

Local Versus Remote Modules

One of the most interesting considerations for a nested module is where the module is sourced from. As presented in the HashiCorp documentation, there are multiple places from which to reference a module. Depending on the expected or permitted use of the module, a particular location may be more appropriate than others. In particular, modules that are presently intended for use as part of a larger module — and not expected to be leveraged by multiple, distinct other modules as a common dependency — may be most reasonably stored in a subdirectory of the very same repository for which it was built and referenced using the (relative) local filepath.

Such *local modules* are, functionally, no different than any other module. However, because a local module is expected to be found alongside other modules within a filesystem, it's highly likely that these modules are also recorded in the same source control, such as a Git repository. It therefore will share a common changeset history and some type of lifecycle or versioning, not to mention access control. In "Local Modules", we discuss the implications and some (possibly odious) workarounds for this common history.

Moreover, the overuse of local modules may lead to severe duplication — at least functionally — of nontrivial configurations. This is especially

plausible for organizations that spread the responsibility of developing TF projects across several teams. As you'll see in the latter half of the book, it's wholly appropriate to introduce this level of overlap, and even redundancy, in terms of skills and responsibilities, but — as you'll also see — it likely cannot be done without significant oversight or without proportional investment. In the section "Additional Considerations", you'll get some recommendations on how to introduce such oversight for several components, including local modules.

In contrast to local modules are *remote modules*. These might be referenced from a number of places including an artifact registry, a Git repository, or a storage bucket in your favorite CSP. The main challenge with remote modules is typically access control: you must have access to read the module from the source location wherever you may be initializing the TF project to use it in a `module{}` block. Additionally, if you want to make any functional changes to the implementation of the remote module, you must have write access to the module's source control. In terms of providing this access, it should be a primary consideration for any module expected or permitted to be used as a remote module. Although the prevailing approach may be more of a laissez-faire perspective, by which the safe, remote use of a module is the responsibility of the referrer rather than that of the referenced, this approach should be taken deliberately, rather than as a result of carelessness. Expose modules to only the audience you expect to use it and choose a preferred location that matches the audience appropriately, with the awareness that you may be exposing them in multiple ways. For example, modules provided in publicly accessible Git repositories should probably also be made available in a public artifact registry, with the latter expressly preferred in the module's documentation, such as its root-level *README.md* file. If this module also has nested, local modules as part of the same Git repository, these local modules will also be exposed via the GitHub or Git references, regardless of whether they're made available in the artifact registry. In "External Distribution" and "Internal Distribution", you'll see some ways to balance such considerations.

Again, there's nothing inherently special about local modules; they certainly may be used as remote modules, and they may even be used as root modules (although the rationale for this is curious indeed and certainly must be supported by a well articulated, business-approved justification). Generally, when referring to local modules, the only thing that really matters is the common source control history it (very likely) has with a nesting, outer module for which it was established. Likewise, the only truly important quality of a remote module is that it (very likely) *does not* share a common history with the nesting, outer module that references it in a `module{}` block. This should be the only meaningful distinction between local and remote modules.

Logic Modules

Logic modules aren't intended to provision or manage any resources. They are meant to capture any expressions that should be abstracted away from the resource definitions.[1] This can be useful for complex validation expressions — with multiple phrases combined together — or in constructing fragile values such as with string interpolation — especially those that are dangerous to update. Normally, such expressions would probably be placed in the `locals{}` block. Extracting the most intricate cases of the logic out of a root module's `locals{}` block and into a logic module can improve the experience for developers, creating a strict separation of concerns. This keeps the root `locals{}` block intelligible and bounded, allowing its use for values that are less fragile and dangerous. Example 5-1 depicts such a module.

Example 5-1. Logic module

```
#------------------------------------------------------------------
-----
# outer module
#------------------------------------------------------------------
-----
module "local" {
  source = "./modules/local"

  app_code      = var.app_code
```

```
  business_unit = var.business_unit
  region        = var.region
}

resource "google_storage_bucket" "example" {
  name     = module.local.safe_name
  location = "US"
  labels   = module.local.safe_labels
}

#-----------------------------------------------------------------
-----
# logic module in ./modules/local
#-----------------------------------------------------------------
-----
output "safe_name" {
  description = "Safe name constructed from the inputs."
  value       = local.safe_name
}

output "safe_labels" {
  description = "Safe labels constructed from the inputs and a
timestamp."
  value       = local.safe_labels
}

locals {
  safe_name   =
"${var.business_unit}-${var.app_code}-${var.region}"
  safe_labels = {
    app_code      = var.app_code
    business_unit = var.business_unit
    region        = var.region
    modified_at   = lower(formatdate("MMM-DD-YYYY-hh_mm",
timestamp()))
                    # for example, 'jan-02-2018-23_12'
                    # note, this will always result in a diff
  }
}
```

Typical reasons to include a logic module are to restrict resource attributes, ensure naming conventions are respected, and enforce other policies. It places all of these expressions into, at the very least, a separate directory. This gives a strong scoping boundary that can be leveraged in access controls and validation mechanisms. This means it can be considered to

have a distinct software lifecycle. Further placing an unusually high standard on making changes to such a module will reduce the chance that it gets misconfigured and reduces anxiety of making changes at all for fear of breaking something delicate and critical. It should also give a strong indication that its contents are not arbitrary and not to be modified frivolously. That being said, achieving these constraints doesn't necessarily require additional tools or processes, but can be as simple as having a strong team structure and robust review process — both of which should be natural goals of any team.

How to Nest a Module

There are many reasons to use a nested TF module embedded within your root module. Nested modules can help organize CSP resources into distinct domains or self-contained, interdependent combinations. There are also plenty of scenarios in which the use of nested modules — or the way in which nested modules are constructed and integrated — is inappropriate. Very likely, with a thoughtful exercise, such as by leveraging something like Domain-Driven Design (DDD) or by constructing an Entity-Relationship (ER) model, whereby modules are constructed according to their strong and weak relationships (and lack thereof), a well-founded arrangement of modules, including module nesting — and in some cases, a total isolation of resources (or strict separation of their lifecycles) — can be achieved that effectively bundles and balances your resources such that changes are decoupled and the evolution of the ecosystem is moderately predictable.

Over time, it's likely that you'll find some resources that would benefit from being grouped and configured as a unit. Because of the possible challenges associated with manipulating the TF state, it's important to make the decision to introduce a custom module judiciously and thoughtfully.

In terms of how to construct a nested module from existing resources, the first step is to determine what goes in the modules (you'll typically *not* be constructing a single module, which you'll explore next). Then it's simply a

matter of refactoring the existing module, such as by moving the `resource{}` blocks to another location and then referencing this location according to the location's type.

In the following sections, you'll explore when to nest a module and get a comprehensive approach to performing this refactoring activity.

When Not to Nest a Module

Just as important as when to nest a module is when *not* to nest a module. It's typically not wise to nest a module in the following cases:

Differences in resource update frequencies

> If some resources require updates more frequently than others, confining them to a nested module will provide no improvement in terms of separation of concerns in this context. However, as you'll see soon, constructing modules may be appropriate if there are stark differences in how regularly updates are made to the configuration for the resources (the actual TF code).

Few resources that are already related

> All the resources in your current module are related, and there are only a few of them. In this case, the module is most likely already well-sized. Unless there's serious impending plans for expansion, that will cause a significant imbalance in the nature of these relationships — which we will explore more in "How to Refactor for Nested Modules".

Mixing resources and modules

> You are encapsulating only some of the resources in a nested module and will end up with a mix of `resource{}` and `module{}` blocks.

Singleton modules

> Aside from contrived examples, there's typically little value to constructing a nested module that includes only a single `resource{}`

block. In many cases, the presence of a single `resource{}` block appearing among totally unrelated resources — in terms of physical, logical, functional, or in resource's lifecycle — in a TF module is itself an oddity. In most cases where a singleton module seems appropriate, there's an alternative construction that results in a less scattered and more flexible module, as you'll see in "Dealing with Singletons".

Additionally, if you have dependencies between collections of resources and will need to combine, marshal, and unmarshal great swathes of information, or you find yourself using the same module source repeatedly for different domains or application infrastructure stacks, as illustrated in Example 5-2, these collections probably should be created together in a bundled use case type of module, as described in the next section.

Example 5-2. Multiple references to the same module source and marshalling/unmarshalling module output

```
locals {
  server_subnets = {
    for id, snet in module.subnets.subnets_map: id => snet
    if contains(var.server_regions, snet.region)
  }
  firewall_data = {
    for id, snet in module.subnets.subnets_map: id => {
      region   = snet.region
      to_range = snet.range
      from_ips = (
        contains(var.server_regions, snet.region)
        ? concat(
            module.jumpboxes.ips[id],
            module.clusters.cluster_ips[id]
          )
        : module.clusters[id].cluster_ips
      )
    }
  }
}

module "subnets" {
  source = "./modules/subnets"

  regions = var.available_regions
}
```

```
module "server_iam_identities" {
  source = "./modules/iam_identities"

  identity_data = var.server_identity_data
}

module "cluster_iam_identities" {
  source = "./modules/iam_identities"

  identity_data = var.cluster_identity_data
}

module "jumpboxes" {
  source = "./modules/compute"

  subnets_data  = local.server_subnets
  identity_data = module.server_iam_identities.identities_map
}

module "clusters" {
  source = "./modules/clusters"

  subnets_data  = module.subnets.subnets_map
  identity_data = module.cluster_iam_identities.identities_map
  jumpbox_ips   = module.jumpboxes.jumpboxes_map
}

module "firewalls" {
  source = "./modules/firewalls"

  data = local.firewall_data
}
```

Bundled Solution Use Case

Perhaps the most obvious use case for creating a nested module is to bundle `resource{}` blocks that are closely related, especially those that may be considered a single unit. In this section, you'll review several canonical examples that may be considered for combining in a single module that may be nested within other modules.

Compute instance configurations

In many CSPs, compute instance resources are dependent on other, technically distinct, components such as disks, IAM identities, or network interfaces. These can be combined into a single module that constructs these components in a predictable and standardized way, while allowing for any desired or necessary variability such as the size or number of the attached disks. For example, your organization may have a particular licensed OS it wants to enforce on all of its cloud compute instances, support for optional attached disks, and the ability to connect to multiple networks beyond a specified default to which to attach interfaces, as depicted in Example 5-3.

Example 5-3. Combining compute-related resource definitions

```
variable "os_image" {
  description = "RHEL OS image for compute instance"
  type        = string

  validation {
    condition     = strcontains(var.os_image, "rhel")
    error_message = "os_image must be a rhel image"
  }
}

variable "subnetworks" {
  description = "Subnetwork names for compute instance"
  type        = list(string)

  validation {
    condition     = length(var.subnetworks) > 0
    error_message = "subnetworks must not be empty."
  }
}

data "google_compute_subnetwork" "supported" {
  for_each = toset(var.subnetworks)

  name = each.value
}

resource "google_compute_instance" "generic" {
  name         = var.name
  machine_type = var.machine_type
  zone         = var.zone
```

```
  boot_disk {
    initialize_params {
      image = var.os_image
    }
  }

  dynamic "network_interface" {
    for_each = data.google_compute_subnetwork.supported

    content {
      subnetwork = network_interface.value.self_link
    }
  }

  dynamic "attached_disk" {
    for_each = google_compute_disk.optional

    content {
      source      = attached_disk.value.self_link
      device_name = attached_disk.value.key
    }
  }

  service_account {
    email  = google_service_account.default.email
    scopes = ["cloud-platform"]
  }
}

resource "google_compute_disk" "optional" {
  for_each = { for disk in var.disk_data : disk.device_name => disk
}

  name                      = "${var.name}-${each.key}"
  type                      = "pd-ssd"
  zone                      = var.zone
  physical_block_size_bytes = each.value.size_bytes
}

resource "google_service_account" "default" {
  account_id   = "${google_compute_instance.generic.name}-account"
  display_name = "Service Account -
${google_compute_instance.generic.name}"
}

# Several variables and fields omitted for attempted brevity
```

This example could be used as a nested module as part of a larger module that establishes the components for an application stack. This would allow the compute configuration to remain distinct from other application stack components, such as managed databases. This may be desired to reinforce a logical separation between different stack domains with different module scopes. By using different input variable assignments (and proper TF state management), developers can construct replicas of the same stack, or even different stacks for distinct applications. For example, a similar configuration to the preceding example may be leveraged to construct production resources using a Red Hat Enterprise Linux (RHEL) boot disk and certain non-production (non-prod) resources using a Rocky Linux distribution boot disk.

Specialized appliances

While CSPs may provide extensive support across many different cloud considerations, even partnering with third parties or supporting prebuilt third-party solutions, none can provide support to everything you may conceivably want to implement. Certain highly proprietary tools or platforms may require additional installation or configuration effort to fully support all valid configurations. For example, if you're configuring a third-party Intrusion Prevention System (IPS), you may need to construct specialized compute resources that connect to networks in highly specific ways. Doing this alongside other `resource{}` blocks would likely be confusing and therefore potentially dangerous if certain esoteric components or operations are required for proper operation and run the risk of improper modification.

Instead, these appliances and other highly specialized configurations — especially those that are singular or exceptional in nature — may benefit from the logical isolation that surrounding the configuration in its own, nested module affords. In many cases, this may simply be an extension of the previous example, perhaps with more prescriptive resource field assignments. However, the more specialized the solution, the more complexity or variability that would need to be supported by a generic

module, which begins to drive down its value. Instead, it may be more effective to, for example, fork the generic module's Git repository and customize the fork to embed in the appropriate additional components.

> **TIP**
>
> When leveraging a fork, it may be possible to keep it updated with changes to the repository it was forked from. This may be especially effective to keep from needing to make updates in two places if additional features of the underlying resources become adopted later and need to be introduced to both modules.

Networking

Sometimes, it may be prudent to combine all the primary resources of a particular domain into a single TF state to ensure they share a common lifecycle. For example, a networking-oriented root module may combine different networking aspects representing different layers of your favorite networking module such as the Open Systems Interconnection (OSI) model. For example, the VPC, subnet, and routing components may be in one module, firewalls in another, and Domain Name System (DNS) zones and records in yet another module, as shown in Example 5-4.

Example 5-4. Splitting networking components into distinct nested modules

```
variable "network_data" {
  description = "Network data for VPC networks."

  type = object({
    domain = string
    ranges = list(string)
  })
}

module "vpc" {
  source = "./modules/vpc"

  network_data = var.network_data
}

module "firewalls" {
  source = "./modules/firewalls"
```

```
  network = module.vpc.network
  ranges  = module.vpc.ranges
}

module "dns" {
  source = "./modules/dns"

  network = module.vpc.network
  ranges  = module.vpc.ranges
}
```

This example takes a web domain as an input and constructs a set of dedicated network components to support software that uses the domain for DNS resolution of web services. This can be reused to construct multiple fully established cloud networks with all the necessary components organized together for each dedicated domain, servicing multiple products or lines of business.

Mismatching Frequencies of Code Changes

Another use case for refactoring into nested modules is when a module includes resources with very different code lifecycles. For example, if one set of resources is relatively stable while another is subject to more regular updates or experimentation with different features, it might be useful to lean into this behavior by segmenting into different modules. In this case, you should also utilize dedicated source control repositories to formalize this distinction.

This is really more of a software lifecycle design detail than a TF module development one. This likely would be utilized in only extreme cases, rather than as a preferred tactic. In most cases, the other considerations in this section (bundled solutions, resource imbalances, and so on) will produce stronger module boundaries. However, in the presence of many resources in one module with verifiably extreme difference in update frequency among `resource{}` blocks, and a lack of any other strong differentiator, this may be the best option available. Although this rather conditional consideration suggests a low degree of implementation, it may

be more common than it would seem. For one, it's one of the few that has a straightforward measurability: it's a matter of determining how regularly different areas of code are updated relative to one another and applying a statistical analysis that likely need not be highly sensitive to be valuable — because you'd be looking for stark differences in code lifecycle anyway, only simple grouping heuristics need be considered.

> **NOTE**
>
> Refactoring to partition the resources into nested modules, either local or remote, has no impact on the lifecycle of the resources. For guidance in separating the items in a TF state to establish distinct lifecycles for resources, see Chapter 6.

Unbalanced Resource Relationships

As a TF project grows in scope and more resources are included to fill in any gaps or to scale environments effectively, certain imbalances in relationships may begin to arise. For example, some resources may not depend on other resources, but may be a technical dependency for other resources. In other cases, the dependency is more logical in nature such components that will be integrated together. At first, these relationships may be somewhat uniform in nature, and further segmentation may seem more arbitrary than tactical. However, as time goes on, and more resources are included into the module, making segmentation through nested modules more attractive, it can be useful to consider how interconnected various groups of resources are. For example, a module containing several network security-related components, including the cloud network resources for several trusted networks, the same for a shared untrusted network, and the various resources used to establish an IPS may be split into three modules. Certainly, the latter IPS appliance configuration falls into the bundled solution use case, and while the former network groupings undoubtedly share many common traits, they may be functionally distinct enough that a common module may not be desirable. Even in the case that the same

module is used, it may be leveraged in multiple places in an outer module with distinct module inputs, as shown in Example 5-5.

Example 5-5. Using nested modules to enforce distinct relationship boundaries

```
module "server_iam_identities" {
  source = "./modules/iam_identities"

  identity_data = var.server_identity_data
}

module "cluster_iam_identities" {
  source = "./modules/iam_identities"

  identity_data = var.cluster_identity_data
}
```

As another example, a module that services an entire application stack, including caches, databases, document storage, subnetworks, custom firewalls, custom routes, and so on, may benefit from grouping the components that are logically related. This might mean splitting up storage, network, and resource identifier support into multiple modules, as shown in Example 5-6.

Example 5-6. Grouping storage, network, and service account definitions into a module

```
module "app_1_dev" {
  source = "./modules/stack"

  app_name   = var.app_1_name
  project_id = var.app_1_dev_project
}

module "app_1_prod" {
  source = "./modules/stack"

  app_name   = var.app_1_name
  project_id = var.app_1_prod_project
}

module "app_2_dev" {
  source = "./modules/stack"
```

```
  app_name   = var.app_2_name
  project_id = var.app_2_dev_project
}

module "app_2_prod" {
  source = "./modules/stack"

  app_name   = var.app_2_name
  project_id = var.app_2_prod_project
}

#-----------------------------------------------------------------
-----
# Application stack module
#-----------------------------------------------------------------
-----

resource "google_storage_bucket" "default" {
  name     = "${var.app_name}-bucket"
  location = "US"
}

resource "google_compute_network" "default" {
  name = "${var.app_name}-vpc"
}

resource "google_service_account" "default" {
  account_id = "${var.app_name}-account"
}
```

TIP

Although splitting the same `resource{}` block types across different modules as shown in the preceding example runs against the principle of DRY, there are legitimate instances where it could be considered. Examples may include certain network components or IAM controls.

How to Refactor for Nested Modules

Once you have determined a module needs to be refactored into nested modules, you may need to take a highly controlled — and possibly one shot, big bang–style — approach to conducting the refactor. For one, the longer it takes to implement the change, the more likely it impedes or

interferes with other initiatives in the same module or TF project. No one wants to deal with merge conflicts, especially those that span multiple filesystem directories or source control repositories. Additionally, you generally want to refactor an entire module into the nested components at once to avoid mixing `module{}` and `resource{}` blocks. Finally, changes to include the same resources as nested modules will modify the structure that defines the TF state, adding more layers of complexity.

As an example, suppose we have a module that includes network components, IAM controls, and compute resources. We want to decompose this into modules that represent the most logical groupings according to the various considerations you've seen (and whatever others you come up with); for example, we might first segment by the service area, and then further into purpose-oriented groups, remembering to consider repeated `resource{}` blocks that may be bundled and whether the `for_each` will appear inside or outside the module. We may also construct multiple `module{}` blocks with the same module source but with distinct inputs, if there are logic distinctions between similar resources that you wish to make explicit. Each new module may then be constructed from the same `resource{}` blocks as in the original module (up to consolidating repeated resources). See an example of this in Example 5-7.

Example 5-7. Before and after refactoring a module

```
#------------------------------------------------------------------
-----
# Before - Root Module
#------------------------------------------------------------------
-----

resource "google_compute_subnetwork" "default" {
  for_each = { for stack in var.stack_data : stack.app_name =>
stack }

  network = var.network_id
  name    = "${each.value.app_name}-snet"
  region  = each.value.region
}

resource "google_compute_instance" "default" {
  for_each = { for stack in var.stack_data : stack.app_name =>
```

```
stack }

  name         = each.value.app_name
  machine_type = var.machine_type
  zone         = each.value.zone

  boot_disk {
    initialize_params {
      image = var.os_image
    }
  }

  network_interface {
    subnetwork =
google_compute_subnetwork.default[each.key].self_link
  }

  service_account {
    email  = google_service_account.default[each.key].email
    scopes = ["cloud-platform"]
  }
}

resource "google_service_account" "default" {
  for_each = { for stack in var.stack_data : stack.app_name =>
stack }

  account_id = "${each.value.app_name}-account"
}

#-----------------------------------------------------------------
-----
# After - Root Module
#-----------------------------------------------------------------
-----

module "nested" {
  for_each = { for stack in var.stack_data : stack.app_name =>
stack }
  source   = "./modules/nested"

  network_id   = var.network_id
  app_name     = each.value.app_name
  region       = each.value.region
  zone         = each.value.zone
  machine_type = var.machine_type
  os_image     = var.os_image
}
```

```
#-----------------------------------------------------------------
-----
# After - Nested Module
#-----------------------------------------------------------------
-----

resource "google_compute_subnetwork" "default" {
  network = var.network_id
  name    = "${var.app_name}-snet"
  region  = var.region
}

resource "google_compute_instance" "default" {
  name         = var.app_name
  machine_type = var.machine_type
  zone         = var.zone

  boot_disk {
    initialize_params {
      image = var.os_image
    }
  }

  network_interface {
    subnetwork = google_compute_subnetwork.default.self_link
  }

  service_account {
    email  = google_service_account.default.email
    scopes = ["cloud-platform"]
  }
}

resource "google_service_account" "default" {
  account_id = "${var.app_name}-account"
}
```

In terms of approach, the general process is fairly straightforward. First, the `resource{}` blocks would be moved to the new location — for example, another filesystem subdirectory — and arranged in whatever way you choose. The `variable{}` block definitions that these `resource{}` blocks rely upon would then need to be represented in the new location as well, possibly as direct copies of the existing module's definitions. Then, a `module{}` block will be introduced, with the necessary variable wirings,

thus completing the process for a single new nested module. Repeat this process for all collections of resources to be combined into other nested modules.

> **TIP**
>
> Even if you intend to source the modules as remote modules, either from another repository or even from an artifact registry, it may be beneficial to first refactor into local nested modules. Once the module is validated — for example, by actually applying a `tf plan` or by at least reviewing the plan to identify any problems — the entire filesystem directory may be extracted and source controlled separately.

A common choice as part of a refactor to nested modules is what to do with repetitive `resource{}` blocks. In cases where the entire collection is fully replicated, it would be most appropriate to include the `for_each` meta-argument in the outer module, rather than baking it into the nested module. However, don't let a replicated nature of certain resources be the only attribute you consider when refactoring to nested modules, especially if there are other genuine relationships to more singular or constant-in-number resources, such as a VPC `resource{}` block grouped alongside a variable collection of subnetwork `resource{}` blocks.

> **NOTE**
>
> In addition to the changes to the code itself, such as refactoring to use a nested module and placing the `resource{}` blocks into another filesystem directory, it may be necessary to update the TF state to match the new state addresses of running resources, if TF cannot determine this readdressing on its own. For guidance in rearranging the items in a TF state corresponding to the refactor to nested modules, see Chapter 6.

Dealing with Singletons

When refactoring modules into component parts, there may be singleton resources — those that appear as singular `resource{}` blocks, with no resources upon which they depend, and with loose relations to any other

resources — that seem arbitrary to include into other groupings. One option is to put the single resource into its own nested module. Another option is to leave the `resource{}` block in the outer module, violating the recommendation to avoid mixing `resource{}` and `module{}` blocks. Neither option is especially great. A third option would be to embed this singular resource in whatever module it most closely relates to, however loosely. This third option may also be deemed too arbitrary or awkward to provide sufficient justification.

In these cases, where no option is especially better than any other, consider whatever pattern is most likely to be used again. In truly one-off scenarios, it may be most appropriate to put this in its own module to adhere to the common pattern of not mixing `resource{}` and `module{}` blocks. This keeps the pattern consistent and also leaves room for highly specific logic or later expansion of the module, to include additional `resource{}` blocks that are closely related to the current resource.

If this is something that is a fairly regular occurrence — especially with the same resource type — across different modules in different places across your codebase, it may be prudent to relax the policy against mixing `resource{}` and `module{}` blocks. This should be done with utmost care and consistency in documentation, justification, and governance. In terms of guardrails, you may introduce some conventions and validation that disallows certain practices without the presence of specific qualities, or a multistep (one-time) approval process for allowing any such block mixing.

Certainly, there's nothing inherently wrong with taking each deliberation on a case-by-case basis. In fact, the more thought you put into these types of decisions early on, the less likely you are to build significant and risky technical infrastructure debt. However, promoting a standard that is agreed upon by primary stakeholders can both improve clarity for developers (thereby reducing the chance a piece of configuration becomes at risk due to confusion) as well as accelerate the solution design for architects, as they can posture the exercise as a validation, or rejection, of the preferred approach.

Modules as Protected Interfaces

Modules are TF's mechanism for grouping resources and establishing relationships between the groups. While this is inherently useful because modules are defined by all the TF configuration found, for example, in a particular filesystem directory (or similar document collection), it's subject to whatever access controls are available. This means that, typically, read and write access to modules may be restricted individually, which may be highly desirable if you consider your TF configuration proprietary or sensitive (which, who doesn't?) and intend to disallow anyone outside your team or organization any access to your modules. If you're instead developing a module that you intend to share with the world as open source (or as disclosed source), you likely want to expose read access to the underlying configuration for consumers to peruse and may need to develop additional controls to restrict users from outdated or incomplete renderings of your module. The following sections showcase the challenges associated with various distribution types and considerations.

External Distribution

One of the major benefits to developing modules in TF is that they may be reused to accelerate subsequent infrastructure development work. For modules that are designed to be distributed externally, such as through public channels, it's important to also balance ease of use with the risk of overexposing unprepared configuration. For example, with a module made available on a public artifact registry, the registry itself may be expressive enough to expose the input variables and output values, including descriptions, alongside the module's *README.md* document, but this requires you to still make a high degree of effort to keep these documentation components up-to-date. However, this may not even be enough, as simply exposing input and outputs may fail to disclose all included resources or functionality, meaning that the underlying components are not apparent until, for example, a `tf plan` operation is executed and the planned components are written to the command-line output. This introduces a great deal of uncertainty, which further elevates

the risk inherent to IaC. As you'll explore more deeply in later chapters of this book, all cloud infrastructure is production — there's no such thing as a development environment for IaC because even the development environment for your applications must be hosted on resources that are well defined, properly and effectively secured, and cost-effective. Just like production (the Service Level Objectives [SLOs], Service Level Agreements [SLAs], and other nonfunctional requirements may change, but the level of care and scrutiny with which you construct development environments for your applications' software delivery lifecycles should be no less than that with which you apply to your production environment), any uncertainty should be mitigated and minimized as much as possible. Implementing validations — such as `validation{}` blocks in variable definitions or `precondition{}` and `postcondition{}` blocks in resource or data source definitions — is a great way to mitigate some of this uncertainty by providing additional feedback when something is misconfigured or inappropriate.

Beyond a primarily documentation perspective, publicly available software is generally heavily scrutinized by security teams before its use is officially permitted, which may place a decision to adopt a third-party TF module too far off in the future to be especially useful. Most likely, a team would instead opt to build their own module in the meantime, likely resulting in a much more tailored (albeit, perhaps overfit) solution.

It therefore becomes important to expose the underlying code for any module you produce for external distribution. This way, anyone looking to evaluate your module for adoption in their own TF projects may simply review the source code at whatever commit was used to generate the available module. It's also important to include a license and terms of use in the repository; typically, these are provided as Markdown documents.

> **NOTE**
>
> Exposing source control (specifically on GitHub) for the module is a requirement to publish your module on the public Terraform registry hosted by HashiCorp, as explained in the documentation.

Obviously, there's a great deal of risk associated with making your software repositories public. No one wants to make a mistake on a public repository. Beyond the material risks associated with broken code, there's also a perception element that can be hard to overcome: end users always expect perfectly working software, so if your software appears to be unstable or unreliable, it can create a statistically outsized amount of damage to your brand. This means that branches that are more experimental or incomplete pose a great deal of risk, possibly resulting in unfortunate outcomes such as a reluctance to commit and push changes often, leaving them as single points of failure on a developer's workstation. To better protect code that is considered a work in progress, implement whatever controls are available in your source code tool to allow, for example, only the branches and tagged commits that are truly ready for use.

> **TIP**
>
> Remember that exposing a branch will expose all the commits on the branch's history. Certainly, checking out a random commit is solidly in the "use at your own risk" realm, but it's likely worth including some type of clause explicitly cautioning against it in the terms of use you provide. A stronger approach may be to also require private forks for all new feature development (possibly even squashing the commits when the fork is merged back into the original repository to keep from any possibility of checking out a commit that had incomplete configuration), and perhaps to only perform bug fixes, dependency updates, and other quality or reliability measures directly in the public repository.

Internal Distribution

In many cases, it's not appropriate or wise to expose modules publicly. For one, any configurations that are especially specific to your running systems

may reveal too much information, however indirectly, about them. An overly contrived example might be providing support for an optional second network interface controlled by a Boolean input variable with a suggestive name, such as `include_management_nic`; this would suggest that your architecture includes isolated network channels for certain administration tasks. While this is unlikely to be damaging on its own, other similar clues might be gathered that could be used to shape a kind of profile of your systems' components. It's therefore fairly common to keep most, if not all, the TF modules you produce for your organization private.

This raises a new challenge: one of potentially wasteful redundancy. Instead of each team developing their own support for the same set of resources, curate a set of approved modules that may be used for standard resource stacks. These can be distributed in a number of different ways, including through source code repositories, managed storage buckets, and private artifact registries. Typically, it's most appropriate to support at least the code repositories and ideally one other option. While this undoubtedly creates redundancy, too, the differences in how the assets are generated, as well as distinct access management that may be applied, can make this redundancy a desirable feature.

Providing read access to the modules' source code repositories is a simple and powerful way to give transparency to the underlying resources and other components. By exposing how the module works, consumers of the module can determine if the configuration they will provide will be sufficient or appropriate for their needs. If you also expose all the feature branches, it can provide users with a glimpse of the future features that will be supported by the module.

In addition to a Git repository, expose the ready-to-use modules in at least one other way, such as a cloud storage bucket or an artifact registry that you control. Assuming you are not running `tf apply` manually, and instead are using an automation platform to execute the plan's steps, consider setting strict controls on how the modules are referenced in the `module{}` block's `source` attribute. A clean approach would be to grant users read access to the Git repositories, but disallow any access (read or write) for the

identity associated with the automation platform's actions. Instead, the automation platform is granted read access to only ready-to-use modules in your private artifact registry, for example. This way, the `module{}` block's source attribute would only ever reference the artifact registry copy, or copies, of the module. This setup even allows for developers to perform some local validation of the works in progress or otherwise unreleased modules by referencing the Git repository version of the module (to which they would have access). However, because the Git reference would be unsupported by the automation platform, developers would need guardrails, or at least sufficient discipline, to ensure such updates do not get included in any configurations that are to be applied.

> **TIP**
>
> There are certainly other valid approaches that use a combination of source control repositories and other module hosts. What's most important is consistency across the organization. Avoid situations where one set of modules is available from a private artifact registry and another from a managed storage bucket. This requires additional overhead to enable multiple solutions, which likely exceeds any serious justification for long running support.

Versioning

One important aspect of module distribution, which is often overlooked for internally distributed modules, is module versioning. Depending on the service that hosts the module, versioning will be implemented in different ways. While these differences might suggest a unique approach for each, it's recommended to design a strategy that includes or may be applied to each location as similarly as possible. This might mean a common naming system for versioning modules across each location.

The public HCP module registry — as well as the private registry available with HCP Terraform and any artifact registry that supports the Terraform Registry API — must use the Semantic Versioning system (SemVer) for any of its specified versions. This includes support for version suffixes using the extended SemVer format whereby a hyphen is appended following

alphanumeric characters. Although there are limitations to how these extended versions may be referenced, that will greatly influence their usage, (as you'll see later in this section) the extended format may be introduced to designate modules as prerelease using a hyphen followed by dot-separated alphanumeric characters and hyphens — such as `-beta` or `-rc.1` — or as part of a validation and release pipeline's build metadata using a plus sign followed by dot-separated alphanumerics and hyphens — such as `+2515807` to include a truncated commit hash or `+4820571396` to identify a GHA run ID. The SemVer website provides guidance for constructing valid SemVer versions. As described previously, the public registry requires source code to be publicly available in GitHub. The private registry from HCP Terraform supports additional version control systems.

For source code repositories, the most natural way to implement versions is through tagged commits (tags) and branches. *Tags* are essentially custom-named commits and can be named in whatever way deemed appropriate. For example, an appropriate tag might use the same SemVer versions as described previously, such as *v1.0.1*. Thus the same copy of the module's codebase is used to render the module wherever it's referenced. In addition to explicit versions, it may be desirable to establish a *release channel* approach using specific source control branch names. In the absence of any other strong rationale, the most logical branch to consider as the release channel is the main branch, but there may be a reason to support multiple versions, and additional branches may be introduced. These additional branches could even follow a similar naming approach to the tags: suppose two versions are supported — the current version, v2, and the previous version, v1. Current consumers of v1 might not be ready to upgrade to v2, but updates to v1 might need to be propagated to v2 for backwards compatibility. Regardless of whether they're consuming from v1 or v2, users would benefit from any updates pushed to the respective branch without having to modify their own `module{}` block's `source` field.

> **TIP**
>
> As recommended in the HCP documentation and specifically for Git repositories, you can set the special `depth` attribute to `1` to create a shallow clone of the repository that includes only the most recent commit in the history.

For storing module versions in other locations such as storage buckets in AWS or GCP, the versioning is much more up to you in terms of design. With storage buckets, a filesystem-like addressing approach may be used, and multiple versions of the same module can be supported using unique filenames and common prefixes. With this type of approach, you have different ways to choose how to establish versions. For example, one successful approach may be to store major versions with different prefixes (that is, the simulated filesystem location) and all minor versions within the same prefix, with all patch versions replacing its previous minor version.

TF also supports the use of HTTP/S URLs for fetching module sources, which provides even more variation to how versions may be constructed. As long as the URL includes certain response elements, it will be treated as valid by TF, and essentially any archive type, directory, or path may be retrieved as the module source. This means that any versioning schema that can be devised could theoretically be supported using the HTTP method for module hosting.

In some cases, it will be appropriate to construct a comprehensive versioning schema for one hosting solution and a subset (or generalization) of the schema for another solution. For example, in the case of using a source code repository and a storage bucket in tandem, it may be appropriate in the code repository to aggressively tag commits for very fine-grained versioning and to introduce long-running branches that correspond to the latest stable copy of a particular major version number, and then to upload only certain versions of the module to the bucket at a location that matches the corresponding major version. Whenever the automation platform references the module, it would use the storage bucket at the static major version location, but all previous minor and patch versions are

available for review. This approach can be extended to support referencing the minor version, in addition to (or instead of) a major. It may also be replicated using the other module source options, with certain analogous features to control the modules version — however, these features may be significantly different in how they're implemented.

Most likely, modules will not share a common lifecycle — that is, module versions will likely be produced at their own pace, not sharing a common release calendar so that all new module versions are released at the same time (not that this is a bad practice; it's just not a particularly common practice nor one that will work at all organizations). This can introduce wide differences in how regularly one module is updated in relation to another. For example, one module may typically update quarterly and only with major versions, whereas another may update every sprint with a minor version, and yet another may be updated sporadically and with no fixed schedule or roadmap. Each of these may require different combinations and proportions of various communication mechanisms such as email campaigns, standard documentation notes (in the root *README.md*, for example), and possibly more creative modes like blog posts, various website notification techniques, and chat messages on your favorite communication platform. You'll probably have to tune these methods over time for different release frequencies to find the right fit. For internal modules, these adjustments should be a combined effort between the team producing the module and the teams consuming the module, rather than a unilateral decision from any one party.

Referencing Modules

When referencing modules in the `source` field of a `module{}` block, the format of the reference and other available options depends on where the module is stored. Local modules may use a relative filepath as the value of its `source` field. For modules stored in module registries (those that adhere to the Terraform Registry API requirements), the `source` field includes the address of the module — the name of the location within the registry it appears. It does not include version information, which may be

specified by the special `version` field, as described later in this section. The source address for such modules also includes a base URL, which may be omitted for the public Terraform registry.

For Git-based module sources, you typically target the Git repository URL itself as the module source, which supports an optional `ref` parameter in the parameter map of the URL corresponding to a branch or tag. For HTTP/S URL sources, a version may be specified using the normal `GET` request's parameters. You would have the responsibility of constructing these API endpoints, which also gives you full control over what the parameters would expect. Each of the other module source types have a filesystem-like structure, allowing for a versioning system based on directory hierarchy.

In some cases, you want to access modules that are found within a module. For local references, buckets, and Mercurial repositories, this is meaningless, as they already have a directory-like structure and may just provide the full path to the module. However, for accessing modules found in a subdirectory of a module found in a module registry, Git repositories, or modules retrieved as a ZIP file, it's necessary to use the double slash, `//`, to delineate between the module's repository and the path to the desired module. Be aware that different types have slightly different syntaxes for where this double slash appears: for Git-based types, it would appear before the parameter map, whereas for ZIP files, it appears after the file extension. Examples of valid module references may be found in Example 5-8.

Example 5-8. Valid module references from various locations

```
module "local" {
  source = "./modules/local"
  ...
}

module "public_registry" {
  source = "hashicorp/vault/aws"
  ...
}

module "private_registry" {
  source = "app.terraform.io/tas/private_module/aws"
```

```
  ...
}

module "private_registry_version" {
  source  = "app.terraform.io/tas/private_module/aws"
  version = ">= v1.0.1-alpha, < v1.2.0"
  ...
}

module "git_repository" {
  source = "git::ssh://mydomain.com/git/private_module"
  ...
}

module "git_repository_version" {
  source = "git::ssh://mydomain.com/git/private_module?ref=v1.0.1-
rc.1"
  ...
}

module "git_repository_nested_version" {
  source =
"git::ssh://mydomain.com/git/private_module//modules/nested"
  ...
}

module "bucket_zip" {
  source = "s3::https://s3.amazonaws.com/tas/c5/module.zip"
  ...
}

module "bucket_zip_nested" {
  source =
"s3::https://s3.amazonaws.com/tas/c5/module.zip//modules/nested"
  ...
}
```

As shown in the previous example, for those modules sourced from a module registry, TF supports a special `version` field that allows developers to control which version to use. If no version is set, TF will fetch the latest version that was published to the module. The version field supports version constraints, which can define a range of acceptable versions and cause TF to retrieve the most recently published module included in the range. The recommendation around version constraints from the HCP documentation is worth reiterating, which is to "explicitly

[constrain] the acceptable version numbers to avoid unexpected or unwanted changes." You can implement the procedures described previously whereby a module is published as a specific version, with major, minor, and patch versions, but the version constraints tell TF to retrieve the most recent major (or minor) version. Another recommendation would be to introduce organization-wide standards around the use of version constraints. A constraint schema might be published in a central documentation wiki, with individual modules providing overriding guidance for its specific consumers in the root *README.md* file.

One challenge with TF module versions is that once a module is fetched using `tf init`, the module will be stored in, and sourced from, the local cache. As a consumer, when working with remote modules, especially any that are stored in buckets, fetched using HTTP, or referenced using version constraints, it's a best practice to include `tf init -upgrade` at various places in the development lifecycle. This will force TF to fetch the latest module version that was specified.

Deprecating and Removing Modules

In addition to versioning your modules, have a standard way to deprecate and remove module versions from support. This would generally follow typical software development practices in this space and is valuable for the same reason as for any software product: it keeps you from having to maintain support for too many versions of your module. And, just like with any product, it will be primarily influenced by how you conduct the versioning of your module. For example, if the module you produce is fairly simple or fit to a specific purpose, you may never need to produce another major update. Even if the underlying resources and providers go through large changes or support gets expanded in small ways, it may be possible and acceptable to issue only patch and minor version updates. In this case, there may be no formal deprecation process. If, however, the module is intended for high flexibility, perhaps introducing support for new features over time, it would likely be most appropriate to run multiple versions of the module to support a high degree of backwards compatibility

and allow consumers ample time to evaluate the new version and schedule any tasks they'll need to perform to adopt the updated module.

In the latter case, where modules undergo many large changes that would culminate in major version updates, it will likely be heavily dependent on the frequency with which these new major versions are released. A good formula for constructing a deprecation schedule might look something like the following: a module is flagged as deprecated for no less than 6 months, and there are never more than three available modules (with up to two supported). If 6 months is too short, 9 or even 12 months might be a better lower bound. This would be a consistent schedule for a module that sees major updates approximately every 6 months, which is, in practice, fairly rapid. A more frequent schedule, such as quarterly releases — possibly a mandated, but technically arbitrary, schedule that coincides with corporate marketing initiatives, for example — would require some modification of the numbers, but the heuristic framework would still be an appropriate way to approach such policies.

For modules intended for external distribution, the deprecation schedule should be seriously considered around the time the second major (or otherwise significant) version of the module is released. By then, the evolution of the module should be fairly foreseeable — enough to determine a broad course of action in terms of releases and version support, allowing you to establish proportionally broad timelines for deprecated components. It may not be necessary to publish a document outlining the deprecation schedule until a module version is actually marked for deprecation, but it should be clear from the various communication channels established for the versioning methodology when a module becomes deprecated and when it will become unavailable. Although it's also not strictly necessary to stagger deprecation with releases, you may choose to allow for some overlapping time period where both the new(est) module and old(est) module are both supported. This would give consumers a grace period for upgrading their references and any necessary modifications to the `module{}` block fields associated with the module's updated version.

The importance of deprecating and removing modules is just as true for internally distributed modules as it is for those externally distributed, and all the points mentioned previously apply. It should not be left to individual teams to determine for themselves (or in many cases, not determine at all). In early stages, when only one team is developing modules for use by only a few other teams, it may be acceptable to establish this process organically, with no set standard for how many versions are supported and how quickly deprecated ones become totally unavailable. However, once internal remote module consumption becomes saturated across your infrastructure, with several teams relying on your modules for their workloads, it becomes important that all your modules begin to share a common approach to deprecation with as similar a formula for scheduling it as possible. For example, consider two modules, module A and module B, with module A updated with a significant change every 6 months, and module B updated every 12 months — they both may share a maximum of two available modules — including at least one supported and at most one deprecated — and could even provide a 6-month grace period, as depicted in Figure 5-1.

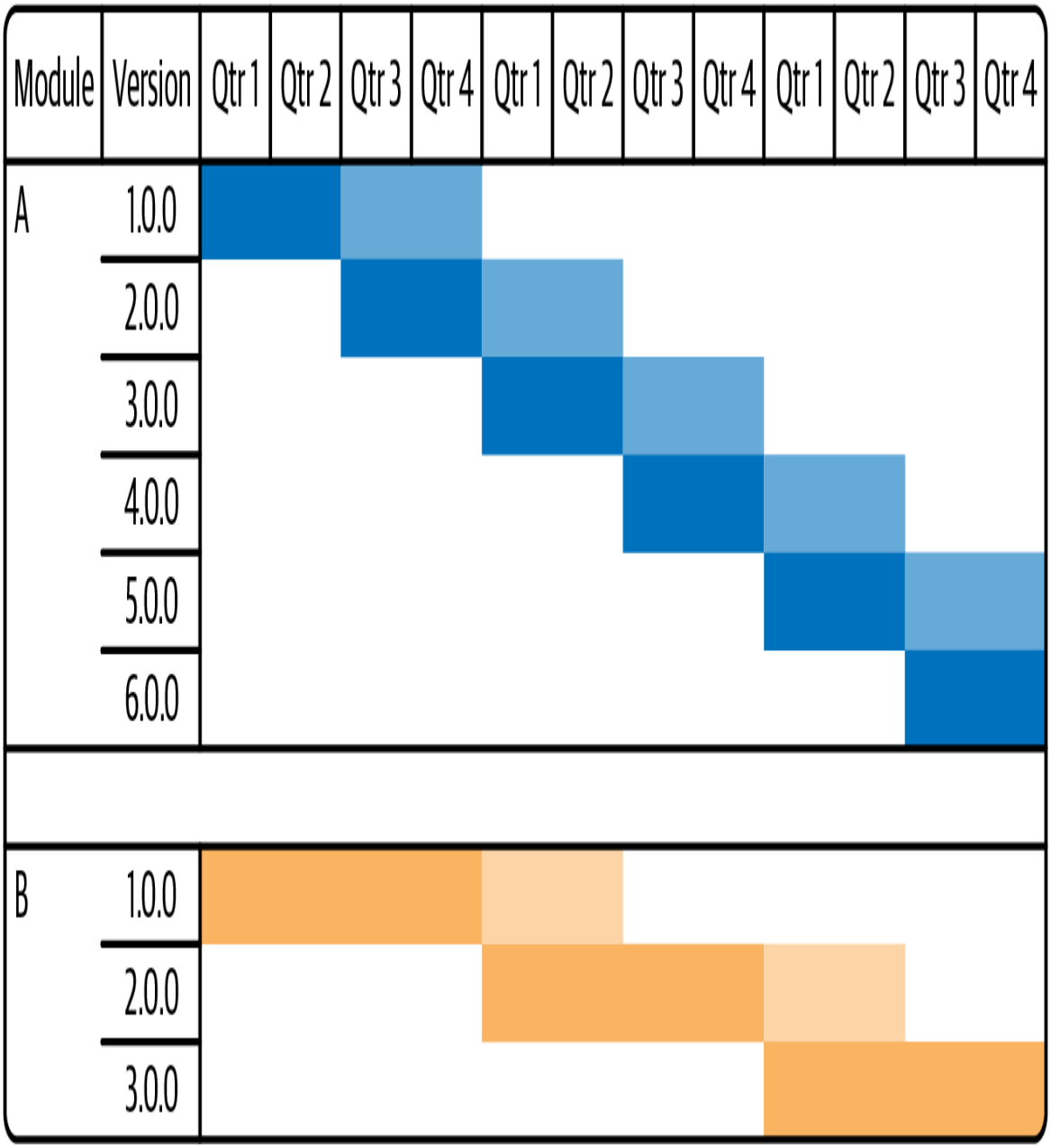


Figure 5-1. Common deprecation schedules for two modules with different versioning frequencies

Even in cases where versioning is not on a fixed schedule, or where new versions are uncommon and relatively unpredictable, it's good practice to introduce a deprecation plan that can as closely resemble your wider standard as possible. In practice, it's much more likely that this lack of foreseeability results in longer periods between releases, rather than shorter ones, which will typically fall within the bounds of a policy without modification. However, in the rare but not impossible case where

significant module updates can happen both unpredictably *and* often or in quick succession, it's probably appropriate to first extend the number of allowed modules that are available and marked deprecated before considering either an extension to the number of supported modules or a reduction to the period of time a deprecated module is available.

In addition to the deprecation schedule formula you establish, sufficient and appropriate communication channels and mechanisms will again be a top consideration for module deprecation. You'll want to make sure the consumers of your modules are well informed and get regular reminders both sent to consumers directly or left in highly visible places such as on your team's home page or at the top of your root *README.md* files. Similar to how communication practices for versioning are established, different deprecation schedules will likely lead to different combinations and proportions of communication channels and frequencies.

Local Modules

When it comes to where to record modules, you've seen that the primary consideration is whether the module is intended to be only part of a larger collection of modules or if it is expected to be more universally available. This decision is generally made at the time of developing the nested module: is the nested module's development part of a refactoring activity of a highly tailored use case, a greenfield configuration that has dedicated architectural oversight, or included as a shared capability that happens to be useful in a present use case? Each of these will influence the appropriate location for a particular module.

In the early stages of an organization's adoption of TF, and especially in cases where a single team is operating TF, there's a low separation of concerns in terms of responsibility. It's therefore common to place all `resource{}` blocks in the same root module. As root configurations become more and more complex and include too many, or too lengthy, *.tf* files, local modules may be an appropriate way to introduce encapsulation and a separation of concerns while preserving whatever customization is

necessary. The biggest benefit may be that introducing this segmentation of scope, through separate module boundaries, is that converting a module from local to remote has no impact on the TF state. As you'll explore in Chapter 6, making updates to the TF state (especially to preserve running services so that they remain unchanged) can be both risky and cumbersome, so any opportunity to reduce the chance of such activities is worth considering.

For modules that are intended to be used only locally, there may be few ways to actually enforce this intent for a particular bundle of modules. The problem is the fact that the only access rights necessary to successfully reference a remote module is read access, and if the root module of the bundle is exposed by its source control, its local modules are exposed as well. Arguably, this accessibility is highly appropriate to enhance transparency of the module. Rather than hiding the local module's implementation — such as by extracting it into a separate, private Git repository, registering it to an artifact registry, and referencing in the outer module from this new location — it's likely more appropriate to ensure these local modules adhere to the same level of quality as its associate root module.

TIP

If you're considering whether to introduce a module as local or remote, a good rule may be to introduce the module as a local module only if one or more of the following are true:

- There is currently only one team developing TF anyway.
- The module is fairly specialized, such as for a specific use case not expected to be leveraged elsewhere.
- The resources contained in the module are uncommon or experimental.
- The resources contained in the module are extraordinarily specialized or singular in nature.

Of course, this can become a slippery slope if the consideration begins to venture into the realm of overengineering, whereby too many problems that don't currently exist are included into architectural decision making. Alternatively, it may become an arms race, as local modules are effectively leaked out of its current module and become referenced in other nesting modules (thereby manifesting as a remote module). Because two distinct TF projects are now dependent on the same module (one local, one remote), there are naturally also now two agendas for this module. You may find that these agendas coincide enough to not present a serious issue; however, the risk of deviation grows inversely proportional to the level of coordination between the teams, and perhaps even exponentially so. As you'll see later in the section "Best Available Practices", the most consistent and conservative approach, especially beyond the early stages of adoption, may be to all but disallow local modules.

Module Composition versus Embedded Modules

When it comes to module nesting, the recommendation from HashiCorp is to follow a module composition approach — whereby each `module{}` block in the root module references a leaf module (one with no `module{}` blocks of its own), rather than the root module referencing one module that embeds another module within it — to take advantage of the concept of inversion of control. This is a reasonable ideal and should indeed be an aspiration and guiding principle. However, especially in the case of mixing third-party and internally developed remote modules, it may not be entirely appropriate to maintain over time, as resources expand and interdependencies shift.

A reasonable extension to the HashiCorp recommendation would be to restrict root module composition to leaf modules for internal or local modules and to consider third-party modules as if they were opaque systems and were terminating in their own way. In this way, modules may still pass information in a dependency injection fashion, and all the modules

you have direct influence over adhere to the "only one level of child modules" recommendation from the HashiCorp documentation. Without this relaxation, it may be too risky to include any third-party modules for fear that their internal implementation may change to leverage embedded modules — this is certainly possible for modules that fully adhere to HachiCorp's own Standard Module Structure, given the guidance around local modules. An example of this extended recommendation can be found in Example 5-9.

Example 5-9. Relaxing the recommendation for module composition for third-party modules

```
# assume this module only contains resource{} blocks
module "local" {
  source = "./modules/local"
  ...
}

# this module has module{} blocks embedded within it
module "remote" {
  source  = "terraform-google-modules/network/google"
  version = ">= 12.0.0, < 13.0.0"

  project_id = module.local.project_id
  ...
}
```

Suppose you have a stack of cloud resources that are replicated for different teams and use cases and that are logically separated into distinct groupings of disparate resources, such as firewall rules and compute instances, for example, designated as either trust or nontrust. Assuming these are all part of the same use case or application, it likely makes little sense to separate the lifecycles of the trust from the nontrust components. However, rather than building all the networks in one leaf module and all the compute components in another, which may obscure important differences inherent to the different functional domains — differences that may need to manifest in both the network and compute modules — not to mention the complex matching game that you must play to ensure the components created in the upstream module are available in the downstream one, it may be more appropriate to combine a leaf module for networks and a leaf module for

compute into a *super bundle* that will encode all these subtleties in a standard way. This super bundle may even be generic enough to support multiple functional domains, as shown in Example 5-10, but the differences between the domains are brought to the forefront by how the module encodes the dependencies between resources.

Example 5-10. A module that packages components that belong together within a logical domain

```
#------------------------------------------------------------------
-----
# Super-Bundle Module
#------------------------------------------------------------------
-----

module "app_trust" {
  source = "./modules/stack"

  app_name   = var.app_name
  project_id = var.app_trust_project
  os_image   = var.os_image
  trusted    = true
}

module "app_untrust" {
  source = "./modules/stack"

  app_name   = var.app_name
  project_id = var.app_untrust_project
  os_image   = var.os_image
}

#------------------------------------------------------------------
-----
# Stack Module
#------------------------------------------------------------------
-----

resource "google_compute_network" "default" {
  name = "${var.app_name}-vpc"
}

resource "google_compute_firewall" "default_deny_tcp" {
  count = var.trusted ? 0 : 1

  name      = "${var.app_name}-default-deny"
```

```
  network   = google_compute_network.default.self_link
  priority  = 65535
  direction = "EGRESS"

  destination_ranges = ["0.0.0.0/0"]

  deny {
    protocol  = "tcp"
  }
}

resource "google_compute_instance" "default" {
  name         = var.app_name
  machine_type = var.machine_type
  zone         = var.zone

  boot_disk {
    initialize_params {
      image = var.os_image
    }
  }

  network_interface {
    network = google_compute_network.default.self_link
  }

  service_account {
    email  = google_service_account.default.email
    scopes = ["cloud-platform"]
  }
}

resource "google_service_account" "default" {
  account_id = "${var.app_name}-account"
}
```

With this in mind, you may begin to see areas in which a totally flat composition of leaf modules may not be the most optimal construction in every case. Instead, in certain cases, it may be appropriate to construct module embeddings where a group of otherwise unrelated services, which may be logically grouped into leaf modules, are logically related and where this super grouping of resources is itself a repeated occurrence. This super grouping allows for more complex and complete infrastructure stacks, such as a super bundle that combines a data and storage module, a compute

module, a network module, and an IAM module all into one repeatable package. Many times, this super grouping simply suggests the use of a root module (which may be reused in various ways and which we'll explore in detail in Chapter 6 and throughout the rest of the book). However, if this super grouping fits within a larger scope that includes resources that are shared across these super groups and it is preferred to share a lifecycle among these infrastructure components, the use of an embedded module could be appropriate.

When all else is more or less equal, using embedded modules should be the exception, not the standard procedure. For one, creating module embeddings is not always the most efficient way to leverage reusable code. Two, if service domain-oriented leaf modules are regularly recombined to make super bundle modules, and leaf modules are never used directly in root modules, it may expose fragility in the leaf module's construction. In this case, it may be worth exploring ways to make the leaf module more robust and accessible by introducing more explicit relationships. Finally, overusing super bundle modules can introduce excessively complex input data schemas, as resource fields are nested within different scopes and collections. These considerations must be justified whenever you include an embedded module approach.

On the other hand, if you find yourself jumping through hoops in your root module to deconstruct, reconstruct, and selectively inject information from the result of one leaf module into the inputs for another leaf module — or worse, your leaf modules themselves are engaged in such data gymnastics and steeped in conditional expressions — it's high time to consider alternatives. When you do decide to include an embedded module approach, it's important to fully document the decision, including the business justification, to ensure it's clear that the approach should not be mimicked thoughtlessly. Include instructions for the appropriate use of the super bundler module so potential end users are aware that it's intended to be used as a unit.

Common, Useful Practices

Unlike most modern, established software projects, there are surprisingly few practices that can truly be considered *best* when it comes to TF. Part of this is simply due to the fact that every organization is different in terms of different people with different skills and different levels of commitment doing slightly different things in different proportions, and never in a static capacity. If you believe Melvin Conway that "organizations…produce [system] designs which are copies of [their own] communication structures" (Conway's Law), and assume that each organization has a unique communication structure, you may reasonably conclude that every organization will naturally design a unique TF corpus. Layer on the fact that TF is interpreted and that very different combinations of TF code can result in the exact same TF execution plan, and you begin to see why determining anything as definitive as a *best practice* becomes tough. Unlike a language like Python or Java, where two functionally equivalent procedures may have drastically different measurements for performance or efficiency, and thus one is undoubtedly superior to the other, it can be unclear or even arbitrary to determine which is the better of two functionally equivalent sets of TF code.

There is also the ever-present specter — if not season — of change that every organization experiences. A collection of TF may be by all accounts of the highest quality, perfectly segmented, and fully comprehensive today, but tomorrow, it becomes improperly fragmented and wildly incomplete, leading to a need for drastic demolition and remodeling. How is one expected to build for a particular organization when the organization is expected to change? The question is not if but when will this change occur and by how much? Adding a department may be much less disruptive to an organization's communication structure than a reorganization of teams. Likewise, adding a new product line, regional market, or even line of business may be far easier to fit into an existing TF corpus than realigning the responsibility of certain resources to different teams: while the former may simply be additional configuration or a replicated configuration, the latter may require a complete dismantling of an existing TF project to tease

apart the resources into new projects owned by their respective teams. Even the most well-designed TF projects will remain so for only a period of time.

Very few practices in TF will be successful and proper in 100% of situations. These have all but been exhausted in various documentation and blogs. A great deal more practices are worth considering in many cases. In this section, you'll be presented with several of the latter such practices: those that you should be aware of and can lead to success in certain scenarios.

HashiCorp's Standard Module Structure

HashiCorp's standard module structure is a great set of recommendations that may be used to organize module code. In addition to how to organize the TF code files, it suggests including a number of valuable ancillary items, such as a license and example code, and where all of these components should appear in a module's repository. However, as described on HashiCorp's documentation, the model is really only a file and directory layout, so recommendations about how to organize the actual code are totally absent from the model.

> **NOTE**
>
> The only guidelines actually related to code are to include a value for the `description` field for `variable{}` and `output{}` blocks, reference local modules by relative path from other modules within the repository, and not reference modules by relative path from the examples.
>
> All portions of the standard module structure are required by both the public Terraform module registry and the private registry provided by HCP Terraform.

As presented in the HashiCorp documentation, generally speaking, everything is optional except for the root module, which means that in some cases it may make sense to omit certain components (the previous note notwithstanding; certain systems, like a module registry, have more strict requirements). For example, a module that appears at the root of its

repository and that is also terminating would not have any need for a module/subdirectory. Similarly, for a module that is intended to be a true root module, as in the root of the TF state, there is little value to including any examples of it being referenced from a `module{}` block. All other components of the standard module structure are generally valuable to all modules.

Along with the possible omitted components, there are extensions to the standard module structure that you might consider in certain cases. One possible extension is to separate the modules/subdirectory, creating a new subdirectory named *locals/*, for example. This would make a clear distinction between modules that are usable as remote modules and those that are purely intended for internal use by other modules in the repository. In such a case, you might also choose to ignore the standard module structure guidance to omit a README in internal-only local modules, since the locals/subdirectory already conveys the intention.

Another extension that should be considered is a standard order of appearance of block types.

This should be agreed upon by relevant stakeholders within your organization. Many options can be considered, but a typical setup might be to store all blocks other than `variable{}` and `output{}` blocks (which already have their own respective files, per the standard module structure) in the module's *main.tf* file and order them starting with the `terraform{}` block, if present, followed by a single `locals{}` block, then followed by any `data{}` blocks; the final resource type would be either the `resource{}` blocks or the `module{}` blocks. An alternative order for modules that have many components may be to first group `data{}` and `resource{}`/`module{}` blocks by domain and within each domain separate the `data{}` blocks from the `resource{}`/`module{}` blocks. Any similar combination or ordering would likely be appropriate. The important part is to document and circulate the approach and ensure it's maintained.

Another alternative to the previous approach would be to separate each of the block types into their own *.tf* file. For example, `data{}` blocks would go in a *data.tf* file, the `locals{}` block in a *local.tf*, the `terraform{}` blocks (which would include the `backend{}` and `required_providers{}` subblocks) and `provider{}` blocks in a *terraform.tf* file. In this scenario, the `resource{}` blocks (or `module{}` blocks) would still appear in the *main.tf* file. Although this is a slight deviation from the standard module structure guidance, it doesn't necessarily violate the structure (as the module would still include the required files) and indeed follows HashiCorp's own recommendations around filenames. It also keeps the *.tf* files from getting too long in cases where local values and data sources are heavily utilized.

Best Available Practices

Continuing the review of useful practices, here are some of the most useful practices available at the time of writing, starting with several items that are widely available in such places as HashiCorp's documentation, blog sites, and other compilations. Although these are well-worn topics, they are repeated here for completeness.

Here are the best available TF practices:

Version control

> TF source files should be stored in a version control system like Git to record the development history and to enable multiple development tasks to happen concurrently on the same set of code. This is one of the most important components of any IaC platform, which is why it shows up so often in this book.

Workspaces and root modules

> Workspaces in HCP Terraform and TFE are used to maintain root modules and TF states. In an open source setting, a similar concept may be introduced. Workspaces should be environment-specific, and multiple workspaces may service the same root module.

Delegation and decentralization

As you've seen elsewhere in this chapter, as your TF ecosystem grows and becomes more and more complex, it's important to split components across multiple code repositories and TF states. Be sure to designate roles with clear responsibilities that include infrastructure operations, application stacks, and solutions architecture.

Use a module registry

As you saw earlier in the chapter, using a module registry is a great way to establish a separation between the concerns of writing the TF code and reading from a reusable module. Your code is stored on a version control system, but it is only read for the purposes of including its use in a TF state from a module registry.

IAM and role-based access control (RBAC)

As you've seen in multiple places, it's highly important to establish clear access control through the use of the various service provider's permissions management. It's also imperative to configure digital identities for automated processes. Team members should have read access to all code, write access to the code for which they're responsible, read access to the managed service instances for which they're responsible, and write access to no managed services. Digital actors have read access to the latest completed modules and write access to the service provider within the domain for which it services.

Parameterization

Just as with almost every software development, parameterizing field values is highly valuable. The primary mechanism for parameterizing values is through the use of input variable values. The other main component for parameterizing fields is local values.

Formatting and linting

Use the `fmt` command to ensure the basic formatting is applied. Consider the use of a linting tool like TFLint, ideally with the intent to develop custom controls. Include these in precommit hooks to ensure they are always applied before code is pushed to a shared location.

Fully consistent formatting is particularly rare across all projects within an organization's TF ecosystem. There are a few resources that you may consider compiling to get something of a standard, but there does not seem to be an industry-wide consensus across all syntactic features within TF. The following is the basic guidance that is available from HashiCorp and the community. Take what resonates and incorporate it into your organization's specific guidelines:

Entity names

The typical guidance is to surround the identifiers for TF entities with double quotes, use an underscore as a token delimiter, and not include the resource type.

Indentation

It's generally advised to use two spaces to indent each level of encapsulation scope.

Fields

Aside from the order that fields should appear, it's typical to group similar fields and place blank lines around these groups.

Multiline expressions

Surround multiline expressions with parentheses.

Multiline ternary expressions

Break very long ternary expressions into multiple lines, with the special characters appearing as the first element of the second and third lines, as shown in Example 5-11.

Example 5-11. Breaking a ternary expression into multiple lines

```
locals {
  all_items = (
    contains(var.list_to_check, item.attribute)
    ? concat(var.item, module.other.items)
    : module.other.items
  )
}
```

Things to Avoid

If best practices are few, the practices to avoid are all but ignored. However, they may be much more prevalent, and they can certainly crop up if you're not mindful of them. In many cases, they're relatively natural choices of an approach, but they can introduce unnecessary friction that saps productivity and flexibility.

One recommendation that is certainly less widely discussed but that has been referenced a number of times in this text is to not mix `resource{}` blocks and `module{}` blocks in the same module. This helps cut down on potentially confusing relationships between different TF state levels. A more extensible approach would be to fully commit to the use of module composition when introducing nested modules is deemed valuable and to use only `module{}` blocks. Refactoring to introduce only a single nested module, leaving the rest of the `resource{}` blocks in the root, for example, is likely not worth the effort. It may be better to extract the new module as its own root module, with its own lifecycle. If the shared lifecycle is explicitly desired or required, it may be appropriate to consider simply isolating the resources in question in its own *.tf* file. Likely, the best route is just to find a way to justify groupings of the other resources into additional nested modules.

If you're using OpenTofu and don't have easy access to a private TF module registry, it may be tempting to use a Git repository approach to sourcing remote modules that are internally developed. After all, your code is already (probably) in a Git repository, so why take the time, and possibly budget, to configure the module registry protocol (or other service that adheres to the

format of the Registry API)? One potential challenge with using Git repositories to source nested remote modules is authentication. You never want to tie your TF configuration to a single user, so the best available option may be with an SSH key. This becomes its own challenge: how are SSH keys generated, how are they secured, how often are they updated, and who is responsible for them? This makes using an instance of the module registry protocol, for example, much more attractive, leaving a private Git source as a workaround and tech debt that will hopefully be quickly paid off.

One thing that can be frustrating when first encountering it is the fact that certain special fields, like the `source` field of the `module{}` block, are static and do not accept variables or expressions as a value. At first, this is probably totally overlooked, especially early on when certain fields may be hardcoded rather than parameterized. Over time, it may become more obvious as your code trends toward full parameterization and fields like `source` begin to stick out as the only hardcoded strings remaining. Eventually, perhaps quarters or years later, you may decide that you want to upgrade a module that is treated as a remote module to incorporate new features or services, following an established, agreed-upon versioning mechanism as described earlier in the chapter. However, it may not be desirable to apply this upgraded version everywhere the module is used. In cases where each reference to the module is part of an otherwise distinct collection of TF code — whereby the outer module referencing this nested, remote module appears only within the hierarchy of a single TF state — there is likely no issue: simply update the source field in the place where you want to leverage or vet the upgraded version, and ensure all other references are explicitly set to the previous version. If the module is embedded within a root module that gets reused — such as by using a different set of input variable assignments, as discussed briefly in Chapter 4 and explored in further detail in Chapter 6 and later chapters — if only one of the TF states the root module services should receive the upgraded version of the module, you have a conundrum. How do you keep the versions of these root modules separated long enough to validate the new version of the embedded module, while allowing whatever necessary

updates (or drift corrections) to the root module for all the TF states it services? Suddenly, an exercise that should feel like progress may be stymied by the various workarounds you now need to employ to juggle the two versions. This is an area where not only are there no *best* practices, there aren't really any *good* practices. The best we can hope to do may be to avoid the *worst* practices. One highly risky practice that occasionally is considered in this case is to introduce variability to fields like the `module{}` block's `source` field by leveraging a command-line utility like `sed` or `awk` to perform string replacement, whereby a special string, such as `[VERSION_REFERENCE]` that appears within the string value of a `source` field, may be replaced by another string, such as `?ref=v2.0.1`, before a TF planning phase is executed, as shown in Example 5-12.

Example 5-12. Using `sed` *to replace a string in a* .tf *file before executing* `tf plan`

```
#!/bin/bash
...
sed -i 's@[VERSION_REFERENCE]@?ref=v2.0.1@g' main.tf
tf plan
...
```

As innocent as this string replacement may seem, it's very difficult to regularly validate, as there is no direct way to test it, and it relies on the discipline and awareness of developers to never use the special strings. It also requires a great deal of extra overhead in the form of documentation and input management. Once the level of extra work is realized, other slightly less unsavory measures begin to become more acceptable, such as deploying the TF state that includes the upgraded module from a different branch than you normally would. Unfortunately, this string replacement approach appears in many projects that support critical workloads, and not just to control the value of the `source` field, but to introduce variability in other places that don't inherently support input variables.

INTRODUCING STRING REPLACEMENT

It should be mentioned that it's rarely, if ever, the express intent at the start of a given TF project to incorporate string replacement as a deliberate measure. It's usually months and years into a project that these types of retrofitting of logic are considered. It's typically offered as a workaround but is also usually left as untouchable arcana, confusing and frightening future developers who stumble across it.

This is true for most of the things presented in this book as things to be avoided: they crop up over time, when the rocks meet the hard places and concessions are made over and over to continually move forward, however ever more sluggishly, rather than pausing to reassess and possibly redesign for robustness and further extensibility. If this sounds familiar, don't worry, you're not alone. It happens to many teams on a long enough timeline.

Additional Considerations

Once again, Conway's Law arises, as the full collection of TF code your organization develops will undoubtedly align to the way teams are structured within your organization, however imprecisely. This implies a collective stake held by the entire organization. Although the ability to execute or write a TF project should be the responsibility of a single team — or possibly split between two teams, separating the concerns of development and delivery, and either centrally controlled or distributed among several teams or departments — the development of TF code should not be totally hidden from the majority of your organization, especially other engineering teams. Perhaps the assumption is that it's irrelevant or uninteresting to other members of your company, but this should not keep you from being transparent. This is not to say show the sensitive values, but these should not be found in code repositories anyway. There is little benefit to keeping nonsensitive values hidden. This can result in unnecessary silos or boundaries and may sow discontent with other engineering teams,

creating unnecessary tension and friction. Instead, allow all members of your engineering teams read access to your TF code repositories by default.

Many large or widely dispersed organizations take a distributed approach to TF development. Typically, this means a somewhat layered or scoped approach with the most foundational, critical, or fragile elements owned and operated by core operations and security teams, which enables more product-focused teams to construct the more standard, common, and localized elements on top. Sometimes this leads to a centralized opaque system of infrastructure configuration managed by a single group of highly skilled and dedicated operations engineers who work behind the scenes and only appear on merge requests as the invisible hand of fate for work products. However, it's better to commit this team to more active oversight functions. You should not be throwing or receiving things over the proverbial fence when it comes to the quality of your architecture, governance included. Instead, find opportunities for collaboration and touch points between your core infrastructure and security operations teams and the other teams responsible for TF development.

One of the best ways to start introducing oversight, if only for its simplicity and budget friendliness, is to include regular, live code reviews as part of your development process. This is well-worn territory, in theory, but it's worth repeating given how uncommon it is in practice. Review your code. With your team. Every time. Don't conduct your code reviews asynchronously and separately, for example, only in a pull or merge request. Too often these get rubber-stamp reviews rather than serious consideration. It's always awkward at first, but just like any process, you get used to it, and the rough edges smooth out over time. After a while, you may find the reviews pay dividends as you stop having to re-solve the same problems over and over, and you'll wonder why you ever worked any other way. If you run an agile framework like Scrum, make it part of your team's rituals or a common component of the task's acceptance criteria. It could be an optional follow-up to your daily standup or a separate regularly scheduled meeting at whatever frequency you choose. If a regular cadence is too clunky, and developers are waiting hours or days to close tasks that await a

review, another option might be to introduce wide windows of time in which team members can expect to be pulled into an impromptu review call. If your team is scattered across different time zones, you might consider some kind of relaxation, such as allowing for only a quorum of the team to be present. If, instead, you have too laid-back a culture to impose overly strict schedules, you're already plagued by too many meetings, or there is some other reason preventing you from conducting live team code reviews, you should at least consider some way to get everyone's eyes on all code changes. Maybe gamify it with some (noncompensatory) metric or bounty that rewards high scrutiny of key defects like known inefficiencies, risky expressions, or improper syntax or style. Individual performance is not to be overlooked, but the best teams are those in which every member assumes responsibility for all of the work products created by the team. Aside from the more direct improvement to the quality of the code, it has the additional effect of spreading knowledge and skills among the team by exposing everyone to everything, which gives them some context where they otherwise may have none. This elevates the cross-functional nature many teams are intended to exhibit today. This is because, over time, team members are knowledgeable about more of the overall codebase.

In addition to conducting team-wide code reviews, it's valuable to have a standard or shared level of scrutiny across the entire organization. This seems to be already a fairly common practice, whereby a centralized operations-oriented team is responsible for providing review of code changes. Sometimes, this is restricted to only a portion of your modules — those that are most critical, foundational, or fragile — and this limitation may be somewhat immutable due to budgetary considerations. In this case, perhaps a quarterly or yearly audit across a sample of the modules that are not constantly reviewed would be sufficient to course correct any teams that have strayed from the standard level of quality.

Another easy component of effective oversight is up-to-date documentation. Really, creating the documentation is not usually the problem; doing so is often a corporate policy you have, whether written or unwritten. The problem is keeping it up-to-date, well organized, and, perhaps most

importantly, removing old documentation. Documentation is useful only if it's accurate. Although no documentation is obviously not helpful, wrong or outdated documentation can be worse if it sends someone down a rabbit hole that leads nowhere. Especially now that tools that implement retrieval-augmented generation (RAG) are becoming increasingly prevalent, it may be less important to keep documentation perfectly organized; but keeping it current and striving for completion are now both primary productivity drivers rather than nice-to-haves, afterthoughts, or begrudging tasks necessary to maintain compliance with audited standards. Often, the best documentation is the code itself, which highlights the point made earlier that everyone should have read access to all the TF code. It also compliments the concept of clean code that is referenced in a number of places in this book.

Finally, if you find yourself creating several disparate unitary modules with relatively weak justification, it may be worth taking a broad look across your modules and considering a more standardized approach. Introducing wrapper modules as a primary component is a totally valid approach so long as it's part of a standard, rather than a one-off case or an occasional occurrence. Singleton modules should generally be very uncommon and might need to be entirely exceptional. For example, exceptions may be made for highly specialized components that include multiple data sources and complicated expressions, or for components that benefit from sharing a lifecycle with another group of components to which it otherwise does not relate. Keeping singleton modules the rarely granted exception and using wrapper modules in a well justified and standardized manner will reduce the chance that these practices veer into problematic territory.

Summary

In this chapter, you reviewed several important concepts regarding the use and development of TF modules, including the following:

- Different types of modules, their special properties, and how they may be used together

- Where modules that are embedded in other modules may be sourced from
- How to construct modules intended to be embedded from collections of resources
- Various reasons to construct a nested module
- Mechanisms for effectively distributing TF modules
- Establishing versioning, including how to reference versions from various sources
- How best to construct modules that avoid resources too deeply embedded in a state
- How to extend HashiCorp's standard module structure for your company and use case
- The best available practices for developing TF modules
- Common practices that should be avoided or heavily controlled

You now have seen many successful practices and patterns and also know that none of them are a silver bullet. You recognize that as your TF ecosystem grows and your organization evolves, the collection of projects and modules you develop will also go through potentially significant overhauls to stay balanced and realign to the new communication structures, as Melvin Conway would put it. You understand that the journey never ends, and that it requires constant vigilance and evaluation of both your current infrastructure landscape and whatever is coming on the horizon.

In Chapter 6, you'll review the various ways in which TF state may be managed and manipulated to successfully run your TF projects. The TF state is established by a root module with specific input variable values and represents the state of the resources a TF project supports. You'll learn how to optimize the state for your needs, including maintainability, portability, and robustness.

[1] This was suggested to me after a talk I gave on logic in OpenTofu at OpenTofu Day Europe, 2026 — shout out to Artur for the recommendation.

Chapter 6. Optimizing State

Early in the effort to encode your existing infrastructure, the natural inclination is just to get as many things encoded and automated, orienting the codebase around the architecture, possibly into existing resource or responsibility domains, without thinking specifically about how TF works. Most likely, a sufficiently thought-out separation of concerns, even without strong TF exposure, will be serviceable for large swathes of effort, establishing the vast majority of your infrastructure and running without problems for months and years.

As your TF ecosystem grows and evolves, you may begin to experience challenges with some of the behind-the-scenes operations TF performs to keep its record of your infrastructure consistent. In particular, you may begin to notice unfortunate couplings across your codebase, recurrent incidents of inconsistencies between TF state and the running instances, TF operations that seem to take longer than they should, or all of the above. This chapter reviews various concepts, challenges, and approaches regarding the TF state and discusses how to converge toward their optimal organization.

State Concepts and Definitions

The TF state represents the "memory" for TF, preserving what it knows in terms of where and how all managed resources are configured. There are several components and concepts that make up and support the TF state. Understanding these makes it easier to discuss practices for successful operation of TF. This section presents and describes these components and concepts.

State Backend

Each state has a location where it's stored called the *state backend*. This can be a local file location, a cloud storage bucket, or various other managed storage services. What's important to note is that this is the source of truth for a TF state, and it's most typically registered as of the last `tf apply` operation.

The state backend is configured by a `backend` field as part of the `provider` configuration block, which means that without additional logic, a repository is one-to-one with a state backend. In "Building for Enhanced Maintainability and Balance", you'll explore ways to make a single codebase support multiple states by varying the `backend` field.

State Object

The *state object*, or *state tree*, is a tree-like relationship construct stored in a JSON structure that represents the state of the instantiated services at the time of a `tf apply` operation. The JSON file itself is stored in the state backend location.

> **NOTE**
>
> The state tree can be considered a directed acyclic graph (DAG), but for the purposes of considering how the states are traversed (and how they are stored in memory), it's essentially treated as a tree, but with specific rules about how it may be ordered.

Again, this state object can be thought of as the memory for TF. It records what resources TF observed and configured (including data references and new or removed resources) when it performed the most recent `tf apply` operation. It reflects a point-in-time snapshot of the operational state of the resources that are mentioned in the codebase upon which the operation was applied (Figure 6-1).

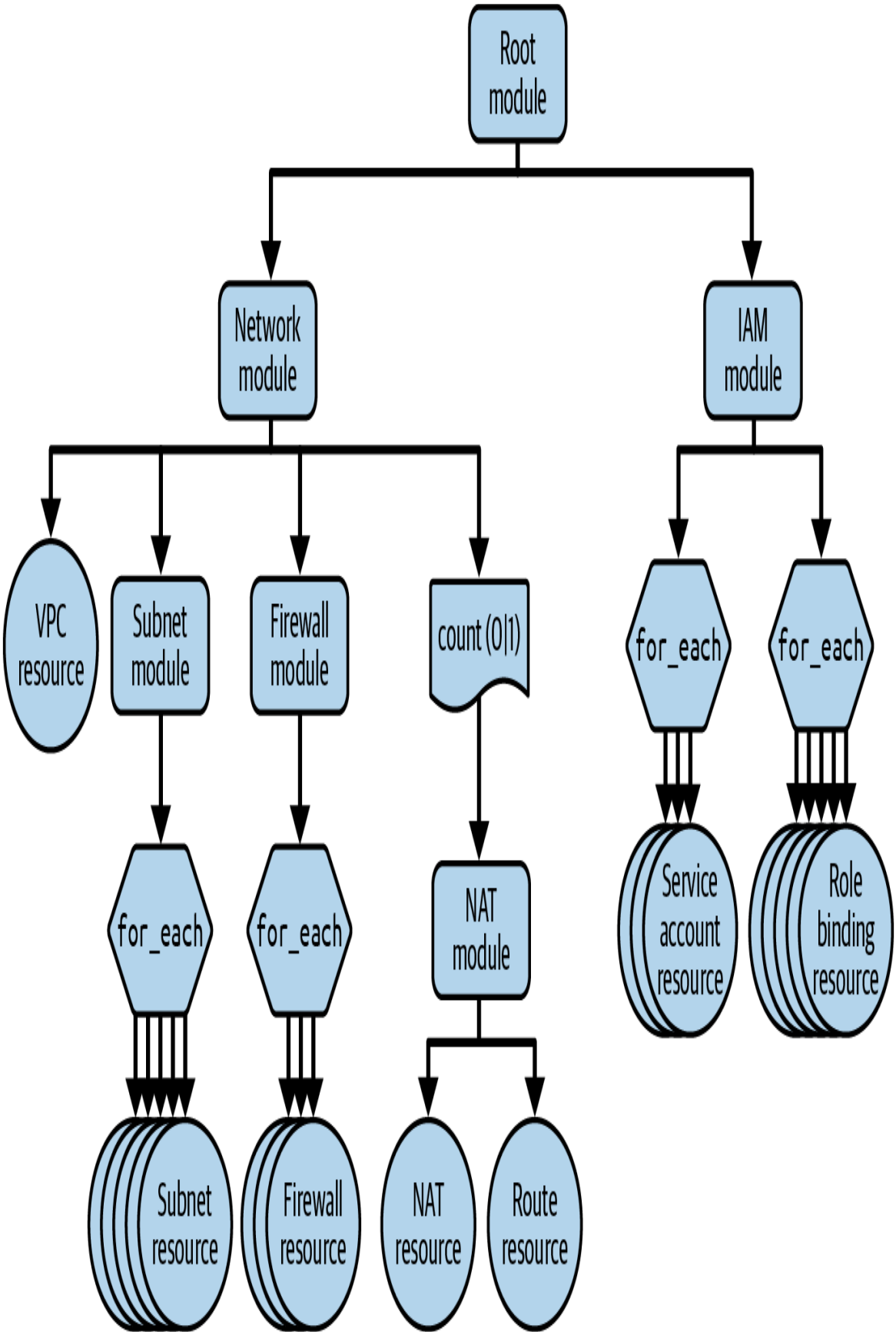

Root module
Network module
IAM module
VPC resource
Subnet module
Firewall module
count (0|1)
for_each
for_each
for_each
for_each
NAT module
Service account resource
Role binding resource
Subnet resource
Firewall resource
NAT resource
Route resource

Figure 6-1. Example state object, depicted as a tree

Because it records only what was configured through a `tf apply` operation, the TF state object will have no awareness of any other resources you may have (even if they too are configured with TF but are codified in a separate codebase). Moreover, because it records this only at time of execution, it does not alone constitute a source of truth for *any* of your cloud resources (even those in the codebase under operation), at least not for very long; only immediately upon a successful `tf apply` can you be sure a state object reflects reality in near absolute certainty.

State Triplet

As alluded to, the state object creates a three-way relationship along with the codebase upon which a `tf apply` is executed and the actual running resource instances. Throughout this section and the remainder of the book, I refer to this relationship or the combination as the *state triplet*. Typically, the relationship goes as follows: the code itself is read and applied using a `tf apply`; the resources get instantiated and configured according to the code; finally, the success of this operation causes TF to create the state object, which gets stored in the state backend, as depicted in Figure 6-2.

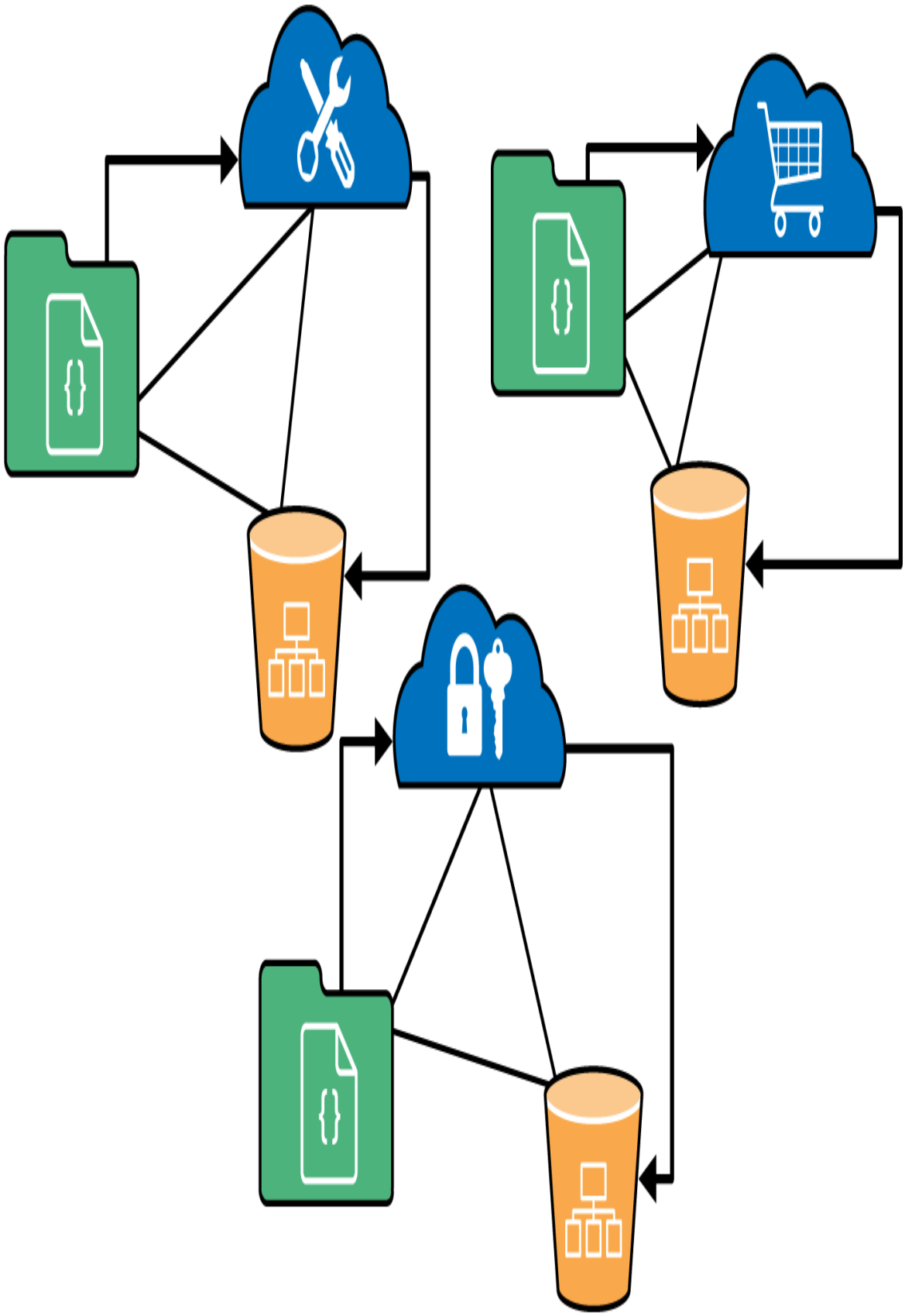


Figure 6-2. Multiple state triplets for separate TF states, for example, platform services, commercial workloads, and security controls (IAM)

> **NOTE**
>
> The nature of the relationship between the backend and the TF state means that changing the backend (and successfully running an `apply`) effectively moves the source of truth for the TF state.

A Quick Vocabulary Lesson in Graph Theory

I've already asserted, without many details, that state objects are also valid trees in the mathematical sense, which means they are also graphs (or networks). Without going too deep, it's illustrative to discuss the TF state object in the terms of graph or network theory. In its simplest conceptualization, every graph or network is a collection of two things: some set of discrete entities and peer-to-peer relationships thereof. In the parlance of graph theory, such a discrete entity is referred to as a *vertex* or *node,* and a relationship is called an *edge* (or sometimes *link*). In evaluating any two random nodes in a graph, it's most appropriate to consider their relationship in terms of degrees of separation or the shortest sequence of nodes and edges that separate them (referred to as a *path*). It's very possible to have two distinct paths between distinct nodes; when this happens, the combination of paths is called a *cycle*. However, *trees*, as the mathematical construct, are special types of graphs: those with no cycles. This means that between any two nodes there is exactly one path.

Alongside this defining aspect of trees, it's common to impose additional considerations and relationship constructs. The most important of these is the concept of a *root node* around which the rest of the tree is oriented. This is especially useful in depicting hierarchies or ancestries. The root node is considered the *top* (or *bottom*) of the tree, and each of the nodes it is connected to over an edge is described as its *child* node (thus the root node would be each of these child nodes' *parent* node). Nodes that have a parent node but no child nodes are referred to as *leaf* nodes. Finally, nodes that do

have children are considered *internal* nodes (note that the root node is a special type of internal node: one without a parent).

In terms of TF, data sources and resources are mapped to the leaf nodes and the modules to the internal nodes (including the root node — more information in the upcoming section "Root Module"). Figure 6-3 shows the previous example labeled and annotated with the components of a tree.

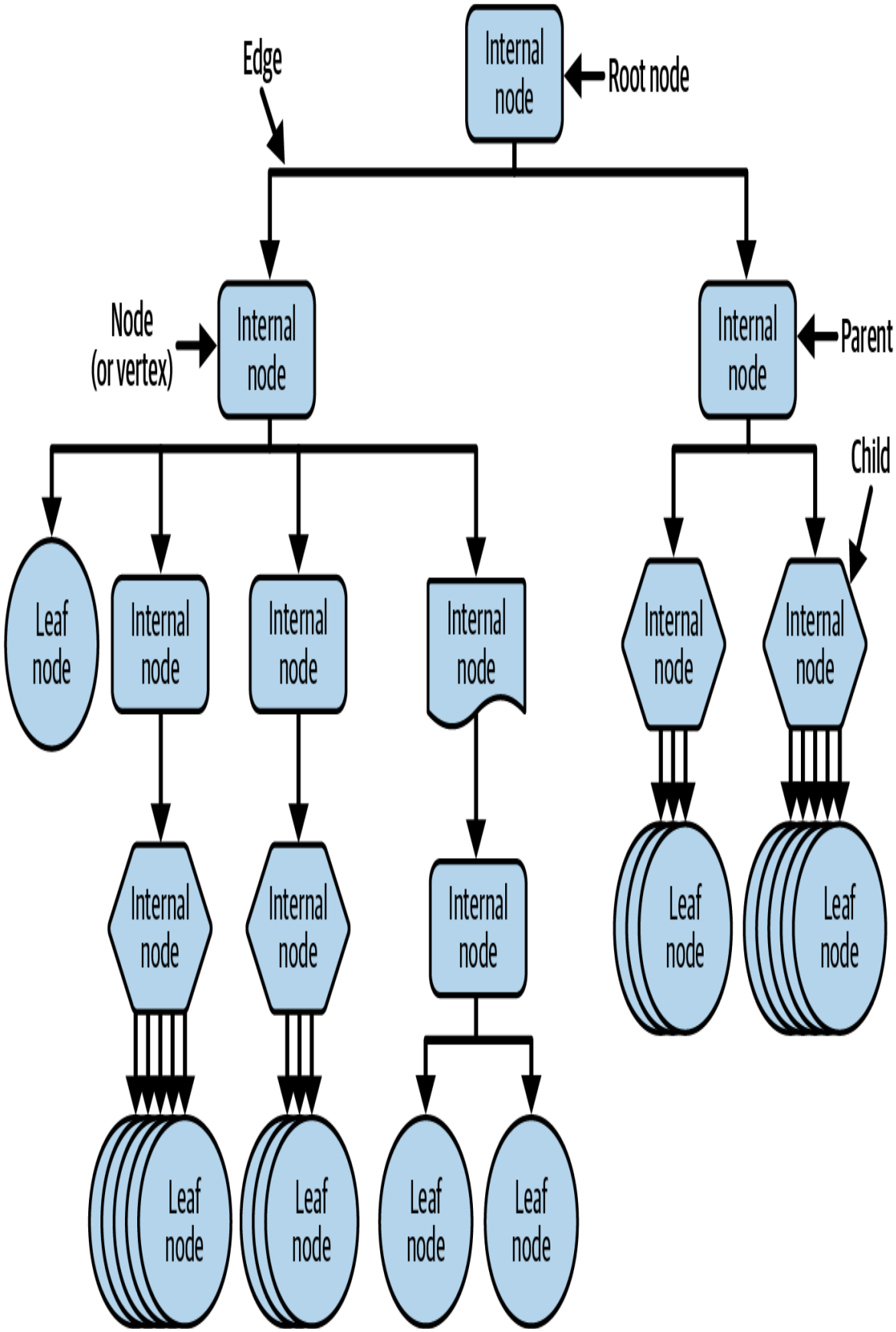

Edge
Internal node
Root node
Node (or vertex)
Internal node
Internal node
Parent
Child
Leaf node
Internal node
Internal node
Internal node
Internal node
Internal node
Internal node
Internal node
Internal node
Leaf node
Leaf node
Leaf node
Leaf node
Leaf node
Leaf node

Figure 6-3. Representation of a tree with labels and annotations denoting its parts

> **NOTE**
>
> It's also worth noting that any valid JSON object can be considered a tree (cycles are disallowed); YAML can also be depicted in this way. Conversely, every tree can be encoded as a JSON or YAML.

State Address

The location of a particular node in a mathematical tree construct is most commonly addressed as a progressive path from the root node. As mentioned earlier, any tree can be encoded as a JSON object, and indeed this is the approach TF takes in determining how to identify a particular module or resource for its own records (that is, the TF state object). Starting with the named identifier of the root node, every resource can be found by traversing the ancestry path. Because the TF state object is encoded as a JSON object, you can use dot notation to travel from one node to a child node successively until you reach the node in question, as depicted in Figure 6-4.

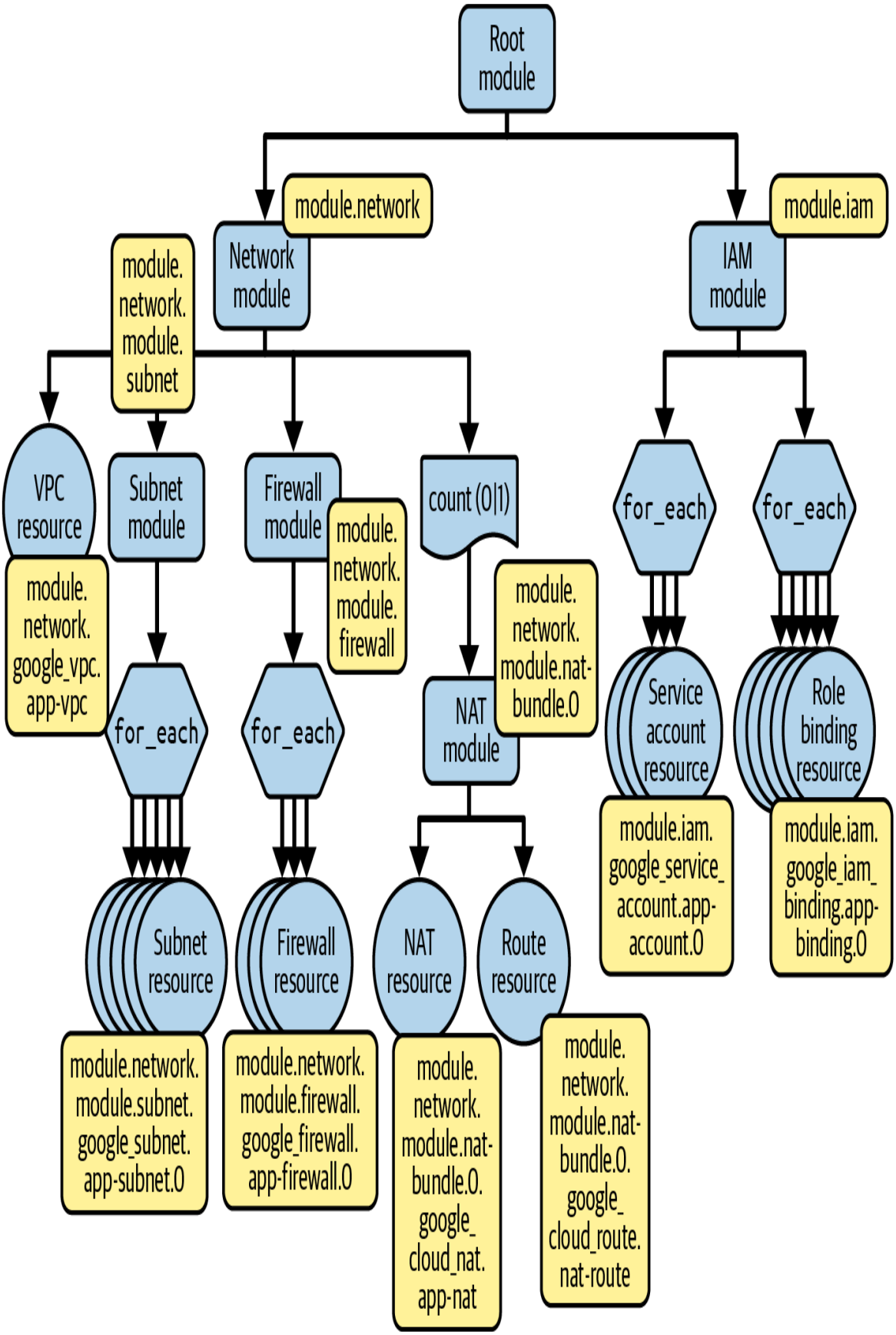

Root module
module.network
Network module
module.iam
IAM module
module.network.module.subnet
VPC resource
Subnet module
Firewall module
count (0|1)
for_each
for_each
module.network.module.firewall
module.network.module.nat-bundle.0
module.network.google_vpc.app-vpc
for_each
for_each
NAT module
Service account resource
Role binding resource
module.iam.google_service_account.app-account.0
module.iam.google_iam_binding.app-binding.0
Subnet resource
Firewall resource
NAT resource
Route resource
module.network.module.subnet.google_subnet.app-subnet.0
module.network.module.firewall.google_firewall.app-firewall.0
module.network.module.nat-bundle.0.google_cloud_nat.app-nat
module.network.module.nat-bundle.0.google_cloud_route.nat-route

Figure 6-4. State addresses for resources and modules

State Replication

You can use the same codebase with a different `backend` field (and ostensibly different inputs) to introduce additional reusability and domain separation/isolation within a particular service area. This can be especially effective in the case of reselling standardized cloud infrastructure or developing department/customer-specific infrastructure.

Importing State

The TF CLI provides some functionality to manually manipulate the TF state object, including ways to import existing resource instances. This can be effective in the case where a resource is configured manually but a decision has been made to preserve it in the organization's IaC. In this case, perhaps it was determined that a highly specialized resource configuration should be prototyped manually, but not too much later the "prototype" becomes a "pilot solution" that begins to provide measurable value; despite the intended temporary nature of the solution, suddenly no one wants to clobber it just to "do it the right way."

Enter `tf import`. You can encode the resource into your codebase and import the resource (without modifying it) into your TF state object using the addressing scheme as described earlier.

It's highly recommended to evaluate the configuration that gets incorporated into the codebase by running a `tf plan` to validate this state address, as shown in Figure 6-5.

```
Terraform will perform the following actions:

  # google_compute_subnetwork.tas_6_5_snets["snet-03-terraform-at-scale-6-5"] will be created
  + resource "google_compute_subnetwork" "tas_6_5_snets" {
      + creation_timestamp         = (known after apply)
      + enable_flow_logs           = (known after apply)
      + external_ipv6_prefix       = (known after apply)
      + fingerprint                = (known after apply)
      + gateway_address            = (known after apply)
      + id                         = (known after apply)
      + internal_ipv6_prefix       = (known after apply)
      + ip_cidr_range              = "10.0.2.0/24"
      + ipv6_cidr_range            = (known after apply)
      + ipv6_gce_endpoint          = (known after apply)
      + name                       = "snet-03-terraform-at-scale-6-5"
      + network                    = "projects/terraform-at-scale-book/global/networks/vpc-terraform-at-scale-6-5"
      + private_ip_google_access   = (known after apply)
      + private_ipv6_google_access = (known after apply)
      + project                    = "terraform-at-scale-book"
      + purpose                    = (known after apply)
      + region                     = "us-east1"
      + self_link                  = (known after apply)
      + stack_type                 = (known after apply)
      + state                      = (known after apply)
      + subnetwork_id              = (known after apply)

      + secondary_ip_range (known after apply)
    }

Plan: 1 to add, 0 to change, 0 to destroy.
```

```
robertglenn@Roberts-MacBook-Air import % tf import 'google_compute_subnetwork.tas_6_5_snets["snet-03-terraform-at-scale-6-5"]' terraform
-at-scale-book/us-east1/snet-03-terraform-at-scale-6-5
google_compute_subnetwork.tas_6_5_snets["snet-03-terraform-at-scale-6-5"]: Importing from ID "terraform-at-scale-book/us-east1/snet-03-t
erraform-at-scale-6-5"...
google_compute_subnetwork.tas_6_5_snets["snet-03-terraform-at-scale-6-5"]: Import prepared!
  Prepared google_compute_subnetwork for import
google_compute_subnetwork.tas_6_5_snets["snet-03-terraform-at-scale-6-5"]: Refreshing state... [id=projects/terraform-at-scale-book/regi
ons/us-east1/subnetworks/snet-03-terraform-at-scale-6-5]

Import successful!

The resources that were imported are shown above. These resources are now in
your Terraform state and will henceforth be managed by Terraform.
```

Figure 6-5. An existing resource that TF would attempt to create and the operation used to import the resource into the TF state

The `tf import` command can also be used in combination with other TF CLI state operations to reorganize or refactor existing IaC, moving it from one codebase (or location in the state object) to another, as we'll explore later.

However, importing the resources into the TF state only establishes the second leg of our state triplet. Any subsequent `tf apply` will read from the encoded specification, so the resource must first be reflected (at the same level in the hierarchy and with all the nondefault fields that were configured in the new instance) in the codebase, or the operation will remove it.

> **TIP**
>
> Operations like `tf import` are extremely sensitive. It's highly recommended (if not imperative) that you as an operator ensure that the result of a `tf plan` matches your expectation (that is, it's reporting only necessary and expected changes). With a `tf import` and matching resource definition, there should be no change reported from a `tf plan`.

State Trimming and Propagation

In addition to importing state configuration, TF also provides functionality to remove portions of a state object using the `tf state rm` operation. Typically, you should reserve this for extreme cases and instead drive all resource removals from the codebase itself, as this is often the easiest way to align the elements of the state triplet.

However, there's one scenario in which using `tf state rm` can be the best way to preserve running instances: refactoring a codebase so that it breaks the TF state down into multiple states. Doing so may be useful, especially in cases where resources in a state object (or their respective lifecycle needs) are sufficiently unrelated.

It's important to note that the `tf state rm` operation does not remove the running resources; it simply removes a specified resource from the TF state object. The resources will be unaffected. Because of this, a subsequent `tf apply` on an unaltered codebase will expect to recreate the instance and reinsert the state into the state object. If the instance created by the resource is still present, this operation may fail altogether if it would result in a duplicate resource that is impossibly similar to the old one.

Because of this, complete all the operations across the state triplet: remove the portion of the TF state object, remove the code (or move it to another repo or part of the codebase), and destroy the running instances (if necessary) all in the same sitting.

Drift

TF is generally *idempotent*, meaning the same code should produce the same outcome, no matter how many times it's run. Also, TF runs when you tell it to. Even if it's running on a cron, it'll only check on the schedule and be accurate to that point in time. It won't automatically detect changes you make in other contexts such as the platform's CLI, UI, or API, or applied to a different TF state.

Thus, unless all updates to the platform are made through TF (or unless the TF/provider version is not kept well enough up-to-date), there's a possibility of the dreaded *configuration drift*, whereby the running services have properties that are at odds with what is recorded in the TF state. When the drift is detected, TF will tell you how it will attempt to correct the drift, which often just means undoing the detected change. Sometimes, however, the configuration drift forces a resource to be recreated, which in turn destroys the original (drifted) resource.

Atomic States

Unless additional overhead is deliberately constructed (a priori, as the academics might say), each TF state is completely ignorant/agnostic of every other state. Thus, each `tf apply` results in exactly one state object.

If you consider a `tf apply` execution to be associated with exactly one pipeline/job (or map to the parlance of your choosing), then you can consider every pipeline to be completely ignorant/agnostic (again, a priori) of every other pipeline.

Moreover, and perhaps more importantly, under normal conditions one TF state won't detect changes you make in other contexts that don't directly affect the encoded resources. Even changes to similar resources (for example, a manually created VM that appears in the same project/account as the encoded VMs) via the platform's CLI, browser console, API, or another TF state will remain undetected.

Thus, each state tree can be thought of as atomic. Any connections between TF states would need to be implemented outside the confines of the state triplet: the TF state object stored in the designated backend, the TF codebase, and the configured cloud resources.

> **NOTE**
>
> It's certainly possible to encode a resource in two locations and import the instantiated resources into two distinct TF states; however, I would challenge the rationale for doing so beyond a temporary (and cumbersome) unavoidability (that is, a necessary evil — or more often a prescribed-by-superiors evil) or a contrived oddity.

Root Module

Because the TF state object is represented as a tree, it has a tree shape with the applied module in the surrounding code repository as its root node (as described in "State Object"). I call this module (the one you run the `tf apply` against) the *root module*.

State Dos and Don'ts

The state object is one of the three elements in the state triplet, and managing the state object is important to keep the state triplet consistent.

The way in which one approaches the state object can have a major impact on the integrity and maintainability of the project. Because it's not as visible as the code or the running resources, the state object may be overlooked until it becomes a problem.

This section presents some state dos and (many) state don'ts to guide practitioners toward effective approaches and procedures away from potentially troublesome practices.

State Don'ts

I find it is more impactful to begin with the problematic practices, especially because many practitioners will relate to both how they can crop up and how time consuming and risky they can be to resolve. Some of these arise from common misconceptions, out of an uncontrolled evolution of a project, whereas others are essentially improper workarounds due to a lack of sound software practices.

One-off operations

TF does not lend itself well to operations that must be run once and only once, or those that are expected to be fire-and-forget. The very nature of TF is to support the consistency of the instantiated resources that make up your IaC ecosystem. Many one-off operations may be more appropriately performed directly with the underlying platform API request or CLI command.

Under typical use, the entire state under execution is checked any time `tf plan` is run. TF does support a `-target` option, which can be used to apply incremental changes and could conceivably be leveraged for one-off operations (maybe rolling some key or incrementing a stored object version). However, the use of `-target` should likely be reserved for emergency situations. You should build policies and governance around its use.

Using TF for one-off operations, especially within a larger, idempotently consistent state, typically includes the cost of additional operational

overhead in the form of encoded guardrails — for example, building abstractions around the `-target` option to enable the update of individual resources within a TF state object and implementing the use of highly controlled, temporarily elevated access controls for additional control over when it is leveraged. While these are generally recommended to introduce as capabilities for use in emergency scenarios, especially for sufficiently large, wide-reaching organizations, they're fairly costly to develop, and they're rarely foolproof.

Ideally, even something like state manipulation is done in a highly controlled and automated fashion. However, especially early in the development of your TF, it may be appropriate to make some concessions in the name of quick issue resolution. That said, incorporating one-off operations into normal TF operation is highly discouraged. The more you deliberately violate the idempotent nature of TF, the more overhead you must engage in and support. This goes beyond using a hammer to screw in a nail — this is like using a screwdriver to pry apart a hammer and using the hammer bits to screw in a nail. To put it another way, you already have the tools you need to address the policy: whatever software that was written to wrap around and operate TF in this nonstandard way can be repurposed to leverage APIs, CLIs, or SDKs directly.

> **TIP**
>
> A good practice is to introduce a *wildcard* job that allows any one of a specified set of TF operations, with the flexibility to allow for a wide range of flags and freetext to enable certain operations that would otherwise be performed manually. An example of this approach is provided in "Approach 5: wildcard jobs". Also, consider using an allowlist for the operations and a denylist for the command flags.

Overusing the -target option

As referenced earlier, the `-target` option allows the execution of write operations to be focused on an individual resource (or all the resources managed by a particular `module` block), rather than the entire TF state object/relevant codebase. And as mentioned, its use is probably best

relegated to emergency corrective situations where you must quickly push a vital change, like a security patch, but a specific resource (often not even in the patch) is currently misconfigured/drifted from the TF state object and is effectively breaking the state; rather than resolving this discrepancy, you instead choose (or are told) to essentially ignore it for now and to force the update through by using `-target` to bypass the full TF state object check against the complete collection of running resources

It is strongly discouraged to include the `-target` option directly into normal infrastructure operations, such as deployment pipeline execution or *self-service* platforms. Use the `-target` option at your own risk and with heightened caution. Overuse may indicate systemic issues that should be addressed more holistically — for example, by refactoring portions of the codebase to enhance its robustness or to isolate the lifecycles of problematic and highly critical resources. It is also a good idea to enforce procedural/access policies preventing the regular use of the `-target` option.

Tightly coupled states

Two states can become tightly coupled when there is a high degree of dependency between them. A contrived example might be one state including iterative VPC and firewall configurations and another state containing the corresponding subnetwork configurations, as depicted in Example 6-1. Whenever a new VPC is added, it will need subnetworks, implying changes to both states in a specific order. If a VPC is removed or modified, it may incur changes to the respective subnetworks, again implying changes to both states in a specific order. If these states are established by codebases owned by different teams, a heightened level of coordination between them is required whenever such a change is made. Even when they're both controlled by the same team, this relationship can cause frustration if other changes to the project containing firewall rules must wait for updates to the VPC project (as well as the subsequent changes to the subnetwork project) before proceeding. I will refer to such a sequence, in which a `tf apply` in one project (the *upstream*) requires a

nontrivial `tf apply` in another project (the *downstream*), as a *cascading* change.

Example 6-1. Two projects with one directly dependent on the other such that updates in the upstream project incur changes to the downstream project

```
#------------------------------------------------------------------
----------
# Upstream module
#------------------------------------------------------------------
----------
#
////////////////////////////////////////////////////////////////////
////////
# main.tf
#
////////////////////////////////////////////////////////////////////
////////
resource "google_compute_network" "tas_6_vpc" {
  for_each = var.vpc_names

  name                    = each.value
  auto_create_subnetworks = false
}

resource "google_compute_firewall" "tas_6_fw_allow_http" {
  for_each = google_compute_network.tas_6_vpc[*].id

  name    = "${var.fw_http_name} for VPC ${each.value}"
  network = each.value

  allow {
    protocol = "tcp"
    ports    = ["80", "443"]
  }
}

resource "google_compute_firewall" "tas_6_fw_deny_icmp" {
  for_each = google_compute_network.tas_6_vpc[*].id

  name    = "${var.fw_icmp_name} for VPC ${each.value}"
  network = each.value

  deny {
    protocol = "icmp"
  }
```

```
}

#-----------------------------------------------------------------
----------
# Downstream module
#-----------------------------------------------------------------
----------
#
/////////////////////////////////////////////////////////////////
////////
# main.tf
#
/////////////////////////////////////////////////////////////////
////////
resource "google_compute_subnetwork" "tas_6_snet_01" {
  for_each = { for vpc in var.vpc_info : vpc.network_id => vpc }

  name          = each.value.snet1.subnet_name
  ip_cidr_range = each.value.snet1.subnet_cidr
  region        = each.value.snet1.subnet_region
  network       = each.value.network_id
}

resource "google_compute_subnetwork" "tas_6_snet_02" {
  for_each = { for vpc in var.vpc_info : vpc.network_id => vpc }

  name          = each.value.snet2.subnet_name
  ip_cidr_range = each.value.snet2.subnet_cidr
  region        = each.value.snet2.subnet_region
  network       = each.value.network_id
}

resource "google_compute_subnetwork" "tas_6_snet_03" {
  for_each = { for vpc in var.vpc_info : vpc.network_id => vpc }

  name          = each.value.snet3.subnet_name
  ip_cidr_range = each.value.snet3.subnet_cidr
  region        = each.value.snet3.subnet_region
  network       = each.value.network_id
}
```

A cascading change can be especially problematic if the projects are mutually *interdependent*, that is, they have dependencies going both directions: one project relies on resources in another, which itself relies on resources from the first project. Mutual interdependency can happen as projects evolve and more and different resources are introduced. To extend

the earlier case, let's say that later on, custom routing rules were added to the project controlling the subnets with the rationale that they were more related to the connectivity and less related to security. Then, even later, some routing rules were introduced that relied on specific firewall configuration. Suddenly, both projects rely upon each other. Here, a cascading change can even make a cycle, where the first project must actually be changed twice — first to make the initial update (subnetwork), then to update the downstream project in response (firewalls), and then to again update the first project (routes).

Uneven state

It's almost impossible (and certainly more trouble than it's worth at the margin) to construct perfectly balanced state objects. However, there are three clear practices to avoid that greatly increase the chances of excessively unbalanced states:

Mixing module and resource blocks in the same root or remote module

> This is probably the easiest to follow and enforce. Chapter 5 discussed this in more detail. For the purpose of this discussion, the point is that such mixing automatically creates imbalance, as the resources represent a leaf node (and thus its depth ends) whereas the resources under its sibling module nodes have a strictly greater depth.

Mixing module types

> Chapter 5 discussed module types. If one module bundles complementary resources that interconnect and behave as a unit (for example, a compute instance, a boot disk, and a network interface), and another module bundles disparate resources (or even other modules) as, for example, an onboarding package (such as a set of IAM controls, a bucket for application logs, and a dedicated subnetwork), these should not be invoked within the same module. This is a lot harder to enforce, especially if the modules are remote and developed by a third party.

Including collection nodes and noncollection nodes in the same code module

By collection nodes, I am referring to modules and resources treated with the `for_each` meta-argument. Like the first practice, this automatically creates an imbalance within the code module. This is also hard to enforce in the presence of third-party code modules.

> **TIP**
>
> It's often best to avoid using the `for_each` meta-argument outside of a root code module altogether. There are some clear exceptions to this (for example, a single subnet bundled with a static set of standard firewall rules). The point is to not introduce this mixing without good reason. This is covered in more detail in Chapter 5.

Sometimes, the only way to truly achieve the desired degree of balance (especially of depth) would be by introducing code modules that simply wrap one of the offending modules, but this is not advised. Most likely, a more appropriate approach is to split up the module, possibly into multiple states (and projects/repos).

Too many cooks

For better or worse, multiple teams working in (sometimes loose) coordination on a single IaC ecosystem is fairly standard. This might happen over time, as individuals move from team to team or are promoted into new roles, and bring with them skills and tribal knowledge that might be tapped in cases of emergency or out of convenience. Or the phenomenon can occur suddenly as teams experience department-wide reorganizations and responsibilities are shuffled among new teams and individuals.

When a single repository is developed by multiple teams, it can quickly cause friction with the presence of any locked or broken states. These are, in turn, at an increased potential, as multiple teams can compound the likelihood of including dissimilar resources with independent lifecycles. Extremely distanced teams or team members can exacerbate these

problems, especially if one team or individual can impede (or outright block) the other project contributors. Issues can be further magnified if the resources within such a project have widely divergent degrees of exposure (such as number of end users impacted), severity (like sensitivity of the data a resource contains or processes), or frequency (like a well-worn remediation path versus a "novel" problem), as this can result in equally divergent levels of commitment applied to the issue's resolution.

In the earlier example, a better way might be to require a formalized knowledge transfer (KT) and documentation activity before a team member is approved to move roles or teams. This can be followed by a strict deprecation period and a set end-of-access date (for example, utilizing an available auto-expiration feature in the IAM platform). Thereafter, any actions taken on behalf of this former team member (after all, emergencies don't just go away, and sometimes it's necessary to invoke an all-hands-on-deck approach to issue resolution) may be governed using temporarily granted access controls.

In the latter example, it's highly recommended to schedule and implement a formalized refactor activity, whereby a small group of individuals (drawn from the relevant teams) conducts a series of steps to encode, validate, and execute the reorganization of resources into projects and TF state objects that may be fully served by a single team. We'll see an example process for implementing this latter recommendation in "Scenario 4: organizational change management".

ClickOps

An emergent phenomenon is most commonly referred to as *ClickOps*, where resources are created manually to fully configure and ensure a working solution and then tagged, named, or labeled to signify readiness for the resource to be incorporated into the IaC ecosystem as code. This can be problematic, especially if anything that is modified manually may be undone by the next `tf apply`. It also doesn't scale well, as any useful operations must be replicated manually if they ever need to be taken again. ClickOps is the antithesis of IaC, with none of the benefits or mitigated

risks. It doesn't mix well with IaC, either. There's a compounding effect whereby mixing IaC and ClickOps results in an amount of effort greater than the sum of its individual efforts, as there is undue repetition, unwanted reversion, and blocking actions as operations drive toward convergence.

Finally, ClickOps presents a poor security posture. Typically, it puts an unnecessarily high level of personal responsibility on individual specialists with the relevant access. Or, in the case of extreme separation of duties, it requires complex coordination among multiple such individuals. In either case, there's a great deal of risk placed on a few team members' physical and mental health (and general desire to stay employed) that can be mitigated only by spreading the responsibility, thereby undermining whatever compliance measures influenced the specialist policy in the first place.

> **NOTE**
>
> While ClickOps is wildly inefficient, especially in contrast to well-architected solutions, it can bridge the gap in the case of inexperience, unfamiliarity, or undercoordination (or perhaps severe asynchronicity). Perhaps a more economical use case would be to reserve ClickOps for rapid prototyping or proof of architecture, after which a full IaC ecosystem architecture can be designed and scaled in TF.

State Dos

Now that we've seen the don'ts, it should be easy to understand the dos and their rationale. The dos are presented as more theoretical than the don'ts, partially because there are many ways to achieve the recommendations here, and any prescriptive path would likely have its own drawbacks or superseding approaches. However, being aware of them, even theoretically, and using them as North Stars can help architects and teams avoid missteps: by implementing these practices, engineers can avoid all of the don'ts.

Balanced states

As previously stated, the TF object is represented as a tree structure in JSON with the root module (as defined earlier) as its root node. Ideally, develop the code in a way that keeps the tree representing the TF object reasonably balanced: that is, with approximately the same volume of child nodes under each branch of each node, with no depth more than $2\log_x(n)$ (with x as the average of the child branches across all nodes), and a difference between shortest and longest depth at most $\log_x(n)$.

This is an example, and while it allows a lot of wiggle room, it's not exactly trivial to achieve as projects get increasingly complex and modular. We'll see how to develop for balance in "Scenario 1: organizational scaling".

Environment-aligned state objects

One of the most common ways to effectively separate state is by environment: production, integration, staging, and so on. There are many ways to implement this, and once a path is selected, changing direction can be difficult (and costly). It's best to implement some kind of environment graduation process whereby some set of core components are configured so that updates can be introduced first in an environment that is not meant for your production workloads. We'll see an example of this in "Approach 3: graduated environment configuration".

> **NOTE**
>
> From the point of view of an operations or infrastructure specialist, all (or a vast majority) of your infrastructure is *production*, as it's live and has active users (the application developers). However, there's a strong and reasonable desire to vet significant changes (such as new resource types or features thereof) outside of your applications' production environments.

It's recommended to drive most of the environmental differences with a combination of versioned remote modules (so we can control when and where a new resource or feature is enabled) and the input variables (to provide the necessary changes in size, enumeration, and other environment-specific configuration). Doing so will enable greater control over creating

and destroying entire environments. We'll see examples of this later in the section "Common Approaches".

> **NOTE**
>
> Regarding ephemeral environments, there are two typical scenarios: those with long-running backends (where the environment is ephemeral but where its state is stored is persistent), and those whose backends are also ephemeral (they're effectively abandoned once the environment is created or destroyed). The major difference is that the former supports only one ephemeral environment at a time, whereas the latter supports any number. This becomes a trade-off between control and flexibility. A compromise might be to give each application team (or appropriate grouping thereof) one fixed ephemeral environment, with the expectation that they will clean up after themselves.

Clean up unused states

This could have been presented as a don't, as in *don't leave unused states lying around*. As you've seen in other sections, and as you'll see later in the chapter, there are times when resources in a particular state (or the entire state itself) must be transferred, removed, or otherwise rearranged. This may introduce new state backends and possibly render existing ones obsolete. Sometimes these now obsolete states persist even if all the resources are removed or relocated, as TF will still make a record of the empty object as its state and store it at the configured backend.

Many organizations have policies around archiving or otherwise preserving nearly everything. Such a policy highlights the value of cleaning up state objects: keeping around state files with nonempty objects that no longer reflect reality can confuse anyone who happens to review them in an audit scenario (the only scenario in which such an archival policy is actually fully implemented). Making sure all state files reflect empty state objects is the best way to show any would-be auditors that your resources are kept clean and fully tracked.

Domain-aligned code modules

You shouldn't keep all of the TF code for an entire global conglomerate in a single state object. On the flip side, it's more of an art than a science to construct code modules that don't do the following:

- Overlap in some way — for example, they both configure resources of the same type somewhere in their state object
- Create cascading dependencies, as described in the section "Tightly coupled states"

BEST AVAILABLE PRACTICES

There is a noticeable lack of best practices for TF. There are a few resources, such as the HashiCorp Terraform Style Guide and the Terraform Best Practices presented by the venerable Anton Babenko, but even taken together, there are gaps in coverage of features and concepts, as well as some conflicting advice. At first, this may seem like a failure of either the tools' developers or the community writ large, but as you progress in your journey with TF, you'll realize that the flexibility of TF is seemingly both a gift and a curse. It's extremely expressive and can be constructed in many different ways that are valid and viable (each bringing their respective benefits and drawbacks), so it becomes difficult not only to choose an approach, but also to stick to it indefinitely.

It can also feel arbitrary to make "improvements" in the name of codebase optimization, state balance, or method alignment. Many times, it's only after a serious issue arises (such as gridlocked state) that state and project organization is even considered, which is often too late to prevent costly issues.

If you determine that you do need multiple projects (and their respective state[s]), it's best to keep their dependencies to a minimum (and eliminate or otherwise mitigate any mutual dependency loops), ideally with the upstream project changing at a slower rate than the downstream. This way,

any cascading changes are also minimized, allowing more controlled (or even atomic) changes.

Ideally, resources within the same technical domain should be grouped together, where appropriate, as much as possible. For example, all the network-related components for a particular workload, platform, or department may be configured in the same module, as depicted in Example 6-2.

Example 6-2. Several resources within the networking domain grouped within a single TF module

```
resource "google_compute_network" "tas_6_vpc" {
  project = "terraform-at-scale-book"
  name    = "vpc-terraform-at-scale-7-7"
}

resource "google_compute_subnetwork" "tas_6_snet" {
  project       = "terraform-at-scale-book"
  name          = "subnetwork-terraform-at-scale-7-7"
  ip_cidr_range = "10.0.0.0/24"
  region        = "us-central1"
  network       = google_compute_network.tas_6_vpc.id
}

resource "google_compute_firewall" "tas_6_fw" {
  project = "terraform-at-scale-book"
  name    = "firewall-terraform-at-scale-7-7"
  network = google_compute_network.tas_6_vpc.self_link

  allow {
    protocol = "icmp"
  }

  allow {
    protocol = "tcp"
    ports    = ["80", "443"]
  }
}
```

Lifecycle-aligned code modules

Lifecycle-aligned code modules are similar to their domain-aligned counterparts, except they group resources that might be updated in the same (or highly related) lifecycle. An example might be the resources supporting

a standardized tech stack that updates in tandem. Whenever one component needs to be updated, the others are likely to need updating as well. Constructing the project that can be reused for the stack allows both bundled maintenance and separation of concerns.

Common State Problems

Before you can optimize your TF state, it's worth discussing the various problems that can crop up organically. By considering these common issues, you'll understand their gravity without having to experience them during an emergency. This section also describes common causes of each issue as well as mitigation techniques, both preventative and corrective.

Not all of these are easy to avoid, even with this information.

Locked States

For backends that support a state lock, which prevents planning changes at the same time as an existing planning or application phase, the *lock file* is the mechanism that protects TF from attempting to make multiple concurrent attempts at writing to state. Especially in the case of automated builds, it can prevent unintended actions. Unfortunately, the lock file also prevents intentional actions, potentially requiring otherwise automated jobs to be manually restarted if the job fails due to the occurrence of a state lock. Optimally, the state is sized, organized, and operated in such a way that each write action (for example, the execution of a `tf apply`) completes before any subsequent write action is attempted (or even planned).

In addition to normal state-locking behavior, there may be certain practices that unintentionally introduce or preserve a state lock. For example, aborting the execution of a write operation, such as a `tf apply` command, may prevent the state lock from being removed (perhaps among other, less straightforward issues). Safely addressing this may require manual intervention or additional controls.

Typically, unlocking a state involves, at a minimum, deleting the actual state lock file in the backend location. Additionally, it may require the reconciliation of the state triplet members: the TF state, the codebase, and the live, running (or paused) resources. Any such remediation activities should be done with utmost awareness and care; a seemingly simple, but incomplete, fix may turn a nuisance into a crisis scenario.

Broken States

Perhaps worse than a locked state is a broken state. A *broken state* occurs when TF discovers the state object does not reflect reality and either reports a plan that makes a number of unexpected changes or fails altogether with an error message. It can also arise with mismanaged resource and module referencing or stale TF or provider versions. In any case, a broken state causes confusion and impedes further development until the state is reconciled (or the deltas are ignored and simply bashed away upon the next `tf apply`). On the extreme (but by no means rare) end, a broken state requires field triage surgery, with an intrepid professional making momentous segmentations of the state tree.

There are possibly uncountable paths to a broken state. Here is a short list of common missteps:

- Misusing `for_each` and `count` expressions. Only use the `count` expression to provision a resource/module conditionally. Use `for_each` sparingly and never use a generated map whose keys are determined by an ordering/sequencing function. These were discussed in greater detail in Chapter 3.
- ClickOps development undoing changes applied with IaC (and vice versa).
- A data resource (perhaps indirectly) references a managed resource elsewhere in the state tree. The nondeterministic data resource is then referenced in another managed resource for a field that forces the replacement of the resource each time `tf apply` is executed

(for some resources, such a temporary toggle of existence, may be largely imperceptible to the system or any users, but it should generally be avoided, nonetheless).

- Version constraints are introduced and not kept up-to-date as the TF CLI version is advanced per organizational policy, resulting in a version mismatch.

In terms of resolution, it's almost always on a case-by-case basis. However, in many cases, the resolution involves first analyzing all the relevant resources as they appear or are recorded in each of the state triplet components: the resource's entry recorded in the latest TF state object, the corresponding encoded configuration you're attempting to apply, and the resource's status as reported in the cloud provider's web interface. This way, any discrepancies can be identified and addressed as needed (for example, by correcting one or more of the state triplet components).

Gridlocked States

A *gridlocked state* is when the state is such that any subsequent `tf apply` operation will fail, regardless of the state of the codebase (whether the codebase has been modified since the last run or not). This is rare, but it can occur if the resource instance was modified in some way that cannot be resolved cleanly with a code change.

One way this can occur is if an unsuccessful `tf apply` cannot fully undo an operation. A contrived example might be `tf apply` that fails to configure one resource, while it successfully creates a resource that is designated as protected by the CSP: the failed resources causes TF to attempt to rollback its updates, but the protected resource refuses to be cleaned up given the protection. Suddenly, any new `tf apply` (even those that correct the failure with the one resource) now fails because the protected resource already exists. The only options are to remove the resource another way (preferably not manually) or to remove the code that configures the protected resource to at least move past the gridlock.

Note that this latter approach leaves an untracked resource. It's possible to later import the resource into the state, as we've seen before.

Bloated States

When the state tree has grown so large that it takes too much time and resources, it may begin to cause severe slowdowns to a `tf plan` or `tf apply`. Even with enough resources, the running time can be prohibitive in the case of a typical development team collaborating to rapidly construct a single state. Suddenly teams are met with artificial barriers to introducing their latest solutions or, far worse, a much-needed security patch.

> **WARNING**
>
> The *blast radius* — the resources affected by a particular issue or defect — of a broken or locked state is the entire state tree. Therefore, keep state trees as self-contained as possible, without creating monolithic monstrosities that take the whole company to a standstill when a wrench makes its way into the works. All resources managed by a single state share a single lifecycle. Consider using a domain-driven approach to chunk states to a reasonable size and level of atomicity, as described before.

Without providing benchmarks or other evidence, it should be fairly reasonable to assume that big states take longer to traverse than small states (assuming similar resource types are targeted in proportion, equivalent compute and networking capacities are leveraged, and all other things are equal from an Occam's razor perspective). Logically, we also want a high ratio of leaf nodes to state nodes (for instance, a high ratio of `resource` blocks to `module` blocks) so we're spending memory and state data efficiently on running resources instead of on structural metadata. Thus, in theory, breadth is preferred to depth, as cardinality in breadth always results in a more favorable ratio, whereas greater cardinality in depth always results in a less favorable ratio. To be fair, this isn't necessarily the most critical place to optimize. If an abstraction makes your code more manageable (from any conceivable metric, including accessibility to future maintainers/operators), don't lament the extra requisite memory.

Perhaps unintuitively, this balance of leaf to state nodes can also affect code volume because separate states may introduce data blocks in the case where one state's resources reference a resource maintained by the other state. These references are very useful to ensure the dependencies exist, but they also duplicate (at least) every relevant code change. When the data references are *hot*, or change often, this can be cumbersome. Data blocks are typically evaluated during the execution of a `tf plan`, allowing anyone with read access to validate the existence of referenced resources.

On the other hand, the over-presence of DAGs in multiple places along a state hierarchy implies either a bloated state or too deep of a nesting and would be better served by separating the different module references into distinct state trees. The balance between many hot data references and a bloated state is certainly a key (albeit more qualitative) indicator of IaC maintainability.

The worst case scenario with bloated states is one that takes an extraordinary amount of time to complete the `tf plan` operation, and only one thing ever really needs to be updated. Let's say all your hundreds of DNS entries and thousands of firewall rules for the nonproduction digital assets of a massive multinational conglomerate are configured in the same codebase. The network architects were extremely clever in the network design, and the only change that ever needs to be made is a handful of new DNS records, which may be requested to be created or destroyed at will by application teams in nonproduction environments. Despite only a few actual changes, the entire state needs to be checked against what is in the codebase and what is configured in the cloud. It may not be noticeable to the end-user engineers, but it's clearly something that wastes the compute resources of your software delivery lifecycle apparatus.

Too Many States

The opposite of *too large* when it comes to a state object is *too many* states to manage. This can happen in a number of situations, but they all typically fall into two categories (or, most likely, a combination thereof): separating by service type and separating by product/service offering. Separating by

service type might see a separation between the VM instances and their respective firewall rules, creating, at a minimum, an intrinsic coupling between tangentially related projects. Separating by product/service offering might overfit to a particular technology stack, causing a deviation from standard project structures. In practice, these challenges intertwine, creating even more frustrating results: suddenly, you're supporting countless exceptional behavior across what is supposed to be a set of standard configurations.

If you find yourself tracking your states using a spreadsheet (or wishing you were), you may be suffering from too many states. It's recommended to consider the states in terms of lifecycle and domain to determine which, if any, can be stitched together into a larger state. Even if two states essentially share a lifecycle, their codebases might be structured in a way that does not lend well to tying them together, especially if either codebase is already servicing multiple states. The best way to address the too-many-states problem is to try to avoid it, if you're using the techniques mentioned earlier, *before* it occurs.

Unbalanced States

While there's nothing inherently wrong with a state tree that is lopsided or jagged, I would suggest that uneven distributions of resource definitions increase the overall complexity of troubleshooting. The longer the state hierarchy, the more places that may require attention when resolving a bug. Typically, a new variable definition at a leaf node requires analogous definitions at ancestor nodes. This creates a cascading effect up the state hierarchy, often all the way to the root. Sometimes it's not possible to avoid some unevenness, but having a high ratio of *nesting* cardinality to *node* cardinality along a *longest node* ancestry (for instance, a giant module with a high degree of child submodules somewhere buried down a module chain) is more typical of haphazard detachment than mindful abstraction. Figure 6-6 shows an excessively unbalanced state tree.

A more measurable outcome of an uneven state tree is a severely uneven distribution of activity throughout the tree. Even in well-balanced state

trees, some nodes may see more activity than others; this is neither problematic nor especially interesting on its own. However, when a tree has a few resources that change often/regularly with many resources that change rarely/irregularly, it can suggest an unnatural coupling of the constituent resources. Certainly, including unrelated resources into the same state tree can be especially irksome with locked or broken states, when the offending state node is unrelated to the intended resource changes.

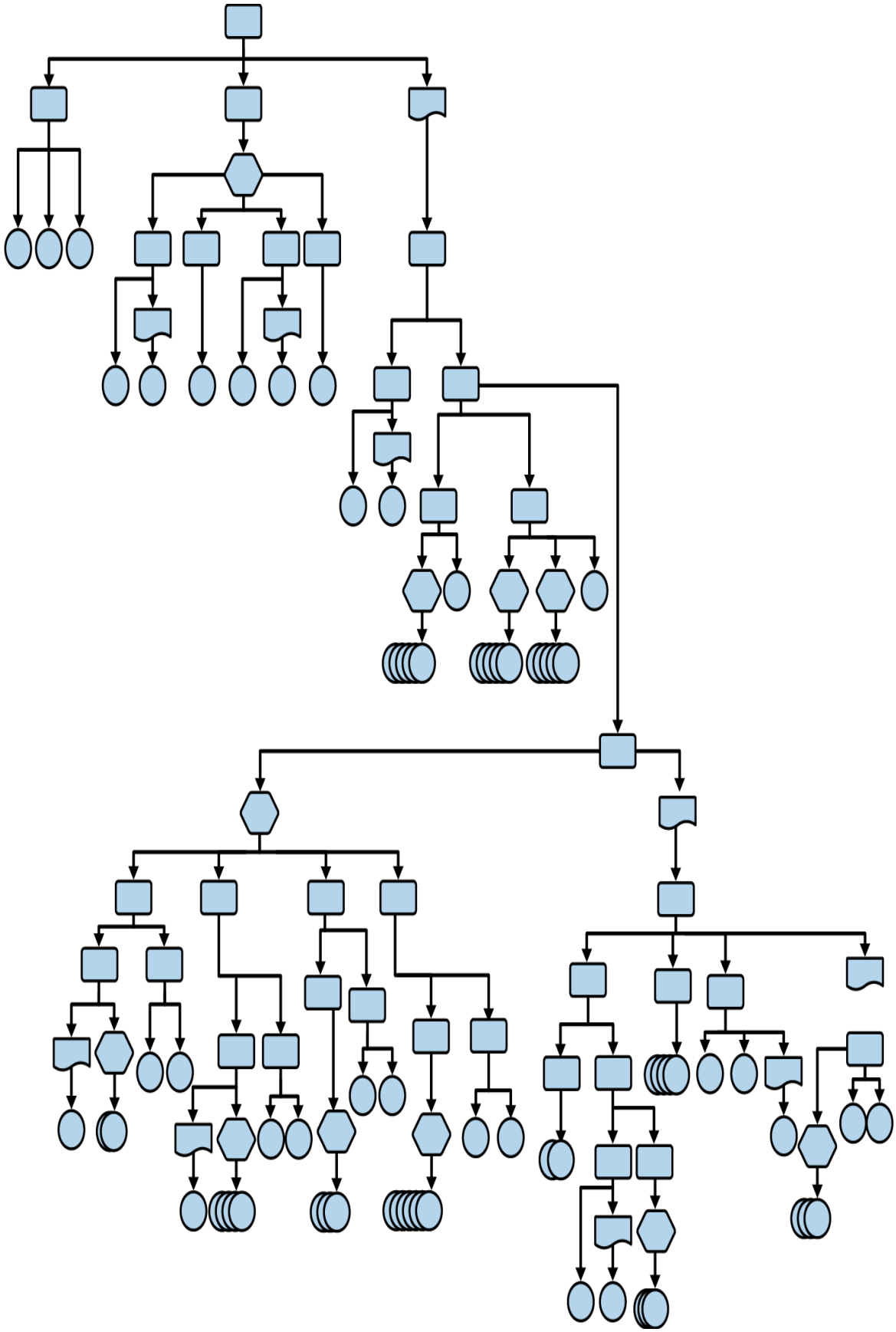


Figure 6-6. A state object constructed in an excessively unbalanced fashion

Never-Clean States

TF has several interfaces for generic operations, allowing uncountable architecture structures and chains of events in a deployment procedure. Unfortunately, not all of these can be effectively tracked by TF, especially if they're recorded in any way into the state objects. For example, if you were to use a data source to retrieve a nondeterministic value that is used for the field of a resource, it's possible to break the idempotent nature of TF, and even executed `tf apply` commands that should be no-ops have to be computed, leading to what TF believes to be valid changes.

A fairly common scenario is when a resource field has an overloaded format that the provisioner interprets only one way. Normally, this causes a no-op operation, but if modifying the field causes replacement, it will first destroy the resource each time before it gets to the replacement portion. An easy fix for this is to construct the preferred format of the field so that it matches in the state object, and the `tf plan` doesn't report any changes.

Building for Enhanced Maintainability and Balance

Ultimately, the goal is to make software changes quickly, safely, and with confidence. This means not blocking one project because of an issue in another. It's not having to make changes in almost every project in order to introduce (or tear down) an entire software stack for a specific product/service. It's not having to engage more than one team in a responsible/accountable capacity.

This goal is not as easy as it sounds, especially at enterprise scale, but it's also not impossible. This section reviews common approaches that can help avoid the problems discussed earlier that lead to an imbalance across the various aforementioned dimensions. I will then describe common scenarios

that may arise (and some that there are essentially no way to avoid) and discuss how to work forward toward a new position of balance.

Common Approaches

So far, I've discussed approaches that are effectively *don't do X* or *try to do Y*. While this guidance is certainly sound, and its generic nature is indicative of the number of solutions and techniques that can produce valid, maintainable, and effective codebases, it isn't exactly the most satisfying or actionable. Rather than being implemented directly, the guidance first must be interpreted and then mapped to a particular organization and infrastructure before it can be developed as code.

In this section, we'll look at a few tactical approaches that can be incorporated directly. It's still up to you (and your architects and others) to determine when to introduce these approaches, if at all. Not every approach necessarily needs to be introduced in every code repository, and in fact there may be sound reasons (or compliance policies that dictate) to *not* incorporate an approach that would be theoretically advantageous. At the very least, these should illustrate a philosophical perspective to balancing TF ecosystems and codebases.

Approach 1: backend key variation

Often, the same resource or module configuration needs to be applied multiple times, with slight variation in naming or other identification-related fields. Multiple similar environments for a single workload may be so configured: this way each environment has the same shape, keeping parallel copies of the same architecture, possibly for various testing or software integration purposes. Alternatively/additionally, you may want to share the same, standardized configuration across multiple workloads, thereby reducing the variation and complexity of your infrastructure as more workloads and teams are onboarded into the TF ecosystem.

In this case, it's generally not necessary nor desirable to include all such configurations into the same TF state object. In the case of multiple

environments, it's beneficial to have the state objects exclusive in case one needs to be reset. In the other example of a standardized configuration shared by multiple products and teams, it's often desirable to keep the state objects separate and allow for each workload's infrastructure to have its own software delivery lifecycle.

The question then becomes how to use the same codebase with multiple state objects. Luckily, there is no direct relationship between a particular repository and a state backend. As you've already seen, each state object has a unique backend (which is populated after the most recent `tf apply` operation). Therefore, to introduce multiple, distinct state objects, we must introduce multiple, distinct state backends. This way, a single repository may service multiple states.

Fortunately, the `backend` configuration fields may be partially configured and then passed in at execution time using the `-backend-config` option. Suppose you use a Google Cloud Storage (GCS) bucket as your backend for your TF state objects. In this approach, the prefix field (this would be equivalent to the key field in AWS or Azure) is partially configured by including either a default or an empty value. Then, in the execution of a `tf init` operation (or equivalent), the updated values designating a valid and unique bucket location are passed in. This can be done either by setting the fields individually (such as via `-backend-config="KEY=VALUE"`) or by targeting a configuration file (via `-backend-config=PATH`, as depicted in Example 6-3), and can be designed to guarantee that the copy of the configuration that TF will use during the `tf plan` operation is distinct. Each execution against different backend keys/prefixes will result in distinct and exclusive state objects, providing the desired separation across environment or workload.

Example 6-3. Setting partial backend configuration and passing in the values from a config file at time of `tf init` *execution*

```
#
////////////////////////////////////////////////////////////////
////////
# state.tf
#
```

```
////////////////////////////////////////////////////////////////////
////////
terraform {
  backend "gcs" {
    bucket = ""
    prefix = ""
  }
}

#
*****************************************************************
********

#
////////////////////////////////////////////////////////////////////
////////
# state.config
#
////////////////////////////////////////////////////////////////////
////////
bucket  = "terraform-at-scale-book"
prefix  = "examples/6-3"
```

> **TIP**
>
> To provide increased governance and protection of the TF state objects stored in remote backend locations, it's highly recommended to leverage an external location (or even a secure data store, such as Hashicorp Vault) to store backend configurations. In that case, the file must be downloaded to the local disk before running the `tf init` command.

Approach 2: specifying distinct variable files

In varying the state backend, you're likely looking to intentionally create distinct backends. While there are use cases or combinations of resources in which these states can be successfully constructed from the exact same inputs and codebase, in most cases you'll want to provide distinct variables for the different TF states you're managing. This is especially true if any components would clash (for example, network ranges, resource IDs, storage bucket names, and so on), and thus applying more than one copy of the same configuration would be rejected by the CSP.

Luckily, TF provides multiple ways to provide input values, with a well-defined precedence among them. If you're using the enterprise Terraform UI or a custom-built automation solution (for example, using the `-var` flag) you can use the platform's variables to set custom input variables. These can be preconfigured with default values and then overridden in the platform's UI as needed.

Another common approach is to introduce multiple *.tfvars* files, e.g., one for each TF state as constructed through the various methods described previously (for example, by environment, domain, or lifecycle). These might be named in ways that signify the application, department, domain, or workload to which the TF state object pertains, or they may be nested in subdirectories whose path expresses such designations.

In all cases it can also be effective to have a common *.tfvars* file for common input values along with the state-unique *.tfvars*. These might be common, nonsensitive variables such as a department code, standard cloud region, or project/subscription ID. Generally, depending on how these files are named (and in certain cases, how they're invoked), the variable assignments in a common file would be subject to being overridden by other applications of *.tfvars* files later in the precedence or by invocations of the `-var` option.

INPUT PRECEDENCE

Given the precedence of inputs and context of the TF command, consider structuring your input values in a way that coincides with the following precedence:

Environment variables

> These are the first to be checked, so they're the most vulnerable to change; it is advised to use this sparingly, perhaps only for necessary variables that have no default value, and that would be significant only to the team configuring the TF automation.

The terraform.tfvars *file*

> This is a generic file that gets automatically read if present in the directory in which the specific TF command is run; it may be most appropriate for common configurations as described earlier.

Any *.auto.tfvars *files*

> These are applied in lexical order, meaning you have to construct their names in the correct order of precedence, again in the directory in which the TF command is run. This can be a great way to introduce an automatically applied configuration that's by design intended to override any previously assigned variable values; however, it's ill-advised to use multiple such *.auto.tfvars* files in a single directory because it complicates the order in which variables are assigned; this might be most appropriate in the case where there are exactly two available configuration files in the directory within which the TF operation was run: the *tf.tfvars* file for normal variable assignments and, for example, an *overrides.auto.tfvars* file for variable assignment value overrides.

Any `-var` *and* `-var-file` *options*

> These are applied in the order they appear in the particular TF command (or in the order they appear in the HCP Terraform workspace); this gives developers of the automated configuration scripts (or workspace runs) the most control over the precedence with which configuration files are read; as such, utilizing these options can produce highly expressive and structured layering of variable files and assignments. However, they also introduce complexity into the structure of the specific TF command, which is difficult or impossible to validate (perform, for instance, static code analysis). Use `-var-file` whenever there are other *.tfvars* files (not *.auto.tfvars*) that need to be applied, in any case where the variables files are stored in a directory other than the one in which the TF operation is run from — or really in any configurations that require more than two variable files. And it's recommended to include any use of `-var` after all uses of `-var-file`, giving them the highest precedence among variable assignment mechanisms.
>
> Avoid mixing the use of **.auto.tfvars* files with `-var-file` options. There are no doubt countless configurations that are perfectly safe and reasonable, but any such configuration is left as an exercise to the reader and is in no way endorsed by the content in this section.

There may be an inclination to impose a precommit hook to disallow changes to certain configuration (*.tfvars* or *.tfvars.json*) files and include another check in the automation script that runs the TF operations. This seems especially attractive with default configuration files that theoretically should not be changed (perhaps as part of a configuration factory pattern whereby a configuration is copied and configured as part of an automated onboarding process). Unfortunately, imposing such "protection" on one configuration file does not prevent it from being superseded by another configuration file that appears later in the precedence.

A possible (though not perfect) alternative solution could be to encode any extremely important and immutable variable assignments as local values in

a designated local values-only file (for instance, a *static-locals.tf*) and then to put the aforementioned file protections on this file. Because duplicate local values are rejected by TF, there would be no way to override or modify these values without one explicitly working around or undoing these file protections. See Chapter 4 for more information.

TF allows for both the *.tfvars* (a simplified encoding for assigning values to variables developed by HashiCorp) or *.tfvars.json* (JSON-encoded) constructions for assigning variables. While it's technically possible to include both encodings, taking advantage of additional layers of precedence, it's most certainly a more convoluted solution than it is worthwhile. Choose either encoding and stick with it.

Suppose you're developing a solution in which standardized cloud configurations are effectively resold internally as part of a "self service" infrastructure platform. There are some configurations that must remain static, some that must be common across all workloads but that may change over time, others that enable optional (but still standardized) portions of the configuration, and many more that are expected to be the full purview of the application team for which the infrastructure was requested.

One possible valid configuration might have the static configurations encoded as local values (rather than variables) in a *static-locals.tf* file with user-configured file protections as described previously; those that must be common across all workloads, but that may change over time, may be stored in a *terraform.tfvars* file in the root directory; those that govern the optional components in an *optional.tfvars* variable file located at a workload-specific subdirectory path; and those that are fully workload-specific in a *workload.tfvars* variable file also located at a workload-specific subdirectory path. Then, as part of the command that plans and/or applies a TF configuration, include `-var-file=./workload/optional.tfvars -var-file=./workload/workload.tfvars` at the end, as depicted in Example 6-4.

Example 6-4. Multiple variable files to allow for complex layering of variable assignment

```
#
////////////////////////////////////////////////////////////////////
////////
# locals.tf
#
////////////////////////////////////////////////////////////////////
////////
locals {
  private_access_static = true
  routing_mode_static   = "REGIONAL"
}

#
*******************************************************************
********

#
////////////////////////////////////////////////////////////////////
////////
# terraform.tfvars
#
////////////////////////////////////////////////////////////////////
////////
network_project_id_common = "terraform-at-scale-chapter-6"
network_mtu               = 8896

#
*******************************************************************
********

#
////////////////////////////////////////////////////////////////////
////////
# app-configs/app-1/optional.tfvars
#
////////////////////////////////////////////////////////////////////
////////
subnet_count_optional   = 3
subnet_logging_optional = false

#
*******************************************************************
********

#
```

```
////////////////////////////////////////////////////////////////////////
////////
# app-configs/app-1/workload.tfvars
#
////////////////////////////////////////////////////////////////////////
////////
workload_name = "app-001"
allowed_ports = ["80", "443"]
```

Approach 3: graduated environment configuration

Sometimes there is a policy, written or otherwise, of taking a graduated approach to any software release process, including IaC, such that the infrastructure for each environment would be applied at different stages of the release process; for example, the Integration stage of the IaC automation maintains the infrastructure for the Development environment of application workloads, the Staging stage maintains the Test environment, and the Deploy stage maintains the Production environment.

> **NOTE**
>
> It's worth reiterating: all of your infrastructure should be treated as production in terms of the quality with which it's developed and the security measures taken to protect it.

The most common problem that may be encountered almost immediately with this approach is when an experimental feature is introduced in a lower environment, say Development, but before it can be fully evaluated, a different change (perhaps required for security purposes) must be expeditiously applied to all environments. The experimental feature must be removed before the required changes can be introduced (or they can be commented out, but this will elevate the comments to all stage branches, which is very likely not desired).

Really, this approach is perhaps most appropriate when piloting new services or features that are enabled in only sandbox environments that exist outside of an application lifecycle stage, such as an experimental testbed. The new capability can be introduced to the testbed, and duplicates of

production workloads can be staged on it, either to ensure stability or to pilot new application features that utilize the new capability.

Graduated environments could be governed by a protected main branch that applies the infrastructure for the lifecycle environments, and the ability to create feature branches that can be deployed to a sandbox copy of the infrastructure with the new capability enabled, perhaps as a webhook that is triggered by the source control system upon opening a pull/merge request. Then feature branches of application workloads can be deployed to the testbed. This way, there is nothing impeding patch activities to either the application workloads or the cloud infrastructure responsible for hosting application workloads in any environment supporting the software delivery lifecycle.

Approach 4: versioned environment modules

Many times, the true motive behind imposing a graduated approach to IaC is improving maintainability and reducing redundant configuration by way of using the same configuration for multiple environments, while also allowing for elevated control over when the configuration is applied. An example is the ability to fully validate an infrastructure configuration (even a patch) before it's applied to the environment hosting the production application workloads. This goal is both understandable and noble, but it's not exactly straightforward.

Simply putting all the configuration in a single code repository, with distinct *.tfvars* files containing the environment specific configurations and applying the *.tfvars* file corresponding to the lifecycle stage again requires a very linear procedure to the release of changes. This runs into the same risk mentioned earlier, whereby a change must be backed out before a patch can be fast-tracked through the stages.

A cleaner approach would be to leverage remote modules and module versioning. In such an approach, the entire environment is configured in one module (the remote module), but it's applied through a single module's source reference in a separate root module, as depicted in Example 6-5.

Example 6-5. A wrapper module (at the TF state root module) that references a versioned remote module hosted in the Terraform module registry, passing through the variable values

```
module "my_remote_module" {
  source  = "terraform-at-scale/chapter-6/remote"
  version = "0.1.0"

  field1 = var.remote_field1
  field2 = var.remote_field2
  field3 = var.remote_field3
}
```

While this doesn't fully resolve the risk of having to leapfrog a more experimental capability with a patch, it at least simplifies the activity of applying such a patch throughout your cloud infrastructure to a single line of code (the module's version).

Approach 5: wildcard jobs

As discussed earlier in this chapter, there are certain operations that may be appropriate only in very limited situations, such as those that directly manipulate the state object that acts as the memory for TF. These are so unique and surgical in nature that they don't lend themselves well to the rigidity of typical automation procedures.

A successful approach would be to develop *wildcard jobs* that allow for the flexibility necessary to fully operate a TF project without ever having to type into a shell yourself. Such a job can be constructed so that it supports a variety of operations, allows for a wide range of flags, and even supports portions of free text to, for example, assign flag values.

In most automation platforms, users can be permitted to operate particular jobs through the assignment of RBACs or attribute-based access controls (ABACs) within the platform, which is a good way to protect the wildcard job from misuse. An extreme (but extremely effective) version of this might first require obtaining a temporary role or attribute through some request process.

Depending on how your total ecosystem's infrastructure is organized (and how access to your automation platform is scoped), it might be possible to provide such wildcard jobs associated with various points throughout the ecosystem and controlled via RBAC/ABAC. This can enable a particular team to address an issue they're experiencing using highly dangerous TF operations without the risk of affecting any other infrastructure.

> **NOTE**
>
> One major value to a wildcard job is that you can have it record the operations the same way that it records its other processes. This ensures consistent and complete tracking of all TF operations — especially important for the operations discussed here, which can have severe implications and should be recorded accurately and transparently.

While we're purely discussing the use of TF commands that modify the TF state object, the concept of wildcard jobs is not in any way confined to TF state and can certainly be extended to many other areas. Before doing so, however, consider the following guidance regarding the use of wildcard jobs for interacting with the CSP, either directly through Google Cloud (gcloud), AWS, or Azure CLIs, or indirectly through TF CLI.

Usually, the automation platform has its own identity (or a collection of identities) that it uses to authenticate itself with the CSP and with which it's permitted to perform actions through IAM policies (such as a *service account* in GCP or an *IAM role* in AWS and Azure). Generally, this is the desired behavior, as the automation platform is expected to operate autonomously (hence the designation; it's best practice to enforce developers to use unique and accurate identifiers when committing a change to a source repository and for this information to be displayed in the console logs along with the commit ID hash), and therefore should be identified as the operator in, for example, the cloud provider's system access logs. However, in the case of wildcard jobs, require special (preferably temporary) permissions to execute the job and also identify the human operator, rather than the automation's identity (service account/role), in all

actions. Otherwise, the operation of this job is effectively anonymous as recorded in the cloud system access logs.

This means that any cloud IAM permissions necessary to perform the operations on the cloud resources must be extended to this human operator. If supported, leveraging impersonation may be effective, whereby a human operator's identity acts as though it's the automation platform's identity, and this is recorded in the system access logs for audit and tracing purposes. If impersonation is not available (or if it's disallowed), it's again strongly encouraged to extend these permissions only temporarily, with some way to revoke or automatically expire them after an appropriate length window.

Additionally, it's important to remember that because TF requires access to a valid set of code against which to operate commands it is issued (for example, located in the directory in which the TF command is executed, or identified via `-chdir` option, and so on), you should always construct the wildcard jobs to freshly pull down/clone some source repository as an initial step on each execution of a job. In addition to generally being able to operate against a particular TF state object, this also allows for certain other approaches (which may even be necessary for operation of a project): specifically the backend key variation approach, leveraging a string replacement solution to swap in the desired TF state backend location key for the placeholder text. The wildcard job can be constructed to first modify the backend configuration using, for instance, `sed` or `awk` before running the desired TF command. Note that this feature requires the backend key/prefix be exposed as an editable (or selectable) field in the wildcard job.

Common Scenarios

Probably the toughest part of any TF ecosystem is when changes need to be made for a specific reason. Every organization that utilizes TF to maintain its infrastructure should expect to see changes in how the TF code, and its corresponding states, is constructed. Many of these events are impossible to avoid (at least, technologically or logistically) and others are natural occurrences in the lifespan of a normal ecosystem.

> **TIP**
>
> In each of the suggested approaches provided in the following scenarios, run any TF command (`tf plan`, `tf import`, `tf state rm`, and so forth) in a wildcard job as described earlier. In absence of such a capability, the utmost care must be taken in executing these commands. It is also strongly recommended that all steps be done in as narrow a window as possible without recklessly rushing or skipping any steps or compromising the validation of a step's success. Certainly do not leave for the weekend before you complete any of the procedures in the following sections.

Scenario 1: organizational scaling

One scenario many organizations find themselves in over time is an expansion of the technology department, with teams growing, splitting, and replicating to support the growth of technology solutions offered to customers. Frequently, this expansion implies an evolution of responsibilities across the various teams. This means a team may become responsible for portions of a particular TF project. More importantly, it means other teams are supposed to be restricted from modifying said portions. The most effective way to impose such distinction of responsibility begins with separating the configuration into separate locations (such as distinct repositories) within the codebase.

Perhaps you started at a series-A startup a few years ago that had one big cross-functional development team, and at the time, the decision was made to combine the configuration for all the cloud network resources into a single repository and maintain it in the same automation pipeline (thereby creating a single comprehensive TF state object). At the time, it made so much sense: the engineering department was only a dozen folks, each comfortable taking on any software task. Although the architecture was effectively defined by the CTO and VP of engineering, everyone on the team was responsible for maintaining all parts of the network. Splitting it up into separate projects would have been arbitrary and would only have introduced unnecessary complexity; now, instead of a single project to maintain, the same team has to juggle two that together make up the components for which they're responsible.

Years later, the company has expanded by multiple magnitudes and reoriented its software development team makeup. Separate teams have been constructed that together manage the following networking components:

NetSec

> The Network Security (NetSec) team is responsible for the perimeter of the cloud components, configuring external-facing load balancers and DNS zones, baseline DENY firewall rules, and any other organization-level network controls exposed by the CSP (for example, GCP's VPC service controls).

NetEng

> The Network Engineering (NetEng) team is responsible for all internal networking configuration, including internal networking components such as firewall rules, routes, or subnets, for individual application teams.

The NetSec team wants direct (and sole) control over the configuration for which they are responsible. The NetEng team has taken the approach of allowing some components to be further configured (or even overridden) by application teams. This implies the codebase is now divided into at least two (and potentially many more) TF projects — a one-to-many transformation.

To achieve this as effectively as possible, you'll want to avoid dependencies between the projects, especially the core NetSec project and the NetEng project(s). This means both the NetSec and the NetEng projects may have the same types of resources represented. That is, each could have its own sets of VPCs, subnets, firewall rules, and DNS records. Meanwhile, though you want to allow for application team-specific configuration, you also want to avoid having too many contributors with different agendas. This means you'll likely want to maintain a single, centralized set of configuration and provide a means for individual application teams to

include their configuration in a standard and structured way. Figure 6-7 depicts the split into NetSec and NetEng projects.

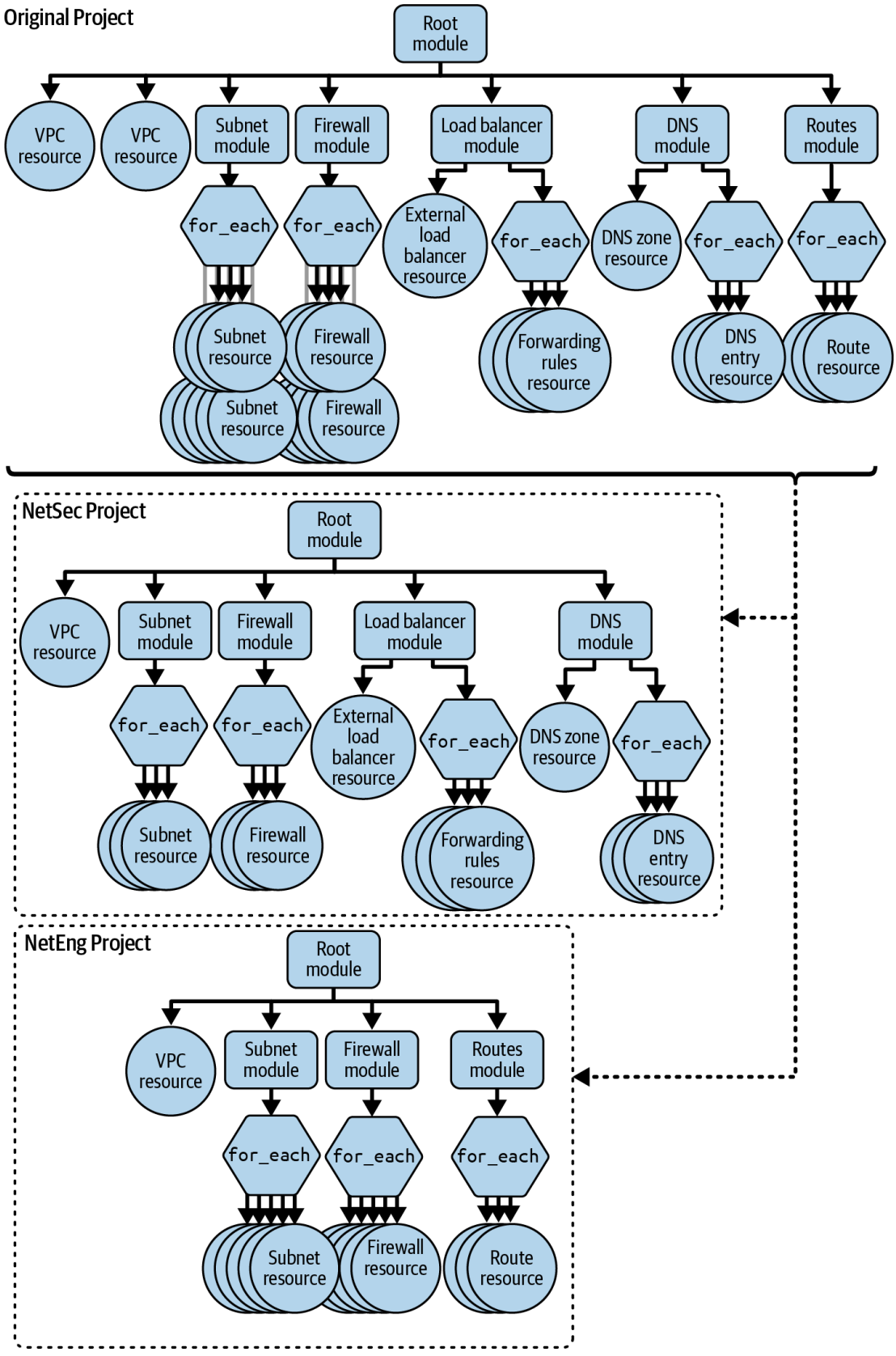

Original Project
Root module
VPC resource
VPC resource
Subnet module
Firewall module
Load balancer module
DNS module
Routes module
for_each
for_each
External load balancer resource
for_each
DNS zone resource
for_each
for_each
Subnet resource
Firewall resource
Subnet resource
Firewall resource
Forwarding rules resource
DNS entry resource
Route resource
NetSec Project
Root module
VPC resource
Subnet module
Firewall module
Load balancer module
DNS module
for_each
for_each
External load balancer resource
for_each
DNS zone resource
for_each
Subnet resource
Firewall resource
Forwarding rules resource
DNS entry resource
NetEng Project
Root module
VPC resource
Subnet module
Firewall module
Routes module
for_each
for_each
for_each
Subnet resource
Firewall resource
Route resource

Figure 6-7. Network infrastructure before and after splitting into two projects (to be maintained moving forward by distinct teams)

Most likely, the evolution described happens over time (possibly in a trial-and-error approach). However, if it somehow needed to be implemented in a big bang format, it could unfold in the following way (application team-specific configuration not included):

1. *Create a new repository*

   This will house the code for the NetEng team.

2. *Copy over the code*

   Cut/paste the NetEng configuration into the new repository.

3. *Discover imports*

   Run a `tf plan` in the new repository (for the NetEng team) to confirm where to import all resources.

4. *Run imports*

   Using the addresses reported in the previous step, import each resource, one by one, using the `tf import` command.

5. *Confirm imports*

   Run another `tf plan` to confirm the `tf import` commands succeeded; no changes should be reported if the imports were successful.

6. *Discover removals*

   Run a `tf plan` in the repurposed repository (which will now be owned by the NetSec team) to confirm which resources to remove.

7. *Run removals*

   Run `tf state rm` on each resource reported in the previous step (or, if you find that you're removing all the resources under a particular

module, simply use `tf state rm` to remove the entire module and all its constituent resources).

8. Confirm removal

Run another `tf plan` to confirm the `tf state rm` commands succeeded in the repurposed repository; no changes should be reported if the removals were successful.

Once you have completed these steps, each team can operate more or less independently with a distinct cadence and lifecycle, coordinating loosely with each other in a producer-consumer capacity, wherever necessary.

Scenario 2: reducing duplication

Despite your best efforts, you may find that as the TF code evolves over time, various instances of working configuration are copied and pasted for various reasons, even into the same repository. This duplication of code may have been done purposefully and with salient rationale, or it may have been introduced by blind mimicry. One or a few appearances of this may be a tolerated nuisance, but bad behavior has a tendency to propagate, and just a few occurrences can quickly turn into several and then into multitudes.

Suppose you're charged with refactoring your department's networking components that have, over time, recorded in what has become a massive flat TF file with dozens of repeated copies of interrelated subnet, firewall, and network route configuration blocks. Each repetition follows a standard baseline pattern (with some variability in the number of firewall rules) and is associated with a typical workload. The goal is to first develop a module that can manage these resources as a bundle for each workload, as shown in Example 6-6, and then to split the TF state object into dozens of workload-oriented TF state objects, so that each can have a unique lifecycle.

Example 6-6. Before and after the refactoring of repeated resource blocks

```
#------------------------------------------------------------------
----------
# Before refactor
#------------------------------------------------------------------
```

```
----------
#
////////////////////////////////////////////////////////////////////
////////
# main.tf
#
////////////////////////////////////////////////////////////////////
////////
resource "google_compute_network" "tas_6_vpc" {
  name                    = "vpc-terraform-at-scale-6"
  auto_create_subnetworks = false
}

resource "google_compute_subnetwork" "tas_6_snet_01" {
  name          = var.snet_01_name
  ip_cidr_range = var.snet_01_cidr
  region        = var.snet_01_region
  network       = google_compute_network.tas_6_vpc.id
}

resource "google_compute_subnetwork" "tas_6_snet_02" {
  name          = var.snet_02_name
  ip_cidr_range = var.snet_02_cidr
  region        = var.snet_02_region
  network       = google_compute_network.tas_6_vpc.id
}

resource "google_compute_subnetwork" "tas_6_snet_03" {
  name          = var.snet_03_name
  ip_cidr_range = var.snet_03_cidr
  region        = var.snet_03_region
  network       = google_compute_network.tas_6_vpc.id
}
...

#
****************************************************************
********

#
////////////////////////////////////////////////////////////////////
////////
# variables.tf
#
////////////////////////////////////////////////////////////////////
////////
variable "snet_01_name" {
  description = "Subnet name."
```

```
  type        = string
}

variable "snet_01_cidr" {
  description = "Subnet CIDR as a string."
  type        = string
}

variable "snet_01_region" {
  description = "Subnet region in GCP."
  type        = string
}

variable "snet_02_name" {
  description = "Subnet name."
  type        = string
}

variable "snet_02_cidr" {
  description = "Subnet CIDR as a string."
  type        = string
}

variable "snet_02_region" {
  description = "Subnet region in GCP."
  type        = string
}

variable "snet_03_name" {
  description = "Subnet name."
  type        = string
}

variable "snet_03_cidr" {
  description = "Subnet CIDR as a string."
  type        = string
}

variable "snet_03_region" {
  description = "Subnet region in GCP."
  type        = string
}
...

#
**********************************************************************
********
```

```
#
////////////////////////////////////////////////////////////////////
////////
# terraform.tfvars
#
////////////////////////////////////////////////////////////////////
////////
snet_01_name   = "tas-6-snet-1"
snet_01_cidr   = "10.0.0.0/24"
snet_01_region = "us-east1"
snet_02_name   = "tas-6-snet-2"
snet_02_cidr   = "10.0.1.0/24"
snet_02_region = "us-east4"
snet_03_name   = "tas-6-snet-3"
snet_03_cidr   = "10.0.2.0/24"
snet_03_region = "us-east5"
...

#-----------------------------------------------------------------
----------
# After refactor
#-----------------------------------------------------------------
----------
#
////////////////////////////////////////////////////////////////////
////////
# main.tf
#
////////////////////////////////////////////////////////////////////
////////
resource "google_compute_network" "tas_6_vpc" {
  name                    = "vpc-terraform-at-scale-6"
  auto_create_subnetworks = false
}

resource "google_compute_subnetwork" "tas_6_snet" {
  for_each = { for snet in var.snet_configs : snet.name => snet }

  name          = each.value.name
  ip_cidr_range = each.value.cidr
  region        = each.value.region
  network       = google_compute_network.tas_6_vpc.id
}

#
****************************************************************
********
```

```
#
////////////////////////////////////////////////////////////////////
////////
# variables.tf
#
////////////////////////////////////////////////////////////////////
////////
variable "snet_configs" {
  description = "A list of subnet information."
  type = list(object({
    name   = string
    cidr   = string
    region = string
  }))
}

#
*******************************************************************
********

#
////////////////////////////////////////////////////////////////////
////////
# terraform.tfvars
#
////////////////////////////////////////////////////////////////////
////////
snet_configs = [
  {
    name   = "tas-6-snet-1"
    cidr   = "10.0.0.0/24"
    region = "us-east1"
  },
  {
    name   = "tas-6-snet-2"
    cidr   = "10.0.1.0/24"
    region = "us-east4"
  },
  {
    name   = "tas-6-snet-3"
    cidr   = "10.0.2.0/24"
    region = "us-east5"
  },
...
```

Again, the goal is also to seamlessly preserve all running workloads, leaving all existing resources up and running, and leaving no existing

resources untracked. In the following procedure, the TF state object is dissected, emptying out all of the resources from the state and importing them into several new TF state objects, but the code repository is preserved (although heavily refactored):

1. *Create local module*

   Create a new subdirectory in the repository to host the configuration for a network-bundle module.

2. *Copy/paste a collection of resources*

   Copy over all the configuration blocks for a particular representative workload.

3. *Refactor module for flexibility*

   Refactor any repetitions of the same resource block type into a single block that leverages the `for_each` meta-argument (do the same with any `dynamic` blocks) and refactor any singular optional resource blocks to leverage the conditional form of the `count` meta-argument.

4. *Include module variable definitions*

   Expose all the fields that vary between workloads (the unique field values in the original construction of the code) as variables in the module. Assuming the original construction of the code already included these variables (albeit repeated, along with the resource blocks that utilize them), it may be effective to copy over (and refactor) the variables for the workload from step 2 (refactoring may see a set of individual `string` variables be re-encoded as a single variable typified as a list of strings).

5. *Construct outputs*

   Include all resources in outputs from the module, so they may be reported by certain TF commands.

6. *Refactor root variable definitions*

Reduce the input variable definitions for the root module to exactly those in the local module.

7. Refactor root variable assignment

Create a distinct *.tfvars* file for each workload, and place all of the relevant values for a particular workload in the corresponding file, refactoring the names and structure of the data to match the new variable structure.

8. Refactor root module

Replace the repeated collections of resource blocks with a single module block, referencing the new local module as its source; ensure that the module variables are properly wired from root module to the new local module.

9. Introduce backend key variation

Refactor the backend block to include a partial configuration for the `key`/`prefix` field.

10. Discover "from" addresses.

Run a `tf plan` on one of the new *.tfvars* and against the original backend and record the state addresses of all resources that would reportedly be removed (or replaced).

11. Discover "to" addresses

Run a `tf plan` command against all of the new *.tfvars* (and against the corresponding backend key/prefix) to determine the state addresses to which the resources should be imported within the new TF state objects.

12. Import resource info

Using the addresses reported in the previous step, import each resource, one by one, with the `tf import` command (again against the

corresponding backend key/prefix).

13. *Confirm imports*

    Run another `tf plan` command against each state backend configuration to confirm the `tf import` commands succeeded; no changes should be reported if the imports were successful.

14. *Run removals*

    Run `tf state rm` on each resource reported earlier (or, if you're extremely bold and confident, you can use `tf state rm $(tf state list)` against each of the old TF state objects to remove the entire root module and all resources — this should be done first with the `-dry-run` option).

> **WARNING**
>
> It's always a good idea to first run a `tf state rm` with the `-dry-run` option to preview the changes the command will attempt to make.

15. *Confirm removals*

    Run another set of `tf plan` commands to confirm the `tf state rm` commands succeeded; no changes should be reported if the removals were successful.

Cleaning up the duplicative code does not reduce the resource footprint or resulting TF state object. However, as suggested, it does reduce repetitive edits in cases where a new standard for a particular resource is adopted and must propagate throughout the codebase.

Scenario 3: merger/acquisition

A merger or acquisition can happen at any time to pretty much any company. While there is generally a great deal of time before deals are complete and activities have advanced to the point of needing to worry about integrating the two companies' codebases and infrastructure, it'll likely eventually come to pass that keeping the two separate is deemed untenable (especially if they're using the same CSP). After all the political jockeying concludes, and all components within a particular area are designated for consolidation, it's likely that one of the now redundant engineering teams (the ones that maintained these components) becomes the dominant, and eventually singular, team for that domain.

In a perfect world, none of the resources would need to be recreated. Because the resources are almost certainly associated with different accounts, there are two options to choose from: consolidate the resources into one of the accounts (the dominant one), recreating the ones from the other account or construct a second `provider` configuration block in the receiving/dominant project to manage the resources from the other account (this was explored in Chapter 4).

In the first option, at least some resources will need to move, consolidate, or otherwise be modified. This might be in the form of combining codebases and TF state objects for certain network components like subnetworks and firewall rules. This may manifest as a many-to-one procedure for codebase and TF state within a particular domain space. Depending on the type of cloud service, it may be possible to establish an equivalent, parallel resource configuration in the receiving account and to perform an atomic cutover operation that minimizes service disruption and avoids downtime.

> **TIP**
>
> Depending on the CSP, it may be possible to leave certain other components in place and to peer internal networks, which may further serve to minimize disruptions and downtime of this type of solution.

In terms of the second option, we'll still integrate the codebases and TF states in question, but we'll ensure each of the existing resources is preserved in its respective cloud provider. As mentioned, this will require a second `provider` configuration block in the receiving project to manage the other account. In the following procedure, assume you're part of the dominant team as described earlier and are responsible for incorporating the code and resources from a deprecated team into your existing codebase and TF state:

1. Specify provider for current

In all the resources and modules already present in the dominant project's root module, include the provider or providers configuration to target the dominant account.

2. Copy over the code

Copy/paste the relevant configuration into the receiving/dominant repository, exactly as it is in the deprecated source root module.

3. Specify provider for incoming

In all the incoming resources and modules now present in the root module, include the `provider` or `providers` configuration to target the incoming account.

4. Discover imports

Run a `tf plan` in the receiving/dominant repository to confirm where to import all resources.

5. Run imports

Using the addresses reported in the previous step, import each resource, one by one, using the `tf import` command.

6. Confirm imports

Run another `tf plan` to confirm the `tf import` commands succeeded; no changes should be reported if the imports were successful.

7. Discover removals

Run a `tf plan -destroy` in the deprecated repository, to confirm which resources to remove.

8. Run removals

Run `tf state rm` on each resource reported in the previous step — or for the bold, use `tf state rm $(terraform state list)` to remove the entire root module and all resources, as described before.

9. Confirm removal

Run another `tf plan` to confirm the `tf state rm` commands succeeded; no changes should be reported if the removals were successful.

Figure 6-8 is an example of the TF state tree resulting from the previous procedure.

> **NOTE**
>
> Constructing a second `provider` configuration block in the receiving/dominant project to manage the resources from another account doesn't ensure that the resulting TF state object is balanced. Although this may not be the most important aspect in terms of consolidating multiple accounts, it's important to be aware of this in case one account greatly dominates another.

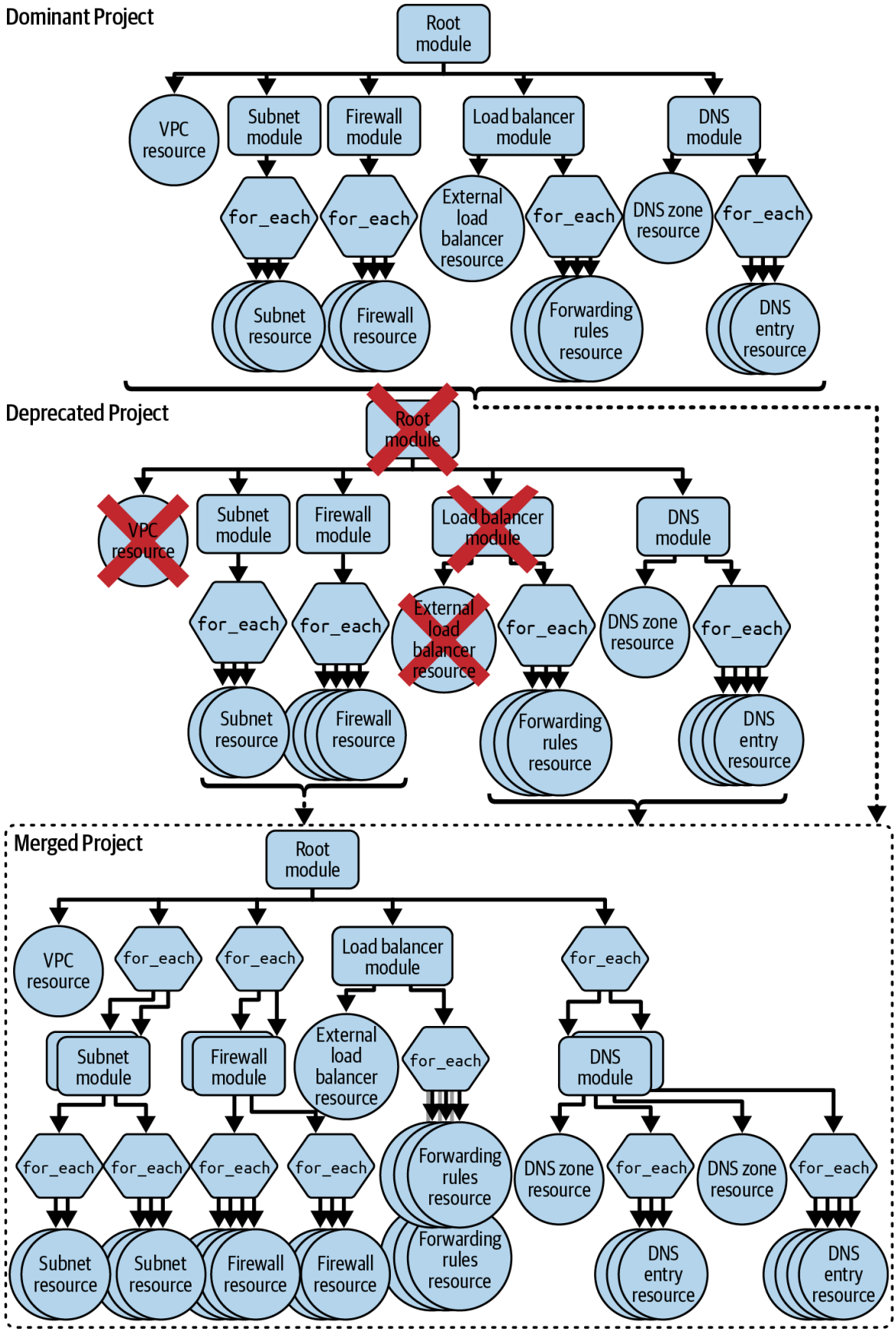

Dominant Project
Root module
VPC resource
Subnet module
Firewall module
Load balancer module
DNS module
for_each
for_each
External load balancer resource
for_each
DNS zone resource
for_each
Subnet resource
Firewall resource
Forwarding rules resource
DNS entry resource
Deprecated Project
Root module
VPC resource
Subnet module
Firewall module
Load balancer module
DNS module
for_each
for_each
External load balancer resource
for_each
DNS zone resource
for_each
Subnet resource
Firewall resource
Forwarding rules resource
DNS entry resource
Merged Project
Root module
VPC resource
for_each
for_each
Load balancer module
for_each
Subnet module
Firewall module
External load balancer resource
for_each
DNS module
for_each
for_each
for_each
for_each
for_each
for_each
DNS zone resource
for_each
DNS zone resource
for_each
Subnet resource
Subnet resource
Firewall resource
Firewall resource
Forwarding rules resource
Forwarding rules resource
DNS entry resource
DNS entry resource

Figure 6-8. Example result of merging two projects within the same domain, such as in the case of a merger or acquisition

Scenario 4: organizational change management

Another almost inevitable event that happens at any sufficiently large organization is the dreaded reorganization (reorg). There are many causes of reorgs, some more legitimate than others, but many bring along policy considerations such as new responsibilities. As we saw before, we can relocate portions of the codebase and TF state. This may be useful to redistribute the resources among repositories aligned to the new organization structure.

Suppose your massive multinational organization has invested heavily into a domain-oriented approach with centralized teams managing the vast majority of network components, security controls, and storage components. Your leadership had to choose between unsavory choices:

- Stretching the team across continents/schedules to provide round-the-clock vigilance
- Calling people up on nights and weekends, compromising peace of mind and also likely budget

Regardless of the result, you may lament the choice they made, but you also likely do not covet the responsibility of having to make such a decision. In the end, it seems to have worked, even if it has some rough edges and the occasional messy incident.

Now suppose the engineering leadership has chosen to reorient the department to more closely resemble the work on team topologies, with a few distinct domain-oriented platform teams supporting a catalog of capabilities intended to service several similarly composed application workload development teams, with a standard practice for facilitating the latter's use of the platform. Instead of a centralized team managing, say, all the firewall rules across the globe (with all the unwanted implications, as described), a (perhaps still centralized) lean team of the top network engineers will develop services that one or a few integration specialists will

leverage in support of more regionally oriented individuals who in turn will be expected to consume and make use of them.

As with "Scenario 1: organizational scaling", the outcome may be multiple new (or repurposed) repositories with distinct configuration encodings and segmented by one or more of the dimensions we saw earlier. However, this is likely a many-to-many transformation, with the resources initially structured and grouped one way across a collection of repositories and redistributed in new groups according to the new structural considerations, as shown in Figure 6-9.

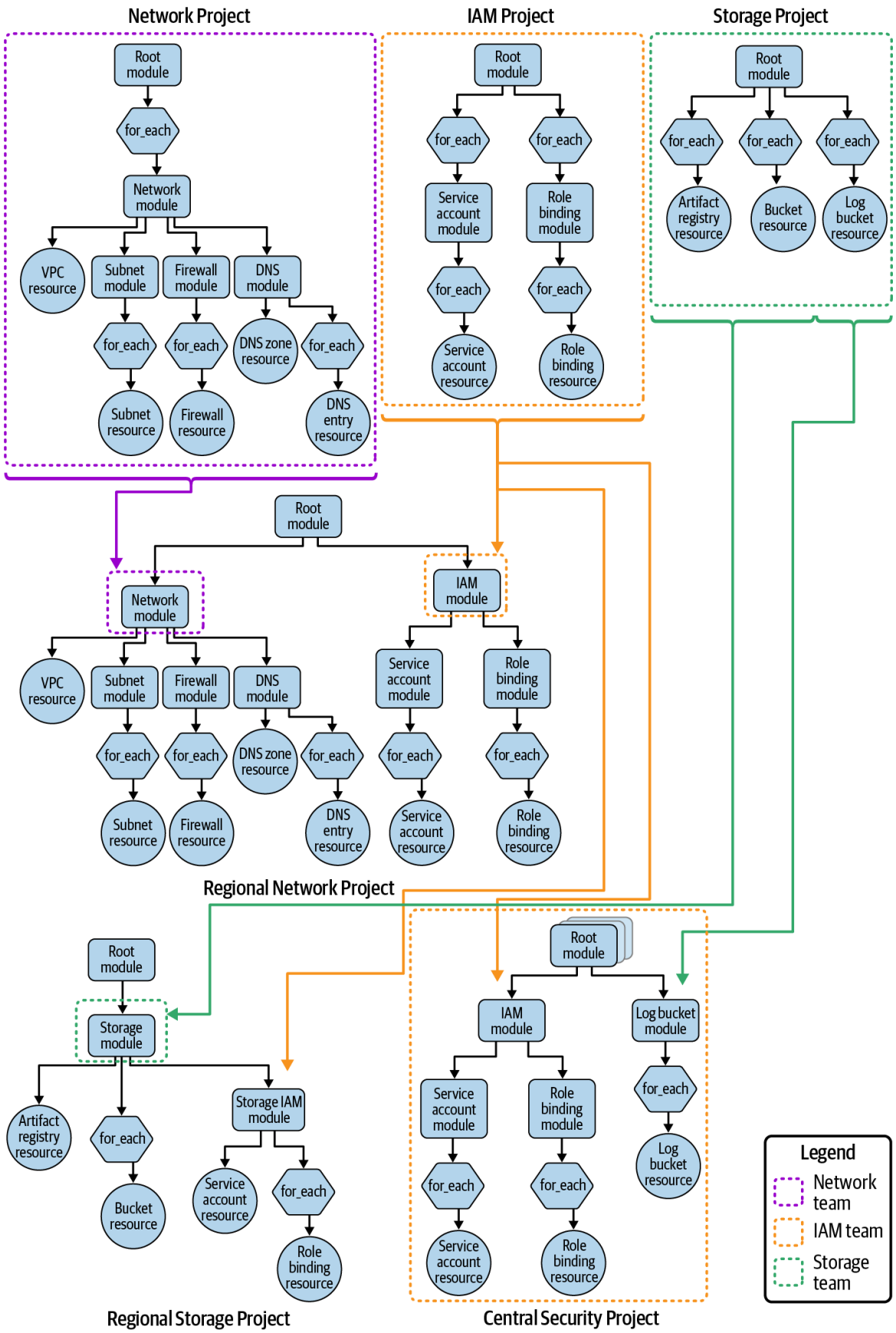

Network Project
IAM Project
Storage Project
Root module
for_each
Network module
VPC resource
Subnet module
Firewall module
DNS module
DNS zone resource
Subnet resource
Firewall resource
DNS entry resource
Service account module
Role binding module
Service account resource
Role binding resource
Artifact registry resource
Bucket resource
Log bucket resource
IAM module
Regional Network Project
Storage module
Storage IAM module
Regional Storage Project
Log bucket module
Central Security Project
Legend
Network team
IAM team
Storage team

Figure 6-9. Refactoring multiple projects to better match the new organization structure

Although it's not as simple as cleaving (or teasing out portions of) a single project into multiple pieces, if you want to preserve all of the existing resources as is, you can take a very similar approach to that from Scenario 1, the difference being you would consider all the existing/input TF state objects as a whole, running whichever of the TF operations (for example, `tf plan`) on each of them in sequence, rather than considering each one separately. This means the operator of this approach must be permitted to edit or administer all of the existing/input repositories, resources, and TF states.

> **TIP**
>
> It's once again highly recommended that whoever is affecting a large-scale refactoring of the codebase be an individual (or a very small team) with temporary permissions who is performing all actions through wildcard jobs.

Because infrastructure updates precipitated by organizational change management may be more prescribed and predefined, they have a higher likelihood of being implemented in a *big bang* format — all at once, rather than piecemeal. Similar to the steps described in "Scenario 1: organizational scaling", the following is a viable approach. In this procedure, however, assume the decision was made to reorganize all the code among completely new repositories/projects, and create (and populate) completely new TF state objects recorded in new backend locations. Assume also a policy requiring that the old repositories be archived, and the old TF state objects be cleaned up (that is, every resource is tracked by one and only one TF state object):

1. Create new repositories

These will house the reorganized code.

2. Copy over the code

Copy/paste the configuration into the new repositories, refactoring as necessary to ensure proper syntax and analogous result (that is, all the equivalent configurations are present, up to the state addresses).

> **TIP**
>
> When copying over the code into the new repositories, it may not be appropriate or possible to preserve the state address of each resource in the new state objects. For example, there may be a desire to bundle (or rebundle) certain resources when developing the new modules (optional step 3). It's highly recommended to track and record the mapping between the "from" TF state addresses and the "to" TF state addresses, for example, in an internally shared spreadsheet. This can ensure that no components become untracked when making the transition.

3. Construct new modules (optional)

Construct new modules that combine the redistributed resources as appropriate.

4. Discover imports

Run a `tf plan` in each of the new repositories, to confirm where to import the resources.

5. Run imports

Using the addresses reported in the previous step, import each resource, one by one, using the `tf import` command; do this across all new repositories.

6. Confirm imports

Run another `tf plan` across the new repositories to confirm the `tf import` commands succeeded; no changes should be reported if the imports were successful.

7. Discover removals

Run the `tf plan` commands in each of the old repositories to confirm which resources to remove from the old TF state objects.

8. Run removals

Run `tf state rm` on each resource reported in the previous step (or again, use `tf state rm $(terraform state list)` to remove the entire root module and all resources).

9. Confirm removal

Run another series of `tf plan` commands to confirm the `tf state rm` commands succeeded; in this case, because we preserved the previous organization of the TF code and simply copied over the configuration into the new repositories, all resources should be reported here if the removals were successful.

Summary

You now can identify inefficient, improper, or otherwise problematic behavior and structure regarding state objects. As your organization's needs and makeup evolve and your codebase and TF ecosystem works to catch up with it, you can foresee the possible issues that may arise and introduce necessary measures to ensure a viable path toward balance and efficiency. Even in cases where it may be deemed infeasible to make valuable changes, the impact can be fully articulated to decision-makers both in terms of the cost to change and the cost of inaction.

In this chapter, you learned the following about optimizing TF state:

- Concepts such as the state backend and state object and how they effectively comprise TF's memory
- How to remove, import, or transport all or portions of a TF state object

- What TF doesn't work well with, such as one-off operations and ClickOps
- Issues such as tightly coupled states and unbalanced states
- General recommendations such as keeping state operations atomic and cleaning up untracked resources or unused state objects
- Organizational recommendations around how to align your codebase in an approach that follows Conway's Law
- Common problems that may be familiar, such as consistently locked or broken states; massive, inefficient state objects; and states that break TF idempotence
- Common approaches to guide you toward effective maintenance of an enterprise TF ecosystem
- Common scenarios that many teams will encounter over time and how to rebalance TF state objects in their wake

In Chapter 7, you'll begin applying the knowledge that you've gained over the first half of the book to establish enterprise-scale, TF-backed infrastructure. You'll start with an overview of how to go from manually executed operations to automated IaC practices that will allow you to scale beyond a single operator or team. You'll also see how to utilize ephemeral environments and autonomous components to establish highly advanced delivery platforms with features such as a self-service request model or autohealing configurations. This will begin to turn the theory established in this and previous chapters into a practice that can be made to serve an organization of any size.

Chapter 7. Automating at Enterprise Scale

In the first half of this book, you learned how to develop modules in a maintainable, scalable, and efficient way. You reviewed several common and emergent practices, including step-by-step walkthroughs of how to proceed, and you evaluated scenarios where they're effective — or not. You recognize that even the best of practices have limitations or diminishing returns, and that documenting and developing policies around deviations and nuances is imperative. If your organization is sufficiently small (or if your specific purview is only a portion of the organization's overall infrastructure/platform ecosystem), you may have most of the tools you need to achieve your goals and expectations — in which case, you may be able to skip ahead to Chapter 10, which presents various validation, troubleshooting, and remediation techniques, until you're ready to expand the ecosystem (or your responsibility) beyond a single project. However, most TF practitioners do operate within a large organization, and even if you're presently not responsible for other portions of the ecosystem, it's valuable to see how the ecosystem as a whole fits together.

In this chapter, and for most of the remainder of the book, we'll shift our focus beyond individual projects and how they manifest more or less in isolation up to module compositions, to the interactions, boundaries, and dependencies — which may be explicit and fully automated, or implicit and largely manually procedural — of all the projects and modules maintained and leveraged by your organization. This will result in a shift away from low level and concrete visual aid constructs like code snippets and filesystem organization toward higher-level ones like diagrams and other illustrations.

In particular, this chapter will explore various ways in which to automate TF projects across an organization and at scale. This naturally assumes your

organization has more than one project/root module that it maintains, that there are logical, operational, or compliance boundaries separating them, and that there is some level of interdependency (however loose or directed) among some of them. If your organization is relatively large, and includes multiple lines of business, and all of its cloud resources are encoded in TF, this is a good assumption. If this is not your current experience, this chapter will help you identify the opportunities for significant reorganization, refactorings, or even reimaginings of your TF ecosystem. One of the most common blunders in IaC is failing to recognize when a traditional pattern no longer serves its purpose or scales effectively.

Because every automation platform has unique capabilities — be it HCP Terraform/TFE or another platform for use with OpenTofu — the choice of platform will have significant implications as to what is available or appropriate. Moreover, platform capabilities change over time with subsequent releases, and what is available (or not) at the time of writing may or may not be available in the same way months and years later. Where a capability is unique to a specific platform, it will be referenced as such; however, the intent is to focus on the capability or process itself, rather than the platform that exhibits it. The theory is that no one platform is guaranteed to have all possible features, but that all will — on a long enough timeline — have most of them, or analogous and reminiscent capacities, and enough of which to achieve significant gains over trial and error and traditionalism.

> **NOTE**
>
> Many of the topics presented here are theoretical and are expressed either as a hypothetical case study or as a thought experiment. In most cases, strict adherence to the guidance is neither necessary nor advised. The general goal of this chapter and the remainder of the book is to establish a framework and perspective with which to approach your specific organization's needs.

Going Beyond the Shell

Initially, most TF development is largely operated by individuals in a shell, possibly in a tutorial or sandboxed capacity. Even when you adopt a hosted platform like TFE, a SaaS like HCP Terraform, or a third-party or open source platform or SaaS with OpenTofu, the operations generally proceed in a way very similar to a shell script, except configured through YAML or a DSL, and perhaps with a level of parallelism of certain collections of operations. This typically means that automating your components is a matter of translating your shell operations and scripts into the language of your choice — or by simply running the same shell operations as a subprocess if your platform supports it, although this can have unforeseen implications if you take a custom shell script that worked on an engineer's workstation to an automation platform's shell, for example.

Automating the TF Process

The first step in adopting a platform that helps to automate the management of TF projects is to evaluate the available options. You would likely want to start by examining your existing processes and researching what capabilities each platform has to see which ones fully support your needs, which necessitate some refactoring or custom tooling to become supported, and which will require additional services to fill in gaps. Ideally, you would even take time to run a tutorial or take a course to get familiar with promising solutions before committing to one, perhaps even going so far as to develop a pilot pipeline to get a direct comparison with an existing well-established procedure. Of course, none of that will matter if your existing licensing agreements cause budget-focused decision-makers to encourage the use of a particular solution because it's included in the existing agreement, helps to satisfy consumption agreements, or precipitates in attractive discounts on existing services when it's enabled.

> **NOTE**
>
> GenAI tools like ChatGPT, Claude, or Gemini can greatly simplify the research step, giving you a shortlist of patterns to explore, or at least use as a starting point to orient yourself around successful patterns. These can often be further refined, whittling what might otherwise seem like an endless sea of potential solutions or portions thereof. How to most effectively interact with a language model is beyond the scope of this book, but often even suboptimal prompts can produce sufficiently representative results. However, remember, with any GenAI solution, the results should be thoroughly reviewed with a great deal of scrutiny by experienced members of your team before adopting any guidance as gospel.

Among more established solutions — or solutions from more established software publishers — feature support and UX is fairly comparable, so there's a low chance of choosing the wrong platform. For more esoteric products or relatively new open source solutions, there may be a higher degree of nuance in operation and gaps in typical capabilities. Either way, there is a significant risk of vendor lock-in to nearly any solution. Although platforms may support functionally analogous processes, they may support different ways of scoping and organizing contexts and operations. For example, the most basic scope in GitHub Actions (GHA) is an action, which packages together a set of scripts as a reusable operation; a *step* is the explicit configuration for an *action* and is itself packaged with other steps within a *job*; a *workflow* is a collection of jobs that typically run in parallel. In comparison, GitLab encodes shell scripts into its own concept of a *job*, which is grouped by a *stage* and run in parallel; a *pipeline* runs the stages in sequence. You'll notice that, although both support parallel operations, they perform them at different scopes, as depicted in Example 7-1. This often makes replatforming a solution tedious and nontrivial, possibly resulting in a complete rearchitecture of certain portions of the process.

Example 7-1. Comparing the same parallel and sequential steps in GitHub Actions and GitLab DSL

```
#-------------------------------------------------------------------
-----
# GHA (excerpt)
#-------------------------------------------------------------------
-----
```

```
on:
  push:
    branches: [main]

jobs:
  unit-test:
    runs-on: ubuntu-latest
    container:
      image: hashicorp/terraform:latest
    steps:
      - uses: actions/checkout@v6
      - run: terraform test

  lint:
    runs-on: ubuntu-latest
    container:
      image: hashicorp/terraform:latest
    steps:
      - uses: actions/checkout@v6
      - run: terraform validate

  plan:
    runs-on: ubuntu-latest
    container:
      image: hashicorp/terraform:latest
    needs: [unit-test, lint]
    steps:
      - uses: actions/checkout@v6
      - run: terraform plan -out=plan.tfplan
      - uses: actions/upload-artifact@v4
        with:
          name: plan-artifact
          path: plan.tfplan

  apply:
    runs-on: ubuntu-latest
    needs: build
    steps:
      - uses: actions/checkout@v6
      - run: ./deploy.sh
      - uses: actions/download-artifact@v4
        with:
          name: plan-artifact

#-------------------------------------------------------------------
-----
# GitLab
#-------------------------------------------------------------------
```

```
-----
stages:
  - test
  - build
  - deploy

lint:
  stage: test
  script:
    - npm run lint

unit_tests:
  stage: test
  script:
    - npm test

build_app:
  stage: build
  script:
    - npm run build

deploy_prod:
  stage: deploy
  script:
    - ./deploy.sh
```

Although, as stated, most automation platforms have roughly the same features, there are two features that should be available in whatever automation platform you select: access control and a DSL. Access control manages user permissions in the system, restricting or allowing behavior based on an authenticated user's access rights. This may be role-based — in the case of RBAC — which would be assigned to a user based on their role within the organization, or it could be based on other attributes such as time of day (work schedule), device used (workstation), and location (geofenced worksite), along with the user's role (job function) — this is commonly referred to as ABAC. Most modern systems support RBAC, which is appropriate for many use cases. If ABAC is not natively supported, it may be layered over an RBAC system, typically by using another tool that issues a temporary credential — which may be restricted based on attributes such as those listed previously. In this case, the ABAC credentials would have a

(static) role in the RBAC configuration, but the role is available to the user only for the lifespan of their (temporary) credential.

A DSL, on the other hand, allows for a pipeline or workflow to be encoded as software files. This means they can be stored in source control, such as Git, and managed just like application code. In particular, changes to the pipeline/workflow procedure may be reviewed before they're introduced, which, in theory, will improve the quality of the included procedure. One common feature of many DSLs is the ability to compose procedures, either by triggering other workflows/pipelines as subprocesses or by directly including the steps of the other, child procedure inline with the steps of the parent procedure. Although this isn't a make or break feature, it can greatly accelerate the development of new workflows or pipelines by encapsulating common functionality, such as invoking a static code analysis task or triggering integration tests. It also better adheres to the software design principle of DRY, which reduces both the chance of making a mistake when updating an otherwise repeated phrase and the total lines of code that are necessary to maintain the entire IaC management process. Typically, every platform's DSL is different, which is likely intentional; however, because they all operate in much the same way, they usually will have many of the same features expressed in ways that resemble each other. For example, most automation platforms expose a way to run arbitrary shell operations, with the operation itself appearing the same way in each platform, as was shown in Example 7-1, previously.

> **NOTE**
>
> Porting from one platform to another is another place where a tool like ChatGPT, Claude, or Cursor may be appropriate. For example, transcribing a GitHub Actions workflow to a GitLab pipeline is relatively straightforward, up to syntax and formatting, something modern language models are more than capable of handling. The reverse can be somewhat more challenging, depending on how the pipeline is constructed, given the change from a sequence of parallel tasks to a collection of sequential procedures all running — by default — in parallel. Here, too, a language model may be an effective tool for making suggested changes or full rewrite solutions that are analogous to the current working system. However, this would almost certainly require significant scrutiny by a highly skilled engineer or architect to ensure the order of operations are appropriate in the new platform. Skilled oversight is recommended for any AI-generated configuration.

At the end of the day, vendor lock-in is almost inevitable at some level. Often, if you're really concerned about lock-in, the best you can do is to use as straightforward a set of operations as possible, and package as many operations as effectively possible to as low a level as possible. Sometimes the best move is to simply wrap as many operations, and as much custom logic, as possible into shell scripts that may be called with a single-line invocation. An example of calling a single shell script, versus a sequence of commands, from within a GitHub Actions script can be found in Example 7-2.

Example 7-2. Before and after extracting a sequence of commands into a shell script

```
#-------------------------------------------------------------------
-----
# Before
#-------------------------------------------------------------------
-----
jobs:
  build:
    runs-on: ubuntu-latest
    container:
      image: hashicorp/terraform:latest

    steps:
      - name: Checkout code
        uses: actions/checkout@v4
```

```
      - name: Install dependencies
        run: terraform init

      - name: Run tests
        run: terraform test

      - name: Construct plan file
        run: terraform plan -out=plan.tfplan

#-------------------------------------------------------------------
-----
# After
#-------------------------------------------------------------------
-----
jobs:
  build:
    runs-on: ubuntu-latest
    container:
      image: hashicorp/terraform:latest

    steps:
      - name: Checkout code
        uses: actions/checkout@v4

      - name: Prepare infrastructure
        run: ./prepare.sh
```

To make the solution even more vendor agnostic, your solution may even support container technology. As we'll see in the next sections, this better insulates the custom operation execution using Linux capabilities such as *namespaces* (which isolates system resources for groups of processes) and *cgroups* (used to organize processes into hierarchical groups) while allowing full specification of the available OS binaries and libraries, and the processes to be carried out. If something is left unsupported in the context of the platform you're using, you can include support in the container's image, leaving the platform's capabilities undisturbed. Likewise, any undesirable custom behavior exhibited by the platform may be totally avoided by running within a container. Once your operations are encoded in a container's image, it may be migrated, as is, to any other platform that also supports containers, thereby unlocking the solution from the platform. An example of embedding the same shell script from before inside a container

image, and then referenced from a GHA script, can be found in Example 7-3.

Example 7-3. Embed a script into a container image and reference from a GitHub Actions workflow

```
#-------------------------------------------------------------------
-----
# Before
#-------------------------------------------------------------------
-----
jobs:
  build:
    runs-on: ubuntu-latest
    container:
      image: hashicorp/terraform:latest

    steps:
      - name: Checkout code
        uses: actions/checkout@v4

      - name: Prepare infrastructure
        run: prepare.sh

#-------------------------------------------------------------------
-----
# Dockerfile for after
#-------------------------------------------------------------------
-----
FROM ubuntu:22.04

# Avoid interactive prompts during package install
ENV DEBIAN_FRONTEND=noninteractive

# Install base dependencies
RUN apt-get update && \
    apt-get install -y \
      ca-certificates \
      git \
      bash && \
    rm -rf /var/lib/apt/lists/*

# Copy embedded script into the image
COPY prepare.sh /usr/local/bin/prepare.sh

# Ensure it is executable
RUN chmod +x /usr/local/bin/prepare.sh
```

```
# Default command (will be overridden by the workflow)
ENTRYPOINT ["/usr/local/bin/prepare.sh"]

#-----------------------------------------------------------------
-----
# After
#-----------------------------------------------------------------
-----
jobs:
  build:
    runs-on: ubuntu-latest

    container:
      image: hashicorp/terraform:latest

    steps:
      - name: Checkout code
        uses: actions/checkout@v4

      - name: Run preparation (from container entrypoint)
        run: prepare.sh
```

Connecting Pipelines

Beyond simply connecting the outputs of one module to the inputs of another, you may want to kick off other processes. For example, once resources are instantiated, you may want to run validation procedures to ensure the instances are behaving properly. In the case of compute services such as AWS Elastic Compute Cloud (EC2) or GCP Google Compute Engine (GCE), you may not be able to bake all of your dependencies into a machine image and need to provision certain application binaries immediately after the instances are ready. Although most of the relevant providers implement a way to run scripts upon a compute instance's creation or restart, it's not always the most dependable solution; it's most likely better to implement the command or procedure outside of the TF script as an immediately following command.

Another common process you may want to include in an infrastructure automation pipeline is to send confirmation, alert, and update notifications through various communication channels. This might be in the form of an

email to a distribution list, a message in your favorite communication platform, or an SMS text blasted out to all stakeholders. Finally, you may need to publish status messages to state tracking pub/sub service, such as Apache Kafka, which may be constructed such that messages may be reviewed or even replayed for a high-degree of auditability.

> **NOTE**
>
> A pub/sub solution may also be implemented to pass information between automation pipelines, steps, actions, procedures, and so on. An operation that applies TF configuration may publish a message to the pub/sub topic that contains the output of the applied module. A downstream pipeline or procedure may be triggered by subscribing to the same topic and reading the information out of the published message.

Although not all platforms and SaaS solutions are created equal, many of them implement these operations natively to some degree, while most others support integration points to various other third-party services that could be configured to cover the gaps. For example, some platforms implement additional services as platform plug-ins, fully integrating them into a common experience and seamless access control. Other platforms support integrations through the use of webhooks, which typically use REST requests to trigger commands and communicate results, remotely. This would generally require managing authentication between the core platform and the third-party systems, and may also require separate configuration be performed in each system to ensure proper behavior and safe operation. Moreover, although either type of platform has its merits and may produce functionally equivalent configurations, the plug-in solution will also generally expose more fine-grained feedback and is able to share this feedback more immediately, since all the components are running within the same runtime context.

Wildcard Jobs

In Chapter 6, wildcard jobs were recommended to run certain ad hoc operations in a protected way, such as when manipulating the TF state for a

particular TF project. This is a special operation that would be established in an automation platform and allows flexible command execution to enable shell-like access with the platform's service identity. It can be a single, highly flexible job or multiple, fit-to-purpose jobs.

A wildcard job can be constructed so that it exposes only certain commands and arguments — maybe as dropdowns or other form fields, and possibly preselected with sensible defaults — perhaps even allowing freetext predicates. The context within which the job runs should have access to only a specific set of software packages, either in terms of what's installed on the available OS or in terms of the effective OS user permissions.

It might even support running an uploaded shell script in cases where the necessary operation is sufficiently complex; however, this should be done only with adequate guardrails in place to ensure that no unauthorized access happens or disallowed operations occur.

Because of how sensitive this job would be in the wrong hands, don't allow persistent access to it for anyone permanently, but rather create a mechanism for obtaining such permissions on a case-by-case and temporary basis. Various identity governance platforms exist that allow permissions to be requested, approved, and assigned. If multiple fit-to-purpose wildcard jobs are constructed, they should each be associated with unique policies. Likewise, a report should be generated and sent to various stakeholders any time such a job is used. It should also be reviewed by the rest of the team in a group setting, possibly in a normal cadence meeting, as part of your compliance discipline. Figure 7-1 depicts an example solution.

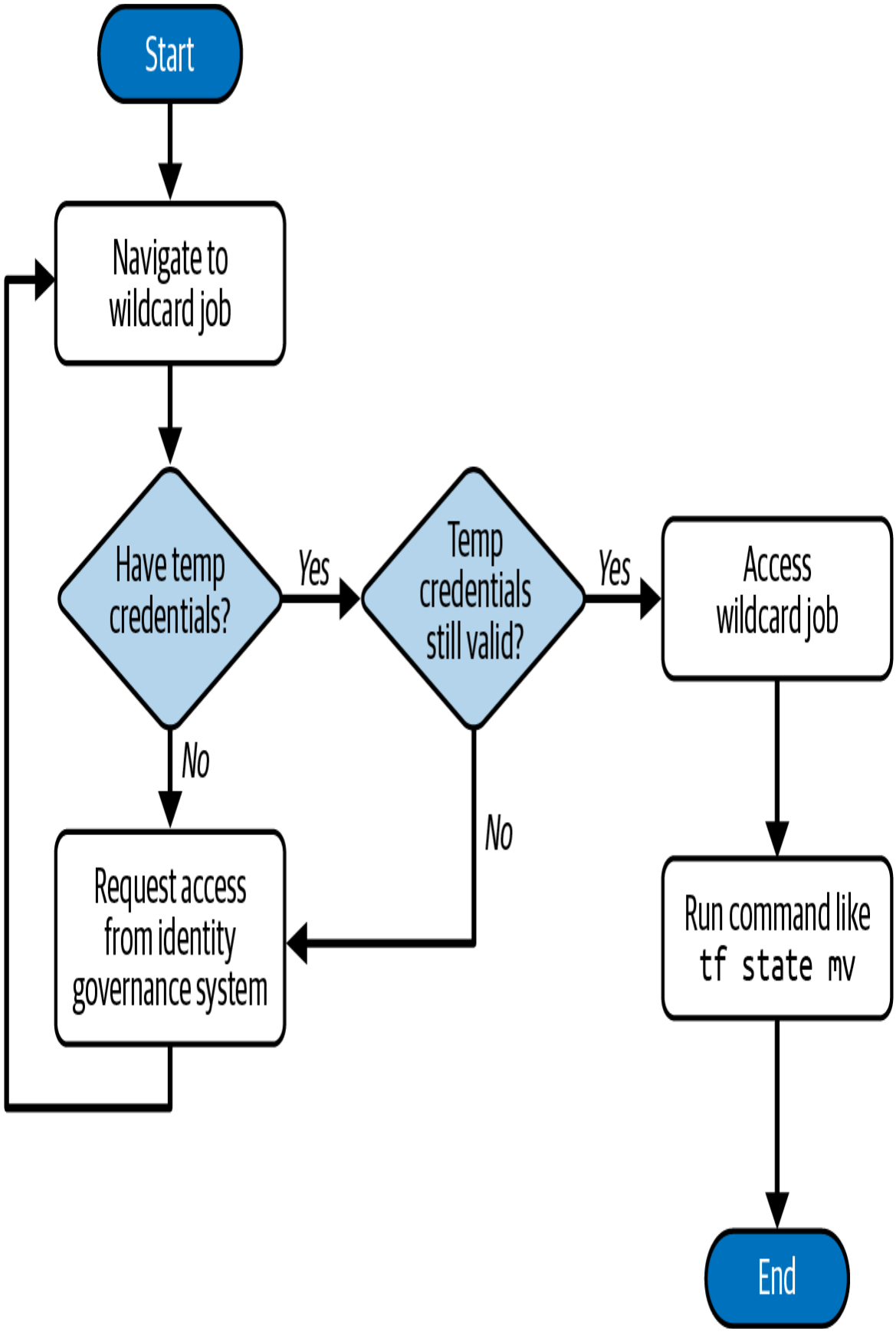

Start
Navigate to wildcard job
Have temp credentials?
Yes
Temp credentials still valid?
Yes
Access wildcard job
No
No
Request access from identity governance system
Run command like tf state mv
End

Figure 7-1. Example wildcard job solution flow diagram

> **NOTE**
>
> Wildcard jobs are really only possible in general purpose automation platforms like GHA or GitLab and are not a viable solution when using TFE or HCP Terraform. In such cases, it may be feasible to establish wildcard jobs in a separate platform specifically for this purpose. This can even simplify the management of permissions by applying them to the platform as a whole, rather than microsegmenting the permissions by job.

Context Is Everything

Many modern solutions run operations within the context of a container, the image for which may even be exposed to users through specific input fields. In these platforms, it's often possible to set a container image at the pipeline or step level, and possibly even as a default across a collection of pipelines or throughout the platform. This enables the construction of custom container images, which may be overly advanced in most cases but would allow for custom dependencies and behavior. In particular, custom-built container images may be the only way to overcome otherwise unsupported tooling or logic. As shown earlier, the ability to run operations within the context of a container has the additional benefit of reducing vendor lock-in by reducing platform-specific nuances such as operational dependencies and context, greatly improving the trust that operations will behave in much the same way regardless of underlying automation platform.

> **NOTE**
>
> Containers may be best thought of as a combination of a template (including a top-level command-line executable statement), an identity (like a Linux "user"), a portion of the filesystem, and a dedicated slice of compute resources — for example CPU, RAM, or disk. The identity runs the included executable statement and crucially believes its fragments of the filesystem and compute resources are all encompassing — that its portion of the filesystem is located at the root directory and its resource allotment is all that is functionally available (so it doesn't try to compete with other containers). Although they're often launched atop VMs, they're not VMs — and in fact need not be virtualized at all. Also, unlike VMs, they share, for example, the Linux kernel of the underlying VM or bare metal server.

With an automation platform container, the command-line executable statement run by the container — the entry point — would likely range in specificity. On one end, at the most generic, it is itself a shell command that expects an input value and evaluates it within the context of the container. This allows for a great deal of flexibility in the YAML or DSL configuration, which may be attractive early on, especially if you're evaluating multiple tools or techniques and the procedures are not yet stable. In this case, the main goal is to ensure the correct dependencies are present in the container, either by leveraging a community image that already includes them or by starting with a sufficiently bare base image, such as an Alpine Linux or a "slim" version of another distribution, and layering on the required libraries and binaries.

On the other end of the spectrum, at greater degrees of specificity, the entry point is likely a structured shell script that is constructed from input elements, as shown in Example 7-4. This is ideal for highly controlled environments, providing strict boundaries and enabling guardrails to guide end users into valid constructions of configuration. This might be a foundational element for highly advanced implementations of automated infrastructure, as it allows for highly customized dependencies, resource allocation, and behavior. It also can improve auditability as containers would generate unique system logs and may even be replayed to evaluate functionality.

Example 7-4. Passing environment variables into the entrypoint script

```
#
////////////////////////////////////////////////////////////////////
////////
# workflow.yml
#
////////////////////////////////////////////////////////////////////
////////
jobs:
  build:
    runs-on: ubuntu-latest

    container:
      image: hashicorp/terraform:latest

    env:
      # These get passed to the container and can be read
      # by the script used in the entrypoint
      JOB_ENV: "dev"
      MAKE_PUBLIC: "false"

    steps:
      - name: Run commands in the container
        run: echo "This runs inside the custom container"

#
********************************************************************
********

#
////////////////////////////////////////////////////////////////////
////////
# entrypoint.sh
#
////////////////////////////////////////////////////////////////////
////////
#!/usr/bin/env bash
set -euo pipefail

JOB_ENV="${JOB_ENV:-dev}"
MAKE_PUBLIC="${MAKE_PUBLIC:-false}"

case "$JOB_ENV" in
  dev)
    export TF_VAR_environment="dev"
    export TF_VAR_network_tier="internal"
    ;;
```

```
  prod)
    export TF_VAR_environment="prod"
    export TF_VAR_network_tier="restricted"
    ;;
  *)
    echo "ERROR: Unknown JOB_ENV '$JOB_ENV'"
    exit 1
    ;;
esac

if [[ "$MAKE_PUBLIC" == "true" ]]; then
  export TF_VAR_enable_public_networking="true"
else
  export TF_VAR_enable_public_networking="false"
fi

terraform init
terraform apply -auto-approve

echo "Provisioning complete."
```

Instead of constructing a more or less purely procedural solution, you can introduce a more sophisticated sequence of events by including custom logic with a more traditional programming language like Python or Go. This gives you greater control over the construction of inputs necessary to operate the modules and in interpreting the results of the operation. For example, as suggested in Chapter 3, developing application code to drive TF projects allows complicated logic required to marshal and unmarshal the module inputs and outputs to be abstracted outside of the module itself. This allows for greater level of consistency and governance associated with a traditional software development approach, even enabling true unit testing that further elevates the consistency and determinism of a module's use. In this case, the app code's invocation statement becomes the entry point for the container image, ensuring it's executed at the top level of the container.

Livestock, Not Pets

One of the biggest lessons that it takes many teams, and especially the members of leadership, a long time to come to terms with is the ephemeral nature of cloud services and cloud native resources. Compute instances —

ahem or even entire regions — will go down, and when they do, you want to recover from them quickly and gracefully. Unlike resources hosted on corporate premises, where some poor soul can be compelled to find, restart, and remediate the errant server, an instance that goes down in the cloud is possibly down for good. While this may seem like a bad thing on the surface, it has the fortunate implication that resources can (and should) be treated as both immutable and disposable. Once this principle is understood and embraced by all members of the organization, architecture and solution designs that would otherwise seem unreasonable or absurd become sensible if not almost obvious.

The common technology colloquialism for this (or a variation thereof) is to *treat the resources as livestock and not as pets*. The idea behind the phrase is that livestock is, by definition, expendable, whereas pets are treated with care and consideration. The other insinuation is that livestock is numerous and undistinguished, whereas pets are much fewer, unique, and special, with specific recognizable names. This means that less fixed and special naming conventions are necessary, and more and richer metadata is required to identify components running in cloud environments. It also means that replications and replacements become ordinary and regular and become much more feasible than SSHing into a failing instance and poking it until you figure out what's wrong, and then begin to resolve it.

Luckily, because the cloud is also driven by APIs, cloud resources may be created and destroyed at will and often billed through consumption rather than as a capital expense that quickly depreciates. This means that your choices in implementation and configuration depend on your functional needs and nonfunctional guarantees, rather than what's sitting out on the floor. You aren't physically limited to the regions in which you currently operate. You aren't technologically constrained to a single OS, chipset, or architecture pattern. You can quickly test out a theory on ephemeral resources, without the need for a significant de-risking exercise necessary to justify the use of corporate datacenter resources that were likely budgeted for "mission critical" workloads. And you don't need to repurpose old hardware just because you bought it three years ago and your accountants

won't let you move on just yet; terminate the instances and use whatever resource configurations you actually need.

The same goes for defective components: instead of attempting a remediation process, simply destroy it and recreate with a good version. This is a cornerstone of the *roll forward* approach that many successful contemporary development teams take: rather than trying to *roll back* bad changes to a prior state, we deploy a fix that becomes the next state. An even better solution may initially cordon off the defective component, while still creating a replacement, and attempt an initial investigation of the flaw before fully destroying it. Ideally, a majority of defects are caught before a new version is even fully deployed, as part of a *blue/green* or *canary deployment*, whereby a new version is deployed alongside the old and validated with live, or mirrored, workloads and traffic.

This way of working can be uncomfortable at first. You might hear frantic questions such as "What do you mean you aren't going to fix prod, which is losing us $X by the minute, but instead you're going to create *additional* instances?!" or strange statements like "When should we schedule the 4 hours of downtime we require for this update?" which is merely a relic of a bygone era of having a set inventory and having to fix things in place. One way to mitigate this early on is by providing estimates on how efficiently you can make such a change. Eventually, your guardrails should be robust enough that any such upgrade or repair event is almost indistinguishable from a newly developed feature. We love our pets and it's truly painful to let them go. Moving on from expensive hardware, albeit a far less traumatic experience, may still evoke strong feelings from the accounting department and anyone else who is subject to amortized technology costs. But deleting cloud resources is and should be painless, casual, and ordinary.

Coordinating Multiple States

At this point, your organization's corpus of IaC has likely grown beyond a single TF state. You may be using the same code with different inputs, maintaining parallel instances of similar infrastructure stacks, or you may

be leveraging completely different modules that serve different portions of your infrastructure, possibly with dependencies between modules. Probably you're doing some combination, both incorporating modules and reusing modules at the root level. Whatever your setup, it will require a great deal of coordination across the various states. There are surely plenty of ways to do this, and there are likely merits to most, if not all, of them. The following section will review several options and their benefits and drawbacks.

Tracking Dependencies and State

Arguably, the worst way to handle tracking dependencies is to effectively not handle it and simply react to problems as they arise. Unfortunately, this is not a rare approach. It can arise naturally along with an evolving project, where additional process becomes necessary, but has not yet been implemented. Often, the decision to break a single project into two or more will include forgotten dependency relationships that don't become problematic until after the split has been completed. It could be minutes or it could be months, but eventually these dependency relationships will surface. Initially, they may come as an unwelcome surprise, but as issues recur, it will become an instinctive convention to consider the dependency any time it could have an impact. This then becomes a cost-benefit analysis: how does the development cost of implementing an automated process compare to the recurring cost of reacting to an issue? Assuming the problem is infrequent and the fix is well known, it may not be worth it to add the extra overhead. If it becomes a consistent ritual, and especially if it becomes the de facto responsibility of a single senior team member, it's likely well worth the development costs to mitigate the recurring operational cost.

If the cost of automating any such processes is indeed deemed too high, it's certainly worth tracking interproject dependencies in some way. This is much easier said than done, and even fairly sophisticated tools generally will require a skilled and thorough operator to effectively conduct a complete investigation of all dependencies across all projects. Rather than relying on tooling to elucidate them, it's likely beneficial to track

dependencies proactively, documenting them in some fashion. Doing so in a digital spreadsheet, although cumbersome and error-prone, may be the best available option to do this completely. It would either require early investment in an extendable schema or the understanding that making updates to the schema must be done with the utmost care, lest a change compromises the integrity of the entire tracking system.

Another option for tracking dependencies is to use a database solution that supports data normalization, such as a relational database. Such solutions will likely be easier to modify later, given the high degree of consistency and integrity to which modern database software strives to achieve: modifications to the dependency schema would be introduced through a structured language like SQL that would be validated for consistency before it may be applied to a running database. Utilizing a database this way would likely still be highly tedious if done manually, but it would be a great component to an automated tracking system. Thus, introducing it early on, even before processes are fully operationalized, can be a valuable stepping stone to the ideal solution.

Domino Effect

Beyond simply tracking the dependencies, they must also be addressed when modifications are made to upstream projects. As more project dependencies are introduced, these dependencies may require cascading updates to downstream projects. Depending on how many projects are related this way, this will add significant time to any update, as the downstream dependencies must be run sequentially, with an upstream project fully completing its own `tf apply` step before any downstream TF operations may be initiated. Doing this manually may not be especially unwieldy at first, but at a sufficient level of complexity and completeness, it often becomes necessary to automate large portions of the infrastructure. A successful update to one project automatically triggers an update to all downstream projects in a cascade of automation pipelines, each running its own TF operations, that ripples through your infrastructure until all TF projects are sure to be properly up-to-date. Although developing such

automation won't significantly out-perform a manual version, from the perspective of end-to-end running time, it will significantly reduce normal problems associated with manual processes: typos, forgotten or out-of-order steps, and other similar oversights. In other words, it will improve consistency and reduce unnecessary mistakes.

Unless you're uncommonly lucky, the changes that cascade throughout your infrastructure will not always simply require a rerun of the downstream projects in order. There will likely be some level of information that must be passed from an updated upstream project to a downstream one. Even if you've spent a great deal of time constructing your TF projects to be fully self-consistent (even with interproject dependencies) with data source elements intelligently mapping upstream components, there will still likely sometimes be information that cannot simply be detected, such as when infrastructure is migrated or replicated to a new region or container resource. Leveraging output values from the upstream project is likely to be a valuable component to a solution that accounts for this information sharing, but reading these values and interpreting them as inputs to a downstream project will very likely require engineering effort.

The way you construct automated triggers that manage the dependencies across projects depends on your choice of automation platform. In many cases, the platform will provide some native way to trigger one workflow or pipeline from another, such as internal keywords or commands. Hopefully this allows for information to be passed along with the trigger, such as GitHub Actions' reusable workflows or using GitLab's `trigger` keyword, as shown in Example 7-5. In the absence of a built-in way to kick off other pipelines, most platforms support a webhook approach for initiating an automated process, which typically allows for arbitrary key-value pairs to be provided and can map to the input values of a downstream TF project. This can even allow for multiple platforms to be integrated, if so desired, such as to shift certain operations into CSP-managed solutions. This might have significant access and security benefits and can even have cost implications if such solutions mitigate the need for elevated subscription tiers in third-party products you license.

Example 7-5. Triggering one pipeline from another in GitLab using the `trigger` *keyword*

```
#
////////////////////////////////////////////////////////////////////
////////
# parent.yml
#
////////////////////////////////////////////////////////////////////
////////
stages:
  - build
  - deploy

plan_infra:
  stage: build
  image: hashicorp/terraform:latest
  script:
    - terraform init
    - terraform plan -out=plan.tfplan
  artifacts:
    paths:
      - plan.tfplan
    expire_in: 1 hour

trigger_apply:
  stage: deploy
  trigger:
    project: platform/deploy-pipeline
    branch: main
    strategy: depend
  variables:
    PARENT_ID: $CI_PIPELINE_ID

#
****************************************************************
********

#
////////////////////////////////////////////////////////////////////
////////
# child.yml
#
////////////////////////////////////////////////////////////////////
////////
stages:
  - deploy
```

```
apply_infra:
  stage: deploy
  image: hashicorp/terraform:latest
  script:
    - terraform apply plan.tfplan
  needs:
    - pipeline: "$PARENT_ID"
      job: plan_infra
```

State of States

Along with well maintained and sufficiently precise dependency tracking, it can be highly valuable to maintain a higher-order account of the various TF projects, a *state of states*. This would increase the visibility and integrity of your organization's overall infrastructure ecosystem by giving precise information about the various configurations at a point in time. Coupled with a robust module versioning system, this would allow for very fine-grained versioning across your entire organization, which can help in cases where you discover things have gone wrong and it's unclear when exactly the problem was introduced.

These state of state versions can be tracked again in various data receptacles such as a digital spreadsheet or database. If you're using Git to manage your TF module code, you should create tagged commits to represent versions that, based on all available observations, produce stable infrastructure. These tagged commits can be linked to directly in the tracking solution, providing a clear provenance of each state at a point in time. This is extremely valuable in running audit exercises, such as to determine length of exposure, confirm cost attribution, and measure the impact of various changes.

> **NOTE**
>
> This is not to say a version or tagged commit will always be valid. CSP services evolve, with some features or entire offerings becoming less relevant, obsolete, and eventually removed from support. As such, most likely all versions of your TF configuration have a shelf life, and all of them are more or less unique. You should aggressively consider old versions largely invalid, and only ever consider the immediately previous stable version as *potentially* valid, using older versions purely as an audit trail.

Another element of the state tracking that could be valuable to an audit exercise is the results of running a pipeline or workflow. Assuming your solution of choice remembers old runs, the output will typically have a unique link which you can include in your tracking solution. This would naturally match up with system access logs, but it would tie these — otherwise essentially anonymous — actions to specific versions of the TF configuration, such as a tagged Git commit, which should have a specific author. This even provides a lower risk solution to system logs — which are inherently risky, due to their inclusion of sensitive organization and identity information — as how these are generated can typically be modified, and their access can often be controlled in a more fine-grained fashion. This can potentially allow for more regular auditing practices, and even enables self-audits by individual teams on the components for which they are responsible. Allowing such high-fidelity yet fully governed control over system logs is as intricate as it is precarious.

Ephemeral Environments and Environments on Demand

Ephemeral environments, or *environments on demand*, are fully self-contained copies of your infrastructure (or portions thereof) that are constructed and that may be eliminated at will and without friction. These may be full renderings of a particular stage of your software delivery lifecycle, such as a staging or even production environment, and allow for safe investigation or manipulation of the environment's configuration

without compromising mission-critical operations such as development activities, release procedures, or production runtime.

Beyond simply building an environment whenever desired, ephemeral environments require the environment to be easily torn down. Not only is this important for cost recovery, it's also an important security consideration, given that no publicly accessible service is perfectly secure and thus any environment presents some level of risk simply by existing. Thus removing an environment that no longer serves a purpose — or that goes dormant for significant amounts of time — is part of any high-quality security posture. Certainly, you don't want to tear down an environment only to realize immediately that you need it again, especially if the teardown takes so much time that the realization occurs before the teardown completes. Therefore, teardown should be appropriately performant so it never proves to be a bottleneck itself. That being said, you should also be constructing your ephemeral environments such that no environment gets *recreated*, but each newly established ephemeral environment is totally unique. This follows a roll forward approach to environment construction and control.

In addition to on-demand teardown, implementing a limited duration to ephemeral environments is another valuable feature you should consider for your TF setup. This guarantees no environment that isn't mission-critical stays too far past its usefulness. This can manifest as a uniform or fixed duration, at the end of which triggers the teardown operation or as a collective policy that automatically tears down ephemeral environments, for example, at the end of the week. A highly mature solution would also allow for some level of overriding or extending the expiration, perhaps with additional oversight or governance to safeguard against abuse. Any solution should include some way to tag or label each resource with its most up-to-date expiry date. This information is relevant to all stakeholders who have the appropriate level of read access rights in your environment; alternatively, this information can be exposed through a dashboard or console that is exposed in whatever way you prefer.

Once a collection of infrastructure can be established and demolished effortlessly, we can consider this collection *ephemeral*. It lasts for only as long as you want it to last, and when it's removed, it's removed completely, surgically, and without friction, as if it never even existed (if you don't look at system logs or other metadata that gets aggregated). This keeps your overall infrastructure lean and manageable and likewise avoids sprawl and abandoned services by ensuring stale or unused environments are pruned out. Moreover, this allows for significant flexibility. One, it removes the excuse that you don't have a place to put something experimental, unfamiliar, or simply new since now you can just create a new, isolated environment to host whatever you want. Two, it can lower the cost of even most custom configurations by reducing it to implementing its difference with a standard configuration.

Ephemeral environments can — and ideally, would — be sandboxed environments that you may use for experimentation with unfamiliar features or services and for piloting prototype solutions that may be risky if exposed to other environments or their data. Another use case is production-like environments that may be used for load testing, where heavy user activity is simulated and performance or availability is measured, or chaos testing, whereby components are consciously corrupted, data is deliberately deleted or manipulated, or dangerous traffic is intentionally injected into the network. These environments can be as similar to production as you want them to be, typically up to a point in time, and, depending on your infrastructure architecture, could have a significant body of services that are billed by the minute or hour, so having them exist only as long as the test itself can have significant cost implications.

In addition to environments that do not coexist with your software delivery environments, environments on demand may be used to introduce new production environments for environment-level *green/blue* or *canary* deployments, where an entirely new application layer for the production environment is provisioned and connected to the live production data and traffic is redirected to the new environment in either a full switch or graduated switchover fashion. If your environments are both ephemeral and

constructed from layers of root modules, you can likely even use this approach for certain A/B testing exercises. It's worth noting that these solutions might require very sophisticated control over networking components such as DNS, firewall rules, and network routes. If cloud network constructs such as peered VPC networks or GCP's Shared VPC are utilized, the relevant limits should be considered in any such solution. You don't want your release process to be shut down because you didn't account for subnet cardinality or overlapping CIDR ranges. As old environments are replaced with new ones, they can be torn down altogether cleanly, rather than having to surgically remove them, posing the risk of removing the wrong thing. Similarly, new environments that are invalidated or otherwise rejected can be cleaned up without ever affecting or coexisting among the valid components.

For more information on ephemeral environments and environments on demand, check out *https://ephemeralenvironments.io/features/on-demand*.

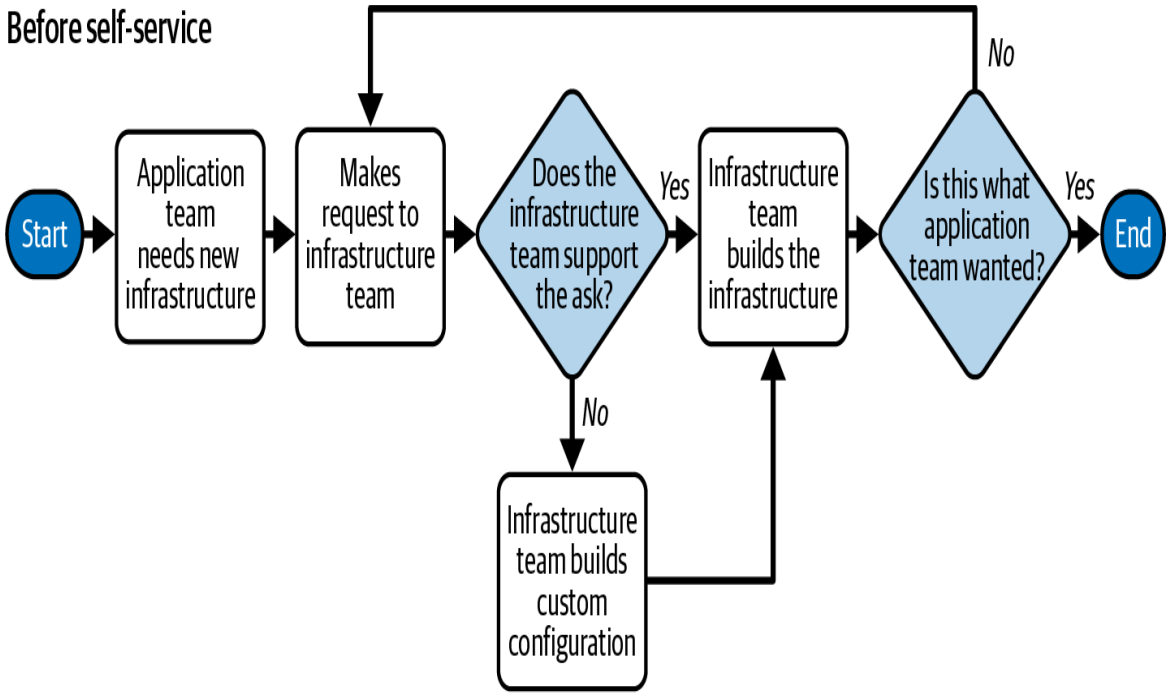
Self-Service Infrastructure

Once you have the capability to create and destroy entire stacks of components on demand, and once you've put sufficient guardrails around creation and controls around cleanup, the traditional holy grail of DevOps, the self-service infrastructure, no longer seems out of reach but becomes a realistic next (major) step to take. This means that these infrastructure stacks are available on demand and are initiated and configured through what amounts to a single request form. With the ability to create ephemeral components and environments, you simply need to construct an interface, such as an API and typically a web-based UI, to expose this capability to end users. This gives a great deal of engineering control to individuals who may not have strong technical acumen in IaC but are responsible for creating application environments. It also allows for highly prescriptive configurations that can adhere to your organization's policies from the beginning. Since this is encoded rather than a custom implementation that's

needed each time, you can tailor your configuration without incurring a repeated cost.

In addition to the efficiency and governance benefits, self-service infrastructure decouples the process of instantiating new infrastructure from the TF development process. As a customer, you request infrastructure through a system instead of having to do so from an engineering team — potentially through multiple layers of communication channels and possibly requiring an extended discussion to align expectations and conduct negotiations. As an infrastructure engineer, you're no longer assigned cookie-cutter tasks to create the same stacks over and over with different input values (if you were so lucky as to have such tasks and didn't have to constantly improvise); instead you build reusable capabilities that may be embedded in the self-service system — and spend significant time ensuring your customers are aware of and able to use the system and its latest features.

With self-service, no longer is infrastructure an arms race or a tough negotiation, with one side (the customer) struggling to define their true requirements, appreciate the time it takes to provision infrastructure, or understand the various technical limitations thereof, and the other side scrambling to tease nice-to-haves from non-negotiables, to account for invisible and esoteric activities, and communicate the various constraints they must operate within. Now, infrastructure teams can focus on enabling new and better features for their customers with stronger and more efficient safeguards, always taking steps forward and never having to retrace or fearing a misstep on a well-worn yet precarious path. They can treat customers as stakeholders and take feedback or direction as feature requests, rather than spending their time replicating successful patterns repeatedly, manually. This makes the process more collaborative and turns anxiety and dissonance into anticipation and harmony. A depiction of before and after implementing self-service can be found in Figure 7-2.


Before self-service
Start
Application team needs new infrastructure
Makes request to infrastructure team
Does the infrastructure team support the ask?
Yes
No
Infrastructure team builds custom configuration
Infrastructure team builds the infrastructure
Is this what application team wanted?
No
Yes
End

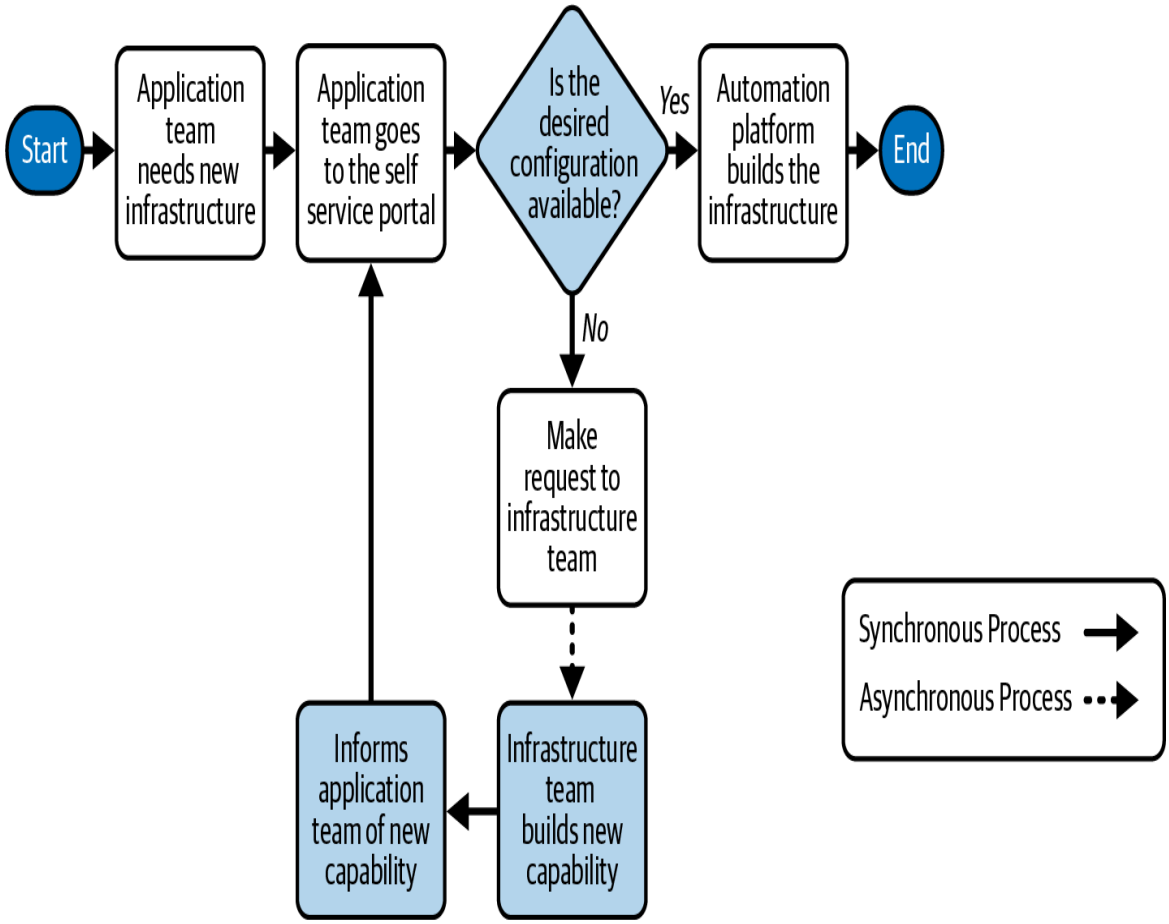

With self-service
Start
Application team needs new infrastructure
Application team goes to the self service portal
Is the desired configuration available?
Yes
No
Automation platform builds the infrastructure
End
Make request to infrastructure team
Infrastructure team builds new capability
Informs application team of new capability
Synchronous Process
Asynchronous Process

Figure 7-2. Before and after implementing self-service

Once you have reusable automation scripts that work consistently and manage ephemeral environments, or components thereof, you can wrap these scripts within a long-running program. The simplest variation would be to just directly call the automation scripts from within the automation platform using the platform's DSL. This may also be the most common version because it's the quickest path to getting something working. However, it's not the most robust solution since there is no great way to validate that it works consistently, beyond running it and evaluating the outcome. A more verifiable option would be to introduce an interface such as an API or CLI to expose its functionality. Depending on what programming language is used to drive these operations, you could have much more control over the inner workings and even enjoy elevated troubleshooting techniques such as using a debugger or tracing solution. Then it's simply a matter of finding a way to configure the inputs to the API or CLI. Because these inputs are so highly structured, it allows for them to be exposed using very simple UIs like a web form: for example, the form fields can be directly mapped to the API's inputs. This can certainly all be done with custom software, but a typical approach is to again abstract this further using an automation platform either to host the CLI and interact directly with it or to make the API calls in a structured, ordered fashion.

> **NOTE**
>
> It's worth noting that this approach of including layers of abstraction better follows the single responsibility principle, as each layer has a different role: the automation platform drives a specific sequence of actions, each of which is structured as its own self-contained process and interface, and each of which runs the TF CLI (which itself abstracts the providers API) and which is effectively at the bottom.

One potential drawback of using common automation platforms to drive the actions for a self-service system is that it requires a good way to trigger the automated workflow. Luckily, many platforms allow workflow jobs to be exposed using their own APIs, commonly referred to as *outbound*

callbacks. These may be called just as any HTTP request would be, and the inputs to the scripts would be supplied as query parameters. As your workflows include more steps, you may want to incorporate SaaS solutions into your workflows, such as to evaluate or communicate the results of an operation, and webhooks are how many SaaS products expose their functionality as well. Webhooks act as the links between your self-service system's automated pipeline and any other systems.

One of the most important ways to prepare for self-service infrastructure is to develop robust visibility across your cloud ecosystem. This means building a combination of comprehensive monitoring dashboards, meaningful and timely alerts, and a variety of flexible audit tools. Monitoring dashboards may be tailored to specific engineering roles or activities. Developing baseline views provides visibility on the most common problem types, and the ability to extend or refine them enables individuals to shape their view to maximize their personal effectiveness. The best practice is to establish various alert types and levels, and they should be continuously evaluated and validated so they don't become noise or otherwise defective. Having so many indiscernible alerts that engineers begin to ignore them — or worse, miss a critical one — is worse than having no alerts because it lures you into a false sense of security. The engineers who receive and address the alerts should have influence over their contents, differentiation, and sensitivity so alerts stay highly actionable and relevant. Audit tools with various inputs and capabilities may be used to dig through different system scopes. The deeper the purview of the inspection, the greater levels of access should be required, ideally, with a temporary lifespan for each grant. A common practice is to introduce a separate mechanism for managing these access rights, and to require a significant approval process. Such heightened control over audit capabilities means these capabilities can be extremely comprehensive, without compromising sensitive information by leaving the inspection solutions open and accessible.

Rather than relying only on the results described in your TF outputs, you'll want to correlate this with what can be periodically detected using API or

CLI requests, along with the running tail of system logs. You may be able to develop advanced alert triggers, or even certain resource events, based on computed behavior or characteristics that go beyond single metric thresholds or binary attributes. Such triggers may also be driven primarily by webhooks, making such an interface extremely valuable for common operations.

Once you have complete visibility across your cloud infrastructure, it's important to place engineers in the correct roles to ensure the dashboards are actually being monitored, the alerts are being addressed, and the audit tools are accessible whenever appropriate. This likely means a combination of roles and schedules such as with late or night shifts, rolling on-call personnel, and offshore teams. This way, you can establish round-the-clock support for your infrastructure, regardless of what gets created or destroyed. This treats your self-service platform as a mission-critical application, which it surely is. Don't skimp on personnel for this support. Although a skeleton crew of a handful of engineers across three or four time zones can be effective most of the time, and they can be greatly enhanced with AI-powered tools that perform actions such as anomaly detection, if some level of role redundancy isn't maintained, you risk a weak or broken link — even if your staff remains perfectly loyal and busy, you run the risk of losing them to stress-related illness or burnout. Consider the two-pizza teams championed by Amazon for each shift or region, and don't try to stretch a pizza across them.

As has already been alluded to at various points in the chapter, introducing a self-service system does not amount to a silver bullet; there's still work to do to extend the solution, resolve any issues, and keep it up to date. Moreover, certain features that are required by a particular workload may not be appropriate to implement as a standard process within the self-service platform. Therefore, there's still a chance that certain components will need to be configured in a custom way. Ideally, these customizations can be layered atop the standard infrastructure configured by a self-service system, still providing a massive benefit to the infrastructure provisioning process. Realistically, you should really only attempt to perfectly support a

vast majority, but probably not all, possible features. Considering something like the Pareto Princple and implementing services to support at least 80% of your organization's teams or workloads is both massively beneficial and likely to be actually achievable. Any customizations that you do introduce, whether small ones on top of your standard components or fully custom solution stacks, should be tracked as exceptions, as they deviate from the standards that are encoded in your self-service systems. Policies should be developed to support the standards, and the exceptions must explicitly override relevant policies, with elevated oversight from the various responsible parties.

Over time, you'll turn some of the features that are initially presented as exceptions into new standards and capabilities. It's important to develop a process for evaluating exceptions and determining which should become standards and which policies must be updated or modified to better support the features that lead to the exception. Doing this evaluation regularly and across all exceptions gives you the opportunity to evaluate the breadth of the exception across all workloads, allowing you to determine how widespread the practice is. If the practice is common enough, you have incentive to introduce it as a capability directly in the system. It gives you the evidence you need to support the decision to prioritize the feature.

Autonomous IaC

Even while true and full self-service IaC is still an unattained aspiration of many organizations of all sizes, nowadays, self-service is becoming something of an antiquated paradigm. It's replacement (unattainable?) practice du jour is fully automated IaC driven by modern language models. In this methodology, a language model is trained to focus on TF code, reference architectures, and other material that helps to express possible and successful architectures. Although there are many tools and services that promise autonomous IaC, the delivery is largely lacking. This is likely due to how unique every case truly is. Because language models must be trained on existing data, if we observe Conway's Law, it would have to already

know how your organization operates to know how to construct the systems. Therefore, today there is very likely still a highly skilled human operator, or team of operators, who help validate and mold the immediate results presented by a TF code-generating large language model (LLM). In some cases, the tool or service is effectively a Mechanical Turk, whereby skilled human engineers manually arrange the code, and the AI tool is used as more of a guide or templating tool to scaffold out portions of the codebase, rather than allowing it to make unsupervised changes. Surely the tools will get more effective, but until artificial general intelligence (AGI) becomes a reality, the tools are only largely able to recreate what they've seen before (or combinations thereof), rather than reliably generating truly emergent solutions — most commonly, when an LLM does produce an emergent result, it's referred to as a *hallucination*, and, outside of artistic mediums, is rarely considered valuable.

> **TIP**
>
> The paradigm of *vibe coding* has gained popularity lately. However, if you're operating within a safe environment with ample guardrails, such as the many hosting platforms that are currently available, you may not run into major problems like unsecure networks or weak access controls. If you're building the environment, you're responsible for establishing these guardrails; they don't magically exist, and they may not be fully included in a totally vibe-coded infrastructure stack. A skilled infrastructure engineer and significant architecture oversight must guide your infrastructure development, and even if AGI is achieved, these decisions should be the responsibility of humans, not software. In other words, vibe code your applications, not your infrastructure.

One well-worn point of consideration for any solution based on machine learning, such as LLMs, is the concept of *garbage in, garbage out*: regardless of how sophisticated the solution, if it's provided with unrealistic, corrupted, or otherwise bad data on which to base its analysis, it has a low chance of performing effectively in real-world scenarios. This has many implications on infrastructure architecture design and maintenance patterns. For example, if you ask an AI agent that generates TF code to extend your solution that was poorly architected to begin with, it will likely lean into whatever practices and conventions it finds, regardless of merit. Similarly,

if the system is trained on popular forum answers, incomplete documentation, and tutorial-level code repositories, it may not be able to effectively reason through complex organizational requirements, which more than likely include conflicting or contradictory policies and priorities that need to be negotiated and mitigated.

Even if your organization's complete infrastructure design can be calculated satisfactorily by a GenAI tool, you would benefit from a responsible party with a pulse to answer for any issues that arise from curious decisions — and, more importantly, that cannot be switched off, forcibly upgraded, or become unlicensed, potentially losing the ability to understand how it got to earlier decisions altogether. If you can reduce the experience to a simple approval process or series thereof, it leverages the power of language models while keeping the power of decision making in human hands, combining the best of both approaches.

An autonomous IaC approach can be built on top of a self-service solution. Autonomous agents can be designed to read the self-service requests, interpret them, and provision the predesigned stacks. The agents will be able to interpret requests better than any mapping of form fields can hope to do, and they may even have the ability to make inferences or find otherwise undiscovered efficiencies, redundancies, or commonalities across stacks, potentially leading to more flexible or complete solutions. The agents act as the interface, and your scripts and TF code uphold the standards and guardrails you require to preserve. Depending on the agent service or configuration, these agents can even automatically learn from mistakes, and may possibly be able to generate more detailed explanations for why a request is invalid or incomplete.

> **TIP**
>
> A possible extension to this automatically generated feedback would be to also feed the explanations back to documentation, such as an FAQ collection. These can even be ordered or sampled based on frequency — or recency — of appearance.

An automated interface to your infrastructure management system might be able to operate with a great deal of true, unsupervised autonomy, preferably with some level of oversight and certainly with well-developed audit capabilities. The assurance in this case comes from your core system built from TF and your automation scripts. However, if you intend to incorporate autonomous agents into your TF or script authoring procedure, it's unwise to do so in expectation that you will be able to reduce your DevOps team to triviality, or that you can shed experience, hiring underskilled replacements at a discount. In other words, don't lay off your infrastructure or operations teams upon adopting an autonomous IaC approach. Instead, retrain and reposition them to be custodians and curators of the infrastructure resources and provisioning assets. This may mean introducing training seminars to rapidly establish knowledge and skill baselines and pilot programs to validate a proof of concept of how you may reorient your support structure within your organization.

One pattern that can be adopted early on is the use of AI-powered code generators to facilitate the construction of certain components, such as resource elements, within a TF module. Many such generators plug directly into your development environment (or are a foundational component), and thus would be aware of all the code in the workspace, such as the contents of your repository. A major benefit to this is the service is typically able to consider the structure and style of your existing code. In many solutions, you can even opt out of allowing your data to be included in its training data, the knowledge base on which the language model operates.

However, many organizations are hesitant to turn over their intellectual property, no matter how favorable the terms of service, until the service proves itself to have value — or strong evidence of potential value. In these cases, it's still possible to leverage LLM-based solutions in a relatively anonymous fashion by simply constructing specific prompts on how you need your cloud resources to be configured or organized and how the language model should respond, such as with TF code or your automation platform's DSL. This allows you to keep certain portions of your code anonymous by having the LLM output placeholder values for sensitive

fields. Due to the nature of language models, which are highly effective at making connections and inferences among seemingly irrelevant information, it may still be desirable to disallow the model to use this prompt history for its training data.

However, going back to the earlier discussion regarding garbage in, garbage out, you need to be careful in what you ask for from an LLM-based code generation solution, since it may not explicitly call your attention to — or even catch — a defect you're introducing or security hole you're leaving open, especially those you ask for. You can mitigate this potentially dangerous behavior by providing the model additional context, such as a *system prompt*, that establishes standards and guardrails for the model's responses to adhere to. It's also an excellent practice to request that the model ask any clarifying questions it comes across in its evaluation, so as to reduce the chance that it makes an inappropriate assumption.

> **TIP**
>
> In addition to including standards, safeguards, and expectations in the system prompt, you can also leverage it to provide information on the other input values, such as information used to apply a degree of randomness to the calculation (seed or temperature, for example), and other output components, such as the confidence level the model assigns to its response. This information can be printed along with the normal response to provide additional evidence as to how trustworthy a response may be.

If you don't have robust infrastructure and operations teams, perhaps because your organization is lean like a startup or has recently reoriented from a service or goods industry toward a heavy digital offering, there are software solutions that claim they can generate your entire cloud infrastructure for you. No doubt, there are several language models that, with highly skilled oversight — not only in architecture and engineering but also in business aspects such as organizational management and strategic vision — can help to produce top-quality TF code and, when properly directed, can even help develop entire multilayered cloud environments and ecosystems. It may even be possible to produce condensed models or

construct highly specialized system prompts that significantly outperform stock configurations. However, creating a fully autonomous agent based on modern language models that can do this for any given company type, size, or industry would be a tall order indeed. This is because today's models consistently perform well only on things they've seen before or that may be combined from things they have seen before, and every organization is unique. Not only is it highly unlikely that two businesses consist of the same organizational makeup, but because of the nature of individuals, there are almost certainly countless subtle differences that cannot be so easily detected. Personal relationships or preferences drive decision making more often than most would like to admit, resulting in unpredictable behavior and otherwise indefensible rationale. Unless the model is granted access to observe (or review) prior behavior for a significant amount of time (weeks or months) and to validate its understanding along the way (which requires full transparency, possibly beyond an individual's capacity to guarantee, considering everyone has blindspots, biases, and beliefs), it will likely perform poorly when predicting an individual's future decision-making behavior. This suggests that it will also struggle to accurately articulate all the decisions and rationale that resulted in the organization's structure, which means that it likely must be informed directly of the nature of the structure. The result may be serviceable, but it's unlikely to be exemplary.

EMERGENT THOUGHT IN AI

Today's language models largely rely on pattern matching, so much of the model's "reasoning" capability hinges on the arguments available to it. Language models don't *know* anything, but because they're trained on data that contains at least some written material that is authored by folks who ostensibly *do* know something, it's often simply a matter of selecting the most plausible notion (and the one that is most consistent with other perspectives it considers to be highly plausible). If the right answer does not directly appear in a training set, the model may be able to combine enough complementary material to construct a response that is both accurate and technically emergent. As language models grow and evolve, there may be some that contain enough material to effectively appear truly intelligent, rather than just extremely knowledgeable and proficient. There are already some that even consistently pass the Turing test, which is indeed remarkable, but likely suggests the need for a better test (or more discerning participants) instead of a chance to collectively hang up our thinking caps.

Truly emergent thought — the groundbreaking epiphanies made by the Newtons, Gödels, and Einsteins of the world, and that is promised any day now by the so-called AGI ideal — is achieved not by virtue of having the most knowledge, but through imagination. In a way, there's a phenomenon exhibited by language models that may be likened to imagination: the dreaded so-called *hallucination*. Whenever a model produces a response that is wrong or inconsistent with reality, or when it gets into a reasoning spiral that either never ends, results in something useless or meaningless, or overflows the memory, it's said to be hallucinating. Usually, this behavior is considered detrimental or inappropriate, since the expectation is typically that one can trust the model's output (unintuitively, this behavior becomes even more acute when the model is usually highly accurate, as it lures operators into a false sense of security). However, if a hallucination arises from the recognition of a pattern among seemingly disparate concepts, fields, or mediums — and if this is relatively consistently distinguishable from

> hallucinations brought on by combining incorrect, inconsistent, or conflicting narratives — it could theoretically approximate imagination. Unfortunately, although interesting and exciting, we may not yet have a proper framework for evaluating imagination (even in humans), so even if we were to achieve AGI, it might take additional time to effectively verify it.

Consider again Conway's Law, which suggests that an organization's platforms will approximate its communication structure. Assuming the company's IaC ecosystem and the mechanism by which it is developed and maintained is such a platform, it should generally resemble how the company operates, internally. However, because the company's communication structure not only incorporates relatively obvious aspects such as corporate hierarchy, production methodologies, and collaboration rituals, but critically includes the informal communication channels established among individuals over chat clients, emails, or even at the proverbial water cooler, it would require a language model to be aware of all such channels to effectively construct a corresponding solution. This means that no solution is likely to be fully plug and play. Instead, it relies on a great deal of (accurate) information, either through discovery or fed to it by an informed operator. If you choose to go the route of a fully autonomous IaC solution, it will likely require highly skilled stewards to curate the body of knowledge (minimally including the institutional knowledge, but perhaps also even the gossip and lore) and justification for how your organization operates, and thus how the system should be structured, to go beyond serviceable and achieve a satisfying outcome.

Definition of Done and Diminishing Returns

One question on every stakeholder's mind is, "How long will this take?" Each layer is clearly a different effort, and thus each phase of your platform will have a different scope in terms of personnel, timeline, and operating costs. Some platform features may also be more tolerant to throwing more resources against it, such as more engineers, system integrators, or premium

licensing, ito accelerate the initiative. However, there may not be a perfect formula for estimating the time it will take to complete any one phase. Moreover, every phase will experience diminishing returns as solutions become more and more comprehensive, as there is a high chance for corner cases and idiosyncrasies that are not obvious until some progress has been made and your team begins to peer under the hood of your existing procedures.

An overly simplistic, yet fairly common, approach to estimating the scope of an engineering effort is to come up with a best faith assessment, double it, and then double it again. Although this seems like it would produce excessive padding, it helps to account for the *unknown unknowns* that always seem to appear when you least expect them — natural delays in development due to external forces, societal occurrences such as holidays, good old-fashioned bureaucracy, and the obligatory stakeholder check-ins, which tend to grow in frequency, length, and urgency as an effort progresses toward significance and becomes more and more pervasive. It also serves to mitigate the chance of losing out in budget or scope negotiations. However, it's one thing to declare a length of time for an effort and another to assess when the effort is truly complete.

An important exercise to conduct as you begin to close in on your scheduled time of completion is to evaluate whether the effort will achieve a satisfactory level of completeness or if it should be extended. This should not be done ad hoc, with stakeholders ambushing engineering teams in their normal cadence meetings or in impromptu grilling sessions, but must have clear terms of engagement, including good faith contingencies, that are determined along with the other scoping considerations before the effort gets underway. Even in the case that the overall initiative is smooth and relatively uneventful, with well-forecasted delays or dips in material output, this provides stakeholders a framework for evaluating the progress or fulfillment of a phase with a high degree of confidence.

Maintaining the ongoing theme of this chapter, the definition of *done* for a particular effort will be different for every organization. Considering again the Pareto Principle, you may find that the majority of your components

transition quickly to adopting a new feature or introducing a new operational layer, while attempting to continue the effort in order to fully support all of the remaining assets proves to dominate the overall timespan. Similarly, you may find that a satisfactory level of support is somewhere south of 100% coverage. And as your processes become more mature and widespread, the overall solution becomes more stable. Eventually, you'll likely begin to see diminishing returns with every additional component of your infrastructure ecosystem that you support with the new solution. This is why regular stakeholder reviews as you advance toward high levels of saturation and stability are valuable.

One example of diminishing returns is the elevated effort it may take to include support for a particular process that doesn't get much attention or see many executions. If it's something that is highly cumbersome and has a low chance of needing to be repeated, it may not be worth the effort of including its support in your platform. Another example is the granularity with which you establish access controls. It's often most beneficial to cover specific roles and tasks and to combine these for different members with different sets of responsibilities, rather than try to create distinct, team-specific suites of grants. The former scales much more effectively and better exposes undesirable overlaps or expresses intentional similarities in responsibilities between teams. Similarly, another example is the granularity with which you subdivide your workflows. Assuming you take a top-down approach, decomposing significant pipelines into smaller, composable operations, you probably want to stop once you have comfortably reduced significant repetition between otherwise distinct subprocesses. However, you probably don't want to extract single steps that appear in only a couple of instances — you may even stop far short of this if you have a robust code review process in place. Instead, you should have a straightforward process for evaluating repeated processes and establishing when to create a reusable subpipeline.

Just as important as recognizing that a platform layer has reached satisfactory maturity and an effort is complete is knowing when to begin the next engineering effort. This again is unique to your organization and is

subject to other priorities, appetite for advancement, and even economic winds. Certainly do not begin a new phase before an ongoing phase reaches a sufficient degree of completion or maturity. However, once the phase is complete and the proverbial dust has settled, but before the experience becomes a distant memory, it's likely the best time to consider the next phase, drawing not only on the insights of the team that shouldered the engineering effort of the newly complete layer, but also on individuals or groups that have first-hand experience in establishing these features. This way, you provide the decision makers with the highest quality evaluation of the effort necessary for the subsequent phase.

Successful Practices

This chapter describes various initiatives that amount to the development of a platform for infrastructure management based on TF as the central process software. In many cases, these can be layered on top of each other, moving to more and more automated solutions with the need for less and less human-provided input and interaction. The following set of practices all lead to greater degrees of success. They are partially ordered, starting with those that are both relatively low effort and likely to be included in any highly mature platform, followed by practices that are highly effective, somewhat costly, and the bulk of the automation platform, and finally those that likely resemble an operational goal of your organization, built atop an already fairly complete and robust platform built from the other practices.

Low effort, high value practices include:

Shell scripts

> Record the shell commands and procedures as a shell script. This reduces the chance for a typo in a command or a missing step.

Containerized steps

> Run the shell script from within a container to ensure consistency in available dependencies and execution context. These can greatly

improve the experience of running commands from different computers, such as an engineer's workstations.

Custom containers

Custom container images may be constructed, for example using a Dockerfile, to impose greater control over the context in which a process runs and the dependencies that are made available to it.

Automation platform

Use of an automation platform vastly improves consistency in the execution of infrastructure processes because all operators are leveraging the same system, rather than operating from their individual workstations (which may behave differently, no matter how similar they are). Choosing a platform that natively supports containers, job triggers, and webhooks will greatly simplify some of the more advanced practices listed next. This often becomes the baseline platform on top of which all other platform components are layered. To ensure safe usage, it's important to select an automation platform that supports access management support, such as RBAC; luckily, this is a fairly common feature. Most automation platforms also support a DSL that allows pipeline configuration to be stored in a source control system, such as Git, which improves reusability and allows for greater governance through practices such as code reviews.

Highly effective, primary platform practices include:

Triggering other pipelines

Decompose processes into individual procedure sequences that act as reusable workflow subprocesses and that can be triggered from parent pipelines.

AI-generated recommendations

Architects, operations personnel, and infrastructure engineers leverage an AI assistant or code-authoring tool that can make recommendations for new elements or improvements to existing environments.

Programming language

Consider transitioning from a shell language such as bash to a more general purpose language like Python, Go, or JavaScript. This way, you can develop unit tests to ensure desired functionality is maintained.

Platform visibility

Implement visibility best practices and best of breed tools. Audit the monitoring, alerting, and auditing tools.

Webhooks

Wrap pipeline triggers in webhooks/HTTP requests to expose them over the internet and utilize features of REST architectures.

Temporary access controls

Implement temporary ABAC or RBAC grants for all operators and stakeholders of your infrastructure platform.

The most advanced platform practices are:

Simplified self-service

Integrate a form-based ticketing tool that implements webhooks.

AI chatbot interface

Introduce a chat-based AI agent as an interface for your infrastructure platform that monitors your environments and initiates update requests.

Supervised TF code generation

Incorporate an AI assistant or code-authoring tool to recommend further efficiencies or improvements as code changes, for example, as pull

requests that are automatically opened. Don't allow unsupervised updates on infrastructure that underlies your critical business services.

Implementing all of these practices is not necessary to achieve a high degree of success. There are even reasonable progressions that will skip certain practices along the way and still achieve the highly advanced setups. However, it's worth considering which practices your organization would benefit from and how far along the path you're looking to go.

Summary

In this chapter, you learned about the different features that can be combined and layered to make fully automated TF ecosystem platforms, including:

- Turning shell operations into shell scripts — and possibly into more traditional programming languages such as Python or Go
- Executing or embedding operations in container images to isolate the process
- Componentize reusable pipeline components and connect them to establish your various infrastructure environments
- Developing ephemeral or on-demand environments as needed to establish a definitely scoped or time-framed exercise such as a battery of integration tests
- Layering a form-based platform over your baseline automation platform to enable a self-service approach
- Introducing AI agents into the IaC code development process

You now realize there are several practices that, when deliberately layered, can be combined to establish a best-of-class platform for maintaining your organization's infrastructure. You know that there is a multiphase strategy to get from manual operations to a nearly hands-off approach. You also

recognize that these layers can work together and that a best of class automation solution does not skip to the most advanced or exciting features, circumventing more rudimentary practices, but takes a patient, deliberate approach, ensuring each phase has reached a sufficient level of stability, saturation, and accessibility before declaring victory and moving on to subsequent features.

In Chapter 8, you'll take a deep dive into the process of decomposing a project to allow for enterprise scale. You'll also encounter various characteristics that indicate the need to evolve a project past a single repository or module. You'll see ways to expand beyond a single, simple workload in order to support all of your organization's operations.

Chapter 8. Evolving Projects Into Enterprise

Each organization has a unique experience adopting IaC, but they often have the same few characteristics: they either start with a single TF project and end up with a massive module and state, or they start with multiple projects but end up collecting most of their components into the most active one, and still end up with a massive module and state. At some point, the team becomes overwhelmed with endless merge conflicts, locked states, or an elaborate review procedure that requires multiple Git branches or tags to track effectively. In many cases, this causes a panic when things go wrong around a major release or when there is a significant exposure of sensitive information. The choice then becomes fairly binary: hire consultants to evaluate the existing system, create a plan of action, and execute, ideally, within the constraints laid out in the contract and likely over multiple quarters; or take on the initiative, internally — after all, you already have the domain knowledge and know where all the problem areas are and what all the possible procedural snags will be.

The purpose of this chapter is to expose you to the various factors and indicators that influence the need to split. Whether or not you start with a single project, it's important that you recognize the likelihood that you will need to split or reorganize your infrastructure into multiple root modules and organized into layers or a hierarchy. You should start tracking, however informally, these factors. You should also begin to draw attention to the potential future need with would-be stakeholders. This need not be regularly reported, but it should be included as a caveat when first adopting TF as your IaC tool.

Although it doesn't resolve the question of how responsibilities are allocated or costs are absorbed, the more familiar the relevant parties are with the inevitable deconstruction of TF modules, the smoother the eventual

process is likely to be. It reduces the chance of an escalation being overlooked or wrongfully deprioritized in favor of a shinier or more tangible enterprise, or from being improperly scoped or budgeted out of an over-abundance of either caution or confidence.

Perfect awareness of the considerations expressed in this chapter also does not guarantee a totally smooth experience. In fact, such exercises almost never go as smoothly as hoped during the planning phase. It may not be possible to sufficiently anticipate and fully mitigate all the snags or drags that occur during implementation. The best approach is to build in a balance of cushion to absorb minor delays with regular and high-priority stakeholder reviews to reveal more significant impediments. However, if deconstructing components is done in a haphazard and panicked way without giving it the time and exposure it needs to materialize and mature, it almost certainly results in an even more tumultuous ordeal.

However the journey transpires — be it prolonged and messy, with great impediments to overcome and skirmishes along the way, or be it smooth and expedient, guided by wisdom and foresight — the desired outcome is for the resulting TF root modules to be better aligned to your organization's responsibility hierarchy and, pointing once again to Melvin Conway, able to better support the communication channels inherent to your organization.

Signals for Change

As asserted many times throughout this book, every organization is different, and given that there are several factors that influence the need to split a TF project, every organization has a unique evolutionary story. As presented in this chapter's introduction, you should start tracking these factors from the beginning, whether you start with a single or dozens of TF root modules. This may be through rough aggregates of common problems in source control or approval procedures, as referenced earlier, and could be reviewed internally on a quarterly basis. Once these factors begin to trend upward, it should trigger the escalation of a report that exhibits this trend,

providing key decision makers with the information they need to evaluate the net outcome of a redesign.

Some of the indicators are directly measured and thus easier to track than others. Others are obvious, with no other viable option but to proceed with a refactor. Others still require indirect or highly subjective measures of detection such as labor or operating expense (OpEx) cost attribution, internally conducted surveys of team members, or a heightened sensitivity to empathetic signals. The following are the factors and indicators identified and considered in this chapter, grouped in terms of their measurability, followed by expanded commentary for certain ones. Each topic is considered for a single TF root module or state.

Measurable, easy-to-track indicators include:

Lines of code

> This includes the length of both the *.tf* files and the *.tfvars** files. Although this is likely the easiest metric to track, on its own it may be one of the most arbitrary. However, combined with the number of resources, it can express the growth of the TF project and may indicate sprawl or suggest distinct, emergent domains.

Number of resources

> This is the total number of resources tracked by a single TF state. Combined with lines of code, a dependency graph, or number of local modules, this can help identify distinct domains or hierarchical relationships.

Number of child modules at root

> This is the number of local and third-party modules that appear at the root module as child modules. Although this is a wholly viable approach for constructing tight dependencies between distinct collections of similar or related resources, combined with indications of highly active or largely inert child modules can elucidate incongruent lifecycles among them.

Frequency of delays due to locked states

Whenever a plan is being applied, the state becomes locked. This is to prevent concurrent changes that may clash, undo each other, or otherwise conflict. As such, it's an extremely valuable fail-safe mechanism to ward off such *race conditions*. However, if states stay locked too long or are locked too often, it can cause delays in subsequent changes as developers wait for the apply process to complete and the lock to be released. Combined with the existence of multiple teams contributing to a single TF state, this points to a glaring need to deconstruct the project into multiple projects, one for each team.

Aggregate duration of delays due to locked states

One common way to mitigate frequent delays in applying a sequence of changes is to implement a process queue where one apply phase begins after the prior one completes. This often works well in practice, but it can break down if an apply phase is ungracefully terminated and the state lock is not released.

Number of distinct operating teams

Perhaps the most glaring metric suggesting the need to split is how many distinct teams directly operate the TF project. If it's more than one, then you have a decision to make: split the project such that each team is responsible for a single project, shift the responsibility of one team to the other team, or combine the teams.

Evident factors include:

Mergers and acquisitions

The process of combining two organizations in a merger or acquisition often includes splitting up, or vastly refactoring, root modules.

Transitioning to the cloud

Assuming you're already using TF to manage the software workloads running in your on-premises datacenters, bringing in a CSP TF provider and combining the resources into one state can be confusing and complicated.

Extending to another CSP

Extending your existing cloud infrastructure to another provider will likely require an evaluation of the services available from the new provider to determine what components may be directly transferable or redesigned in the new provider.

Adopting a new infrastructure or data platform

Similarly, the decision to adopt a new infrastructure or data platform as a core component of your business operations is an opportunity to determine whether the new components can fit in an existing TF project or deserves its own.

Significant, disproportional expansion

When a portion of a TF project, or a branch of a TF state, becomes disproportionally massive, it often indicates an emergent domain that may be best served as a separate TF project and root module.

> **NOTE**
>
> Yet another example, in theory, would be if an organization were to wholesale migrate from one CSP to another. However, in practice, this is never done at the flip of a switch. The majority of the effort would resemble that of the extending to another CSP example.

Indirect, subjective indicators include:

Dependency graph

Hierarchical and other relationships among services can also suggest the emergence of distinct infrastructure domains or layers.

Non-normalized development activity

> Although it's unlikely that development activity across any project will manifest as perfectly normal statistically, realizing where your project's *long tails* are — whether a few resources or child modules see the vast majority of action, or whether a small group becomes almost inert — can help highlight domains and layers.

Frequency of merge conflicts

> Recurrent instances of merge conflicts, especially between two distinct teams, may suggest a reorganization of components to reduce the chance of conflicting code changes.

Material cost of making an update

> If you measure the average total costs of making a change — including labor, resource consumption, and subscription attribution — a drastic trend upward may indicate challenges that might be addressed only by reorganizing or redesigning the infrastructure components.

Team velocity

> Many popular software development methodologies measure the velocity of a team, typically through some mapping of tasks completed over a time period. Ideally, over time, as the team becomes more familiar with the work and more accurate in their planning, the team's overall velocity should trend upward. A downward trend in team velocity indicates a growing cost to making updates.

Engineer stress and burnout

> Team member attrition or other HR-related considerations can strongly indicate an overly complex configuration.

To expand on these, if you have more than one distinct team operating the same TF project, you have competing interests, and this can lead to other

problems, such as merge conflicts or disproportional development activity. Evaluate the rationale for keeping the teams separate and for assigning them overlapping responsibilities. If you determine that the teams must continue to be distinct, there should be an overhaul of either the responsibility or operation of the relevant infrastructure components. To consolidate the responsibility into a single team, you may establish a request system to ensure the previously responsible teams remain accountable for the effort of constructing and maintaining the relevant resources. Keeping the responsibilities intact would strongly suggest the project be separated into correspondingly distinct TF projects. Finally, combining teams, however loosely, or partially assigning engineers to both teams, may be the most effective way to reduce the friction related to continuing multiteam TF projects.

The transition from on-premises datacenters to a CSP is daunting. It requires new architecture considerations, potentially with entirely different infrastructure stacks and likely with largely different network structure for service delivery. It also requires relinquishing control of previously strictly controlled appliances and tightly clutched responsibilities. Both of these can provoke what amounts to an identity crisis in the culture of your technology business unit. Worse, it may even present a mismatch in the skills needed with what your engineering teams currently possess in expertise, further exacerbating the predicament. In other words, one does not simply make the decision to transition to the cloud. Even when the goal is to rehost the workloads — and not to replatform or rebuild your digital ecosystem in a cloud native or managed service-oriented way, a common choice for large enterprises and anyone with multiyear licensing deals — the networking considerations alone typically require an extensive assessment or design process to establish a corresponding degree of availability and protection in the new setup.

Often, adopting a new infrastructure or data platform has some degree of greenfield development, with net new stacks of software services that may need to be fully constructed and tested before they're put into production. Although not a hard requirement, this does naturally suggest the value of a

new, self-contained operations mechanism. Any dependencies that the new project has on the old can be codified through triggers between workflows (or workspaces). Examples of such stacks include the implementation of Kubernetes and containerized services, big data platforms, and data streaming solutions. However, there may also be components that eventually get transferred from an existing project to the new one once the integration is complete, if that simplifies access boundaries or lifecycle management.

Because these changes are so big and systemic it may seem to introduce just as much friction as there was before the split. Now there are even more engineers' hands on the levers and even more stakeholders to which to answer. Assuming the effort was conducted in good faith, the best path of action is to resist the urge to revert back to old ways and instead take the time to reduce points of friction as they arise. This requires a great deal of continued faith and buy-in from decision makers — or a heroic effort on behalf of a handful of engineers to keep the friction from escalating beyond their scope of responsibility.

Finally, such an endeavor should be treated just like any mission-critical initiative. Its scope should be thoroughly analyzed with the help of project managers, product owners, and tech leads, and its design should include the input of relevant architects and subject matter experts. Only once these exercises are completed, or otherwise reach a definitive conclusion, can the initiative be scheduled.

Evolutionary Patterns

Once you have determined it's time to evolve your IaC codebase, there are several available patterns. Some of these are better suited for certain situations, and none of them are a silver bullet that may be universally applied. All such patterns require significant deliberation and thoughtful scheduling to reduce friction and account for challenges. Typically, the best approach is a strategic combination of approaches. It's especially important not to develop a preference for a specific one or to default to techniques that

were previously successful purely based on familiarity. It's often worth considering two or more approaches to estimate how each would scale or further evolve.

Building on Top of Base Infrastructure

The first pattern to consider is to build new components in a new project that is established atop or within previously established infrastructure. This works best in the case where a new platform or architecture is scheduled to be established. This also becomes a natural result of an effort to extract a distinct domain from a project that also includes a parent domain.

In a way, this is the ideal approach for new infrastructure or platform stacks: to not need to refactor old components, but to purely introduce a self-contained, greenfield project that fully establishes the new components. In practice, this has the potential to work smoothly only in cases where there has been a high degree of consideration taken in previous work and the environment is already well designed, componentized, and layered. More often than not, there are components that cross or bridge logical domain boundaries. In such cases, the greenfield development can still be maximized if not the totality of the effort. Figure 8-1 is a depiction of the resulting state tree renderings of this approach.

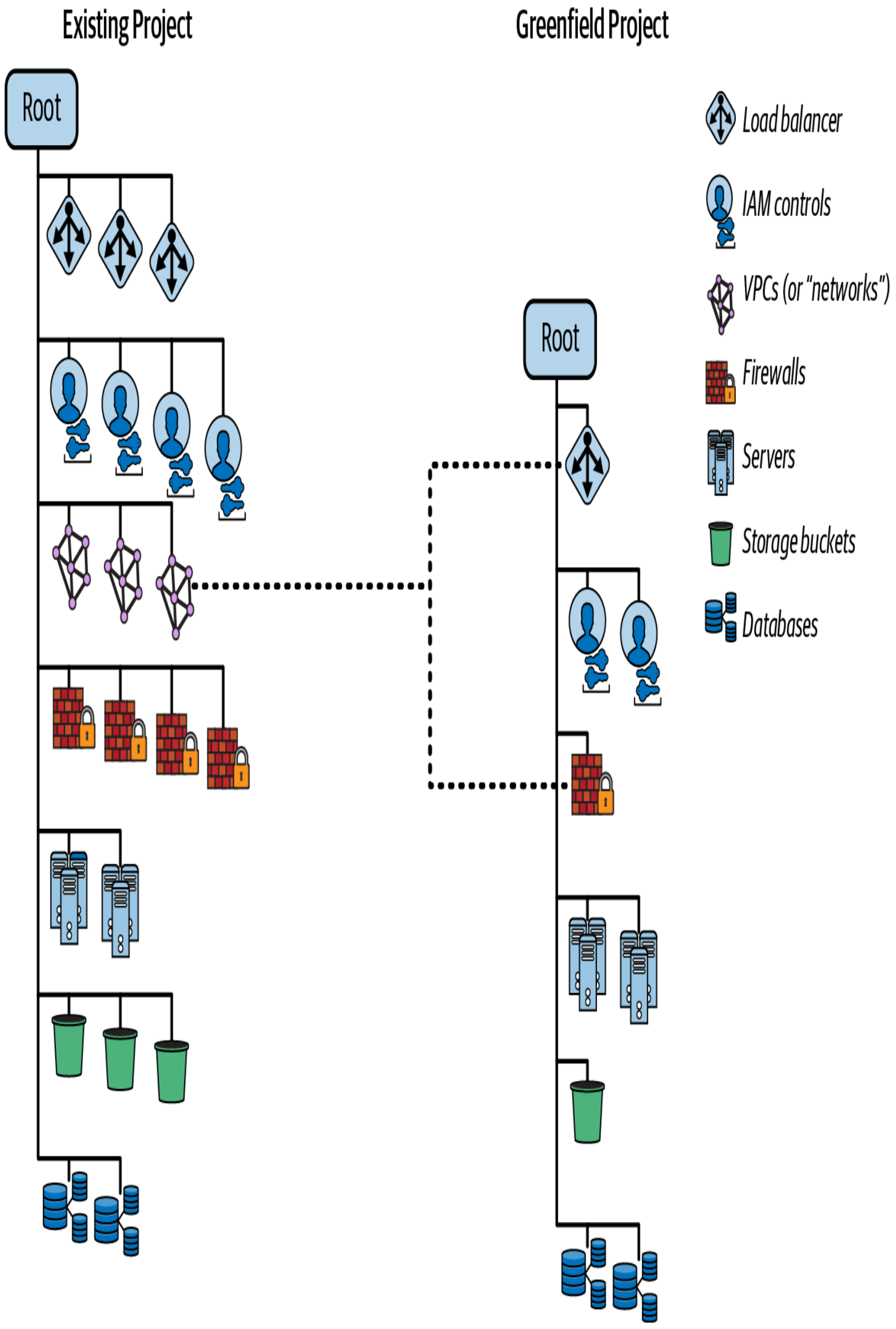

Existing Project
Greenfield Project
Root
Root
Load balancer
IAM controls
VPCs (or "networks")
Firewalls
Servers
Storage buckets
Databases

Figure 8-1. Building a greenfield TF project that sits atop and depends on an existing TF project

Strangler Fig

As engineers and scientists, it's typical to try to apply accepted procedures and techniques from a familiar context to new situations, perhaps only conceptually, in an attempt to accelerate the establishment of a baseline approach. Although this sometimes leads to misconceptions, especially when the difference in context is overlooked or downplayed, when utilized with a healthy degree of doubt or modesty, it can simplify both the implementation design and the communication of new procedures to otherwise uninitiated parties. One pattern from which we might draw such a parallel is the Strangler Fig pattern. This was designed to break down a monolithic digital workload, perhaps running on a mainframe, into a related collection of smaller workloads, exposed as network-addressed services. These services are expected to replicate current behavior of the monolith with the intent of eventually replacing said behavior.

At a sufficiently high level, the Strangler Fig pattern can conceptually be applied to the effort of breaking apart a large, monolithic TF project into layers and domains that may be self-consistent and share a common lifecycle. Often, when breaking down a TF project, the original project remains (preferably with a highly diminished volume of resources it tracks). Therefore, this pattern would be most appropriate in cases where it's determined that a single project should be broken down into more than two TF projects, implying multiple new TF projects. Each domain is identified and prioritized, with a dedicated validation exercise. This way, each new project can have the full focus of the team, and any variations can be mitigated and minimized. Otherwise, you run the risk of trying to do too much at once, and it may not always be clear whether undesirable variations are self-inflicted or natural occurrences.

The decision to employ the Strangler Fig pattern should be followed by a serious consideration to establish distinct, dedicated teams that would be responsible for their management. Especially in cases where the emergent domains already have dedicated teams assigned to them, such as security

domains like access control, networking, and device policy enforcement, the segmentation of responsibilities across multiple teams is highly valuable. All of this implies a significant degree of foresight and planning, which may not always be available in cases where issues are piling up or work is coming to a grinding halt because of the presence of other factors like frequent merge conflicts or significant delays due to locked states.

Naturally, the central thesis in implementing an old pattern in a new context is the similarities between old and new. However, an essential step is to assess how the contexts *differ* and especially where a pure interpretation begins to collapse. In the case of the Strangler Fig pattern, while we may be extracting or teasing out significant portions of a large, singular, self-consistent collection of components into multiple smaller — also self-consistent, but potentially not totally independent — reorganized collections of components, we're certainly not decomposing a single monolithic workload into, say, microservices. This requires a different means of validating the efficacy of the new components, as we're not (necessarily) driving computational logic or network traffic through the entire system. Luckily, the inherently modular nature of cloud services allows for relatively more coarse-grained, direct, and focused comparisons than what one can expect from replicating logic written in one language and embedded within a much larger system into an externalized microservice that likely has a completely different runtime environment and computation features. You can be highly confident that resources with identical attributes, perhaps up to their identifiers and any directly applied access policies, will operate in largely the same way.

Another major caveat to the Strangler Fig pattern is that certain attributes of cloud service resources must be unique, which can complicate the expectation to have coexisting components. These certainly include the CSP's internal identifier, but there are sometimes other fields as well. However, most CSPs expose various ways of identifying resource instances through the combination of other fields that themselves need not be unique. With this understanding, naming conventions may be established that allow the construction of combinations that together also uniquely identify

provisioned resources. This might be through a combination of name, locality, and resource tags, and requires a mechanism that can ensure the combination is indeed unique. Figure 8-2 depicts extracting multiple domains in separate steps.

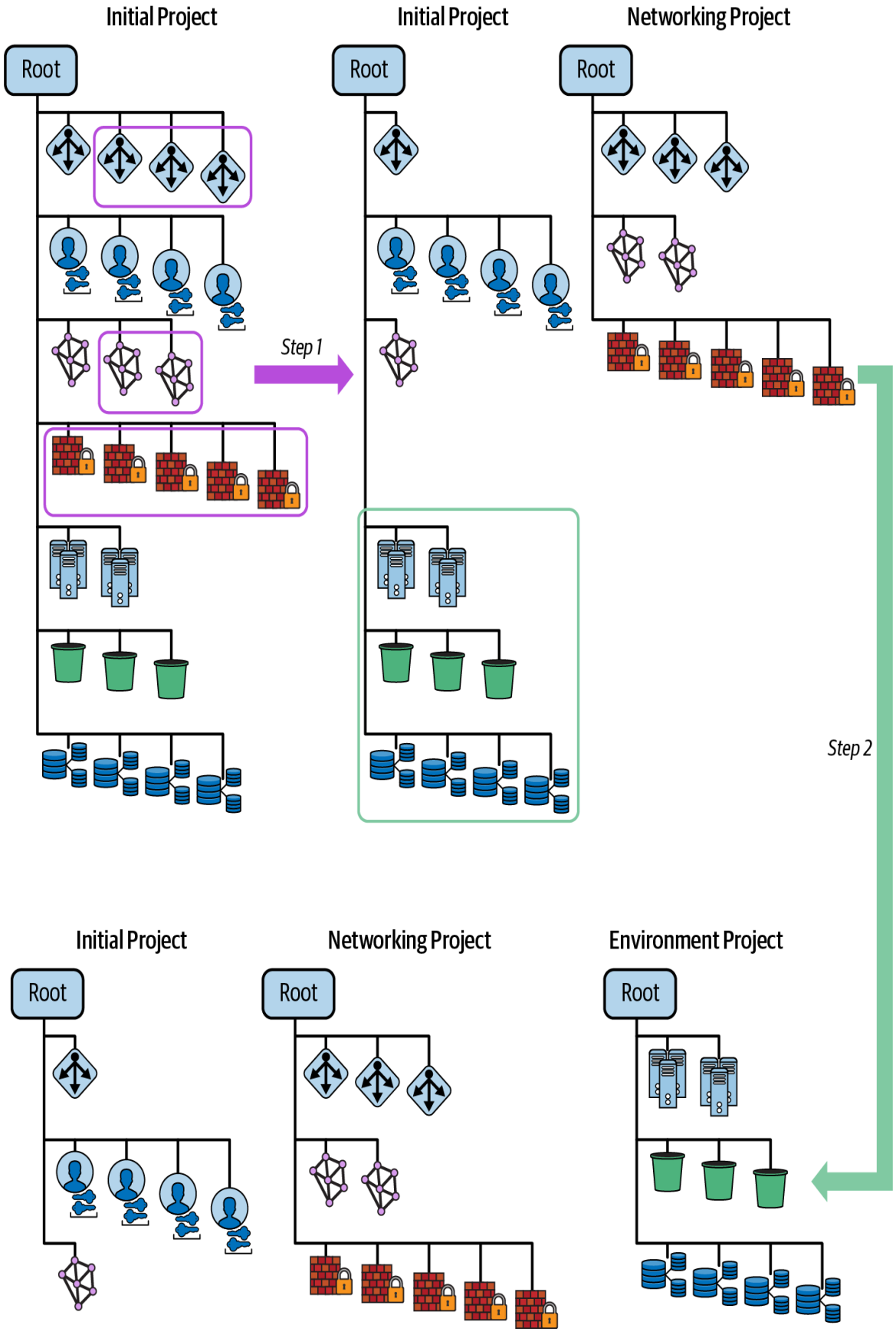

Initial Project
Initial Project
Networking Project
Root
Root
Root
Step 1
Step 2
Initial Project
Networking Project
Environment Project
Root
Root
Root

Figure 8-2. Strangler Fig pattern on a monolithic TF project, producing smaller TF projects

This is where the TF state can provide a great deal of assistance, especially when we're not able to actually construct additional resources, but are attempting to leave all the existing resources intact. Instead of using the new TF projects to create similar portions of the existing ecosystem, the new project can be used to directly operate the same resources by importing them into the new project's TF state. Then you may use TF operations like `tf plan` and `tf apply` to validate the new project against the existing resources, allowing you to achieve a high degree of confidence in the new solution.

State Reproduction Swap

If the changes to the existing projects are so extreme that none of the existing projects are expected to remain — something that might happen when completely reorganizing resources, such as when applying a logical domain hierarchy to a relatively flat and sprawling root module or as part of a merger or acquisition effort that sees both sides' entire codebases reconfigured to better coalesce — the best course of action may be to create a completely greenfield code setup to replace the current solution.

Just like in the Strangler Fig pattern, we rely on the TF state itself to determine changes to the system. This means the effort does not need to actually establish additional resources, you can simply initialize a new TF state (or set of TF states) and import the resources into this state. This is done by using the `tf import` command to establish a parallel set of backend locations for the TF states you're establishing with the newly organized TF code. This is typically cleaner to validate than in the Strangler Fig pattern, as it would typically be done once, altogether. The exact same interface, in terms of input values, may be validated to produce the exact same result in terms of the output up to the address TF assigns as part of its state tracking mechanism. Once this is established, it may be fixed as a basic acceptance criteria that allows operators to focus on the inner workings of the solution: the logical hierarchies, dependencies, and scaling

factors embedded within the new solution. Other key performance indicators (KPIs) can then be compared with the assurance that they do not compromise the integrity of the solution, meaning it may be possible to refine the solution, achieving more optimal outcomes. Figure 8-3 depicts two sets of backend locations and TF projects that encode the same exact cloud resources.

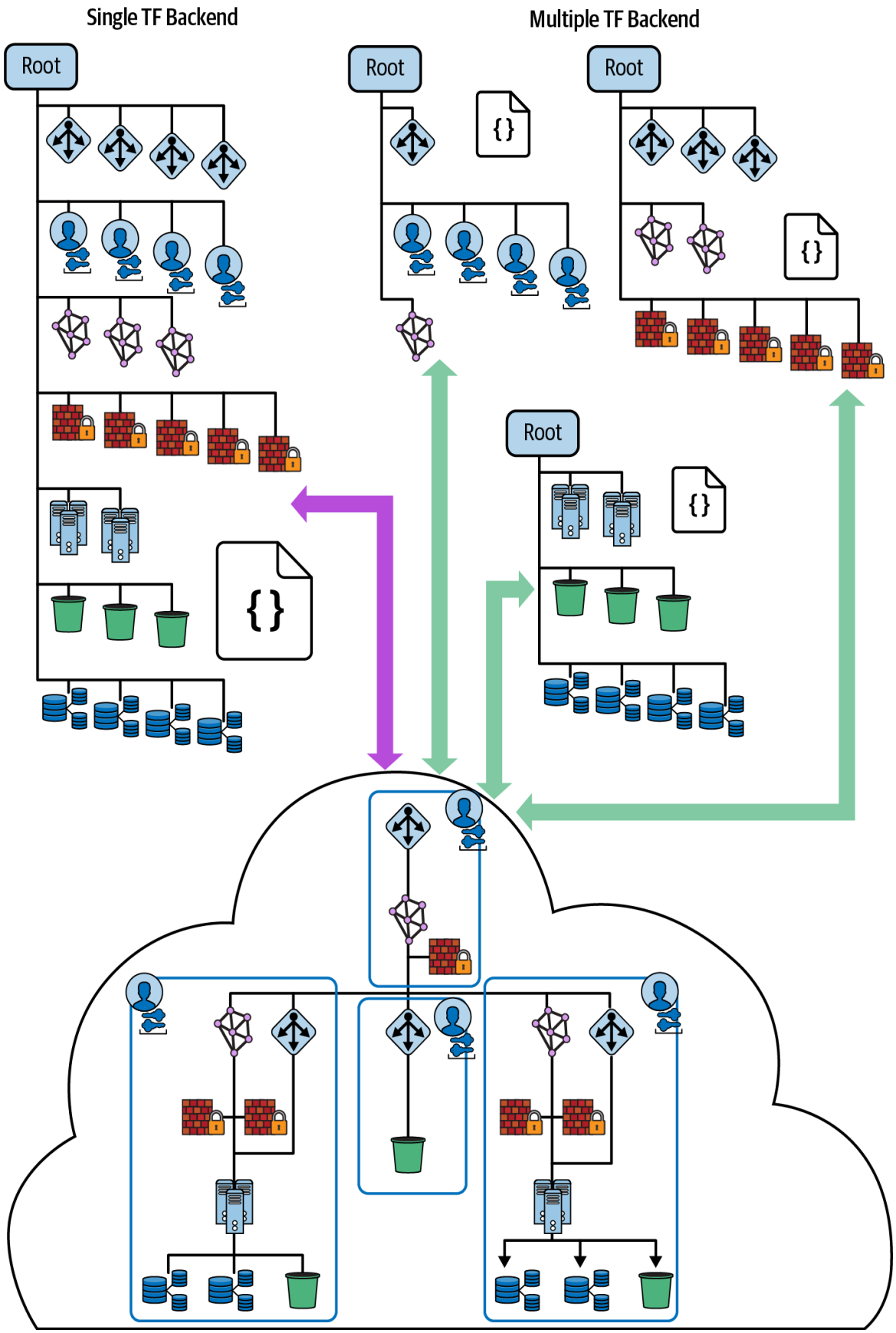

Single TF Backend
Multiple TF Backend
Root
Root
Root
Root
{}
{}
{}
{}

Figure 8-3. Two TF projects, and their corresponding backends, recording the same resources

To ensure there is nothing missing, it's important to establish metrics and targets, such as KPIs, that can be measured in both solutions to facilitate the validation of the resulting solution. The most obvious KPI is the solutions actually maintaining the same exact resources, or as close to the same as is deemed appropriate. Implementing the custom output values for all resources, ensuring the outputs match across projects, and using the `tf output` and `tf state list` commands can facilitate the comparison. Other indicators might be around how tight various dependencies are in the new solution; a very tight relationship may be preferred to establish rigidity and a deterministic ecosystem, whereas a looser relationship would allow for more autonomy between projects. Another set of factors relate to lifecycle management and access permissions. For example, unrelated things typically do not need to share a lifecycle, nor do they necessarily require the same set of access rights, so establishing goals around these can help to achieve high-quality outcomes. Some indirect, and somewhat subjective, indicators may include the end-to-end time it takes to process a change in the new system versus the old; although the measurements may be made highly accurately, interpreting their implications is often highly nuanced — in fact, it's typically based on a totally different set of metrics, capabilities, and policies inherited from the wider organization.

Special Cases: Mergers and Acquisitions

If every organization is unique, then it follows that every merger and every acquisition is also unique. This is further reinforced by observing the substantial presence of firms and individuals that specialize in shepherding such activities. Although there will undoubtedly be thorough audits of assets that would surely help elucidate the resources managed by both organizations, it's likely not done by the organizations themselves (at least, not by both at once), and thus there is likely to be insufficient information to determine the best approach for combining codebases. Often, the decision is made to have one system subsume the other, likely forcing the subsumed system to conform to the standards and policies of the

subsuming. However, ideally the effort to evaluate the systems still includes representatives from both systems. This ensures the effort has the benefit of all the institutional knowledge of both organizations.

> **NOTE**
>
> If an acquisition leaves one of the companies as an autonomous subsidiary, fully integrating the TF codebases is unnecessary. However, there may be certain portions of the subsidiary system that become redundant — such as network boundary security appliances — and may be removed from the subsidiary's codebase.

The first step to combining the codebases of two distinct organizations is to first establish an oversight team that combines members from each organization's architecture teams. The tactical plan for how to combine the TF projects is the primary output of the oversight team's efforts. Additionally, each organization's codebase should be subject to a potential reorganization before they are to be combined, aligning the codebases in terms of corresponding layers and domains. This can vastly de-risk the overall effort because it ensures things stay self-consistent before they are combined.

In some cases, the two organizations may primarily occupy different CSPs, and combining the codebases is not especially valuable. However, it's still worth evaluating both organization's TF projects and consider reorganizing them to better align and resemble each other. Similarly, the various teams should also be aligned so they can share approaches and techniques. It also is a way to expose any gaps in one of the organization's team makeup. That way, if the decision is made later to consolidate workloads and infrastructure components onto the same CSP, the components may also be readily combined, if that's desired and appropriate.

Assuming the decision is made to consolidate workloads and surrounding cloud ecosystems, including certain supporting appliances and platforms, into a single CSP, there are typically two scenarios: actually combining the two, cherry-picking the best techniques and tooling from each side, as

depicted in Figure 8-4, which preserves compute and data components but consolidates certain network components; and having one ecosystem effectively absorb the other. The latter case, with one ecosystem consuming the other, is the most common and straightforward: the assimilating workloads are reconfigured to work within the absorbing ecosystem. The former case, however, where elements from each ecosystem are included in the final product, can be extremely difficult to pull off if not done in a somewhat greenfield approach, where the new ecosystem is set up as a third ecosystem that combines the desired elements. In practice, it's more common to take something of a hybrid approach, designating one of the ecosystems as the foundation for the resulting ecosystem, including the workloads of the other, and slowly incorporating the tools deemed appropriate to bring along. You'll encounter example case studies and considerations for each of these at the end of this chapter.

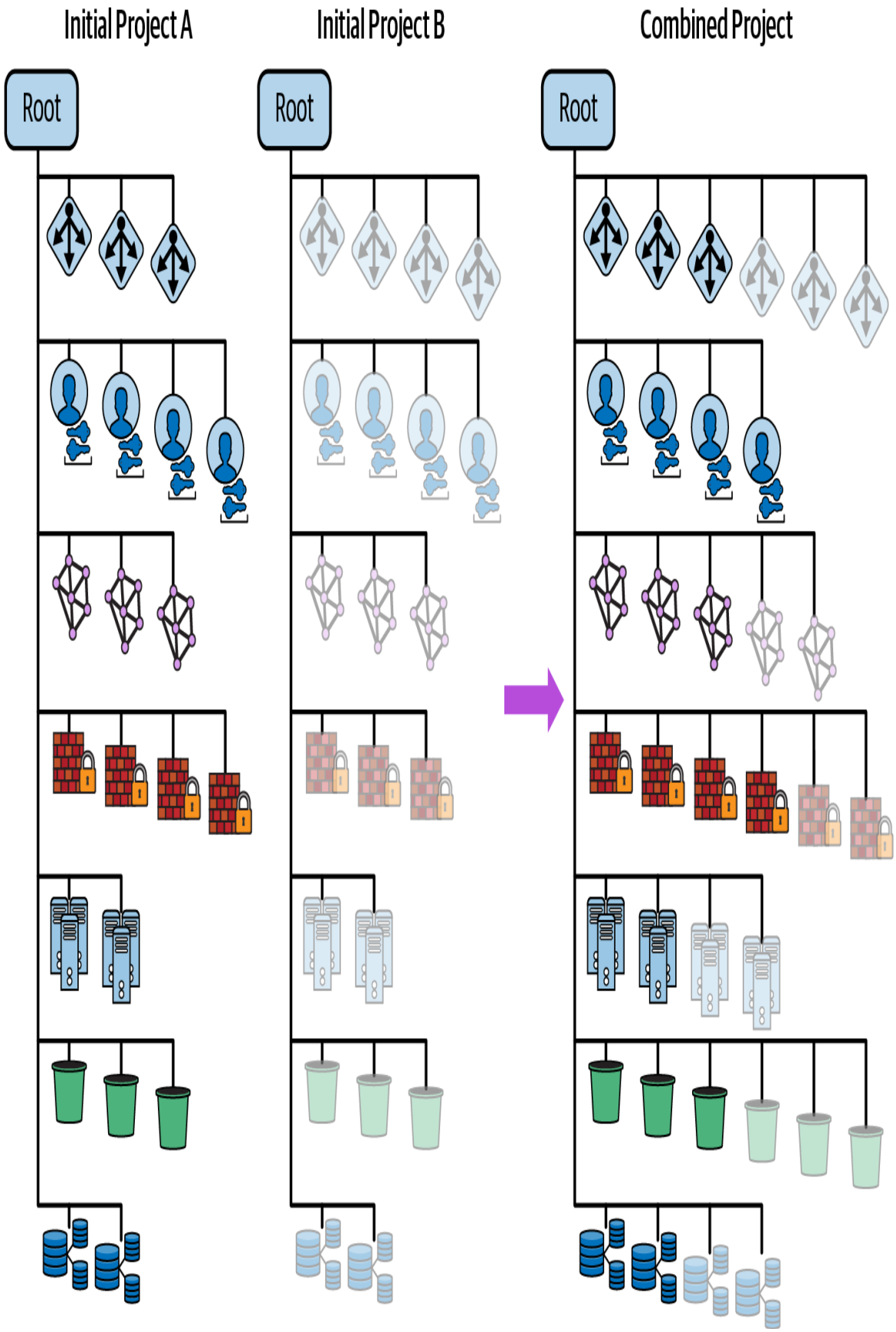

Initial Project A
Initial Project B
Combined Project
Root
Root
Root

Figure 8-4. One ecosystem consuming another

Once the organizations' cloud components are combined into the same sets of projects, it opens the opportunity to begin reducing the personnel that maintain them. This should not be done prematurely, before the effort is complete and any friction or rough edges are smoothed out. It should also be done slowly and methodically. Often, the most insidious issues are those that lie dormant for months and are deeply seeded in complex, esoteric institutional rationale that requires a variety of skills, experience, and exposure. Thus, especially early on, the redundant responsibilities can be highly beneficial to address any surprise issues that emerge over time. Instead of removing people who have been valuable to the team, consider investing in training to ensure the knowledge and experience remains within the institution. Don't just consider cross-training on technical skills, either. Also consider preparing individuals for management and leadership roles, including encouragement and support to pursue a business degree or other higher education.

Change Management for Infrastructure Support

As presented several times already in the chapter, a major piece of any effort to split up, combine, or reorganize TF projects is to reconfigure the support teams that maintain and operate the new structure. Too often, this is done haphazardly and without sufficient prior planning, leading to tension and friction, and ultimately impacting culture. Certainly, perfect planning doesn't guarantee perfect execution, but it's best to go into the effort of reorganizing your infrastructure management with the awareness that this will require a different team organization and makeup.

It's far beyond the scope of this book to present strategies for actually executing change management, but it's worth considering various end configurations. The following considers various typical approaches. A typical enterprise solution will generally combine elements of each of these to benefit from their advantages, and attempt to mitigate their drawbacks.

One common approach is a few centralized, domain-oriented teams that manage all the TF projects and cloud infrastructure. This would most likely be aligned to existing teams such as Information Security (InfoSec), NetSec, Development, Security, and Operations (DevSecOps), and Data Operations (DataOps) teams, either by embedding TF specialists as members of those existing teams or by setting up corresponding teams that interface directly with them. This works best for companies with relatively small technology departments, whose primary product is not digital and doesn't overly rely on a lot of commodity technology. It also works well for highly regulated industries, especially those with highly uniform regulations or those that serve a relatively small locality.

Another approach is to again focus around large centralized teams and to add project-oriented teams distributed throughout the organization to provide support. This would be most relevant to very large organizations that have engineering teams distributed across several physical sites. Each site could have one or more TF teams supporting the projects and products developed there. This works well for companies with large technology departments that aren't subject to a high degree of regulation. It can work especially well in cases where the level of regulation or sophistication is nonuniform across localities.

Yet another approach is to diminish the centralized teams (which even theoretically can hardly be eliminated) and to add more substantial regional support that has a high degree of influence and autonomy over their respective domains. Universal considerations such as the edges of internal networks and organization-wide policies can still be centralized — although it may not need to be a colocated team in a singular physical office — whereas operations that are much more regionally oriented, such as regional regulations or localized markets, can be located accordingly. Layered over this, it's still typical to have product-oriented support, with individuals embedded into product teams, directly or as part of teams that are dedicated to supporting the infrastructure for a product or series of products. This may work best for companies that have different operations in different global markets and a large, highly distributed engineering department, especially

those in industries that are subject to very different regulatory considerations depending on where they are in the world.

However you organize your teams, it's important to make sure the teams that are accountable for your infrastructure are also responsible for implementing it via TF. You don't want to be in a situation where the team that is accountable for the infrastructure isn't able to actually implement it. Instead, it's better to embed engineers responsible for the TF code directly into the technology domain teams or to construct teams that directly answer to the technology domain teams.

A final common approach is to develop a center of excellence. This can be a centralized group, but it can often be better served by tapping individuals throughout the organization that can collaborate on standards, techniques, and artifacts that simplify, accelerate, or reinforce successful software practices. Regardless of the composition, it's highly valuable to invest in purposeful collaborative exercises such as scheduled meetings or workshops, and dedicated documentation assets and accessible repositories. This requires nontrivial buy-in, since it likely includes regular internal conferences, with otherwise remote team members traveling to unite together and meet in solidarity.

Getting Outside Help

Taking the necessary measures to evolve your organization's TF projects and the teams that operate them requires specialized skills and certainly benefits from prior direct experience. This strongly points to leveraging outside help for the initiative. This can manifest in various forms such as oversight in the form of strategy or architecture consulting or as direct implementation via contracted teams.

Consultants may be used to make recommendations on tactical direction, systems design, or change management concerns. They should work closely with your internal teams — not as isolated silos — so they can both benefit from the institutional knowledge and begin conditioning your teams for adopting new policies and procedures. It's important to ensure internal

stakeholders have strong visibility and line of sight to the new ideas and procedures. This not only accelerates the spread and buy-in of internal stakeholders, it also avoids extensive discovery activities, presenting a mutually beneficial result.

When contracting entire teams for implementation, it's again recommended not to silo the effort. It can be highly valuable to include internal members onto the team for much the same reasons as keeping internal oversight of consulting efforts. It's certainly better to shuffle in internal team members into primarily third-party teams than it is to attempt to have totally separate teams, especially if either team has dependencies across the divide. This would only increase the risk of experiencing delays due to miscommunication or conflicting schedules, policies, or procedures. That being said, it's important to let the contract teams otherwise behave independently, with some insulation between internal stakeholders and the teams making the changes. They should have the autonomy to conduct the initiative largely on their own, with minimal direction. Introducing procedures around reporting progress and ensuring monitoring tools are sufficiently robust will guarantee the internal stakeholders are still properly informed and have the necessary line of sight to operations. The internal personnel embedded within these teams should be responsible for maintaining internal standards and procedures and ensuring policies are followed, as appropriate, as well as facilitating the documentation and messaging needed to spread the new assets throughout the organization.

Similar to the caveats presented in Chapter 7 around relying on unscrutinized AI agents to make material changes to your cloud infrastructure ecosystem, it's important to fully understand the result of their work. Although it need not require constant vigilance, you shouldn't accept their outcomes without scrutinizing both the code and the resulting resources their work manages. This review can be done regularly, or it can be done as part of the offboarding process. If done on a regular schedule, it should not be an impediment to the work. Instead, it should be treated as a certification exercise seeking acceptance of what the team has developed. Although it may seem more convenient to do this as part of a knowledge

transfer exercise, having a regular check-in can provide assurance that the teams are not veering off track or diverging in their approach or conventions.

Example Case Studies

Often, the best way to ensure the effort of evolving your infrastructure goes well is to already have experienced such an endeavor. This is not always possible, so the next best thing is to see examples and case studies of how this goes in practice. Too often, the public presentation of such case studies are sugar-coated, with only the most positive portions of the experience represented. Rather than cherry-picking a happy path, the following example exercises also present the points of friction and other less glamorous features that commonly arise.

Expanding from a Web App to an Entire Organization

Most companies that get to enterprise scale operate in a highly structured way with some nontrivial degree of hierarchy. Much earlier in a company's lifespan, it's likely to have more variation, with individuals executing multiple roles and holding multiple responsibilities to keep operations going. In several industries, including retail, professional services, and consumer packaged goods, companies can get pretty far with minimal digital assets, such as a website landing page and a point of sale (POS) system. These assets are also likely to be highly externalized, such as through SaaS products and professional services. If such a company were to see massive, rapid growth, to the point of becoming a globally recognized brand, at some point, it becomes economical to bring in some amount of technical skills in house.

> **NOTE**
>
> By the time a brand in one of these industries reaches global scale, it's typically through an acquisition by a larger, more established brand. This is primarily included as a thought experiment.

Suppose you started with a single website hosted on a website builder, but you now need specialized features such as highly available premium networking features or preparation to adhere to data compliance regulations like General Data Protection Regulation (GDPR) or Health Insurance Portability and Accountability Act (HIPAA) — or maybe you just need more control and wider oversight over changes you make to the site. You may consider recreating your website with a JavaScript framework like React or Angular and hosting it on a managed service like AWS Elastic Beanstalk or Google App Engine as a way to begin the process of establishing cloud hosting patterns that you can extend and more easily modernize. You'll then want to extract whatever information you've gathered in the website builder, such as users, profile data, and digital assets, and store them in suitable database or storage solutions. You can then begin to layer on additional managed services that unlock new capabilities. Done without sufficient planning, this will result in a correspondingly disorganized digital ecosystem. However, at a small enough stage, this infrastructure can still be sufficiently managed by a few folks, especially if you leverage IaC with TF, as shown in Example 8-1.

Example 8-1. TF code for a small-scale, web-based product (excerpt)

```
#------------------------------------------------------------------
-----
# Networks
#------------------------------------------------------------------
-----
resource "google_compute_network" "main" {
  name                    = "main-vpc"
  auto_create_subnetworks = false
}

resource "google_compute_subnetwork" "app" {
  name          = "app-subnet"
```

```
  region        = var.region
  network       = google_compute_network.main.id
  ip_cidr_range = var.subnet_cidr
}

# Serverless VPC Access Connector for App Engine -> SQL
resource "google_vpc_access_connector" "serverless" {
  name          = "app-serverless-conn"
  region        = var.region
  network       = google_compute_network.main.name
  ip_cidr_range = var.serverless_cidr
}

#-----------------------------------------------------------------
-----
# App Engine
#-----------------------------------------------------------------
-----
resource "google_app_engine_application" "app" {
  project     = var.project_id
  location_id = var.app_engine_region
}

#-----------------------------------------------------------------
-----
# Cloud SQL
#-----------------------------------------------------------------
-----
resource "google_sql_database_instance" "main" {
  name             = "main-db"
  database_version = "POSTGRES_15"
  region           = var.region

  settings {
    tier = "db-custom-1-3840"

    ip_configuration {
      ipv4_enabled    = false
      private_network = google_compute_network.main.id
    }
  }
}

resource "google_sql_database" "app" {
  name     = var.db_name
  instance = google_sql_database_instance.main.name
}
```

```
resource "google_sql_user" "app" {
  name     = var.db_user
  instance = google_sql_database_instance.main.name
  password = var.db_password
}
```

Without getting too deep into the details, the following is a high-level approach to how you might conduct this exercise:

1. Develop a copy website in software

Select a programming language and web framework, like JavaScript and React or Angular or Python and Flask, and build out an exact copy of the website. There are a number of tools and techniques that can help facilitate this, but it generally takes a small team of skilled frontend developers to rapidly build out a full website with the functionality available on a website builder.

2. Extract data and assets

Certain data may be exported in a structured format such as CSV or JSON. These would likely need to be transformed before the data can be loaded into a database solution. Assets such as images, videos, and other media may be stored in a managed storage solution, either directly or using compressed formats.

3. Create the cloud services in TF

Create all the cloud services necessary to host the website code, data, and media assets and to allow it to be safely exposed to the public internet in preparation for your new application workloads.

4. Import the data and assets

Upload all the data and assets to their corresponding solutions to prepare for use by the new application.

5. Automate the validation and delivery

Build, validate, and release your new website code onto the cloud services you prepared in step 3. This requires additional platforms or services to automate the process, which both accelerates the process and makes it more consistent.

6. Expand to additional environments with TF

A huge benefit to developing the website code yourself is that you can validate it more completely using other environments to ensure the product remains high quality and new features behave appropriately — and, importantly, doesn't include unintentional defects that would be encountered with perfectly normal use — before it reaches end users. Starting with the TF code you developed in step 3, you can modify it to interpret a different set of values for input variables and to produce a similar environment (likely with some notable differences in certain scaling or security features), as described in Chapter 7.

Once you have this, you can consider the effort a success. You now have a fairly robust solution with greater control and extensibility, and theoretically lower operating costs. You may never need to make significant updates, beyond updating framework components or implementing new styling or other cosmetic features. Alternatively, you may have used this effort as an initial step to a much larger expansion program with an extensive list of features to introduce. Similarly, this may have been a necessary step if your business is rapidly scaling, and your existing solution is running into limitations that may be unacceptable or too costly to overcome directly. In such cases, you'll likely want to rebuild your solution using even more scalable and extensible cloud services and features. For example, when you want to implement new features in your website, you may choose to build the new functionality as externalized services that run in the cloud and are accessed using an API. This can provide greater levels of sophistication in terms of where data is stored and how secure the mechanism of accessing the data is. In a similar fashion, you may choose to offload certain functionality initially handled on the client side, within the user's web browser, onto API-exposed services hosted in the CSP. These

services may be running on VMs, within containers, or hosted in a serverless solution such as AWS Lambda or GCP Cloud Run Functions, and they may have their own evolutionary journey over time.

> **TIP**
>
> If the website you're using doesn't provide you with the software components necessary to host the product yourself, you can quickly generate a copy of your website by issuing a sufficiently detailed prompt to a code generation tool that instructs that tool to replicate the style and functionality of the existing website. The initially generated project will likely need to be cleaned up, either manually or through a series of subsequent prompts, by an experienced engineer before it can be put into service. In this case, you would not allow the resulting software to be hosted by the tool itself, but by the components you're building in the CSP.

Preparing this for the next level, with multiple environments your development teams use to validate their work as it prepares for production and regional replicas of entire stacks of infrastructure, means first segmenting your infrastructure into its respective domains and then splitting up the resources into separate TF projects along those domains. You can also begin to use the same components to expand to multiple environments, as depicted in Example 8-2. Certain portions can also be replicated as separate, isolated environments, allowing for a high-quality software ecosystem on which to conduct an industry standard software delivery lifecycle.

Example 8-2. Expanding TF project to create additional environments

```
#-------------------------------------------------------------------
-----
# Example execution: tf plan -var-file=environments/dev.tfvars
# (Possibly after setting up backend with tf init)
#-------------------------------------------------------------------
-----
#
////////////////////////////////////////////////////////////////////
////////
# dev.tfvars
#
////////////////////////////////////////////////////////////////////
```

```
////////
project_id      = "tas-8-2-dev"
subnet_cidr     = "10.10.0.0/16"
serverless_cidr = "10.8.0.0/28"
db_name         = "appdb-dev"

#-----------------------------------------------------------------
-----
# Example execution: tf plan -var-file=environments/prod.tfvars
# (Possibly after setting up backend with tf init)
#-----------------------------------------------------------------
-----
#
////////////////////////////////////////////////////////////////
////////
# prod.tfvars
#
////////////////////////////////////////////////////////////////
////////
project_id      = "tas-8-2-prod"
subnet_cidr     = "10.10.0.0/16"  # CIDRs can be the same in dev
and prod
serverless_cidr = "10.8.0.0/28"
db_name         = "appdb-prod"
```

Working with a Vibe-Coded Project

A relatively infant software development paradigm that is ballooning in popularity is *vibe coding*: using a code generation tool that automatically pushes the code it produces to a designated branch in a source control system. Features are built by giving descriptions, in plain language and with as granular detail as you can provide, to the generator. Its output will typically include instructions for how to bundle, validate, and publish the produced code. Defects are addressed by feeding the error messages or a detailed description of the issue to the system, possibly with additional context or a directive for how the generator should present the output. It greatly accelerates the development of software, reducing what might otherwise take a small team of developers with a mix of skills a day, a week, or more to draft down to a minute or less — and sometimes more, but on the order of a minute nonetheless.

> **NOTE**
>
> For students of the Pareto Principle, it gets warped with the vibe-coding paradigm: 80% or more of the work can be completed in approximately 1% of the overall time, which is indeed remarkable. However, this implies that the remaining 20% or less will take 99% of the time! This suggests using code generators in rapid prototyping scenarios or in making large, sweeping changes, and not relying on it as the only means of authoring code or troubleshooting issues. Not to mention the fact that you're typically allotted only a certain number of tokens in a month, for example, with which to make changes through the code generator, meaning small or simple changes, such as getting pixel-perfect styling or reorganizing markup, may be best done by hand, if possible.

Theoretically, an approach based on vibe coding could be plugged into many typical collaboration strategies directly in only a few steps, and can even be controlled to protect main branches, shipped versions, and so on. However, in practice, and in the true spirit of the paradigm, it's customary to simply trust the output of the generator. With a sufficiently sophisticated software delivery system, generated code can be validated to a high degree without the need for heavy scrutiny over the code. However, this does not preclude the value of oversight by an expert engineer. Although these code generators are extremely powerful tools to accelerate the coding process, doing so without also reviewing the results of changes with some level of rigor is at best risky, at least operationally. An appropriate level of rigor may be overlayed through a pull request (PR) process, ideally after big chunks of changes are not only functionally and cosmetically validated, but also refactored for clarity, extensibility, and easy upkeep. After all, you may not have access to the same code generator forever, either from a licensing or security posture standpoint, so ensuring that the code that is produced is intelligible to humans is extremely important.

Especially in application development, such as for a program on your phone or computer, the organization and complexity — that is, the number of dependencies, interdependencies, and the presence of any cyclical logic — are top drivers of architecture design. Much of this type of software has to be preprocessed or compiled, producing a binary that your device's OS is able to interpret and execute. This involves translating the human-friendly

code presentation into an efficient, OS-ready representation — the *binary* — which sometimes leads to a disconnect between what you think your logic does and what it actually does at runtime. Abstracting further by not even manipulating the code yourself and rather having a code generator manage it means there is an even lower degree of awareness of the runtime behavior.

Luckily, in the TF model, TF itself *is* the OS binary, and it's merely interpreting the highly structured input you present it at the time of invocation, meaning that as long as your code looks consistent, it will likely be interpreted effectively by TF. Moreover, this suggests a relatively low risk to having a code generator drive the coding process. However, unless the generator heavily leverages data sources for dependencies and you ensure it either has a complete list of existing resources or sufficient guidance in naming conventions for identification-oriented fields, there's no guarantee that resources that would be initiated with generated TF code will not clash with existing resources. This suggests the majority of scrutiny should be shifted to reviewing the changes described in the TF planning phase. This makes it important that all the fields that could produce problems if duplicated aren't computed during the apply phase but are available during the planning phase. That way, they can be validated against your existing resources to ensure uniqueness. In short, it still is best to employ humans with strong TF skills to operate the project.

Code generators can even be made to impose the constraints and suggestions you have encountered in this book. Using well-structured directions, you can ensure that any logical calculations are made in the `locals{}` block and all of the important information fields are present in the module outputs, for example. If your tool supports a system prompt, or a similar static directive, you can impose constraints such as restricting logic or excessive string interpolation, inline, as the value of a resource attribute; disallowing untyped variables; or requiring validation on complex variable types. With a high degree of existing sophistication possessed by human operators, code generators can be an extremely valuable tool for expanding, polishing, and maintaining TF projects.

What's more relevant to the purpose of this text is the experience of evolving an application project, as described earlier, that was effectively vibe coded into one that utilizes the latest software delivery practices. This might be motivated by a desire to delegate more and more portions of the application to other members of your organization. More than two or three individuals attempting to develop the same vibe-coded app may have challenges if they're working on the same repository at the same time. If you're at the point of organizing multiple teams, you should really be splitting the project up into independently developed components, anyway. This implies a need for validating the integration of these components, which further suggests an environment — separate from the one your customers interact with — in which to validate such integrations. Doing this without an IaC tool like TF, even with the help of code generators and other AI-powered facilitation, is an exercise in tedium ad infinitum.

Let's say your app was developed with a service that not only supports the generation of code, but also hosts portions of your application, such as the website and data services. Perhaps you want to introduce complex, server-side logic or have full ownership of the data you store and collect. Some portions of the application may be leveraged directly, or with minor modifications or additions, such as the website code. However, they'll need to be hosted by some other service such as AWS Elastic Beanstalk or GCP App Engine, or possibly as a single-page application (SPA) hosted on a storage bucket and served by a content delivery network (CDN). Whether or not you intend to rehome the data, the new tables need to be created and configured in the CSP, most likely with additional considerations such as database users, in the case of MySQL, Postgres, and similar database solutions, and resource considerations such as instance size or replication factors.

As a starting point, it may be most prudent to first stand up a direct replacement, without fully leveraging all the high availability features and the various multiple environments that you intend to build. Once your workloads are transitioned over to a fully cloud-hosted configuration, you can remove the reliance on the code generator for hosting your application.

You can also begin to expand the solution, leveraging additional features of the CSP-managed resources, such as considerations for resilience, high availability, and disaster recovery. Additionally, you can begin to replicate the entire collection of cloud service instances that supports your customer-facing workloads as validation environments in which you confirm things like effective interaction between integrated components or maintaining, if not exceeding, performance benchmarks.

It's even possible to do all this with a code generator helping you write the TF code. Each stage can be developed with a well-constructed prompt, or series of prompts, that creates the infrastructure, workloads, and automation configurations at each step of the evolutionary journey. Although this can be a great way to speed up certain aspects of the development process, the speed should not be overemphasized. It's still worthwhile to be deliberate and thorough with the validation of each major step, even if it slows things down to a less impressive pace. It's better to be slow and steady than to rush through the journey and end up with something everyone has to wrap their heads around, or that leaves your team members or leaders scratching their heads. That's not to say you can't make big changes rapidly, just that you should give the organization time to adjust to the changes.

What follows is a potential strategy for fully migrating from a code generator and website-hosting platform to a cloud environment and evolving it to a world class software delivery solution complete with infrastructure management. It assumes the generated code is recorded in a Git repository. For brevity, it primarily includes high-level descriptions, making the assumption that you have the skills necessary to implement each step within your organization — or that you're willing to continue to trust generated code and strategies. I've provided extended commentary where applicable:

1. Retrieve the codebase from Git

> You may need to clone the repository if you've never accessed it before, or you may simply need to update to the latest commit on the main

branch. You'll need to authenticate with the remote Git repository in some way, which may be enforced as a standard by your organization.

2. Confirm the application launches locally

Execute the instructions for instantiating the app locally, such as building/compiling the runtime artifacts and running them in a local runtime. If the build and local run steps aren't available in the *README.md* or obvious to you from the structure or contents of the repository, you can generally ask your code generator to provide them.

3. Determine data model

This would likely be clarified by reviewing the code that retrieves or stores the data in the application. Most likely, the data is structured either to prepare it for sending along with a request or when it's extracted from a response. Although this won't directly amount to a full table schema or the primary and secondary keys, it would give the fields of each datapoint, which in many cases will be enough to make strong inferences about what the structure should be, especially for a seasoned data specialist. You likely can even get the code generator to directly produce a data model that would work for popular solutions such as MySQL, MongoDB, and so on, many of which are directly accepted by CSP-managed solutions.

4. Select data solutions

Depending on the data model you determined in Step 3, you may have to select a variety of data solutions, such as relational, NoSQL, and so on. In the case of most web-hosted applications, it would be theoretically possible to use a single type of solution. However, there may be availability, cost, or speed considerations for which a single database solution may not be able to account.

5. Create the foundational cloud services with TF

The first step in your CSP will be to set up some of the normal foundational components such as baseline access controls, logging and monitoring considerations, and organizational components — such as GCP's Project and Folder or Azure's Resource Group constructs — for shared support services. You'll also need to generate root-level access controls to use with TF itself. Depending on how you're configuring the TF automation, you may need to stitch these components back into the automation system.

6. Create the supporting cloud services with TF

Unless you choose to use external SaaS tools to manage your software delivery lifecycle, you'll need to establish infrastructure and services for this, too. There may also be certain metrics dashboards available in the current hosting solution that you want to recreate in the CSP's monitoring services. These components would typically be shared among all workloads and are typically built on top of the foundational components, alongside the application-specific infrastructure.

7. Create the main cloud services with TF

Before you can launch your application in the CSP, you need to establish the underlying infrastructure components for your application, such as compute resources, data solutions, and various network components. This can come in various forms and configurations, but assuming the simplest configuration, it typically amounts to a single service for compute, a database, an optional storage bucket, and a single subnet in a VPC. Additionally, there will undoubtedly be other network components such as firewalls, DNS records, and traffic routing. There should also be certain application-specific access controls, including service identities such as AWS service roles and GCP service accounts.

8. Establish static data/schemas/keys

Depending on the data solution(s) you selected in Step 4, you'll need to establish the various structural components such as table schemas,

primary and secondary keys, and static reference data to prepare for your application workloads.

9. Migrate data

Unless you intend to totally reset the application with no users, session data, or records of any kind, you'll also almost certainly need to migrate at least some of the data, such as users' profiles, account data, and historical records. How you migrate depends on a plethora of factors, not the least of which is how much distinct data you intend to move.

10. Establish software delivery automation

Once you have the infrastructure on which your application workloads will run, you can establish mechanisms for deploying the workloads. The goal is to set up the basic framework, access controls, and organizational considerations, and the solution may see multiple updates as you begin to run the subsequent couple of steps to actually deploy and validate the solution.

11. Deploy the application to the service(s)

You now have the foundation on which to host your application workloads and the means with which to launch them. You'll need to build the artifacts and deploy them using the software delivery automation. These steps again should be available in the *README.md* and may again be requested from the code generator, as needed.

12. Confirm cloud solution works

Once the solution is deployed, you should validate that it works. This can range greatly in both sophistication and comprehensiveness.

13. Update DNS to switch traffic to cloud solution

Reconfigure DNS to ensure it's serving the new solution at the appropriate domain name. Assuming you already have a domain name registered somewhere outside of your CSP, this almost certainly

includes redirecting name servers to a new cloud-hosted public DNS zone, along with building out the A, CNAME, and other records.

14. *Replicate the environment to support graduated software delivery*

    Now that you have replaced the code generator-hosted solution with a cloud solution and automated delivery procedures, you can begin to extend the solution by incorporating representative replicas of your production environment — which may be scaled down, or include dummy data, depending on the purpose of the environment. Although with the right approach and components, new environments can be added as needed, it's prudent to establish at least two additional environments: one for quickly validating functional or behavioral changes, which might be shared by multiple team members and incorporate multiple works-in-progress, and which might break or behave unexpectedly at any time; and another that closely resembles the true production and is meant to validate nonfunctional aspects of working code.

15. *Expand software delivery automation to other environments*

    Create additional automated procedures for deploying the workloads' software components to the new environments. Ideally, you'll not rebuild the runtime components for each environment, removing that step from the pipelines that service the production environment, and only implementing in the earliest stage (or stages).

16. *Implement graduated software delivery*

    Once you have automated delivery pipelines for each environment, you can stitch them together to create a full, end-to-end graduated software delivery workflow.

> **TIP**
>
> Determining the optimal database or storage solutions is beyond the scope of this text, but there are a few traditional hallmarks of various solutions. For example, if the data you get back is all very flat and the app pieces it together based on a common identifier, you are working with *relational data*, which could be placed into a relational database solution like MySQL, PostgreSQL, or a service like Google Cloud SQL. If the data you get back has a hierarchical structure, perhaps encoded as a JSON object, you can typically use a NoSQL database directly — an alternative is to build some middleware to *denormalize* or deconstruct this data into flat structures and again use a relational database, decomposing or compiling the hierarchical structures leveraged on the frontend. Big blobs of data that are retrieved with a single key, such as an image, video, or other media, suggests the need for a document store solution such as a managed storage bucket. This makes no claim as to availability, cost, or speed of such solutions, but is simply meant to provide a direct alternative to a SaaS-hosted solution from purely an interface standpoint.

When you begin to build your cloud infrastructure with TF, you should have good insight into what components you'll need. You may not know exactly how many environments you need, but you recognize the need for multiple environments to support software delivery. Given that, it's worth targeting a reasonably sophisticated arrangement, starting with core, foundational elements. This should include considerations of hierarchical or scoped services, baseline security controls, and resource naming conventions. In this scenario, you're porting a single application made up of several components, and you may have no intention of building anything else; regardless, you should at least evaluate how you would extend your ecosystem to support additional products. Moreover, at some point you'll likely want to develop cloud infrastructure for internally managed SaaS solutions, such as collaboration services, automation platforms, or HR tech solutions, and these should be easily distinguishable, if not effectively isolated, from your product's infrastructure and workloads.

These initial components are used to support the development of all the other components you will build. They rely on root-permissions assigned to a user or service identity that will act as the operator of these foundational elements. If you're using an external SaaS solution, you may be able to take a nigh purely TF-driven approach, save this root permission. If, however,

you'll be developing your infrastructure automation system within your cloud ecosystem, you will probably need to instantiate these components manually. In this case, as soon as you have this system, configure it to manage these root components, in addition to the management of the infrastructure for your product and shared services. Either way, you may also consider importing the root permission configurations into your foundational TF project, bringing it full circle.

This is only one way to structure such an exercise, but there are many variations on the theme. For example, although Steps 5 through 7 are presented as distinct, they can certainly be developed together as part of a coordinated effort. Similarly, an alternative ordering could place Step 7 after Step 10, for example, if the software delivery automation will be handled external to the CSP. Also, Step 5 assumes this is a new organization, and not a skunkworks side project embedded within an established company that has an established set of foundational cloud components. Assuming you are indeed carrying out Step 5, you may choose to include additional services that protect the surface layer of your cloud ecosystem, namely public-facing network security controls; certain components and features of network traffic inspection appliances may be taken for granted if they were included by a third-party hosting solution.

There are also numerous choices in terms of what environments you should build. The canonical trilogy of environments, the classic *dev*, *test*, *prod*, suggests an environment for developing new features or bug fixes, one to test them against a battery of predefined validation tests, and the one that customers use. The strategy presented in the preceding steps also suggests three environments in total, but slightly repurposes the *test* environment to one that might commonly be called *staging*: an environment that appears to be a copy of production, but that is not available to end users, where controlled validation of behavior and appearance can be conducted. You may find value in both a test and a staging environment, with the purpose of the *test* environment to validate functionality, leaving the later *staging* environment to be primarily focused on the look, feel, and sprawl of production.

However, this may still not be enough. You may find that you're performing multiple, distinct validation exercises within the same environment, such as ensuring a production-like system can operate under simulated production-like load, or running a so-called *chaos test*, where portions of the system are randomly corrupted or deleted to validate both the behavior during the experience and the recovery process. Rather than doing this all in the same environment, you may find it is less confusing to set up dedicated environments for these exercises. You may also find that a single environment in which to develop all the various components is too disorderly, with multiple changes causing whatever issues that arise to be obscured by the simultaneous efforts, and you create multiple development environments, one for each workstream to use however they see fit — especially to fix certain dependent components to clarify behavior of the features under development — and an additional *integration* environment in which all the various workstreams merge together — once they have reached a sufficient level of stability and completeness. Since you already recognize you will have multiple environments, and that they need to resemble each other as much as possible (up to known and deliberate limitations and relaxations), you should construct them in such a way that extending to any number of additional environments becomes almost effortless. Many of the nonproduction environments do not need to be available 24/7, and thus you may even be able to tear them down (or partially down) when they're not in use, such as in the case of ephemeral or on-demand environments. See Chapter 7 for in-depth guidance and strategies for introducing additional static and ephemeral environments.

> **NOTE**
>
> I believe by now, in 2026, testing one's application is so commonplace that the discussion is never why one needs tests, but what one needs to test and how to go about conducting the tests objectively. Leaving out testing from the steps listed is not intended to diminish the value or to imply that tests are unnecessary. However, in the scenario that was presented, in which a product was vibe coded and you're developing enterprise-grade guardrails, it's unlikely that a full suite of tests was already developed. Since the goal was primarily to replicate a hosting environment, to unlock yourself from the overhead of a fully SaaS hosting solution, the steps included were intended to produce a relatively like-for-like result. Certainly, the action of including tests can be done before, during, or even after the course of these steps.

In terms of automating the delivery of software to each environment, the preceding steps suggest creating these automations in more or less isolation and chaining them together at the end. This is mainly because the steps also create the production environment before the preproduction ones, and it therefore already assumes updates are made to a pipeline along the way. Certainly, a more direct approach could be made, developing the environments in a different order and having each subsequent environment's pipeline pick up from the result of the previous. Either way, if you're comfortable taking a more gradual approach, or you're highly sensitive to de-risking the over exercise, you may choose to delay Step 13 until after all the environments are in place and at least superficially validated.

Given this commentary regarding the introduction of additional environments, perhaps at a later date it's valuable to consider each environment's pipeline as a self-contained unit, either as composable collections of environment-specific steps or as totally separate pipelines/workflows that either watch and wait for upstream procedures to complete or trigger downstream procedures. In the case of ephemeral environments, such as for an infrequent chaos testing exercise, it's likely desirable to somehow embed the infrastructure automation procedure within the overall chaos testing validation process. If infrastructure automation and software delivery is managed by distinct systems, there may be additional configuration or components that need to be in place to integrate them.

As mentioned, the approach outlined assumes a single customer-facing application, but you should consider how you can structure your solutions to support extending to other external or internal-facing products. This is usually made up of a number of techniques and in particular should include the management of any application or workload-specific identities and permissions. Specifically, you should first examine a way to construct fit-to-purpose service identities that operate different portions of the cloud ecosystem: one (or a set) that operates the infrastructure automation; a set that delivers the software to specific environments — possibly grouping certain pre-production environments — and that are distinct among products — or possibly distinct among business units or other scopings; and another set that acts on behalf of the workload services themselves. The architecture and procedures here generally require a formal approach and should be heavily guided by cybersecurity professionals, either added ad hoc or only with canonical entities.

As you can see, the experience of porting a generated application from out of the code generator's hosting platform is a nontrivial exercise, to say the least, possibly taking multiple sprints-worth of work performed by experienced individuals. You recognize it can be greatly expedited, and in some cases informed, by further using a code generator or other AI-powered tool, but that it would still need highly competent operators to ensure the efficacy of the resulting software.

> **NOTE**
>
> One thing to point out with vibe coding is that it presents a different set of considerations for so-called zero-day exploits. On the one hand, they have been proven to actually be able to find them, which has both positive and negative implications. On the other hand, by design they require data to train on; if they don't have relevant data — perhaps due to intellectual property constraints — they may not have a chance to detect them in the first place. Certainly, once the exploit is found, a language model can help address it and patch it once a fix is discovered.

Absorbing an Acquired Software Ecosystem

In a typical merger or acquisition effort, two or more companies combine to make a single company. This is never a trivial nor singular effort, but one that requires multiple activities happening in concert and in succession in a loosely ordered sequence. Especially in an acquisition, the canonical scenario is a larger company absorbing a smaller company, with the smaller company conforming to the processes and policies of the larger — at least, for the most part, and up to legal compliance and so on. Extending this, the expectation might be that the larger company has the larger, and heavier, software ecosystem and the smaller company has the smaller, lighter-weight ecosystem. In such a scenario, with the bigger company's processes and policies dominating the path forward and the larger software ecosystem with more inertia, conventional wisdom would suggest the smaller ecosystem should be likewise absorbed into the larger.

A more enlightened and long-lived approach might be to evaluate each ecosystem and conform to the one with the most mature and modern approach. Although it's not exactly unusual for such impactful decisions that have long-lasting implications to be made with a focus on more immediate goals and to take a less enlightened approach, an acquisition is already so complicated it's worth considering a somewhat happy path and recognize that many scenarios are variations on the same theme, simply with more caveats and proverbial hoops to jump through. Regardless of which codebase is absorbing which, the general steps are the same, so it's reasonable to assume a baseline approach, rather than considering something with even more complexity.

Suppose Company A is absorbing Company B and has the larger codebase, more modern ecosystem, and more mature processes and platforms. This simplifies the situation significantly, as the most logical approach would indeed be for Company A's ecosystem to absorb Company B's. Although there's always friction and often resentment that comes with any such drastic conformance forced upon a team, this is an all but predestined outcome, so fighting against it is likely to be an effort in futility. As such, it's worth assuming the friction in this scenario is minimal and is actively mitigated. Suppose also that Company A uses a container-based platform

for running its workloads. Even if Company B doesn't, this would greatly simplify the process of rehosting Company B's workloads, since whatever OS it runs its workloads on can be constructed as a container — some systems can even be made to manage VMs or operate on bare metal machines. Finally, let's add the challenge of the two companies operating in different CSPs — but let's assume only a single CSP per company, regardless of whether this could have a strategic advantage. Finally, assume that the effort of integrating the new version of Company B's workloads and components into Company A's ecosystem will be considered workload by workload, as discrete efforts, be they implemented in series or parallel.

As you learned in the commentary in "Special Cases: Mergers and Acquisitions", there likely will be at least a short time in which the two companies will continue to operate immediately after the acquisition, but this should only be long enough to allow for a thorough discovery exercise to determine the best course of action. If the companies choose to stay separate long-term, it means unnecessarily duplicated resources and licensing, especially at the system's foundation and edge. It also likely means duplicate investment in software development patterns, certain tooling, and skill development. Especially with two different CSPs hosting distinct and mutually exclusive workloads, you're adding effort and cost. First, you may be losing out on volume-based discounts, and second, your organization has to stay up-to-date with two providers both in skills of the workforce and in adoption of new features or services. Finally, it keeps your engineering teams separate, which can have negative effects on the culture of your organization if left unchecked. The sooner you're able to combine the ecosystems, the sooner you can benefit both fiscally and culturally, soothing the pain you may face in the short term.

> **NOTE**
>
> While psychological and sociological theory and its implications are far beyond the scope of this book, it doesn't necessarily take a PhD to at least recognize when teams begin to feel neglected, overlooked, or deprioritized relative to other teams. However, it does take awareness and objectivity — which can make it hard to recognize if you're part of one of these teams, and certainly if you're one of the ones sharing the feeling.

When it comes time to combine the software ecosystems, the first thing to recognize is that it's not something that can be done by decree. Like any digital transformation, it takes time both to implement the integrations and to let them stabilize, possibly in a phased or graduated way. The following is a recommended approach to the situation that's been set up between Companies A (the absorber) and B (the assimilated):

1. Proof of concept (Optional)

Conduct a proof of concept (POC) to trial the effort of converting Company B's assets and to expose as many unknown snags or points of contention that might emerge as possible.

2. Construct infrastructure

Construct the infrastructure components necessary for the first round of fully production-ready workloads.

3. Pilot happy path

Select a combination of team and workloads from Company B that are considered the easiest to convert and fully onboard them into Company A's ecosystem. Record what worked and what didn't, and produce guidance and assets with which to accelerate the subsequent rollout.

4. Roll out happy path

Identify all other team-workload combinations that resemble that of the pilot and conduct the procedure refined from the pilot exercise.

5. Assess rollout

Once the rollout exercise is complete, review and analyze the experience to confirm success and record any additional learnings from the experience of the rollout.

- Deprecate old solution. As part of the assessment, officially deprecate the old solution through a variety of communication channels.

- Tear down old solution.Once the previous solution has been deprecated for an appropriate amount of time, the solution should be fully dismantled.

6. Construct additional infrastructure (Optional)

Construct whatever infrastructure is necessary for the subsequent classes of workloads that must be incorporated.

7. Pilot harder migration (Optional)

Select another of the remaining team-workload combinations that falls into a more complex solution category to conduct another pilot of the next wave of workloads to be onboarded.

8. Roll out harder migration

Once again, identify all other team-workload combinations that resemble that of the pilot (or that fall into a common group, as suggested by the optional pilot step) and conduct the procedure identified in the pilot exercise.

9. Assess latest rollout

Again, review and analyze the completed rollout exercise, deprecating and tearing down the old solution, as appropriate.

10. Repeat Steps 6 through 9 (Optional)

Assuming there are more Company B teams and workloads, repeat Steps 6 through 9.

11. *Stabilize before extending further*

    Stabilize before making additional significant changes, such as introducing a new foundational data analysis solution.

First, you should consider starting with a POC exercise in which a member or a small cadre from Company A embeds (perhaps in a part-time capacity) into a team of Company B — say, Team B — and guides them through the experience of rehosting, retrofitting, or converting Team B's digital assets to fit into Company A's development procedures, qualitative policies, and automated processes. The goal at this stage is not perfection, but to shake out as many of the impediments and pain points as you can as quickly as possible. The focus here will be on the infrastructure and other aspects of software delivery, and preferably not require any major changes to the workload itself. Ideally, you can select a workload that is extremely stable, one that doesn't need new features or require a high frequency of updates, such as in its list of dependencies. This is optimally done in isolation from Company A's existing software ecosystem, with an expectation that it will be promptly torn down once the exercise is complete. You probably won't hit all the problems that will arise in the full integration of Team B's workloads into Company A's ecosystem, but you'll significantly de-risk that effort.

NOTE

A piece of advice worth passing along is that once something is built in the cloud, it has the tendency to stick around. Given the general urgency that leadership typically has in moving on from initiatives that don't directly improve the bottom line, it can be too easy to forget about a resource that was instantiated for the purpose of validating a managed service. For this reason, it's wise to include a tear-down step as part of the ultimate acceptance criteria of the initiative.

Before the pilot is initiated, the necessary infrastructure components for the pilot workload and workloads like it must be created. This can be done in a layered and graduated fashion, starting with security considerations such as access controls for team members and other entities, necessary network controls, and other shared or foundational infrastructure components. The subsequent software delivery environments portion of this step can even somewhat overlap the pilot effort, allowing the piloted workload to be validated on an early preproduction environment before later environments are constructed. This way, any unforeseen differences in behavior due to environmental incompatibilities in the new environment can be detected early.

> **NOTE**
>
> Although it may be tempting to construct all the infrastructure you expect for the entirety of Company B's workloads that will be incorporated into Company A's ecosystem in one go, it may not be worth it. It's highly unlikely that all of Company B's workloads will remain static throughout the effort. In particular, the longer the overall effort to combine the companies' ecosystems takes, the higher the chance for major changes to be necessary in later workloads. If any of these have infrastructure implications, constructing the workload's supporting infrastructure too early may mean updates anyway. Even worse, if the workloads become deprecated or halted, the effort of constructing any infrastructure for it may be a complete waste of time. It also can become overly cluttered to establish several unused components if they aren't actively contributing to the company's meaningful assets.

The next step is to conduct the pilot effort, ideally with the same workload and team identified in the POC step. This would be the real effort of integrating Team B's digital assets into Company A's ecosystem. Because the POC was likely somewhat sandboxed, in a highly simplified version of Company A's true ecosystem, there may be additional friction that arises when the components are created in the real thing. Just as in the case of the POC, the focus will be on the infrastructure components necessary to host Team B's workloads with as little modification as possible. Unlike in the POC, the infrastructure will be built directly in Company A's ecosystem. Because Company A is assumed to leverage container-based platforms, a

major step to the pilot would be to develop a container image for Team B's workloads to run within. If the workload ran on a VM before, it can take some time to ensure all the contextual dependencies and ancillary binaries are available in the container image. If the workloads already run in a container, the goal is to evaluate the container image for use in Company A's environment. Often, although the container could almost certainly run just fine in Company A's environment, in practice there may be constraints imposed within the environment, such as required cybersecurity dependencies or requirements around the use of certain features such as standardized readiness checks and liveness probes. The lifecycle of Team B's workloads and their infrastructure will also be fully integrated into Company A. As you've seen in Chapter 7, different automation platforms take different approaches to developing a particular automated procedure, and there may be organizational requirements and standard steps that Company A imposes. Although the POC can likely expose certain problems such as required permissions and networking concerns, a fully automated workflow may not be deemed sufficiently valuable to construct. Thus, there will likely be the need for considerable effort to ensure that all the requirements of each company are addressed in the automated procedures.

In addition to the purely technical concerns, there are also other concerns associated with this exercise. For example, there will undoubtedly be new stakeholders who have immediate vested interest in the configuration and workings of the workloads. Coordinating with these stakeholders even before the initiative begins will allow time to better coordinate across development cycles and align expectations to meet everyone's needs. Additionally, unless the companies happen to operate under highly similar software methodologies, it may take some time for Team B to adjust to the new way of working.

The pilot effort ends with a review and a report on the outcomes and findings. This work should be scheduled like any normal development task and be a required outcome of the pilot effort. It also allows time for the creation of any assets that can accelerate the subsequent rollout of the solution to other similar workloads.

Once a pilot initiative is completed, a subsequent full rollout of the emergent approach should be conducted on all other workloads that are similar to the piloted one, possibly also in a phased approach. Especially in the case of many workloads, the phases can be oriented around the teams that manage the workloads, perhaps starting with a single team and including more and more teams with each subsequent wave. If the team that incorporated the piloted workload manages other similar workloads, it would be a good candidate for this initial team. If the appetite for speed and tolerance of risk are both sufficiently high, running the rollout with this pilot team and a second team could be a great way to ensure the knowledge the pilot team gained begins to spread to other parts of the organization. If possible, it's beneficial to assess each phase for completeness and incorporate any new findings into the rollout procedures you're developing. Once this initial rollout is complete, an extensive assessment should be conducted to determine any additional considerations and reflect on what worked and what didn't. This final exercise of the rollout also includes deprecating the old Company B services and scheduling them for timely deletion.

Someone must be responsible for leading, tracking, and reporting the rollout. This responsibility could be assigned to a product owner, a technology manager, or any other number of traditional roles. Depending on the size and speed of the rollout, this could even be a new responsibility tacked onto an existing role. That being said, the more conservative you are with a rollout, the longer the overall effort will take, further delaying any new feature development for the workloads that are actively migrating; a small handful of teams and a dozen or two workloads may be fully addressable with a part-time role, but several dozens of workloads and more than a few teams increases the complexity super-linearly. If both companies' technology teams and workloads all see high activity, to the point where sparing a manager or product owner (or overloading them with additional responsibilities) is infeasible, bringing in external help becomes highly attractive. This way, the budget for the role, which is by design temporary, can be fully recovered after the initiative completes. However, the contracted position must be well supported with sufficient lead time to get

oriented within the organization, and ideally is part of the pilot team in some way, perhaps observing the effort or shadowing the management in order to quickly align on internal policies, cultural expectations, and traditional processes. They must also have sufficient autonomy, authority, and leadership assistance to ensure the transition process is not hampered by unnecessary bureaucracy, the untimely absence of decision makers or gatekeepers, or active resistance by voices in positions of power. This might be effectively addressed by designating an internal counterpart whose only relevant responsibility is to cut through the red tape and clarify the contractor's path to remove any administrative impediments they face; in this case, there should be a clear, documented line of succession for this internal counterpart to pass the responsibility off when they are unavailable. Regardless of who is tapped to lead the rollout, it should preferably be the same person or small group of people who oversees the rollouts for the entire effort of incorporating Company B's workloads into Company A's ecosystem. Having different people lead different rollouts can greatly reduce the momentum that has been built, potentially inviting back previously overcome impediments and other points of friction.

The next step, assuming you have additional workloads that *don't* resemble those you've already incorporated in Company A's ecosystem, would be to construct any additional infrastructure components you need in order to support them. As stated earlier, although it's possible to construct these components earlier, depending on the speed with which the rollout is conducted and the organization's appetite for moving targets, it's also possible that by the time you get around to them, something has changed in the requirements anyway. It's best to defer the additional components to when it becomes necessary. This goes for standard components too. Rather than creating placeholder components for future workloads, it's better to initiate them soon before they become necessary, as a well-established preliminary step for a workload's integration.

For each remaining class of workloads, those that operate within the same infrastructure and platform stacks, you can repeat the prior steps. If the new class of workloads is sufficiently different that it requires validating

delivery or infrastructure considerations, it's a good idea to again run a pilot with a relatively static or stable representative workload in order to smooth out the rollout experience. The rollout itself could again be done in phases, if you have many teams and many more such workloads. However, your earlier successful steps should make this much more straightforward and well structured, meaning you might be able to go more aggressive than the first rollout. If this rollout is indeed accelerated, it should be managed by a dedicated role or possibly even a small team, and again, preferably with the same individual at the helm as in the previous rollout. Once this rollout completes, it should have formal steps to assess the effort and initiate the deprecation and ultimate deletion process in Company B's ecosystem. This entire effort can be repeated, again and again, for subsequent classes of workloads that remain in Company B's environments.

Although it often comes as an ad hoc addendum to an otherwise fully planned (and budgeted) exercise, the deprecation process for Company B's components must be well planned and executed. Just as a new internal tool or capability would be socialized throughout the organization to would-be consumers and stakeholders, so must the deprecation of Company B's assets in their previously isolated environment. Without sufficient communications across multiple channels, the process of deprecating the old services may feel somewhat like a game of whack-a-mole, as uninformed internal stakeholders begin to complain about the unavailable services and their complaints make their way through the corporate communication pathways until they finally reach the desk of the initiative's leadership team. Once workloads are deprecated, they should be removed from active service and their underlying workloads and foundational components should be assigned a date on which to be deleted. This gives everyone a chance to raise alerts or issues with the new versions of the workloads, while also ensuring the old workloads don't long overstay their use.

The entire effort of incorporating Company B's workloads into Company A's ecosystem should be given time to stabilize before additional systemic changes are made. This way, the teams managing the workloads brought

over from Company B have time to converge on a working velocity. Preferably, they even are given the chance to undertake meaningful feature development within the newly adopted policies and procedures set forth by Company A. Any changes that do become scheduled should be limited to the original Company A workloads, unless there is a strong business justification for targeting one that came from Company B. There is a real risk in making one big change after another to the foundational components and configuration of your workloads, since for any problems that do arise over time, it may not be immediately clear if one of the foundational changes caused it, and if so, which one.

> **NOTE**
>
> This commentary is knowingly oversimplified in terms of the level of care and coordination it takes to migrate certain workloads, especially data systems. It's not meant to replace a thorough planning exercise, but to help to provide context to the assertion that this is not a simple flip of the switch, but a multistage, multiquarter process and to provide some guidelines for how to frame the overall effort.

At the end of this exercise, you have a single ecosystem that resembles that of the original Company A's. All of the teams are now operating in the same environments (or similar, but perhaps distinct, environments), using the same processes, and adhering to the same policies. This is almost never the end of the story, however, as many companies will then review the team structures from the top down and consider efficiency gains by way of combining teams and reducing team members. On the other hand, there will also likely be influence from the members of the previous Company B in terms of new communication channels brought into Company A, which will in turn, according to Melvin Conway, influence the newly combined company's systems as well. If done right, it can be achieved on the order of one year, give or take a healthy tolerance.

Combining Two Ecosystems

In the case of a merger, there may be a lot fewer obvious benefits to selecting one company's ecosystem to which to align over another. Certainly, if the merger is a strategic way to combine the high-quality digital assets of one company with the connections, brand recognition, or market penetration of another, the strategy may require little to no deep consideration. However, in many cases, both companies' digital ecosystem may be roughly the same level of maturity, complexity, or size, and they may operate under incompatible, but competing, software delivery methodologies. If the two departments are both deeply entrenched in their existing ways of working, and they both have comparable ability to produce high-quality, profitable products, it may be sufficient to combine the ecosystems and procedures around the automation platform and retrofit the assimilated company's existing delivery methodologies to operate within the new ecosystem.

For example, assume the decision is made to incorporate one company's workloads, Company D, into Company C's ecosystem, without getting bogged down into the details of how the decision was determined. This will roughly follow the same procedure as for an acquired company in the previous example, with one company seeing all of its workloads, infrastructure, and platforms preserved, and the other migrating its workloads. Unfortunately, because of the higher chance for odd situations such as a bigger workforce, codebase, or number of components coming in from Company D, considerations such as the procedures with which each company delivers their work tend to have a heavier gravity, giving even more credibility to the concept of keeping Company D's delivery procedures intact.

Rather than reviewing a full exercise of enacting such an effort, it is likely more valuable to identify common impediments or points of friction that may arise along the way, in addition to the software delivery methodologies. In some cases, there may be systemic, philosophical differences that manifest in practices that seem almost diametrically opposed. In others, choices may have initially come down to preference or tradition but are now regarded with an almost fanatical mindset, leading to a

strong aversion to change. In most cases, there are merits to the arguments on both sides, which usually implies that neither is objectively better or worse than the other. Therefore, whatever decision is reached often appears largely arbitrary, because indeed it is, which can be both uncomfortable to accept and difficult to defend.

The following are some examples of such points of contention — which may even manifest as only a partial assimilation of Company D's workloads, and, additionally, a partial migration of tools or techniques. They are certainly not exclusive to mergers, but the ostensibly comparable standing each company has at the corporate level combined with the seemingly partisan decision to consolidate onto a unified solution arbitrarily selected between the two may produce more acute protest from prominent voices within the assimilating company. In other words, there's a higher chance that a strong enough argument can be made for certain Company D components and techniques be allowed or adopted by Company C:

Network intrusion

> Company C might implement a highly layered approach to network security, with a trust/untrust or demilitarized zone (DMZ) configuration and an appliance inspecting traffic at the networks' edge for possible instances of intrusive behavior, without actively preventing the behavior, with an Intrusion Detection System (IDS) expecting it to be dealt with by an internal mechanism. Meanwhile, Company D might totally block all potentially dangerous traffic with an IPS, creating in the process a more direct path to application workload services. Company C teams may view Company D's approach as unsophisticated and uninspired, whereas Company D may view C's approach as excessive. Ironically, both may view the other approach as risky.

Compute runtime resources

> Company C might prefer a serverless approach, focusing on the application software, and letting the managed service handle scaling, restarting, and monitoring services. Simultaneously, Company D might be highly oriented around a cloud native ecosystem, hosting its

workloads on a managed Kubernetes platform. Luckily, these may be more compatible than they initially appear because most serverless solutions ultimately leverage container technologies directly, and appropriate container images can usually be exported for use in a container orchestration system, like Kubernetes, with minimal required modifications. The challenge remains as to which approach becomes the dominant one, which may not be resolved outside decree. However, it's worth noting that a cloud native approach is theoretically much more portable than a serverless one.

Overlapping database solutions

Perhaps each company has a different set of licensed relational, NoSQL, and key-value databases. Deciding which solutions to align to can sometimes be as simple as choosing the tools with the most advantageous licensing agreements and scheduling a migration from the old solution to the new. However, any features that are specific to a particular solution and that the assimilating teams are leveraging to any significant degree may need to be configured as a service or process external to the landing database solution.

Automation platform

Company C might use GitHub Actions, while Company D might use GitLab. Initially, it may be decided to keep both systems for the delivery of software. However, unless you experience uncommon fortune in structuring your licenses, you'll likely be pressured at some point to consolidate down to a single system. Although these and other solutions behave in similar ways, there are notable differences, for example, in how they manage parallel procedure steps and in how distinct processes may be integrated or sequenced (this was covered to some degree in Chapter 7). As such, it can become an intractably prolonged debate that has no satisfactory solution, and that may be resolved only by another arbitrary decision made as objectively — or at least, dispassionately — as possible.

Open source posture

Company C might heavily leverage open source solutions that Company D views as overly aggressive and risky. While the decision would undoubtedly be made by the newly combined company's legal team, it would be logical and prudent to coordinate with the engineering teams to determine the best course of action. Many times, even in the case of highly controlled industries, there are ways to de-risk the use of certain open source components. For example, you may be able to tolerate an open source version of your preferred, licensed Linux distribution — such as Rocky Linux in place of RHEL — in nonproduction environments, such as development or integration, or to host your software delivery workloads and other supporting tools to save money on per-node licensing.

Preferred CSP

Suppose Company D uses one CSP and Company C uses another. D now must reconfigure its workloads to work within C's ecosystem, transferring as many of its components as possible to directly analogous services in the new CSP. However, not only is there a chance that a service in D's CSP has no corresponding service in C's CSP, it's also highly likely that some of the matching services are not perfectly like-for-like and have subtle differences that can confound the uninitiated, no matter how experienced with the parallel service they may be. Although this is usually a fairly subtle issue, it can be especially difficult, slow, and painful to resolve, which means it can cause lingering resentment for the decision to switch providers.

Summary

In this chapter, you encountered several patterns and theoretical case studies regarding the experience of evolving simple projects into an enterprise-scale solution, including the following:

- Signals that indicate the need to split, merge, or otherwise refactor TF projects
- Patterns for refactoring TF projects using familiar and emergent approaches
- How major evolutions or scaling factors to TF projects result in personnel changes
- Recognizing the value of external and contracted support
- Expanding from a no-code website builder to a cloud-based solution to enhance the quality of software changes
- Transforming a vibe code hosted application to a high-quality, enterprise-scale solution
- How to approach the scenario where one TF ecosystem consumes another
- How to approach combining two TF ecosystems by stitching together portions of each

You now recognize both the importance of affecting evolutionary refactor exercises and the gravity of such an endeavor. You understand that it requires a variety of skills and perspectives including architecture oversight, deep skills in TF, and business leadership input. You also realize it's possible to make significant refactors to the TF code without disturbing the actual cloud resources, drastically reducing risk and skepticism.

In Chapter 9, you'll continue to explore practical approaches to common situations, this time drilling deeper into organization-specific considerations such as your company's management hierarchy, policies and traditions (both recorded and unwritten), and regulatory constraints. Additionally, you'll gain a pragmatic perspective on how to organize hierarchical network controls and network topologies, cloud access control policies, and access control for third-party tooling.

Chapter 9. Organization Structures and Policy Constraints

Continuing the study of enterprise-grade TF, you'll now get a deep dive into how to handle various organizational structures and policy constraints, especially those imposed on an existing company. This can come up at almost any time and to almost any-sized company in any industry. The material here should be familiar to most who have extensive engineering experience, as this type of change is common at almost any growing technology company as it gains popularity and moves into new markets.

For example, new compliance measures may be imposed by external forces, such as governments. Due to the nature of governmental policymaking, it can be unclear how long and in what final form a set of compliance measures will take. A measure also may not come with especially clear instructions for every case, leaving certain provisions up to interpretation or imagination. Similarly, systemic changes can be brought about by internal forces, such as a new organization structure imposed by corporate leadership, or an initiative to better comply with a voluntary audit control list such as SOC or NIST. In either case, it can feel like it's work for the sake of work instead of motivated by sound business justification. One of the goals of this chapter is to offer legitimate rationale — to members of leadership or governance teams and to developers and engineers alike — behind these changes, while another is to provide strategies for adhering to new compliance or organization structures.

Other common scenarios that will appear in this chapter are updating outdated policies, working in the absence of complete policies, and engineering for the future. These are more common situations for rapidly growing companies and may be less familiar to organizations that are well

established; however, they can still be illustrative of how to evaluate the merits of a change and implement those changes that are nonnegotiable or have achieved critical mass support.

Important Principles

There are several important foundational principles in software engineering. Although TF is not a programming language — nor is HCL — there are several concepts that can be useful to apply, albeit perhaps indirectly. These can be illustrative, especially in terms of conveying the merit and benefits of an otherwise highly abstract concept by drawing parallels to more well-established concepts.

Some principles can be almost exactly applied to TF development, such as:

Principle of least privilege

> According to the definition by Dr. Jerome Saltzer and Dr. Michael Schroeder in Proceedings of the IEEE Volume: 63, Issue: 9, "Every program and every user of the system should operate using the least set of privileges necessary to complete the job." This means that operators, both human and service identity, should have as narrow a set of access controls as possible both to the cloud resources and to the HCL code they manage. Likewise, the cloud resources themselves should have as few permitted capabilities as possible to perform their required responsibilities.

Principle of abstraction

> As defined by Dr. Benjamin Pierce in *Types and Programming Languages* (MIT Press), this principle states: "Each significant piece of functionality in a program should be implemented in just one place in the source code. Where similar functions are carried out by distinct pieces of code, it is generally beneficial to combine them into one by abstracting out the varying parts." TF's concepts of modules, variable definitions, and local values — as well as more advanced

instrumentation around the TF backend to manage parallel TF states with the same TF project — are all manifestations of this principle.

CAP theorem (aka Brewer's theorem)

> According to Dr. Eric Brewer, among consistency, availability, and partition tolerance, "at most two of these properties [may be guaranteed] for any [distributed] data system." This is consistent in cloud computing, an inherently distributed system that is highly data oriented, including metadata regarding resource utilization and consumption and different types of activity logs, in addition to the data gathered, generated, and utilized by your application workloads. In terms of TF, you may have consistency among the state triplet and availability of the state backend and the CSP, but there's no way (nor a strong need) to partition your TF state.

When mapping some of these concepts to TF development, in some cases, the principle is much more aspirational than the original, with many corner cases and caveats, and typically with some nontrivial abstraction or shift in perspective to substantiate the analog. This further justifies their review, since it can be easy to draw the parallel and just as easy to overlook their deficiencies or caveats.

Single Responsibility Principle

The first borrowed concept is the *Single Responsibility Principle* (SRP), which can be stated as "a module should have one, and only one, reason to change." This principle comes from the work of Robert C. Martin and is openly based on Conway's Law — it's presented as an "active corollary" — and although it's well discussed, it's not always well understood. In particular, if applied in this form to TF projects, it might lead one to develop extremely small local or remote modules that contain single resources — or worse, to develop a layered wrapper approach to modules where each layer controls a single resource attribute that gets passed along to a final, singleton module that collects all the passed-along values.

Luckily, in the final version he offers in his book, *Clean Architecture* (Pearson), Martin states "a module should be responsible to one, and only one, [group of users and stakeholders]." This reframing is much more practical, in general, and also more consistent with the recommended practices for TF module development. However, it requires some care in defining what is meant by *module*, *users*, and *stakeholders* from the principle's statement. Moving backward, the *stakeholders* would be the group of engineers, architects, business entities, and anyone else that together drives the development of a TF module's code. The *users* would be anyone who references a TF module or uses it directly as a root module. In particular, in the case of root modules, the users are the subset of the stakeholders consisting of the engineers who manage it. Finally, the *modules* referenced in the principle are indeed more or less the TF modules themselves. However, because of the modular and telescoping nature of TF modules, and the value of reusable modules hosted remotely, it begins to blur the boundaries of what is considered the single group of users and stakeholders as the users of one module become stakeholders of any nested modules, especially remote modules. It even calls into question whether the SRP module considered here is simply a single TF module or if it contains all included nested TF modules (or perhaps just the local nested modules, mapping roughly to a TF project). This may be reconciled by placing the responsibility of using a nested module with the module that references it, suggesting that the operator is assuming responsibility for the nested module. This still means a remote module could be influenced by these end users, either via a request process or as an initiative to better serve a wider audience, but since a similar module could simply be developed, rather than requiring the existing module be updated, it reduces the reliance on users for influencing change.

In terms of policies, such IAM policies or other policies that may be assigned at various scopes (as you'll come across later in the chapter), this can influence an important design consideration: should a policy that is applicable to two scopes or groupings of resources be applied as a single policy binding — perhaps at a higher, shared parent scope — or should the policy be duplicated and each scope/grouping have a dedicated binding?

According to SRP, one needs to treat them as separate only if each scope or grouping has a different group of users and stakeholders. If both scopes or groupings have the same group of users and stakeholders, the policy likely should be assigned in a single shared capacity, whereas if they simply happen to align on the needed policy, and they have different teams responsible for them, they should be assigned dedicated policies, as shown in Table 9-1. This is also true for other resources that may be assigned to different scopes of resources, such as network controls and log management. For example, the same layer 4 firewall configuration may be suitable for multiple workloads, but likely should be duplicated if they are needed in different applications or environments so they may be dedicated to that context, even if they are present in the same VPC, as shown in Example 9-1.

Table 9-1. Deciding how to bind policies to groups of users and stakeholders

Group scope	Policy binding logic
Singular group of users and stakeholders	Assigned in a single policy
Distinct groups of users and stakeholders	Assigned in dedicated (but otherwise equivalent) policies

Example 9-1. Comparing a policy applied to two contexts versus dedicated policies

```
# Single firewall for two application workloads
resource "google_compute_firewall" "combined" {
  name    = "tas-9-1-combined-fw"
  project = var.project_id
  network = google_compute_network.default.name

  allow {
    protocol = "tcp"
    ports    = ["80", "443"]
  }
```

```
  source_tags = ["app1_web", "app2_web"]
}

# Dedicated firewalls for each application
resource "google_compute_firewall" "app1" {
  name    = "tas-9-1-app1-fw"
  project = var.project_id
  network = google_compute_network.default.name

  allow {
    protocol = "tcp"
    ports    = ["80", "443"]
  }

  source_tags = ["app1_web"]
}

resource "google_compute_firewall" "app2" {
  name    = "tas-9-1-app2-fw"
  project = var.project_id
  network = google_compute_network.default.name

  allow {
    protocol = "tcp"
    ports    = ["80", "443"]
  }

  source_tags = ["app2_web"]
}
```

Separation of Concerns

Separation of concerns (SoC) means each module handles one aspect of the infrastructure in isolation, with minimal knowledge of other modules.[1] In TF, this translates to loose coupling, one-way dependencies, and modules that do not leak implementation details to their callers. Cristina Videira Lopes described two levels of separation in "Aspect-Oriented Programming: An Historical Perspective (What's in a Name?)" (2002), paraphrased here:[2]

Conceptual level

SoC provides a sufficient abstraction for each concern as an individual, sufficiently primitive, concept.

Implementation level

SoC isolates the concerns from each other, through separate blocks of code, and provides for a loose coupling of these blocks.

This fits nicely with SRP, as it describes how a module (in the SRP parlance) interacts with other modules — that is, individual modules interact loosely and are disinterested in one another's implementation or side effects. As it applies to TF development, it suggests that modules should be self-consistent and loosely coupled. This is consistent with the guidance throughout the book to ensure that dependencies between child and parent TF modules, as well as among sibling modules, are minimal and that no mutual dependencies — whereby one module relies on information from another and vice versa — exist. Extending this to entire TF projects, especially in the layered approach to TF project organization presented in Chapter 8, it's most prudent to adhere even closer to the principle, allowing only strictly one-way relationships — such as a downstream project that relies on information from an upstream project but that *does not* relay any information back to the upstream that the upstream requires to continue its procedure. This keeps everything clean: for every change, the TF projects are applied once, and only once, in a directed way, and never require a restart to ensure tightly coupled resources or inner modules eventually become consistent, or another run to read from a downstream system's output; every module contains all the information it needs to carry out its responsibility, meaning there's no need to closely coordinate with members of other teams for which there is no explicit stakeholder relationship (meaning the teams were deliberately organized by corporate leadership/management to intentionally separate concerns, possibly to introduce friction for the sake of oversight).

The challenge presented by SoC is how to balance overfitting with verbosity. Overfitting a module, by fixing its internal attribute and module

values and only parameterizing certain fields, can lead to a proliferation of similar modules that must each be managed separately, duplicating work if new policies are introduced that standardize certain fields. On the other hand, overly verbose modules, those that expose many variables with high flexibility — even those with plenty of standard and appropriate default values set on the variables — might either expose too much power to users of the module that must be mitigated or gated in some external way or might require highly intricate `validation{}` or `precondition{}` blocks to ensure users don't step out of bounds. As with many solutions at the enterprise scale, organizing and constructing TF modules is an exercise in juggling several — often conflicting or paradoxical — agendas, obligations, and risks.

> **NOTE**
>
> Overfitting is a term borrowed from statistical analysis and is meant here to portray a module that is overly fit-to-purpose to the point of being rigid and not useful far beyond its original intended purpose. This property can certainly be mitigated by simply parameterizing the pertinent fields and can again be made to be fit-to-purpose (now with two subpurposes) by implementing narrow `validation{}` blocks. However, it becomes a tedious affair to continuously update the validation with more and more conditions, not to mention how each additional condition adds an order of complexity to the statement that is not easily tested.

Moreover, in either case of overfitting or verbosity, there is a strong risk of one team building a competing module, rather than reusing an available module, due to a lack of transparency or agency regarding the existing module. Additionally, taking the separation of concerns to the extreme, conceptually, modules developed by other teams can be viewed as off limits or dangerous if their inner workings are opaque to possible consumers — remember, the separation is most specifically among digital components; any separation between teams is a byproduct, rather than a feature. This makes an approach inspired by the OSS paradigm, whereby would-be consumers are able to view and possibly even offer changes, highly valuable even for internal project teams — the projects need not be truly

open, but the general procedure found in OSS can be made available to your internal engineering teams. Such an approach gives a great deal of power back to consumers, essentially providing a path for opting into becoming a stakeholder for interested parties.

From a solution design standpoint, SoC shows up even more prominently as a way to separate and, to a degree, isolate dissimilar concerns such as authorization controls from data management in web application products. The concept aligns perfectly with familiar paradigms such as OOP, microservices, and domain-driven development, as it suggests software behavior should be encapsulated, integrated through strict interfaces, and undisclosed to one another, hiding its implementation and appearing as opaque systems. Indeed, you've seen that even TF, as a tool, implements a strong SoC in the state triplet from Chapter 6, which separates the concerns of developing code in HCL, tracking what was most recently applied with the TF state, and running the services and instances in the CSP. In this way, changes are made deliberately and carefully, and untracked changes are disposed of with the assumption that they were unintentional, temporary, or even detrimental.

Service identities, such as roles in AWS or service accounts in GCP, also enable a separation of concerns, allowing individual instances, services, or procedures to have their own identity and set of access permissions. This keeps the workloads and processes isolated from each other and prevents inappropriate activity between entities. Service identities also act as an interface between your automation platform and your CSP. The workflows or pipelines themselves act as the module from an SoC or SRP perspective, enclosing their processes and providing additional interfaces, in the form of webhooks, for example, to trigger and pass data between each other.

Separating concerns is the primary rationale for implementing a request or ticketing system whereby an application or platform team requests cloud infrastructure through a standardized form that can be assigned to the infrastructure engineering team responsible for maintaining the cloud resources. Similarly, maintaining a strong SoC is a keystone principle in establishing the guardrails necessary for introducing a self-service system

and especially when embedding AI agents within the automated TF toolchain. It keeps unforeseen and detrimental behavior from wreaking havoc on your infrastructure supply chain — effectively composed of the various repositories that store TF configuration and the platforms that are used to operate them — due to overexposure of capabilities beyond their relevant entities and responsible parties.

Defense in Depth

The major security attack vectors in the cloud are oriented around networking and access permissions. Incoming, or *ingress*, traffic can theoretically come from anywhere, and it's not guaranteed that requests are always sent with good intentions. Services that your workloads connect with that are external to your ecosystem can also be dangerous, possibly hosting corrupted or malicious software and posing as legitimate web locations. It's a highly secure posture to restrict outbound, or *egress*, traffic to only those known locations that are vetted and allowed by your network security teams. Restricting operations in the cloud to only whom and what you allow is a central facet of any security posture. The main idea behind defense in depth is that no solution is able to perfectly secure its respective vector, and also that different solutions can have significantly different security concerns and considerations, so having multiple solutions that operate in a layered configuration can cover gaps that any one solution misses. These layers are generally implemented both within different layered TF projects and by introducing security controls on the infrastructure supply chain.

By design, defense in depth adds overhead that comes with a cost and its own inertia, in terms of the cost of maintaining and upgrading services and appliances. This can make big changes such as choosing to adopt a comprehensive suite of tools unexpectedly risky, as the institutional knowledge gained in one stack may not translate perfectly to another, at best leading to the awareness of *unknown unknowns* — or the admission that you don't know what you don't know — and at worst manifesting as an exercise in hubris. There will almost certainly be gaps that were filled in

one tool that must be reconsidered in the new tool, and there will very often be aspects or considerations that were covered by the existing tool and that become gaps in the new solution. On the other hand, it can be awkward to promote the incorporation of new tools that cover newly identified security vectors or gaps, as it can feel like an admission of failure in sufficiently securing the organization's assets. Even when the attack vector is only recently becoming adopted in the industry, it can be a tough sell to convince nontechnical decision makers or influential individuals to appreciate the need for a new expense. It's perfectly appropriate to make the case in earnest and without shame, and for the risk, rather than the proposed solution, to be the one adopted. What's not appropriate is to avoid presenting the gap or solution for fear of rejection.

In some cases, a defense in depth posture may even intentionally introduce friction, such as by including manual approvals be authorized by select (human) individuals for certain automated workflows, or by introducing an access control procedure that temporarily upgrades an individual's permissions through a secondary authentication mechanism. These might be reinforced by components that monitor access logs and flag any anomalous, overwhelming, or potentially detrimental behavior for continuous auditing of your ecosystem, or that even triggers corrective or remediation measures to disallow, throttle, or otherwise restrict said behavior. For example, a vast, unexpected increase in API requests from a particular user may indicate a problem: perhaps the individual's credentials were stolen and a malicious third party is probing your systems looking for things it can attempt to penetrate; or perhaps the individual is simply in some kind of endless loop or cascading operation that makes gobs of unintended requests. Certainly, it could also be a legitimate practice, or at least innocent in intent, such as due to local testing of a new automation platform that is designed to be highly chatty with the CSP; locking down this individual's account puts a drag on their productivity and probably incurs additional effort for others in the organization to unlock it. Therefore, just as it's important to prevent detrimental anomalous behavior, it's equally important to have a way to certify beneficial behavior that happens to be anomalous.

> **NOTE**
>
> In the previous scenario, where the individual is testing a tool locally, perhaps there should be a way to allowlist or otherwise designate the individual beforehand, or perhaps they should have a way to generate a service identity that may be used or impersonated locally. All of these are good options or capabilities to adopt as part of a security posture, but they all require work to enable and configure effectively and they must be prioritized. In other words, they're almost certainly not all available on day 1 but require significant effort to put in place and ensure efficacy.

Of course, no principle is without its drawbacks. In the case of defense in depth, it can have steep diminishing returns. This can lead to incompatible philosophies in how to assess and mitigate these diminishing returns, and unfortunately can put multiple individuals with sufficient influence, power, or budget at odds with one or more teams defecting from the leadership-endorsed standard, either by gaining an exception or by engineering a back channel to get around the rules. Ironically, this almost always exacerbates the diminishing returns: refusing to use one tool in favor of an analogous tool adds configuration and maintenance effort beyond the licensing sticker price, making the solution even more costly; setting an auto-approve to circumvent manual approvals removes the value of the defense layer without removing any of the cost to license and maintain it.

Another caveat to keep in mind is that once multiple layers of defense are adopted, making certain changes requires updating multiple defense layers. This cuts both ways: both in blocking a new-found attack vector and in forging a passageway for newly-identified appropriate behavior — in which case, it may require some additional components or configuration to compensate for the exposure, such as heightened scanning of requests or additional permission requirements. For one example, there may be regions that are associated with bad actors or dubious activity and that you don't have a vested interest in serving or conducting operations, so you may choose to disallow any workloads be initiated in these regions and likewise block all traffic that appears to originate from related CIDR ranges or domain extensions. If you later decide to relax these conditions, such as to begin operations in the region, not only will you have to modify the existing

controls, you may need to enhance intrusion detection or prevention or implement additional, more fine-grained networking and access constraints to maintain a robust defense posture.

As a full example, you can set the following controls in a defense-in-depth approach, a portion of which is depicted in Figure 9-1:

Account-wide policies

: These provide baseline controls around what behavior may occur across the account, such as allowing only IDs from your domains to be assigned IAM policies and network traffic from allowed origins.

Scoped policies

: If available, these may further refine the account-wide policies to further restrict behavior.

Layer 7 firewall

: Application-layer filtering for network traffic, including originating domain and content inspection.

Known threat analysis

: IDS/IPS that analyzes network packets to assert or prevent disallowed or unexpected traffic.

IAM policies

: Bound to resources, groups of resources, or possibly wider scopes, and define what authenticated users are authorized to do.

CSP proxy

: Used to allow direct access to private network attached resources.

Layer 4 firewall

: Transport-layer filtering for network traffic, including IP address or transport protocol (such as Transmission Control Protocol [TCP], User

Diagram Protocol [UDP], or Internet Control Message Protocol [ICMP]).

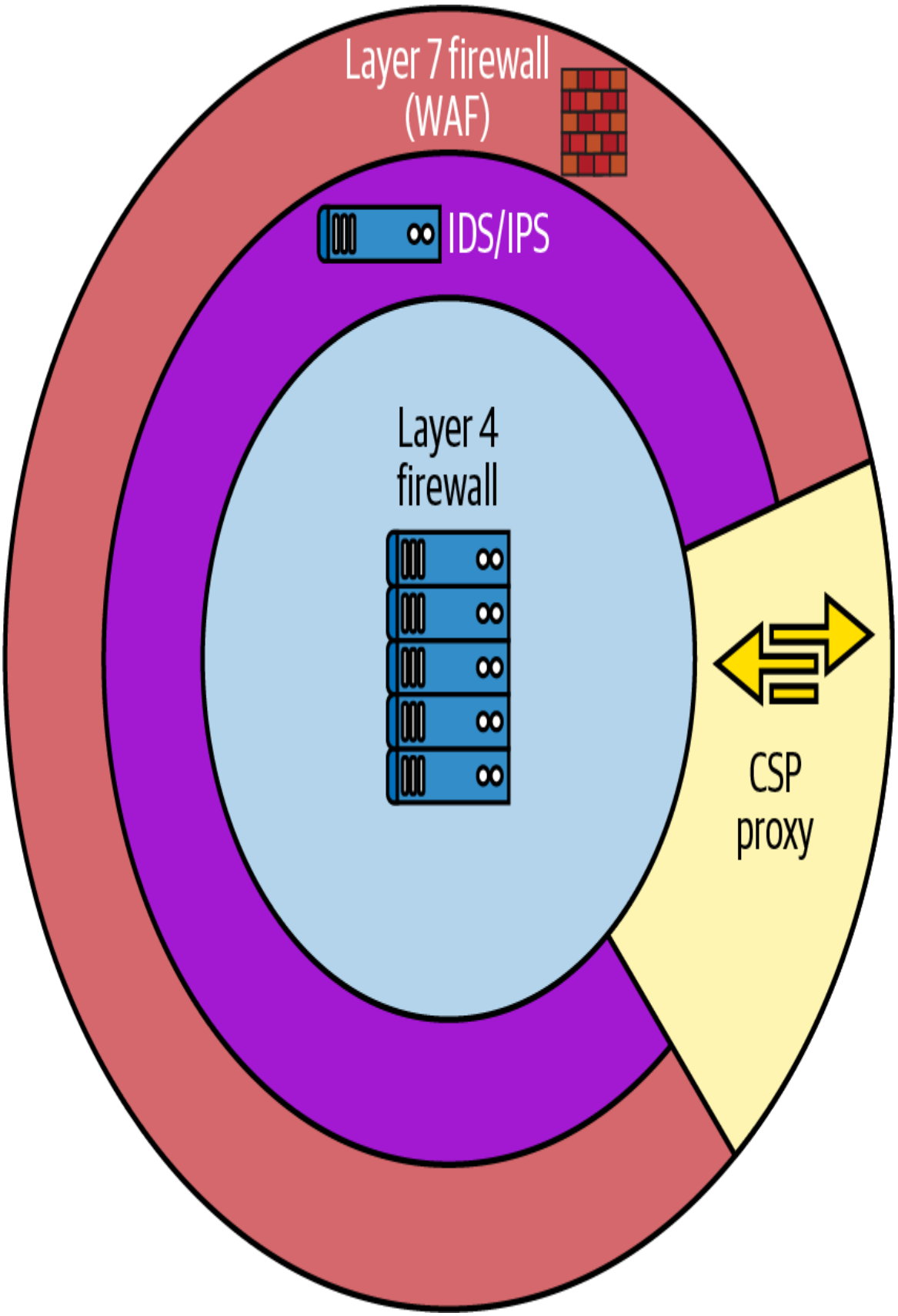

Layer 7 firewall
(WAF)
IDS/IPS
Layer 4
firewall
CSP
proxy

Figure 9-1. Depiction of defense in depth of networking components

Common Paradoxes

Alongside the pertinent principles at the root of enterprise-scale cloud infrastructure, there are several interesting phenomena that seem paradoxical in nature. Typically, these emerge from the presence of different agendas, which all may be both virtuous and likely independently feasible. However, the corresponding perspectives may lead to different expectations and conclusions with respect to technical implementation, resource organization and allocation, and product lifecycle management. This requires a way to transcend these agendas, taking both into account and balancing their benefits and merits. That these paradoxes exist at all naturally benefits integrators, contractors, and consulting firms, as they guarantee there's no single right way of doing things, but that every solution is nuanced and unique.

Vendors: A Double-Edged Sword or a Thornful Rose

You can protect against only those threats known to you, which implies that there could be (and indeed have been) periods of time where another entity is aware of an exploit unknown to you. Moreover, entire classes of attacks are introduced as technology advances, such as the latest wave enabled by AI-like prompt injection, model poisoning, and deepfaking an individual's behavior. This leads to the realization that digital security is a constant arms race. It's not something that can be achieved once, with no need for reinforcement, upgrades, or additional protection. Likewise, the never-ending battle has no one collective frontline, agreed-upon rules of engagement, or even a set orientation, meaning multiple perspectives must be kept at all times.

Furthermore, much advancement in digital security is driven by edge cases and vanishingly long tails: the exploit may be due to a perfect combination of configuration and functionality that can be difficult to pinpoint from a single incident, much like the origination of a virus is impossible to

determine from a single patient. The vulnerability may even lie dormant for years, and if it's included in third-party software, it's up to the third party to ensure not only that the vulnerability is addressed but also ensure that you as a customer are informed so you may install an appropriately patched version. Because admitting an exploit can be damaging to a brand, this communication is not always made explicitly, potentially leaving some customers vulnerable if they don't have a policy to keep all dependencies fully upgraded all the time — which is almost every company.

> **NOTE**
>
> The point about upgrade policy is not as tongue-in-cheek as it might read. In some cases, there is sound rationale for delaying certain widespread upgrades, such as to validate that the upgraded solution is safe for use according to your other internal policies, or to ensure that the upgrade does not interfere with other mission-critical behavior. Certainly in the case of a zero-day exploit, for example, there should be contingencies that accelerate adoption, but blindly accepting every new version of a vendor's software is not a universal norm.

That digital security is a constant disproportionate battle of outliers is, in a way, consistent with the Pareto Principle taken to an extreme: the 1% of issues that you don't currently cover (because you don't know they exist) could take up to 99% of your time searching for it. At the same time, someone may randomly or accidentally stumble upon an exploit without exactly meaning to, leading to a severe disproportion of power. When you further consider that properties like skill and experience across any population resembles a normal distribution, it becomes infeasible for every company in the world to field the multiple entire teams of software and network security experts it would take to fully build all their solutions in house.

Because of this costly nature to finding and mitigating exploits (including those that are well known), it's far more economical to license third-party solutions, even if this opens you up to the risks inherent in relying on a third party. Of course, the more third-party solutions you include, the more such risk you assume, with each third party representing a new attack vector to

consider. At the same time, it's already established that no solution is perfect nor eternally effective, meaning that it's almost certain that no two vendors have a perfectly equivalent solution. It may not be prudent to fully trust a single vendor or solution to protect the entirety of a single attack surface. Meanwhile, no financial wing of any organization will support the unbounded increase in licensing agreements. Not to mention the inherent risk of the other side of the vendor spectrum, whereby all your controls are configured in a single solution or by a single vendor — if the vendor is compromised, you're compromised, and largely at the mercy of the vendor to implement the fix. This makes the overall effort of securing your ecosystem a constant balance between too many and not enough third-party and in-house solutions, with multiple arguments that don't exactly line up in direct opposition, but rather create a multifaceted carousel of logic that (hopefully) is continuously approaching consensus and optimization, but may never fully get there.

Competing Policies

Sometimes, the mismatches across the various security considerations do lead to competing or conflicting requirements. This can lead to a system so secure in theory, it becomes dangerous in practice, preventing legitimate parties from carrying out their official responsibilities. On the other hand, it can lead to combinations of solutions such that one undermines the value of another, both wasting time and effort and inducing a false sense of security. Usually, this is not as simple as two tools directly competing, but is more often a combination of multiple tools and configurations. This can obscure the issue when it arises, leading to multiple *red herrings* or dead ends, and can complicate the development of an appropriate solution.

An oversimplified example of this is when an organization imposes a strict restriction on SSH access to general compute resources, requiring either the use of a reverse-proxy tunnel solution or first accessing special designated jumpboxes, and then resolves to manage a third-party licensed security appliance solution. Because this solution will run on compute resources, it's facially subject to the restriction, requiring the additional components.

However, a separate constraint may be imposed that disallows any other resources to be attached to the networks to which the appliance is attached. Settling this requires the participation of all relevant stakeholders, and may very well result in an exception to the general rule, but with extremely narrow network controls that allow SSH only from very specific origins and only by certain users.

Another example is an organization-wide restriction on any new technology, including updates, that are developed by a third party, requiring it to be first thoroughly vetted by internal security teams. Under normal conditions, this ensures a high degree of trust to be placed on the new software, as it can serve to confirm that a vendor's claims are accurate and identify any vulnerabilities that may have been introduced. However, in the case of an important security patch, such as for a zero-day vulnerability, any delay to adoption may incur catastrophic damages. This suggests some available mechanism for use in an emergency such as to fast-track the application of a patch or a way to (temporarily) suspend or jettison affected components that only degrades the UX but does not make the entire system inoperable.

Similarly, an agreement may be made to disallow remediation of certain components, such as VMs running on compute instances. A typical approach is to prebuild a machine image, layering in all the necessary components and validating it, then storing it for actual use. In this way, the most recently validated image may be executed on a fresh VM, without modification. Again, in the presence of a zero-day vulnerability, this may simply take too long to appropriately close the gap, requiring a workaround, likely escalating to senior management or even leadership levels.

Finally, adopting a framework like TF for managing your cloud infrastructure components implies a lean toward idempotent operations. If you're also subject to immutable components, as described previously, you'll likely have to include additional controls, such as lifecycle conditions, on such components to avoid subsequent modification by TF. Depending on the component, lifecycle conditions may be used to cause TF to ignore the changes it detects, or they may be configured to incur a

replacement of the component, rather than an in-place modification. In either case, there should be a well-documented procedure that is directly associated with a business justification.

Access Control

Access control is a central security component in cloud computing. It controls who has access to what, making it arguably the most important aspect in securing your cloud resources. In most CSPs, permissions can be grouped and scoped to produce highly fine-grained, fit-to-purpose, yet manageable access policies. These can be associated with individual resources or various groupings of resources, and can then be assigned to groups of users or service identities — CSP-internal identities that are meant to be associated with instantiated cloud services — based on the responsibilities of the group or service. This creates a vast, rich ocean of possible approaches with which to assign access permissions. This section intends to provide pragmatic guidance around building access control for your cloud ecosystem.

Practical Access Considerations

Each CSP has its own way of managing IAM, largely made up of extremely granular permissions, typically supplemented with some logical combinations thereof, and with mechanisms for combining several permissions and/or assignments into policies. Indeed, there are usually several valid general architectures that could be applied, and for every such configuration, seemingly uncountable variations thereof. Unfortunately, many more invalid, or suboptimal, IAM configurations exist, and worse, what is appropriate for one organization's management structure may be inappropriate for another. In lieu of deep experience — possibly running into significant problems and almost certainly encountering such suboptimal solutions — the following will give you practical guidance on how to approach IAM, in general. In many cases, these are fairly obvious to a seasoned professional, or at least quickly picked up as a limitation or

consideration. They're included primarily for completeness and also serve as a quick summary of the various concerns related to enterprise-scale TF ecosystems.

One concern is around the management of permissions for human operators and stakeholders that typically will be organized in teams that have unique responsibilities shared among team members. In some systems — such as a CSP or an automation platform — it's possible to define users directly in the system or to import them in. This can be useful in early stages, when a solution is still being evaluated, before it's deemed appropriate for widespread use. However, this approach has severe diminishing returns, especially assuming an existing identity provider solution is in use. Instead, it's better to integrate the existing solution in such as using a single sign-on (SSO) solution or an OAuth provider. This greatly improves consistency of identity authentication across experiences, reducing the overall effort to purely authorization management. Implementing this early on, and onboarding identities even before meaningful authorization policies are fully established, can be valuable as team members begin to be directed to the system.

Another important topic that is often overlooked is how to communicate credentials such as key files or passwords. The most secure way is to not transmit them at all but to somehow develop a way for users to generate credentials on their own. Unfortunately, this isn't always effortless to establish — unless it's an included feature of the identity provider — and some less-than-best practices may need to be introduced in the meantime. In this case, it's important to avoid especially bad practices, such as sending anything unencrypted (remember, base64 encoded is *not* encryption), including both the identifier (such as employee ID, email address, or other title unique to the user) and the secret credential element (such as password or private key) in the same message, or sending credentials over team communication platforms — which is typically not treated as the most private channel and may be accidentally left on the user's screen when it's visible, virtually or physically, to other individuals. It might be acceptable to leverage a file sharing system that encrypts items at rest and in transit

and *does* already integrate with your corporate SSO. Especially if you're able to manage the encryption keys yourself, rather than relying on the service provider, you may even find the shared responsibility inherent to such a solution to be less risky — from certain perspectives — than a solution you develop and completely own.

> **NOTE**
>
> Most modern identity providers include support for individuals to create their own password. However, many of them first require a temporary password be generated (or assigned by the operator), and these still must be transmitted to the user in some secure way.

The final consideration is regarding certain metric-based guardrails, most of which may be modified and some of which are largely immutable. Depending on the CSP, there may be different limits, capabilities, or implementations of the various IAM components, giving each CSP a unique experience when it comes to managing access control composition, bindings, and inheritance. For example, both GCP and Azure allow IAM controls to be set at different scopes or groupings of resources — with GCP introducing a *Project*, which groups resources; a nestable *Folder object*, which groups Projects; and the *Organization*, which is the account itself; and Azure implementing the concept of a *Resource Group*, which groups resources; a *Subscription*, which also groups resources and may include Resource Groups as a subdivider; and a *Management Group*, which groups Subscriptions — with a lower scope inheriting the policies of its parent scope. This means a solution in GCP that uses heavy Folder nesting and sets policies at the Organization level will likely not translate well to Azure without significant reorganization. Likewise, although GCP's Project object is roughly analogous to the Subscription entity in Azure, a solution in Azure that invests in Resource Groups may better map its Subscriptions to the GCP Folder object, with Resource Groups instead mapping to Projects, as depicted in Figure 9-2. This makes it highly unlikely that the exact same setup will be optimal in multiple cloud providers, meaning each new CSP

will likely be best served by a new IAM architecture. This also makes duplicating an ecosystem in an additional CSP — perhaps to take advantage of temporarily available discounted compute resources or to keep a warm disaster recovery environment ready in the case of catastrophic failure of a particular CSP's region or datacenter — almost never a direct mapping transformation.

Instead of trying to align the organization of components in each CSP, it's better to focus on how to enable a similar operational experience so that operators aren't bogged down making the same functional change in different ways every time, requiring duplicate deliberation and implementation and halving their effective output. As long as an engineer has effectively the same control — both in what it can and cannot do — over a resource for which they're responsible, irrespective of which CSP hosts it, a configuration can be considered successful.

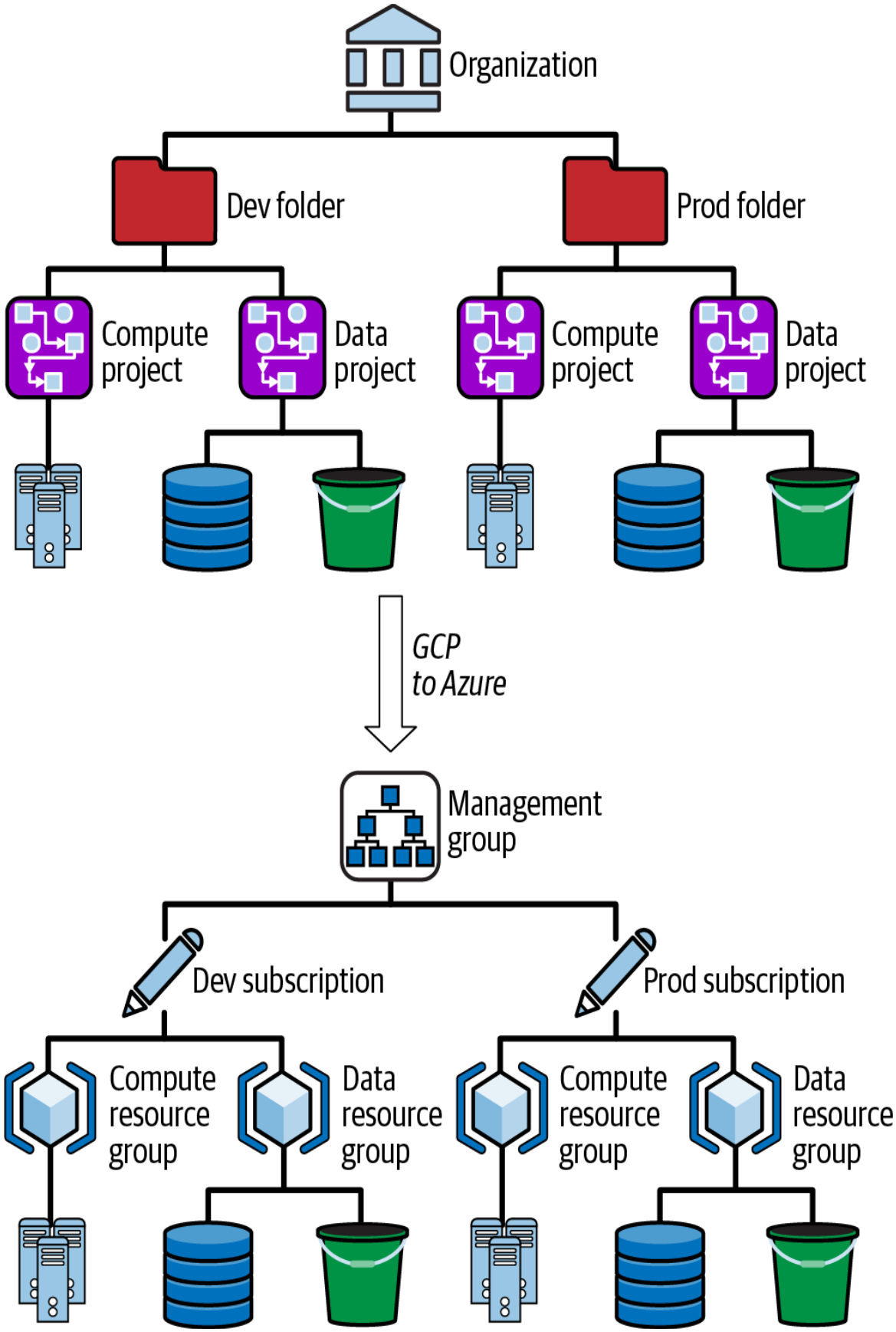

Organization
Dev folder
Prod folder
Compute project
Data project
Compute project
Data project
GCP to Azure
Management group
Dev subscription
Prod subscription
Compute resource group
Data resource group
Compute resource group
Data resource group

Figure 9-2. Reorganizing resource organization objects when rearchitecting for a different CSP

Authoritative Versus Nonauthoritative

In some CSP's TF providers, the types are both *authoritative permission binding* — meaning that applying the binding will overwrite previous bindings — and *nonauthoritative permission binding* — meaning the binding is purely additive and will append to the permissions that are bound. Depending on how your company chooses to organize its teams and responsibilities, one or the other type might be most appropriate. For example, a highly centralized IAM posture will benefit from authoritative policies because they can guarantee the permissions they bind in the TF project are the only permissions bound to the respective resources. In contrast, a highly decentralized approach with very loose dependencies may require nonauthoritative permissions to allow for customization. In this latter case, the TF project will worry about only the permissions it has or will bind, and in most cases, will not impact permissions bound in other projects. Table 9-2 describes lockout risk and recommended use cases for the various binding types available in GCP.

Table 9-2. Comparing various binding types available in GCP for lockout ri recommended use

Policy type	Authoritative/nonauthoritative	Lockout risk	R u
`*_iam_policy`	Authoritative	**Absolute.** Wipes out all inherited or preexisting access.	E b o s r
`*_iam_binding`	Authoritative	**Targeted.** Destructive to users and groups sharing that specific role.	C e o w s g
`*_iam_member`	Nonauthoritative	**Low.** Cannot purge unauthorized or manually created access.	L p d t f

> **NOTE**
>
> There is one major exception to permissions bound in one project not affecting permissions bound in another: if two TF projects are binding the same permission to the same *resource* (the object of the permission) and the same *principal* (the subject of the permission), if one of these projects, project A, removes the binding, it will impact the other project, project B. This is one of the few cases where the same thing can be managed by two TF projects, without being imported into either. In this case, if project B were to run, it will reestablish the permission removed with project A. Thereafter, if project A were to be run, it would *not* re-remove the permission (unless it's first re-added to the TF project). This might be perfectly desirable in the case where the responsibility of the permission is deliberately passed from one project or team to another, but it can be highly confusing in cases where the two bindings were introduced without knowledge of this overlap.

Clearly, the main benefit of authoritative access policies is that you know exactly what you're going to get once they're applied. The corollary to this is you'll remove any permissions from the resource that are not mentioned in the policy. This could be valuable in cases where the resource is highly sensitive or delicate, or whenever you want to fully dictate the permitted users and actions. However, overusing authoritative access policies can lead to undesirable outcomes. If authoritative policies are included in different TF projects for the same resource, for example, the two projects will effectively compete, with the most recently applied TF project as the dominant one, establishing its encoded version of the permissions and overwriting whatever was established by the other TF project — until it's run, reversing the dominance. For this reason, authoritative permissions bindings should appear only in certain places throughout an organization's entire codebase and should not repeat.

In the presence of scoping objects, such as GCP's Project or Folder services, what is feasible is to establish authoritative bindings for the general service or scoping object as a foundational step, and to establish instance-specific authoritative bindings in a later step, as shown in Example 9-2. Because these are bound to different resources, from the CSP's perspective, they do not interfere. This is because authoritative policies do not interfere with inherited permissions, so the more general permissions that are inherited by

the resource would remain. This makes IAM policy architecture design an interesting challenge, likely with multiple valid approaches even for the same organization.

Example 9-2. Foundational and instance-specific authoritative bindings working together

```
#------------------------------------------------------------------
-----
# In the core/foundational project
#------------------------------------------------------------------
-----
resource "google_project_iam_policy" "core" {
  project     = var.project_id
  policy_data = data.google_iam_policy.core.policy_data
}

data "google_iam_policy" "core" {
  binding {
    role = "roles/viewer"

    members = [
      "group:stakeholders@tas-9-2.com",
      "group:developers@tas-9-2.com",

"serviceAccount:ops@${var.infra_proj_id}.iam.gserviceaccount.com",
      "serviceAccount:app-
id@${var.app_proj_id}.iam.gserviceaccount.com"
    ]
  }

  binding {
    role = "roles/editor"

    members = [

"serviceAccount:ops@${var.infra_proj_id}.iam.gserviceaccount.com",
    ]
  }
}

#------------------------------------------------------------------
-----
# In the workload-specific project
#------------------------------------------------------------------
-----
resource "google_bigquery_dataset" "dataset" {
```

```
  dataset_id = var.dataset_id
}

resource "google_bigquery_dataset_iam_policy" "specific" {
  project     = var.project_id
  dataset_id  = google_bigquery_dataset.dataset.dataset_id
  policy_data = data.google_iam_policy.specific.policy_data
}

data "google_iam_policy" "specific" {
  binding {
    role = "roles/bigquery.admin"

    members = [
      "group:developers@tas-9-2.com",
    ]
  }

  binding {
    role = "roles/bigquery.connectionUser"

    members = [
      "serviceAccount:app-
id@${var.app_proj_id}.iam.gserviceaccount.com"
    ]
  }
}
```

Although such a layered approach to applying IAM policies is certainly feasible, it does require discipline or tooling to ensure that changes propagate the way you need. Likewise, a purely nonauthoritative approach will generally require additional verification to ensure the resulting policies adhere to a least privilege approach. In many cases, the most optimal strategy involves both authoritative and nonauthoritative types together. For example, as before an authoritative policy binding may be introduced first to establish a baseline set of — perhaps fairly restrictive — permissions for an organization in a foundation-layer TF project, and nonauthoritative policy bindings may be applied in a later layer to ensure individuals are able to conduct the tasks for which they are responsible, without accidentally restricting themselves from general access rights.

> **TIP**
>
> It's especially not recommended to include both types in the same module/TF state. Even if you ensure a specific order of evaluation, either through the use of resource attribute references, an explicit `depends_on` meta-argument, or, for example, by setting `-parallelism` to `1` and relying on the lexicographical evaluation order of independent resources, it's more appropriate (and way less confusing) to consolidate the bindings into the same type. Moreover, including both types of resources associated with the same principals, resources, and permissions will completely break idempotence and result in the dreaded never-clean states.

This kind of setup — with a set of core, authoritative permissions and workload permissions layered on — may be highly effective in the situation where you deliberately want to revoke all write and admin permissions for your product teams for an extended period of time, such as during a development freeze or a holiday break. Because it's only the permissions that would be impacted by the authoritative IAM bindings (and only in the resources managed by this core project), anything established in a downstream project, such as compute, storage, or network resources, would remain intact. Moreover, assuming the service identities are established in the downstream TF projects, close to where they are utilized, they wouldn't be affected by the core project, either, meaning they would continue to be permitted the necessary access for their associated services to work. An example of this is explored in detail in Chapter 12.

Policy Scopes

As already suggested, access control policies are made up of multiple components. They include some subject — such as a user, group of users, or a service identity, and usually referred to as a *principal* — an object to which it's permitted access — such as a cloud resource instance or group thereof — and the authorized actions the subject is allowed to perform on the object. From the CSP's perspective, they're generally additive, meaning a subject's capabilities are governed by the union of all assigned policies (making the effective capabilities the combination of all of the policies), rather than simply the last applied policy or the intersection of all policies

(which would narrow the effective capabilities to only those permissions that appear in each of the policies).

> **NOTE**
>
> Be aware that explicit Deny policies will take precedence regardless of where they appear. If a user is subject to a Deny policy, the Deny always wins. This is because Deny policies are generally observed before any Allow policies are evaluated, and the CSP will conclude denial before it ever assesses other policies a user is subject to.

In most cases, aside from the root user or account holder, an IAM user has no permissions by default. Thus, each IAM policy that is bound to the individual, either directly or through group access, will monotonically expand the user's access. This implies a way to layer policies from the most general and restrictive to the most specific and permissive. In foundational TF layers, generic, company-wide policies that mostly enable read-only permissions may be constructed such that new members imported into the system after the policy is introduced are inherently included in the policy through group membership. Likewise, TF projects that service application workloads can assign permissions to a development team that allow all members of the team to modify or instantiate certain cloud services or specific resource instances. Finally, there may even be special responsibilities that are temporarily assigned to individuals, giving them further access in a highly targeted fashion.

Depending on the CSP, different capabilities may be available to implement policy scoping. Group-based scoping is available in all CSPs, whereby a user may be assigned to multiple groups with different associated policies such as a wide, general group that gives narrow access permissions (perhaps only read access), and more specific groups (such as the one representing the immediate team to which the user belongs) that widen access for particular resources and for specified actions. Some CSPs introduce additional ways to establish scoped groups of resources to which policies may be assigned such that all resources in the group inherit the policy. In this case, the same approach applies: least privilege assigned to

widest scope, and necessary, team-specific permissions assigned to narrower scopes.

In terms of managing the TF configuration, it's not uncommon to largely centralize the responsibility of maintaining IAM controls to one or a small few teams. However, if you have decentralized portions of your TF configuration, embedding the responsibility within the teams developing the application workloads — or if you construct intermediary infrastructure teams that directly interface and coordinate with such teams — it will likely be beneficial to allow the same team the ability to manage the IAM controls associated with the respective resources, ideally in the same TF project or TF state — assuming you want the ability to make changes in a single, atomic procedure, such as a job within a GitLab pipeline or a step in a GitHub Actions workflow. Even if a corporate policy requires that the central team be responsible for the entirety of such controls, the configuration themselves would still benefit from appearing alongside the controls to which they apply in terms of the TF project hierarchy or even within the same TF state, perhaps as a nested remote module that is configured with default values and validation that protects against disallowed configuration.

In many CSPs, there are other policy types that can interfere with IAM controls, either directly or indirectly. For example, in some CSPs, special policies may be set at the root account level that restrict the format for acceptable IAM principals — such as the domain name to which an email address-formatted identity belongs, as shown in Example 9-3. This usually will result in an error when attempting to assign a permission to an unsupported format, quickly exposing the conflicting policies. Similarly, certain CSPs introduce account-wide networking controls that can indirectly restrict a user — such as by their originating IP address — from actions that would otherwise be permitted according to the individual's IAM permissions. In this case, it's likely not possible to detect the issue until the individual attempts to perform the restricted action. Even still, recognizing an issue in this case may not always clarify what the underlying restriction is that causes the issue.

Example 9-3. Equivalent policies in GCP and Azure

```
#-----------------------------------------------------------------
-----
# GCP version
#-----------------------------------------------------------------
-----
resource "google_organization_policy" "gcp_domain_restriction" {
  org_id     = "1234567890"
  constraint = "constraints/iam.allowedPolicyMemberDomains"

  list_policy {
    allow {
      values = ["is:C01234abc"]
    }
  }
}

#-----------------------------------------------------------------
-----
# Azure version
#-----------------------------------------------------------------
-----
resource "azuread_b2b_invitation_policy" "azure_domain_restriction"
{
  policy_type = "allowList"

  allowed_domains = ["yourcompany.com"]
}
```

> **NOTE**
>
> Usually, the account-wide policies that are concerned with network considerations take the opposite approach to IAM: with the default being full access (from this particular defense layer) and the effective policy being the intersection of policies to which an entity or resource is subject, meaning additional such policies monotonically restrict access — which is typical for network-oriented policies like firewall rules. Because IAM controls are unioning and firewall rules are intersecting, it's likely most appropriate for account-wide policies that may interfere with other policies to be constructed in a way that includes only scoping sparingly. This will reduce confusion and the chance for unintended restrictiveness that may be difficult to detect.

Service Identities

Service identities are the way most CSPs assign an identity to a cloud service operation or instance. This is important both to distinguish between operations or instances and to provide access controls to the service or instance by including it as the principal element of a policy. They allow resources to be treated as credentialed identities alongside human operators and elevate the security posture of an organization's operations by associating them with components of your operational tool chain, thereby reducing human error that comes with doing it by hand, and enable fine-grained permissions for what resources may do in the CSP, in general, and what they may do among themselves.

Because service identities can be associated with different resources, they should be generally managed near, if not alongside, those resources in terms of the TF ecosystem. It may be attractive to group them in a single, centralized TF project, and dole them out as needed to the various platform and application teams, but this can create an unnecessary bottleneck in many cases. Similarly, it may seem reasonable to allow the platform and application teams to manage their own identities. Assuming sufficient guardrails are in place to ensure appropriate usage is maintained, this can allow teams to move quickly with their necessary controls. However, with a dearth of such guardrails, leaving it up to the whims of each team can result in either an extremely bloated and unwieldy collection of identities or excessively permissive policies assigned to such identities — or both. Centralizing the oversight of service identity access controls, regardless of who possesses the maintenance responsibilities, can be a compromise between the two extremes. Additionally, constructing IAM-specific TF projects at various layers of your TF ecosystem so that they may reside closer to their assigned resources while maintaining strict isolation can further adhere to corporate policies based on the separation of concerns principle that place IAM responsibilities on a different team than the one managing the other cloud resources.

> **NOTE**
>
> In TF providers that support both authoritative and nonauthoritative IAM bindings, it's generally most appropriate to assign permissions to service identities through authoritative bindings. This promotes a least privilege posture, as a service behaves in a highly specialized way, with clearly defined — indeed, encoded — behavior and therefore access requirements.

Although the primary use case for service identities is to assign access controls to them, treating the service identity as a principal, because service identities are themselves instances of a cloud resource, may make them the object or target of a policy. For example, the developers and stakeholders of a particular identity should have access to at least view the service identity resource and review the permissions it's assigned. Similarly, the individuals or entities that are permitted to modify the identity, such as to assign additional policies, should be granted this capability through IAM controls. In particular, it's common to have the service identity associated with your automation platform be assigned wide permissions to create and assign policies to other service identities within its purview. This can undoubtedly be taken to extremes, with many identities granted tight controls in a chained fashion, leading to extremely brittle and fragile configurations — whereby a single corrupted entity in the chain can lead to total loss of control of a particular component for all but the top level admin users — but if used sparingly, only when there is a clear instance, operation, or group that can be uniquely identified, it can produce an extremely robust access control posture.

Just as in dealing with users' credentials, safely managing the credentials of service identities can be challenging. For service identities that are assigned directly to a resource hosted in the same CSP, many times there is actually no need to concern yourself with credentials, as the CSP handles it for you. However, in the case of an external service, or when a user attempts to act on behalf of a service identity, it may require the use of special credentials associated with the service identity. Communicating such permissions requires at least the same level of care as is given to a user's credentials —

depending on the purpose of the service account, it may benefit from even stricter controls. In either case, although the identity is created as a resource with TF, due to the idempotent nature of TF, the credentials themselves should be managed in a different way, outside of the TF project that creates the identity. This can be achieved using one-off invocations of the CSP's API for example.

> **NOTE**
>
> The best solution is to avoid transmitting users' credentials altogether. In some CSPs, there's a way to obtain service identity credentials using a user's personal credentials. For example, GCP supports service account *impersonation*, whereby an authorized user may obtain temporary credentials that are equivalent to those bound to a specified service account.

Network Policies

The other major suite of security-oriented policies in the cloud computing paradigm is around networking controls. Whereas IAM controls authorized access of APIs exposed by the CSP itself, networking controls are designed to enable or restrict network traffic over your organization's cloud-hosted networks. This might be anonymous traffic from the public internet and may need to be treated as inherently untrusted, regardless of whatever redeeming qualities it may include, such as an access token, or it might be highly visible traffic originating from within your closed, internal networks that have been securely connected to the CSP over physical links and should be considered reasonably trustworthy. In most cases, the overall cloud ecosystem includes some combination of both trusted and distrusted contexts. Different contexts may even apply to the same cloud network, so these must be configured so as not to create interference or leakage among contexts.

Some networking controls are intended to be specific to workload components, some are designed to be shared by particular groups of such components, and others are shared by all components in an environment, a

class of environments, or the entire cloud account. This makes considering all of the networking components as a single domain and managing them all together unreasonably cumbersome. Even combining the internal networking components — those that are insulated from networks external to the CSP — with external-facing components, or *edge networks*, would normally be unnecessarily over-complicated given the distinct concerns these two contexts serve. Moreover, there are multiple ways to architect your internal networks, as well as multiple types of edge networks. This makes organizing these controls a nontrivial exercise. Because networks are core components in almost any cloud ecosystem, as many of the workload-oriented resources are attached to a network as a functional requirement, or even imposed as a platform constraint, it can be disruptive to make any changes to them that the CSP requires TF to recreate. This makes reorganizing the TF configuration for an existing ecosystem with mission-critical workloads a risky exercise. Therefore, it can be highly advantageous to get the configuration right — or in a form that is expected to be long-lasting — early on. The remainder of this section considers the major contexts, internal and external — going into detail of the options for their construction and how to organize them within your TF project configuration.

Internal Networks

For internal networks, and depending on the CSP, there may be multiple valid configurations with countless variations. Many ecosystems incorporate a hub-and-spoke configuration whereby multiple workload-, application-, or environment-specific private networks (that is, VPCs) are connected to a central hub network, typically through the CSP's VPC peering mechanism. Another common approach is the classic three-tiered architecture, which segments presentation, service, and data components of an application into separate networks (possibly at the VPC level, but more commonly through dedicated subnets). Yet another common approach is to have high interconnection between various networks in a mesh approach, which can allow for a great deal of shared data channels while maintaining

segmentation between domains. In many cases at the enterprise scale, the true configuration will incorporate elements of all of these, and possibly even additional configurations.

Regardless of the configuration chosen, there's still usually a question of how to relate networks and environments. It's almost certain that you'll want to largely isolate production workloads from nonproduction ones, including in terms of the networks to which the components are attached. If this is extended through all other environments, maintaining a distinct, self-contained network configuration for each, it might suggest that the components be associated with the other environment-specific ones, and intermingle the TF configuration either in closely related projects in the TF hierarchy, in a layered approach, or even in the same TF project. However, to separate the concerns of administering the different domains of network controls and application-specific components, not to mention their very likely unique lifecycles, the network components should be sufficiently partitioned, with distinct access controls around how the components may be manipulated and how the code may be introduced.

Especially if some of the nonproduction environments see their components attached to the same private cloud networks (usually, subnets of the same VPC), it becomes reasonable to isolate the TF projects in their own logical branch of the hierarchy, separate from the application-specific infrastructure projects and states. In many configurations, there will also be certain components that span environments, possibly to create an intentional traffic bottleneck that interfaces with an edge as described next. These would potentially serve multiple, if not all, environments, so it would likely also be best configured separately from the environment-specific TF projects. Figure 9-3 portrays two ways to organize your internal network TF projects, serving both environment-specific and shared-environment configurations.

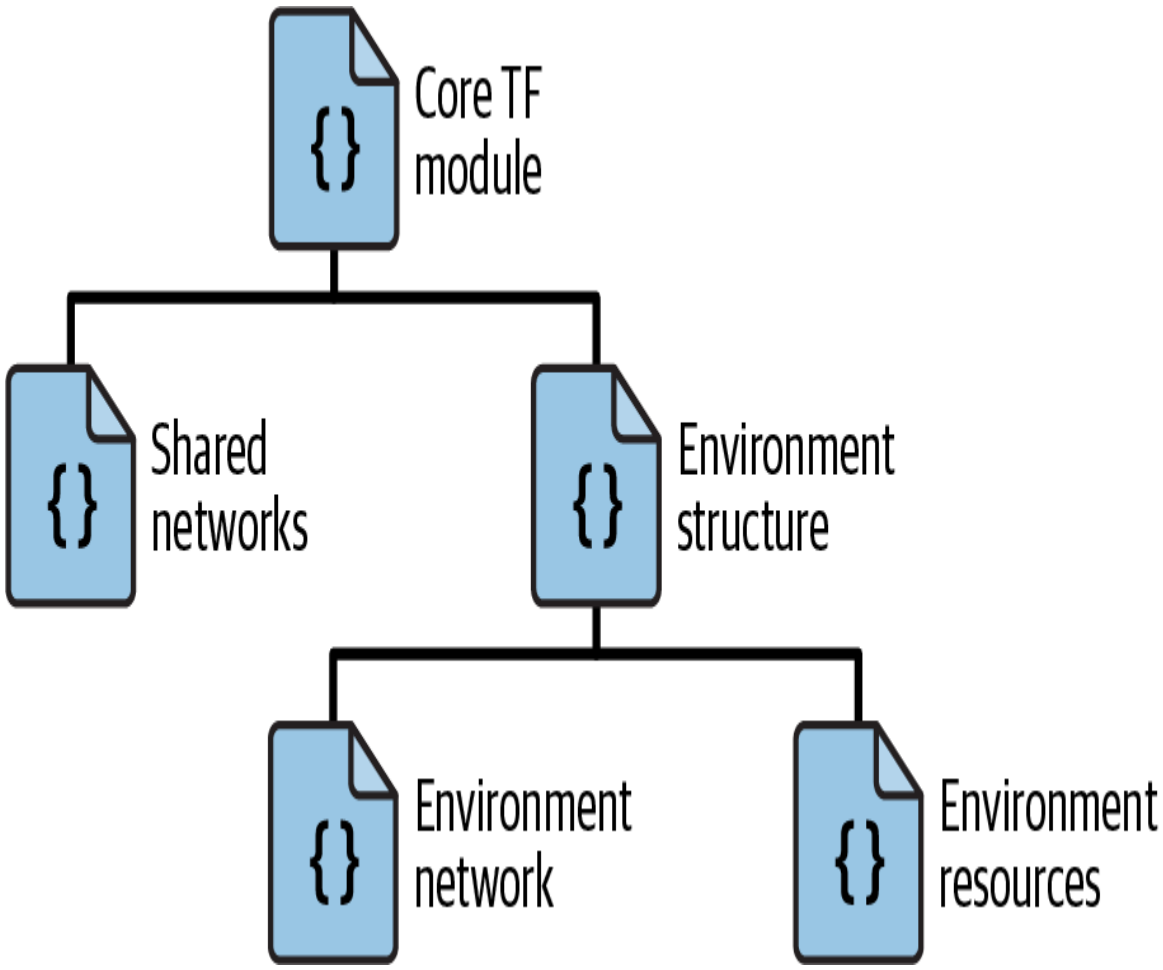


Figure 9-3. TF projects for shared and environment networks in the TF project hierarchy

Edges of Cloud Networks

Wherever your private cloud networks connect to networks external to your cloud account, it can be considered a *cloud network edge*. In reality, this is a method of exposure and, depending on the connection type, different additional security measures should be considered. It's likely that there's a single team responsible for enabling and securing the entirety of a particular entryway. Here are the major types of edges and represent the categories available at the cloud network's edge:

Public edge

> This is where your cloud networks — which are otherwise private — interface directly with the public internet. For traffic leaving your networks, this would generally be through an internet gateway, which

must emit from some component that has been assigned a publicly addressable IP, and thus is not anonymous, or a Network Address Translation (NAT) resource, which itself has a public IP and will effectively act as a proxy address for the internal traffic. Although there are typically multiple managed services and solution architectures for protecting the public edge from incoming traffic, the best solutions will usually include multiple managed proxy services to create a layered, defense-in-depth approach. This will be explored in further detail soon.

Private edge

These are the links to your other private networks, typically those hosted on your physical premises. For physical links, this is enabled through the use of a colocation service, whereby your on-premises networks are physically extended to the closest of the colocation service's datacenters that are co-occupied by the CSP that hosts your cloud network. In the absence of such a physical link, which can take weeks to provision and generally requires lengthy contracts, a VPN may be constructed using a VPN appliance on premises and a managed VPN service in the CSP.

Edges between cloud accounts

Sometimes, there's traffic coming from or going to a highly trusted third-party's cloud network that is, by default, traversing the public internet. Especially in cases where high rates of traffic are expected, or where the traffic is especially sensitive, it may be highly desirable to connect directly, taking advantage of benefits such as the CSP's guarantees around network performance and throughput and reduced network traversal costs. Common solutions are what's typically referred to as *VPC peering*, whereby two VPCs are able to effectively operate as a single VPC — with caveats — as well as, again, a VPN. This will be explored in slightly more detail soon.

CSP-proxied edge

The final network edge available in many CSPs is a managed proxy solution that is reachable publicly and can be configured with some interface into your cloud networks. Unlike the proxy services in the public edge, these would normally be associated with a single component or group of components, rather than the entire cloud ecosystem. It would amount to a workaround to your public edge security and is most appropriate for highly trusted individuals or traffic initiated by UIs developed internally. Depending on the CSP and your use case, this might be through a managed API service, a TCP forwarding service, or related to a specific service such as a managed SQL solution.

EDGE NETWORK SECURITY

The best practices around public edge network security design are outside of the scope of this book, but as already stated, it generally would include multiple layers that perform, for example, packet inspection, application layer filtering — or *layer 7* in the OSI model — which can vastly improve upon IP (*layer 3*) and protocol (*layer 4*) filtering and request proxying, as well as the various other network controls, such as VPCs, subnets, resource-attached firewalls, routing components, load balancers, and so on. This is undoubtedly a massive subject that has comprised entire texts and cannot be done justice succinctly.

However, what is fair and within the scope is what *not* to do:

Multiple load balancers

: It's not recommended to manage multiple publicly accessible load balancers, one for each workload.

Ephemeral public IP addresses

: Avoid using ephemeral public IP addresses.

Public IP addresses

: It's especially not recommended to assign public IP addresses directly to production compute or database instances and only protect them with layer 4 firewalls. In other words, there will be several components in any enterprise-grade public edge protection solution.

Regardless of the architecture you're targeting for your public edge security, you'll almost certainly have multiple components that operate in a coordinated and layered fashion, each performing a specific check on the traffic as it flows into your private cloud networks and to your workloads

and other components. Moreover, unless you intend to manage all of these components in a single TF project and root module, there will most likely need to be some TF modules that combine responsibilities of multiple teams, such as application or platform level requirements alongside the edge-layer components — an example of this is assigning the backend resources of a global proxy service to specific application workloads, perhaps through an internal-only load balancer; these resources can be requested through a ticketing or self-service solution, or it may be deemed appropriate to grant the downstream product teams access to develop these resources themselves. In Figure 9-4, you see an example of how these various components might be grouped by function in separate TF projects, and these projects would likely be related.

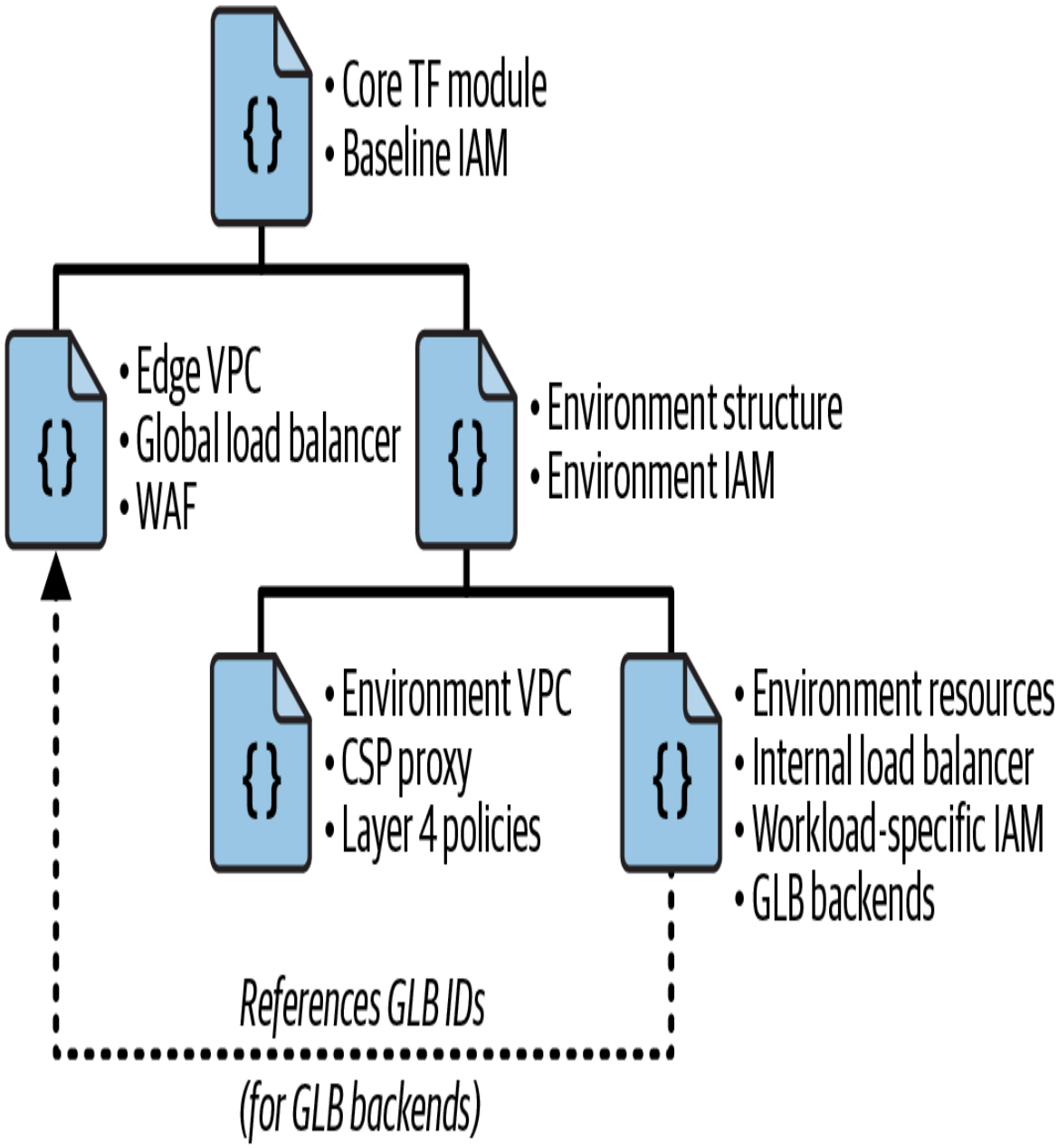


Figure 9-4. A hierarchy of TF projects that comprise a basic public edge security solution

And in Example 9-4, you see just how many components are required to establish a typical edge solution in GCP that allows only two application workloads be securely exposed.

Example 9-4. Setting up a global external application load balancer with WAF in GCP

```
# IP
resource "google_compute_global_address" "global_alb" {
  name = "${var.alb_name}-ip"
```

```
}

# WAF
resource "google_compute_security_policy" "waf" {
  name = "${var.alb_name}-waf"

  # Basic OWASP
  rule {
    priority = 1000
    action   = "deny(403)"

    match {
      versioned_expr = "PRECONFIGURED_WAF"

      expr {
        expression = "evaluatePreconfiguredWaf('sqli-v422-stable')"
      }
    }
  }

  # All other traffic
  rule {
    priority = 2147483647 # This is the lowest priority (lowest
evaluated)
    action   = "allow"

    match {
      versioned_expr = "SRC_IPS_V1"
      config {
        src_ip_ranges = ["*"]
      }
    }
  }
}

# Application Load Balancer Frontend
resource "google_compute_url_map" "https_map" {
  name = "${var.alb_name}-url-map"

  host_rule {
    hosts        = [var.service_host]
    path_matcher = "gke"
  }

  host_rule {
    hosts        = [var.assets_host]
    path_matcher = "gcs"
  }
```

```
  path_matcher {
    name            = "gke"
    default_service = google_compute_backend_service.gke_backend.id
  }

  path_matcher {
    name            = "gcs"
    default_service = google_compute_backend_bucket.gcs_backend.id
  }
}

resource "google_compute_target_https_proxy" "https" {
  name             = "${var.alb_name}-https-proxy"
  url_map          = google_compute_url_map.https_map.id
  ssl_certificates = [google_compute_ssl_certificate.cert.id]
}

resource "google_compute_global_forwarding_rule" "https" {
  name                  = "${var.alb_name}-https-forwarding"
  ip_address            =
google_compute_global_address.global_alb.address
  port_range            = "443"
  target                =
google_compute_target_https_proxy.https.id
  load_balancing_scheme = "EXTERNAL_MANAGED"
}

resource "google_compute_url_map" "http_redirect" {
  name = "${var.alb_name}-http-redirect"

  default_url_redirect {
    https_redirect = true
    strip_query    = false
  }
}

resource "google_compute_target_http_proxy" "http" {
  name    = "${var.alb_name}-http-proxy"
  url_map = google_compute_url_map.http_redirect.id
}

resource "google_compute_global_forwarding_rule" "http" {
  name                  = "${var.alb_name}-http-forwarding"
  ip_address            =
google_compute_global_address.global_alb.address
  port_range            = "80"
  target                = google_compute_target_http_proxy.http.id
```

```
  load_balancing_scheme = "EXTERNAL_MANAGED"
}

# Security
resource "google_compute_ssl_certificate" "cert" {
  name        = "${var.alb_name}-cert"
  private_key = file(var.ssl_private_key_path)
  certificate = file(var.ssl_cert_path)
}

# Application Load Balancer Backends
resource "google_compute_backend_bucket" "gcs_backend" {
  name                = "${var.alb_name}-gcs-backend"
  bucket_name         = var.gcs_bucket_name
  enable_cdn          = false
  edge_security_policy = google_compute_security_policy.waf.id
}

resource "google_compute_backend_service" "gke_backend" {
  name                  = "${var.alb_name}-gke-backend"
  protocol              = "HTTP"
  load_balancing_scheme = "EXTERNAL_MANAGED"
  port_name             = "http"
  timeout_sec           = 30
  security_policy       = google_compute_security_policy.waf.id

  health_checks = [google_compute_health_check.default.id]

  backend {
    group = var.gke_neg_self_link
  }

  log_config {
    enable      = true
    sample_rate = 1.0
  }
}

resource "google_compute_health_check" "default" {
  name = "health-check"

  http_health_check {
    port = 80
  }
}
```

VPC PEERING

Without going into too much detail, *VPC peering* is a way to connect two distinct VPCs — each with their own subnets, routing rules, gateways, and so on — to treat them as a single VPC from the perspective of attached resources, such as a compute or database instance. Although each CSP has its nuances, especially with respect to how a VPC is constructed, the following properties generally hold true:

CSP-specific

> VPC peering is not available across CSPs. Another solution, such as VPN or a direct link method, such as AWS Direct Connect, GCP Interconnect, or Azure ExpressRoute, is required in such cases.

Other resources still apply

> The configurations regarding available routes, gateways, and layer 4 firewalls are all still applicable within the context of each VPC, regardless of the presence of a peering connection. Indeed, these controls are important to provide some protection, even to or from such a highly trusted source.

IP ranges must not overlap

> None of the CIDR blocks that are configured in each VPC, either directly or through its subnets, may overlap.

Peering groups

> Beyond one-to-one CIDR overlap restriction, some CSPs also disallow overlapping CIDRs in a group of peering connections. For example, GCP considers a peering group from the perspective of each individual VPC, say VPC_A in $Group_A$. If VPC_A is peered with both VPC_B and VPC_C, all the CIDRs must be nonoverlapping — even if VPC_B is *not* directly peered with VPC_C.

Nontransitive routes

Assuming three VPCs, VPC_X, VPC_Y, and VPC_Z, such that VPC_X and VPC_Z are both peered with VPC, but not with each other, traffic originating in VPC_X may directly route to destinations in VPC_Y but will not be able to route to VPC_Z. This is because each directly peered VPC can share its routes across the peering, so VPC_X and VPC_Y can swap routes, and VPC_Y and VPC_Z can swap routes, but VPC_X has no way of swapping routes with VPC_Z. Thus VP_X will know what to do with traffic destined for destinations in VPC, but won't know anything about VPC_Z, let alone how to route traffic to it.

Unencrypted traffic

Traffic that is unencrypted, such as HTTP or FTP traffic, is not encrypted when traversing the peering connection. This may raise objections from security teams or regulatory bodies.

VPC peering can be a common occurrence in internal network design, with several different ways to organize within and across VPCs. However, there are fewer use cases in peering connections that span across accounts, leading to less flexibility and nuance in architecture. Typically, the choice of a peering connection is rarely to enable unrestricted bidirectional access across VPCs that each contain mission critical workloads, but rather introduced as a security, performance, and cost control measure. As such, cross account peerings are generally between highly constrained and relatively workload-free VPCs that are configured effectively as a DMZ, whereby controls are in place to allow expected traffic and disallow everything else.

If a VPC peering connection is not appropriate, such as due to internal policies that disallow such direct visibility into your cloud networks or because the VPCs are in different CSPs, other solutions available include a

VPN and a direct link. A VPN will generally traverse the public internet, but will be encrypted, end to end, and when using managed services, would circumvent the public edge security solution. A direct link solution, such as AWS Direct Connect, GCP Interconnect, or Azure ExpressRoute, requires lead time to provision and can be used to establish a private, high-bandwidth connection between two VPCs, even across CSPs; but similar to VPC peering, they're generally unencrypted. Combining a VPN and a direct link solution makes it possible to construct private tunnels between CSPs that are encrypted end to end.

In addition to the CSP-managed services that you configure directly using TF, there are likely to be other components that you want to incorporate that are not available through your preferred CSP, such as specialized security appliances like an IPS. This may be due to licensing considerations or a requirement as part of your organization's security posture, or it might simply be due to the fact that a CSP-specific implementation of a third-party solution almost always lags behind the solution offered directly by the third party. For some solutions — even those that are made accessible through a CSP, perhaps offered through an internal marketplace — there are several configuration controls that must be introduced and that are not well suited for an idempotent workflow as offered by TF. For example, certain operations, such as replacing placeholder configuration values, may be required upon initialization but can cause severe problems if performed later, once the components are in active duty. As such, specialized appliances should be treated with extra care and consideration. You may need to introduce certain lifecycle conditions for such resources managed in TF. Furthermore, these components may have unique lifecycles, in general — for example, to follow the vendor's release schedule in addition to supporting the various relevant workloads. Finally, in some cases, your security policies from a different perspective — such as a requirement for immutability or disallowing remote access protocols such as SSH — can impede your ability to configure these components after they're initialized, effectively rendering TF no more useful than a highly structured, single-use procedure.

Because of these factors, it may be most prudent to isolate specialized, self-managed appliances, such as within a remote TF module — for example, so that static code analysis validations can be configured uniquely for the needs of the appliance — and possibly in their own, dedicated root module — for example, with a dedicated service identity, separate from a more general edge network identity, so as to separate the concerns of managing something so specialized with those of more standardized components. Keeping it separate also allows you to keep any policy exceptions that the appliance must make to operate properly — such as specialized access or networking controls that are disallowed in normal contexts — focused tightly around it, rather than having to apply the exception to a wider group of components. However, if you do choose to isolate the code or the TF state, you'll have to make a decision for where certain ancillary components — such as the networks to which the appliances are attached — reside: alongside the appliance configuration or as a special case within a wider network-oriented TF project layer. In either case, the operational responsibilities become jumbled and, unless these responsibilities are all shared by a single team, at least one of the teams will have responsibilities in both TF projects, violating SRP. As long as this is understood and the decision is documented with a solid business justification and acceptable rules of engagement among parties, or a procedure that guards against friction, separating out the appliances may still result in more streamlined lifecycles and fewer, more tight-fitting policy exceptions.

Along with the operational benefits of isolating the network edge components in their own TF projects and introducing specific service identities for executing them, there are also security implications. For example, it can be extended to improve the security posture by introducing specific access controls for the repositories that store the TF configuration. This way, only the teams that have a specific responsibility to modify these highly important, and often somewhat brittle, components have the ability to do so. For any one-off operations like manually input commands or scripted procedures, it may require elevated access, possibly granted only temporarily for the duration of scheduled engineering efforts, especially in production contexts. A superior solution is to introduce a way to execute

these one-off operations from an automation platform, such as through a special wildcard job workflow or pipeline that can execute a wide range of commands as presented in Chapter 6.

In terms of where these edge network-oriented components appear in the overall TF hierarchy of components, such as in a layered TF project configuration, each of the four edges should be considered on its own. For the public edge, one perspective may consider it foundational, since it's a highly necessary and important domain that many of your workloads are likely to use to a high degree. On the other hand is the point of view that your public edge should come later in the implementation and be layered atop an internal network configuration that has been certified to be highly secure. Both perspectives have merits, and it's reasonable to consider them. The question remains on where to place them in the hierarchy. If it's considered so foundational it's configured early on, the instinct might be to place it at or near the top. If it's established later in the overall implementation, it may be natural to also introduce it later in the hierarchy. In general, the most appropriate solution is usually a combination of approaches: keep it separate from the core TF project (the one at the root of the TF project hierarchy) as a domain, but construct it as a direct child component to the core; this way, it operates as a standalone entity, alongside other major domains such as your internal networks and application environments, but also has the flexibility to build fit-to-purpose TF projects around the sensible subdomains, such as your layer 7 firewall components or an IPS appliance.

For the private edge, this is likely to be considered even more foundational than the public edge, since it connects back to your internal networks and is often configured early on, to avoid bad practices like IP allowlisting individual workstations to gain direct access into private cloud networks and establish necessary data channels between components in the cloud and on premises without traversing the public internet. Again, in a layered TF approach, it's likely worthwhile to configure this as a child domain of the core TF project, isolating it from other layers, while enabling its establishment early in the overall implementation.

The other edges are less foundational and more specific to a workload. This implies that their TF components should appear near the components for the workloads they support. However, because these components are also highly specific and nonstandard, it's likely worthwhile to consider isolating them, too, either just as a remote TF module, or as an entirely separate root module and as a logical child TF state within the relevant workload's TF domain, such as an application delivery environment. Figure 9-5 shows an example hierarchical relationship of TF projects in a layered configuration, highlighting where the various edge network components reside.

> **WARNING**
>
> Figures 9-4 and 9-5 suggest the GLB backend elements be configured alongside the environment resources they belong to. Although this is operationally efficient, it may be disallowed for security reasons in which case the reference would logically be reversed, and the GLB backends (now in the TF project establishing the GLB) would reference the environment resources to which they belong.

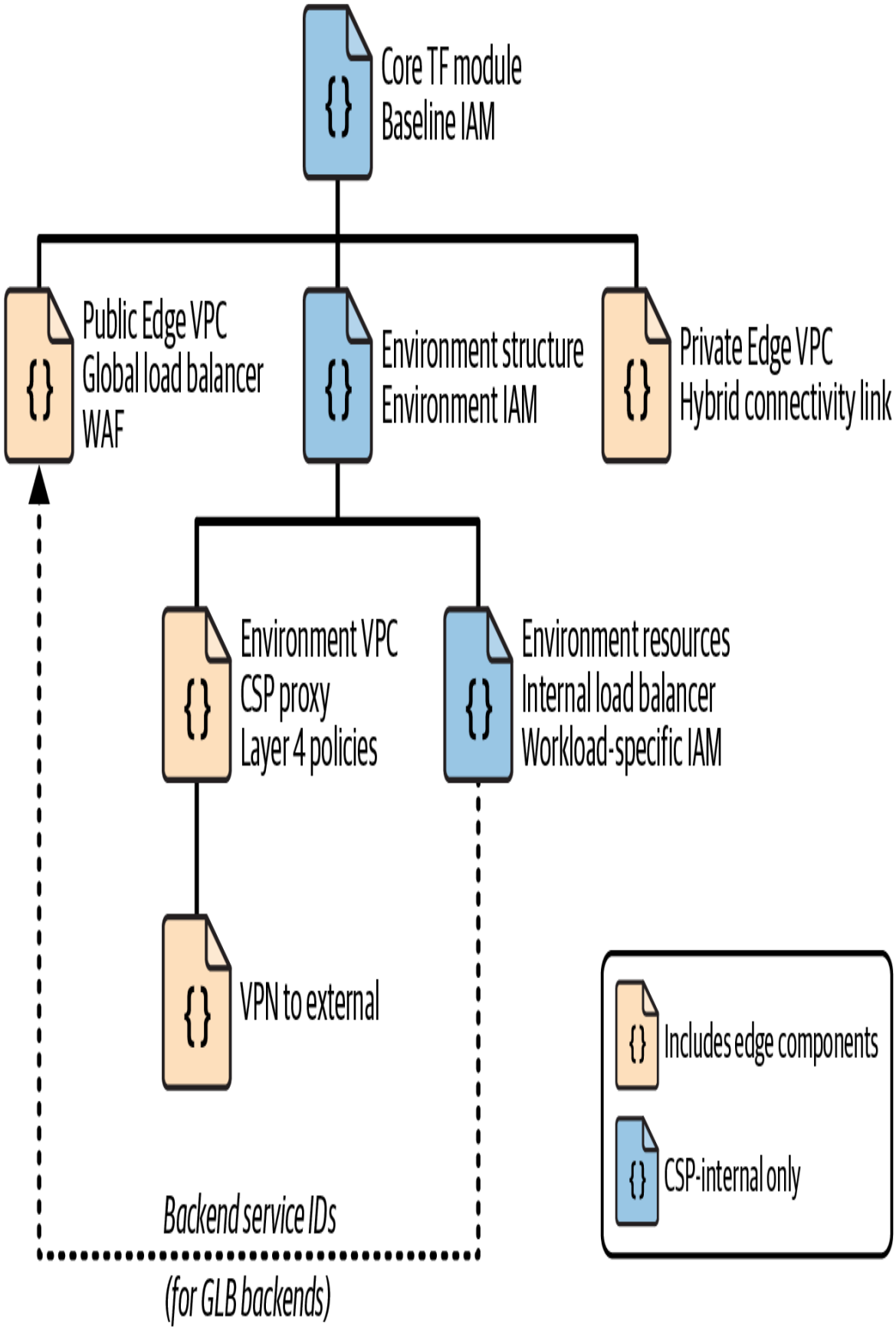

Core TF module
Baseline IAM
Public Edge VPC
Global load balancer
WAF
Environment structure
Environment IAM
Private Edge VPC
Hybrid connectivity link
Environment VPC
CSP proxy
Layer 4 policies
Environment resources
Internal load balancer
Workload-specific IAM
VPN to external
Includes edge components
CSP-internal only
Backend service IDs
(for GLB backends)

Figure 9-5. Example relationship of TF projects highlighting edge networks

Account-wide Policies

Some policies can be applied all the way at the account level. Account-wide IAM controls can establish baseline access permissions for members of your organization, meaning when new members are onboarded, they automatically inherit these permissions. Depending on the CSP, other account-wide policies you can include are the following:

IAM policy principal restrictions

> In most CSPs, access policies are assigned by an email address. Policies may be introduced that restrict to only allowed email address domains, for example, which can help to protect against unintended external actors.

Network layer 7 restrictions

> Policies that restrict where traffic may come from at the DNS domain level.

Resource constraints

> Certain policies may be established to restrict where certain resources, such as VMs, may be physically located or to establish limits for how individual instances may be configured.

Rather than attempting to implement perfect security with account-level policies, take a maximal allowed access approach, whereby you restrict only behavior that is inappropriate for any of your existing or scheduled resources and workloads. For example, allowing only a shortlist of known originating systems to access — or disallowing known bad actors from accessing — all of your resources, irrespective of intent or exception, is appropriate at the account level. At the same time, you should not

implement overly strict policies for which you'll have to create exceptions in smaller scopes such as at the resource level.

Some CSPs support a way to apply policies at different scopes in addition to the account. In such cases, the smaller scope is subject to both the policies set on the scope and any policies inherited from a parent scope. In terms of how a policy is evaluated in the case of an overlap, different CSPs do present different behavior. For example, in Azure, the scopes to which an Azure Policy may be applied — in order from widest to narrowest scope, with each scope a collection of the next smallest scope — are Management Groups, Subscriptions, Resource Groups, and resources. Regardless of where they appear in this hierarchy, the most restrictive policy in evaluating any overlap will be the one applied, meaning a very restrictive policy at the Management Group level will not be overridden by a more permissive policy at, say, a Resource Group level. In GCP, on the other hand, the resource hierarchy — from widest to narrowest — is Organization, Folder, Project, and resource, with the ability to nest up to 10 folders. Organization policies may be applied down to the Project level, and those introduced at a higher scope will, by default, be inherited by a narrower child scope. However, GCP allows these policies to be overridden, meaning a highly restrictive policy at the Organization level may be replaced by a highly permissive policy for a specific Project. In either case, the cleanest approach is usually to again assign the most permissive policies at the highest possible scope and introduce more restrictive policies in narrower child scopes as needed. For one, taking this approach makes extending or migrating the policies from one CSP to another a more direct mapping. Additionally, it's more likely that a group of resources that is made more permissive will include a resource that should not have been exposed this way and will go unnoticed for longer, than such a resource that is made more restrictive: if a resource becomes less permissive, any newly restricted operations will undoubtedly be asserted at the very next opportunity. In other words, if a resource is accidentally granted wider exposure, it will not be necessarily noticed by anyone already capable of interacting with it, as it will behave the same; but if a resource becomes subject to a restriction that

unintentionally prevents normal activity from responsible parties, it will be incumbent upon these parties to address it promptly.

CASES WHERE ACCOUNT-WIDE POLICIES MAY BE OVERRIDDEN

This is not to say never override a policy in the CSP. There may be extremely robust configurations that construct a highly locked-down baseline posture that is appropriate for the vast majority of workloads and a highly cordoned Folder or Project that overrides certain policies to make them more permissive. This could be part of the establishment of certain edge networks, such as an org-org or a CSP-proxied edge, which may require configurations that would be disallowed by the baseline. Such an approach would most likely be highly scrutinized and be associated with a business justification. The point here is that, without a strong justification otherwise, successively more restrictive policies down a particular scoping hierarchy will be easier to understand and more defensible in an audit than successively widening the access or having a seemingly random or haphazard approach to scoping resources and applying policies. In the case of GCP, an alternative method would be to keep the Organization-level policies wide and set the highly locked-down baseline posture at a top-level Folder that may be inherited by all the Folders, Projects, and resources within it, with the cordoned resources and Projects in a top-level, sibling Folder potentially with its own set of policies that only allow the special behavior.

These policies should again reside in or alongside the components to which they are bound, such as the Management Group or Subscription in Azure or the Folder or Project in GCP. Ideally, once these are set, they'll not experience rapid changes. Updates would be highly deliberate and tied, for example, to a specific business justification or a new capability in the CSP. Unless there is a specific mandate that requires these to be managed in a separate TF project or TF state, the lifecycle of the policies should not be

logically separate from the lifecycle of whatever scope to which the policies apply, making it generally appropriate to cohabitate the policy configuration in the same TF project as the associated scope object. Keeping the policies separate can create unnecessary exposure for the time that the scope object is introduced by TF and before the policies are included.

> **NOTE**
>
> In the case of GCP, an Organization does not have a TF resource but is created as the resource object directly associated with the account. Therefore, policies bound at the Organization level would be configured separately, in practice, but would effectively constitute the Organization-level configuration anyway.

Once again, it's important to impose access policies around the software repositories that house the TF code for account-wide policies, restricting the ability to modify the configuration to a very specific group of users. Additionally, this may be one of the few cases where you may choose not to introduce read access across all members of your organization, keeping it to only those who have a need to know, since it can explicitly outline allowed behavior — even IAM controls may only be fully evaluated for an individual with perfect knowledge of all the groups to which they belong.

Policy in Action

In many cases, external policies, such as those imposed by governmental or other regulatory bodies, do not come complete with instructions for how to implement them. Instead, they're largely *declarative* in nature, to borrow the computing terminology, describing the desired end state. This is consistent with the notion that every organization is unique and that their systems should largely follow the internal communication structure: every company's solution will be unique and dependent on how your teams and responsibilities are distributed and organized.

That being said, there are certain considerations that help to orient the architects of a particular solution. For example, it will become clear early on that it's likely not appropriate to attempt to maintain all TF configuration in the same TF project or TF state. Rather, a combination of centrally managed and distributed configurations that appear throughout your TF ecosystem will likely be most appropriate. While staying relatively declarative, the following explores different common policy scenarios, both regulatory and voluntary, focusing on how to organize the TF configuration for each component.

Health Insurance Portability and Accountability Act (HIPAA)

Suppose you have a product or platform that has been identified as potentially valuable in a healthcare system setting, perhaps as a data analysis tool for discovering new treatments or therapies, and you're presented with an opportunity to serve a US-based organization. If your product currently doesn't account for sensitive data and, as such, is not compliant with HIPAA, there are several resources and configurations that you'll need to introduce. Unfortunately, these are not always best served as a single additional module that represents an entire solution, but typically require updates throughout an existing TF ecosystem. This section explores the typical components necessary to implement HIPAA, including where they're likely to appear in the TF project hierarchy.

> **NOTE**
>
> This section is largely applicable to System and Organization Controls (SOC) 2 compliance, however it focuses on Protected Health Information (PHI), which is an extremely sensitive form of PII, and has uniform requirements. In contrast, SOC 2 is much broader, while also having more nuance depending on the industry in which it's considered.

The first thing to introduce is encryption. Each of the major CSPs has its own native solution, usually called a Key Management System (KMS),

with which to manage encryption keys. The best practice is to manage your own keys, such as using a Customer Managed Key (CMK) feature, so that you may produce full verifiability in the case of an audit. It's important to utilize key rotation with a schedule defined by your security teams. These controls should reside near whatever context with which they are shared or to which they are specific, ideally in the same TF project or TF state. For example, a key that is shared by a group of resources, such as a Project in GCP, that are logically related and require a common way to encrypt and decrypt data should appear alongside this grouping (Project). Likewise, if a specific component requires its own key, such as a key used to encrypt a highly sensitive data store, the key should be created alongside this component. If this is a standardized practice, it can be bundled with the resource as a nested module that simplifies the construction of such specialized resources.

You'll also need to introduce a data-loss prevention (DLP) solution that will obfuscate sensitive data in your databases, which can be applied at the available scopes to ensure it applies to all included resources — this is in addition to encrypting the data at rest, as it's used for protecting the sensitive data even if it's successfully read from a database. Although these controls may be associated with specific databases and storage solutions, it may be most prudent to control these policies from a central location. After all, it's not exactly typical for a unique set of policies for a specific workload: either the workload and resource is subject to the controls, or it isn't. Ideally, these controls can be set up before — or at least upstream from, in terms of the TF project hierarchy — the resources they govern, so they may benefit from the protection from their instantiation.

NOTE

Even if you're indeed managing multiple levels of sensitivity, such as PII and PHI, unless there's a specific business justification, there is little reason for maintaining multiple sets of DLP policies. Regardless of whether you only expect PII data in a particular component, PHI data should be properly handled; it should not only be applied where you *believe* it should appear.

Although the primary consideration when conforming to HIPAA is obviously in implementing the protective controls to fully comply, it's also necessary to introduce audit and data integrity capabilities of the behavior of the system and the activity within it. This means introducing logging and monitoring controls that will capture and store the necessary information for the required amount of time (six years at the time of writing). Very likely, this will need to be isolated from your operational logging and monitoring components because it may require a different set of qualities than what you need to operate effectively, such as data retention duration, and the operations teams almost certainly do not need full access to the activity involving sensitive health data for day-to-day tasks (it's worth noting that any audit logs regarding PHI are themselves PHI and require similar levels of safeguarding). Depending on the solutions involved, there may be multiple valid approaches to organizing and developing a logging and monitoring solution, in general. However, for these types of highly sensitive information, the simplest valid approach may be to centralize the retention and analysis of the information, such as in an individual GCP Project that acts as a destination container for all relevant logging and monitoring activity and isolate it from the rest of the account through tightly governed IAM controls — ideally, with little to no constant access for individuals, but with a way to generate temporary access. A single TF project would likely be appropriate for the components that store and surface these audit capabilities, but unless there's a way to impose a log forwarding solution that is inherited by all components in your account (or whatever the necessary scope is), it's probably necessary to embed the log forwarding configuration as a default component in a standard way across your ecosystem. For example, in GCP, it may be bundled alongside the Project entity itself, in a standardized nested module, perhaps managed as a remote module to reduce the chance of unexpected changes.

Assuming you do not already separate administrative traffic (such as for triggering a task or issuing a query request) from the data traffic (that is, the actual data points your analysis operates upon, and that are read from data storage solutions and transmitted between workloads within your solution) to retrofit your existing solution, it's probably appropriate to create new

private networks (VPCs) that will be specific to the transmission of PHI data. These must be configured to allow only encrypted traffic, such as with Transport Layer Security (TLS), which implies the use of specific network controls to enforce mutual TLS among resources and a CSP proxy solution for any direct communication from trusted individuals to components within these networks. The CSP proxy can typically be managed alongside the VPC as part of a standard TF module. Unfortunately, CSPs do not have perfectly analogous support for establishing and enforcing TLS and mutual TLS. That being said, each solution is generally specific to a group of resources, such as a load balancer, and would be best configured as a standard component in a TF module for the resource itself, or grouping thereof. Because of this separation of responsibilities, but a general consolidation of concerns, it's prudent to introduce additional validation or guardrails that ensure these standard components are indeed utilized in the VPCs that require them and that will disallow improper constructions. You'll see such validation approaches in Chapter 10.

In addition to the strong controls you introduce in your cloud account, it will be important to impose strict, audit-friendly RBAC controls on your repositories if this has not yet been introduced. It's certainly important that they have only edit capabilities on the code for which they're responsible, and it's likely valuable for most engineers to have the ability to read most of the available code; however, it may be wise to introduce auditable, temporary credentials for accessing the code for certain components like security policies. Just as it's generally unreasonable to keep this information completely secret, as it can create an artificial bottleneck due to missing information, it's just as unreasonable to leave it fully exposed to hundreds or thousands of individuals who truly do not have a business need to access it at any time and for any arbitrary amount of time.

Only after you have all of these components in place should you sign the respective Business Associate Agreement (BAA) supplied by your CSP, legally signifying your responsibility to comply with HIPAA.

General Data Protection Regulation (GDPR)

Suppose you have a product that you want to make available in EU markets. Assuming you gather PII, you now need to comply with the GDPR requirements to protect the sovereignty of this gathered data. This may mean adopting new practices around how you configure database instances and document stores, as well as developing support for entirely new services and procedures that were not strictly necessary before.

The primary concern in GDPR is around PII (and more sensitive) data. In particular, this data generally must remain within the European Economic Area (EEA), except in specific cases such as processing data for journalistic or research purposes, and even then only through appropriate channels such as tools that provide end-to-end encrypted (E2EE) messaging. This means not using globally hosted data and document stores. Also, in cases where data is transmitted out of the EEA, it must be encrypted, likely with the use of a VPN. In fact, all data must be encrypted at rest and in flight, meaning a KMS is an integral part of any GDPR-compliant architecture. Again, it's best to introduce CMK to provide full visibility in the case of an audit. In terms of TF configuration, these components will appear in the typical places, alongside the specific application resource or whatever context to which it belongs.

Although it's not expressly required, it's worth implementing controls around the system logs and metrics, avoiding global storage resources or centralized collection locations outside the EEA for any such data gathered within it. This may mean redesign of a standard logging solution to direct logs to meta-region-specific locations.

In addition to the measures necessary on the typical cloud resources, there are certain reporting and validation tools that should be enabled to provide additional assurance that you're compliant. Each CSP has its own repertoire of offerings that cover the concerns differently, some of which simply need to be enabled, whereas others require individually configured controls. The following are examples of such solutions from the major three CSPs:

AWS

AWS Artifact and Config

GCP

Google Cloud Trust and Safety Center, Cloud Asset Inventory, and Security Command Center

Azure

Microsoft Service Trust Portal and Azure Policy

GDPR imposes a requirement to detect, report, and investigate breaches of relevant data within 72 hours. This usually implies an elevated level of scrutiny and observance on behalf of human operators and stakeholders to detect such a breach, likely with entirely new roles and teams necessary to introduce round-the-clock vigilance, in addition to the fine-grained controls necessary to detect them in the first place. A dashboard or alert is meaningless if there is no one to interpret and act on it. Similarly, once a breach is detected, it must be reported to affected individuals, which means there must be clear responsibilities and standard formats regarding how these reports are generated and communicated. It should not be an ad hoc exercise or a haphazard scramble to announce a breach. Finally, there must be contingencies in place for investigating breach events, which may require additional on-call support if the team responsible for the investigation is different from the one that monitors systems for breach. Implementing clear and minimal IAM controls for the various roles and responsibilities, possibly including a mechanism for issuing temporary, elevated credentials, is an important feature of any GDPR-compliant cloud solution.

GDPR also imposes requirements for users to be able to access, rectify, or delete their personal data gathered by your product. This means exposing basic editing capabilities of data points and documents to each user, but only that user's data. Taken to the extreme, this would be an authenticated UI that surfaces all data related to the user's account and allows direct edit controls that, once submitted, are immediately propagated throughout your product. However, in practice, this has many implications, especially for

modifying data, not the least of which is protecting against injection attacks and other false information intended to harm the system. It's therefore necessary to include some form of moderation, such as a human operator as is commonly recommended, in the process to avoid issues. The normal time frame to honor a user's request to retrieve, modify, or delete their data is 30 days, given plenty of time for human verification and fulfillment. This means special internal roles and processes must be established to enable the secure access and transmission of existing user data, as well as to securely review a user's request including updated sensitive information. Most likely, the IAM and other access controls that are managed by TF should be consolidated in a single place in your cloud ecosystem and managed by a single TF project to ensure tight control.

Once again, only sign the respective GDPR data processing agreement supplied by your CSP, formalizing your responsibility in complying with GDPR requirements, after you have all of these components in place.

Zero Trust Architecture (ZTA)

Zero trust architecture (ZTA) is an extreme approach to cybersecurity, whereby no traffic is inherently trusted, regardless of where it originates, including from your application workloads attached to your private cloud networks. As traffic traverses your networks, it's verified along the way at every possible connection point before being passed to the next checkpoint in the path on its way to the destination — this is from your cloud's perspective, and the destination may be an edge component such as a NAT or VPN.

Such an architecture requires many of the components presented so far, with a few specific configurations and additional considerations. First, the internal network components will still follow one of the highly segmented configurations, often still a combination of hub-and-spoke and three-tier configurations. However, they may be even more tightly segmented, with as tight a grouping around components as is feasible, and as few connections between these networks as possible. These highly segmented components should be grouped in a way that aligns networks that are related in their

lifecycle, either logically or in practice. This may mean managing multiple networks together in the same TF project and TF state.

To ensure traffic is as tightly controlled as possible, certain other internally aligned components like firewall rules can be associated with service identities — both as the source and destination, wherever possible — rather than attached through a loose relationship such as network tags or targeting IP ranges. For firewalls, which are specific to the direction of the traffic, either ingress or egress, traffic that flows between two components that are both included in your CSP account, both directions should be implemented. This naturally requires high granularity in service identities and their management, as assigning such identities to too many components undermines the value of using service identities to distinguish between network traffic sources and destinations. Configurations that are specific to single-direction traffic, ingress or egress, should likely appear as close to the related component, destination or source, respectively, as they are part of the protection of that component, defining what may enter or exit the component itself.

An extremely important element in a ZTA is a secure way for engineers and external systems to observe or operate components in your private cloud networks directly, without having to open public entry points. This is usually best served by a CSP-proxied (or CSP-tunneled) edge solution, such as Session Manager and Verified Access in AWS, identity-aware proxy (IAP) in GCP, and Entra Application Proxy and Easy Auth in Azure. Because these are CSP-specific, their identity is the CSP itself, meaning it should be available to only a few small groups of individuals. This is obviously best served by once again introducing highly granular IAM controls, specific to certain responsibilities, that may be applied together in a composed fashion for the relevant users and teams. Both the TF configurations for the proxy components and the access controls should be established close to the relevant workload components, most likely embedded directly in the same TF project, unless a policy specifically precludes it.

Another one of the most important components in a ZTA is extremely fine-grained monitoring controls, including access logs, which record the operations executed within the CSP, and network logs, which track the progression of network traffic moving through your cloud networks. This can be extended to trigger alerts in cases where improper traffic is detected or blocked, or even be made to initiate destructive or remediation actions to protect against subsequent such behavior. The majority of these monitoring components, including the storage and the transport configurations, are likely configured in a dedicated TF project likely as a child of the foundational/core TF project. Baseline access controls for users and service identities that enable review of monitoring controls may be established in the dedicated TF project, with additional, workload-specific controls established within the various environments.

It may even enhance the solution to include additional security components such as IDSs or VPNs, even between certain internal networks, to enhance the isolation between segmented networks. Place these components within or near the TF projects that include the relevant networks. For individual resources that establish both sides of the connection, place them near the downstream TF project. For components that are composed of multiple pieces, one for each side of an end-to-end connection, such as a VPN, either create them in each place that establishes the components, or, again, place both in the downstream TF project. As suggested in "Edges of Cloud Networks", any specialized components should be isolated from other components and should again be constructed near the networks to which they're associated.

A ZTA may see more stringent policies at the account level to enforce an elevated level of control across all resources within the account. However, it can still be somewhat counterproductive to over-specify these controls, since in a zero trust posture, you already assume that A) you're experiencing a breach and B) every action could be malicious, so your approach is less focused on creating defense perimeters and more about verifying and monitoring all activity. There should be no chance that a policy inherited from an account or other scope might impede this system.

Summary

In this chapter, you reviewed various organization-related TF configuration, as well as multiple types of policies and how they manifest in TF configuration, including:

- How common computing principles apply in TF for cloud solutions
- Different roles, stakeholders, and decision makers have conflicting agendas and how these typically must be balanced rather than reconciled
- Practical considerations for access control policy management
- How to establish scopes of policies to manage them in a modular, layered approach
- How to manage service identity resources and their IAM controls
- How to approach various internal network architectures
- There are various ways to construct cloud edge networks
- How to introduce account-wide policies, as well as at other scopes (where available)
- How to approach various policy architecture structures

You can now approach corporate-imposed policy and organization structure constraints with determination and confidence. You recognize there is innumerable nuance to how components, including network and access policies, may be organized to support your company's regulatory and voluntary constraints and requirements for interacting with external systems. You recognize that competing agendas need not end in stalemate or gridlock, but in compromise — and perhaps some additional controls to mitigate exposure in the case of relaxed restrictions.

In Chapter 10, you review various options for validating your TF projects, both using TF controls within your modules and external controls that can

enhance the governance and efficacy of your TF configuration. You'll be presented with options for verifying consistency in important cloud resources such as IAM controls. Finally, you'll review ways to extend your observability controls to the TF ecosystem itself, enhancing your visibility and line of sight to the critical framework underlying your cloud infrastructure.

---

1 SoC was first attributed to the immensely influential Edsger Dijkstra; he described SoC in his essay "On the Role of Scientific Thought" as a "characteristic for all intelligent thinking," it can be paraphrased as the in-depth consideration of each aspect of one's subject matter in isolation, rather than to address all aspects simultaneously, to establish consistency with the aspect.

2 Presented in her summary of the report she coauthored with Walter Hürsch, entitled *Separation of Concerns*.

Chapter 10. Validating at Scale

For digital products hosted in CSPs, the resources that host, store, and serve your content, your infrastructure, are the foundation for your operations. It's like the land and structures on and in which a purveyor of commodity goods conducts business. Without it, you have no permanence and no guaranteed business continuity. In the case of physical infrastructure — the buildings you occupy and the ground beneath them, for example — you're easily able to see, with your own eyes, the state of affairs: the building is in need of repair after an unforeseen event or due for an upgrade from normal wear and tear; that it is sinking due to unstable ground. You would even likely have protocols and contingencies in place, either codified in corporate guidance or provided for you by local government bodies, in the case of both deprecation and disasters that were established and refined over hundreds — if not effectively thousands — of years of evolution in commercial practices. Given that the concept of web-based software is still only a few decades old and the cloud computing paradigm even much younger, and since we don't have the ability to physically see with our own eyes the components that make up our infrastructure, we must fast-forward the advancement of validation and auditability with limited stock visibility.

Over time, different techniques have been developed to validate software. Some of them, such as the classic debug-with-logs technique, are applicable to a wide range of software contexts, whereas others are effectively a software design pattern themselves, such as test-driven development (TDD). A common one for computing machines — which in a CSP may be designed for a specific purpose such as a database or a network controller or may be exposed as a more general purpose computer on which to launch and manage your own processes — is the ability to run various diagnostics reports. In this way, even a remote operator may get very low-level and fine-grained details about a particular component. Setting up a process to regularly gather certain resource metrics from your cloud infrastructure

components can provide a great deal of insight into their efficiency and of the software that the component hosts. Setting thresholds at which alerts are issued or certain measures are taken can enhance your ability to discern vulnerabilities before they become costly problems.

Central to validation is visibility. Although CSPs generally make it possible to view your resources both through a web-based console and an API/CLI, the resources are generally grouped in specific ways, such as by service and region, making viewing all resources an exercise in determining all possible combinations and requesting them sequentially. Moveover, these group representations generally don't express all the information, the rest of which must often be requested resource by resource. Thus, putting a full picture of the cloud ecosystem together — especially doing so without providing an overwhelming amount of data to the point of losing the signal in the noise — is generally a nontrivial task. Because of this, there are many tools, both proprietary and open source, available to gain better visibility and develop custom perspectives or virtual vantage points. There are also a plethora of tools and components available in the various CSPs that can be combined — and perhaps blended with other tools you already utilize — to build a highly effective solution in house. However, it need not always be an extremely complicated or advanced solution, and certain simple techniques can go a long way.

Although most of the recent focus in software engineering regarding validation as a practice is part of — or as a following activity to — the normal release of software, in the case of cloud infrastructure, simply validating your landscape once is generally not enough. Ideally, you have full visibility, in real time and at all times; in practice, you're more likely sampling and polling the data, hopefully often enough and effectively enough to approximate this. As soon as you recognize this, it becomes a question of how much you're willing to expend to effect better and better approximations. In this chapter, you'll review various common and emerging options for validating infrastructure established with TF configurations, including their merits and limitations, with the goal of recognizing which are appropriate to your organization.

On Testing

Testing is a well-established industry standard practice across many disciplines. Consumer software, business services, scripted routines, and helper methods are heavily composed of functions and function-like constructs that take input data — and possibly request other information from the system — and produce a result or outcome. These constructs can be tested against a representative set of inputs by comparing their actual outputs against their expected outputs, resulting in a battery of *unit tests*. Considering how these units interact with each other to produce more advanced results or outcomes might lead to integration, end-to-end, or functional tests, depending on your perspective, goals, and purpose. In a highly structured way of working such as Scrum, SAFe, or extreme programming (XP), these can even be used as a way to guide software development by writing tests that encode the criteria and expectations expressed by the user story that captures whatever value you're intending to deliver — the basis of TDD.

> **NOTE**
>
> In some cases, a particular unit that you want to test is really an abstraction over multiple smaller units or external systems, intended to combine their outputs in some meaningful way. This is where mocks and stubs can be useful. This way, you're still only validating the unit, and not the helper components — which should have their own suite of unit tests anyway to ensure they operate as expected.

In addition to tests that validate some subset of your software workloads, you probably also want to test your system as a whole. For example, it's generally considered a best practice to understand how your workloads perform under normal and extreme usage load with *load testing*. Aside from confirming the availability and usability of your system in different scenarios, it will also help to validate — or experiment with — your resource allocation, enhancing your ability to forecast your consumption and develop workload budgets and cost management. In the case of *chaos testing*, you're actively simulating various disaster scenarios — such as an

offline datacenter, a network attack, or other high-profile event — to determine how well it handles workload degradation and if or how quickly it can recover. Detailed coverage of chaos testing is out of scope for this book; however, there are many great resources available, including *Chaos Engineering* by Casey Rosenthal and Nora Jones (O'Reilly) if you'd like to learn more.

Some tests are even specific to the type of software it supports, such as *penetration tests* that are used to validate the robustness of your network security controls. It's worth recognizing that by using a CSP, you're (heavily) using other people's code. When you use a CSP-managed network service, you're benefiting from the shared responsibility model (SRM) which, in essence, is an explicit assertion that the CSP itself has a vested interest in keeping its components secure and available. For this reason, it's common for CSPs to require joint coordination when it comes to penetration tests, as their systems are the first line of defense.

Tests are important. They help to reduce unexpected consequences from changes you make in good faith. Any software you develop in almost any programming or scripting language can benefit from some test paradigm — sometimes even a small set of simple tests, sometimes referred to as *smoke tests*, can go a long way. However, as you've already seen, HCL, the language in which TF projects are written, is not a programming or scripting language. Yes, it takes inputs, and yes, it can produce outputs — but in particular, it does not have a guaranteed order of operations (unless you overuse the `depends_on` meta-attribute), so the various outcomes are not always obvious. The TF tool itself is the software workload in this scenario. It should surely be validated, but it need not be your job to perform the validation. The point is, the tests used to validate TF work differently than the tests you run on your application workloads: different enough that they should be considered on their own, without preconceptions from other software testing practices.

Remember, everything you build with TF in your CSP *is* production. It's live, and it's doing work that should be fulfilling or supporting the mission of commercial viability. Certainly, if a particular TF configuration is

experimental in some way, there's likely to be some unconventional constructions that should be validated. And if the occurrence is likely to be repeated, or if their modification runs the risk of being catastrophic — such as the components undergirding your network edge appliances — it *is* likely worth encoding these in a repeated validation, such as those for `tf test` or Terratest. However, some resources have restrictions on how they may be named, such as storage buckets — which are often unique across the CSP, not just your account — and the grouping elements such as the GCP Projects or Azure Subscriptions.

Creating multiple instances of such resources, even in seemingly isolated environments, can be disallowed for a certain window, even after they are deleted. This is where validating nested modules only can result in more effective tests than trying to validate root modules in a parallel configuration. In the case of Terratest, you may even choose to construct tests that validate your environment resources directly, rather than building duplicate resources in an ephemeral environment. You'll encounter further commentary and examples of this in the section "Behavior Analysis".

> **NOTE**
>
> Unless a TF project is especially stable and changes to certain configurations are sufficiently risky, tests are likely not worth constructing for every resource. Thus driving TF development with tests is probably not nearly as valuable as it is for consumer products. Moreover, requiring a specific percentage of test coverage criteria is likely not the right way to govern them. Similarly, if projects get updated frequently, tests that create real resources may incur much more cost than they provide value. It's best to be judicious about the use of such tests. In particular, it should not be a given that these are included in each of your normal development pipelines. With a well-architected organization of TF projects, running resource-creating tests on them on a case-by-case basis will likely be more than sufficient in the majority of cases. Beyond that, problems that you still experience can be encoded as new tests, filling in whatever gaps do exist. Finally, this commentary is primarily for the modules you build and execute. If you're developing modules for third parties, some semblance of tests should be considered a necessary part of the workflows that validate them before they're released.

Sometimes the best way to validate a new resource or configuration is to establish them in a pristine, sandboxed environment. If this can be created

as an ephemeral, on-demand environment, this will be superior to running tests that might build resources in a different order than would happen in practice. Ultimately, no amount of validation can guarantee success, and you shouldn't trust anything until it's running in the expected place in your system and verifiably operating the way you expect.

Validation of TF Configuration

There are several methods for validation available, ranging in complexity and scope. Some are focused on the input values to modules and resources, and some can work with live resources. In this section, you'll review several validation methods ranging in complexity and purpose.

Types of Validation in TF

The categories of validation that are valid in TF are:

Internal input/output (I/O) validation

> Using TF constructs such as the `condition` attributes present in input variable `validation{}` blocks, the resource `lifecycle{}` block's `precondition{}` and `postcondition{}` subblocks, and the output's `precondition{}` block.

Data sources

> Using the TF `data{}` entity to ensure resources managed outside this particular TF state exist and are accessible at the planning phase. This is opposed to simply trusting a value provided by an input variable and potentially having it fail during the application phase.

Static code analysis

> Several tools exist that are intended to identify common configuration issues, security gaps, and compliance violations.

Behavior analysis

Unit and integration tests that can build or analyze real resources to validate functionality and behavior.

Externalized I/O validation

Creating ancillary services that support your TF projects, validating any custom inputs, and confirming the results of the expected outcomes.

CSP auditing/monitoring

Using the services provided by the CSP, or a UI that wraps the CSP services, to determine if the running services match what is reflected in the code.

TF-driven validation

Using TF commands, such as `tf apply`, to validate the consistency across the *state triplet* — that is, the HCL code, TF state, and running CSP services.

Access logs

Ingest access logs and match them against the results of known `tf apply` operations.

With the exception of the TF entity attributes available in HCL, these generally require a great deal of read access across the CSP. Ideally, these permissions are not assigned to individuals, or even groups, by default, but to service identities — such as an AWS IAM Role or GCP Service Account — with the expectation that reports may be provided to users that have a responsibility to ensure infrastructure state. Only in the event of an active investigation or ongoing incident would there be a specific need for regular access to this data by an individual. This means a pipeline job or workload would be developed to execute these validations. Most likely, it's appropriate to group these tasks by frequency or triggering event — such as running CSP services, TF operations, and filtered access logs in the same context, like a GHA step or a GitLab job — so the data may be cross-

referenced, and untracked activity can be exposed — except, of course, in cases where there's a specific business or regulatory policy that prohibits such unification of data sources.

> **NOTE**
>
> Workloads you develop to perform any of these validations bring along the requirement to implement their lifecycle management. This often will emerge as scope creep for an infrastructure team. You should be vigilant; when it occurs, it must be identified quickly and addressed immediately — it could require a dedicated team, based on a solid business justification.

Validation Blocks in HCL

One of the best ways to establish validation of your TF projects is to leverage the various blocks in which input or output values may be checked. They come in three varieties: the variable entity's `validation{}` blocks are guaranteed to be evaluated at the planning phase; TF will attempt to evaluate the `precondition{}` blocks during the planning phase, but if they rely on computed values that are not yet available, they cannot be evaluated until the apply phase; and `postcondition{}` blocks are never evaluated until the apply phase, as part of an entity's lifecycle configuration. Because TF evaluates the variable `validation{}` blocks during the planning phase, they're uniquely valuable in detecting invalid input values, especially since one variable's `validation{}` block can reference any other variable value (since v1.9), local values, or even data sources, enabling highly comprehensive logical evaluation.

As mentioned, the variable `validation{}` block allows developers to construct a Boolean expression from any input variables, local values, and the attributes of available data sources. They're evaluated during the planning phase, which means it's available to operators with read-only permissions. This means it can be made available to individual engineers as part of a development workflow such as by using a Git pre-commit hook,

which will be discussed in "Low-Tech, Low-Cost Solutions". An example of a variable `validation{}` block that incorporates multiple sources of information can be found in Example 10-1.

Example 10-1. Variable `validation{}` block that incorporates multiple sources of information

```
# NOTE - "region" variable definition not shown
variable "type" {
  type        = string
  description = "Allowed EC2 instance types."

  validation {
    condition     = local.instance_check
    error_message = "The type must be t3.micro (us-west-1) or
t3.small (us-east-1)."
  }
}

locals {
  micro_check    = (contains(["us-west-1"],
data.aws_region.region.region)
                        && contains(["t3.micro"], var.type))
  small_check    = (contains(["us-east-1"],
data.aws_region.region.region)
                        && contains(["t3.small"], var.type))
  instance_check = local.micro_check || local.small_check
}

data "aws_region" "region" {
  region = var.region
}
```

In a similar way, the `precondition{}` block available to `resource{}` entities and `data{}` source entities allows developers to construct a Boolean expression from the input variables, local values, attributes of other resources in the TF state, attributes of available data sources, and outputs of any nested modules. They don't have an explicit limit in terms of how many terms may be included, meaning they can be arbitrarily comprehensive. However, as established in Chapter 4, to ensure your TF projects are accessible for reviewers, any complex calculations should be extracted into a local — or combination of local — values, instead of including it inline. Moreover, unlike the `variable{}` entity's

`validation{}` blocks, which are guaranteed to be evaluated during the planning phase, `precondition{}` blocks that contain resource attributes may not be evaluated until the TF application phase, for example, if they're referenced from a resource that does not yet exist. If a `precondition{}` block can be evaluated during the planning phase, it can be included as a variable `validation{}` block instead. This effectively advertises the expectation that the evaluation is guaranteed during the planning phase. If instead the logic relies on multiple variables and data sources that are otherwise unrelated, combining them as a `precondition{}` becomes more reasonable, as otherwise it may become arbitrary which variable should include the logic as part of its `validation{}` block. Examples of functionally equivalent `validation{}` and `precondition{}` blocks are in Examples 10-2 and 10-3.

Example 10-2. Equivalent expressions in `validation{}` *and* `precondition{}` *blocks,* `validation{}` *preferred*

```
resource "aws_instance" "example" {
  ami           = "ami-12345678"
  instance_type = var.instance_type
  region        = data.aws_region.region.region

  lifecycle {
    precondition {
      condition     = local.instance_check
      error_message = "Allow only t3.micro (us-west-1), t3.small
(us-east-1)"
    }
  }
}

variable "instance_type" {
  type        = string
  description = "Allowed EC2 instance types."

  validation {
    condition     = local.instance_check
    error_message = "Only t3.micro (us-west-1) or t3.small (us-
east-1)."
  }
}
```

```
locals {
  micro_check    = (contains(["us-west-1"],
data.aws_region.region.region)
                        && contains(["t3.micro"],
var.instance_type))
  small_check    = (contains(["us-east-1"],
data.aws_region.region.region)
                        && contains(["t3.small"],
var.instance_type))
  instance_check = local.micro_check || local.small_check
}

variable "instance_region" {
  type        = string
  description = "Allowed regions for EC2 instances."

  validation {
    condition     = contains(["us-west-1", "us-east-1"],
var.instance_region)
    error_message = "The region must be either us-west-1 or us-
east-1."
  }
}

data "aws_region" "region" {
  region = var.instance_region
}
```

Example 10-3. Equivalent expressions in `validation{}` *and* `precondition{}` *blocks,* `precondition{}` *preferred*

```
# NOTE - several variables and data sources not shown for brevity
resource "aws_instance" "example" {
  ami           = data.aws_ami.example.id
  instance_type = var.instance_type
  region        = data.aws_region.region.region

  primary_network_interface {
    network_interface_id = aws_network_interface.example.id
  }

  lifecycle {
    precondition {
      condition     = local.instance_check
      error_message = "Allow only t3.micro (us-west-1), t3.small
(us-east-1)"
    }
```

```
  }
}

variable "instance_type" {
  type        = string
  description = "Allowed EC2 instance types."

  validation {
    condition     = local.instance_check
    error_message = "Only t3.micro (us-west-1) or t3.small (us-
east-1)."
  }
}

locals {
  micro_check    = (contains(["us-west-1"],
data.aws_region.region.region)
                        && contains(["t3.micro"],
var.instance_type))
  small_check    = (contains(["us-east-1"],
data.aws_region.region.region)
                        && contains(["t3.small"],
var.instance_type))
  instance_check = local.micro_check || local.small_check
}

resource "aws_vpc" "my_vpc" {
  cidr_block = var.vpc_cidr
  region     = data.aws_region.region.region
}

resource "aws_subnet" "my_subnet" {
  vpc_id            = aws_vpc.my_vpc.id
  cidr_block        = var.subnet_cidr
  availability_zone =
data.aws_availability_zones.available.names[0]
}

resource "aws_network_interface" "example" {
  subnet_id   = aws_subnet.my_subnet.id
}
```

An example of a `precondition{}` that won't be evaluated until the apply phase can be found in Example 10-4.

Example 10-4. A `precondition{}` block that won't be evaluated until apply phase

```
# Several resource, data, and variable definitions not shown for
brevity
resource "aws_instance" "example" {
  ami           = data.aws_ami.example.id
  instance_type = var.instance_type
  region        = data.aws_region.region.region

  primary_network_interface {
    network_interface_id = aws_network_interface.example.id
  }

  lifecycle {
    precondition {
      condition     = local.instance_check
      error_message = local.error_message
    }
  }
}

data "aws_ami" "example" {
  name             = var.ami_name
  region           = data.aws_region.region.region
  deprecation_time = var.ami_deprecated_on
}

locals {
  micro_check    = (contains(["us-west-1"],
data.aws_region.region.region)
                        && contains(["t3.micro"],
var.instance_type))
  small_check    = (contains(["us-east-1"],
data.aws_region.region.region)
                        && contains(["t3.small"],
var.instance_type))
  ami_expires    = aws_ami.example.deprecation_time
  type_valid     = local.micro_check || local.small_check
  ami_valid      = timecmp(local.ami_expires, timestamp()) > 0
                   # timestamp() is not evaluated until apply time
  instance_check = local.ami_valid && local.type_valid
  type_error     = "Only t3.micro (us-west-1) or t3.small (us-east-
1)."
  ami_error      = "AMI must not be deprecated."
  error_message  = "${local.type_error} ${local.ami_error}"
}
```

The `postcondition{}` block is also available to `resource{}` entities and `data{}` source entities. It can be a good way to ensure the resources were configured the way you expected. They may be most appropriate in cases where you allow attributes to be computed by the CSP for you. However, unless the attribute is read-only, meaning it cannot be set in the `resource{}` block definition, it should be precomputed with the validation happening *before* TF attempts to create or modify the relevant resource. This keeps the configuration from causing problems during the apply phase, where it could error out with a failed postcondition. This doesn't incur a rollback, which can leave resources partially configured.

PRECONDITION OR POSTCONDITION?

Preconditions are likely best used when focusing on the results of other components, especially modules, and not to validate input variable values. It's better to use the `validation{}` blocks in the `variable{}` entities instead, since one variable's `validation{}` block can reference any input variable's value, not just the specific variable that includes it. Since this is guaranteed to be evaluated at the planning phase, it can provide whatever preliminary check that a resource or module may need. This allows the `precondition{}` blocks to be focused on checking for valid combinations of multiple values that include the attributes of other resources or outputs of nested modules. It's worth reiterating the guidance to avoid mixing `resource{}` and nested `module{}` blocks in the same TF module, effectively restricting the use of `precondition{}` blocks to checking the combination of resource attributes and possibly certain variables, such as to compose a constructed string.

In terms of precedence, a reasonable convention may be to prefer variable `validation{}` blocks to a `precondition{}` block on a resource or data source entity, reserving the latter for situations such as when a resource requires information from another resource. Similarly, you might prefer resource `postcondition{}` blocks to an output's `precondition{}` block, reserving the latter in this case for only situations where an output combines information from multiple entities. It could even be viable to largely avoid `precondition{}` blocks on data sources, all `postcondition{}` blocks, and output `precondition{}` blocks, preferring to focus the investment on only the variable `validation{}` and resource `precondition{}` blocks. Aside from keeping a consistent pattern that avoids a duplication of logic — or at least a duplication of responsibility — establishing a precedence based on evaluation order can also ensure any invalid values are found as soon as possible.

Data Sources

As the Terraform documentation states, "Data sources are an abstraction that allow Terraform to reference external data. Unlike managed resources, Terraform does not manage the lifecycle of the resource or data. Data sources are intended to have no side-effects." This provides a great deal of information already, but it misses some key points. TF will attempt to read the data source during a planning phase. Unless the information with which you retrieve the data is not available during the planning phase — such as a value from a computed resource attribute — this means that the external data will be read before any definite changes are made to the environment. If the external data isn't available, the planning phase will result in an error, and corrections to the code or remediations to the environment can be made before an apply phase is executed. Moreover, beyond simply referencing the external data, it also incorporates the information into the TF state. This means while the lifecycle of the resource isn't managed by the TF project in which it appears, its existence (or not) is recorded. This establishes a stronger relationship between the TF project and the external data; while this may seem antithetical in the age of decoupled components, they're already interconnected, but without explicit data source entities, this connection is largely obscured — until it's leveraged during an apply phase, which may be too late if the information is inaccurate or the external component is unavailable.

In most use cases, the external data you're retrieving with a data source is regarding other resources established through your CSP that happen to be established external to the TF project in which they're referenced, very likely by some other TF project. However, it's possible to use a different TF provider and service provider to retrieve data from truly external sources. One of the canonical use cases in this scenario is to retrieve sensitive information from a secret registry such as HashiCorp Vault. This promotes a separation of concerns between the establishment and maintenance of sensitive information like encryption keys from the lifecycle of the resources that rely on them. Moreover, the use of an external system for things like keys will be more appropriate in cases where multiple CSPs are

used and require the same (or functionally equivalent) such sensitive information.

> **NOTE**
>
> In cases where sensitive information is retrieved through a data source, it's imperative that you wrap the information in the `sensitive()` function so that the value is not inadvertently recorded in logging output. While HashiCorp's own documentation recommends using the sensitive attribute for variable and output values, there's no other way to protect values referenced from a data source element.

The more widespread use case of data sources is to reference CSP resources that are established outside of the TF project. This can ensure that values that are used as input arguments to the resource definitions in the TF project are valid and up-to-date. Rather than using an attribute that might be modified, you can reference a more stable value — such as an attribute that uniquely identifies a resource — as the TF project's input value and retrieve the necessary information from the resource's data source element. Even in cases where the identifier is the information required as the resource's input argument, such as a VPC or subnet resource referenced in the configuration for a network attached resource like a compute instance, it's recommended to introduce a `data{}` source block for the resource and to reference the identification value from that data source. Again, this serves the purpose of ensuring the resource is indeed present and available. Moreover, certain resources may be referenced in multiple ways, with varying degrees of specificity, as shown in Example 10-5. For values that are less specific, TF will attempt to fill in the blanks with other available information, such as a GCP Project ID assigned to the TF project in a `provider{}` block. This makes certain constructions safer than others. Because the most specific version is almost always available as an attribute of a resource or data source, referencing this through a data source would theoretically be safer than attempting to construct it manually. Additionally, because TF will attempt to read the data source during the planning phase, you'll catch any

mistakes with how you're referencing the information before you determine to make any real changes in the CSP.

Example 10-5. Two constructions of a reference that are accepted as a resource argument

```
provider "google" {
  project = "instance-10-5"
  region  = "us-west1"
  zone    = "us-west1-b"
}

resource "google_compute_instance" "self_link" {
  name         = "tas-10-5-gce-1"
  machine_type = "n2-standard-16"

  boot_disk {
    initialize_params {
      image = "debian-cloud/debian-12"
    }
  }

  network_interface {
    # Self-link includes the correct Project ID
    subnetwork = "projects/network-10-5/regions/us-
west1/subnetworks/snet-10"
  }
}

resource "google_compute_instance" "name" {
  name         = "tas-10-5-gce-2"
  machine_type = "n2-standard-16"

  boot_disk {
    initialize_params {
      image = "debian-cloud/debian-12"
    }
  }

  network_interface {
    # subnetwork_project is not specified; will default to provider
project
    # this will produce an error in the apply phase
    subnetwork = "snet-10"
  }
}
```

> **TIP**
>
> Referencing information through a data source is also safer, even when you're using variable validation, since that isn't able to determine if the associated resource is actually available; it can only confirm things such as the format, logical comparisons, or computed values constructed from the variable or a set of variables. By the same token, you should still include variable validation even in cases where you're ultimately referencing the value through a data source because it's still valuable and a recommended practice.

Although data sources are explicitly not managed by the TF project in which they're referenced, the `data{}` source block supports a `lifecycle{}` subblock, similar to that of the `resource{}` blocks, but with more limited functionality. In particular, they support the same `precondition{}` and `postcondition{}` subblocks, allowing for additional validation expressions before and after the data source is fetched. The precondition version can be useful in cases where the argument values used to retrieve the data source are computed from the result of a resource managed by the TF project. The postcondition can be useful in situations where the information extracted from the data source might be subject to some degree of randomness or nondeterminism, such as when retrieving available region information from the CSP.

> **TIP**
>
> If, for whatever reason, you find yourself making the same validation expression in the `precondition{}` as the `validation{}` block, it's likely best to prefer the variable validation method to reduce duplication and the extra work associated with it. Certainly, any sufficient business justification that demands this duplication will supersede the guidance found here.

Static Code Analysis

You should always include `tf validate` as part of your automated IaC workflow, and possibly as part of your normal development procedures. However, depending on your level of risk tolerance, and the trust you have

in your processes, you may choose to include additional tools that will more deeply analyze the TF project, checking for any known vulnerabilities and resource misconfigurations. These can range in complexity and expressiveness, from those that include comprehensive coverage out of the box to highly extensible frameworks, or even a combination thereof. It's fairly common to include multiple such tools because they can range in scope and underlying measurement mechanism. This can create overlaps in coverage, which as per usual is a double-edged sword: it can be useful to ensure a misconfiguration is detected in extremely corner cases where a particular tool may miss a problem, but it can also create double the work in the case of a refinement to the rule (either to cover a newly discovered gap or to widen the scope of acceptable configuration).

The first tool to consider is TFLint, which goes beyond the syntactic consideration of `tf validate` and establishes provider-oriented configuration validations. The canonical example use case is to ensure CSP-specific configurations — such as the stock machine type you're targeting in a compute resource definition — is a valid option. Beyond that, TFLint also comes configured to enforce style and cleanliness industry standards and may be further customized to enforce your organization's internal standards. It also will identify unused configurations and *dead code* that is present in the code base but that has no bearing on the operation or outcome of the TF project. Finally, it will also reveal deprecated configurations, alerting developers of the changes that will become necessary for continued and current support. TFLint comes with its own configuration file for extending or customizing support; this file is written in HCL just like TF. Additionally, when TFLint is run, it will read variable value configuration the same way that a `tf plan` would, for example automatically detecting files in the same directory named *terraform.tfvars* or those referenced using the `--var-file` argument.

By default, TFLint will run only against the root module in a TF project. If you're leveraging nested modules, you may want to extend the support to the nested modules. This can be done for only local modules or for all module paths. It's very likely worthwhile to include all module paths,

regardless of where they're hosted. Moreover, there's little reason not to extend the configuration to local modules — frankly, this should probably be the default configuration. An alternative solution for local module validation is to run TFLint with the `--recursive` flag, which will cause it to run in the root directory and every subdirectory, meaning it will check the local modules, assuming they're in subdirectories from the context of the `tflint` command. In theory, this can mean the various local modules can have distinct configurations, for example, if nested modules are expected to adhere to a different standard than the root module of a TF project. However, this should probably be confined to early stages of a TF project (or even the overall TF ecosystem) to promote flexibility. As the project or ecosystem matures, the standard should become more universal, with few, if any, exceptions and special cases.

> **NOTE**
>
> If you choose to configure TFLint to check all module paths, you'll first need to run `tf init` in the same directory to ensure any external modules are available to validate against.

Most likely, if you're checking the configuration for security vulnerabilities, you would benefit from a predefined set of standards. External organizations, such as National Institute of Standards and Technology (NIST) or Center for Internet Security (CIS), publish security controls and standards. Adhering to these controls can ensure a high degree of security if done completely and across the entire ecosystem. Additionally, you may need to adhere to regulatory compliance in the localities in which you operate, which would require a different set of infrastructure considerations and a distinct set of standards. Many such control sets have corresponding representations in various compliance enforcement tools and frameworks such as the Keeping Infrastructure as Code Secure (KICS) tool by Checkmarx, Aqua Trivy, and Checkov, and because most of the tools in this category operate at a higher level than TF, they will typically be applicable

to more than just TF projects, providing more comprehensive infrastructure validation.

While each of these may be integrated in some fashion into automated workflows, many tools are heavily aligned to more overarching tool suites. This might be a desirable approach in the case where you will be leveraging other tools in the suite. However, it can be overly costly — either in terms of the effort required to establish the underlying framework or in licensing a SaaS edition — if you want only the specific TF evaluation tool. As an alternative, some tools, such as Trivy and Checkov, are singular binaries, which make them effectively plug-and-play. These are low-effort solutions that provide a high degree of coverage out of the box.

> **NOTE**
>
> Open Policy Agent (OPA) is a highly configurable open source tool written in Rego that is intended to act as a general purpose policy framework for IaC and cloud native ecosystems. Although it can be directly configured to evaluate TF, many other tools, such as Trivy, actually use OPA as the underlying rules engine. Moreover, because Rego is also the configuration language used in several other tools such as KICS and Trivy, moving from one tool to another can be very straightforward, and existing teams should be able to pick it up with minimal additional training.

With so many options with slight to significant differences, it can be difficult to choose the right tool. Should you focus on speed to compliance or your long-term strategic goals? If you implement a tool and find that you're not compliant, do you intend to stop everything to fill any policy gaps, or will you implement these over time? Can you fully adhere to a stock NIST or CIS benchmark, or will you require — highly controlled — exceptions to enable core functionality? What if your tool becomes deprecated in the next six months? If the tools can primarily check only individual resources, is that even sufficient to claim full compliance, or do you need other tools operating at a wider purview to ensure universal adherence? If you implement these tools, will you be inviting both a false sense of security and a never-ending quest of refinement? These get increasingly difficult to answer, and the tools are less technical and tactical

and more arbitrary and nuanced than this text is willing to address. However, it's likely beneficial to implement at least one of these tools and unhelpful to implement many more than one. The decision on which to choose is likely more preference-oriented. In short, do your own research and choose whichever tool seems most practical and accessible to the teams charged with incorporating them into your delivery platforms.

> **NOTE**
>
> A tool such as Checkov builds a resource graph similar to the one generated by TF, allowing graph-based checks of relationships among resources similar to how they would relate in the TF state, which is certainly superior to the checks of resources in isolation provided by most other tools. However, Checkov can see only what is available in the directed state graph rooted at the root module; it cannot infer anything about other relationships across your ecosystem that span two or more TF projects. If you're heavily invested in a layered, domain-driven approach to IaC — a common pattern for large organizations — Checkov's advantage may be severely diminished.

One thing well within the purview of this book is how to organize custom controls introduced to the solution. At the highest perspective, there are at least two types of custom control: those intended to fill a gap present in a stock configuration and those designed to relax another control. These can be further segmented into the following list:

Gaps in regulatory compliance

: Although many tools come with support for regulatory controls like GDPR or HIPAA, there may be other governmental or industry-specific policies that are less widely applicable.

Gaps in specific controls

: Similarly, there may be controls similar to NIST or CIS that do not have a predefined control set in your chosen tool.

Gaps in corporate policies

Certainly, your company may choose to go beyond the regulatory or recommended compliance measures and develop policies that are even stricter than those that come with stock configurations.

Relaxation of stock controls to reduce false positives

There are cases where one control may be determined as violated when it's in fact not applicable, or the risk is assumed and documented, and ideally addressed in a different (but no less acceptable) way.

Relaxation of stock controls for unconstrained workloads/use cases

There are other cases where a control must be explicitly relaxed to support a particular use case or workload's responsibility. Whereas the other policies in this list are likely applied across all components and workloads, depending on the granularity, cardinality, and complexity of a fully defined solution, and depending on how your infrastructure code is organized, different domains may be best served by distinct policies.

The resources such policies apply to must be sufficiently sequestered and identifiable to eliminate any possibility that they could impact the compliance of the components within the ecosystem that *are* constrained to the particular control.

This list is likely not exhaustive. Likewise, it's probably not worth breaking every project down to that level of detail. However, there may be certain classes of controls that are more sensitive than others and may lend themselves to a stricter level of access control, suggesting some segmentation would be worthwhile. Considering these designations can identify reasonable domains around which to group controls, providing this high-level segmentation. The order of the controls is also suggestive. The regulatory controls likely should be the first to be fully expressed. Similarly, relaxed controls for workloads that are not constrained by a particular security or privacy consideration may be the most ambiguous initially, and may have a long refinement tail as additional such workloads are identified or established.

Ideally, these custom controls are stored in source control, so they can be consistently referenced wherever they're required and changes are tracked with a high degree of fidelity. Using source control features to verify new policies can ensure potentially disruptive changes are developed in a highly controlled way. A feature branch could be introduced to execute dry-run reports of your existing resources. Moreover, in cases where a different set of controls are used in different contexts, it can be beneficial to use features of Git such as branches, forks, and tags to organize policies in a way that reduces repetition or significant deviation. For example, the baseline policies can be stored on a core repository, with workload-specific controls reflected in a long-running fork that gets updated with the latest commits to the upstream core repo.

Another appropriate topic to consider is how to approach moving from one tool to another. Although, as you've seen, the decision of which tool to choose is largely arbitrary, unfortunately, the decision to make a switch is usually not. It's typically based on some larger business-oriented decision that, in theory, is intended to precipitate a more favorable commercial position and may operate in a less decipherable and deterministic (or even stochastic) context.

Luckily, changing from one tool to another generally doesn't require an extremely clever or novel approach, but it does require significant attention. Assuming you intend to deliver at least the same level of control and coverage of your existing tool, you can start by implementing the new tool against an existing repository in a sandboxed context — preferably leveraging a general purpose wildcard job already available in your automation platform, as described in Chapter 6 — to generate a report, and compare the report to the current tool's latest report and control set. The items in the current tool that aren't available in the new tool's report becomes the primary gap that should be addressed with custom rules in the new tool. For tools that leverage the same configuration language, such as those that use Rego to define policies, it may be a simple operation that is largely copy-pasting existing rules and modifying certain fields and identifiers to match the new tool's argument signature. Even in cases where

a different configuration language is needed, mapping the new controls to the old will reveal the uncovered policies that become the list of tasks needed to achieve capability parity between the old and new tools. Hopefully, any controls that were relaxed in the current tool are well documented and repeatable in the new tool. If you have developed them in a specific location, you can review them against the new tool's report to see if you will need to recreate them in the new tool because the new tool reports a failure for the exact same control.

> **NOTE**
>
> What's especially curious is if the new tool and the old tool have a different conclusion for a particular control. This may require a deep investigation into how this disparity might occur and is addressed on a case-by-case basis. As described before, it could be a matter of how and at what scope each tool operates. For example, because Checkov attempts to make inferences across resource interactions, it may have a different conclusion about whether a particular policy is satisfied or violated. A trickier example is when a control that was relaxed in the current tool to circumvent a policy violation is deemed inviolate in the new tool. This may, again, be due to the way the tool operates, but it can sow significant skepticism of the new tool's reliability.

Behavior Analysis

The preceding tools can be useful in maintaining necessary compliance standards, and could be sufficient to maintain a high degree of security, especially from a passive and structural standpoint — similar to making sure all the windows and doors are secured, safety codes are maintained, and the building itself is structurally sound and undamaged. To extend the analogy, what about the operations that occur within the building? Are the machines operating as expected? This is where TF's `tf test` command and tools such as Terratest can further extend the capabilities of your automation platform.

Whereas the static code analysis tools are intended to validate compliance to various policies, testing tools validate the intended functionality of your TF code. TF's `tf test` command allows you to validate that a particular resource doesn't deviate from an expected format from the perspective of a

planning phase. Moreover, unlike many of the static code analysis tools, which are designed to evaluate against a preconfigured set of controls, TF's `tf test` command won't do anything by default. You define all the tests in HCL with specific blocks using the same logical expressions available in your TF code. Tests can be an alternative to placing the logic in `validation{}` or `precondition{}` blocks if decoupling this logic from your resource definitions — or duplicating it as part of a defense-in-depth posture — is beneficial in some capacity.

By default, `tf test` operates using the `tf apply` command as a subprocess, meaning it will provision real resources. This is useful to validate certain values that aren't knowable until after the corresponding resources are established. This feature will execute an actual apply phase of the codebase in an ephemeral state and will attempt to remove the created components through an internal `tf destroy` command. However, this means any `lifecycle.prevent_destroy` attributes or similar provider controls, such as setting the `deletion_policy` of a `google_sql_database` to `ABANDON`, will cause the tests to fail due to components that cannot be cleaned up as part of the final destroy phase of the `tf test` lifecycle.

> **NOTE**
>
> Although the default is to use the `apply` command, `tf test` can also be configured to run tests that only leverage the planning phase by setting `command = plan`. This means the resources won't actually be created, saving both any costs you might incur as well as a significant amount of execution time. In this configuration, `tf test` acts more similar to the static code analysis tools. However, such tests can validate local values. Because local values are essentially functional in nature — that is, they are pure functions, without any side effects — they can theoretically be unit tested in the conventional software sense. Functionally, this would be equivalent to `validation{}` and related blocks, but it would be evaluated outside of a standard `tf plan` or `tf apply` command.

Additionally, even in cases where the resources created with a `tf test` that uses the `apply` command would be cleaned up completely, the

resources will be subject to resource limits and quotas when they are created. This will cause an otherwise successful test suite to fail if a resource cannot be created due to an exceeded limit or quota.

Terratest also creates running cloud resources that are configured similarly to the ones you have defined in your TF project by default, but it additionally supports more extensive functionality, such as making HTTP requests and executing shell statements. It also provisions the entire module it's targeting, meaning you can validate runtime attributes and behavior more completely and across multiple related resources, measuring what TF *will* do in your environment — not simply what it's supposed to do. Whereas tests used in `tf test` are written in HCL, Terratest tests are written in Go, which is how it has such a much more expressive feature set. This enables test developers to run more general purpose operations as part of its validation, effectively making it open ended in its scope. However, you'll also have to manage the access to the components that operate outside of the TF context, such as any necessary API authentication.

Because Terratest is designed to create live, running cloud resources, it needs to clean up these resources after a run is complete. Unlike `tf test`, which will attempt to run a `tf destroy` as a final operation, Terratest relies on the test developer to explicitly tear down the resources created in the run using the `terraform.Destroy()` function. However, also unlike `tf test` — which doesn't always gracefully exit if it fails, and in particular, may exit before it has a chance to run the `tf destroy` — the recommended approach in Terratest is to use the `defer` statement available in Go, which will ensure the `tf destroy` operation is run, even if the `tf apply` command fails.

> **NOTE**
>
> Terratest can also have trouble with certain attributes. Like in the case of `tf test`, this is because Terratest also executes an actual apply phase of the codebase in an ephemeral state and recommends you attempt to remove the created components through its `terraform.Destroy()` function — which is still a wrapper around `tf destroy`. Once again, this means any `lifecycle.prevent_destroy` attributes or similar provider controls will cause the operation to fail. It also suffers from the same restrictions as the `tf test` command regarding CSP limits and quotes.

Although, by default, Terratest tests are expected to run against a specific module or collection of modules because they're implemented in Go, in theory — and not uncommonly, in practice — they can be configured to read information from any existing resources, regardless of how they're managed. This is arguably where a tool like Terratest really shines: rather than including it as a regular part of your infrastructure delivery procedure — and a rather heavy and potentially expensive one, at that — you can instead use it to build a suite of integration tests that don't actually provision any ephemeral resources, but that simply reads the existing resources across your ecosystem and compares them to expected configuration or behavior, as shown in Example 10-6. These can even be further expanded and made to interact with live resources, such as making real HTTP requests against a web-facing service.

Example 10-6. Terratest to validate network properties regardless of how they are managed

```
// imports not shown; 2 space indentions used for brevity
func TestVPCConstraints(t *testing.T) {
  projectID := "proj-tas-10-components"
  netName := "vpc-tas-10"

  requiredRegions := map[string]bool{
    "us-west1":    false,
    "us-central1": false,
    "us-east4":    false,
  }

  svc, _ := compute.NewService(context.Background())
```

```
  // --- Subnets ---
  subnets, _ := svc.Subnetworks.List(projectID, "regions/-").Do()

  count := 0
  for _, s := range subnets.Items {
    if strings.Contains(s.Network, netName) {
      count++
      parts := strings.Split(s.Network, "/")
      region := parts[len(parts)-3]
      if _, ok := requiredRegions[region]; ok {
        requiredRegions[region] = true
      }
    }
  }

  assert.LessOrEqual(t, count, 10)

  for _, found := range requiredRegions {
    assert.True(t, found)
  }

  // --- Firewall rules ---
  firewalls, _ := svc.Firewalls.List(projectID).Do()

  for _, fw := range firewalls.Items {
    if strings.Contains(fw.Network, netName) && fw.Direction ==
"INGRESS" {
      for _, r := range fw.SourceRanges {
        assert.NotEqual(t, "0.0.0.0/0", r)
      }
    }
  }
}
```

Both the apply-mode tests in `tf test` and Terratest might be best served for validating nonroot modules, such as local modules or those that are intended to be consumed by third parties. In such cases, Terratest can also be focused to specific modules by using the `TerraformDir` field configured in the `terraform.Options` struct, which controls what directory the tests apply to. In some cases, especially for specific local modules or those that are intended to be consumed by third parties, a common successful practice for Terratest is to introduce the actual modules you're testing against in the examples subdirectory as introduced by the standard module, and to assign this directory to `TerraformDir`. This

way, you can develop canonical use cases for the modules, and these can be fully isolated from your business-critical software ecosystem.

Finally, because both `tf test` and Terratest establish real resources, they can take a nontrivial amount of time to run, which may significantly increase the time it takes to deploy changes. Also, it's worth reiterating that because these tools establish real cloud resources by default, you'll likely be billed for these resources, even if it's only for a small fraction of your overall cost.

Low-Tech, Low-Cost Solutions

Speaking of reducing costs, there are several no- or low-cost solutions that can be valuable to a highly effective development approach. Some of them have a development cost that quickly pays off, while others are incorporated directly into the development process as part of the team's operational discipline.

One of the most effective validation solutions is the practice of conducting live code reviews, whereby the entire team reviews the same code changes and is able to provide feedback or ask clarifying questions. Unfortunately, this is often overlooked entirely or downgraded to a formality, amounting to a costly waste of time rather than a source of resilience and maintainability. Often, code reviews are left to the end of a task as a last step before updates are merged into a shared repository. This makes sense intuitively because it gives the code the best chance to be acceptable. However, if the code appears "too good," it may not elicit meaningful scrutiny, and any actual problems or deviations from the standard can become obscured. It may be more effective to focus on consistency and discipline with the practice and require a regular review of whatever code is in flight, no matter the current state. This is much more likely to require critical attention just to make sense of the changes, let alone determine if they're valid. It may also have the side effect of keeping people on task, without having to establish a strict culture; although it may seem scary at first, the typical result is a more cohesive team not only in terms of code consistency and scope, but also in camaraderie, as folks become more comfortable with being open with each

other and neighborly in their interactions. After all, seeing the unfinished product has more of a chance of showing your thought process in how you approached the problem than the finished product ever would.

> **NOTE**
>
> Live code reviews could even be incorporated into the stand-up ritual. If your team is pushing code at least once a day — which should be a mandatory policy in any team, and is frankly a fairly lenient one — this should be as simple as pulling up the code differences in your Git provider.

It can be helpful to include the `tf validate` command, which ensures your code is syntactically sound and self-consistent, before the planning phase of your workflow to have a fast-fail check before even reading your TF state. Unfortunately, the `tf validate` command does not run any of the variable `validation{}` blocks. Nevertheless, incorporating `validation{}` and `precondition{}` blocks for important policies and standards is a low-cost way to enforce them. Finally, ensuring all nonsensitive information that would be important to operators or downstream operations is reflected in the TF outputs should be a requirement for all resources. Outputs should have a standard naming convention to ensure the information is understood by anyone reading them.

A solution that likely can be built in a day — or perhaps up to a sprint, depending on its sophistication and complexity — is a Git pre-commit hook that will run when you execute the `git commit` command. This would be a shell script that is intended to execute linting or other static code analysis on the codebase. In theory, this could be configured to run arbitrary commands, such as any of the TF commands like `tf validate` or even a `tf plan` — if everyone has the proper read access — to review how their changes might affect the TF state, or other validations, such as KICS, that you have incorporated into your workflows.

> **TIP**
>
> The downside to the pre-commit hook is that if it is too severe, it leads to fewer commits. This can be bypassed with the `--no-verify` flag, but a better solution might be to prepend an environment variable that can skip or override certain portions of your pre-commit script without forgoing all the benefits that the script provides. Even simply ensuring a failed validation doesn't prevent the commit from proceeding can be a great way to gauge how much work is left to do after reaching a point worth committing to the Git history.

Policy Enforcement with Modules

As you saw in Chapter 5 on modules in TF, one potential use case is to develop a *logic module* that isn't intended to provision or manage any resources, but is instead designed to better codify the complexity that might otherwise be placed in the `locals{}` block. This can improve the experience for developers that consume the module because it creates an SoC, thereby removing their responsibility to maintain the integrity of the logic — not to mention eliminating the risk of unintentional and undesirable modification — while allowing them to still use the `locals{}` block for items that are more arbitrary or less strictly controlled.

In the case of enforcing certain naming or tagging conventions computed from input variables, this is indeed an improvement over placing these expressions in the `locals{}` block because it isolates this logic from the resource definitions. It can even be constructed in such a way to develop purely functional unit tests of the logic by validating the module outputs based on a curated collection of input values. This makes including tools such as `tf test` or Terratest, using the `plan` command rather than the `apply` command, as part of your normal TF software delivery much more viable because it would focus only on the logical expressions you develop, rather than (at least partially) validating the underlying framework, Terraform or OpenTofu, or the specific CSP's provider.

Also as presented in Chapter 5, this technique works best with logic that is intended to construct or validate the inputs to a particular module: an *input-*

only logic module. This is because all of the arguments supplied to this logic module — and, perhaps more importantly, all the outputs — must be available at the same time. Although it's possible to construct a logic module that relies on computed attributes from resources, the same logic module may not be read from a resource that it reads attributes from. This is further complicated by interdependencies among resources, which may not be immediately obvious or explicit, but which would cause a circular dependency in the TF resource map. Example 10-7 depicts such invalid configurations. The more interconnected your resources, the more such logic modules you may need to govern these intermediate logical expressions, which may not be appropriate. Given this commentary, *output-only logic modules* — whereby a dedicated module computes the values used, for example, in the `precondition{}` subblocks of `output{}` blocks — are also technically feasible and may indeed be worth constructing for highly complex `condition` statements.

Example 10-7. Logic modules that create a circular dependency

```
module "local" {
  source = "./modules/local"

  read_from_bucket = google_storage_bucket.example.time_created
  read_from_subnet = google_compute_subnetwork.example
}

resource "google_storage_bucket" "example" {
  name          = "example-bucket"
  location      = "US"
  force_destroy = module.local.pass_to_resource
}

resource "google_compute_subnetwork" "example" {
  name          = "example-subnetwork"
  ip_cidr_range = "10.0.0.0/16"
  region        = "us-central1"
  network       = google_compute_network.example.id
}

resource "google_compute_network" "example" {
  name                    = "example-network"
  auto_create_subnetworks = module.local.pass_to_resource
}
```

> **TIP**
>
> Rather than constructing logic modules that rely on computed resource attributes and are intended to validate the input arguments to other resources — such intermediary logic modules — it could suggest a domain boundary that you should consider splitting into dedicated nested modules. That way the logic module becomes input (or output) only, ensuring all the arguments and outputs of the logic module will be available at the same time.

Although using a module in such a way would be a violation of the recommended standard module approach, whereby module nesting is minimized in favor of flat dependency graphs, and while mixing `module{}` and `resource{}` blocks violates the general recommendation presented in this text, if it is used consistently as an optional add-on that is purely meant to validate inputs or other values, and it strictly does not maintain any resources, it can certainly be a valuable technique to strengthen resource attribute integrity. Examples 10-8 and 10-9 show this.

Example 10-8. Using variable validation before constructing safe output values

```
#------------------------------------------------------------------
-----
# root module
#------------------------------------------------------------------
-----
module "local" {
  source = "./modules/local"

  app_code      = var.app_code
  business_unit = var.business_unit
  region        = var.region
}

resource "google_storage_bucket" "example" {
  name          = module.local.safe_name
  location      = "US"
  force_destroy = true
}

#------------------------------------------------------------------
```

```
-----
# ./modules/local module
#------------------------------------------------------------------
-----
variable "app_code" {
  type        = string
  description = "Codename for the application."

  validation {
    condition     = contains(["app1", "app2", "app3"],
var.app_code)
    error_message = "Must be one of app1, app2, app3"
  }
}

variable "business_unit" {
  type        = string
  description = "Codename of the business unit."

  validation {
    condition     = contains(["bu1", "bu2"], var.business_unit)
    error_message = "Must be one of bu1 or bu2"
  }
}

variable "region" {
  type        = string
  description = "Region things run in."

  validation {
    condition     = contains(["us-west1", "us-east1", "us-
central1"], var.region)
    error_message = "Must be one of us-west1, us-east1, us-
central1"
  }
}

output "safe_name" {
  description = "Safe name constructed from the inputs."
  value       =
"${var.app_code}-${var.business_unit}-${var.region}"
}
```

Example 10-9. Using output preconditions to validate after constructing safe output values

```
output "safe_name" {
  description = "Safe name constructed from the inputs."
```

```
  value       = local.safe_name

  precondition {
    condition     = local.valid_inputs
    error_message = join(" ", [
      "The computed resource name must be like",
      "{app_code}-{business_unit}-{region}"
    ])
  }
}

locals {
  valid_app    = contains(["app1", "app2", "app3"], var.app_code)
  valid_bu     = contains(["bu1", "bu2"], var.business_unit)
  valid_region = contains(["us-west1", "us-east1", "us-central1"],
var.region)
  valid_inputs = local.valid_app && local.valid_bu &&
local.valid_region
  safe_name    =
"${var.app_code}-${var.business_unit}-${var.region}"
}
```

Observability Considerations

One of the best ways to validate your TF configuration is to develop fine-grained observability components such as monitoring dashboards of provisioned resources and strategic compliance violation alerts, along with typical system access log search and query capabilities. These controls operate continually, in addition to the value they provide on demand, making them invaluable to ad hoc assessments and reports, fitting nicely with the more specific point-in-time nature of the tools embedded within your delivery workflows that are focused on change events. Besides their obvious value to auditability and troubleshooting, these tools can enhance compliance and robustness of your ecosystem by surfacing resource failures or infrastructure drift — whereby a resource is modified in some way that A) isn't tracked by TF and may be reverted in a subsequent `tf apply` operation, and, more importantly, B) may result in the resource falling out of compliance or becoming exposed. These controls generally manifest as monitoring dashboards, which display some graphic such as a chart or other

reporting visual, and alerts, typically a message sent to one or more channels.

> **NOTE**
>
> This commentary is mostly relevant to OpenTofu, as both TFE and HCP Terraform have extensive — and extendable — observability capabilities built-in.

Monitoring and alerting is inherently event-oriented, meaning they work in response to some event. Traditionally, software monitoring and alerts are based on logs, whereby a system is watching entries to a logging repository and checking against a specified set of rules. If the rule gets matched, a dashboard or other visual UI is updated in the case of monitoring, or an action is taken in the case of alerts — usually a message on one or more communication mediums sent to an operations engineer or to a collective channel followed by a group of such engineers. The most basic observability of your cloud resources is to monitor your system access logs and report on unexpected or dangerous actions. This requires the initiator of all such actions to be well identified, meaning your automation platform and any other operator (human or otherwise) must have dedicated identification credentials — really, this should already be in place — with strict access controls, to ensure only the systems or entities may perform the action in the first place.

Although system access log monitoring is instrumental in identifying when and who/what initiated an action, it doesn't capture the intent behind the action, and it can't always explicitly convey the impact of the action. Introducing additional observability controls — some requiring more custom configuration, and even additional systems, than others — can provide different perspectives and details that help to fill in these gaps. Validation monitoring of your TF ecosystem, and how well it reflects your wider CSP totality, is one such perspective that can track how various actions relate. For example, an update to a critical or sensitive component may be instituted by your TF automation platform to correct configuration

that has drifted from the specification. Without this information, the action may be flagged as a violation when it's really correcting a misconfigured component. Unfortunately, it's highly likely that there is little overlap among the group of individuals who have access to review system access logs and those who are responsible for actions that precipitate them. A very noisy system may cause the reviewers to overlook issues as they begin to lose meaningful signals among the copious noise, turning a would-be solution into a waste of effort and resources.

To provide the personnel charged with monitoring your cloud ecosystem with additional information in the case of seemingly anomalous behavior, you can incorporate your TF validations, thereby achieving TF observability. One way to implement TF observability is to establish a regularly scheduled validation process and simply logging the results in a well-structured format as part of this process. This leverages the existing observability platform, enhancing visibility by cross-referencing the access logs with the infrastructure changes or updates — assuming data correlation is supported in the platform. This can be the quickest path to implement a solution, but it's limited in functionality.

Alerts, by the most conventional definitions, are communications meant to inform operators that something is wrong. However, given the distributed nature of cloud-based applications, nearly every workload relies on communication. This suggests alerts can be constructed to communicate instructions to such workloads, such as through a webhook, remote procedure, or messaging queue. Surely you don't want a fully autonomous system making changes to your cloud infrastructure, especially on imperfect information. That being said, it may be highly beneficial for alerts to trigger additional checks and validations, essentially automating the initial audit of whatever action generated the alert. A report can be generated and this can be sent along to the same alerting channels.

For easy, low-risk, high-reward corrections, such as targeted `tf apply` commands for certain resources, like firewall rules and IAM permissions, these could be triggered remotely through a secure API or webhook. This can be introduced as an additional, but hopefully underused, protective

layer as part of a self-service or even fully automated TF system. The rules that govern this system would need to be extremely well crafted and unambiguous. A policy tool such as OPA may be used to construct these rules.

> **NOTE**
>
> For every rule, there is an exception. For automated remediation systems, there may be certain highly special components, or classes of components, in your wider ecosystem that have to be configured a certain way to work properly. These can be tagged somehow to indicate to the system that the component should not be automatically modified. However, it's probably worthwhile to generate a (highly specific) alert or report so that the event is still at least captured and relevant parties are aware (who, in theory, should already be aware). As frequently suggested throughout this text, such exceptions should eventually be turned into additional rules.

Strategies

Generally speaking, the strategy for observability is to start with the system access logs, as discussed, as a foundational source of information, and layer on additional contexts, refine reports, and filter out the signal from the noise. Although system access log analysis is readily available — if not the default — in all of the top CSPs, stock configurations will not be enough for all organizations, so even setting up this initial foundational layer is nontrivial and requires significant planning and architectural design. It's likely to be something that you want to start with on day 0, but it's also usually something that cannot be completed — or considered sufficient — until after a significant portion of your workloads are in place. This goes beyond filters and reports and also includes the locations of log collectors, rules around persistence, and how logs are sampled or aggregated (or not).

Next, you'll want to incorporate your TF validation into your observability capabilities. As presented before, this can leverage the existing logging, monitoring, and alerting framework used for system access logs, although they should be stored in a location distinct and isolated from the system access logs and should be made accessible through dedicated access

controls, distinct from those allowing access to the system access logs. Alternatively, a completely different solution, such as using a database or data warehouse, may be constructed to capture and record the outcomes of the TF validations, which will further distinguish these reports from the system access logs. Perhaps obviously, in this case, it's critical to build the TF validation reports such that they may be matched — or otherwise compared — against the system access logs, ideally using similar, if not the same, field names. Once the data sources are in place and comparable in some way, additional dashboards and alerts can be constructed that compare and correlate the data generated from each system. These can be layered on top of existing mechanisms or constructed so as to replace existing dashboards and alerts with ones with greater detail or a more focused message.

> **NOTE**
>
> Although this is focusing on cloud infrastructure, in reality, there may be many more observability components and domains that have high importance. It's up to you and your organization to determine where these other considerations fall within the timeline and roadmap of your observability capabilities. In other words, the strategies presented here are strictly along one dimension in the multifaceted world of software.

Of course, when you add additional sources of data, you naturally invite the possibility of additional noise. Over time, especially as your alerts become more focused and actionable, with clear patterns of how to address or further investigate anomalies, it may be beneficial to reduce how often the TF validation checks are run and to use the alerting mechanism from the system logs to trigger them, as needed. This can significantly reduce noise without compromising on clarity or depth of investigation. Depending on how extensive the TF validation steps are, it can also be a significant savings of resource utilization.

The best of both worlds is to do both: run TF validations regularly and on demand. This requires separating these validations into two levels: one that is checked regularly, as before, and another that is intended to run on

demand. By creating this distinction, the validations themselves can be shaped to accommodate each type. Validations meant to run regularly can be superficial and focused on interpreting attributes read from individual resources or TF state addresses, while the on-demand versions may go much deeper, reading from related resources, or even probing the resources to better determine their state. There may even be situations where you want to create a live running resource to validate against, but for which you want a short, finite lifespan. Examples of this might be in the case of a `tf test` or Terratest validation, to determine if an issue under investigation is a result of the environment in which the resource is hosted or if it's somehow specific to the circumstances of your account, such as a problem with a billing account, or even if something inherent to the CSP such as a change in a resource's behavior or a systemic (and hopefully temporary) issue in the CSP itself like a failure of core components. Rather than activating several on-call personnel to investigate the occurrence with little upfront data, these deeper reports act as the first layer of mitigation, potentially saving hours of investigative work, collectively, in a situation where time to resolution is the primary KPI.

As you can see, each of these strategies builds on the former, giving a clear direction of how they may be introduced. You can also extend this approach, such as by stitching in autonomous agents, by further layering controls on top of these strategies.

Recommended Dashboards and Alerts

There are a few typical dashboards and alerts that should be considered as baseline.

> **NOTE**
>
> Again, these dashboards are largely included by default in TFE and HCP Terraform.

The basic dashboards that you should consider establishing are:

Resource drift

: Captures what infrastructure has diverged from the last applied state. It can indicate instances of unmanaged or manually changed infrastructure.

Pipeline metrics

: Helps determine how efficiently the `tf run` pipeline is operating. This can highlight where bottlenecks are forming.

Queue depth

: Displays how long runs are waiting before they're able to begin. This can suggest the system is underprovisioned or overloaded.

Run duration

: Records the run duration of a pipeline. Should be configured to highlight where time is being spent (such as in the planning or apply phase).

Workspace activity heatmap

: Reveals where infrastructure changes are concentrated and which workspaces are most active.

Healthchecks

: Signals whether the components that the platform establishes are actually working as expected.

Versions

: Software versions of the TF CLI, the included providers, and any remote modules leveraged for a project.

The following alerts are a good start, but must be tuned and shaped to ensure they are heeded and actionable:

Drift of certain layers or resources

Triggered when specific resources drift from their expected configurations. These should be directed toward the infrastructure platform team responsible for the relevant resources.

Run queue backlog too high

Triggered when the number of jobs waiting to start goes above a predefined threshold. This should be sent to the team responsible for the TF automation platform.

Run duration SLA breach

Triggered when an IaC job's run takes too long. Might warrant a warm-up threshold or cool-off period to avoid being too noisy. This should also be sent to the TF automation platform team.

Resource saturation

Triggered when a workflow runner's resources exceed a certain threshold. This again is sent to the TF automation platform team.

Health check failure

Triggered when certain health checks fail. This also may require tuning to reduce noise due to false positives or temporary disconnects. These should be directed toward the infrastructure platform team responsible for the relevant resources.

Other Touch Points

Although this chapter is specifically about validation, strong observability controls can be extended to, or embedded within, other systems as a core source of information. As suggested, this can be included in an autonomous system that corrects configuration or takes other remediation measures to ensure the integrity of the workloads it supports and ongoing compliance to the various policies to which they are beholden. Even without advanced

language model capabilities, there are likely to be certain actions that are taken to address various well-worn solution paths to common concerns, meaning these can be hardcoded as a standard procedure in an automated workflow triggered whenever the concern arises.

For other, more ambiguous or less familiar activity, an autonomous agent may be included within the investigation chain to further investigate aspects of the activity and to suggest subsequent mitigation steps. These suggested instructions may be presented to the personnel charged with determining and reporting root cause or even to those responsible for possibly correcting the action, if necessary. In a sufficiently advanced configuration, such entities may also be codified as autonomous agents, reducing the need for on-call engineers or round-the-clock vigilance by human workers.

> **NOTE**
>
> Some might read the previous commentary as advocating for a reduction in workforce. While it indeed suggests a change to how various operations are executed and by whom, it's likely better to enhance the existing personnel with such autonomous agents. Even over time, it's probably beneficial to realign the responsibilities of individuals who at first are benefiting from the autonomous agents to do their job more effectively and efficiently to instead act as support personnel for these same autonomous agents, improving their capabilities and reducing mistakes, inefficiencies, or improper access controls.

Summary

In this chapter, you reviewed several options for validating your TF ecosystem, including:

- The various classes of validation
- Validation blocks in variable definitions, as well as precondition and postcondition blocks
- Data sources to ensure values that reflect resource attributes are valid and available

- Static code analysis tools and how they can be included in automated TF workflows
- Tests, such as with `tf test` and Terratest, to validate how running resources behave
- Low-tech solutions such as live code reviews to improve team discipline and to spread the knowledge of a particular feature or fix across the team
- Using a nested module to establish and validate certain input values
- Observability tools and techniques that can fill in gaps between explicit validations

You're now prepared to improve the code quality of the TF you build and the underlying code you write. You realize TF testing is very different from application testing and has a different cost that may lend itself to a separate lifecycle and schedule from the various TF projects they support. You recognize that sometimes the most effective validations are the ones built into TF, such as the variable validation expressions. Finally, you understand how observability controls complement any validations included in the TF automation workflows.

In Chapter 11, you'll see how all of the concepts presented in this book come together in successful patterns that together establish a world class ecosystem in which to develop your application workloads. You'll review a number of unfortunate situations and strategies for how to extricate yourself from them. Finally, you'll walk through a number of realistic scenarios you may encounter at some point in the life of a TF project or your organization's overall ecosystem.

Chapter 11. Forever Forward

Although the primary purpose of this book is to prevent the scenarios found here, by now it should be clear to you that there is no truly foolproof solution. What may be less clear, but is reflected throughout the text, is that there are so many corner cases to make the chance of painting yourself into one statistically significant. Moreover, another goal of this text is to meet the reader where they are, which is likely to be in the context of an existing ecosystem that may not have been built with the guidance presented here in mind. Thus a comprehensive manual for TF at scale should include something of a field guide to getting out of predicaments.

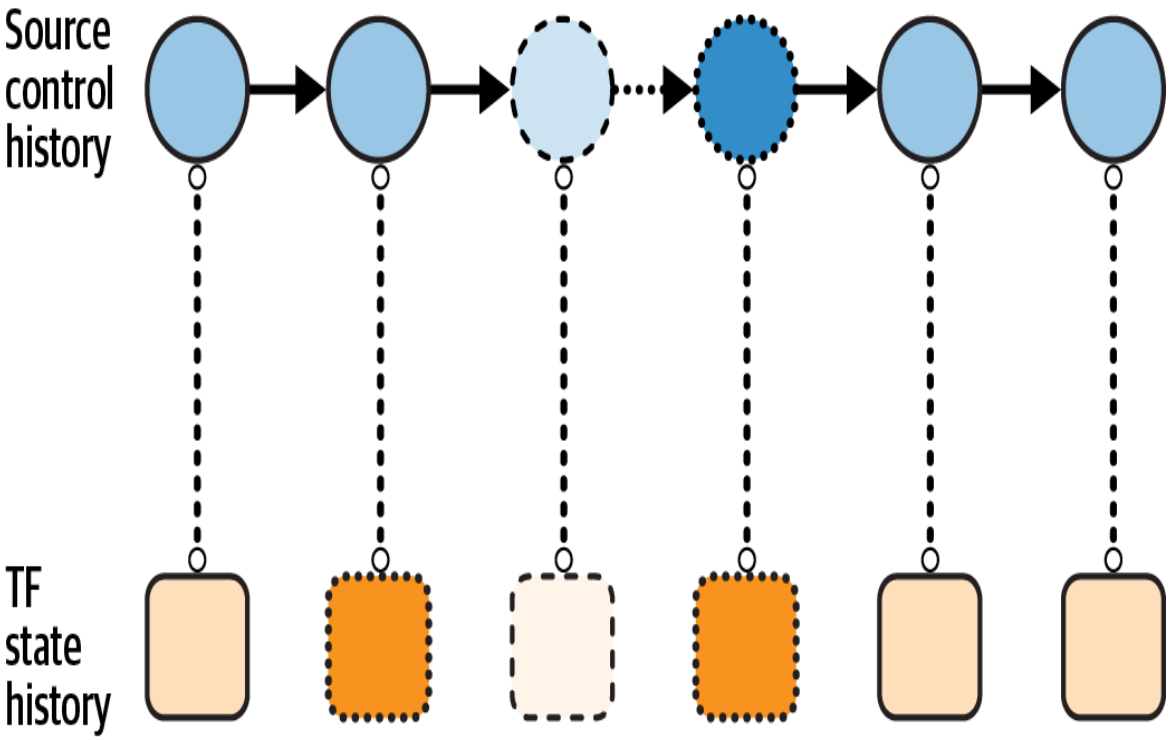
An alternative title for this chapter could be "Falling Forward," as successful application of many of the procedures presented here don't always feel like success. This chapter is intended to provide guidance for when things get messy, confusing, or overwhelming. It's likely that some of the techniques presented here are not available in every organization, either due to explicit restrictions or missing guardrails that would otherwise enable them. It's also likely that better solutions exist in some, if not all, cases. In the parlance of spiritual circles, life coaching practices, and TikTok influencer advice, *take what resonates, and leave what doesn't*.

Things to Remember

Throughout this book, you've encountered multiple themes expressed as caveats. For the most part, they're framed in terms of common concepts and industry practices borrowed from other software practices, but they have different conclusions in the context of TF or IaC in general. In some cases, they are obvious conclusions within the context of the cloud, whereas others may seem totally paradoxical in nature. The following is a summary of these themes and some commentary on how these should fit into the mental model.

Moving Forever Forward

The concept of *forever forward*, as it's used here, indicates a forward progression of software and state changes, centering around the evolution of the TF state. TF does not support a rollback or revert command, meaning any changes that get applied do so on top of all the previous changes. Although some systems support, or can be configured to support, a concept of TF state history by versioning the TF state backend, it's much easier to comprehend the changes made to a TF project by reviewing the source control history. Git *does* support a way to undo changes, with `git revert`, but this does not remove the errant commit, rather making a brand-new *inverse commit*, preserving the history. Thus, using such a command to undo a change that was applied in TF preserves this relationship, as depicted in Figure 11-1.


Source control history
TF state history

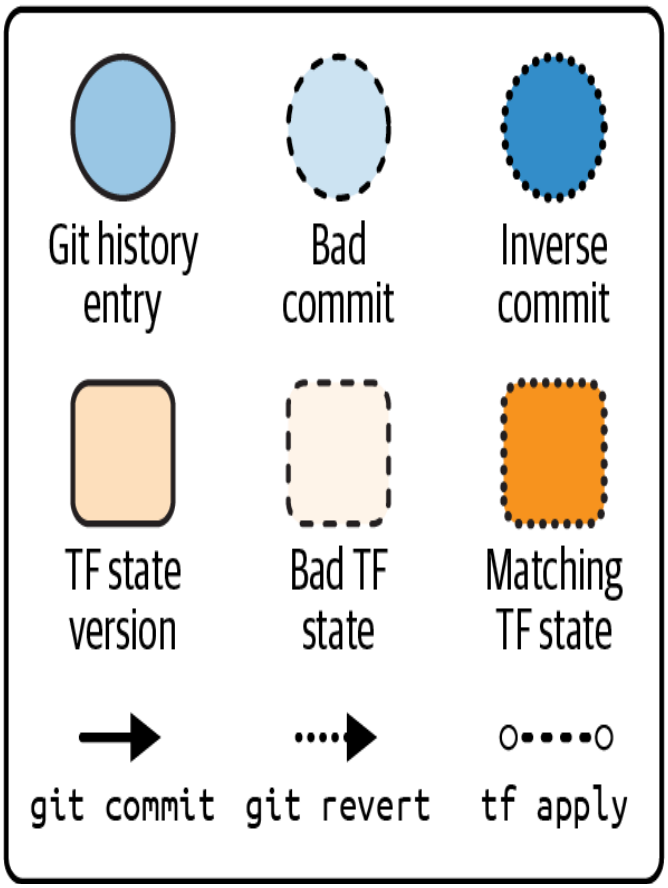

Git history entry
Bad commit
Inverse commit
TF state version
Bad TF state
Matching TF state
git commit
git revert
tf apply

Figure 11-1. Git and TF history moving together, even through revert

If you're working off of a shared branch, for example, as part of a typical Git workflow, you're likely making *merge requests* (also commonly referred to as *pull requests*) from a feature branch of a common main branch. In most popular Git tools, entire branches can be reverted as well, also producing *inverse commits*, as needed, to undo the changes.

No Pets

Before the cloud, generally speaking, software had one of two options: it was distributed to end users — for example, through physical mediums such as compact disks or floppy drives — or it was hosted by machines physically managed by members of your organization. In the case of such self-hosted software, if something went wrong, you might have had a multistep troubleshooting process similar to the following:

1. First, see if the process can be restarted without affecting the wider underlying context of the operating system.
2. Next, stop the process, remove the binary, install a clean copy of the binary, and start it up again.
3. Try running the process on a different machine.
4. Debug binary code.
5. If that doesn't work, consider reinstalling the operating system.
6. If none of these work, consider replacing the underlying machine.
7. Throughout the process, comb through a multitude of system and process logs.

Software is useful only if it's usable. This means it's available and accessible, performs a calculation or a task, and (traditionally) does so deterministically. In this context, if a product became unavailable, it was up to you to figure out how to make it available again. And because you only had the machines available to you based on your most recent capital

expenditure, you were very likely constrained to resolving it on the same device. These machines may have a significant amount of additional customized processes necessary for the system to work properly, and (at least, back then) these may require installation on top of the operating system, which itself would often need to be freshly installed on a machine — which even when virtualized is not a quick process. In other words, resetting a machine to pristine conditions was a tedious and time-consuming effort. In such a case, the primary KPI to your availability SLA was the time it takes to remediate the defect.

If this was you, these machines were your pets. You cared for them. You cleaned up after them. They had their nuances, quirks, and nicknames. They almost had a personality. You made sure they were healthy and happy. And you felt real pain when they experienced potentially fatal unknown problems.

With the cloud, we have an additional hosting location: namely, the cloud itself. Because the cloud is really a widely distributed network of datacenters, each containing thousands of physical machines with various hardware and software configurations, it opens many more options in terms of architecting for the distribution of your software and signals a shift in perspective when it comes to distribution. In particular, it basically democratized the SaaS business model by enabling a pay-what-you-use subscription approach to hardware expenses. Because anyone with a credit card could now rent (nearly) whatever machine they wanted in whatever supported location they wanted, workloads and content could be duplicated however desired in different locations — this might be done to reduce network latency, handle heavy network traffic load, or support recovery in the case of a disaster that renders one or more datacenters unusable.

> **NOTE**
>
> You don't operate the machines that you're renting in a cloud context. The physical machines are located in the datacenter controlled by your CSPs, which, as already described, may be located far away from you and locked within highly secure corporate facilities. In other words, you cannot simply "turn it off and turn it back on" (or "unplug it and plug it back in"). If a machine goes rogue and cannot be accessed through a remote interface like SSH or remote procedure call (RPC), it may be lost forever. Beyond the technical implications, this has a psychological effect on how these machines are viewed.

This commoditizes computer resources to the point of making them interchangeable. This is the core of the pets versus livestock analogy: these resources are no longer highly curated and cared for. Instead, they can be indiscriminately used and discarded.

In today's context, availability almost seems synonymous with fault tolerance: both are achieved with a certain amount of redundancy and both can be thought of as providing your software consumers with the lowest-latency connection — interface, network protocol, or what have you.

This makes the primary KPI in the case of a problematic service something more like time-to-replace. The quicker the service becomes available again, the quicker the application regains the sought-after level of service.

> **NOTE**
>
> Software-defined network controls enable tight control over traffic routing and in many cases can be applied instantaneously. Certain DNS-related configuration may take time to propagate through the public internet; for a more immediate response, require additional actions such as forcing a flush of a DNS cache.

Everything Is Production

From one perspective, the cloud is a combination of infrastructure as a service (IaaS) and platform as a service (PaaS), which are both software-

driven systems owned by a CSP, making everything you do a production workflow.

Even in the case of sandbox environments, which are usually designed to run experimental, unfamiliar, or, really, arbitrary workloads, you have to remember the resources are live and cost real money, and because the machines are somewhere else and administered remotely, there's a real risk that they're infiltrated by a nefarious operative. Certainly, they need to be flexible, perhaps even more so than your lowest level development environments, such as by widely enabling cloud services and relaxing certain access controls. At the same time, the sandbox must be safe enough to operate these services without it being a security or financial risk, possibly to the point of being more locked down than your actual production environments. This might mean no public access, with only private IPs and the only network connections being direct links back to your on-premises networks.

> **TIP**
>
> GCP supports a mechanism for safely sending certain network traffic to inside a VPC network using a secure proxy and access controls, its IAP service. This can allow for fully remote, fully private access without the need for a physical link from your internal network to the CSP.

Also, in the case of sandboxes, you can introduce additional mechanisms, such as one that resets the environment's resources on a weekly basis so things don't stick around too long or become too messy or stale. Moreover, protecting an environment like a sandbox or production also means developing whatever available security controls that can create a perimeter around it, to prevent, say, the included experimental workload from breaking out of the environment and interacting with other components in other environments. Unless specifically justified, segmenting all environments in such a capacity is always a good practice to consider.

Recap of Common Problems

Throughout the book, you've encountered several common problems. Every team will likely experience at least one of these issues over a long enough timeline. They're collected and summarized here for review and convenience:

Using `count` *for a list of resources*

> If you use `count` instead of `for_each` for a list of resources and you need to deprovision one of the resources (other than the last one), you cannot do so without removing any of the other instances. As presented in Chapter 4, this is because of the whole number indices of the resulting set of resources. If you choose to replace the `count` with a `for_each`, you can rewrite the TF state to move the `count`-based addresses with the `for_each`-based addresses.

Interdependent projects

> If two (or more) TF projects have interdependencies — such that one project manages resources that rely on attributes from resources provisioned by another TF project, and this other project has resources that rely on resources from the first TF project — you may have to run the first, then the second, then the first again to make sure certain updates propagate throughout. In this case, the fix is to reorganize the projects, separating out the related domains and ensuring the dependencies go in only a single direction. This will again require updates to the TF state.

Disconnected resources leave a halfway-applied state

> When one resource depends on another resource but it doesn't relate directly, the dependent resource may not be created successfully the first time, despite otherwise correct configuration, and running `tf apply` a second time will succeed. Ideally, you can use an attribute of the depended-upon module to make the relationship specific. Otherwise, the `depends_on` meta-argument can be utilized.

Halfway-applied states

Sometimes a `tf plan` will consistently produce a clean report, but a `tf apply` fails midway each time. The long play is to establish strong validation expressions, comprehensive data sources, and maybe even `tf test` or Terratest. However, the short play is to temporarily cut the errant configuration, push the working changes through, add back isolated errant configuration, and implement a fix.

Never-clean states

Some resource fields take multiple constructions of a value, with varying degrees of specificity. Generally, less specific fields rely on other information. Assuming this information is indeed available, the resource can be created. However, subsequent planning phases will likely report that the field will be updated (in place) to the less specific construction, although in reality it will remain the same. The (perhaps obvious) fix is to use the most specific field construction available. If it's not possible to use this construction, you may be able to leverage the `ignore_changes` field in the `lifecycle{}` subblock.

A Better Way

Developing TF at scale is full of nuance, quandary, and constant transformation. Just as there are few practices that are universally optimal, there are many options that are equally mediocre or equally not that great. In this section, you'll review several such options and alternative solutions that will generally prove more viable.

The most popular example is the `count` vs `for_each` meta-arguments. The popular approach is to use `count` only for introducing conditional resources — using a ternary to map a Boolean expression into the numbers `0` or `1` — and `for_each` for arbitrary collections of resources. However, in reality, `for_each` can also be used for conditional resources: by supplying `for_each` an empty collection, you achieve the same effect as

setting `count` to `0`. If, for example, the support for multiple such optional resources is an unfathomable prospect — perhaps it must be unique within a particular context, such as a singular App Engine application per Project in GCP, or perhaps it's intentionally singular, such as some data source of truth — `count` is likely to be forever adequate. However, if a feature's singular nature is arbitrary or only based on current requirements — such as a cloud VPN instance that is both optional and replicable in same combination of Project, VPC, and Region, to scale out bandwidth — constructing this as a `for_each` expression would be more functional, long term. There's certainly the case to be made against overengineering in an attempt to future-proof your solution, but it's a fairly small price to pay to avoid extensive surgery on your TF state(s).

A less examined debate is how to construct nested modules that will incorporate a `for_each`. If the module is intended to bundle a related stack, and the purpose is to replicate the stack in total in a cookie-cutter fashion, it's best to place the `for_each` in the `module{}` block, instead of, for example, in the module's `resource{}` blocks (as shown in Example 5-6 back in Chapter 5). Inversely, if the module is intended to bundle a specific set of resources with a variable number of a specific resource type (or a group thereof), the `for_each` should go with the `resource{}` block definition. This may be obvious if you operate in perfect knowledge, but unfortunately, modules aren't always created in a single iteration, a decision made early may be suboptimal later, and inverting the relationship can be tedious and risky.

Over time, the use of `count` to support conditional resources may grow to the point of reaching a critical mass. If several such resources use the same ternary expression formula, for example, especially those that are closely related, packaging them together in a nested module will better reflect this correlation. If this new module is purely downstream from everything else in the TF project, it may be more effective to extract these optional configurations into a dedicated TF project and separate root module. This way, the configuration that is applied only in certain cases can be invoked only in these settings. In other words, optional configuration becomes

domain-oriented, which is likely quite appropriate. In cases where this optional module would be purely downstream from everything else in the current root — and when it's a significant portion of the TF state wherever it's included — you may consider a separate TF project that gets applied only in the necessary contexts. This not only avoids a mixed module, but also the situation where you end up having to create several more modules to prevent mixing `resource{}` and `module{}` blocks.

There are many ways to parameterize values in TF projects. The following is a review of how each can be used in successful configurations:

- Provider-documented resource field defaults, the value of the field if it's otherwise unset in the `resource{}` block, is best for when this default is a necessary setting for your purposes or what your corporate policy dictates.
- Module variable defaults are best for true default values: those that serve a majority of such resource instances, but that may be overridden in legitimate scenarios. These are not a safe way to fix values that must not be changed.
- Local values are more appropriate for values that you would like to parameterize but that must remain fixed.
- Logic modules can be utilized as a way to further abstract various values out of a root module. Although they're typically used to compute certain values, they can also be configured to return static values, further protecting them from modification by operators of the root module.

Throughout the book, you've encountered the guidance to avoid placing logic directly inline with the `resource{}` block field assignment, for example. Although there may be acceptable forms that are promoted by the organization — such as simple string interpolation that combines values referenced from another resource's attributes or from the `each` iterator object within the context of a resource that has the `for_each` meta-argument — the general recommendation is to abstract this logic out of the

`resource{}` blocks and into the `locals{}` block or a logic module. This enables several improvements, including ordering or grouping values and naming them such that they indicate their result or purpose, taking a *self-documenting* approach to value construction.

One approach that seems to be especially attractive is using command-line string replacement tools such as `sed` or `awk` to replace placeholder values in places that cannot leverage input variables or to cut out portions of the configuration, such as environment-specific configuration. This latter method can definitely work, as exhibited by several Fortune 500 companies, but aside from being inelegant and inefficient — and aside from the massive files this commonly produces — it can greatly suffer from repetition, with many more changes in the case of a new policy or standard affecting resources throughout. A better solution is typically to parameterize resources such that the same configuration can be used in all of the environments, each with a separate set of variable values, perhaps in a dedicated *.tfvars* file, and a dedicated backend, such as by passing in a different key or prefix for each.

On Source Control

In most modern TF ecosystems, and IaC in general, a source control system such as Git is a key element. It stores the configuration files and allows for collaboration among a team through a push/pull mechanism. Of course, it's not uniquely useful to IaC and is an equally integral tool in many other types of software. As such, although this section will be framed in terms of TF projects, the considerations presented could also apply in other settings.

> **NOTE**
>
> It's worth noting that although this section effectively assumes you're using Git, it's not the only choice in source control management tools. However, the points presented here may largely carry over to whatever tool you are using.

Branch Names and Commit Messages

As presented briefly in Chapter 10, two of the most undervalued practices in using source control is choosing effective branch names and including detailed commit messages. Assuming your team adheres to a particular branching strategy, you may have certain elements that are supposed to be included, such as a prefix of `feature/`, `bug-fix/`, or `release/`. These may even be enforced, such as by using rules in your source control system. Certain other methodologies also include branch name considerations, such as requiring a task or issue ID as part of the branch name to tie code changes to assignments on a task tracking dashboard.

Similarly, your processes may expect certain information to be found in commit messages — for example, this may be where the task ID appears, rather than the branch name. Beyond these requirements, there may not be a strong set of criteria for constructing effective branch names and commit messages.

The following are bad examples of branch names and commit messages that may not violate any technical requirements:

Branch name minimally adheres to policy

> For example, `feature/TAS-11-123`, which identifies the team and ticket but is otherwise meaningless at face value.

Branch name includes your name and the date

> For example, `bug-fix/robert-4-23`, which, for one, is also basically worthless, and moreover, may not even convey the correct information — those who write dates with the day first may not recognize this as April 23.

Branch name omits part of purpose

> For example, `feature/does-x`, but that also includes a completely separate change not captured by the name. This gives anyone who doesn't dig into the changes of the commit a false understanding of what

it does, which can serve to (perhaps temporarily) thwart a root-cause investigation that begins with a review of the commit history.

Commit message is overly generic

For example, `"[TAS-11-456] fixes the thing"`, `"[TAS-11-789] needs more work"`, and `"[TAS-11-1011] implements robert's suggestion"`. Aside from relating back to a task, these messages are empty.

Commit message omits part of contents

For example, in a commit that introduces two resources, `resource_a` with display name `instance-a` and `resource_b` with display name `instance-b`, using `"[TAS-11-1213] Adding a resource_a named 'instance-a'"` as the commit message. This omits the other change, hiding it from a review of the Git history.

By seeing these bad examples, you can begin to formulate standards for what a good branch name or commit message looks like. Branch names should encode the full purpose of the branch in a format meaningful to most, if not all, of the team. Although, for your own sake, the branch name will likely be somewhat constrained in length, it can theoretically stretch to around 250 characters (minus the name prefix length). Therefore, you can get highly specific and comprehensive with branch names, following a self-documenting approach, if you so choose. For commit messages, there is no specific technical limit to how long they can be. They are also intended to be purely *free-text* — rather than a connected string of nonwhitespace characters, as in the case of the branch name — which means you can (and should) include complete sentences. Assuming you're pushing more often than once per day or per branch, you can greatly expand on the purpose encoded in the branch name by detailing all of the changes of each commit. This serves to document the solution and the journey of implementing it.

> **NOTE**
>
> An industry standard that is commonly promoted for commit messages is the so-called *50/72 rule*, which recommends a maximum of 50 characters for a commit title and 72 characters for each line of a commit message. This is primarily due to how commit messages are presented in popular source control SaaS solutions such as GitHub and GitLab. Getting so far as for this to be a point of consideration is likely beyond the guidance presented here.

Certainly, in the process of addressing a particular task, such as a feature request, you may realize the purpose of your work has changed. Perhaps to implement the new feature, you had to correct or modify another portion of the codebase. If this is relatively minor or innocuous, it may be sufficient to only incorporate this additional scope into the commit message, at least ensuring that it's recorded in the commit history. However, a systemic or significant change may be appropriate to surface to the branch name, itself. In which case, the existing branch name may now be insufficient. Luckily, it's simply a matter of establishing a new branch off of the existing one with the more accurate name. Then the old branch can be safely deleted from your local Git system using the `git branch -D` command. If your existing work hasn't been pushed to the central repository, you may even avoid any confusion that may arise by pushing a second branch for the same task. Otherwise, you'll likely want to delete the remote copy, either within your source control platform's console or using the `git push :` command (prepending the branch name with the `:` character).

Likewise, it's not uncommon for the author of a big chunk of work to temporarily forget something that was included. Fortunately, again, Git includes a mechanism for modifying a commit message using the `git commit --amend` command. However, this should be done before pushing the commit to the remote repository — changing the commit message after it has been pushed requires using the `--force` flag, which effectively rewrites the Git history (and which should be reserved in only very extreme cases).

Merging Branches

Another component to source control hygiene is how branches are merged back into a common branch. The options are as follows:

Merge commit

> This is the default in many popular systems. It adds the same commits, identified by their original hash, into the common branch, and adds an additional, immaterial, commit that is purely meant to identify when the branch was merged. That way, all the commit messages are preserved, with an additional message denoting the merge event.

Rebase and merge

> This option is similar to the merge commit, except the commits from the merging-in branch get rewritten as new commits (with new hashes) directly on the common branch. This also preserves commit messages, but it doesn't create any superfluous commits.

Squash and merge

> This final option reduces all the commits in a branch to a single, combined commit and writes this to the common branch. This loses fidelity in terms of the individual changes that were made throughout the process of developing the branch.

> **NOTE**
>
> This commentary doesn't consider all the merge strategies available in Git, but is focused on the pull/merge request process.

Arguably, the merge commit method is indeed the most appropriate in the majority of situations. First, it's easiest to comprehend, as the exact same commits show up in the common branch as they existed in the, for example, feature branch. Second, it can help in the situation where you need to pinpoint when certain issues were introduced. These situations are

(hopefully) extremely rare, which can cause them to be overlooked or never considered until they become problematic. They are also usually not especially obvious, instead appearing intermittently or only manifesting as symptoms expressed in unexpected settings. If there is any historical record that exists of the issue, the merge commits, which identify the points in time when each branch was introduced, can be compared against it to help pinpoint when it was introduced. Although this may enable the surgical extraction of a specific commit, this would violate a *forever forward* approach. Instead of rewriting your Git commit history, you can use the errant commit as a guide for the remediation exercise. Finally, the merge commit itself typically will include the name of the branch that was merged in, so if you follow a well-defined naming convention for branches, this preserves that information in the common branch's history.

Rebase is also most likely fine in many situations, especially if other source control quality considerations are maintained, such as well-defined branch naming conventions and descriptive commit messages that include whatever syntactic sugar is needed for ancillary systems. However, rebase does rewrite history, and in particular, it rewrites when the commit was made, changing it from the time the original author made the commit to the time that the feature branch, for example, was merged into the common branch. For branches that span multiple days and saw significant changes to more than just the initial purpose, this can confuse the memories of anyone trying to recall when and how whatever changes were made. That being said, rebase does, arguably, as good a job as does a merge commit for the purpose of indicating when a problem was introduced into the common branch, which may be used to significantly narrow down the list of possible commits.

Squash merging also rewrites history, glossing over the evolution and losing out on the thought process that went into the solution, focusing on only the end result. As such, it should probably be reserved for specific cases, such as condensing the various works-in-progress to record to a release branch, get rid of a commit that included sensitive information from the history while preserving all the other changes, or when you're developing open

source software on behalf of a company and want to avoid showing any mistakes made along the way. For normal work on a common branch with proprietary code, it's likely best to preserve the paper trail of branch names and commit messages that the merge commit method brings.

One place where squash merges can be highly effective in almost all settings is for personal exploration in the midst of an uncertain path forward. For example, if you know you need to include a package that performs some utility, but you're not sure, say, which package, or version, to target, you may find yourself making a number of different changes, any of which you might want to preserve, especially if this stretches several hours or days. You can create a branch to track this work in progress (WIP) and make several exploratory commits to it. When you have a solution that works and is compliant, you may locally squash-merge it into your actual feature branch. If you intentionally do not push up this WIP branch, you can theoretically even preserve sensitive or noncompliant code that simplifies local development, as long as this is removed before the branch is merged — the squash will eliminate it from the Git history, ensuring you're not slipping in any sensitive information in the changesets. Likewise, and more importantly, none of the unnecessary (or sensitive) changes will be preserved when the feature branch is set to merge into the common branch (perhaps this time with a merge commit approach).

.gitignore

Many tools create files that aren't especially beneficial to share. They aren't meaningful to others and, if anything, they probably would complicate things on their own systems; frankly, you wouldn't want certain files from your team members, either. This is what a *.gitignore* file is for in a Git repository. It keeps random and sensitive files that you create or that get generated out of your version control. There are dozens of suggested *.gitignore* files available and many tools that will generate one for you. Example 11-1 is yet another version of such a *.gitignore* file.

Example 11-1. Stock .gitignore *file*

```
# Local .terraform directories (includes plugins, modules, and so
on)
.terraform/

# Terraform state files
*.tfstate
*.tfstate.*

# CLI configuration files
.terraformrc
terraform.rc

# Plan output files
*.tfplan
*.plan

# Log files
*.log

# Temp files
tmp/
temp/
```

stash

A sometimes overlooked feature of Git is the `stash` command. This places unstaged changes — those changes that haven't been added using the `git add` command — on a local stack for later recall. While this may seem like a limited tool, it has many useful implications.

This is especially useful in cases where you want to temporarily restore your local branch to whatever is on the local `HEAD` commit. For example, you may be working on a shared branch and want to make sure you have the latest version from the central repository before you commit your changes. You can quickly store your changes in the stash stack, pull down the latest copy of the branch, and apply those changes from the stack. Another example is when you think you're on a particular branch, but you're actually on the wrong branch, and you have several changes you want to include. You can quickly stash your work, checkout the correct branch, make sure your local copy is up-to-date, and again apply the changes from the stash stack, as depicted in Example 11-2.

Example 11-2. Using `stash` *to store changes before pulling down the remote branch (some lines removed for brevity)*

```
% git status
On branch feature/wrong-branch
Changes not staged for commit:
    modified:   main.tf

no changes added to commit (use "git add" and/or "git commit -a")

% git stash
Saved working directory and index state WIP on feature/wrong-
branch: 468d684
previous commit's message

% git checkout feature/right-branch
Switched to branch 'feature/right-branch'
Your branch is up to date with 'origin/feature/right-branch'.

% git pull origin feature/right-branch
From github.com:your-org/your-project-repo
 * branch            feature/right-branch       -> FETCH_HEAD
   abcd2e2..1a2b999  feature/right-branch       ->
origin/feature/right-branch
Updating abcd2e2..1a2b999
Fast-forward
 main.tf                                                          |
14 ++++---
 1 file changed, 9 insertions(+), 5 deletions(-)

% git stash pop
On branch feature/right-branch
Your branch is up to date with 'origin/feature/right-branch'.

Changes not staged for commit:
    modified:   main.tf

no changes added to commit (use "git add" and/or "git commit -a")
Dropped refs/stash@{0} (aa205bb3332904593ef26e817bc3cb8ebec246cf)
```

> **NOTE**
>
> One caveat to using `git stash` is that the `pop` subcommand effectively merges the changes stored on the stack into the branch. Therefore, using it can produce merge conflicts. However, unlike a normal `git merge` command, merge conflicts produced by `git stash pop` cannot be canceled directly because it does not have an equivalent to `git merge --abort`. Instead, you must reset your `HEAD` commit with `git reset --hard HEAD`. Unfortunately, this will not only remove the changes introduced from the stash stack, but also any other uncommitted changes you may have. Also, in the case of `git stash pop` that results in a merge conflict, the commit is not removed from the stack. Thus, if you accidentally ran `git stash pop` on top of a stale version of a branch before you ensured it was up-to-date with `git pull`, you can reset the `HEAD`, pull the latest from the remote branch, and `git stash pop` again. If you end up resolving the merge conflicts by hand, you would need to manually remove the entry from the stash stack using `git stash drop`, as described next.

Because `git stash` pushes to a stack, you can store many changes, one after another, and recall them in a last-in-first-out (LIFO) fashion. Likewise, rather than recalling the last change on the stash stack with `git stash pop`, you can choose to eliminate it with the `git stash drop` command. These two commands unlock several interesting techniques, all of which are beyond the scope of this book.

Over-Trusting Branching Strategies

Throughout the book, you've encountered the perspective that because TF (and IaC in general) is very different from application development, it takes a different approach to development. A typical, high-quality infrastructure automation platform will likely indeed resemble an equally quality DevOps or DevSecOps setup to support your organization's software delivery lifecycle (SDLC). Both will have a concept of workflow or pipeline, with a collection of tasks that may be combined or triggered in different configurations for a specific purpose. Most likely, both will have many such collections, some of which are meant to be operated in a strictly ordered fashion. Finally, both will probably be triggered primarily by certain changes to a specified source control repository.

> **NOTE**
>
> If you aren't running TFE or HCP Terraform, such as if you're leveraging OpenTofu, you may even use the exact same platform for your TF as for your SDLC, such as GitHub, GitLab, or a CSP-managed solution. In these cases, it's even more crucial that you recognize the differences between the concerns of infrastructure and application and the implications that this has.

However, the purpose of TF as an IaC platform is to be the foundation for your applications. This usually means it must be constructed so as to support multiple copies of the application in a segmented fashion, the environments in your SDLC, along with singleton resources that may be shared across your account or large portions thereof. As such, whereas an application's SDLC may tolerate certain defects, inefficiencies, or vulnerabilities in early environments, there's no such tolerance anywhere in your cloud infrastructure. Likewise, even though conceptually you may be managing seemingly raw infrastructure components, in reality, you're really running software-defined resources. Most likely, you did not build these cloud services, meaning it's not your software. You've seen how this changes the approach for validating your TF projects, theoretically reducing much of it to static code analysis and monitoring for things such as unhealthy resources and configuration drift.

> **NOTE**
>
> Even if you're using a bare metal solution — which is allegedly a full machine, rather than a slice of much larger machine — the device itself is exposed through software-defined (or perhaps, software-assisted) utility.

All of this to say that popular techniques that are so successful in the context of SDLC may not be appropriate to apply without modification to automated TF systems. In particular, one technique that should be thought through is your source control branch strategy as it relates to your infrastructure release procedure. As you saw previously in the chapter, a pure application of a trunk-based, or similar, Git workflow can result in

situations where the main branch is updated multiple times, but one of them (not the last) has unexpected changes in a later environment.

You may control different environments through the use of workspaces or by configuring a different backend and variable values in different contexts — or you may even be introducing some dark magic that fully expresses all of the environments in a single TF project and optionally applies a portion to the desired environment, each with a unique backend — but you cannot guarantee, for example, that what worked in a development environment will work in production. For one, although development and production should resemble each other — with development potentially more flexible in some ways and more constrained in others — production will likely have a more comprehensive configuration than development does and may include enhancements such as higher-throughput networks, longer-term retention policies, and more complete availability and recovery objectives. You may even narrow a development environment to a single region or small set of regions. If you don't ensure a newly supported feature is available in all your regions, you may not catch the lack of support until you attempt to configure it in an environment that contains the unsupported region(s).

This quality is further intensified in the case of needing to hotfix your production environment, for example. Although the change may be progressed through your various environments using your normal IaC workflow, the change may not actually affect earlier environments. This means there's no way of validating the change before it reaches production. If the change is especially risky, and assuming the change does not involve a singleton resource, you may choose to temporarily introduce the feature or resource in a nonproduction environment for the purpose of validating it. However, unless this is done outside of your normal IaC workflow, such as using a wildcard job or within an ephemeral environment, and is cleaned up accordingly, you'll need to make one more change: to remove the temporary configuration and work this through the IaC workflow to at least the environment that served as the testbed.

What you certainly should avoid is attempting to maintain multiple long-running branches, one for each environment. The main problem with this is it invites hotfixes to individual environment-dedicated branches, which then have to be merged into the other branches. Ideally, this would not have a material effect — resulting in a no-op `tf apply` command — but it still can run the risk of a merge conflict, if any of the *.tf* files were modified in this other environment. Using a common branch to release against is typically the cleanest way, but there must be a mechanism for modifying a single environment outside of the normal workflow without violating the forever forward paradigm, such as by using a wildcard job.

Example Nightmare Scenario

In the absence of strong policies — that are not only encouraged, but also enforced — it really only requires a moderate level of discipline and accountability to ensure these elements are well-kept. This is often best addressed by live, thorough, team-wide code reviews that are designed to call out departures from the guidelines. However, it doesn't take much for this to begin to deteriorate. Consider the following scenario:

- An urgent issue on, say, the evening before a long weekend causes the team to gather for an all-hands-on-deck situation.
- As folks trickle in, they're brought up to speed and begin to investigate.
- A group strategy session is scheduled after an hour or two to compare notes and discuss next steps.
- A small team — the *fix team* — is selected to continue a more focused investigation along multiple vectors, and a follow-up session is scheduled (for not too late) to reconvene and present the fix or latest findings.
- The fix team gets further along, but is still working on a solution when the follow-up session begins.

- The fix team presents its findings, current steps, and an estimation for when the fix can be released — perhaps this projection is later that night, but beyond a comfortable time to require everyone to be available (maybe because of contractors or hourly engineers).
- A minimal cadre of the wider team — the *approval team* — that can responsibly approve such a release is tapped to reconnect at the agreed upon deadline. As suggested, this likely does not include many, if any, of the engineers outside of the fix team.
- The fix team completes a solution and presents it to the approval team as soon as possible.
- The approval team effectively rubber stamps the release, focusing on functionality and ignoring unenforced quality requirements.
- The fix team — or most likely one or two members of it — shepherds the resolution through the release workflow.
- Everyone signs off for the night.

In all the excitement, and because of the highly inopportune timing, normal practices that rely on discipline run the risk of being deprioritized in order to reach a resolution as soon as possible. They may even be directly undermined by authority figures that must weigh risks and considerations that go beyond purely technical quality concerns. This creates the perfect conditions for cultivating technical debt. If the subversion of your normal practices is done *de facto* — as a result of such a wider decision process that may not even consider the practices in the calculation — this technical debt may not be accounted for.

Unless it's caught early, this technical debt can have serious ramifications that become more and more esoteric as time goes by. Unfortunately, the experience was so uncomfortable and inconvenient that everyone got it off of their radar and put it out of their mind as soon as possible. Similarly, no one wants to touch the code involved because of the trauma or fear it evokes. It becomes a fixture that is forgotten or deliberately ignored. The

debt associated with the fix is kept indefinitely, and any additional relevant changes are constructed around it, likely adding additional debt through otherwise unnecessary workarounds. If anything truly problematic with the fix arises, it may take an overinflated effort to reassemble the context within which the fix was created; this may only present itself after a significant amount of time, which is unfortunate since the experience is only exacerbated over time.

Sometimes worse than a single especially bad commit (or a few) is the damage it can do to the discipline necessary to keep up the expected quality. Usually the fix team is populated by more senior personnel, all of whom (traditionally) are human and can therefore make mistakes or cut corners in the heat of troubleshooting. In well-regimented teams, or for highly fastidious individuals, this may be quickly mitigated. However, any residual fatigue may cause a longer lapse in rigor that can start slow but become unwieldy after just a few sprints. Reeling this behavior back in may be met with great resistance or chagrin, which is also intensified as time goes by. Even if this does get corrected, the time that the procedures were relaxed can corrupt a coherent historical record. Even more frustrating, if a sleeper issue hides somewhere in the source control commit history, and you don't have an intelligible narrative of what changed over time, you may find yourself combing through Git blames and commit diffs to understand how and why things changed. A sprint's worth of commits is one thing; a program increment or quarter's worth risks a descension into madness.

The Flawed Merge Conflict

Suppose you're working to combine a feature branch with the latest version of the common main branch and you run into merge conflicts. If you're the sole author of the feature branch, and it's not too extensive a conflict, you should be able to effectively resolve the conflicts without much issue.

However, suppose hypothetically you do make a mistake in resolving the conflict, and you accidentally remove a necessary resource. It's possible what you end up with doesn't violate any of your static code analysis or tests or fail in the planning phase of a TF operation. Although you should

be thoroughly analyzing your plan, if the inadvertent removal was buried among other legitimate resource removals, it's conceivable that you could overlook it. Unless there are CSP-level protections, it may not even fail to apply.

Most likely, if it gets this far, it won't be immediately obvious that things are wrong. It might be minutes or hours before anyone even notices, and it may go through multiple channels before the missing resource is identified and the problematic changes to the TF code are discovered. Depending on when the deletion occurs, it can feasibly take several hours, or even days, before a solution is released. Among the worst case scenarios is if the deleted resource is one that cannot be easily and expediently recreated and reconfigured.

Most popular Git SaaS products support a high-quality diff tool that can be used to simulate a merge from one branch to another. What's most useful is that they will even assert potential merge conflicts. However, it's recommended to resolve these reported merge conflicts in a new branch, locally, instead of on the actual branch that causes the conflict. This way, any mistaken merges can be discarded, even after accepting a merge — thus not having the ability to simply cancel the merge using `git merge --abort`. Once you have such a branch that resolves the merge conflicts, you can use this in the merge request, rather than the original feature branch. Alternatively, you can merge this conflict-resolving branch into your feature branch and continue as usual.

Getting Unstuck

Let's face it, developing TF is not a straightforward exercise. It's at the nexus point of so many considerations — the needs of your application workloads, your security posture, and your company's organization structure. Most likely, your cloud is constantly evolving, with new features and resources being added, and old ones being removed, fairly frequently. There's no "done and dusted." Everything is subject to review, modification, and reorganization at all times. On top of that, TF itself has many

idiosyncrasies, gotchas, and limitations, and practitioners sometimes go to extremes to get around these. Usually, the thought process goes like this:

1. You have a massive TF project.
    a. Some of the resources can be recreated without problem, and some must remain running.
    b. The system is too fragile to risk any deviation from the existing resource configuration.
    c. (You might not even be aware of all the resources.)
2. You need to make a systemic change to the project, such as split it into multiple TF projects or spread among multiple backends.
    a. Perhaps this is due to some technical limitation, ordinance policy, or leadership mandate.
3. You realize that rearchitecting it is A) time consuming and B) risky.
    a. Maybe you're in a highly regulated industry, and even constructing the plan would take longer than expected (let alone the actual implementation).
    b. Maybe you have proper guardrails that prevent you from manual changes, but you don't have a safe way to make certain changes like editing the TF state.
4. You come up with a workaround solution.
    a. This might use other tools to account for limitations in TF.
    b. This might have multitude corner cases that are hard to predict.
    c. This might be something that one colleague did right before they left.

5. It works until it doesn't.

6. Repeat Steps 1 through 5.

In other words, the solution that your entire cloud footprint relies on has run into a challenge and the "right way" to solve it is too costly, leading you to "think outside the box" and come up with a hacky solution. It's only natural to seek a quick workaround to fit the proverbial square peg that is your current solution into the round hole representing your requirements and constraints. The problem with this quick workaround is that it's often framed in theory as a temporary solution but in practice it ends up staying indefinitely, until a problem occurs and the path forward is unclear.

Because your TF ecosystem establishes your foundational cloud resources, when things begin to go wrong, it can get very treacherous very fast. It can quickly become an emergency, usually triggering the involvement of far more people than can possibly help to resolve it. Without the experience or awareness of how to proceed in such situations, you may be stuck making web searches and reading industry forums. In this section, you'll explore ways to get out of precarious situations that you weren't able to mitigate or avoid.

Duplicating Modules

One phenomenon that seems to happen at all organizations at some point is a duplication of modules. The typical scenario is something like the following: one team (Team A) creates a module that bundles a compute resource along with its underlying components such as boot and attached disks, identity elements, and network attachments; later, another team (Team B) creates a second module that bundles some or all of these components. Very likely, this is done rationally. For example, Team A's module may not support the same features that Team B intends to utilize. Or the opposite is true: Team A's module leverages features Team B wants to avoid, and does not expose a way to control them. In some cases, Team B may not even be aware of Team A's module, especially if it's a local module

tucked away inside a repository that Team B has no obvious business combing through.

Surely, along with accidents, there are justifications (both contrived and legitimate) for intentionally producing such duplicates. These might be based on some policy disallowing remote modules — thereby requiring every TF project to create whatever modules it needs, locally — or an imposed distinction between modules used in different contexts, such as one for highly sensitive workloads and another for general purpose workloads. It may even be a policy — written or unwritten — that each team uses its own modules exclusively. None of these situations are inherently problematic, but it does violate the DRY principle, assuming you take it to the extreme of applying it to your entire organization. The canonical problem with repeated logic is the need to update all copies of the logic in the event that it becomes subject to a new policy or other constraint. In our previous example, if the organization to which Teams A and B belong restricts the OS that may be launched on a cloud compute service instance, both team's modules will need to reflect this restriction. If the company switches managed support from Canonical to Red Hat, both teams will have to update their modules accordingly, repeating the work.

When many teams have their own copy of effectively the same module, in terms of intent and primary components, it not only becomes a nuisance to update common features, it also gets increasingly harder to impose a common set of policies across them. As you impose more and more constraints on the modules over time, they'll begin to resemble each other, perhaps roughly, in terms of inputs, outputs, and attributes, anyway. Taking this to the extreme, at some point the only differences between modules will be their input variable names and default values, output value names, and any custom logic embedded in them, in which case their differences become arbitrary and keeping them distinct becomes more of an exercise in tradition rather than justification.

Once it's determined that two or more modules are functionally equivalent, and you decide to reduce them to a single supported version, there are several things to consider. If one module (Module 1) is strictly more

expressive than another (Module 2), with every feature supported by Module 2 also fully supported by Module 1 — such that there is nothing configurable in Module 2 that cannot be exactly recreated in Module 1 — it would be possible to drop the support for Module 2 and reconfigure any Module 2 reference to use Module 1. If, instead, it's determined that Module 1 is actually too flexible, supporting features that don't comply with your corporate policies, and Module 2 (which includes a strict subset of support) is shown to comply, regardless of configuration, it's possible, but not guaranteed, that it would fully support the needs currently fulfilled by Module 1, perhaps with some expanded features. Unfortunately, these happy paths are rarely trodden; rather, most often two such modules will have significant differences both in the features they support, with each module including at least one feature or resource not supported by the other, and neither module is obviously superior.

In the most typical case — where no one module is adequate for all teams that would consume it and you have to figure out which module to rally around — it's important that the decision for how to proceed isn't reduced to purely a popularity (or seniority) contest. The most appropriate module may not be the most utilized one. Instead, you must consider all factors involved: expressiveness, compliance adherence, maintainability (which includes readability in addition to the delivery toolchain and procedure), and so on. Ideally, an objective review of the modules will reveal a clear path forward, and one of your modules is considered significantly more suitable than all others. If this is not the case, hopefully there is at least one module that is at least as suitable as all others and is superior in at least one factor. In cases where the various modules are all effectively equally lacking, an optimal solution may be to create a net new module that does promote such factors. In reality, this may be the best approach, in general. It's likely that the team that will own the module moving forward is not the team that developed it. Although the permissions and responsibilities for the module and source repository can be transitioned to a new team, this may be more trouble than it's worth. Even when you have a module that is a clear winner, it may be better to use it more as a guide and create the new module with analogous supported features and at least as effective quality.

The best way to prevent this is by investing in a module registry early on. Both TFE and HCP Terraform have a module registry built in, but there are several other options that exist, many of which are free to use. Although these may differ in support and certain self-hosted solutions may require additional consideration and maintenance effort, the real challenge is usually in simply using the registry. Even when only a single team manages the infrastructure with TF, it's valuable to use a registry for any modules that get reused in different TF projects. As the TF ecosystem evolves and the responsibility of managing it is spread among several teams, it becomes even more valuable to already have experience with a registry. You can leverage the experience of having full reign over the registry in devising the operational strategy, and you can mentor new operators in how to consume from, and produce for, the registry.

Although unrestricted ability to create and update modules is understandably unsettling, an overly restrictive system may severely impact its usage. Introducing sensible, but not overbearing, restrictions early on, coupled with a few sensible observability controls that provide clarity into what modules are seeing in terms of development and use, can cultivate a practical assortment of modules that enjoys a high degree of value and relatively low superfluity in terms of duplication and cruft, organically.

Another challenge is developing a company-wide standard for versioning and releasing new modules. Certainly, modules will advance at their own pace, but if every module has its own notion of a version and different rules, for example, for advancing major and minor versions, they lose clarity in purpose, especially when considered by an initiated party. In the case of deliberate module overlap — where two modules include some of the same resources — this standard should also indicate when modules should be combined and when they should remain distinct.

Don't Own the Remote and Need an Update?

One common situation is when you're using a third-party remote module, which undoubtedly has some limitations, and you need it to handle a case that isn't supported. Assuming you have access to the module's source code

and have some mechanism for suggesting an update such as an issue ticketing system — or better yet, offering an update for example through a merge or PR of a forked repository — you can attempt to utilize this channel for implementing the desired support. However, further assuming you do not have permission to edit the module's source repository directly, the module's maintainers must accept the modification. Even in the case of a complete and well-formed PR served up without frills or conflicts, there's still a chance it will be rejected either as an explicit or an implicit design choice, meaning the feature implementation is either counter to or outside the scope of the module's purpose.

In the case of an unreachable or intractable module owner, the next best solution might be to use a fork of the module source directly. This gives you control over the module's future and can be kept in sync with the original repository. However, leveraging a Git repository as the module's source location requires managing credentials, such as an SSH key, which can be cumbersome and may not even be allowed according to corporate policy. This may mean setting up an artifact registry of some sort to store the module for distribution. In the situation at hand, this may be a much heavier solution than you're willing to take on, especially for a seemingly minor change like supporting a binary flag. Also, while keeping the fork in sync with its original is possible, it may need to be done manually (or through some additional process), which can be tedious if the original repository gets updated regularly.

> **NOTE**
>
> TFE and HCP Terraform both have a built-in private module registry, resolving the question of where to host a remotely consumed module.

Sometimes what you want to modify about a module is not an attribute of an included resource, but instead the introduction of an additional resource that you want to create and manage alongside the resources currently supported by the leaf module. In this case, a local *bolt-on module* —

whereby a module is introduced into the TF project that is designed to bundle a resource or group of resources with an existing module — becomes worth considering as a (hopefully temporary) workaround. In this case, there's no additional system or credentials needed, and you don't even really need read access to the remote module's source. The bolt-on module becomes middleware that establishes the desired additional resource and otherwise passes along the necessary inputs to the same remote module as before. However, you'll likely have to modify the TF state, resource by resource, which can be cumbersome if the remote module manages many of them.

Unexpected Changes Reported In Specific Environments

Another scenario from which no system seems to be immune is the chance of an unexpected change reported in the automation workflow for an advanced environment such as staging or, hopefully not, production. It can be only one of two things: either something was changed in the environment outside of TF and simply needs to be reverted — in which case, the question is "why?" — or, regardless of how it emerged, the change that will be reverted actually needs to be preserved. In the former case, while the fix is quite literally a push of a button, the investigation likely requires reviewing access logs to pinpoint how the modification was introduced. If it was introduced by a human operator, it may require an update to the IAM controls that govern the modified resource. If it was made by a service identity, it may require digging through whatever platform configuration or custom logic that the identity is used in conjunction with (it may also require an IAM update).

The latter case, where the change should be preserved, is far more frustrating since it requires a code change that typically must make its way through the earlier workflows before it can be applied in the correct environment. If there are other changes making their way through the pipeline, it adds a different layer of complexity. You may not be comfortable merging the patch into another change — or it may not be allowed — and you have to leapfrog the change to establish a clean baseline

before moving on. In the worst version of this case, it can even mean having to extract the changes that were already slated to be applied and cleared in the lower environments — an especially unfavorable prospect.

Taking the forever forward perspective, we would want to do this without undoing the existing source control history, such as by using the `git revert` command. If you're using a popular branching strategy like Gitflow or trunk-based, you may have to take some funky measures such as bypassing the normal release procedure and using a wildcard job from Chapter 6 that can target a specific environment and source control branch. Then, the narrowly applied branch can be merged into the same branch that was previously released to the early environments, and the workflow can proceed as per normal, releasing to each environment. The expectation is that now the correction has been applied in the affected environment and should not be reported this time. Figure 11-2 depicts this process.

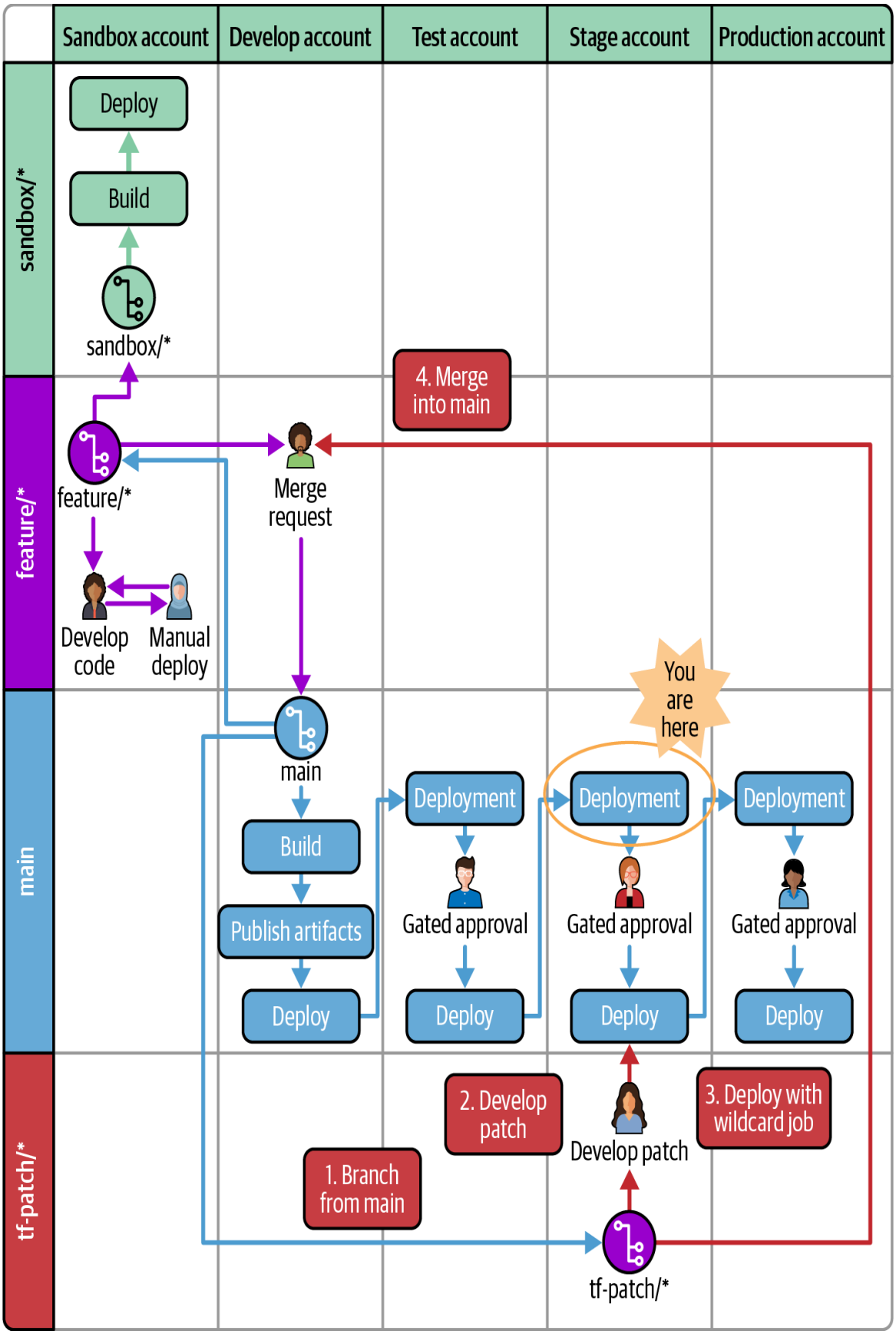
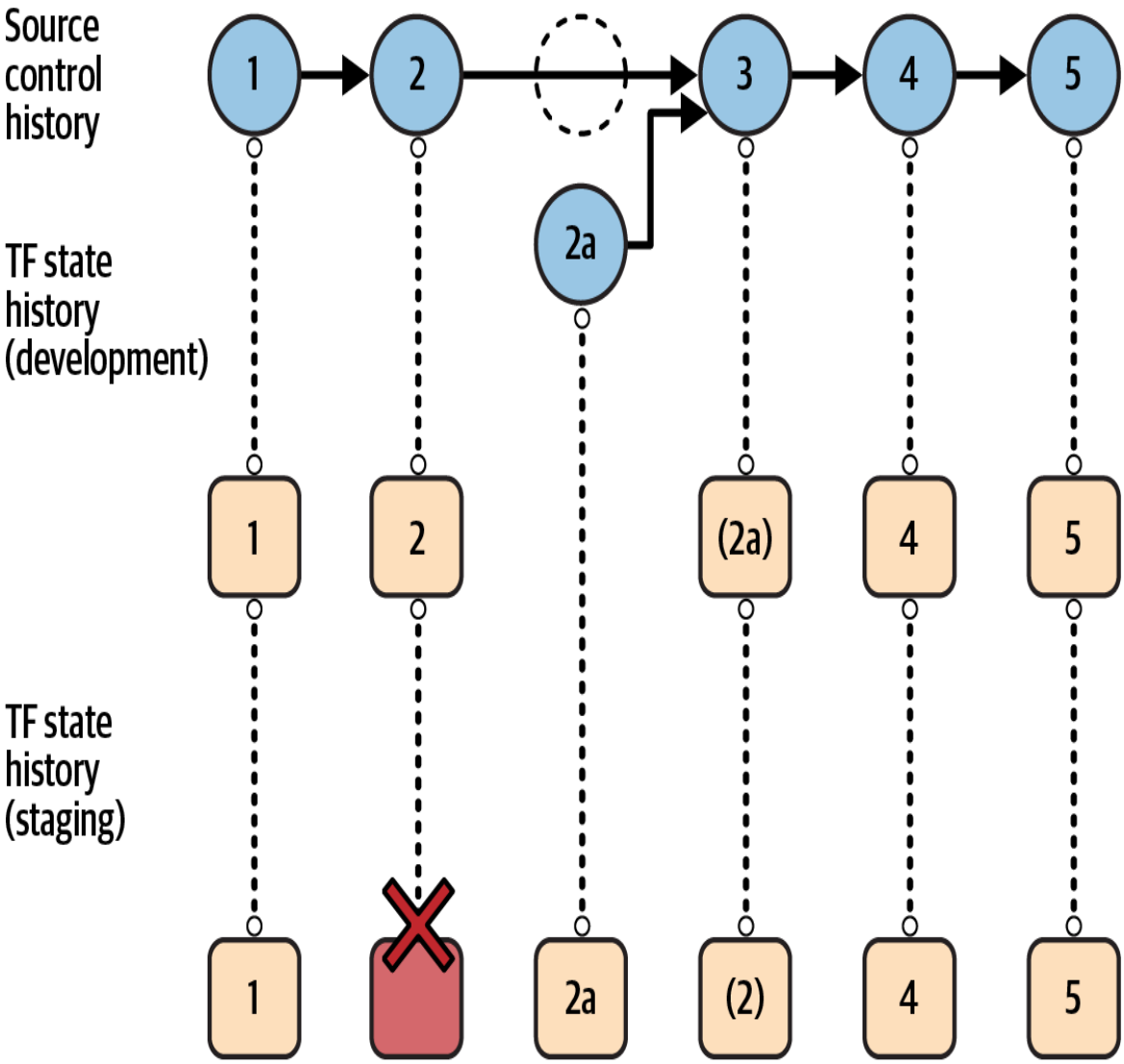

Sandbox account
Develop account
Test account
Stage account
Production account
sandbox/*
feature/*
main
tf-patch/*
Deploy
Build
sandbox/*
feature/*
Develop code
Manual deploy
4. Merge into main
Merge request
main
Build
Publish artifacts
Deploy
Deployment
Gated approval
Deploy
Deployment
Gated approval
Deploy
You are here
Deployment
Gated approval
Deploy
2. Develop patch
Develop patch
3. Deploy with wildcard job
1. Branch from main
tf-patch/*

Figure 11-2. Procedure for patching a single environment before evaluating new changes

Technically, this does put things out of order, as shown in Figure 11-3. However, it doesn't create a missing entry in the Git history, and reconciling the out-of-order TF state updates could be as simple as looking through your commits to find the message or branch name that indicates the patch — assuming you aren't squashing your merge requests. Moreover, the change applied to the specific environment should be a no-op for any prior environments (unless something really weird is going on).


Source control history
1
2
3
4
5
2a
TF state history (development)
1
2
(2a)
4
5
TF state history (staging)
1
2a
(2)
4
5

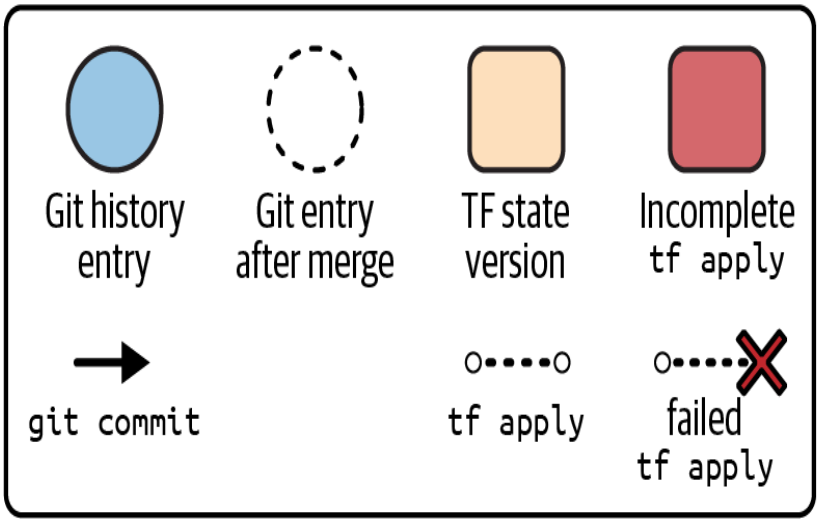

Git history entry
Git entry after merge
TF state version
Incomplete tf apply
git commit
tf apply
failed tf apply

Figure 11-3. Historical representation of patching a single environment

Environment Conflicts

A related phenomenon that can also occur when a single TF project is supporting multiple TF states — such as through workspaces or reinitializing with a new backend — is where a change has the desired effect in one environment but an objectionable or insufficient effect in another. A hypothetical example might be a property that is suppressed in early, development-oriented environments that conflicts with a new feature you intend to introduce in all environments. The first few environments may not report any issues, hiding the problem until it's deployed to release-oriented environments like a staging environment. If the conflict is artificial — for example, if the configuration is technically possible, but not tactically achievable through TF, or a single provider, alone — it may be possible to design a pattern that acts as a workaround or stopgap. If the conflict is real, and the existing feature is nonnegotiable in the environments it's utilized, it may require a totally different solution for the new requirements. Again, taking a forever forward perspective, you would manually remove the configuration rather than utilizing, for example, the `git revert` command. However, instead of releasing a patch directly to the environment, as described in Figure 11-3, the forever forward resolution path would probably follow the normal feature branching process.

> **NOTE**
>
> This scenario is distinct from that of the previous section because that scenario focused on a planned change that you didn't expect, whereas this one implies you do expect a change, but the change that is reported doesn't reflect the correct configuration. One way this can happen is in unsatisfactorily resolving merge conflicts such that the resolution removes support for something specific to one (or a subset) of the environments. The missing support may not be clear until reaching the affected environment.

Preventative Measures

Throughout the book, several preventative measures have been suggested:

Live code reviews

: Making code reviews a group activity. This can be gamified or otherwise structured to ensure participation.

Static code analysis

: Including a tool that checks for known policy violations and security risks.

Policy as code

: Encoding your corporate policies and conventions into a validation step.

Tests

: Developing unit or integration tests that create real resources as a verification of expected behavior. These can also be made to evaluate existing resources — such as those in your live environments — and even be paired with logic modules to construct true unit tests.

Git pre-commit hooks

: Adding in commands like `tf fmt` or `tf validate`, or even one of the other preventative measures, as an operation that happens before your code is committed to the repository can prevent some issues from ever leaving your workstation.

Automated IaC platform observability

: Logs, monitoring dashboards, and alerts that report both anomalous and problematic behavior and that prioritize trust and practicality.

Although it would no doubt be great for each of these to be perfectly implemented, they still would probably miss some unforeseen issues. What's usually most valuable in practice is a culture of discipline that pursues these measures earnestly and pragmatically. Not only will this result in a high level of quality and a reduction in preventable issues, it will

instill your people with confidence and ownership when problems do appear.

Putting It All Together

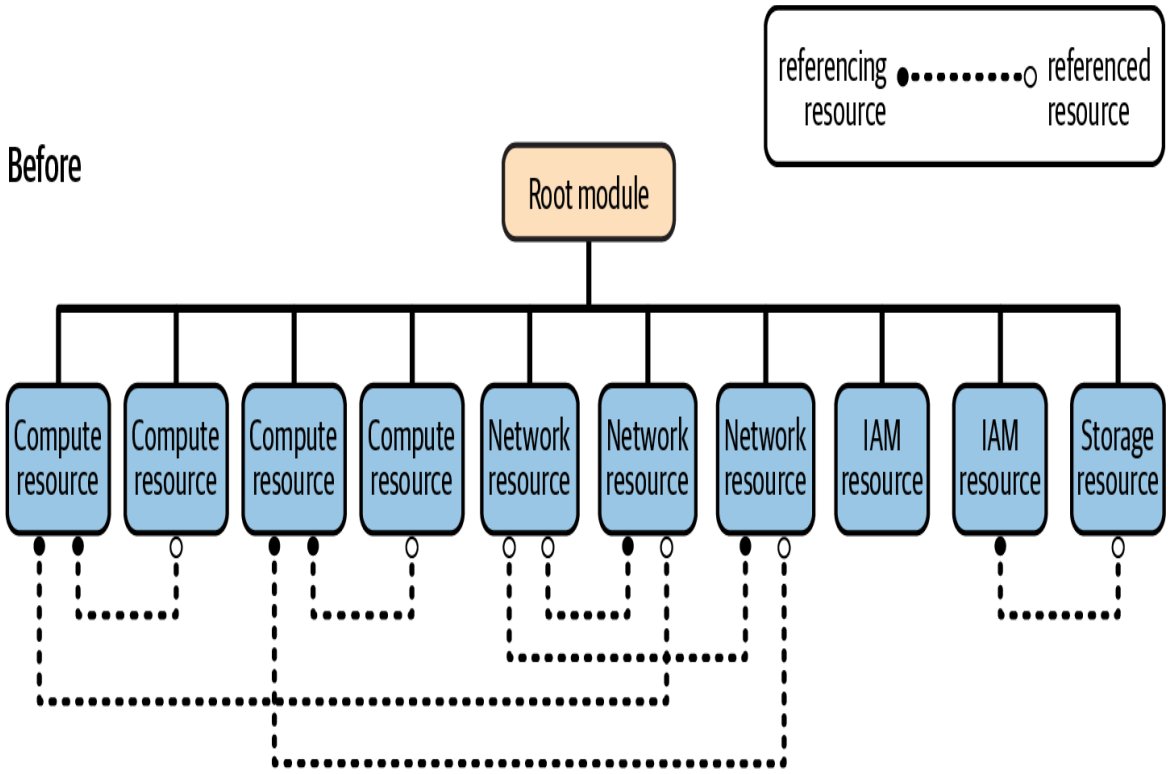
Throughout this book, you've encountered examples of how to perform certain tasks across various subjects: logic and block construction, modules, TF state, automation, validation, and so on. While the goal of these examples was to go beyond the purely canonical or the proverbial *hello world* or *toy example* versions of the specified practice, they were always considered in isolation — with key activities abstracted from other such considerations and relevant integration points, downplayed — leaving it up to you to figure out how the pieces fit together. In this section, you'll see full, end-to-end walkthroughs of various initiatives. This will put a focus on these integration points and provide continuity across all steps.

The format of this section is an ordered list of activities, described in words, followed (in most cases) by an example. It's meant to be a guide: to show the steps involved in a successful implementation, going significantly beyond the tutorial level, which can make assumptions that only hold in perfectly pristine or highly sandboxed settings — an unlikely quality of the vast majority of advanced or mature solutions. It's certainly not intended to be definitive, and it will still truncate or otherwise compress certain components such as command-line outputs and the full contents of modules.

> **NOTE**
>
> Note that certain steps will not have corresponding examples or figures. These steps are less technical in nature; however, three glaring exceptions are any steps related to automation, validation, or source control. Steps that describe these activities or configuration will be unaccompanied by an illustration. The first case, automation, is due to the variation among different platforms; an example in one platform may not be directly translated into another. For validation, it is because many validation steps are predefined and would be executed without modification, and also because of the variation and specificity found in custom checks that are unique to your specific needs, rather than something that can be made universally relevant without inflating this already extensive section. For source control, it is simply due to the assumption that source control is universally understood — to the degree necessary for performing the step — and the step's description will be adequate.

Extracting Resources Into a Module

The first case to consider is where you want to extract a collection of resources into an internally shared remote module, with the purpose of making it usable by other internal teams. At a high level, this means you'll extract the resources into a separate repository, make it available for remote reference, and ensure there are no changes reported by the resulting resource graph. Figure 11-4 depicts the before and after TF states.


referencing resource
referenced resource
Before
Root module
Compute resource
Compute resource
Compute resource
Compute resource
Network resource
Network resource
Network resource
IAM resource
IAM resource
Storage resource

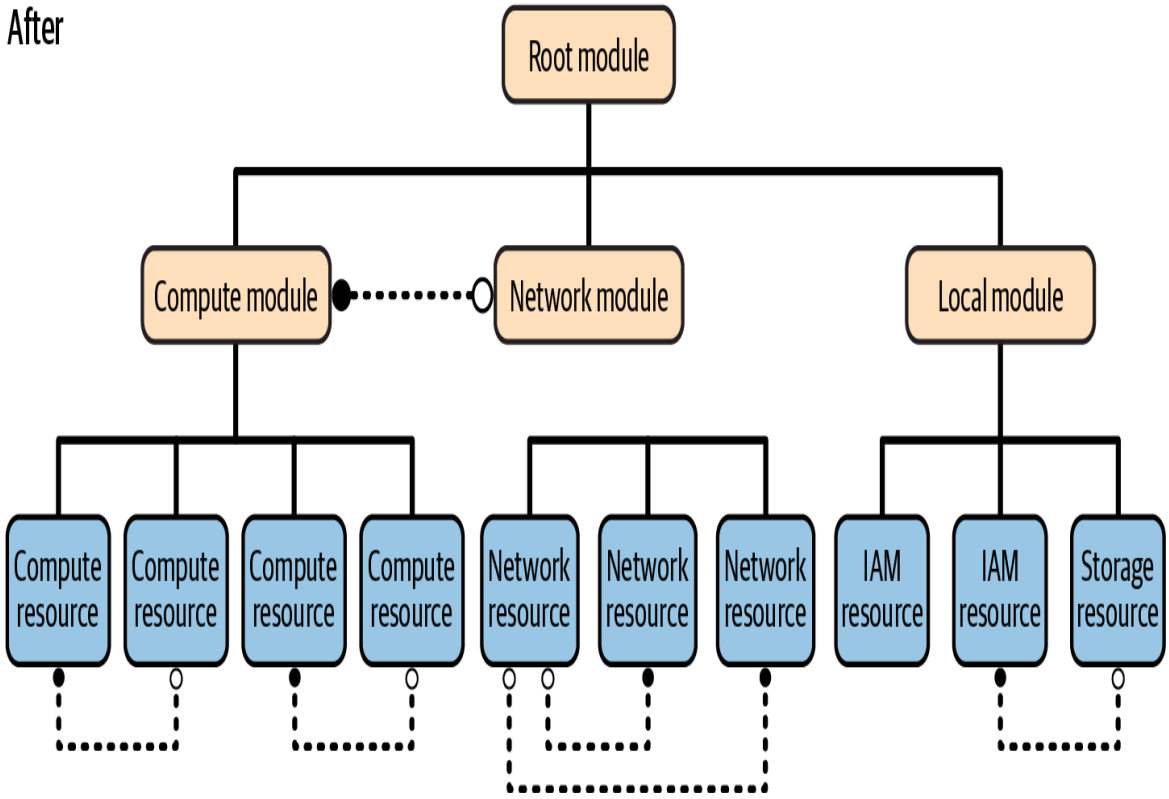

After
Root module
Compute module
Network module
Local module
Compute resource
Compute resource
Compute resource
Compute resource
Network resource
Network resource
Network resource
IAM resource
IAM resource
Storage resource

Figure 11-4. TF state representation before and after extracting the remote module

Step 1

First, you'll need to identify the resources that will move to the module. You should have an idea of what resources you want to standardize with the module — let's call them *primary resources* — so this should be a question of which of the other resources — the *secondary resources* — that the primaries depend on should be packaged along with them. You should consider resources that occupy the same domain, such as a VPC along with a subnet-oriented module. Another good heuristic to consider is to determine whether the resource in question is relevant to only the primary resources, in which case you would definitely include it, or if it's also referenced by other resources that are mutually exclusive with the primary resources. In the latter case, where the resource in question is referenced by various other resources, you should heavily lean on the domain analysis. Suppose we actually identify two modules: a compute module and a network module.

Step 2

Next, you'll want to create a new Git repository to store each remote module. You might consider restricting write access to these repositories, but you should introduce read access, perhaps only to certain branches or tags, such as release branches or published version tags.

> **NOTE**
>
> To reduce excessive sprawl, you may have a desire to bundle your remote modules together into a single repository. Such a repository could be restricted to modules developed by a specific team, allowing a single place to manage access controls. However, this will likely require a complex method of relating the code versions to the module release versions.

Step 3

You'll next want to create a feature branch in the root TF project's repository to track this work. That way you can control when the changes get committed to the main branch and prepare a comprehensive code review with your team.

Step 4

Next, you can create the remote modules. You might cut and paste the resources from the root TF project, establish variables and outputs for any references that will cross either of the new modules' boundary, and identify the variables and outputs that should be directly copied over to the new modules. Remember, you'll still need these variables and outputs exposed in the root module, so you shouldn't remove any of them from the root TF project — although you'll need to fix any output values that reference the resources directly. This also helps in validation as you won't need to change your variable value assignment, such as a *.tfvars* file. Don't forget to update the root TF project with the `module{}` block reference, replacing the `resource{}` definitions that were cut out of it. The configuration for one such remote module and how it's referenced in the root module is depicted in Example 11-3.

Example 11-3. Showing the codebase before and after extracting resources to a remote module

```
#------------------------------------------------------------------
-----
# Before - Single root module containing all resources
#------------------------------------------------------------------
-----
#
////////////////////////////////////////////////////////////////////
////////
# previous-root-module/main.tf - Excerpt (region one, only)
#
////////////////////////////////////////////////////////////////////
////////
resource "google_compute_network" "default" {
  name                    = local.vpc_name
  project                 = var.network_project_id
  auto_create_subnetworks = false
}
```

```
resource "google_compute_subnetwork" "region_one" {
  name          = local.region_one_snet_name
  project       = var.network_project_id
  ip_cidr_range = var.region_one_cidr
  region        = local.region_one
  network       = google_compute_network.default.id
}

resource "google_compute_disk" "region_one" {
  name    = local.region_one_disk_name
  project = var.compute_project_id
  type    = var.disk_type
  zone    = local.zone_one
  size    = var.disk_size
}

resource "google_compute_instance" "region_one" {
  name         = local.region_one_compute_name
  project      = var.project_id
  machine_type = var.machine_type
  zone         = local.zone_one

  network_interface {
    subnetwork = google_compute_subnetwork.region_one.self_link
  }

  attached_disk {
    source      = google_compute_disk.region_one.self_link
  }
}

...

#-----------------------------------------------------------------
-----
# After - Root module contains remote modules
#-----------------------------------------------------------------
-----
#
////////////////////////////////////////////////////////////////
////////
# new-root-module/main.tf - Excerpt
#
////////////////////////////////////////////////////////////////
////////
module "network" {
  source = "tas-module-registry.com/tas-11/network/google"
```

```
  vpc_name        = local.vpc_name
  region_one      = local.region_one
  region_two      = local.region_two
  project_id      = var.network_project_id
  region_one_cidr = var.region_one_cidr
  region_two_cidr = var.region_two_cidr
}

module "compute" {
  source = "tas-module-registry.com/tas-11/compute/google"

  region_one      = local.region_one
  zone_one        = local.zone_one
  project_id      = var.compute_project_id
  disk_type       = var.disk_type
  disk_size       = var.disk_size
  machine_type    = var.machine_type
  boot_disk_image = var.boot_disk_image
  boot_disk_size  = var.boot_disk_size
  subnet_one_link = module.network.subnet_one_link
  subnet_two_link = module.network.subnet_two_link
}

...

#
********************************************************************
********

#
////////////////////////////////////////////////////////////////////
////////
# remote-compute-module/main.tf - Excerpt (region one, only)
#
////////////////////////////////////////////////////////////////////
////////
resource "google_compute_disk" "region_one" {
  name    = local.region_one_disk_name
  project = var.project_id
  type    = var.disk_type
  zone    = var.zone_one
  size    = var.disk_size
}

resource "google_compute_instance" "region_one" {
  name         = local.region_one_compute_name
  project      = var.project_id
```

```
  machine_type = var.machine_type
  zone         = var.zone_one

  network_interface {
    subnetwork = var.subnet_one_link
  }

  attached_disk {
    source = google_compute_disk.region_one.self_link
  }
}
```

> **NOTE**
>
> This series of steps assumes that there isn't ongoing work in the main branch that will interfere with this reorganization. This may be a terrible assumption, in which case, you'll likely want to defer removing anything from the root TF project until just before "Step 10" and you're ready to confirm the new construction.

Step 5

Assuming you want to avoid mixing `resource{}` and `module{}` blocks in your root module, you'll next extract the remaining resources into a local module. This follows much the same steps as in "Step 4", except that you're not moving the configuration into a new repository but into a subdirectory, such as *./modules/main/*. Remember to pass through variables to the local module, reflect necessary outputs within the module, and configure the `module{}` block in the root TF project. Example 11-4 shows the result of this change.

Example 11-4. Extracting remaining resources into a local module

```
#-------------------------------------------------------------------
-----
# Before - Single root module containing all resources
#-------------------------------------------------------------------
-----
#
////////////////////////////////////////////////////////////////////
////////
# previous-root-module/main.tf (starting after compute resources)
#
```

```
////////////////////////////////////////////////////////////////////
////////

  ...

  attached_disk {
    source = google_compute_disk.region_two.self_link
  }
}

resource "google_storage_bucket" "default" {
  name     = local.bucket_name
  project  = var.project_id
  location = local.location
}

resource "google_storage_bucket_iam_member" "bucket_admin" {
  bucket = google_storage_bucket.default.name
  role   = "roles/storage.admin"
  member = local.developer_group
}

resource "google_project_iam_member" "compute_admin" {
  project = var.project_id
  role    = "roles/compute.admin"
  member  = local.developer_group
}

#-----------------------------------------------------------------
-----
# After - Root module contains remote modules
#-----------------------------------------------------------------
-----
#
////////////////////////////////////////////////////////////////////
////////
# new-root-module/main.tf - Excerpt (starting after compute module)
#
////////////////////////////////////////////////////////////////////
////////

  ...
  subnet_one_link = module.network.subnet_one_link
  subnet_two_link = module.network.subnet_two_link
}

module "local_main" {
  source = "./modules/main"
```

```
  developer_group = local.developer_group
  project_id      = var.project_id
  location        = local.location
  bucket_name     = local.bucket_name
}

#
*****************************************************************
********

#
////////////////////////////////////////////////////////////////
////////
# new-root-module/modules/main/main.tf
#
////////////////////////////////////////////////////////////////
////////

resource "google_storage_bucket" "default" {
  name     = var.bucket_name
  project  = var.project_id
  location = var.location
}

resource "google_storage_bucket_iam_member" "bucket_admin" {
  bucket = google_storage_bucket.default.name
  role   = "roles/storage.admin"
  member = var.developer_group
}

resource "google_project_iam_member" "compute_admin" {
  project = var.project_id
  role    = "roles/compute.admin"
  member  = var.developer_group
}
```

Step 6

If you have validation steps as part of your release procedure, you'll need to spend some time devising and implementing a way to ensure the steps remain covered in the root TF project, or by extracting the relevant tests, for example, into their respective module repositories.

Step 7

Because the remote modules will be packaged into a module and stored in a registry, you should ensure it includes only necessary information. As such, you should include a comprehensive *.terraformignore* file to restrict specific files or certain file extensions from being included. More information on the *.terraformignore* file can be found in ".terraformignore".

Step 8

At this point, you're ready to package up your module and store it in the module registry. This is different for different module registries. Ideally, you can do this from a wildcard job instead of your local workstation.

Step 9

The previous Step 8 assumes that you're performing an initial push to a module's registry as a one-off activity. However, this is not an especially sustainable procedure, and you'll want to construct release automation around the module to ensure consistency and version parity between the module repository and the registry.

Step 10

Now that you have the remote module, you need to validate that the new construction will work. This can be done by running a `tf plan` command, for example, in a wildcard job, against the feature branch. Because you haven't modified the TF state yet, you'll almost certainly see many changes reported. In particular, you should see an equal number of resources reported to be created as reported to be destroyed, with exactly matching create-destroy pairs. Although it may require some rearrangement of the reported changes, it will help you identify the exact TF state element addresses you'll need in the next step. It can also highlight any missing components before you modify the state. Finally, this allows you to validate that the remote module can indeed be retrieved from the registry. An excerpt of this behavior is in Example 11-5.

Example 11-5. An excerpt of a `tf plan` *that shows the same resources destroyed and recreated*

```
...
  # module.network.google_compute_network.default will be created
  + resource "google_compute_network" "default" {
      + id            = (known after apply)
      + name          = "tas-11-vpc"
      + project       = "tas-11"
      + routing_mode  = (known after apply)
      + self_link     = (known after apply)
    }

  # google_compute_network.default will be destroyed
  # (because google_compute_network.default is not in
configuration)
  - resource "google_compute_network" "default" {
      - id            = "projects/tas-11/global/networks/tas-11-
vpc" -> null
      - name          = "tas-11-vpc" -> null
      - project       = "tas-11" -> null
      - routing_mode  = "REGIONAL" -> null
      - self_link     = "https://.../networks/tas-11-vpc" -> null
    }

  # google_compute_subnetwork.region_one will be destroyed
  # (because google_compute_subnetwork.region_one is not in
configuration)
  - resource "google_compute_subnetwork" "region_one" {
      - gateway_address = "10.2.0.1" -> null
      - id              = ".../subnetworks/tas-11-us-central1-snet"
-> null
      - ip_cidr_range   = "10.2.0.0/16" -> null
      - name            = "tas-11-us-central1-snet" -> null
      - network         = ".../networks/tas-11-vpc" -> null
      - project         = "tas-11" -> null
      - purpose         = "PRIVATE" -> null
      - region          = "us-central1" -> null
      - self_link       = ".../subnetworks/tas-11-us-central1-snet"
-> null
    }

  # module.network.google_compute_subnetwork.region_one will be
created
  + resource "google_compute_subnetwork" "region_one" {
      + gateway_address = (known after apply)
      + id              = (known after apply)
      + ip_cidr_range   = "10.2.0.0/16"
      + name            = "tas-11-us-central1-snet"
      + network         = "tas-11-vpc"
      + project         = "tas-11"
```

```
      + purpose          = (known after apply)
      + region           = "us-central1"
      + self_link        = (known after apply)
    }

    ...

Plan: 3 to add, 0 to change, 3 to destroy.
...
```

Step 11

Using the TF state element addresses gathered in the previous step, you must now move the resources to their new location using `tf state mv`. Encode this into a single shell script, as depicted in Example 11-6; to at least avoid a typo or miskey, however, this script should be run in a wildcard job.

Example 11-6. Shell script that moves TF state elements to the new address

```
#!/bin/bash

tf state mv google_storage_bucket.default \
  local_main.google_storage_bucket.default
tf state mv google_storage_bucket_iam_member.bucket_admin \
  local_main.google_storage_bucket_iam_member.bucket_admin
tf state mv google_project_iam_member.compute_admin \
  local_main.google_project_iam_member.compute_admin
tf state mv google_compute_disk.region_one \
  module.compute.google_compute_disk.region_one
tf state mv google_compute_instance.region_one \
  module.compute.google_compute_instance.region_one
tf state mv google_compute_disk.region_two \
  module.compute.google_compute_disk.region_two
tf state mv google_compute_instance.region_two \
  module.compute.google_compute_instance.region_two
tf state mv google_compute_network.default \
  module.network.google_compute_network.default
tf state mv google_compute_subnetwork.region_one \
  module.network.google_compute_subnetwork.region_one
tf state mv google_compute_subnetwork.region_two \
  module.network.google_compute_subnetwork.region_two
```

Step 12

You're now ready to complete the validation of the new construction of the root TF project. Again, using a wildcard job, you can run the `tf plan` against the feature branch, this time expecting no reported plan to modify (or recreate) any resources, as shown in Example 11-7. You should also consider piloting any relevant validation activities at this step.

Example 11-7. Confirming the new construction of the TF project reports no planned changes

```
No changes. Your infrastructure matches the configuration.

TF has compared your real infrastructure against your configuration
and found no
differences, so no changes are needed.

Apply complete! Resources: 0 added, 0 changed, 0 destroyed.
```

Step 13

At this point, you have completed the majority of the work, and you can create a merge request to get these changes onto the main branch. This will trigger your release pipelines, assuming they're configured to watch and run against this main branch. Assuming this reports no changes in the planning phase and all validations pass, you can consider the initiative complete.

Splitting a TF State into Two

The next scenario is when you want to split a TF state and its project into two or more TF projects. For the purpose of this walkthrough, you'll consider the case where you're extracting a group of resources into a new repository, similar to the prior scenario that created a nested, remote module, except this time you'll use this new repository as a separate, distinct TF project, as shown in Figure 11-5. Depending on how you identify groupings of resources, you may instead decide to abandon the existing repository altogether and create all new TF projects in new repositories, however, this adds some minor complexity that can be ignored here for brevity.

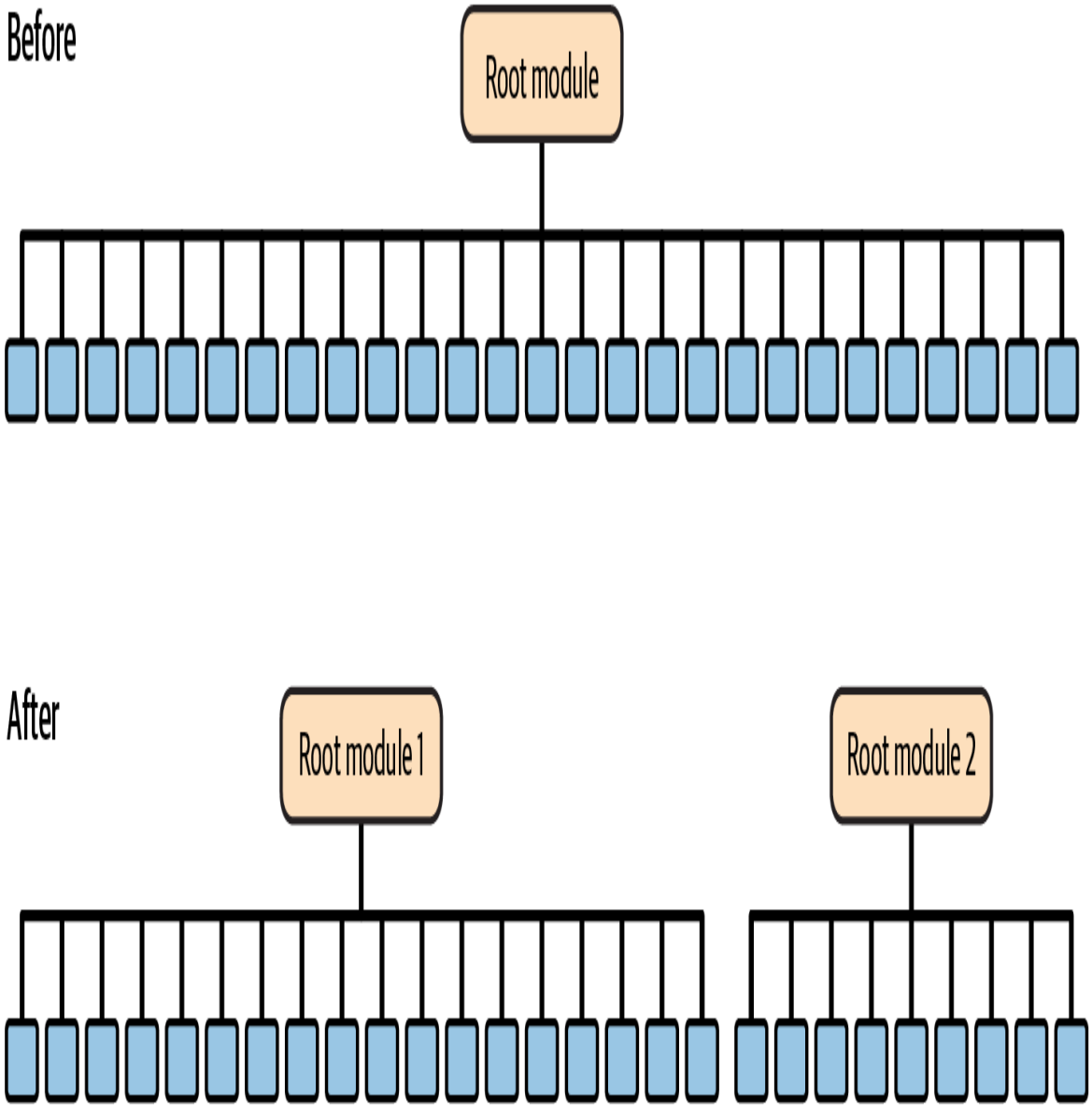


Figure 11-5. Before and after extracting a portion of a TF root project into a distinct TF root

Step 1

The first step in this scenario, similar to the last one, is to determine which components will be extracted. Ideally, distinct domains can be identified that will map to our new TF projects, as depicted in Figure 11-6.

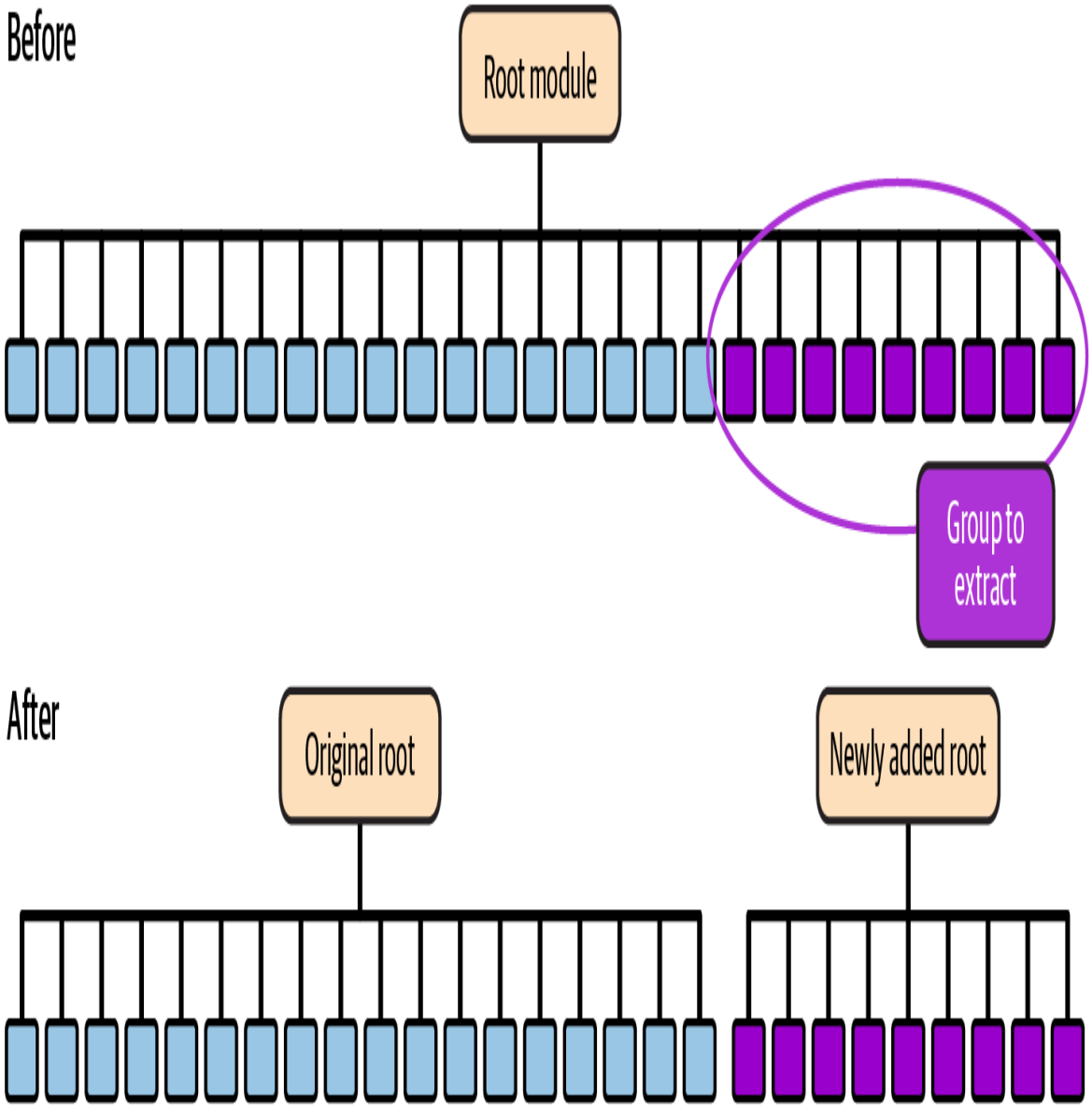


Figure 11-6. Identifying the portion of the TF state that can be extracted into its own TF project

Step 2

Just as in the previous scenario, you'll next create a feature branch in the existing TF project's repository to track this work.

Step 3

You'll need to create a new repository for the new TF project. Because this will be a partial replacement for the existing TF project, you can even shadow clone its repository as shown in Example 11-8, as long as the new

repository is created completely empty, for example, with no *README.md* or *.gitignore* files included.

Example 11-8. Shadow cloning the existing TF project to populate a new, empty repository

```
repos % git clone git@git-registry.com:tas-11/old-repo-name.git
new-name
Cloning into 'new-name'...
remote: Enumerating objects: 241, done.
remote: Counting objects: 100% (241/241), done.
remote: Compressing objects: 100% (154/154), done.
remote: Total 241 (delta 114), reused 201 (delta 77), pack-reused 0
(from 0)
Receiving objects: 100% (241/241), 411.32 KiB | 957.00 KiB/s, done.
Resolving deltas: 100% (114/114), done.
repos % cd new-name
new-name % git remote set-url origin git@git-registry.com:tas-
11/new-name.git
new-name % git push origin main
```

> **NOTE**
>
> The previous step will not only copy the code, it will bring along the entire Git history to the new repository, which may not be your preference or may violate some policy. In this case, the same operation can be taken, except with an additional command that removes the *.git/* subdirectory before pushing to the new remote repository.

Step 4

Now that you have the respective repositories, you can remove all the extracted resources from the original repository. The remaining resources in the original repository can be used as a guide, removing exactly these from the new repository. Likewise, you'll want to remove no-longer-relevant validation steps from the respective repositories.

Step 5

Assuming you operate your existing TF project with an automation platform, you'll need to duplicate this workflow for the new TF project. This may require a combination of both steps in the platform itself to

establish the workflow and the workflow's step configuration rendered in YAML or the platform's domain-specific configuration language.

Step 6

Ensure you can initialize the new TF project using `tf init` in a wildcard job.

Step 7

Once the new TF project's TF state is initialized, you can determine the correct TF state element addresses by first running a `tf plan`, which will report the additions it would make if it's applied. This too would be from within a wildcard job. This would be similar to the output in the previous Example 11-5.

Step 8

Now that you have the resources to include, you can import the relevant resources into this state. An excerpt of this procedure is in Example 11-9.

Example 11-9. Shell script that imports resources into the new TF state

```
#!/bin/bash
tf import google_compute_network.default \
  projects/tas-11/global/networks/tas-11-vpc
tf import google_compute_subnetwork.region_one \
  projects/tas-11/regions/us-central1/subnetworks/tas-11-us-
central1-snet
tf import google_compute_subnetwork.region_two \
  projects/tas-11/regions/us-west1/subnetworks/tas-11-us-west1-snet
```

Step 9

Once the resources have been imported into the new TF state, you can again run a `tf plan` against the new TF project in a wildcard job. When it reports no changes, it confirms that the import was a success, just like in Example 11-7.

Step 10

Once again in a wildcard job, you can run a `tf plan` against the original TF project to show what it plans to remove.

Step 11

Again, you'll use the result of the previous step to determine what TF state entity addresses to target in a series of `tf state rm` commands, as shown in Example 11-10.

Example 11-10. Shell script that removes resources from the existing TF state

```
#!/bin/bash
tf state rm google_compute_network.default
tf state rm google_compute_subnetwork.region_one
tf state rm google_compute_subnetwork.region_two
```

Step 12

Once the resources have been removed from the existing TF state, you can again run a `tf plan` against the existing TF project in a wildcard job referencing the feature branch. When it reports no changes, it confirms that the state removal was a success.

Step 13

Once again, you have completed the bulk of the work, and you can open a merge request into the main branches of each repository. Once merged, it should trigger your release pipelines. The planning phase should report no changes, and all validations should pass at this point. This again completes the initiative.

Making a Reusable TF Project From a Static One

Suppose you have configured multiple, distinct environments — say *dev*, *test*, and *prod* — that are used to support your SDLC, and you're currently managing them in a single TF project. Now, you want to restructure the project to split the TF state into separate, dedicated ones for each environment, as depicted in Figure 11-7.

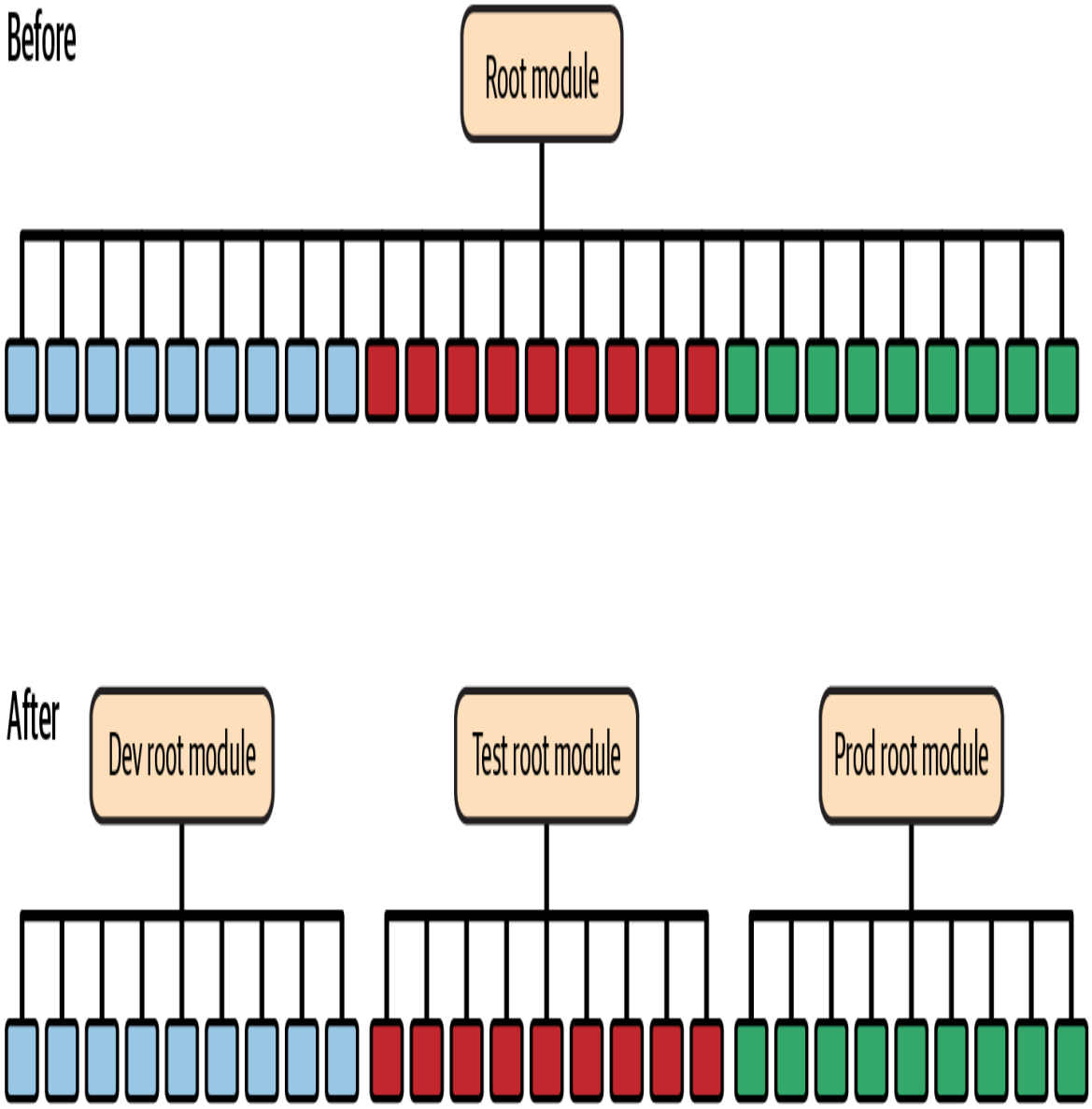


Figure 11-7. Before and after separating a TF state into three, one for each environment

Step 1

First, you'll create a feature branch in the TF project's repository to track this work.

Optional Step 1.1

Next, you may choose to reorganize the project in preparation for refactors. This could mean separating the *.tf* files containing `variable{}`, `resource{}`, and `output{}` blocks into environment-dedicated files —

and a file or set of files containing any resources shared across all environments — and similarly segmenting your *.tfvars* files. If you're currently using a single *terraform.tfvars* file, you should consider introducing the environments in *.auto.tfvars* instead. An example excerpt of this procedure is depicted in Example 11-11.

Example 11-11. Segmenting configuration files by environment within the TF project

```
root-module/
├── providers.tf
├── dev.tf
├── test.tf
├── prod.tf
├── variables.tf
├── outputs.tf
├── dev.auto.tfvars
├── test.auto.tfvars
├── prod.auto.tfvars
└── shared.auto.tfvars
```

Optional Step 1.2

Once your configuration is reorganized by environment, you should confirm that this reports no change in a planning phase by running `tf plan` in a wildcard job.

Step 2

Next, you can create the *dev*, *test*, and *prod* environment subdirectories and move the environment-specific *.auto.tfvars* files into these subdirectories, keeping the shared *.tfvars* in the root of the project. Include dedicated *backend.tf* or *state.tf* files in each environment subdirectory to configure their dedicated backend (see Example 11-12).

Example 11-12. Filesystem after creating environment-specific configurations

```
root-module/
├── main.tf
├── variables.tf
├── outputs.tf
├── shared.auto.tfvars
```

```
└── envs/
    ├── dev/
    │   ├── providers.tf
    │   └── dev.auto.tfvars
    ├── test/
    │   ├── providers.tf
    │   └── test.auto.tfvars
    └── prod/
        ├── providers.tf
        └── prod.auto.tfvars
```

Step 3

Now you'll need to refactor the original root module to genericize the resources in each of the environment-specific *.tf* files. Make sure you're introducing conditional expressions, for example, using the `count` meta-argument, for any resources that do not appear in all environments.

Step 4

Next, you can create symbolic links (symlinks) to link the environment-dedicated subdirectories to the repository root — which still contains the now genericized TF resource configuration — and push this to your feature branch, as shown in Example 11-13.

Example 11-13. Creating symlinks for the environment-specific subdirectories to the repository root

```
#!/bin/bash
ln -s "$PWD" "$PWD/envs/dev"
ln -s "$PWD" "$PWD/envs/test"
ln -s "$PWD" "$PWD/envs/prod"
```

> **NOTE**
>
> At this point, you may need to split up validation. However, it's possible that the same validation steps will work as before with little or no changes.

Step 5

Next, you'll need to initialize the backends for each of the environments, as depicted in Example 11-14. This should be done in a wildcard job.

Example 11-14. Initializing the backends for each of the environment's dedicated TF state

```
#!/bin/bash
tf -chdir=envs/dev init -backend-config="prefix=path/to/dev-
backend"
tf -chdir=envs/test init -backend-config="prefix=path/to/test-
backend"
tf -chdir=envs/prod init -backend-config="prefix=path/to/prod-
backend"
```

Step 6

You're now ready to import the existing resources for each state. However, first you must know what you're importing. Instead of running `tf plan` to see what would get created — and trying to match that up with the respective IDs — you can use `tf show` with the `json` flag, and pipe its output to the `jq` command, as depicted in Example 11-15.

Example 11-15. Output from running `tf show` *command with a filter for address and ID (excerpt)*

```
root-module % tf show -json | jq -r '
  .values.root_module.resources[]
  | select(.values.id != null)
  | "\(.address) \(.values.id)"
'
google_compute_network.dev projects/tas-11-dev/global/networks/tas-
11-dev-vpc
google_compute_subnetwork.dev_region_one projects/tas-11-
dev/regions/us-central1/sub
networks/tas-11-dev-us-central1-snet
google_compute_subnetwork.dev_region_two projects/tas-11-
dev/regions/us-west1/subnet
works/tas-11-dev-us-west1-snet
google_compute_instance.dev_region_one projects/tas-11-
dev/zones/us-central1-a/insta
nces/tas-11-dev-us-central1-comp
google_compute_instance.dev_region_two projects/tas-11-
dev/zones/us-west1-b/instance
s/tas-11-dev-us-west1-comp
google_compute_network.test projects/tas-11-
```

```
test/global/networks/tas-11-test-vpc
google_compute_subnetwork.test_region_one projects/tas-11-
test/regions/us-central1/s
ubnetworks/tas-11-test-us-central1-snet
google_compute_subnetwork.test_region_two projects/tas-11-
test/regions/us-west1/subn
etworks/tas-11-test-us-west1-snet
google_compute_instance.test_region_one projects/tas-11-
test/zones/us-central1-a/ins
tances/tas-11-test-us-central1-comp
google_compute_instance.test_region_two projects/tas-11-
test/zones/us-west1-b/instan
ces/tas-11-test-us-west1-comp
google_compute_network.prod projects/tas-11-
prod/global/networks/tas-11-prod-vpc
google_compute_subnetwork.prod_region_one projects/tas-11-
prod/regions/us-central1/s
ubnetworks/tas-11-prod-us-central1-snet
google_compute_subnetwork.prod_region_two projects/tas-11-
prod/regions/us-west1/subn
etworks/tas-11-prod-us-west1-snet
google_compute_instance.prod_region_one projects/tas-11-
prod/zones/us-central1-a/ins
tances/tas-11-prod-us-central1-comp
google_compute_instance.prod_region_two projects/tas-11-
prod/zones/us-west1-b/instan
ces/tas-11-prod-us-west1-comp
```

Step 7

Now you can run the `tf import` commands with these outputs in a wildcard job against each environment. Remember to split the resources among the environments, as shown in Example 11-16 (note the change in address, genericizing from environment-specific to `default` or simply `region_one`, `region_two`, and so on; the environment distinction moves to the environment-specific states).

Example 11-16. Importing the resources printed in the previous command, per environment

```
# import-dev.sh
tf -chdir=env/dev import google_compute_network.default \
  projects/tas-11-dev/global/networks/tas-11-dev-vpc
tf -chdir=env/dev import google_compute_subnetwork.region_one \
  projects/tas-11-dev/regions/us-central1/subnetworks/tas-11-dev-
```

```
us-central1-snet
tf -chdir=env/dev import google_compute_subnetwork.region_two \
  projects/tas-11-dev/regions/us-west1/subnetworks/tas-11-dev-us-
west1-snet
tf -chdir=env/dev import google_compute_instance.region_one \
  projects/tas-11-dev/zones/us-central1-a/instances/tas-11-dev-us-
central1-comp
tf -chdir=env/dev import google_compute_instance.region_two \
  projects/tas-11-dev/zones/us-west1-b/instances/tas-11-dev-us-
west1-comp

# import-test.sh
tf -chdir=env/test import google_compute_network.default \
  projects/tas-11-test/global/networks/tas-11-test-vpc
tf -chdir=env/test import google_compute_subnetwork.region_one \
  projects/tas-11-test/regions/us-central1/subnetworks/tas-11-test-
us-central1-snet
tf -chdir=env/test import google_compute_subnetwork.region_two \
  projects/tas-11-test/regions/us-west1/subnetworks/tas-11-test-us-
west1-snet
tf -chdir=env/test import google_compute_instance.region_one \
  projects/tas-11-test/zones/us-central1-a/instances/tas-11-test-
us-central1-comp
tf -chdir=env/test import google_compute_instance.region_two \
  projects/tas-11-test/zones/us-west1-b/instances/tas-11-test-us-
west1-comp

# import-prod.sh
tf -chdir=env/prod import google_compute_network.default \
  projects/tas-11-prod/global/networks/tas-11-prod-vpc
tf -chdir=env/prod import google_compute_subnetwork.region_one \
  projects/tas-11-prod/regions/us-central1/subnetworks/tas-11-prod-
us-central1-snet
tf -chdir=env/prod import google_compute_subnetwork.region_two \
  projects/tas-11-prod/regions/us-west1/subnetworks/tas-11-prod-us-
west1-snet
tf -chdir=env/prod import google_compute_instance.region_one \
  projects/tas-11-prod/zones/us-central1-a/instances/tas-11-prod-
us-central1-comp
tf -chdir=env/prod import google_compute_instance.region_two \
  projects/tas-11-prod/zones/us-west1-b/instances/tas-11-prod-us-
west1-comp
```

Step 8

Next, you should confirm the imports worked in each environment by running a `tf plan` in each, which should show no changes, as seen in

Example 11-7.

Step 9

At this point, you'll need to split your automation pilot into environment-dedicated pipelines. These should be associated in a sequence such that a successful *dev* workflow enables the ability to run a test workflow, which in turn enables a *prod* workflow. The *test* and *prod* workflows can even include approval gates to prevent an inappropriate or premature release to the environment.

Step 10

Yet again, you have completed the majority of the work, and you can open a merge request into the main branches of each repository. Once merged, it should kick off the series of environment release workflows. The planning phase for each environment should report no changes, and again all validations should pass.

Step 11

As a final step, clean up the original TF state that combines all the environments into a single state. Because you're effectively abandoning the old configuration, you probably can simply destroy the TF state object in your backend location, as shown in Example 11-17, rather than cleaning it up using `tf state` commands. This should likely be done in a wildcard job. Doing so completes this initiative.

Example 11-17. Forcibly deleting the TF state object in your previous remote backend

```
~ % gcloud storage rm gs://terraform-at-scale/chapter-
11/default.tfstate
Removing objects:
Removing gs://terraform-at-scale/chapter-11/default.tfstate...
  Completed 1/1
```

TIP

Instead of going through and importing all the resources, an alternative procedure to Steps 5 through 7 might be as follows:

- Dump the original, multienvironment state with `terraform state pull` and store in a local file, *terraform.tfstate*.
- Copy this *terraform.tfstate* file into each environment subdirectory.
- Initialize each environment.
- Create the *backend.tf* or *state.tf* file that sets up the backend.
- Initialize the environments again, which will prompt you about whether you want to move the backend to the remote location (which you would answer *yes*).

Importing Untracked Resources

The final scenario we'll consider is what happens when you discover resources that are not managed by TF. Perhaps your infrastructure observability apparatus highlighted these in a dashboard for missing a "managed by TF" tag, or perhaps it generated an alert because the resources weren't introduced by the proper service identity. Maybe you perform an audit across your CSP, comparing the reported resources against the results of a series of `tf show` commands across all your TF projects. Regardless of how you discovered them, you must take steps to ensure they're properly managed, which generally means that they're included in a relevant TF project — either by expanding input values or through new `resource{}` definition blocks — and import these resources into the proper place in the respective TF state. A visual representation of this is in Figure 11-8.

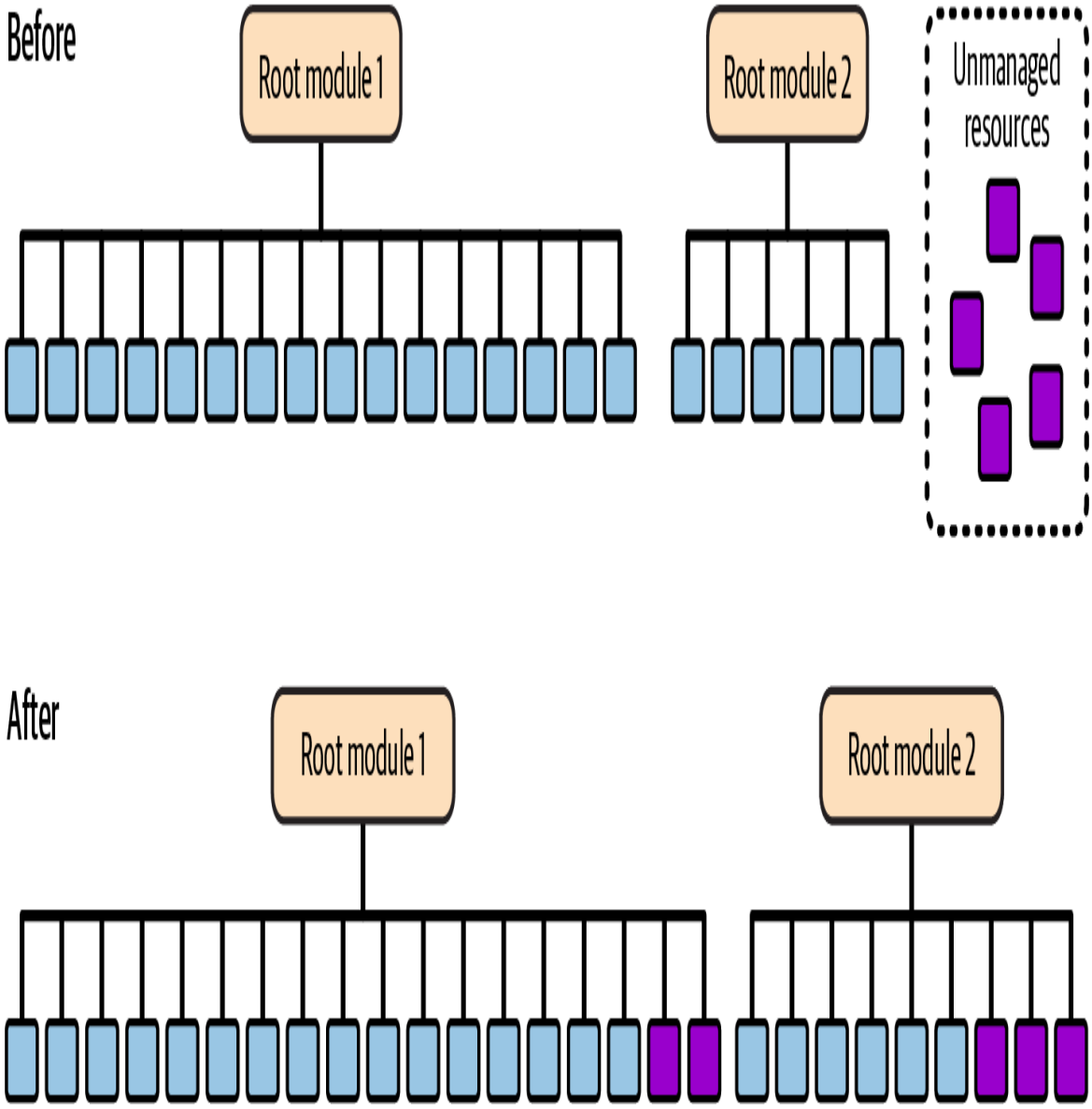


Figure 11-8. Before and after importing untracked resources into respective TF states

Step 1

First, you need to enumerate all the untracked resources that you'll be working with. Although you may have a complete list of untracked resources, it's likely that at least one of these is either an artifact from some prior initiative (and can be deleted), shouldn't exist at all (warranting a separate investigation), or is intentionally ungoverned. This may be considered a preliminary step or an assumed property of the problem statement. It's included here for completeness and because you wouldn't

want to spend time encoding a resource that shouldn't be preserved or managed by TF.

> **NOTE**
>
> The previous scenario notwithstanding, in which a resource is deliberately unmanaged by TF, it doesn't address the merits involved. Whether or not the resource's origination was intentional, there's still the question of whether it was appropriate to build in the first place. Even if you determine it was indeed appropriate, it should likely have certain indicators, such as its own tag formulation, that will signal to the observability or audit controls that the resource should not be considered problematic.

Step 2

Once you have the complete list of resources that you now want to manage with TF, you'll need to analyze which existing TF projects they should be filtered into — or if you need to create net new projects in the case that resources don't clearly belong to an existing project, and including them in one arbitrarily might prove to be awkward in terms of lifecycle or ownership.

Step 3

Assuming you have determined they all *can* be included in existing repositories, the next step will be to create feature branches in all of them. Most likely, you can use a common name across all repositories, as shown in Example 11-18, rather than needing a unique name in each.

Example 11-18. Creating a common feature branch name across all affected project repositories

```
project-1 % git checkout -b feature/untracked-resources
Switched to a new branch 'feature/untracked-resources'
project-1 % cd ../project-2
project-2 % git checkout -b feature/untracked-resources
Switched to a new branch 'feature/untracked-resources'
project-2 % cd ../project-3
project-3 % git checkout -b feature/untracked-resources
Switched to a new branch 'feature/untracked-resources'
```

Step 4

Now you're ready to make the necessary edits for supporting the new resources. Ideally, the resources can simply be included without modifying existing `resource{}` blocks or adding new ones, instead only expanding the variable inputs, for example in the *.tfvars* file(s). However, this isn't always the case — for example, you may need to add net new `resource{}` block definitions, supplement an existing definition with additional resource fields, or introduce a `for_each` meta-argument to turn a definition that supports a single resource into one that supports multiple such resources. You may also need to introduce new `variable{}` and `output{}` blocks to account for the new resources. Examples of such changes are rendered in Example 11-19.

Example 11-19. Example updates necessary to support newly managed resources (as `git diff`)

```
 resource "google_compute_subnetwork" "subnets" {
    for_each = var.subnets

    name          = each.key
    ip_cidr_range = each.value.ip_cidr_range
    region        = each.value.region
    network       = google_compute_network.default.id
+
+   dynamic "secondary_ip_range" {
+     for_each = each.value.secondaries
+
+     content {
+       range_name    = secondary_ip_range.key
+       ip_cidr_range = secondary_ip_range.value
+     }
+   }
  }

  variable "subnets" {
    description = "Subnets to configure for the VPC"
    type        = map(object({
      ip_cidr_range = string
      region        = string
+     secondaries   = map(string)
    }))
  }
```

Step 5

At this point, you're ready to import the resources into the various TF states. Using the same two TF projects as in Step 4, you can first identify what you need to import by reviewing the changes that get planned using `tf plan` — in a separate wildcard job for each TF project in question — before actually running the import. Then — in another series of wildcard jobs — you can import the resources, again ideally using a shell script (one for each project) rather than relying on issuing commands by hand, one by one. After the series of imports, you can confirm they were successful by again running a `tf plan` — yet again in a corresponding series of wildcard jobs. See Examples 11-9 and 11-16 for illustrations of the imports. Existing validation should not change; however, you may need to expand, for example, your tests to account for any newly supported resource types or changes that take a singular-resource definition to one that manages multiple resources, such as by adding the `for_each` meta-argument.

Step 6

By now, you should know the drill: you have completed most of the work, and you're ready to open a series of merge requests into the main branches of the respective repositories. Once these get merged in, it will kick off the release workflows for each TF project. All the planning phases for each project and each environment should once again report no changes. As mentioned in the previous step, all validations should also pass without issue. Assuming this is the case, you have reached the conclusion of this endeavor.

Miscellaneous TF Features

Although you've reviewed several of the tools and features available in TF throughout this text, there are many more that haven't been directly covered. Some of these are presented here, with commentary on how they might relate to developing TF at enterprise scale. This is not meant to be a

comprehensive list either, but to better round out the features you might utilize or find useful.

Using Local Workspaces

Local workspaces are a way to reuse the same *codebase* — that is, the same collection of HCL-encoded TF files generally designated by a filesystem directory — with distinct backends. They allow operators to leverage separate input variable value assignments to manage multiple, distinct copies of the encoded infrastructure. This can support, for example, a TF project that configures multiple environments intended in an SDLC workflow or one that can be applied in multiple regions in a cookie-cutter fashion. It can even support a *white glove* infrastructure resell pattern where you act as an intermediary managed service provider (MSP) and configure and manage all the cloud infrastructure for a third-party client, where each client has a dedicated workspace. Workspaces are powerful tools for operating TF from an operator's workstation. However, they have severely diminishing returns and drawbacks in advanced TF ecosystems, such as when using automation platforms, and they require a high degree of discipline and care when operating manually.

> **NOTE**
>
> TFE and HCP Terraform both share a (slightly different) concept of workspaces that is a core component to both platforms. Although it has several properties in common with a local workspace, the TFE/HCP workspace is a fixed property instead of something that can be switched between, and it acts as a complete execution environment. The commentary in this section is not relevant to these platform constructs and is purely related to local workspaces.

Workspaces (the local ones) function as a way to manage multiple TF states using the same TF configuration and allows selecting between them. In multi-environment TF projects, in which each environment would otherwise require a distinct backend, workspaces allow you to initialize a backend once, rather than having to do it every time you want to switch

between states. Although this may seem trivial, remember that `tf init` will fetch provider plug-ins, remote modules, and other external data, meaning the larger and wider-reaching the TF project, the longer an initialization takes. Even for small-to-moderate-sized TF states, this is a nontrivial time savings.

> **NOTE**
>
> Once a workspace is created with a remote backend, all members of the team can leverage it immediately. They simply need to use `tf workspace select` to do so.

Local workspaces may work fine for teams that run TF operations from their local workstations. However, as your TF ecosystem grows, it becomes more and more of a priority to establish governance and automation to ensure the operations are executed without the possibility of human error — miskeys, skipped steps, and so on. By the time you're at enterprise scale, you really should not be executing TF operations from your local workstation anyway — except in very, very few extremely well-documented cases complete with business justification and sponsorship. Even sandboxed or ephemeral environments should be managed in a consistent way. Instead, you're managing your ecosystem through an automation platform that exposes the various TF operational levers in a highly controlled way. This severely diminishes the usefulness of local workspaces, especially if your organization decides to go as far as to restrict all direct, personal access to the resources or services that act as the TF state backend repositories.

When you create a new workspace using the `tf workspace new` command, it will create an empty state, by default. This means any resources that have already been provisioned are unknown to the workspace, and running a `tf plan`, for example, will report that it will freshly create all the resources included in the TF state. Certainly, if you have operated TF locally for some time using (perhaps unwittingly) the default workspace and decide to use a named local workspace, you'll want

to transfer the TF state to this new workspace. This is done using the `-state` argument with the `tf workspace new` command targeting the default workspace state configuration, typically the `terraform.tfstate` file, for example: `tf workspace new -state=terraform.tfstate dev`. If you've already created the workspace and want to transfer the TF state to the workspace, you can pull down the remote state and push it to the workspace, as shown in Example 11-20.

Example 11-20. Pulling down a remote state and pushing it to a dev *workspace*

```
terraform workspace select default
terraform state pull > default_backup.tfstate
terraform workspace select dev
terraform state push default_backup.tfstate
```

A somewhat simpler method often suggested is to use distinct subdirectories for each desired TF state and to symlink these directories to a root directory containing the root TF module configuration. This way, each of the state subdirectories can manage a distinct backend. If you so choose, you may initialize these backends from the root using the `-chdir` and `-backend-config` arguments, without needing to create any environment-specific files or having to switch between directories, as shown in Example 11-21.

Example 11-21. Using symlink and operating distinct backends from the root directory

```
# create the directories
mkdir -p envs/prod
mkdir -p envs/dev

# create the symlink
ln -s "$PWD" "$PWD/envs/prod"
ln -s "$PWD" "$PWD/envs/dev"

# initialize the backends
tf -chdir=envs/prod init -backend-config="prefix=path/to/prod-
backend"
tf -chdir=envs/dev init -backend-config="prefix=path/to/dev-
backend"
```

```
# plan each configuration
tf -chdir=envs/prod -var-file="../terraform.tfvars" plan
tf -chdir=envs/dev -var-file="../terraform.tfvars" plan
```

> **NOTE**
>
> Other tools, such as Terragrunt, can also manage multiple TF states within a single TF project, simplifying complex management.

.terraformignore

Similar to the *.gitignore* file, TF has a *.terraformignore* file. It expresses which files are excluded from module packaging. It really only applies when you're packaging a module for storage in a remote registry. The typical files you want to include in the *.terraformignore* are the local Terraform artifacts, any sensitive files like certificate keys, code editor-specific files, logs, and any existing test files. An example stock *.terraformignore* file is shown in Example 11-22.

Example 11-22. Stock .terraformignore *file*

```
# Local Terraform artifacts
.terraform/
*.tfstate
*.tfstate.*

# Sensitive data
*.pem
*.key

# Editor files
.DS_Store
.vscode/
.idea/

# Log files
*.log

# Test files
*.tftest.hcl
```

Using tf console

Sometimes it's helpful to try out certain logical expressions before you encode them in your TF code. This can be accomplished with the `tf console` command, which starts an interactive Read-Eval-Print Loop (REPL) environment. With it, you can submit any valid TF expression and it will return the result. It doesn't modify your infrastructure, local files, or the TF state; however, it does allow you to inspect your project's state or to reference portions of it in your expressions, making it ideal for rapid validation of logical expressions with specific input values.

Using a Lock Timeout

The TF lock file prevents data corruption or race conditions during a planning phase such as when running `tf plan`. It's placed in the backend location, and if it is present, a `tf plan` command will fail. By default, when TF fails to acquire a state lock, it will fail immediately. This is because the default retry timeout is zero seconds. You can override this using the `-lock-timeout` argument, which can be configured to set this timeout to a certain number of seconds or minutes. This can be especially useful in environments that might see rapid changes, such as those developed by several engineers. Rather than letting the `tf plan` operation fail immediately, it can wait for a previously started process to complete before it executes.

> **NOTE**
>
> In the likely case that you're leveraging an automation platform, you must make sure your `-lock-timeout` is shorter than the platform's configured (or default) job timeout. Otherwise, you may not cleanly exit or completely record the operation.

Using the -replace and -target Arguments

Two arguments that focus operations on individual resources are `-replace` and `-target`. If you've determined a resource is not behaving

properly, but a planning phase doesn't report a change, you can use the `-replace` argument to indicate this to TF. This will cause TF to destroy and recreate the specified resource during the next `tf apply`.

Similarly, the `-target` argument is used to apply changes to individual resources — or to collections of resources identified by a common TF address prefix — while ignoring any other unapplied changes. It generates a plan that will include the targeted resources, as well as any resources these depend on. Generally speaking, the `-target` argument is an anti-pattern and should be avoided as much as possible. It should really only be used in rare, extreme cases where you cannot afford to wait for the entire TF project to be processed through the TF resource graph. Ideally, it would still be run from a controlled environment such as by leveraging a wildcard job. Finally, it should be immediately followed by a proper, full `tf apply` to keep from leaving a partial state.

Using the -parallelism Argument

By default, TF performs 10 concurrent operations when walking the resource graph. You may use the `-parallelism` argument to control this: setting it to an integer number tells TF to limit operations to that number at a time. Increasing this beyond 10 can speed up the overall running time and may be useful for very large TF projects and states. However, some CSPs have strict API rate limits. Exceeding this limit may lead to your requests being throttled and will likely cause TF to fail. Similarly, if your automation platform has somewhat limited CPU and memory resources, you may need to reduce parallelism to stay within these limits.

Summary

In this chapter, you reviewed several concepts and considerations from the previous chapters, now fully contextualized. Topics you covered include:

- Some guiding points and a review of common problems

- Better approaches to handle several typical scenarios
- A review of source control considerations
- Strategies for getting unstuck from ordinary situations
- Several end-to-end guides for addressing major evolutionary efforts
- Miscellaneous features of TF and how they can be leveraged

You now are well equipped to take on many of the challenges that seem to plague every team at some point or another. Not only can you identify challenges and technical quagmires before you encounter them, you can also extricate yourself (and your company) from them in case they appear anyway. You're ready to guide any TF ecosystem through whatever necessary evolution in TF project design arises.

In the final chapter, you'll wrap up with more examples of how to utilize the most effective features and techniques available in TF. You'll also encounter some philosophical reflections and food for thought to take along as you make your own journey with TF and IaC.

Chapter 12. Last Thoughts

TF at scale is a never-ending journey and not a destination. You'll never get to the end of it. The few best practices it claims are heuristic, at best, and won't work in every situation forever. The way things are done today may not be appropriate when your company wants to take advantage of new CSP features, develop a line of business that resells services to a customer, or move to a multi-CSP ecosystem. This is even more true when talking about nontechnical business decisions such as mergers, acquisitions, and volumetric contract decisions — the latter of which, volume-based decisions, although seemingly arbitrary, are both rather common and often good business sense because they can come with significant discounts and lead to considerable savings when done effectively.

Instead of attempting to reach a predefined success criteria, you're more often trying to reach a local equilibrium: a state of low friction, total coverage, and quick responses to unforeseen incidents, all without exceeding a forecasted budget. To this end, this final chapter will provide a review of hypothetical case studies for organizations at different stages and scales and will equip you with a few heuristics you can consider in your work as a TF practitioner. It will end with a reflection on more philosophical topics that may guide your perspective or approach when considering new challenges not covered in this book. It's included with the goal of covering any remaining gaps, equipping you with a holistic grasp on the IaC space.

Different Scales

This book is about building TF projects *at scale*. But what scale are we talking about? Different industries have different needs and requirements in terms of digital infrastructure. They are also all businesses, led by humans with unique combinations of beliefs, agendas, circumstances, and

expectations. Therefore, even among the largest enterprises in the world, there may be great variation in how cloud resources are architected and managed.

However, as Satya Nadella famously asserted, "every company is now a software company." Every sufficiently large organization — from globally recognized brands to a regional retailer — collects massive amounts of data, has entire data science teams running big data analysis and reviewing business intelligence (BI) dashboards, and requires a data intake and warehousing apparatus. They have websites–possibly multiple, separately hosted websites with different availability and access requirements. They have services. So many services. Services accessed through APIs from their websites and, oh yeah, probably a mobile app for both iOS and Android. Services opened to specific vendors or customers possibly over private channels. Services that are only supposed to be used within the internal network. Services that should never leave their local habitat. And these services probably also need data, but maybe not exactly the same data from before, so you need some more databases, and they have to have this data in this environment and not that data in those environments, but we do need some of that data in this one environment for that random reason we talked about.

The point is that every enterprise-scale company has many of the same needs in terms of capability. The difference is usually in the scale. Whereas the regional retailer may indeed have all these components, they may have a magnitude fewer than the global brand, perhaps several magnitudes fewer if the brand is a software product. The more components your organization relies on to market and deliver its products, and the more products and lines of business you manufacture, the more extensive and complex your organizational management structure is likely to be. As Conway has taught us, this will directly influence your infrastructure needs.

In this section, you'll consider several case studies of various industries and market scales with the purpose of acknowledging common requirements, recognizing where the differences lie, and understanding the extent to which your TF ecosystem may grow.

Regional Company

Consider a regional energy conglomerate composed of several business units (BUs), combined through earlier mergers and acquisitions, that each have a distinct customer base consisting of home owners, businesses, nongovernment organizations (NGOs), and government entities. Each BU operates more or less independently but will occupy the same cloud account for the purpose of standardizing digital considerations. Major office locations are spread throughout the region, roughly aligning to the original brand offices. The central corporate entity occupies one of these original campuses. Along with customer-facing products, they have several internal-facing products that must be accessible outside of their regular corporate networks, such as technicians in the field. They're in a highly regulated industry and must, for example, maintain strict compliance with safety regulations in addition to normal data governance concerns. Moreover, because so many other entities rely on them, they are highly valuable targets to bad actors seeking to economically or politically destabilize the region, meaning they require a robust security posture that must extend to their field equipment. They analyze data across a spectrum of sources, including their equipment, for efficiency and stability, customer usage patterns, quality of service in the field, safety considerations, customer satisfaction, risk mitigation, and cost efficiency.

Their customer-facing workloads primarily support the company's websites and any mobile apps. However, they also have several services and web portals that are accessible only by their internal personnel, such as technicians in the field. Some of their software engineers work in what have become esoteric software languages that run on specialized hardware and devices, but most customer-facing products use typical commodity software stacks, including JavaScript, Java, and Python. They leverage a mix of internally hosted platforms and SaaS accounts, run many heterogeneous databases, and collect processed data in big data and BI tools. They have a mix of access-controlled APIs and leverage (or experiment with) AI models, for example, for RAG. They have resolved to construct a multicloud infrastructure, whereby they have the ability to run their most

critical resources on two different CSPs. This has the benefit of being able to take advantage of favorable pricing and spare resource bidding, but is primarily to operate as an extra failsafe in the case of disaster: they run a multiregion environment in one CSP; if either region fails, the system gracefully transitions services to the already partially warm multi-region environment in the other CSP. The procedure for this is partially automated, and the execution is relatively simple and straightforward. They have some prescribed audit process, and they may have ad hoc audits from time to time.

Despite dozens upon dozens of distinct workloads, they have a relatively small software engineering department. Their engineering teams are split fairly evenly between the central corporate entity and the BU brands. They treat many of their nonsoftware colleagues more like end users and segment their services into a few tiers of sensitivity, with a security mechanism common within each tier for confirming authorized access. Among their corporate offices, only two or three host their dozen or so software teams. Each BU has at least one team dedicated to networking and network security. Because each BU's tech department operates independently from other BUs', it leads to a mix of operational methodologies, mostly aligning to a Scrum or SAFe-style format.

Previously, each BU hosted whatever software services they developed in an *on-premises* environment that they have migrated to the cloud. This gave the TF team the benefit of time to construct their cloud resources and architect it to their needs. They contracted people who have architected these types of ecosystems before, and although things have evolved over time, they're roughly the same TF projects that were initially constructed for the migration. They may have modernized many of their services and leveraged cutting-edge technology to develop additional features and services, so the TF projects themselves have seen many additions and deletions, but because there hasn't been an obvious mismatch or awkwardness in resources lifecycle within or across projects, there has never been a reason to make significant changes.

This likely leads to a relatively compact cloud environment, with a centralized corporate operations space, centralized public edge defenses, and a spoked SDLC environment for each BU. The TF for this organization may be managed by a single team. There is likely one or two core TF projects that set up the central operations services and public edge, and two or three TF projects per BU: one for the structural components that may include foundational networks and at least one for the SDLC environments. If their policies allow it, they may be using OpenTofu for their infrastructure management; otherwise, they're self-hosting TFE.

Enhancement or feature requests come to this TF team through a few different channels and from the various other engineering teams, both corporate and BUs. Unlike the wider software engineering department, they largely work in a Kanban fashion, with ad hoc planning sessions scheduled days to weeks before prioritizing large chunks of work. The team spends more time analyzing cost reduction mechanisms and enhancing the auditing process than they do exploring digital performance gains. The architecture and TF project relationship for this organization can be found in Figure 12-1.

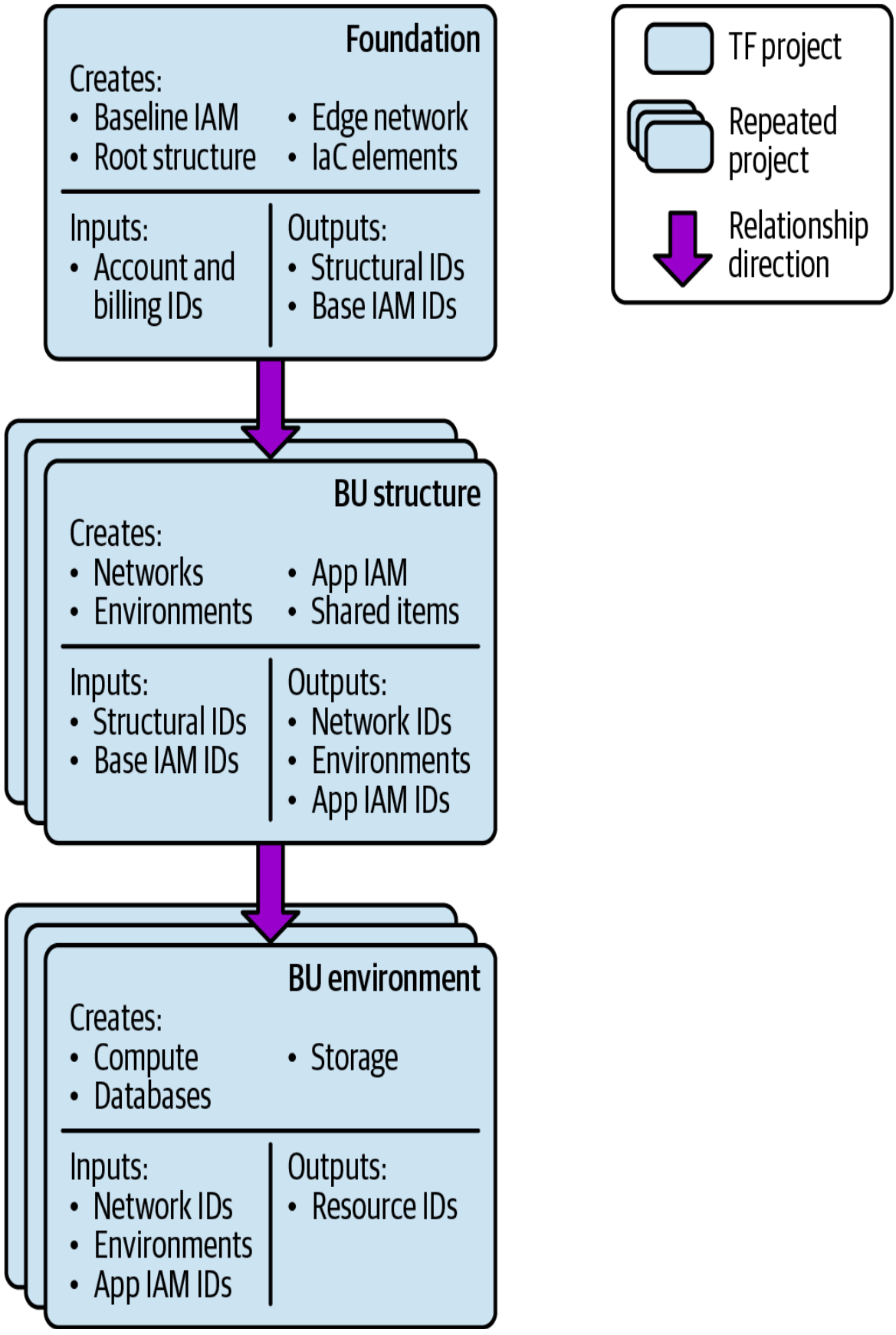

Foundation
Creates:
• Baseline IAM
• Root structure
• Edge network
• IaC elements
Inputs:
• Account and billing IDs
Outputs:
• Structural IDs
• Base IAM IDs
BU structure
Creates:
• Networks
• Environments
• App IAM
• Shared items
Inputs:
• Structural IDs
• Base IAM IDs
Outputs:
• Network IDs
• Environments
• App IAM IDs
BU environment
Creates:
• Compute
• Databases
• Storage
Inputs:
• Network IDs
• Environments
• App IAM IDs
Outputs:
• Resource IDs
TF project
Repeated project
Relationship direction

Figure 12-1. Architecture and TF project relationships for regional firm

SaaS Scale-up

Now consider a consumer financial technology (fintech) software start-up (such as a sportsbook or prediction market platform) that is in the process of scaling up operations. So far, they have focused on a single economic area and specific market demographics. They want to expand to other areas and are determining how best to certify that their product will comply with statutory and industry regulations in those jurisdictions. They currently have two or three offices in or near recognized technology hub cities and a relatively small workforce that is growing rapidly. They are also working through the legal process necessary to open an office location in the new region they're targeting — which will naturally result in yet more hires and contractors. Because they are recent entrants operating in a highly regulated industry — with rapidly evolving laws requiring foresight, clear and documented legal interpretation, heavy scrutiny, and comprehensive auditability — they have both fostered a culture of discipline and self-regulation and prioritized efficient regulatory compliance and provenance. They're constantly evaluating user engagement trends, adoption, churn, customer acquisition cost (CAC), and how effective their marketing channels are.

They have several customer-facing software products including mobile apps, a user web portal, various landing pages, internal administration portals, and various other services such as SMS push notifications. They also support private channels for exposing and consuming certain data services to and from other financial institutions. Their presentation layers are iOS, Android, and React for each respective interface, with a mix of languages for backend and data services including Python, Go, and Rust. They have made a conscious effort to prioritize self-hosting platforms, rather than use CSP-managed services, especially data services, with the notable exceptions including document storage, certain networking controls, content delivery network, and some key and secret management; however, some of these functions have traditionally leveraged managed

services, and there are ongoing efforts to transition managed data services to self-hosted solutions. They disallow fully OSS for any customer-facing components, instead resolving to license the IPS, SQL-like and NoSQL databases, key-value caches, OSes, and encryption software that they support and host; however, they have utilized certain open source platforms after a substantial de-risking exercise for policy enforcement, observability, and validation activities to mitigate licensing costs. They have a service agreement with a popular LLM provider, leveraging available data sovereignty controls; this may be used to support software development, but they disallow direct LLM updates to customer-facing software, instead allowing it for only internal workloads on a case-by-case basis and heavily documented with specific business justification and leadership sponsorship. They have a comprehensive policy rules engine that operates both continuously at runtime and as part of the software delivery lifecycle and IaC automation workflows. They also generate compliance reports both on a schedule and on demand.

The tech department is around 50% of the total workforce, with a mix of full-time employees and contractors. These range in role and experience from specialists to experienced generalists to recent grads. Teams grow at different rates and largely operate without a standardized methodology; however, they roughly follow a Scrum or XP approach with sprint duration and rituals established more or less tacitly rather than explicitly defined and advertised. As a start-up, workdays are relatively fluid, with some localized efforts stretching into nights and weekends, as are roles, leading to uncommon collaborations and even extended stints of pair programming sessions for rapidly developing solutions in high-risk components or troubleshooting pesky issues.

Up to this point, the company has experienced rapid growth. It started with a handful-sized founding team and a proof of concept and more than doubled in size after raising angel capital. This allowed them to launch their initial product. After early successes, they have raised and borrowed additional capital to again grow, this time by more than three-fold, bringing on additional engineering and sales personnel to support the scaling of their

operations, the push into new market regions, and the transition from managed data services to licensed, self-hosted solutions.

In their most recent hiring round, they brought you on as a TF specialist who will effectively operate as the TF team, reporting to the VP of Engineering, though you'll be expected to hire one or two delegates once you're fully oriented (and get a budget approved by leadership). The infrastructure and cloud services were built over time with a mix of approaches, primarily by hand in the web console with some operations initiated with CLI tools. Any existing TF should be considered legacy and can be abandoned. The VP of Engineering has expressed a goal of lean and standardized IaC. Along with the other expressed goal of prioritizing self-hosted tools, this has influenced your resolve to prioritize standardizing compute resources for specific functions such as databases, web servers, container orchestration clusters, and specialized appliances — you recognize this will be an ongoing effort that must be superseded by ad hoc requests from leadership or critical business functions. There is effectively one product, and you are mainly focused on environment and region replication. As such, you're planning to support two TF projects: one for the foundational/core layer — which will include primary structural components, IAM policies, and central networking controls — and another TF project for the application — supporting unique variable values for each supported environment and region. As you build the TF configuration, you're importing existing resources, prioritizing those that have reached a reasonable degree of stability, lest new changes that must be encoded prevent other efforts. There are still substantial infrastructure changes that are performed manually, but these are being tracked, documented, and given justification, with a defined mitigation strategy for reducing this behavior. The practice of vibe coding has been approved for direct development of TF projects, for now. You're leveraging TFE and work closely with the Platforms Team that manages it and the artifact registry. Figure 12-2 depicts the architecture and TF project relationship.

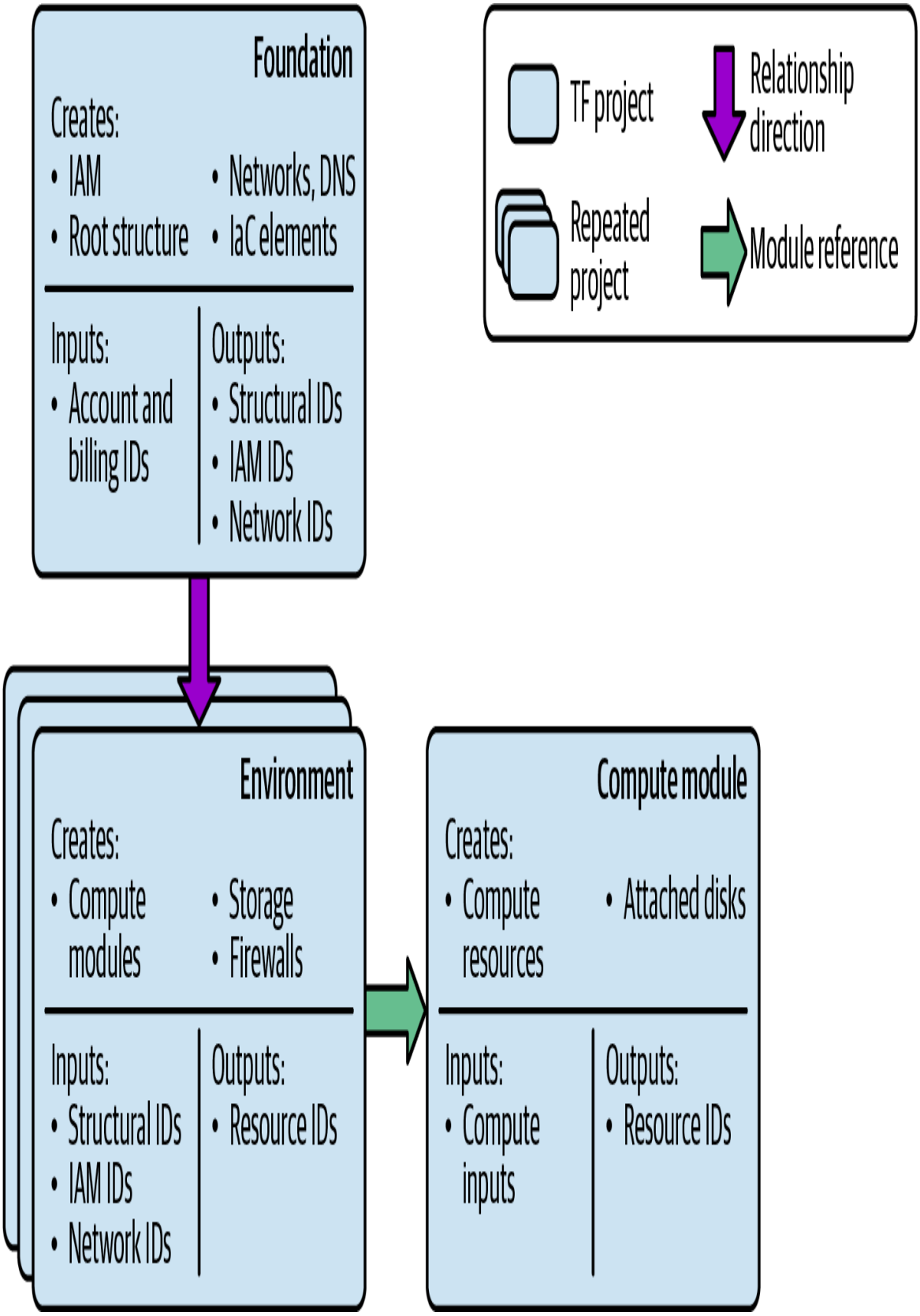


Figure 12-2. Architecture and TF project relationships for SaaS scale-up

Global Firm

For the last case study, consider a food and beverage company with global reach. Suppose they have dozens of brands that are roughly bounded by continents and subcontinent regions and segment themselves into independently governed continental super-regions. They have major operations and offices in each of their super-regions. They have little public-facing software. Most of their digital workloads are intended for inventory management and internal data analysis purposes; they also have some B2B-oriented services for their customers (retail stores, vendor supply, and so on). Most of their notable regulations are food-industry specific and are highly nuanced across various statutory jurisdictions; however, they impose a myriad of internal policies meant to ensure the security of trade secrets. In terms of data analysis, they have extensive market and consumer research and forecasting, as well as analytics intended to inform advertisement spending, the sourcing of ingredients, and the possibility for new markets.

Because most of their workloads are internally consumed or utilized by major customers, many of them are only exposed to internal networks or through private channels. Public workloads are low sensitivity or intended for customer intake. Rather than standardizing on a single software stack across the company, they allow each region to select the stacks they will support and overlay a common security monitoring tool for centralized observability. They have a suite of internally hosted platforms, mostly centrally hosted, and regionally dispersed data warehouses for localized big data and BI analysis. They have a highly heterogeneous database footprint, potentially down to the brand level, but largely impose required database tools and versions for consistency. They employ a highly-secure company-wide RAG system across products, ingredients, and vendors. Due to their highly secretive nature, many systems are accessible only by first obtaining elevated access. Their workloads span multiple CSPs, but this has more to do with notable or traditional differentiators and, in some cases, came about through mergers and acquisitions — there is a persistent effort to standardize and consolidate as much as is feasible.

Their technology department is large, but no more than 10% of their overall workforce and mostly consolidated in a small handful of their corporate sites. Primarily, the tech department exists to collect, store, and surface the vast amounts of data generated across the many brands and markets, with a few cross-cutting teams that operate as both internal auditors and technical shepherds to ensure development teams are aligning to corporate-promoted practices. Multiple branded websites and landing pages may be managed as a portfolio by a single team. There is a mix of SAFe and other agile methodologies, determined (or approved) by regional department leadership.

They have migrated most of their workloads to the cloud long enough ago for them to go through multiple phases of modernization. Over time, TF projects have grown, evolved, split, been abstracted for individual environment support, and decomposed into modules. There is some definite CSP sprawl, with unnecessary redundancy or repetition, and this is also manifested in the TF projects. New exploratory initiatives are developed in sandboxed environments.

The cloud ecosystem is primarily segmented into these super-regions, with a small, centralized space for corporate technology services. Centralized security and networking teams are augmented by regionally distributed departments and task forces designed to address localized behavior and threats. The many TF projects are layered into several rings of protection and standardization, with the most central layers achieving the highest degree of standardization and focusing on common structural and operational needs, and the outer layers constructed from a mix of centrally distributed modules and custom configurations to support the needs of local brands. Likewise, there are many teams that develop TF projects both centrally oriented and regionally distributed. They primarily leverage TFE to manage their infrastructure and integrate an ISP to protect sensitive operations like wildcard jobs — exposed through a general purpose automation platform, separate from TFE — and their production infrastructure release workflows. Despite being spread throughout the BUs, the TF teams operate in a loose SAFe format, with some level of alignment

in program increments across BUs and corporate. They are constantly supporting new lines of business and products and are highly sensitive to unauthorized or unexpected access to their systems. The architecture and TF project relationship for such a company can be found in Figure 12-3.

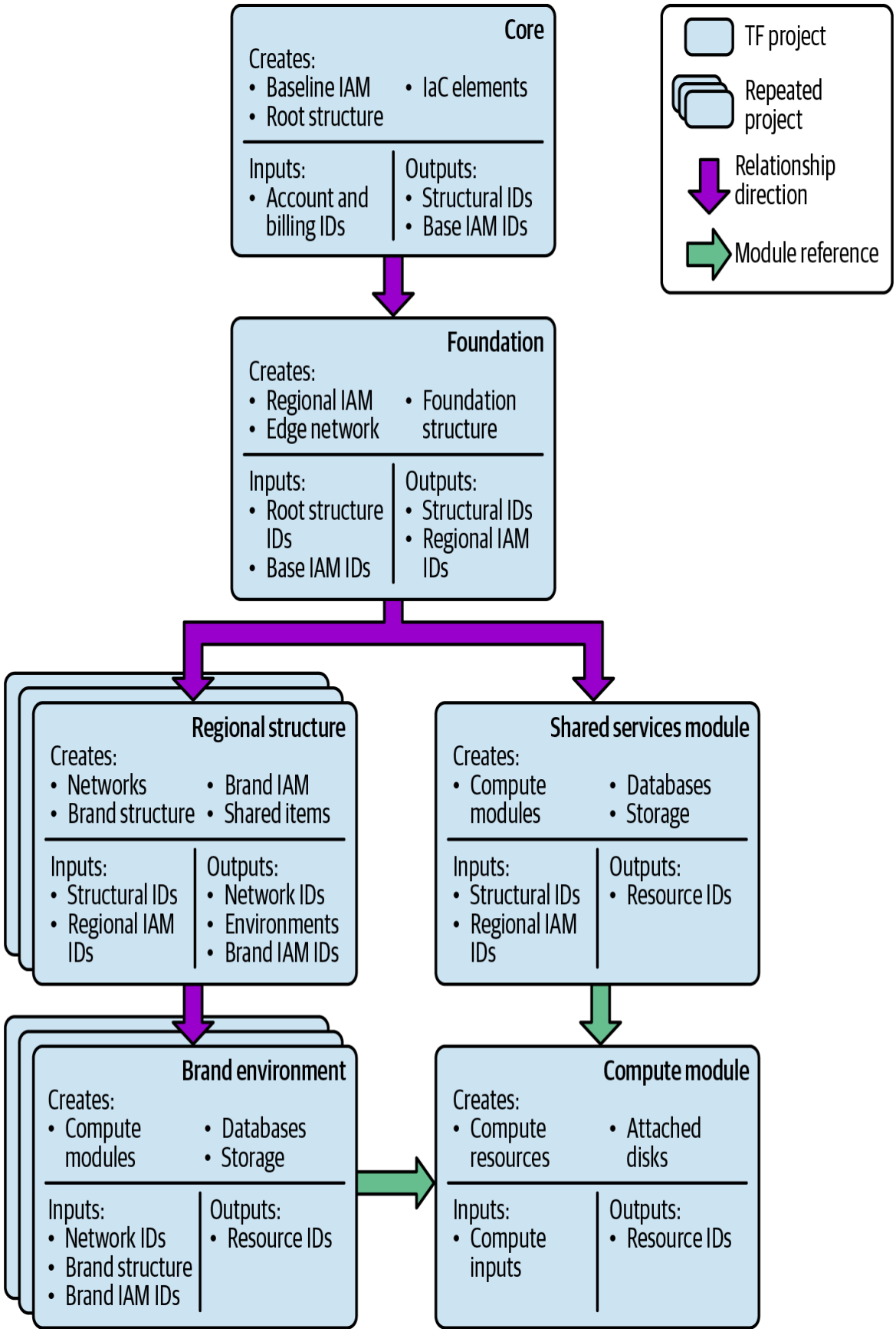

Core
Creates:
• Baseline IAM
• Root structure
• IaC elements
Inputs:
• Account and billing IDs
Outputs:
• Structural IDs
• Base IAM IDs
Foundation
Creates:
• Regional IAM
• Edge network
• Foundation structure
Inputs:
• Root structure IDs
• Base IAM IDs
Outputs:
• Structural IDs
• Regional IAM IDs
Regional structure
Creates:
• Networks
• Brand structure
• Brand IAM
• Shared items
Inputs:
• Structural IDs
• Regional IAM IDs
Outputs:
• Network IDs
• Environments
• Brand IAM IDs
Shared services module
Creates:
• Compute modules
• Databases
• Storage
Inputs:
• Structural IDs
• Regional IAM IDs
Outputs:
• Resource IDs
Brand environment
Creates:
• Compute modules
• Databases
• Storage
Inputs:
• Network IDs
• Brand structure
• Brand IAM IDs
Outputs:
• Resource IDs
Compute module
Creates:
• Compute resources
• Attached disks
Inputs:
• Compute inputs
Outputs:
• Resource IDs
TF project
Repeated project
Relationship direction
Module reference

Figure 12-3. Architecture and TF project relationships for global firm

Useful Heuristics

Throughout the book, you have encountered several scenarios that require changes to a TF project, such as extracting resources into a nested module or into a separate TF project. The following subsections provide some guidance in how to determine whether and how to split up TF projects.

Dealing With Different Resource Lifecycles

Suppose your TF project is split into service-specific *.tf* files. If you realize that certain files don't get more than one update per quarter and others are updated each sprint, perhaps multiple times, you may consider abstracting configuration into two or more *nested modules* or even to split them into full TF projects, as shown in Figure 12-4.

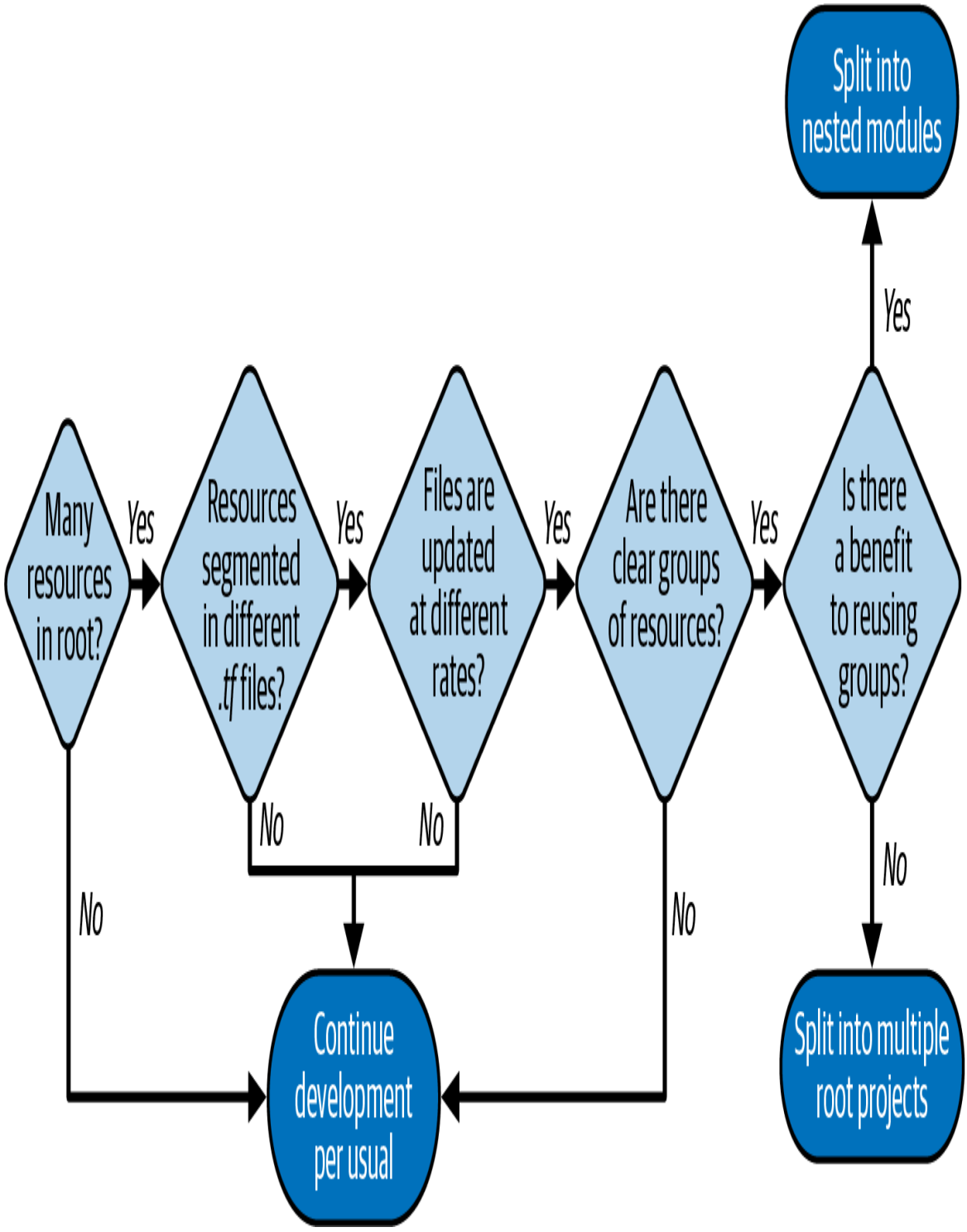


Figure 12-4. Decision tree for splitting a TF project with different lifecycles

Long-Running Validation in Workflows

If you find that you regularly have a queue of changes waiting to work through your IaC automation workflow because you have a substantial set of validations that take a significant amount of time to run all of them, you may consider a few options. First, you might extract these validations from the normal release pipeline and have it run as part of a pre-release process, such as being triggered by opening a merge request in your source control system. If you must keep the validation in your release workflow, you may consider configuring the validation to only run checks that correspond to the resources that are updated by the change, rather than running all validations every time the workflow is triggered. This is depicted in Figure 12-5.

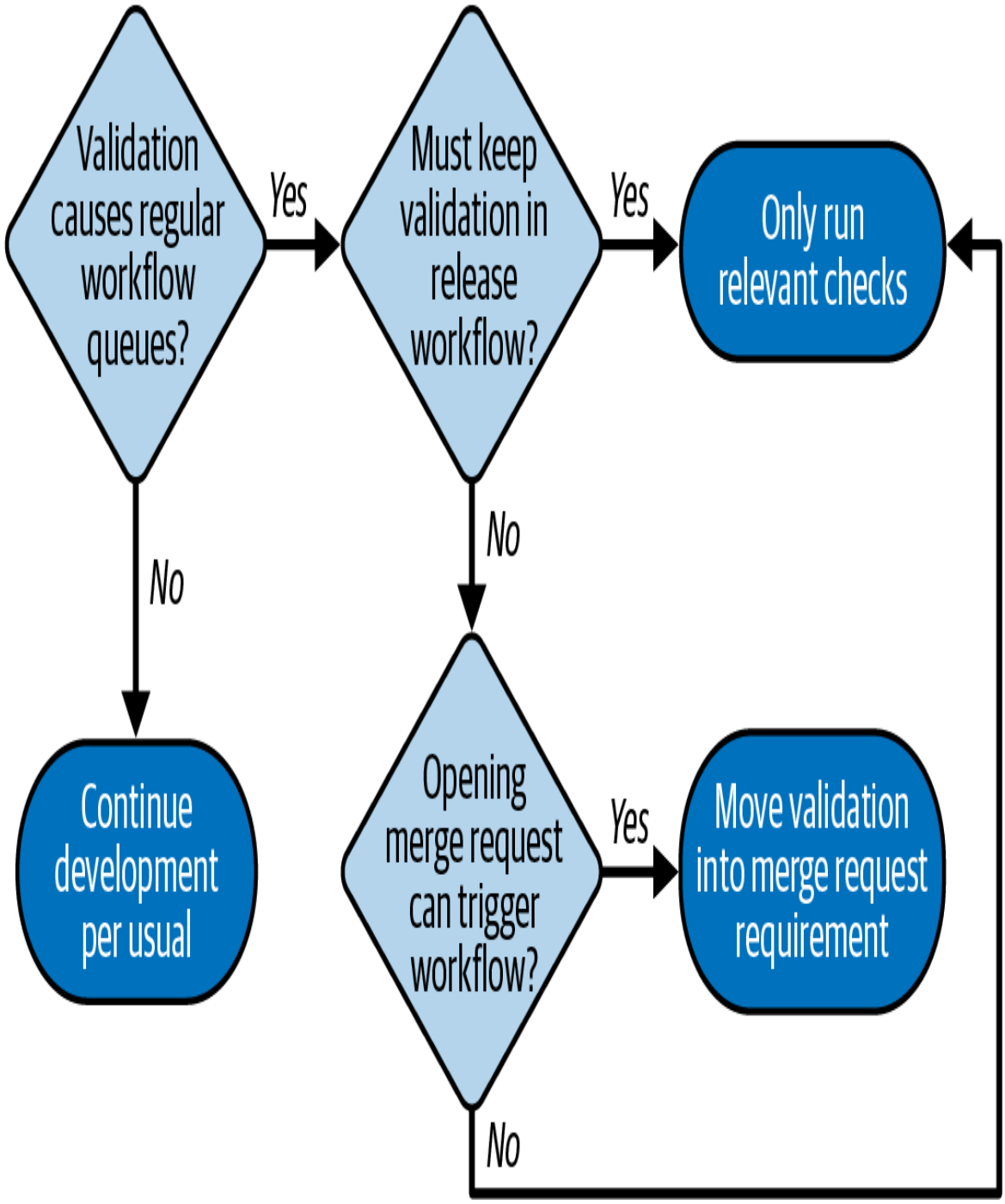


Figure 12-5. Dealing with long-running validations that cause release workflow queues

Extracting Conditional Resources

Suppose you have a large TF project with many `count` expressions for conditional resources. It may be worth extracting them into a module or splitting them into distinct projects. If any of these resources are closely

related, and especially if they're part of the same general resource domain, consider abstracting them into a nested module. Or if, instead, a large-enough portion (say more than 50%), is conditional in an environment-style TF project and the conditional resources are only necessary in a single environment (such as production), you could extract these resources into a project that is dedicated for that environment. A decision tree for this can be found in Figure 12-6.

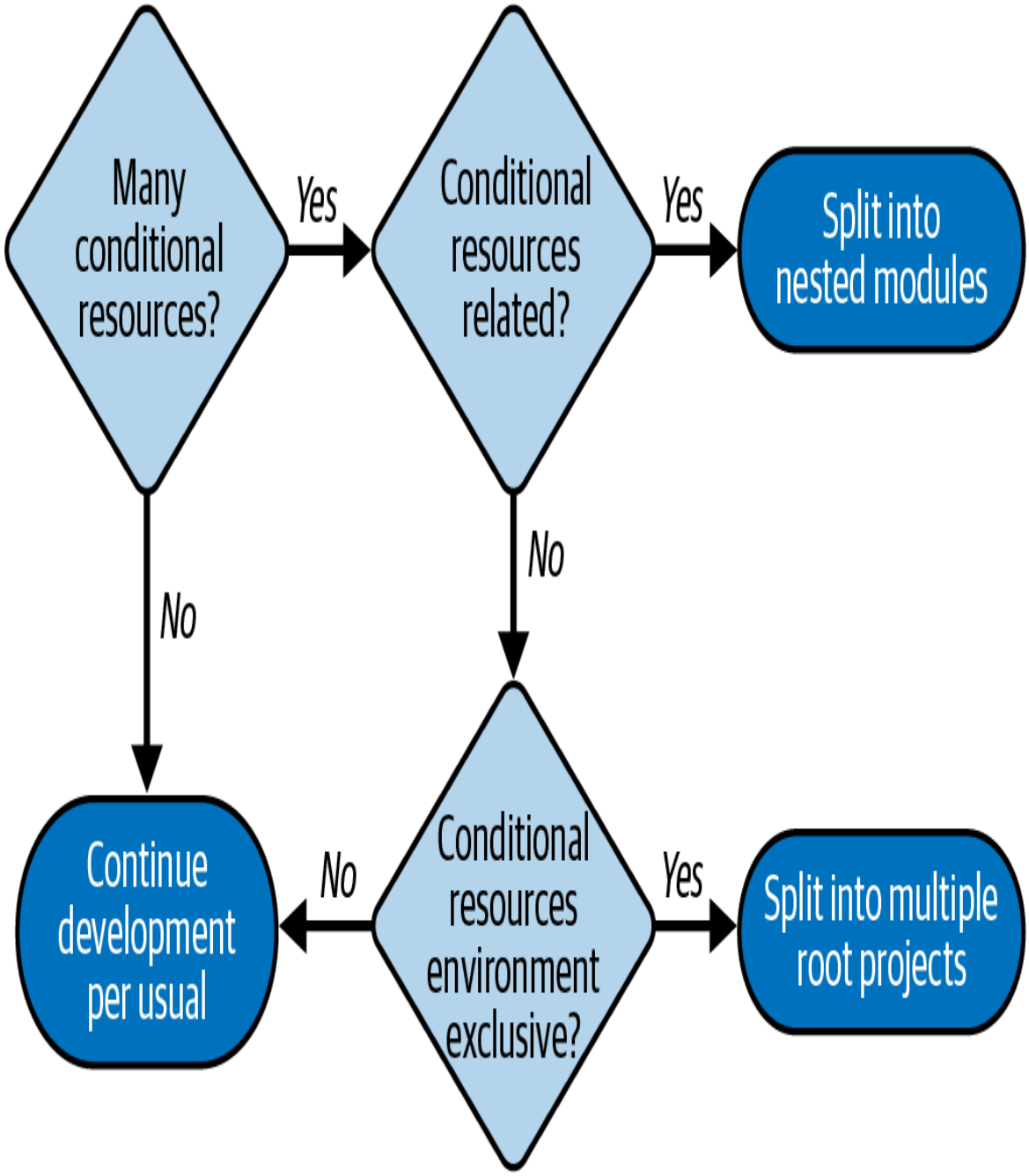


Figure 12-6. Extracting conditional resources into a module or separate project

Dealing with Repeated Resources in Nested Modules

Suppose you're extracting a set of repeated resources — which leverage `for_each` meta-argument — into a nested module. You have to decide whether the `for_each` statements should be moved into the module with the `resource{}` blocks or if it should be applied outside in the

`module{}` block. If the resources are directly corresponding with equal numbers of each resource, perhaps even using the same input variables, the `for_each` likely should be external to the module — in the `module{}` block that references it. If, however, the resources are not all one-to-one, the `for_each` statements should follow the `resource{}` blocks into the module. This decision tree can be found in Figure 12-7.

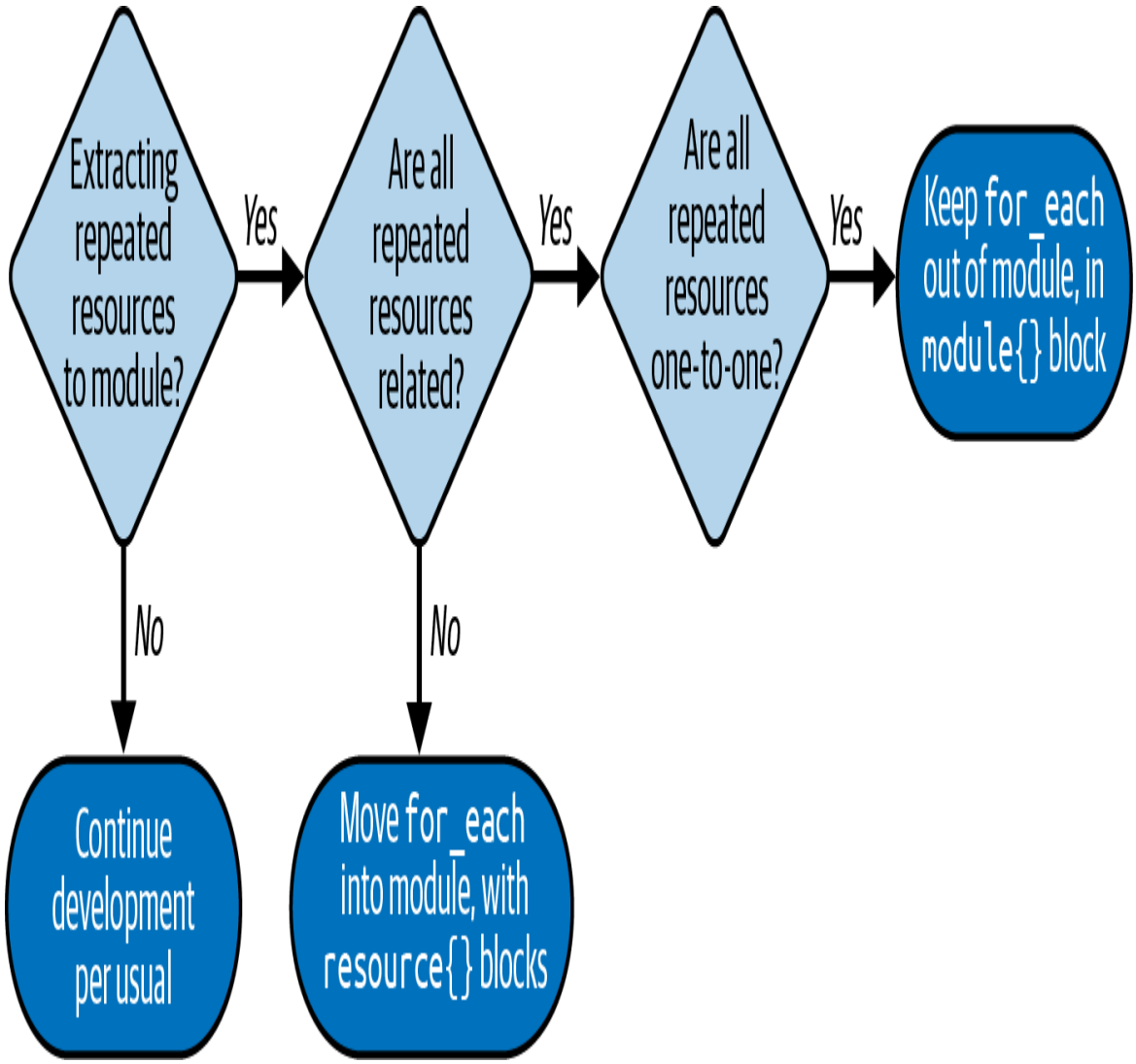


Figure 12-7. Deciding whether `for_each` should go inside or outside a nested module

Dealing with Drift and Manual Updates

Suppose you notice unexpected drift being reported during TF operations. Through further analysis — reviewing system access monitoring

dashboards and logs — you determine that there are inappropriate manual changes being applied to resources managed by TF accounting for many of these occurrences. You know who the perpetrators of the manual changes are, and you know that some of them may have a valid business justification (or the authority to self-justify). As such, you may need to enable a wildcard job capability and potentially even introduce a way to obtain temporary access. However, you may need to investigate further, on a case-by-case basis, what is going on with any occurrences of drift that isn't found to be the result of a manual change. Common culprits include false change reports, such as when using the preferred resource identifier construct, and expected changes that should be ignored in a `lifecycle{}` block of the resource definition, such as a change to a compute instance's network interfaces, storage device mapping, or swap utilization. Therefore, you want to provide a mechanism for determining how to proceed, as shown in Figure 12-8.

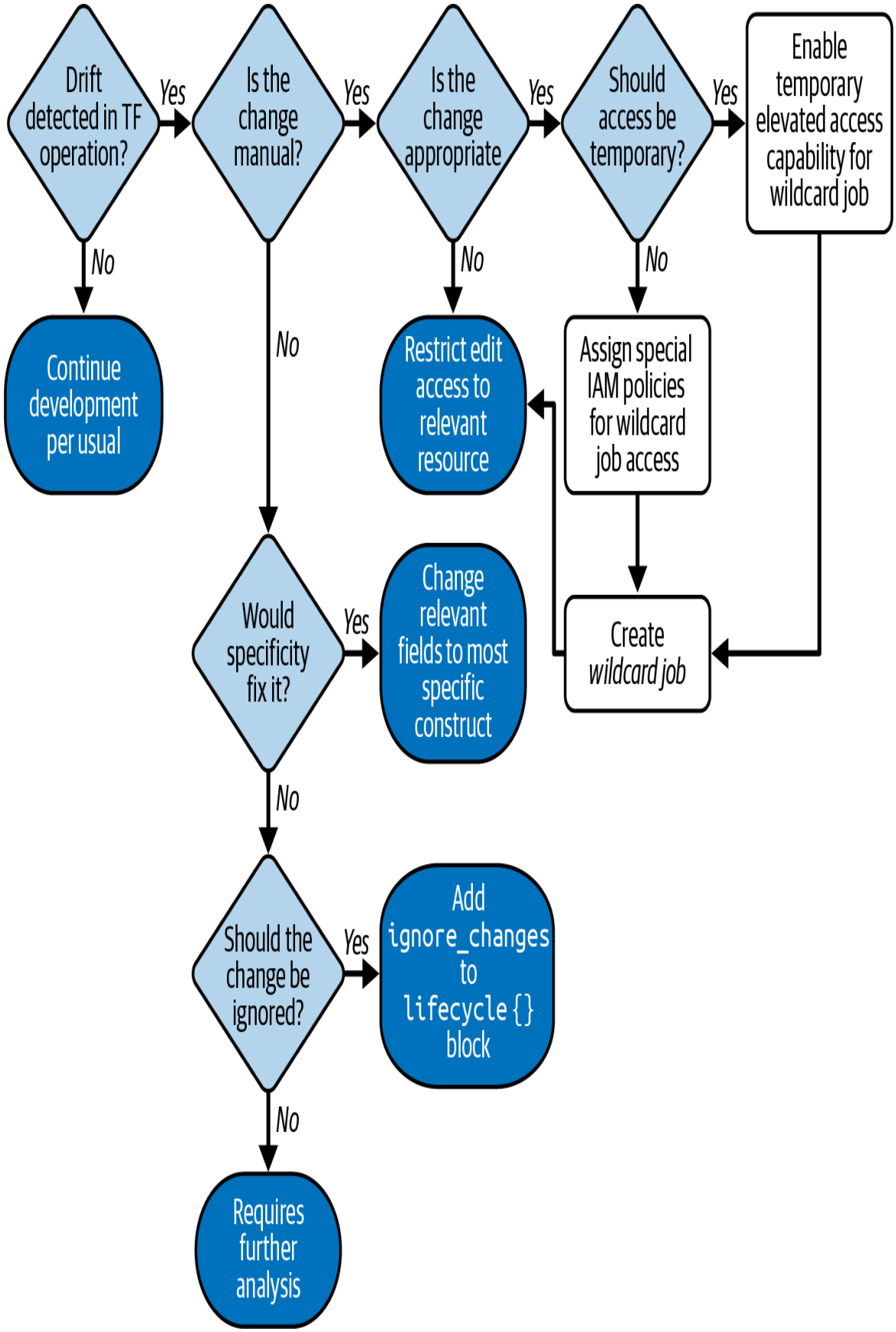

Drift detected in TF operation?
Yes
Is the change manual?
Yes
Is the change appropriate
Yes
Should access be temporary?
Yes
Enable temporary elevated access capability for wildcard job
No
Continue development per usual
No
Restrict edit access to relevant resource
No
Assign special IAM policies for wildcard job access
Would specificity fix it?
Yes
Change relevant fields to most specific construct
Create wildcard job
No
Should the change be ignored?
Yes
Add ignore_changes to lifecycle {} block
No
Requires further analysis

Figure 12-8. Determining the approach for mitigating manual updates

Minimum Viable Infrastructure

The concept of a minimum viable product (MVP) is a common milestone or aspiration in the field of product development, especially commercial SaaS products. In his book, *The Lean Startup* (Crown Business), Eric Ries defines MVP as "that version of the product that enables [iterative learning through a feedback] loop with a minimum amount of effort and the least amount of development time." In other words, what are the minimum features, with minimum effort to build, that will attract and retain users (any users). If we extend this, conceptually, to IaC, we can construct an analogous definition: that version of the infrastructure that enables iterative learning through a feedback loop with a minimum amount of effort and the least amount of development time. Using this as a basis, you can begin to conceive how this unpacks. The resulting infrastructure must be:

- Usable by end users, which in this case of infrastructure is typically your application teams (or data analysis teams, in the case of data infrastructure)
- Developed as quickly as possible
- Manages the fewest amount of resources to be usable
- Includes a mechanism for collecting feedback from users and the system, such as ongoing issues or requested feature support

OTHER DEFINITIONS OF MINIMUM VIABLE INFRASTRUCTURE

There are several other definitions and descriptions of minimum viable infrastructure (MVI) from various sources, any of which may be appropriate in certain settings. However, the majority of them diminish (if not outright ignore) the iterative learning feedback loop, instead focusing on the latter elements of the definition: the minimum effort and development time needed to deliver something usable. Although it manifests in a different way than in the concept of MVP, the feedback loop is a crucial element to the MVI that cannot be overlooked.

In keeping with the *minimum* theme, MVPs are also often delivered with minimum quality. This may not be acceptable in the case of an MVI, which can be far more costly if compromised. It's one thing to ship a product with a bug: for the most part, it's a matter of troubleshooting, finding the defect, and releasing a patch, especially for SaaS products. On the other hand, even a seemingly small gap in your security infrastructure, if discovered by the wrong person or bot, can wreak havoc on your cloud resources, access, and billing account. Thus, a canonical MVI will likely have higher expectations for quality than does an MVP.

Just as every application has a unique MVP, every architecture will have a unique MVI. However, there are several components that will almost certainly be part of any valid MVI. For example, every cloud environment, regardless of its workloads, has access controls. To collect feedback from whatever workloads it supports, there must be a mechanism for collecting logs, such as system access logs and application logs. Most CSPs support a way to assign additional identification mechanisms beyond name and ID, commonly referred to as *resource* tags. These can be used to identify a resource or group of resources in various search features provided by the CSP, such as in log filtering, search features presented in the CLI or console, and cost attribution, all of which are highly valuable and enabled by the same mechanism — which is fairly cheap to implement, even at

scale. Depending on the CSP, there may even be a baseline set of account–wide policies that are worth implementing. Beyond that, an MVI is likely influenced by the type of workloads it will be supporting. This is highly dependent on how specific, for example, the application team's infrastructure requirements are. If they require a specific managed solution with a pinned version and explicit resource properties, the minimum implementation may be largely similar to a maximal implementation — used here as the version that meets the needs of all known end users and cannot be extended without degrading the UX or UI — up to module abstraction or parameterization of field values. If the needs are less well defined, an MVI may start with many of the provider's default resource values by omitting its explicit assignment to a value or variable.

> **NOTE**
>
> One caveat with implementing an MVI approach: changes to your TF state as your infrastructure support evolves may require reorganization of the state object. Although this is almost inevitable over a long enough time, the feedback loop presented by MVI suggests frequent changes. Assuming your changes are indeed informed by feedback gathered from the infrastructure's end users, there is no mechanism for constraining the impact of changes with respect to the TF state, unless you define these explicitly. Taking it to the extreme, this could result in a frequent need to rewrite the TF state. Given the risk of such updates, this is certainly an unwelcome prospect.

In Ries' presentation, he calls the learning feedback loop the Build-Measure-Learn, by which a product's development team first builds and delivers a working version of a product, collects feedback from end users and available metrics of the product, analyzes this feedback — such as by troubleshooting a bug or spiking out a new feature — and builds and delivers a new version of the product. In the case of MVI, there are many things you'll want to measure and collect, including things like access logs, application logs, computer resource metrics, frequency of infrastructure drift by TF project and CSP service, number of resources or attributes affected by each drift, release velocity, and infrastructure costs. Whereas the logs and resource metrics may be largely supported to an adequate degree

by the CSP, by default, other measurements may require custom logic to generate or derive. Certainly, for an MVI, these should be implemented with as little effort and in as few sprints as possible, such as by using scripted calculations with available tools and preserving these results as log entries, thereby reusing an existing platform component; this can be extended if it doesn't meet the needs of the end users, and since you're likely analyzing logs and metrics in the CSP itself, you're technically an end user.

In addition to the feedback from the system, there should be a standard way for the users of the infrastructure — the application or data teams — to report bugs or request new features. For an MVI, this might be as simple as a designated communication channel such as a specific email or chat tool conversation. If you already have a task board or workflow ticketing system, such as Jira or ServiceNow, you may choose to incorporate this into your MVI from the beginning; however, rather than spending significant effort on constructing templates or forms, the initial version of such a solution should be relatively simple and unstructured to support any available feedback.

The following steps illustrate an example workflow for the Build-Measure-Learn feedback loop in the case of MVI:

1. Build

First, build an infrastructure that supports the needs of the application team you're supporting, initially, incorporating a baseline of feedback mechanisms and with the least amount of effort and development time.

This may start with hardcoded values or rampant use of default values and untyped variables, depending on how fastidious you are about proper parameterization.

2. Measure

Next, collect the feedback generated by your infrastructure, the platforms that support it, and the applications that run on top of it, and combine this with the reports issued by your end users directly.

Per the general concept, this will start with limited support and will grow over time and at different stages of your infrastructure: in the earliest iterations, it might be limited to the most basic static code analysis you can comply to regulatory and corporate policies with; in later stages, there will likely be comprehensive dashboards and practical alerts.

3. Learn

Then, analyze the feedback, considering correlations and event sequences, troubleshooting any defects and, possibly, piloting any new features with which you aren't yet familiar.

Piloting here may also have a different meaning at different stages of your infrastructure: early on, it may simply mean a discipline around rigorously reviewing the results of a planning phase; later, it may include tests that provision and evaluate live resources in isolated, ephemeral environments.

4. Iterate

Finally, develop an update to the infrastructure that addresses the findings from Step 3, validating that the change doesn't introduce any defects, and release it back to end users for the next iteration.

Remember, this may require updates to the TF state that may require manual modification using the `tf state` command.

In theory, following these steps will result in a self-consistent infrastructure by the end of each loop.

TF Maturity Model

One potentially fruitful area of exploration is to formulate a TF maturity model. A *maturity model* is a framework for analyzing an organization's capacity for continuous improvement in a specific field of practice. It

reduces the myriad practices of an organization to a graduated set of qualitative designations and is typically meant both as a present assessment of the organization — where the organization is at the moment — and as a guide for attaining a greater degree of such capacity — that is, by achieving the next maturity level, if it exists, through a series of initiatives designed to cover any gaps or develop any missing capabilities.

Many such frameworks exist, such as the Capability Maturity Model (CMM) commissioned by the US Department of Defense (DoD), which includes five levels — initial, repeatable, defined, capable, and efficient — and may be applied to individual processes. Because of the individualized nature of this model, a single organization may have dozens of capabilities at different levels of maturity. In contrast, the Open Source Maturity Model (OMM) has only three levels — basic, intermediate, and advanced — which apply to an open source project as a whole.

For each level of the CMM, a process will have specific goals that it must achieve before claiming the level, and each goal may require performing multiple activities. For example, at level two, repeatable, a typical goal would be the presence of documentation, which can be achieved through a combination of related tasks. Similarly, the OMM defines necessary practices that the project must include to be considered as being at that level. In the OMM, documentation is again one of the practices required, in this case, to achieve the basic level, level 1.

MENTAL MODEL

You've already seen how different maturity models may take different approaches to achieve a qualitative banding of aptitude in their respective field or context. In your work as a technologist, you may encounter multiple such models throughout your career, or even within the same initiative. This can create confusion when considering one model versus another. The following are several common implications and key takeaways that can be applied to every model:

Not an indictment

> People sometimes bristle when discussing their project's maturity. This is likely due to pride and can dampen their interest in using a particular framework. However, it's worth pointing out that every project that achieves a high level of maturity goes through the earlier levels. Assigning a low level is not meant as a denunciation of the work, but is meant to give the project's development team and decision makers a rubric with which to plan or validate their priorities and practices.

Rises to the most business-efficient level

> Although it's natural to want to reach the maximum level of the particular framework you're using, it's not always necessary — or rather, it's not the case that every process (or project) should advance at equal rates. For example, in the CMM, if you have dozens of processes, you may choose to prioritize those processes that are most frequently and regularly executed, taking each of these to level 5 — efficient — as quickly as is feasible. Somewhat less frequent or regular processes may also be prioritized for advancement, but maybe only to the defined or capable levels (levels 3 and 4, respectively), with a future effort to progress these further. Finally, very infrequent or highly ad hoc processes, especially those that are highly nuanced or subjective, may simply target level 2 — repeatable — as a serviceable level at which to stay

indefinitely. Investing in higher levels of maturity in these processes may sap resources from more lucrative activities and may even be more costly than the benefit it would achieve.

No partial levels

As alluded to previously, achieving a particular level requires all of the features described in that level. In particular, there is no useful concept of a *partial level*, whereby you achieve some of the goals of a level, and thus are rated at a fractional level between it and the previous one. Instead, you would simply remain at that previous level. For example, suppose an open source project effectively conducts all of the basic level practices and also curates an available project roadmap, well-defined milestones, and formalized governance, but does not include an extensive test suite or documented comprehensive architecture and interface designs. The project's development team may desire to claim a score of, let's say, 1.4; however, they remain at level 1 until all of the other practices are taken on.

Using a maturity model framework can help decision makers, such as a steering committee or governing board, orient their approach to roadmapping, planning, and budgeting activities and helps to define outcomes and establish business justifications for various initiatives. Rather than focusing on whatever features you're most comfortable with, or capable of, undertaking from a capacity perspective, it highlights areas you may be overlooking or avoiding. It can even clarify and help articulate any skill gaps you may have in your organization, which may necessitate personnel investments in the form of training your workforce, hiring additional team members, or contracting external entities. It can be used both as a structure and rationale to effectively justify a budget request and as a guide for prioritizing and scheduling work based on a granted budget.

EXPLICIT MATURITY MODEL VERTICALS

Whereas most frameworks that claim the mantle of *maturity model* do not explicitly define distinct verticals that span all levels along which a particular grouping of activities progresses, this can be a highly valuable perspective. For example, in my work with the Cloud Native Computing Foundation (CNCF) community group, Cartografos, we developed the Cloud Native Maturity Model (CNMM), which takes the traditional maturity model framework and expands it across different dimensions or aspects: technology, people, process, policy, and business outcomes. This came out of a simple premise: company-wide software initiatives — such as choosing to adopt a platform like Kubernetes, the primary technology in the cloud native space — are not confined to purely technological considerations, but generally impact your people (including technical and nontechnical personnel, as well as customers and other stakeholders), processes (both digital and manual), policies (both written/encoded and unwritten/understood), and business outcomes (the aggregate impact to the proverbial bottom line). In the case of the CNMM, all aspects must achieve a particular level for an organization to claim that level — the corollary being that an organization's maturity is equal to its least mature aspect — but this is not necessarily the case (in our case, this was primarily to avoid a partial level due to averaging across aspects, which we felt was either too arbitrary or subjective).

In terms of TF, what might a maturity model look like? Taking a CMM approach, you might consider a perspective whereby your TF ecosystem maturity is really a collection of capabilities, each of which have their distinct rate of level progression. For example, you may segment by team or role, such as release management, module administration, project team development, and so on. Does it make sense to let these capabilities advance, organically, independently? Would these practices even significantly diverge given the strong relationships between them?

Or, taking inspiration from the CNMM, you might consider non-purely-technical aspects such as the people and policies involved in developing your TF ecosystem. As you've seen throughout the book, the procedures and policies you'll implement as your systems grow and evolve will undoubtedly change over time. Likewise, there will very likely be changes to the personnel directly responsible for your TF ecosystem, perhaps starting with a small, cross-functional team of highly skilled engineers before expanding to multiple layers of roles and responsibilities, perhaps only to shrink again as your systems become self-service and fully automated. If so, is there a benefit to taking on an overall score viewpoint, as CNMM does, requiring all aspects to be at (or above) the same level to claim that maturity level?

Suppose we do consider the aspects from the CNMM. Each of the dimensions might be described in the following way:

Technology

> This is probably the easiest aspect to map and is the primary one found in this book. It would include standards around module development, the processes by which you collect and release changes, and the level of automation your system employs.

People

> As described previously, this aspect would consider the teams involved in authoring TF code or configuration, those that manage the release process, business stakeholders, developers of application workloads that run atop the cloud infrastructure, and even end users of those products.

Process

> These may start largely manual, transition to being manually directed, and later become highly or fully automated. Most processes would have a technology component, but you could consider them to be goals rather than prescriptions.

Policy

These can be a mix of written and unwritten rules and may be imposed by a regulatory body or corporate mandate, or may even be self-assumed explicitly or implicitly. The considerations within this aspect would again likely have a technology component, but there is value to consider the aspect in the abstract before worrying about implementation details.

Rather than attempting to supply a complete framework, it may be more appropriate to describe the broad strokes with the goal of beginning the discussion. Here is a high-level perspective:

- Early on, you may have a single TF project developed by a single, highly skilled team. Your procedures may be largely manual and your policies largely enforced through discipline and tedious audits.
- Later, you might have many TF projects that are loosely associated and developed by different disparate teams in collaboration with centralized teams focused on security, release management, and observability. You probably have a significant degree of automation and may enforce a constraint on direct manipulation of resources, requiring changes to be initiated from within a specific platform in addition to other policy enforcement.
- In the most advanced settings, you may have again shrunk to a small collection of highly skilled teams that supports a massive and intricately connected assemblage of tools and automated processes governed by comprehensive policies enforced at multiple points.

With this perspective, you can see how all aspects must, to some degree, progress together. It is intentionally ambiguous, committing to neither strict goals nor a precise number of levels. This is intended to leave room for the aforementioned discussion.

> **NOTE**
>
> The business outcomes aspect is probably the hardest to map, which is why it is left off of this analysis. Some portions of this can likely be derived from the other aspects. For example, choices made in the technology aspect, such as a licensing agreement you enter into to tackle a particular practice, will obviously come with a monetary cost. Likewise, personnel changes will impact your costs. However, it's difficult — if not impossible — to divine all components of this aspect for TF (or for any technology practice) before the framework is mapped to real organizations or case studies.

TF Versus Others

TF is certainly not the only IaC tool available. Viable options include CSP-managed platforms and tools — such as AWS CloudFormation, AWS CDK, Azure Resource Manager (ARM), Azure Bicep, and Google Cloud Infrastructure Manager — as well as other third-party tools such as Crossplane and Pulumi. In this section, you'll review how each of these tools compare to TF both functionally and in how they are operated.

Whereas TF utilizes a domain-specific language, HCL, in addition to JSON and YAML encodings, CSP-managed platforms like CloudFormation and ARM support only JSON and YAML configuration constructs. Although these are both well-understood ways of encoding information across the industry, they're almost expressive to a fault, whereas HCL is relatively standardized in terms of format and how information is organized. Moreover, TF can manage all CSPs in a common format — and even the same TF project, if preferred — rather than needing completely distinct configurations for each CSP, reducing the necessary skillset in the case of migrating from one CSP to another or in the case of a multicloud ecosystem. Similarly, although even the most directly analogous cloud platform resources are almost never direct mappings, one CSP to another — so switching from one CSP to another may require significant rewrites at the resource level — in theory, TF can reuse much of the same structure. This means you probably won't have to restructure TF projects or their respective inter-relationships as heavily as transitioning from CloudFormation to ARM, for example. That being said, TF can lag in

support since it relies on the provider plug-ins being updated. In the case of the most common CSPs, both AWS and Azure provider plug-ins are maintained by HashiCorp, and the GCP plug-in is maintained by Google. Although there is a clear case for the CSPs prioritizing support for TF, it's unlikely that they will prioritize it *over* their internal tools, when all else is equal.

Tools such as Azure Bicep and Google Cloud Infrastructure Manager are quite similar to TF. Bicep, for example, leverages a DSL structured very similarly to HCL. However, it has certain built-in functionality specific to Azure and compiles directly to ARM. Also, because Bicep compiles to ARM, TF can lag behind Bicep in terms of supported features and resources and may be preferred in cases where a team will only ever be responsible for Azure infrastructure. In contrast, Google Cloud Infrastructure Manager directly accepts TF HCL configurations, using the exact same provider plug-in. It effectively runs a specific version of Terraform as a managed service. This suggests that it will match TF in terms of support.

Crossplane is a platform developed by Upbound and runs on top of a Kubernetes cluster. This means Crossplane is running in a constant loop, the Kubernetes event loop, so it will continuously reconcile resources to the desired state as specified in composite resource definitions (XRDs), which are effectively thin wrappers around Kubernetes custom resource definitions (CRDs). This makes Crossplane more strongly aligned to GitOps principles — which use Git repositories as the single source of truth — whereas TF can really only approximate GitOps, and still requires orchestration (but perhaps the real question is, "do you even want GitOps?"). However, TF has a broader ecosystem and TF providers are generally built by either HashiCorp or the CSP itself, whereas Crossplane providers largely rely on community support. Also, because the free version of Crossplane requires a Kubernetes cluster to run, it suffers from the classic chicken-and-egg problem: what configures this cluster? Meanwhile, TF is easier to bootstrap and can avoid this problem using a local state to initialize, for example, a cloud-hosted backend.

> **NOTE**
>
> A highly effective approach might be to use TF and Crossplane together. Crossplane can be leveraged to manage the majority of your cloud resources, taking advantage of its continuous reconciliation. However, TF can be used to bootstrap Crossplane: initial, core, and foundational resources — such as a remote backend solution, baseline IAM controls for IaC operation, and the Kubernetes cluster that Crossplane runs on — are created with TF, initially using a local backend for the state. This state can then be migrated to the remote location, thereby ensuring it can be maintained consistently. Thereafter, Crossplane would become the primary IaC platform.

Pulumi is a tool developed by the Pulumi Corporation and is both a command-line tool and set of SDKs for popular programming languages. Thus, it has several advantages over HCL in that it can be made typesafe, has much more logic expressiveness, and can be tested more extensively and directly. Just like TF, Pulumi records state to a file and can store this in a remote location. It can also be run anywhere, just like TF, and detects and remediates drift in much the same way. It has its own set of providers, and you can also create a wrapper around any TF provider plug-in, meaning it can theoretically do whatever TF can do. However, although Pulumi is designed to converge idempotently, like TF, the fact that it's inherently a programmatic model gives you flexibility that can result in nonidempotent logic.

Philosophical Musings

Over the course of this book, there have been several unconventional perspectives presented to address what might be considered corner cases in TF development. This section includes additional theories and thoughts that may be on the fringes of TF development. They are included in an attempt to fill any remaining gaps in the mental model presented in this book.

Theory on Nested Modules

As suggested throughout the book, avoid mixing `resource{}` and `module{}` blocks. If you extract resources from, say, a root module and bundle them into a nested module, it's worth considering bundling the other resources into their own modules. This maintains a strict structural distinction between modules that manage resources and modules that manage other modules. However, this is probably not worth it for singular resources that would be placed in singleton modules (although this may suggest that a different change is needed). Finally, assuming you're doing this in a live TF project, you're already modifying the TF state, such as using the `tf state mv` command, so it's not as though you would be going out of your way and doing a great deal of extra work.

Autonomous Agents

Increasingly, autonomous AI agents will find their way into IaC workflows. These agents leverage LLMs that are often trained on public data, for example, resources like StackOverflow and other community forums. Before refining results by providing feedback — which effectively amounts to *supervised machine learning* — these models largely suffer from garbage in, garbage out, whereby the quality of the output is proportional to the quality of the input, meaning they're limited to the quality of the data on which they're trained. In other words, they effectively operate by copying and pasting what they've seen before, which is generally considered an amateur procedure. However, this is not much different from how many legitimate human practitioners started out and through trial and error learned better ways of working. This explains how many code generators can do a serviceable job at developing arbitrary solutions, however, it also suggests some level of diminishing returns or limitations in terms of developing a truly novel approach without significant guidance from an expert-level practitioner.

State Triplet Theory

The state triplet, as presented in Chapter 6, is likely an inherent property, in some form, to any IaC solution. The resources are an obvious commonality,

as are the configurations themselves. Thus it's only necessary to show that there is always a third component that represents the source of truth. In CSP-managed platforms and tools, the platform itself encodes it. Tools like Pulumi explicitly leverage a state that operates exactly like TF's. Even in Crossplane, the composite resource definitions get loaded into the cluster as Kubernetes manifests and are, for all intents and purposes, analogous to the TF state, except instead of a static and passive state backend, the source of truth is stored directly in the cluster that continuously works to reconcile the live resources with it.

Summary

In this chapter, you wrapped up your exploratory journey through Terraform (and OpenTofu) at scale by reviewing several contextual considerations including the following:

- What TF development looks like at different scales
- Useful heuristics for determining when and how TF projects should evolve
- A basic concept of a minimum viable infrastructure
- How you might approach solution maturity in the case of IaC
- How TF compares to similar or analogous tools
- Some reflections intended to evoke philosophical discourse

This fills in some remaining conceptual gaps and highlights various areas requiring further study or exploration. You now see how certain concepts hold up, or break down, in normal and extreme cases. You have also been exposed to fringe concepts and considerations that are otherwise generally reserved for nonserious civil discourse or meandering rants, giving you a chance to form your own beliefs in light (or in spite) of them.

You now have a deep understanding of the various considerations that drive TF development at various scales all the way up to enterprise scale. You

have seen successful patterns and useful procedures, as well as problematic scenarios and techniques for mitigating, remediating, or avoiding them. With the guidance and insight gained from this book, you should be confident to take on any IaC initiative involving TF knowing that you won't have to figure it all out as you go along.

Go forth and provision!

Index

A

abstraction, principle of, Important Principles

access control, Access Control-Service Identities

- authoritative versus nonauthoritative, Authoritative Versus Nonauthoritative-Authoritative Versus Nonauthoritative
- policy scopes, Policy Scopes-Policy Scopes
- practical considerations, Practical Access Considerations-Practical Access Considerations
- service identities, Service Identities-Service Identities
- temporary, Successful Practices

account-wide policies, Account-wide Policies-Account-wide Policies

acquired software ecosystem (case study), Absorbing an Acquired Software Ecosystem-Absorbing an Acquired Software Ecosystem

addition/subtraction (arithmetic and logical operators), Arithmetic and Logical Operators

AI chatbot interfaces, Successful Practices

AI, emergent thought in, Autonomous IaC

alerts, recommended, Recommended Dashboards and Alerts

apply, Nightmare Scenarios

apply command (CLI), Important concepts unique(ish) to IaC-Features, The Basics, The lifecycle Meta-Argument and Custom Conditionals, State

Object-State Triplet, Importing State-Root Module, Tightly coupled states, Broken States, Behavior Analysis, Recap of Common Problems

arithmetic operators, Arithmetic and Logical Operators-Arithmetic and Logical Operators

atomic states, Atomic States

automated logic, outside of Terraform, Automated Logic Outside of TF

automation, Automating at Enterprise Scale-Summary

- autonomous IaC, Autonomous IaC-Autonomous IaC
- avoiding pets in, Livestock, Not Pets
- best practices, Successful Practices-Successful Practices
- connecting pipelines with, Connecting Pipelines
- context in, Context Is Everything-Context Is Everything
- diminishing returns with, Definition of Done and Diminishing Returns-Definition of Done and Diminishing Returns
- ephemeral environments (environments on demand), Ephemeral Environments and Environments on Demand-Ephemeral Environments and Environments on Demand
- going beyond the shell with, Going Beyond the Shell-Context Is Everything
- multiple-state coordination in, Coordinating Multiple States-State of States
  - domino effect, Domino Effect-Domino Effect
  - state of states, State of States
  - tracking dependencies, Tracking Dependencies and State

self-service infrastructures, Self-Service Infrastructure-Self-Service Infrastructure

TF process automation, Automating the TF Process-Automating the TF Process

wildcard jobs, Wildcard Jobs

automation platforms, Successful Practices

autonomous agents, Other Touch Points, Autonomous Agents

B

backend (state management), Important concepts unique(ish) to IaC

base infrastructure, building on top of, Building on Top of Base Infrastructure

behavior analysis, Behavior Analysis-Behavior Analysis

bloated states, Bloated States-Bloated States

branches

merging, Merging Branches

names of, Branch Names and Commit Messages

over-trusting branching strategies, Over-Trusting Branching Strategies

Brewer's theorem, Important Principles

broken states, Broken States

C

CAP theorem (Brewer's theorem), Important Principles

Capability Maturity Model (CMM), TF Maturity Model

change management, Change Management for Infrastructure Support

indicators for change, Signals for Change-Signals for Change

state optimization scenario, Scenario 4: organizational change management-Scenario 4: organizational change management

CLI input

apply command, Important concepts unique(ish) to IaC-Features, The Basics, The lifecycle Meta-Argument and Custom Conditionals, State Object-State Triplet, Importing State-Root Module, Tightly coupled states, Broken States, Behavior Analysis, Recap of Common Problems

console command, Using tf console

fmt command, Organizing Field Values

plan command, Important concepts unique(ish) to IaC, Validation of Variables, Importing State, Scenario 1: organizational scaling, Using a Lock Timeout

setting input variables with, Setting Input Variables

test command, Behavior Analysis-Behavior Analysis

validate command, Static Code Analysis, Low-Tech, Low-Cost Solutions

ClickOps, ClickOps

cloud networks, edges of, Edges of Cloud Networks-Edges of Cloud Networks

cloud service provider (CSP), The Basics

CMM (Capability Maturity Model), TF Maturity Model

code editor, The Basics

collection functions, Effective Use of Collection Functions-Effective Use of Collection Functions

commit messages, Branch Names and Commit Messages

composite variables

- construction, Type Constructors: Composite Variable Construction-Type Constructors: Composite Variable Construction
- mixing essential and nonessential data in, Mixing Essential and Nonessential Data in Composite Variables
- overuse of composite variable types, Overusing Composite Variable Types

conditional resources, extracting, Extracting Conditional Resources

console command (CLI), Using tf console

containerized steps, Successful Practices

Conway's Law, Additional Considerations-Additional Considerations, Autonomous IaC

create_before_destroy rule, The lifecycle Meta-Argument and Custom Conditionals

CSP (cloud service provider), The Basics

custom conditionals (meta-arguments), The lifecycle Meta-Argument and Custom Conditionals-The lifecycle Meta-Argument and Custom Conditionals

custom containers, Successful Practices

D

dashboards, recommended, Recommended Dashboards and Alerts

data sources, Features, Data Sources-Data Sources

date/time functions, Effective Use of Other Functions (Numeric, IP Network, and Date/Time)

default value field, Parts of a Variable Definition, Setting Input Variables

default values, misuse of, Misusing default Values

defense in depth, Defense in Depth-Defense in Depth

depends_on meta-argument, Effective Use of depends_on Meta-Argument

description field, Parts of a Variable Definition

domino effect, Domino Effect-Domino Effect

drift, Drift, Dealing with Drift and Manual Updates

duplication, reducing, Scenario 2: reducing duplication-Scenario 2: reducing duplication

dynamic blocks, The dynamic Block-The dynamic Block

E

element negation (arithmetic and logical operators), Arithmetic and Logical Operators

embedded modules, Module Composition versus Embedded Modules-Module Composition versus Embedded Modules

empty variable{} blocks, Empty variable{} Blocks

encapsulation, Important borrowed concepts

encoding() function, Effective Use of Encoding and Hash/Crypto Functions

entry points, Concepts irrelevant to (or diminished for) IaC

environment conflicts, Environment Conflicts

environment variables, Setting Input Variables

ephemeral environments (environments on demand), Ephemeral Environments and Environments on Demand-Ephemeral Environments and Environments on Demand

ephemeral value (input variables), Parts of a Variable Definition

equality checks (arithmetic and logical operators), Arithmetic and Logical Operators

everything is production principle, Everything Is Production

evolution of projects (see project evolution)

F

field values, organizing, Organizing Field Values-Organizing Field Values

filesystem functions, Effective Use of Filesystem Functions

flawed merge conflict, The Flawed Merge Conflict

fmt command (CLI), Organizing Field Values

forever forward concept, Moving Forever Forward

for_each meta-argument, Effective Use of for_each, count Meta-Arguments-Effective Use of for_each, count Meta-Arguments

G

GDPR requirements, General Data Protection Regulation (GDPR)

Git repositories, with OpenTofu, Things to Avoid

gitignore, .gitignore

global firm (case study), Global Firm

graph theory, A Quick Vocabulary Lesson in Graph Theory

gridlocked states, Gridlocked States

H

halfway applied states, Recap of Common Problems

hash/crypto() function, Effective Use of Encoding and Hash/Crypto Functions

HashiCorp Configuration Language (HCL), Effective Use of Logic-Summary

- automated logic outside of Terraform, Automated Logic Outside of TF
- collection functions, Effective Use of Collection Functions-Effective Use of Collection Functions
- date/time functions, Effective Use of Other Functions (Numeric, IP Network, and Date/Time)
- encoding() function, Effective Use of Encoding and Hash/Crypto Functions
- expressions in, Expressions-Splat Expressions
  - arithmetic and logical operators, Arithmetic and Logical Operators-Arithmetic and Logical Operators
  - for expression, The for Expression-The for Expression
  - if clause, The if Clause
  - splat expressions, Splat Expressions-Splat Expressions
  - string interpolation and directives, String Interpolation and Directives-String Interpolation and Directives
  - ternary conditional expressions, Ternary Conditional Expressions

filesystem functions, Effective Use of Filesystem Functions

hash/crypto() function, Effective Use of Encoding and Hash/Crypto Functions

implicit type conversion, avoiding, Implicit type conversion

inline logic, avoiding, Inline logic

IP network functions, Effective Use of Other Functions (Numeric, IP Network, and Date/Time)

length() function, The length() Function

manual processes in, Manual Processes-Manual Processes

numeric functions, Effective Use of Other Functions (Numeric, IP Network, and Date/Time)

optimized versus shipped code, The Messy Middle

organizing field values, Organizing Field Values-Organizing Field Values

performance, Concepts irrelevant to (or diminished for) IaC

reference attributes, Resource Fields and Reference Attributes

resource fields in, Resource Fields and Reference Attributes

safety functions, Effective Use of Type Conversion and Safety Functions-Effective Use of Type Conversion and Safety Functions

self-documenting elements, Self-Documenting Elements

sensitive value functions, Effective Use of Sensitive Value Functions

standard module structure, HashiCorp's Standard Module Structure

standards and style, Important borrowed concepts

string functions, Effective Use of String Functions-Effective Use of String Functions

Terraform functions and expressions in, Terraform Functions and Expressions

things to avoid with, Things to Avoid

type conversion functions, Effective Use of Type Conversion and Safety Functions-Effective Use of Type Conversion and Safety Functions

validation{} blocks, Validation Blocks in HCL-Validation Blocks in HCL

heuristics, Useful Heuristics-Dealing with Drift and Manual Updates

dealing with different resource lifecycles, Dealing With Different Resource Lifecycles

dealing with drift and manual updates, Dealing with Drift and Manual Updates

dealing with repeated resources in nested modules, Dealing with Repeated Resources in Nested Modules

extracting conditional resources, Extracting Conditional Resources

long-running validation in workflows, Long-Running Validation in Workflows

HIPAA requirements, Health Insurance Portability and Accountability Act (HIPAA)-Health Insurance Portability and Accountability Act (HIPAA)

I

IaC (see infrastructure as code)

idempotency, Important concepts unique(ish) to IaC

identity and access management (IAM), Important borrowed concepts, Best Available Practices, Practical Access Considerations-Service Identities

if clause, The if Clause

ignore_changes rule, The lifecycle Meta-Argument and Custom Conditionals

implicit type conversion, avoiding, Implicit type conversion

importing state, Importing State

inequality checks (arithmetic and logical operators), Arithmetic and Logical Operators

infrastructure as code (IaC), Concepts-Concepts irrelevant to (or diminished for) IaC

- alternative IaC tools, TF Versus Others-TF Versus Others
- autonomous, Autonomous IaC-Autonomous IaC
- backend, Important concepts unique(ish) to IaC
- encapsulation, Important borrowed concepts
- idempotency, Important concepts unique(ish) to IaC
- identity and access management, Important borrowed concepts
- locked state, Important concepts unique(ish) to IaC
- provider, Important concepts unique(ish) to IaC
- resource management, Important borrowed concepts
- standards and style, Important borrowed concepts
- stored state, Important concepts unique(ish) to IaC
- technical debt with, Technical IaC Debt-The Real Cost of Technical Debt

Terraform plan, Important concepts unique(ish) to IaC

inline logic, avoiding, Inline logic

input variables, Input Variables-Summary

avoidable practices with, Things to Avoid-Misusing default Values

empty variable{} blocks, Empty variable{} Blocks

misuse of default values, Misusing default Values

mixing essential and nonessential data in composite variables, Mixing Essential and Nonessential Data in Composite Variables

overuse of any type, Overusing the any Type

overuse of composite types, Overusing Composite Variable Types

CLI input, Setting Input Variables

default value field, Parts of a Variable Definition, Setting Input Variables

description field, Parts of a Variable Definition

environment variables, Setting Input Variables

ephemeral value of, Parts of a Variable Definition

fields of, Parts of a Variable Definition-Parts of a Variable Definition

module fields, Setting Input Variables

nullable value of, Parts of a Variable Definition

sensitive value of, Parts of a Variable Definition

setting, Setting Input Variables-Setting Input Variables

.tfvars files, Setting Input Variables

type field, Parts of a Variable Definition

types of, Variable Types-Using the type() Function to Determine Type

any type, The any Type

primitive types, Primitive Types in TF

type function, Using the type() Function to Determine Type

type() constructors, Type Constructors: Composite Variable Construction-Type Constructors: Composite Variable Construction

validation of, Validation of Variables-Effective Use of the validation{} Block

and custom conditions, Custom Conditions-Custom Conditions

error messages, Error Messages-Error Messages

validation{} block, use of, Effective Use of the validation{} Block

validation subblock, Parts of a Variable Definition

variability of TF configuration and, Variability of TF Configuration-Variability of TF Configuration

variable sets, Setting Input Variables

workspace variables, Setting Input Variables

integration testing, Concepts irrelevant to (or diminished for) IaC

interdependent projects, Recap of Common Problems

internal networks, Internal Networks

IP network functions, Effective Use of Other Functions (Numeric, IP Network, and Date/Time)

L

leaf modules, Leaf Modules Versus Nesting Modules

least privilege, principle of, Important Principles

length() function, The length() Function

lifecycle nested block, The lifecycle Meta-Argument and Custom Conditionals-The lifecycle Meta-Argument and Custom Conditionals

list() type constructor, Type Constructors: Composite Variable Construction

local modules, Local Versus Remote Modules-Local Versus Remote Modules, Local Modules-Local Modules

local values, Features, Local Values-Organizing Local Values

- locals{} block, The locals{} Block and the local Object
- organization of, Organizing Local Values

local workspaces, using, Using Local Workspaces-Using Local Workspaces

locals{} block, The locals{} Block and the local Object

lock timeouts, Using a Lock Timeout

locked states, Locked States

logic modules, Types of Modules

logic, effective use of (see HashiCorp Configuration Language (HCL))

logical and, Arithmetic and Logical Operators

logical operators, Arithmetic and Logical Operators-Arithmetic and Logical Operators

logical or, Arithmetic and Logical Operators

M

magnitude and modulo (arithmetic and logical operators), Arithmetic and Logical Operators

main functions, Concepts irrelevant to (or diminished for) IaC

manual updates, Dealing with Drift and Manual Updates

map() type constructor, Type Constructors: Composite Variable Construction

maturity model, TF Maturity Model-TF Maturity Model

mental model, TF Maturity Model

mergers and acquisitions, Scenario 3: merger/acquisition-Scenario 3: merger/acquisition, Special Cases: Mergers and Acquisitions-Special Cases: Mergers and Acquisitions

meta-arguments, Meta-Arguments-The provider Meta-Argument

- custom conditionals with, The lifecycle Meta-Argument and Custom Conditionals-The lifecycle Meta-Argument and Custom Conditionals
- depends_on meta-argument, Effective Use of depends_on Meta-Argument
- for_each meta-argument, Effective Use of for_each, count Meta-Arguments-Effective Use of for_each, count Meta-Arguments
- lifecycle nested block, The lifecycle Meta-Argument and Custom Conditionals-The lifecycle Meta-Argument and Custom Conditionals
- provider meta-argument, The provider Meta-Argument

minimum viable infrastructure (MVI), Minimum Viable Infrastructure-Minimum Viable Infrastructure

mixed modules, Types of Modules

module registry, Best Available Practices

modules, Features, Developing Modules-Summary

and Conway's Law, Additional Considerations-Additional Considerations

composition versus embedded, Module Composition versus Embedded Modules-Module Composition versus Embedded Modules

deprecating/removing, Deprecating and Removing Modules-Deprecating and Removing Modules

duplicating, Duplicating Modules

extracting resources into a module (full walkthrough), Extracting Resources Into a Module-Step 13

HashiCorp's standard module structure, HashiCorp's Standard Module Structure

input variables in module fields, Setting Input Variables

leaf, Leaf Modules Versus Nesting Modules

local, Local Versus Remote Modules-Local Versus Remote Modules, Local Modules-Local Modules

logic, Types of Modules, Logic Modules

mixed, Types of Modules

nested, How to Nest a Module-Dealing with Singletons

bundled solution use case, Bundled Solution Use Case-Networking

dealing with repeated resources in, Dealing with Repeated Resources in Nested Modules

leaf versus, Leaf Modules Versus Nesting Modules

mismatching frequencies of code changes, Mismatching Frequencies of Code Changes

refactoring for nested modules, How to Refactor for Nested Modules-How to Refactor for Nested Modules

singletons, Dealing with Singletons

theory on, Theory on Nested Modules

unbalanced resource relationships, Unbalanced Resource Relationships-Unbalanced Resource Relationships

when not to nest, When Not to Nest a Module

nesting, Types of Modules

policy enforcement with, Policy Enforcement with Modules

as protected interfaces, Modules as Protected Interfaces-Deprecating and Removing Modules

deprecating/removing modules, Deprecating and Removing Modules-Deprecating and Removing Modules

external distribution, External Distribution-External Distribution

internal distribution, Internal Distribution

referencing modules, Referencing Modules-Referencing Modules

versioning, Versioning-Versioning

referencing, Referencing Modules-Referencing Modules

remote, Types of Modules, Don't Own the Remote and Need an Update?

root, Root Modules

things to avoid, Things to Avoid-Things to Avoid

third-party remote, Don't Own the Remote and Need an Update?

types of, Types of Modules-Logic Modules

unitary, Types of Modules

useful practices, Common, Useful Practices-Additional Considerations

MVI (minimum viable infrastructure), Minimum Viable Infrastructure-Minimum Viable Infrastructure

N

nested modules, How to Nest a Module-Dealing with Singletons

bundled solution use case, Bundled Solution Use Case-Networking

dealing with repeated resources in, Dealing with Repeated Resources in Nested Modules

mismatching frequencies of code changes, Mismatching Frequencies of Code Changes

refactoring for, How to Refactor for Nested Modules-How to Refactor for Nested Modules

singletons, Dealing with Singletons

theory on, Theory on Nested Modules

unbalanced resource relationships, Unbalanced Resource Relationships-Unbalanced Resource Relationships

when not to nest, When Not to Nest a Module

nesting modules, Types of Modules, Leaf Modules Versus Nesting Modules

network policies, Network Policies-Edges of Cloud Networks

edges of cloud networks, Edges of Cloud Networks-Edges of Cloud Networks

internal networks, Internal Networks

VPC peering, Edges of Cloud Networks

never-clean states, Never-Clean States, Recap of Common Problems

nullable value (input variables), Parts of a Variable Definition

numeric functions, Effective Use of Other Functions (Numeric, IP Network, and Date/Time)

O

object() type constructor, Type Constructors: Composite Variable Construction

one-off operations, One-off operations

OpenSource Maturity Model (OMM), TF Maturity Model

OpenTofu plan (see also HashiCorp Configuration Language (HCL); Terraform)

developing in Terraform versus, For OpenTofu Versus HCP Terraform/Terraform Enterprise

Git repositories with, Things to Avoid

going beyond the shell with, Going Beyond the Shell-Context Is Everything

observability considerations with, Observability Considerations-Other Touch Points

optimized code, shipped code versus, The Messy Middle

organization structures and policy constraints, Organization Structures and Policy Constraints-Summary

access control, Access Control-Service Identities

authoritative versus nonauthoritative, Authoritative Versus Nonauthoritative-Authoritative Versus Nonauthoritative

policy scopes, Policy Scopes-Policy Scopes

practical considerations, Practical Access Considerations-Practical Access Considerations

service identities, Service Identities-Service Identities

account-wide policies, Account-wide Policies-Account-wide Policies

common paradoxes with, Common Paradoxes-Competing Policies

competing policies, Competing Policies

GDPR requirements, General Data Protection Regulation (GDPR)

HIPAA requirements, Health Insurance Portability and Accountability Act (HIPAA)-Health Insurance Portability and Accountability Act (HIPAA)

modules, policy enforcement with, Policy Enforcement with Modules

network policies, Network Policies-Edges of Cloud Networks

edges of cloud networks, Edges of Cloud Networks-Edges of Cloud Networks

internal networks, Internal Networks

VPC peering, Edges of Cloud Networks

principles of, Important Principles-Defense in Depth

abstraction, principle of, Important Principles

CAP theorem (Brewer's theorem), Important Principles

defense in depth, Defense in Depth-Defense in Depth

least privilege, principle of, Important Principles

Separation of Concerns (SoC), Separation of Concerns-Separation of Concerns

Single Responsibility Principle (SRP), Single Responsibility Principle

vendors, Vendors: A Double-Edged Sword or a Thornful Rose-Vendors: A Double-Edged Sword or a Thornful Rose

zero trust architecture (ZTA), Zero Trust Architecture (ZTA)

organizational change management (state optimization scenario), Scenario 4: organizational change management-Scenario 4: organizational change management

organizational scaling, Scenario 1: organizational scaling-Scenario 1: organizational scaling

output values, Features, Output Values-Validating Output Values

output{} block, The output{} Block-The output{} Block

strategic use of outputs, Strategic Use of Outputs

validation of, Validating Output Values

output{} block, The output{} Block-The output{} Block

outside help, getting, Getting Outside Help

P

-parallelism argument, Using the -parallelism Argument

pets, avoiding

in automation, Livestock, Not Pets

problem solving and, No Pets

pipelines

metrics, Recommended Dashboards and Alerts

triggering, Successful Practices

using automation to connect, Connecting Pipelines

plan command (CLI), Important concepts unique(ish) to IaC, Validation of Variables, Importing State, Scenario 1: organizational scaling, Using a Lock Timeout

policies (see organization structures and policy constraints)

postconditions, Validation Blocks in HCL

preconditions, Validation Blocks in HCL

prevent_destroy rule, The lifecycle Meta-Argument and Custom Conditionals

primitive elements, for set values, Effective Use of for_each, count Meta-Arguments

problem solving, Forever Forward-Summary

-parallelism argument, using, Using the -parallelism Argument

-replace and -target arguments, using, Using the -replace and -target Arguments

and forever forward concept, Moving Forever Forward

avoiding pets in, No Pets

better options for, A Better Way-A Better Way

common problems, Recap of Common Problems

console command, Using tf console

everything is production principle, Everything Is Production

full walkthroughs, Putting It All Together-Step 6

extracting resources into a module, Extracting Resources Into a Module-Step 13

importing untracked resources, Importing Untracked Resources-Step 6

making a reusable TF project from a static one, Making a Reusable TF Project From a Static One-Step 11

splitting a TF state into two, Splitting a TF State into Two-Step 13

getting unstuck, Getting Unstuck-Preventative Measures

duplicating modules, Duplicating Modules

environment conflicts, Environment Conflicts

preventative measures, Preventative Measures

third-party remote modules, Don't Own the Remote and Need an Update?

unexpected changes reported in specific environments, Unexpected Changes Reported In Specific Environments-Unexpected Changes Reported In Specific Environments

lock timeouts, using, Using a Lock Timeout

source control systems, On Source Control-The Flawed Merge Conflict

branch names and commit messages, Branch Names and Commit Messages

example nightmare scenario, Example Nightmare Scenario

flawed merge conflict, The Flawed Merge Conflict

gitignore, .gitignore

merging branches, Merging Branches

over-trusting branching strategies, Over-Trusting Branching Strategies

stash command, stash

terraformignore, .terraformignore

using local workspaces, Using Local Workspaces-Using Local Workspaces

programming languages, using, Successful Practices

project evolution, Evolving Projects Into Enterprise-Summary

case studies, Example Case Studies-Combining Two Ecosystems

acquired software ecosystem, Absorbing an Acquired Software Ecosystem-Absorbing an Acquired Software Ecosystem

combining two ecosystems, Combining Two Ecosystems-Combining Two Ecosystems

expanding from web app to entire organization, Expanding from a Web App to an Entire Organization-Expanding from a Web App to an Entire Organization

vibe-coded project, Working with a Vibe-Coded Project-Working with a Vibe-Coded Project

change management, Change Management for Infrastructure Support

getting outside help with, Getting Outside Help

indicators for change, Signals for Change-Signals for Change

infrastructure support, Change Management for Infrastructure Support

patterns, Evolutionary Patterns-Special Cases: Mergers and Acquisitions

building on top of base infrastructure, Building on Top of Base Infrastructure

mergers and acquisitions, Special Cases: Mergers and Acquisitions-Special Cases: Mergers and Acquisitions

state reproduction swap, State Reproduction Swap-State Reproduction Swap

Strangler Fig, Strangler Fig-Strangler Fig

provider, Important concepts unique(ish) to IaC

provider meta-argument, The provider Meta-Argument

R

reference attributes, Resource Fields and Reference Attributes

referencing, module, Referencing Modules-Referencing Modules

regional company (case study), Regional Company-Regional Company

remote modules, Types of Modules

-replace arguments, Using the -replace and -target Arguments

replace_triggered_by rule, The lifecycle Meta-Argument and Custom Conditionals

replication, state, State Replication

resource fields, Resource Fields and Reference Attributes

resource management, Important borrowed concepts, Recap of Common Problems

root modules, Root Modules, Root Module

S

SaaS scale-up (case study), SaaS Scale-up-SaaS Scale-up

safety functions, Effective Use of Type Conversion and Safety Functions-Effective Use of Type Conversion and Safety Functions

scale case studies, Different Scales-Global Firm

    global firm, Global Firm

    regional company, Regional Company-Regional Company

    SaaS scale-up, SaaS Scale-up-SaaS Scale-up

scaling, organizational, Scenario 1: organizational scaling-Scenario 1: organizational scaling

self-documenting elements, Self-Documenting Elements

self-service infrastructures, Self-Service Infrastructure-Self-Service Infrastructure

sensitive value (input variables), Parts of a Variable Definition

sensitive value functions, Effective Use of Sensitive Value Functions

Separation of Concerns (SoC), Separation of Concerns-Separation of Concerns

service identities, Service Identities-Service Identities

set() type constructor, Type Constructors: Composite Variable Construction

shell scripts, Important concepts unique(ish) to IaC, Successful Practices

shipped code, optimized code versus, The Messy Middle

Single Responsibility Principle (SRP), Single Responsibility Principle

singletons, Dealing with Singletons

software ecosystems (case studies)

acquired software ecosystem, Absorbing an Acquired Software Ecosystem-Absorbing an Acquired Software Ecosystem

combining two ecosystems, Combining Two Ecosystems-Combining Two Ecosystems

source control, The Basics, On Source Control-The Flawed Merge Conflict

branch names and commit messages, Branch Names and Commit Messages

example nightmare scenario, Example Nightmare Scenario

flawed merge conflict, The Flawed Merge Conflict

gitignore, .gitignore

merging branches, Merging Branches

over-trusting branching strategies, Over-Trusting Branching Strategies

stash command, stash

splat expressions, Splat Expressions-Splat Expressions

SRP (Single Responsibility Principle), Single Responsibility Principle

standards and style (HCL), Important borrowed concepts

stash command, stash

state address, State Address

state backend, State Backend

state object, State Object

state of states, State of States

state optimization, Optimizing State-Summary

common problems, Common State Problems-Never-Clean States

bloated states, Bloated States-Bloated States

broken states, Broken States

gridlocked states, Gridlocked States

locked states, Locked States

never-clean states, Never-Clean States

too many states, Too Many States

unbalanced states, Unbalanced States

common scenarios, Common Scenarios-Scenario 4: organizational change management

merger/acquisition, Scenario 3: merger/acquisition-Scenario 3: merger/acquisition

organizational change management, Scenario 4: organizational change management-Scenario 4: organizational change management

organizational scaling, Scenario 1: organizational scaling-Scenario 1: organizational scaling

reducing duplication, Scenario 2: reducing duplication-Scenario 2: reducing duplication

concepts and definitions, State Concepts and Definitions-Root Module

atomic states, Atomic States

drift, Drift

graph theory, A Quick Vocabulary Lesson in Graph Theory

importing state, Importing State

root module, Root Module

state address, State Address

state backend, State Backend

state object, State Object

state replication, State Replication

state triplet, State Triplet

trimming and propagation, State Trimming and Propagation

don'ts with, State Don'ts-ClickOps

ClickOps, ClickOps

one-off operations, One-off operations

overusing the -target option, Overusing the -target option

tightly coupled states, Tightly coupled states-Tightly coupled states

too many cooks, Too many cooks

uneven state, Uneven state

dos with, State Dos-Lifecycle-aligned code modules

balanced states, Balanced states

cleaning up unused states, Clean up unused states

domain-aligned code modules, Domain-aligned code modules

environment-alleged state objects, Environment-aligned state objects

lifecycle-aligned code modules, Lifecycle-aligned code modules

multiple-state coordination, Coordinating Multiple States-State of States

domino effect, Domino Effect-Domino Effect

state of states, State of States

tracking dependencies, Tracking Dependencies and State

tactical approaches, Common Approaches-Approach 5: wildcard jobs

backend key variation, Approach 1: backend key variation

graduated environment configuration, Approach 3: graduated environment configuration

specifying distinct variable files, Approach 2: specifying distinct variable files-Approach 2: specifying distinct variable files

versioned environment modules, Approach 4: versioned environment modules

wildcard jobs, Approach 5: wildcard jobs

state propagation, State Trimming and Propagation

state replication, State Replication

state reproduction swap, State Reproduction Swap-State Reproduction Swap

state trimming, State Trimming and Propagation

state triplet, State Triplet

state triplet theory, State Triplet Theory

static code analysis, Static Code Analysis-Static Code Analysis

stored state, Important concepts unique(ish) to IaC

Strangler Fig, Strangler Fig-Strangler Fig

string functions, Effective Use of String Functions-Effective Use of String Functions

string interpolation and directives, String Interpolation and Directives-String Interpolation and Directives

T

-target arguments, Using the -replace and -target Arguments

-target option, overusing the, Overusing the -target option

task automation, The Basics, For OpenTofu Versus HCP Terraform/Terraform Enterprise

technical debt, Technical IaC Debt-The Real Cost of Technical Debt

  nightmare scenarios, Nightmare Scenarios-Nightmare Scenarios

  paying it down, Paying It Down-Paying It Down

  real cost of, The Real Cost of Technical Debt

temporary access controls, Successful Practices

terminal/shell, The Basics

ternary conditional expressions, Ternary Conditional Expressions

Terraform, Bad 'Form!

  (see also HashiCorp Configuration Language (HCL); OpenTofu)

  alternative IaC tools, TF Versus Others-TF Versus Others

  automated logic outside of, Automated Logic Outside of TF

  basic tools and services, Recommended Tools and Services-For OpenTofu Versus HCP Terraform/Terraform Enterprise

  common misconceptions with, Common Misconceptions-Common Misconceptions

developing in OpenTofu versus, For OpenTofu Versus HCP Terraform/Terraform Enterprise

features, Features

implementing, Bad 'Form!

integration testing, Concepts irrelevant to (or diminished for) IaC

maturity model, TF Maturity Model-TF Maturity Model

mental model, TF Maturity Model

technical debt with, Technical IaC Debt-The Real Cost of Technical Debt

unit testing, Concepts irrelevant to (or diminished for) IaC

terraformignore, .terraformignore

test command (CLI), Behavior Analysis-Behavior Analysis

testing

integration, Concepts irrelevant to (or diminished for) IaC

unit, Concepts irrelevant to (or diminished for) IaC

validation at scale, On Testing-On Testing

.tfvars files, Setting Input Variables, Approach 2: specifying distinct variable files

tightly coupled states, Tightly coupled states-Tightly coupled states

tracking dependencies, handling, Tracking Dependencies and State

tuple() type constructor, Type Constructors: Composite Variable Construction

type conversion functions, Effective Use of Type Conversion and Safety Functions-Effective Use of Type Conversion and Safety Functions

U

unbalanced resource relationships, Unbalanced Resource Relationships-Unbalanced Resource Relationships

unbalanced states, Unbalanced States

uneven state, Uneven state

unit testing, Concepts irrelevant to (or diminished for) IaC

unitary modules, Types of Modules

untracked resources, importing, Importing Untracked Resources-Step 6

updates, manual, Dealing with Drift and Manual Updates

V

validate command (CLI), Static Code Analysis, Low-Tech, Low-Cost Solutions

validation at scale, Validating at Scale-Summary

- and testing, On Testing-On Testing
- behavior analysis, Behavior Analysis-Behavior Analysis
- data sources, Data Sources-Data Sources
- low-cost solutions, Low-Tech, Low-Cost Solutions-Low-Tech, Low-Cost Solutions
- observability considerations with, Observability Considerations-Other Touch Points
  - autonomous agents, Other Touch Points
  - recommended dashboards/alerts, Recommended Dashboards and Alerts

strategies, Strategies

policy enforcement with modules, Policy Enforcement with Modules

static code analysis, Static Code Analysis-Static Code Analysis

types of, Types of Validation in TF-Types of Validation in TF

validation{} blocks, Validation Blocks in HCL-Validation Blocks in HCL

validation subblocks, Parts of a Variable Definition

validation{} blocks, Effective Use of the validation{} Block, Validation Blocks in HCL-Validation Blocks in HCL

variables, Features

vendors, working with, Vendors: A Double-Edged Sword or a Thornful Rose-Vendors: A Double-Edged Sword or a Thornful Rose

versioning, module, Versioning-Versioning, Approach 4: versioned environment modules

vibe-coded project (case study), Working with a Vibe-Coded Project-Working with a Vibe-Coded Project

VPC peering, Edges of Cloud Networks

W

web app, expanding to entire organization (case study), Expanding from a Web App to an Entire Organization-Expanding from a Web App to an Entire Organization

webhooks, Successful Practices

wildcard jobs, Approach 5: wildcard jobs, Wildcard Jobs

workspace management, For OpenTofu Versus HCP Terraform/Terraform Enterprise

Z

zero trust architecture (ZTA), Zero Trust Architecture (ZTA)

About the Author

Robert Glenn has nearly a decade of experience working with Terraform and OpenTofu as a consulting manager for Accenture. He has built resource bundles as remote modules for internal consumption, reusable solution templates, and entire cloud foundational infrastructures from the ground up. He is currently CTO of Scale the Vibe, where he leverages his extensive experience to turn vibe-coded solutions into enterprise-grade products, and CEO of Kopacetic, developing AI-driven software for improving virtual communication. He also helped found the Cartografos working group with the Cloud Native Computing Foundation (CNCF), and he is an active contributor to their primary asset, the Cloud Native Maturity Model (the premier guide for determining readiness and completeness of cloud implementations for enterprise organizations). In addition to his professional work and collaboration with the CNCF, Robert has a technical blog on Medium on which he shares his unconventional technical perspective: *https://medium.com/devoops-discourse*.

Colophon

The animal on the cover of *Terraform at Scale* is a European pond turtle (*Emys orbicularis*). Found throughout southern, central, and eastern Europe, West Asia, and Mediterranean North Africa, this freshwater turtle is known for its long lifespan, living up to 50 years or more.

The European pond turtle is a medium-sized turtle: its shell length ranges from 4.7 to 15 inches, depending on its geographic location. The color of its carapace (outer shell) ranges from dark brown to black, with tinges of green; its skin, while also dark, contains small yellow spots. Adult males are usually smaller than females; this makes them more agile, which helps them travel farther and faster during breeding season to find a mate.

European pond turtles are diurnal and spend much of their time in the sun to regulate their body temperature. They are strong swimmers and can hold their breath for long periods. Their diet consists of insects, mollusks, small fish, and plant materials like algae. These turtles primarily inhabit freshwater environments, including ponds, lakes, and marshes, and tend to prefer areas with abundant aquatic vegetation, which provides food and shelter from predators.

Currently, European pond turtles are classified as Near Threatened by the IUCN Red List. Habitat loss due to urbanization and pollution has had a significant impact on their population; conservation efforts must focus on habitat restoration and cleaner waters to ensure their survival.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world.

The cover illustration is by José Marzan Jr., based on an antique line engraving from *Lydekker's Royal Natural History*. The series design is by Edie Freedman, Ellie Volckhausen, and Karen Montgomery. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.