Reverse Engineering with Terraform

An Introduction to Infrastructure Automation, Integration, and Scalability using Terraform

Sumit Bhatia
Chetan Gabhane

Apress®

Reverse Engineering with Terraform

An Introduction to Infrastructure Automation, Integration, and Scalability using Terraform

Sumit Bhatia
Chetan Gabhane

Apress®

Reverse Engineering with Terraform: An Introduction to Infrastructure Automation, Integration, and Scalability using Terraform

Sumit Bhatia
Houston, TX, USA

Chetan Gabhane
Pune, Maharashtra, India

ISBN-13 (pbk): 979-8-8688-0073-3
ISBN-13 (electronic): 979-8-8688-0074-0
https://doi.org/10.1007/979-8-8688-0074-0

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: James Robinson-Prior
Development Editor: James Markham
Editorial Assistant: Gryffin Winkler

Cover designed by eStudioCalamar

Cover image by 5598375 on pixabay.com

Distributed to the book trade worldwide by Springer Science+Business Media New York, 1 New York Plaza, Suite 4600, New York, NY 10004-1562, USA. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub. For more detailed information, please visit https://www.apress.com/gp/services/source-code.

Paper in this product is recyclable

"The price of success is hard work, dedication to the job at hand, and the determination that whether we win or lose, we have applied the best of ourselves to the task at hand."

—Vince Lombardi

This book is dedicated to everyone who has ever been told THEY CAN'T.

Table of Contents

About the Authors

Lead author **Sumit Bhatia** is a global infrastructure solutions architect with a focus on addressing difficult business challenges and encouraging green IT through intelligent hybrid cloud strategies, advanced automation, and cost savings in IT operations to support the global oil and gas industry. With more than 15 years in the tech industry, Sumit is driving technology innovations by employing his deep knowledge of automation tactics, DevOps practices, multicloud, and edge computing solutions engineering. With the help of his world-class inventions with Terraform, the introduction of capacity management as service (CMaaS) in hybrid deployments, and cost reductions with cutting-edge on/off solutions, his firm was able to cut their yearly spending by hundreds of thousands of dollars. He received the highest honor from his organization, an award for being a pioneer in the field of energy innovation.

Sumit is also a well-known author and reviewer for leading oil and gas industry journals and publications. His steadfast drive to share information and best practices has aided many professionals in navigating the complicated realm of sustainability and solutions engineering.

Chetan Gabhane is a seasoned expert with more than 15 years of experience as a solution architect and senior technical consultant in the field of hybrid and multicloud computing. His major focus is on developing resilient and efficient hybrid cloud architectures that combine the best of on-premises and cloud-based technologies. Chetan has extensive experience adopting cutting-edge DevOps approaches to improve the agility and scalability of cloud infrastructures. Chetan actively contributes to the creation of cloud solutions' reference architecture, white papers, and tech blogs. His commitment to sharing information and best practices has assisted numerous professionals in navigating the complex world of cloud computing with an unwavering dedication to excellence.

About the Technical Reviewer

Simon Mansbridge has over 35 years of IT experience in the development and deployment of large enterprise systems, often supporting over 100,000 users. He has worked on a range of IT technologies from high-performance computing to core infrastructures, SAP R/3, Microsoft enterprise communication and collaboration systems, cloud, security tooling, and service management. As a technical manager, one of his focuses has always been on efficient optimized solutions delivered via combinations of commercial and in-house developed monitoring and automation tools. His technical insights and innovations have led to some being implemented as, or incorporated into, commercial products.

With his broad and deep knowledge, Simon now acts as Chief Architect, ensuring alignment across the broad range of technologies in the organization he currently works for.

Acknowledgments

This book is the brainchild of two enthusiastic individuals who have had the privilege of growing and thriving as engineers for more than a decade. We started our journey together, and so far, it has been well enriched by the guidance of wonderful world-class leaders, managers, and technology enthusiasts. It is with a deep sense of responsibility that we offer this book to give back to our community of technology and automation enthusiasts who are working hard every day. It is our hope that the knowledge contained within this book can contribute, in some measure, to the collective growth and advancement of our ever-evolving industry. These acknowledgments are a small tribute to the community that has nurtured our passion and allowed us to realize the potential of engineering in all its facets.

We would like to express our gratitude and love to all our family and friends who appreciate and encourage us always.

We would also like to thank Apress for their unwavering belief in our proposal and working with us so diligently in completing and presenting this book.

Finally, we would like to thank all our cherished readers for taking the time to read this book.

Introduction

The idea of writing this book hit us when we were discussing how we should give back to the DevOps community. This book focuses on the need of ZeroOps and how Terraform can enable IT infrastructure architects, infrastructure solutions experts, administrators, and technical managers to achieve the ZeroOps goal.

What is ZeroOps? ZeroOps (or "zero operations") is a concept in IT where the day-to-day operations are so automated and abstracted that there is no need for a dedicated team to manage the infrastructure in-house. The concept of ZeroOps demands synergy across diverse technologies and platforms employed in any typical IT environment. Most important, it requires the use of a common platform that can support diverse integrations, implement security, and ease development, adoption, and management.

With advancements in infrastructure and automation technologies, there has been increasing demand to automate core infrastructure operations. Modern-day DevOps enthusiasts do not necessarily automate every single infrastructure-related job they do, but they do automate most of their level 1 (L1) and level 2 (L2) tasks. Basically, ZeroOps shifts the focus of the core operation teams, moving from doing the monotonous, boring, and repetitive work of handling L1 and L2 tickets to doing more advanced activities where they are involved in writing code and defining and managing their infrastructure as code (IaC). In essence, ZeroOps provides self-service to the end users so they can do basic L1 and L2 tasks themselves.

In a traditional scenario, IT users have infrastructure requirements for which they approach the specialized administrator, who then works on the request and provides the needed support to the end user. In the process of fulfilling the user's request, the back-end IT teams are required to run a set of repetitive steps, which can be boring and tedious to the administrator. To modernize this scenario, power is given directly to the end user (called *self-service*) to fulfil their own needs.

There are numerous benefits to empowering end users and making them responsible for basic infrastructure tasks and in turn less dependent on the back-end operations team. These are the main benefits:

- **Saved time:** End users are not bound by ticketing service-level agreements and can get the desired outcomes almost immediately.
- **Lower costs:** Organizations now need fewer employees to do the redundant, manual activities.
- **Improved security:** Least privilege access and role-based access control have enabled DevOps practices to be more secure than anybody doing it manually.
- **Fewer errors:** There is less opportunity for manual human errors.

These are areas where Terraform has a unique presence in infrastructure automation practices. Terraform has enabled infrastructure admins to define their key infrastructure entities as code that is easy to write and manage. Terraform has built-in integrations with numerous popular public cloud platforms such as AWS, Azure, and GCP, as well as on-prem technologies such as VMware, Hyper-V, etc. With help from Terraform, infrastructure admins can now easily provision and manage their infrastructure.

Terraform provides great integration and infrastructure automation facilities for the resources that started their life cycles with Terraform. However, it poses challenges to IT administrators because they have legacy infrastructure entities that were not provisioned with Terraform and that sometimes comprise a majority of their infrastructure footprints. For example, because of how Terraform operates, it needs to have a "state file," which is created when IT administrators provision the resources first with Terraform. However, when the infrastructure was not provisioned with Terraform first, the tool does not know the current state of the resource. Terraform has provided ways to import the current state of a resource, but it is a manual process where IT admins must write a "config file" for each resource and then import the files into Terraform. Manually importing each resource is a cumbersome job for IT administrators, and it is made more complex when there are thousands of infrastructure resources. The legacy resources that are deployed and running are not going anywhere in the near term. In those scenarios, using Terraform is challenging, especially with objectives such as ZeroOps.

We were in this situation some time ago with our VMware infrastructure, where we were looking for opportunities to implement automation to manage thousands of workloads already running in our data centers with a centralized tool (Terraform). This is where the idea of *Reverse Engineering with Terraform* originated. We were able to automatically import existing resources into Terraform and import them at scale. The reverse engineering piece that was written did the job on most of our already provisioned resources. Once the current infrastructure was imported, it was easy to manage the life cycle of the resources via Terraform.

There is another challenge with Terraform where if a resource is provisioned with it, then the entire life cycle can be managed with Terraform only. If an admin changes the resource directly on the platform, then that will invalidate the existing state that Terraform has in its record,

thus making it more complex for administrators who want to manage their infrastructure automatically. With the reverse engineering that we cover in this book, you are able to eliminate the need for a persistent state file for each resource. The logic creates a fresh state every time for operations that we want Terraform to perform automatically. For example, if any administrator is doing any changes on the infrastructure manually and directly on the platform, our reverse engineering automation enables you to just scrap the old state file and do a fresh and reliable import of the state every time you need to do L1 or L2 tasks with the infrastructure.

Throughout this book, we will use VMware as a reference and show how reverse engineering has helped overcome the limitations of Terraform. In addition, we incorporate a hands-on lab in the book to provide better illustrations of the content. The book focuses on how Terraform is employed in our case as a common platform for achieving ZeroOps on diverse platform we manage (cloud and on-prem). The good news is that the practices that we have discovered can be easily transferred to other platforms where Terraform has built-in integration.

We welcome you on this journey and hope this book helps you to meet your organizational objectives.

CHAPTER 1

Terraform: Infrastructure as Code

In this chapter, we embark on a journey through the realm of modern IT and infrastructure as code (IaC). We will discover the tremendous importance of IaC in today's technology landscape. Terraform, the powerful instrument that is transforming the face of IT infrastructure management, will be the focus of our attention. We will delve into the core aspects that set Terraform apart and make it an industry game-changer. We will begin by exploring the tangible use cases that demonstrate Terraform's power in orchestrating IT environments with unparalleled agility and efficiency.

We will also take a closer look at the two fundamental approaches to IaC: declarative and imperative. Understanding the differences between these methods is key to harnessing Terraform's full potential. One of the cornerstones of Terraform's capabilities lies in its providers and provisioners. We will demystify these concepts, shedding light on how they empower you to interact with diverse infrastructure platforms and fine-tune resources to your exact specifications. We will also compare the open-source version of Terraform with HashiCorp's version, helping you navigate the choices available.

S. Bhatia and C. Gabhane, *Reverse Engineering with Terraform*,
https://doi.org/10.1007/979-8-8688-0074-0_1

Additionally, we'll uncover how Terraform acts as a bridge connecting infrastructure as a service (IaaS) and platform as a service (PaaS), adapting to the unique resource provisioning requirements that organizations encounter in both on-premises and cloud deployments.

Our exploration culminates in a hands-on exercise that guides you through the installation of the open-source version of Terraform on Ubuntu, specifically tailored for VMware infrastructure. This practical exercise is designed to provide you with the tools and skills you need to embark on your own Terraform journey.

Prepare to embrace the power of Terraform and elevate your infrastructure management to new heights.

Infrastructure as Code: A Key Enabler for Today's Technology Landscape

Traditional approaches to infrastructure management fall short in today's dynamic world of modern technologies, where agility, scalability, and dependability are critical. As organizations strive to meet the ever-changing needs of software development and operations, IaC has emerged. In this chapter, we'll look at what IaC is and why it has become such an important tool in today's fast-paced environment.

IaC has evolved from a paradigm shift in the management and deployment of infrastructure resources. Traditionally, infrastructure setup and maintenance were manual operations that were time-consuming, error-prone, and difficult to repeat reliably across settings. However, by treating infrastructure as software, IaC allows organizations to take advantage of the potential of automation, version control, and repeatability. So, IaC is a way of deploying your infrastructure in an automated way using code instead of manual processes.

Since many components of the infrastructure may now be specified as code, IaC abstracts away all of the infrastructure's intricacies. It may also be thought of as a layer that sits between the DevOps world and the IT infrastructure, abstracting all the underlying intricacies and surfacing just those components that are absolutely necessary for effective infrastructure management.

At its core, IaC involves setting up and provisioning infrastructure resources using machine-readable configuration files. These files, which are often composed in a domain-specific language (DSL) or using declarative syntax, serve as blueprints for the creation and management of infrastructure components such as servers, networks, storage, and other resources. It further allows teams to utilize existing software development practices and tools by modeling infrastructure as code, resulting in a more efficient and scalable infrastructure management process.

IaC's transformational strength stems from its ability to apply standard software development practices to infrastructure management. Let's look at some of the important features that make IaC such a game-changer:

- **Standardization and consistency:** IaC defines and provisions infrastructure using configuration files that may be versioned and saved in a version control system. This approach ensures that infrastructure configuration is standardized and consistent across environments, decreasing the risk of configuration drift and making infrastructure management and maintenance easier at scale.
- **Reproducibility:** IaC enables organizations to reliably re-create infrastructure configurations because infrastructure is described in code. It is feasible to spin up similar environments with a few instructions, removing the need for manual procedures and lowering the likelihood of human mistake. This

repeatability is especially useful in circumstances such as testing, development, and disaster recovery, where consistent and repeatable infrastructure settings are essential.

- **Automation and efficiency:** IaC enables infrastructure provisioning and administration processes to be automated. Programmatic provisioning, configuration, and deployment of infrastructure reduce the need for manual operations. This automation improves efficiency by saving time and effort while reducing the possibility of human mistakes that might arise while performing repetitive operations.
- **Collaboration and DevOps culture:** The siloed nature of traditional infrastructure management often slows down collaboration between development and operations teams. IaC breaks down those barriers and encourages the collaborative DevOps culture. Teams may efficiently cooperate on infrastructure settings by utilizing version control systems. Multiple team members may work on various elements of the infrastructure codebase at the same time, making it simpler to manage changes, monitor modifications, and revert to earlier settings if necessary. This strategy increases team cooperation, improves code reviews, and assists team members in exchanging information and deploying the code effortlessly in all three stages of development, testing, and production.
- **Testing and continuous integration:** IaC encourages the use of automated testing practices for infrastructure code. Infrastructure code, like software code, may be tested using unit tests, integration tests, and

acceptance tests. This helps teams to detect faults and misconfigurations early in the development cycle, boosting overall infrastructure quality and dependability. Furthermore, IaC seamlessly interfaces with continuous integration and continuous deployment (CI/CD) pipelines, allowing for the automated testing and deployment of infrastructure modifications.

- **Tools and technologies:** To implement IaC, organizations can leverage a range of tools and technologies such as Terraform, AWS CloudFormation, Azure Resource Manager, or Google Cloud Deployment Manager. These tools provide the means to define infrastructure configurations in a declarative manner, enabling automation and orchestrations.
- **Scalability and flexibility:** The capacity to scale infrastructure resources up and down fast becomes critical as organizations embrace cloud computing and dynamic infrastructure settings. IaC makes this scalability possible by allowing teams to create infrastructure in a modular and flexible manner. Infrastructure code is readily adjusted and expanded to fit changing requirements, making it easier to adapt and grow infrastructure settings as business demands change.
- **Auditing and compliance:** Regulatory requirements and security concerns are continuously growing in importance. IaC offers the advantage of traceability and auditability. Infrastructure configurations, including security and compliance standards, can

be documented and reviewed over time, supporting compliance with the regulatory framework. Figure 1-1 shows the infrastructure as code paradigm.

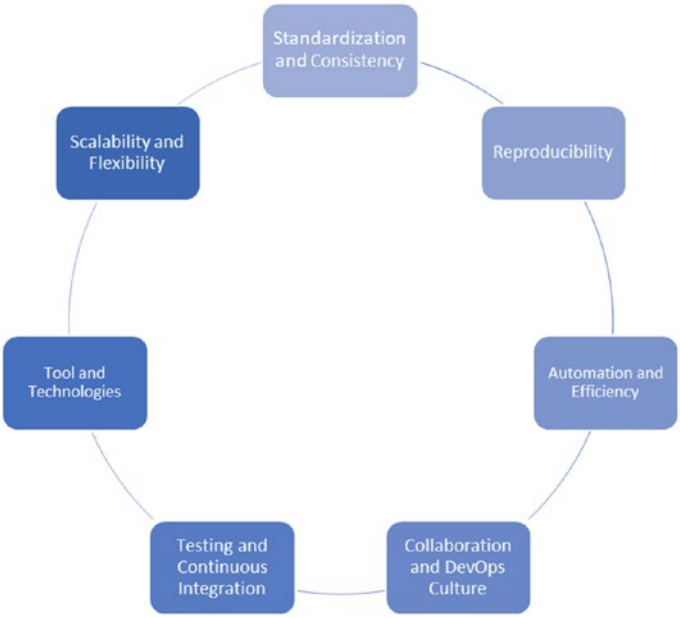


Figure 1-1. *The infrastructure as code paradigm*

Revolutionizing IT Infrastructure with the Power of Terraform

IT infrastructure is the IT world's backbone, allowing any industry to be digitized. It is composed of numerous layers, starting with storage and then networking, computing, operating systems, databases, and lastly applications. Figure 1-2 shows the pyramid structure that distinguishes IT infrastructure.

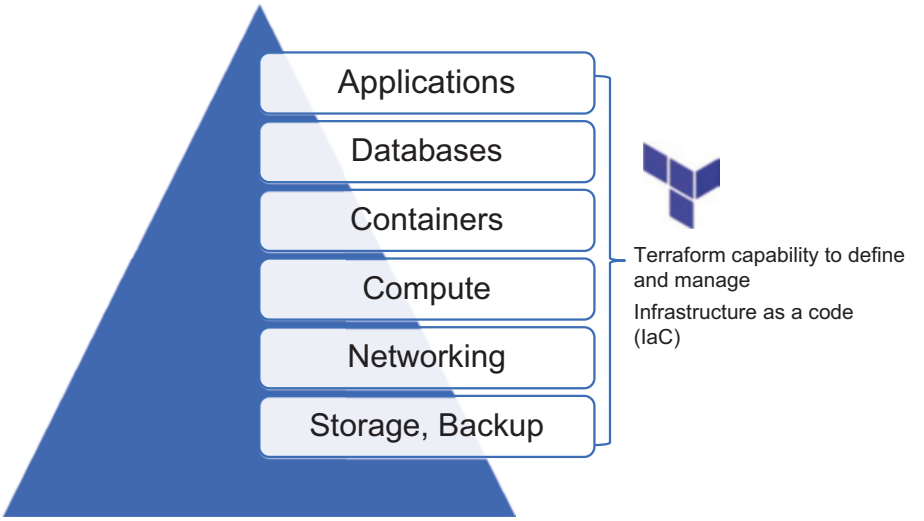


Figure 1-2. *IT infrastructure pyramid*

If organizations are using traditional ways of managing the infrastructure, they will face numerous issues. These issues include inconsistent environments, manual and error-prone processes, lack of workflow automation and reusability, lack of collaboration and documentation, lack of scalability with speed and accuracy, and high price of deployment.

In most situations, these are common infrastructure difficulties encountered by organizations that use a variety of products offered by various technology vendors. These challenges are faced by many organizations regardless of size or sector. Infrastructure management can be a difficult and time-consuming process that frequently necessitates substantial resources and experience. However, with the rise of cloud computing and DevOps practices, new tools and technologies have evolved to assist organizations in more efficiently managing their infrastructure.

Let's look at some of the most crucial infrastructure difficulties that organizations are experiencing nowadays and how Terraform might assist in addressing these challenges.

From Silos to Harmony

One of the biggest infrastructure challenges faced by organizations is the siloed nature of their legacy infrastructure. Few companies are still using a legacy infrastructure despite its outdated nature. Reasons could include cost, risk and stability, regulatory and compliance requirements, business continuity, etc. Companies should carefully evaluate the benefits and drawbacks of maintaining legacy systems and consider a long-term strategy for modernization when feasible.

They should also consider that the siloed structure can make it difficult to collaborate and share resources across teams, leading to inefficiencies and higher costs.

Terraform can help address this challenge by providing a single, unified platform for managing infrastructure across teams and platforms. Using the Terraform platform, teams can collaborate on infrastructure code and share resources across departments, improving efficiency and reducing costs.

Embarking Diverse Infrastructure Platforms at Scale

In this modern business landscape, a major infrastructure challenge is managing diverse infrastructure platforms at scale. As organizations adopt multiple cloud providers, embarking on multiple cloud journeys with some on-premises infrastructure platforms, it can be difficult to manage and maintain consistency across all platforms.

Here as well, Terraform provides a common technique for governing diverse infrastructure platforms at scale. With Terraform, organizations can use a single codebase to manage their infrastructure, regardless of the underlying platform. This simplifies management and reduces the risk of errors and inconsistencies in your code.

Navigating Different Technologies

Organizations often use a variety of technologies in different spaces (for example compute, storage, networking, databases) to manage their infrastructure efficiently, which often can be challenging to maintain and integrate with the existing legacy infrastructure.

Terraform provides a solution to this challenge by supporting a wide range of technologies and infrastructure platforms, including public and private clouds, containers, and on-premises infrastructure. With Terraform, organizations can use a single tool to manage all their infrastructure, regardless of the underlying technology.

Operational Costs

The cost of managing infrastructure can be a significant challenge as well for organizations. As infrastructure becomes more complex and distributed, the cost of managing it can quickly spiral out of control, which we refer to as the *operational cost* for your infrastructure. Companies should also consider this factor while managing the infrastructure for a longer duration.

Terraform can help address this challenge by providing a single platform for managing infrastructure. With Terraform, organizations can use a single codebase to manage their infrastructure, reducing the time and cost required to manage it.

Ensuring Robust Security

Security is a paramount concern for organizations managing infrastructure. As infrastructure becomes more distributed, the risk of security breaches and vulnerabilities are on rise. To address the security concern, one should implement the robust security measures in place to avoid potential risk associated.

Terraform provides an answer to this problem by providing a secure infrastructure management platform. Terraform gives organizations a safe and auditable way of managing their infrastructure code so that it reduces the risk of security breaches or vulnerabilities.

Organizations may handle some of their biggest infrastructure concerns by utilizing Terraform and its key capabilities. For example, it can help organizations to reduce the complexity of infrastructure management by allowing organizations to describe their infrastructure in a consistent and repeatable manner, decreasing the chance of errors and ensuring that the infrastructure is always in a known steady and stable state. Terraform also automates many of the processes associated with infrastructure management, decreasing the time and resources necessary to maintain infrastructure. Furthermore, Terraform enables organizations to effortlessly scale up or down their infrastructure to meet changing business demands. This is particularly essential in today's fast-changing technological landscape, and every organization must at least attempt it.

Imperative vs. Declarative Approaches to IaC

A framework for IaC can be created and deployed using one of two DevOps paradigms: declarative or imperative. Let's understand how they differ. The user is responsible under the imperative method for identifying the specific actions required to reach an end objective. This implies that the user must specify the software installation instructions, database setup, and configuration, among other things. While the execution of these phases is totally automated, the outcomes of an operation are decided by the user-specified specification and execution order. Figures 1-3 and 1-4 show the imperative versus declarative approach.

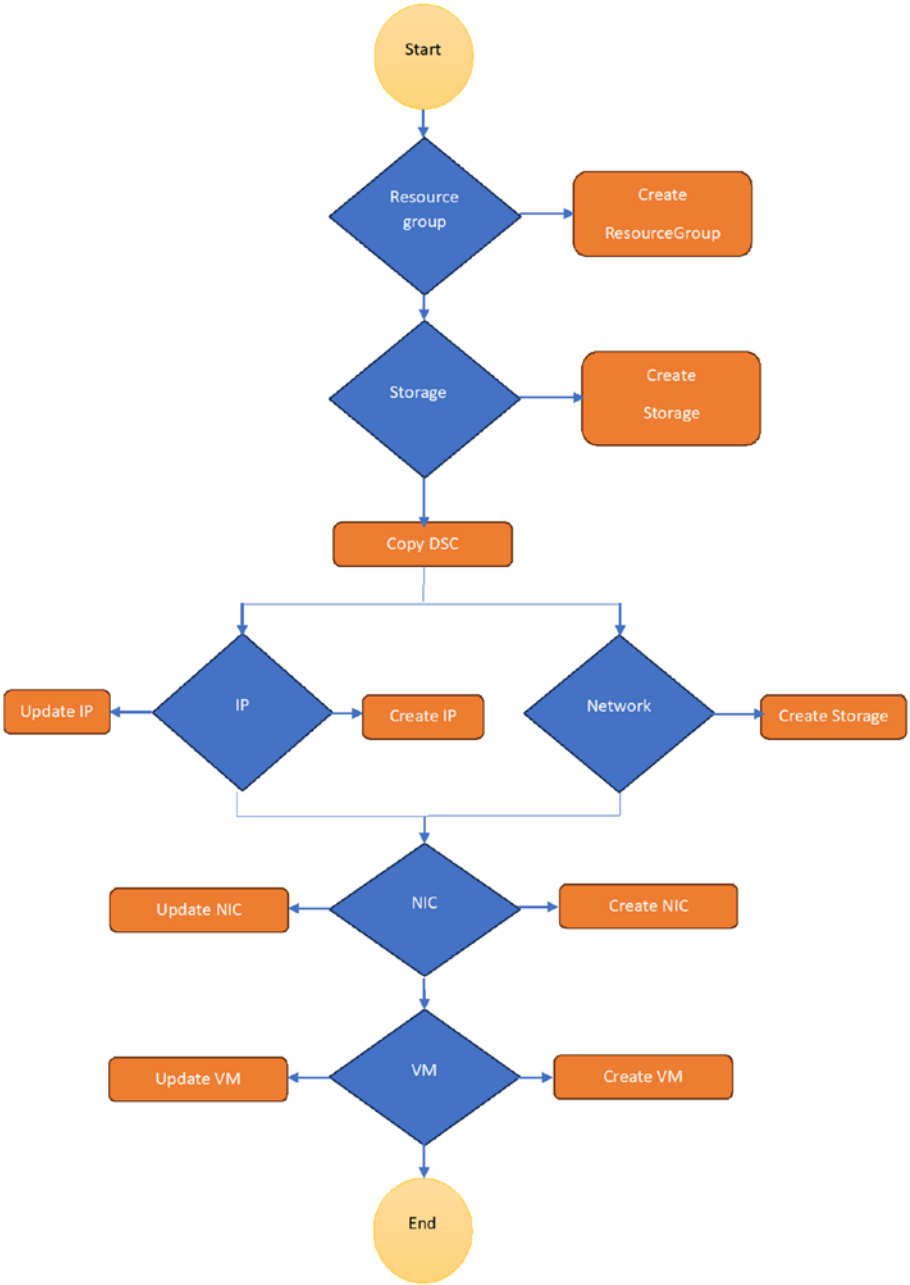


Figure 1-3. *Imperative approach*

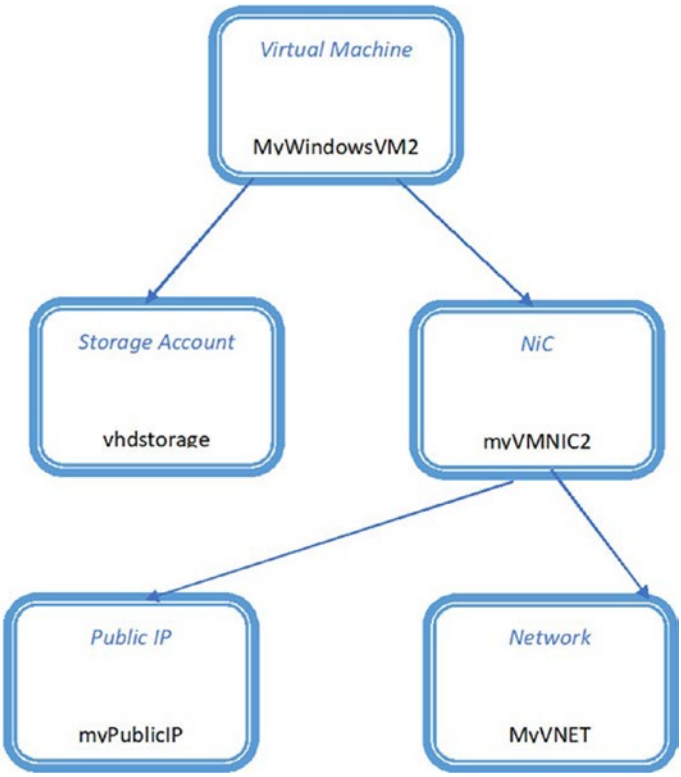


Figure 1-4. *Declarative approach (source: https://techcommunity.microsoft.com/)*

In contrast, a declarative method concentrates on specifying the eventual state rather than the specific stages. A user specifies—or declares—the number of workloads to be containerized or virtualized, the applications and machines that must be deployed, and the configuration of each. However, you do not need to be concerned with the specific actions required to accomplish these operations. A code is performed to complete the actions required to attain the user-specified end state. To gain a clear understanding of the imperative versus declarative approach, I recommend referring to Table 1-1 for a comprehensive comparison.

Table 1-1. *Imperative vs. Declarative*

Topic	Imperative	Declarative
Coding skill	Requires extensive expertise in programming	Requires minimal coding skills
Repeatability	Idempotent	Highly repeatable
Error-prone	Prone to errors due to its explicit nature	Effectively accommodates changes in configuration over time
Control	Complete control over each stage of the process	Less control over the process
Task execution	Ideal for simple tasks	Can over-complicate simple tasks
Approach	Adhere to a familiar sequential methodology	Does not follow step-by-step approach

Unleashing the Power of Terraform Providers and Provisioners

Now that we know the fundamentals of Terraform and IaC, we'll provide an overview of Terraform providers and provisioners. These serve as a foundation of Terraform's current and extensive offerings.

Terraform Provider

Terraform relies on different plugins for its interactions using an API for different IT infrastructure platforms. These plugins are called Terraform *providers*. Terraform providers serve as the bridge between Terraform and the underlying infrastructure. They act as plugins that enable Terraform to communicate with and manage resources in different cloud providers,

on-premises systems, or third-party services. Providers define a set of resources, data sources, and configurations specific to a particular platform or service.

There are hundreds of providers currently available that can be used with Terraform, making it a hugely versatile tool. Today Terraform's popularity stems from the fact that it is *platform agnostic* and can be used so widely, as opposed to languages and tools that are platform specific. Examples of such platform-specific tools are Microsoft Azure ARM templates that work only with the Microsoft Azure public cloud.

Each provider is released independently from Terraform itself, and each version offers additional features and bug fixes. Terraform providers are usually managed by HashiCorp, by third-party companies that release the plugins, or by community groups, users, and volunteers with an interest in the product or platform the provider utilizes.

Here is the list of providers currently offered by HashiCorp:
`https://registry.Terraform.io/browse/providers`

Terraform Provisioners

Terraform *provisioners* are powerful features that allow you to perform additional actions on resources during or after their creation. They enable you to execute scripts or commands on the provisioned resources to perform tasks such as configuration management, software installation, or any custom actions required to set up the infrastructure. While providers focus on resource creation and management, provisioners handle the post-creation tasks.

Provisioners can be used, for example, to update the package repository, install the nginx web server, and start the nginx service on the newly created VM. These commands are executed remotely on the virtual machine using SSH.

Terraform provisioners support various types, including local-exec (run on the machine running Terraform), remote-exec (run on the resource being provisioned), and file (upload files to the resource). They provide flexibility in configuring and customizing resources to meet specific requirements. It's important to note that while provisioners offer convenient ways to execute additional actions, they should be used judiciously. It's recommended to leverage configuration management tools such as Salt Stack, Ansible, or Chef for complex and idempotent configurations, rather than relying solely on provisioners.

We are going to provide more detailed explanations about Terraform providers and provisioners in subsequent chapters.

Terraform Open Source vs. HashiCorp's Version of Terraform

HashiCorp's Terraform is an open-source IaC software solution. It enables administrators to build infrastructure using a declarative configuration language known as the HashiCorp Configuration Language (HCL), or alternatively JSON, and manage their infrastructure primarily through APIs and automation.

Several organizations are turning to Terraform as their go-to tool for managing their cloud infrastructure as the need for IaC grows. However, with two unique versions of Terraform available—open-source and HashiCorp—the question remains: which best matches your organization? Let's consider both the urgency of the task and the cost.

On one hand, the Terraform open-source version provides a lot of freedom and customizability. It is incredibly extendable as an open-source application, allowing users to develop their own plugins and modules to meet their individual needs. Moreover, a vast community of volunteers are regularly updating the open-source version, promising that it is up-to-date with the latest platform provider APIs and features.

The HashiCorp version of Terraform, on the other hand, offers a more streamlined user experience. It eliminates the need of consumers to come up with adaptations of their own with a complete collection of built-in capabilities, making it a compelling choice for those who prefer a more out-of-the-box solution. Furthermore, the HashiCorp version has enterprise-level capabilities such as role-based access control (RBAC) and support for dozens of workspaces, making it a better match for bigger organizations with challenging infrastructure needs.

Refer to Table 1-2 to see the differences between Terraform's open-source and HashiCorp versions.

Table 1-2. *Open Source vs. HashiCorp*

Topic	Open-Source Terraform	HashiCorp Version of Terraform
Remotely managed	Run your Terraform files locally with a remote back end or your central server to manage your Terraform runs.	Nothing to manage; the Terraform managed infrastructure is used to run your Terraform scripts.
Team management	Use Active Directory to provide teams and users with credentials to access the centralized server.	The access control model is split into three units: users, teams, and organizations.
Workspace management	You can reuse the same code across multiple environments.	It gives a dashboard with the run statuses, when the Terraform files were last changed, and which repo it is connected to.
Version control system	Still, you must use your repositories and manual way of handling it to pull the latest code to production.	The setup for GitHub, GitLab, and Bitbucket is automated.

(*continued*)

Table 1-2. (*continued*)

Topic	Open-Source Terraform	HashiCorp Version of Terraform
Secure variable management	There is no secure variable management out of the box with open source.	In the workspace's variable section you can choose what variable is sensitive and what isn't. They manage everything for you.
Remote runs and State	You store your state file in your common storage repository and run your Terraform plan and apply commands.	You are provided with a dedicated page with a history and queue of plans run by a specific user.
Private module registry	It's up to you how you want to organize your modules. You can create a dedicated repository for modules, which is the better option, or have unorganized modules in different repositories.	Its private module registry is extremely effective, with the ability to share it across your organization. It supports module versioning and searching and filtering of available modules similar to Terraform's own public Terraform registry.
Configuration designer	Manually add modules and components to your Terraform `main.tf` file.	Using the configuration designer, you are able to spin up an entire workspace using your private module registry.
Dashboard API	Not supported.	You can access items on the dashboard with the help of an API.
Support	The Terraform community is the best support here.	The HashiCorp SLA works on a severity basis such as Urgent, High, Normal, and Low.

To summarize, both the open-source and HashiCorp Terraform implementations have significant advantages and downsides. Consider your organization's specific requirements, such as extensibility, enterprise-level capabilities, and release cycle frequency, while picking between the two. The choice between the two comes down to determining which solution best meets your organization's infrastructure management needs.

Scope of Terraform Automation

Terraform is a powerful tool with extensive automation features. Its automated features extend to infrastructure provisioning, configuration management, and application deployment. Organizations can use it to describe and manage their infrastructure as code, allowing standard and repeatable infrastructure management practices. Because Terraform is an IaC tool and has a wide range of supported providers, it is platform agnostic and abstracts away the complexity of infrastructure management. Terraform may serve as a single intermediate layer that can communicate with many infrastructure components in the same language (for example, HCL) while totally hiding from consumers the other complexity related to a variety of infrastructure tools and technologies. Terraform may therefore be used in every area where infrastructure automation is conceivable.

Terraform automation covers a broad variety of infrastructure management functions. One area where Terraform automation does well is **infrastructure provisioning**. It enables businesses to automate the development of infrastructure resources on a variety of cloud providers, including AWS, Azure, VMware, and Google Cloud. This indicates that organizations can offer resources quickly and easily in a consistent and repeatable manner, ensuring that their infrastructure is always in a known condition.

Configuration management is another place where Terraform automation shines. Terraform enables businesses to design and manage infrastructure settings declaratively. As a result, organizations may simply enforce configuration standards, lowering the risk of mistakes and boosting infrastructure security. Infrastructure configuration changes can be pushed out in a controlled and repeatable manner with Terraform, decreasing the risk of service interruptions and ensuring that infrastructure is constantly up-to-date.

Another area where Terraform automation can possibly be leveraged is **application deployment**. Terraform provides a flexible and scalable solution to deploy applications across many environments, allowing organizations to migrate applications from development to testing to production swiftly and simply. This can help organizations focus on delivering value to their consumers by reducing the time and effort necessary to deploy apps.

Terraform automation has an extensive scope that includes multiple aspects of infrastructure management in addition to the categories described earlier. Terraform, for example, can be used to **automate network and security setups**. It implies organizations may automate the development and administration of virtual networks, subnets, firewall rules, and other network-related resources, saving time and effort in managing network infrastructure.

Terraform can additionally be used to automate the administration of **containerized environments**. Firms are increasingly using container orchestration technologies such as Kubernetes as containerization and microservices architectures gain traction. Terraform can be used to automate Kubernetes cluster construction and administration, lowering the complexity of managing containerized environments.

Compliance management is another area where Terraform automation can be used. Most organizations are now subject to regulatory regulations and industry standards that need certain security and

compliance practices. Terraform can assist organizations in automating compliance standard enforcement, ensuring that infrastructure is always set up in a compliant manner.

Finally, one can use Terraform to automate **disaster recovery** and **business continuity planning**. Terraform can be used by organizations to create and manage disaster recovery sites and configurations, ensuring that infrastructure is always available in the event of outages or disasters.

Overall, the scope of Terraform automation is broad and covers many aspects of infrastructure management. Organizations can employ Terraform to **automate the building of infrastructure resources, enforce configuration standards, and streamline application deployment**. As such, it is an effective tool for aiding organizations in enhancing their infrastructure management practices, reducing the chance of mistakes, and increasing overall efficiency.

Finally, the scope of Terraform automation is broad and covers many aspects of infrastructure management. Organizations can use Terraform to automate infrastructure resource generation and maintenance, enforce configuration and compliance requirements, streamline application deployment, and manage containerized environments, among other things. As a result, Terraforming is a must-have solution for companies trying to improve their infrastructure management practices and incorporate DevOps ideas. To grasp the extent of Terraform automation within organizations, refer to Figure 1-5, which provides insight into its scope.

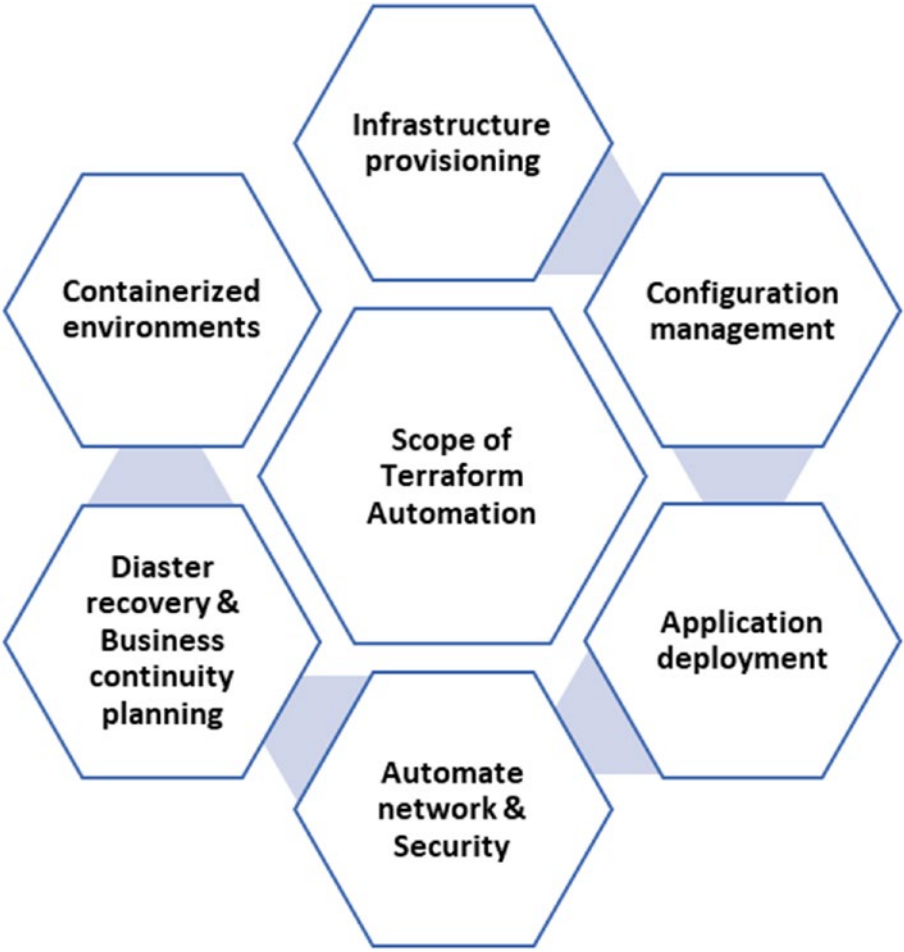


Figure 1-5. *Scope of Terraform automation*

Harnessing the Power of Terraform

Infrastructure automation has become a critical component in attaining agility, scalability, and dependability in the current fast-paced and changing technological ecosystem. Terraform, an open-source IaC tool, has emerged as an effective option for managing infrastructure across

several cloud providers and platforms. Its adaptable features and solid ecosystem have made it a popular choice for organizations seeking to streamline infrastructure setup, testing, and operations. In this section, we will go through some common Terraform use cases and how it is changing the way we build and manage infrastructure.

Mutable vs. Immutable Infrastructure

In the realm of infrastructure management, two contrasting infrastructure approaches have emerged: mutable and immutable. While mutable infrastructure involves modifying existing resources, immutable infrastructure follows a "build and replace" philosophy, where new instances are provisioned for each configuration change. In this section, we will explore the differences between mutable and immutable infrastructure, and how Terraform, an infrastructure as code tool, can empower organizations to leverage the benefits of immutable infrastructure.

Mutable Infrastructure

Mutable infrastructure refers to the traditional approach of modifying existing resources in response to configuration changes. It involves making in-place modifications to running servers, such as updating software packages, changing configurations, or applying patches. This approach allows for incremental updates and adaptability, as changes can be applied directly to the existing infrastructure.

For example, let's say an organization wants to update the version of a web server software on their production servers. With mutable infrastructure, they would log into each server individually, manually update the software, and restart the server to apply the changes.

These are the challenges of mutable infrastructure:

- **Configuration drift:** Frequent modifications to running servers can lead to configuration drift, where the actual configuration deviates from the desired state. This can result in inconsistencies and potential vulnerabilities across the infrastructure.
- **Complexity and risk:** Modifying running servers introduces complexity and increases the risk of errors. Manual changes may differ between servers, making it difficult to maintain consistency and reproduce the infrastructure environment.

Immutable Infrastructure

The concept of immutable infrastructure is one of the fundamental notions offered by Terraform. Rather than manually changing current infrastructure, Terraform enables the predictable and reproducible construction of new, disposable infrastructure. Organizations can preserve consistency and minimize configuration drift by accepting immutable infrastructure, which leads to more stable and reliable systems. Terraform declarative syntax and infrastructure state management make it easy to define infrastructure as code, allowing teams to rapidly provision and destroy environments effortlessly.

Immutable infrastructure takes a different approach by emphasizing the creation of new instances for every configuration change. Instead of modifying existing resources, new instances are provisioned with the updated configurations, while old instances are decommissioned. This approach ensures that infrastructure remains consistent and reproducible and eliminates configuration drift.

For example, using Terraform, an organization can define infrastructure configurations in code and create new instances with the desired changes. For the web server software update mentioned earlier, Terraform would provision new instances with the updated software version and seamlessly switch traffic to the new instances.

Immutable infrastructure with Terraform offers a number of advantages:

- **Consistency and reproducibility:** Immutable infrastructure provides a consistent and reproducible environment as every deployment involves creating new instances with known configurations defined in Terraform code. This ensures that each deployment is predictable and eliminates configuration drift.
- **Resilience and scalability:** With immutable infrastructure, failures and scalability challenges can be addressed by provisioning new instances. The Terraform IaC approach allows organizations to easily scale up or down their infrastructure by provisioning new resources and decommissioning old ones as needed.
- **Rollbacks and roll forwards:** Immutable infrastructure simplifies rollbacks and roll forwards. If an issue occurs, organizations can roll back by redirecting traffic to the previous instances. Similarly, rolling forward involves provisioning new instances with updated configurations and updating routing to the new instances, ensuring seamless transitions.

- **Infrastructure configuration as code:** Terraform enables organizations to define infrastructure configurations in code, facilitating collaboration, version control, and automated deployments. Infrastructure code can be reviewed, tested, and deployed as part of a CI/CD pipeline, ensuring consistent and reliable infrastructure changes. Figure 1-6 and Table 1-3 give insights into the mechanics of mutable and immutable updates, which will help you understand how they work along and the architectural differences between the two.

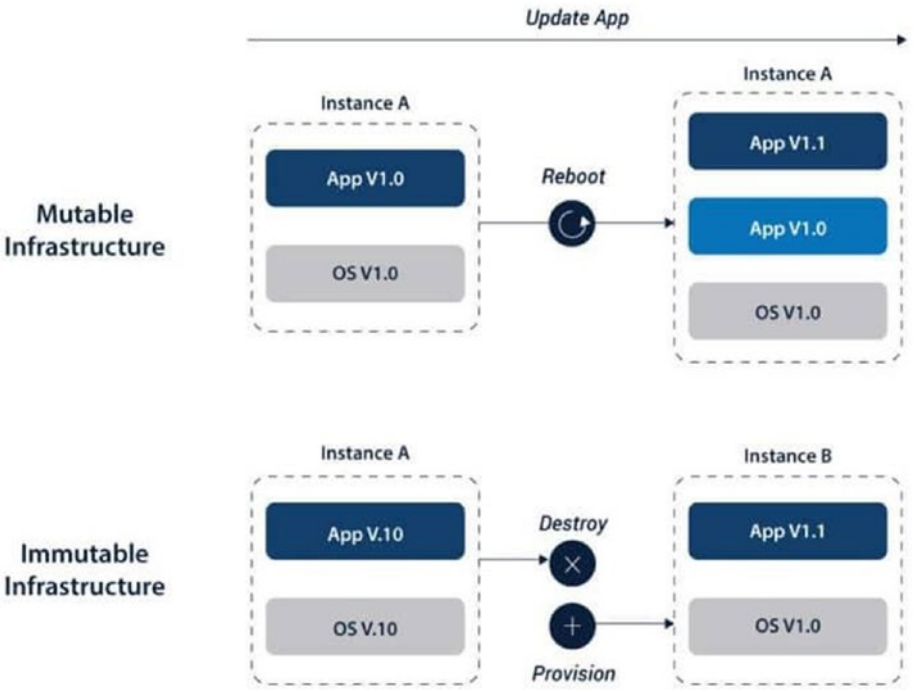


Figure 1-6. *Mutable versus immutable infrastructure*

Table 1-3. *Mutable vs. Immutable*

Topic	Mutable	Immutable
Consistency	Requires reviews to ensure configuration consistency across nodes	Streamlines operations
Software updates	Requires ongoing configuration changes of the underlying infrastructure to support application updates	Supports continuous deployment of a software application by matching infrastructure version to an application version
Security	Exposes risk of configuration inconsistency across instances	Mitigates manual errors that may result in security threats
Scaling	Offers less control in rapidly replicating an exact configuration	Supports scaling of infrastructure by adding and removing nodes as needed
Operational Cost	Increases operational overhead	Reduces operational costs

Bridging the Gap

As more businesses utilize cloud computing, the necessity for fast infrastructure provisioning becomes critical. Terraform enables enterprises to manage infrastructure across many service models, bridging the gap between traditional infrastructure and cloud-native platforms. In this section, we will look at how Terraform can be used to supply IaaS resources such as virtual machines, storage, and networking, as well as PaaS resources such as managed databases and serverless services. In addition, we will discuss the advantages of using Terraform to manage infrastructure across various service models.

Provisioning IaaS Resources with Terraform

Terraform enables organizations to provision IaaS resources by defining infrastructure configurations as code. With its extensive provider ecosystem, Terraform supports leading cloud providers such as AWS, Azure, and Google Cloud Platform, among others. The following are some key use cases for Terraform in IaaS provisioning.

Virtual Machine Provisioning

Terraform allows the definition and deployment of virtual machines with desired configurations, sizes, and operating systems.

Infrastructure can be scaled up or down effortlessly, ensuring efficient resource utilization and cost management.

Storage and Networking Configuration

Terraform enables the provisioning of storage resources such as disks, object storage, and file shares, along with networking components such as virtual networks, subnets, and load balancers.

Network security groups, firewall rules, and routing configurations can be defined and managed through Terraform code, ensuring consistent and secure network setups.

Provisioning PaaS Resources with Terraform

Terraform versatility extends beyond IaaS provisioning; it also facilitates the provisioning of PaaS resources, empowering organizations to leverage cloud-native services efficiently. The following are some examples of PaaS resources that can be provisioned using Terraform.

Managed Databases

Terraform allows the creation and configuration of managed databases such as Amazon RDS, Azure SQL Database, and Google Cloud Spanner.

Database specifications, performance settings, and security configurations can be defined using Terraform code, enabling consistent and reproducible database deployments.

Serverless Functions and Event-Driven Architectures

Apart from the VMware platform, Terraform integrates with public cloud serverless platforms such as AWS Lambda, Azure Functions, and Google Cloud Functions.

With Terraform, organizations can define serverless functions, configure triggers, and manage event-driven architectures, enabling efficient and scalable application development and deployment.

Benefits of Utilizing Terraform for Managing Infrastructure Across Service Models

The use of Terraform for managing both IaaS and PaaS resources brings numerous benefits to organizations, as shown in Figure 1-7.

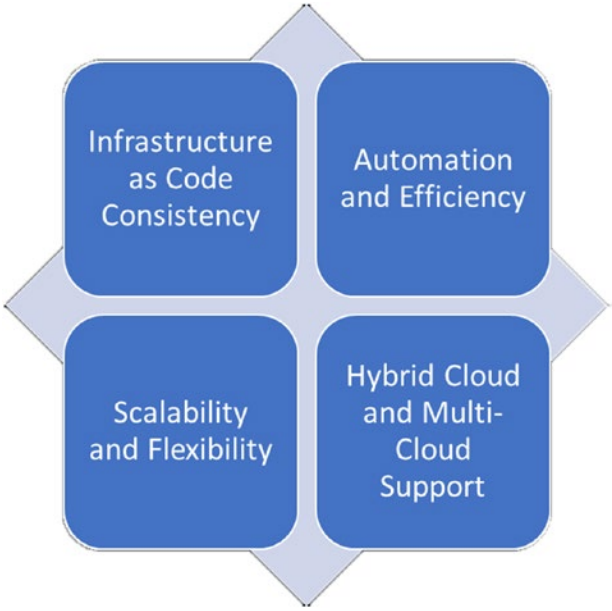


Figure 1-7. *Benefits of using Terraform for managing infrastructure across service models*

Infrastructure as Code Consistency

By treating infrastructure as code, Terraform ensures consistent and reproducible deployments across different service models. Infrastructure configurations are version-controlled, enabling easy collaboration, change tracking, and auditing.

Automation and Efficiency

Terraform enables the automation of infrastructure provisioning, reducing manual effort and minimizing the risk of human error. With Terraform declarative approach, infrastructure changes are efficiently applied, ensuring desired state and accelerating deployment cycles.

Scalability and Flexibility

Terraform's ability to manage infrastructure across service models allows organizations to scale resources up or down based on demand.

The flexibility of provisioning both IaaS and PaaS resources through a unified approach enables organizations to adopt and leverage the most suitable services for their applications.

Hybrid Cloud and Multicloud Support

Terraform's multicloud support allows organizations to manage infrastructure across various cloud providers, on-premises environments, and hybrid cloud configurations. Organizations can avoid vendor lock-in and optimize costs.

Hands-On Exercise: Setting Up Terraform Open Source for VMware Infrastructure on Ubuntu

As we go deeper into the complexity of infrastructure management, the concept of a Terraform provider evolves. A Terraform provider is a plugin that allows Terraform to interface with the API of a specific service provider. Terraform can now deploy and manage resources on the corresponding platform in a timely and efficient manner.

Terraform, as described previously in the chapter, comes with a plethora of built-in providers, including VMware, AWS, Azure, Google Cloud Platform, and a slew of others. Custom providers can also be built to communicate with proprietary or specialized infrastructure. Terraform will connect with the underlying infrastructure using the provider you provide in your configuration file to add, edit, or remove resources as needed. In this regard, the VMware provider is an excellent example of how Terraform

providers perform in practice. The VMware provider is a plugin for VMware's infrastructure platform that facilitates the deployment and management of virtual machines, networks, and storage resources. Terraform can easily provision, update, and delete these resources with a single configuration file by communicating with the VMware API. Furthermore, the VMware partner offers several features that enhance the infrastructure administration experience. Support for a wide range of VMware products is provided. Likewise, the service offers various configuration choices, including customized needs, remote console access, and guest customization. In a nutshell, the VMware supplier emphasizes the power and simplicity of Terraform providers. Terraform abstracts the complexities of infrastructure management and offers a simple, declarative language for specifying desired state, resulting in efficient and scalable resource deployment and maintenance. The VMware provider illustrates Terraform providers' capabilities and features in action. Before we begin, it is important to note that the process of establishing Terraform for VMware infrastructure may vary depending on your specific environment and infrastructure setup. As a result, this exercise is intended to give you a basic overview of the process; you may need to adjust it to match your specific needs.

You will need an Ubuntu system as well as access to a VMware vSphere environment to get started.

Step 1

Terraform requires you to install the necessary packages. This may be accomplished with the apt package manager. Open a terminal window and run `sudo apt-get update` to update the package list. Then, run `sudo apt-get install unzip` to install the unzip package.

Step 2:

You may now download the most recent version of Terraform from the official website (`https://www.Terraform.io/downloads.html`) using your web browser. When the download is finished, use the command `unzip Terraform_*_linux_amd64.zip` to extract the archive, replacing the * with the version number.

Here's the simple command:

```
wget https://releases.hashicorp.com/terraform/1.0.7/
terraform_1.0.7_linux_amd64.zip
```

Step 3

Extract the downloaded file archive.

Here's the simple command:

```
unzip terraform_1.0.7_linux_amd64.zip
```

Step 4

Move the unzipped executable into a directory searched for executables. For Ubuntu it is usually `/usr/local/bin`.

```
sudo mv terraform /usr/local/bin/
```

Finally, change the `PATH` environment variable to include the Terraform binary directory. You can do this by editing the `bashrc` file in your home directory and adding the line `export PATH=$PATH:/path/to/Terraform` at the end, replacing `/path/to/Terraform` with the actual path to the Terraform binary.

You can run the following:

```
terraform --version
```

Here's some sample output:

```
root@ubantu:/home# terraform –version
terraform v1.3.9
on linux_amd64
```

This will provide you with the needed Terraform core product, which is now installed on your Ubuntu computer. Now you have two options for installing the vSphere provider after you have Terraform installed on your system.

- **Automated installation of the vSphere provider:**

  To get the Terraform provider for vSphere, create a `main.tf` file in the current directory from where you can access the Terraform executable.

  Put the following sample contents in the `main.tf` file:

```
Terraform {
  required_providers {
    vsphere = {
      source = "hashicorp/vsphere"
      version = "2.3.1"
    }
  }

required_version = ">=0.13" #This is if you have
Terraform Version13 installed in previous step

}
```

  Navigate to your working directory where you have Terraform `main.tf` file created and run `Terraform init`.

  Example: Initialize and download the provider.

```
$ terraform init
Initializing the backend...
Initializing provider plugins...
- Finding hashicorp/vsphere versions matching ">=
x.y.z" ...
- Installing hashicorp/vsphere x.y.z ...
- Installed hashicorp/vsphere x.y.z (signed by
HashiCorp, key ID **********)
...terraform has been successfully initialized!
```

- **Manual installation of the vSphere provider:**

  The latest release of the provider can be found on releases.hashicorp.com. You can download the appropriate version of the provider for your operating system using a command-line shell or a browser.

  The following example uses Bash on Linux (x64):

  a. On a Linux operating system with Internet access, download the plugin from GitHub.

  ```
  RELEASE=x.y.z
  wget -q https://releases.hashicorp.com/terraform-
  provider-vsphere/${RELEASE}/terraform-provider-
  vsphere_${RELEASE}_linux_amd64.zip
  ```

Note You can check the latest version of the vSphere provider on the Terraform provider registry website.

https://registry.terraform.io/providers/hashicorp/vsphere/latest/docs

---

  b. Extract the plugin.

  ```
  tar xvf terraform-provider-vsphere_${RELEASE}_
  linux_amd64.zip
  ```

  c. Create a directory for the provider.

  ```
  mkdir -p ~/.terraform.d/plugins/local/hashicorp/
  vsphere/${RELEASE}/linux_amd64
  ```

d. Copy the extracted plugin to a target system and move to the Terraform `plugins` directory.

```
mv terraform-provider-vsphere_v${RELEASE}
~/.terraform.d/plugins/local/hashicorp/
vsphere/${RELEASE}/linux_amd64
```

e. Verify the presence of the plugin in the Terraform `plugins` directory.

```
cd ~/.terraform.d/plugins/local/hashicorp/
vsphere/${RELEASE}/linux_amd64
ls
```

Once Terraform and the vSphere provider are installed, you can use the `Terraform -version` tool to validate all your installations.

```
root@ubantu:/home/user# terraform - -version
terraform v1.3.9
on linux_amd64
+provider local/hashicorp/vsphere v2..3.1
root@ubantu:/home/user#
```

Summary

This chapter covered the transformative impact of infrastructure as code on IT infrastructure management. We explored Terraform's declarative model and its integration with providers and provisioners for diverse platforms. We compared the open-source and HashiCorp versions and highlighted Terraform's automation capabilities for immutable infrastructure and standardized deployments.

Additionally, we emphasized Terraform's ability to bridge IaaS and PaaS provisioning and concluded with a hands-on exercise for setting up Terraform on Ubuntu for VMware infrastructure. Overall, the chapter provided a comprehensive understanding of IaC and Terraform's potential for efficient infrastructure management.

CHAPTER 2

Deep Dive into Terraform

In the previous chapter, we delved into the world of infrastructure as code (IaC) and explored fundamental concepts related to Terraform. We discussed Terraform's widespread application across various IT infrastructure domains and highlighted the significant advantages of automating infrastructure using Terraform.

In this chapter, we will take a deeper dive into the practical aspects of using Terraform and its related concepts. We will commence by examining Terraform's role in diverse infrastructure landscapes. Furthermore, we will introduce you to the pivotal components of Terraform operations, namely, the configuration file and state file. We'll closely scrutinize these Terraform files and their associated sections and elucidate how they enable us to define and manage resources using IaC.

Our exploration will also encompass essential concepts that empower Terraform, such as providers, modules, variables, and more. To provide a hands-on experience, we will walk you through a practical exercise demonstrating how to provision a VMware virtual machine using Terraform.

S. Bhatia and C. Gabhane, *Reverse Engineering with Terraform*,
https://doi.org/10.1007/979-8-8688-0074-0_2

Terraform and Its Presence in the IT Infrastructure Ecosystem

As infrastructure continues to evolve, it can be helpful to categorize it into broad groups based on its purpose and function. We are going to explore some of the most common infrastructure domains and how Terraform fits into each of them.

Cloud Infrastructure

Cloud infrastructure is a type of infrastructure that is hosted in the infrastructure space managed by a third-party provider, rather than on-premises. That third-party provider offers IT infra and services to a variety of organizations from its common platform. Examples of cloud infrastructure providers include Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform (GCP). Terraform has a strong presence in the cloud infrastructure market, with support for all the major cloud providers. With Terraform, organizations can use a single codebase to manage their infrastructure across multiple cloud providers, simplifying management and reducing costs.

Network Infrastructure

Network infrastructure refers to the hardware and software components that make up a network, including routers, switches, and firewalls. Especially with the origin of virtualization in the network domain that relates to software-defined networking (SDN), the demand of automated management in this domain has grown significantly. Terraform has a presence in the network infrastructure market, with support for provisioning and managing network infrastructure. With Terraform, organizations can use a single codebase to manage their network infrastructure, reducing management costs and improving efficiency.

Application Infrastructure

Application infrastructure refers to the software and hardware components that make up an application, including web servers, databases, and load balancers. Terraform has a presence in the application infrastructure market, with support for provisioning and managing application infrastructure. With Terraform, organizations can use a single codebase to manage their application infrastructure, reducing management costs and improving efficiency.

Security Infrastructure

Security infrastructure refers to the hardware and software components that are used to secure an organization's IT infrastructure, including firewalls, intrusion detection systems, and encryption tools. Terraform has a presence in the security infrastructure market, with support for managing security infrastructure. With Terraform, organizations can use a single codebase to manage their security infrastructure, improving efficiency and reducing the risk of errors.

According to the 2021 State of DevOps report by Puppet, Terraform was reported as the most popular IaC tool among respondents, with 40 percent of organizations using it for infrastructure management. In addition, the report found that Terraform is the second most popular tool for provisioning and configuration management, after Ansible.

Terraform's popularity is further reflected in its vibrant and growing community of users and contributors, who have developed and shared thousands of open-source modules and plugins for the tool. The Terraform registry, which provides a central repository for modules, currently contains more than 15,000 modules, covering a wide range of infrastructure resources.

Terraform Files Deep Dive

Understanding the essential ideas of Terraform files, particularly its **state** and **configuration files**, is critical. These files are the foundation of Terraform's infrastructure-as-code paradigm, and understanding how they operate is critical to working effectively with Terraform. Let us jump into each of the file types in detail. Please refer to Figure 2-1.

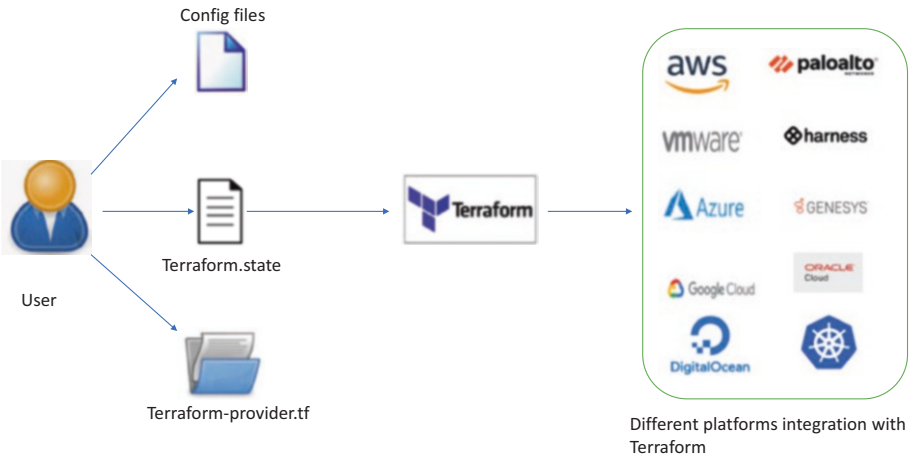


Figure 2-1. *Terraform files deep dive*

Configuration Files

The Terraform configuration file is a plain-text file written in HashiCorp Configuration Language (HCL), or JSON, that describes the desired state of your infrastructure. This file typically contains resource definitions, variables, and data sources, which are used to define and provision the infrastructure resources that make up your system. The Terraform configuration file is used to define the desired state of your infrastructure and what resources need to be created, modified, or deleted to reach that state.

Providers

As we covered in Chapter 1, a Terraform provider is a plugin that enables the communication between Terraform and the API of a particular service provider. This allows Terraform to deploy and manage resources on the respective platform with ease and efficiency. A provider in Terraform is a plugin that allows Terraform to interact with a specific type of infrastructure, such as a cloud provider, a database service, or a networking device. Every platform has a unique provider that needs to be deployed and integrated with the Terraform configuration files to manage the platform using Terraform.

State File

Terraform generates the Terraform state file to keep track of the current condition of your infrastructure. The user has no influence over this file; it is automatically generated by the provider and retrieved straight from the platform. This state file describes the real status of the Terraform infrastructure. This file includes details on the resources that have been generated, their present status, and their relationships. Terraform uses the state file to identify what modifications need to be done to your infrastructure to get it to the desired state. Terraform examines the state file and compares it to the configuration file to determine what changes need to be done when you run Terraform apply.

It is important to note that the Terraform state file is critical to the operation of Terraform, as it is used to track the state of your infrastructure and maintain consistency between your configuration and your actual infrastructure. The state file is also used to store sensitive information such as passwords and access keys, so it should be stored securely and not shared between team members.

In addition, it is important to keep the Terraform state file up-to-date and consistent with your infrastructure. This can be achieved by using version control to track changes to the configuration file and state file and by regularly running "Terraform plan" to preview the changes that Terraform will make.

In nutshell, the Terraform configuration and state files are essential components of the infrastructure-as-code paradigm that Terraform follows. The configuration file describes the desired state of your infrastructure, while the state file tracks the current state and changes that need to be made. By understanding these files and keeping them up-to-date and secure, you can work effectively with Terraform and maintain the desired state of your infrastructure.

As a tool for infrastructure automation, Terraform provides a powerful means of defining and managing infrastructure as code. However, as with any complex system, it is critical to maintain a proper backup strategy to ensure the availability and integrity of your data. In the case of Terraform, this means backing up the state file.

The state file is a crucial component of the Terraform workflow, as it records the current state of the infrastructure being managed. This includes information such as the resources that have been created, their current configuration, and any dependencies or relationships between them. The state file is used by Terraform to determine the changes that need to be made to the infrastructure to bring it into the desired state and is therefore a fundamental aspect of the tool's functionality. Given the importance of the state file, it is critical to have a reliable backup strategy in place. This can be accomplished by following a few simple steps.

Determine the location of the state file. By default, Terraform stores the state file locally in a file named `Terraform.tfstate`. However, it is also possible to store the state remotely using services such as Amazon S3 or HashiCorp console. Figure 2-2 shows the suggested backup steps.

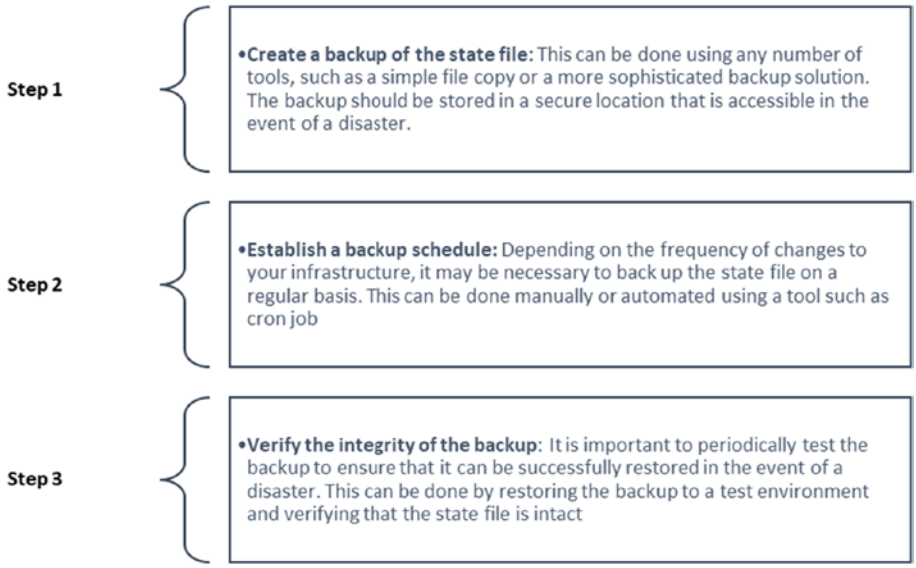


Figure 2-2. *Backup steps for state file*

By following these steps, you can ensure that your Terraform state file is properly backed up and can be reliably restored in the event of a disaster. This can provide peace of mind and help to minimize the impact of unexpected events on your infrastructure.

Additionally, it is important to consider the security of the state file backups. The state file may contain sensitive information such as access keys or passwords, and as such, it should be stored in a secure location with appropriate access controls in place. Encryption can also be used to further protect the state file from unauthorized access.

Another important consideration is the versioning of state file backups. As changes are made to the infrastructure, the state file will be updated accordingly. However, it may be necessary to revert to a previous version of the state file in the event of an issue. To enable this, it is recommended to keep a version history of the state file backups, allowing for easy retrieval of previous versions.

In addition to manual backups, Terraform also provides a built-in feature for backing up the state file automatically. This can be accomplished by configuring the back end to automatically store state snapshots at regular intervals. This feature can help to further improve the reliability and ease of state file backups and can be configured to meet the needs of your specific environment.

To summarize, the state file is a fundamental component of the Terraform operation, and as such, a dependable backup solution is required. You may verify that your Terraform state file is properly backed up and can be reliably restored in the case of a disaster by following the procedures mentioned and considering additional security and versioning issues. This can assist to reduce the effect of unforeseen occurrences on your infrastructure and offer you and your staff peace of mind.

Config File and Its Different Sections

The configuration file, which is the heart of Terraform workflow, defines the desired state of the infrastructure being managed by Terraform. This includes information such as the resources that need to be created, their configuration settings, and any dependencies or relationships between them. The configuration file is used by Terraform to determine the changes that need to be made to the infrastructure to bring it into the desired state.

It is important to understand the different files and sections in the Terraform configuration file, which is the key to describing the desired state of your infrastructure. The configuration file is written in HashiCorp Configuration Language and contains resource definitions, variables, and data sources, among other things. Each section of the configuration file has its own purpose and importance in the infrastructure-as-code paradigm that Terraform follows. The following is a brief explanation of different sections of the configuration file. While explaining different sections, we would provide certain sample code for you to refer to for a typical config file taking example of VMware VM deployment.

Provider Section

The provider section of the configuration file is used to define the infrastructure provider that will be used to provision the resources. This section contains information such as the provider's name, version, and configuration options. Providers can include cloud providers such as VMware, AWS, Azure, and Google Cloud Platform, as well as other infrastructure providers.

The following is the reference provider section for a VMware VM deployment via Terraform:

```
terraform {
  required_providers {
    vsphere = {
      source = "local/hashicorp/vsphere"
      version = "2.3.1"  ## Check for latest provider available
    }
  }
}

provider "vsphere" {
vsphere_server = "<Fully qualified domain name or IP of Vsphere
server>"
user = "administrator@vsphere.local"
password = "XXXX"
#if you have a self-signed cert
allow_unverified_ssl = true
}
```

Data Section

The data section is used to define data sources, which provide read-only access to external data that can be used to inform the configuration of resources. Terraform providers are employed to bring the values referred to in the data section. The values identified are unique for that platform. We will explain more about the variables and referencing in the later part of this chapter. Here we just want to bring the attention to the "data" section required to write a configuration file.

```
data "vsphere_datacenter" "dc" {
  name = var.datacenter
}

data "vsphere_resource_pool" "pool" {
  name          = var.resource_pool
  datacenter_id = "${data.vsphere_datacenter.dc.id}"
}

data "vsphere_tag" "tag2" {
  name        = var.environment_tag
  category_id = "${data.vsphere_tag_category.category2.id}"
}

data "vsphere_folder" "folder" {
  path = var.vcentre_folder
}
... so on.
```

Resource Section

The resource section is used to define the resources that make up your infrastructure. This section contains information such as the resource type, name, and configuration options. Resources can include virtual machines,

networks, storage accounts, and more. Each resource definition in the configuration file corresponds to a resource that Terraform will create, modify, or delete in your infrastructure.

The following is a sample resource section. You can modify the values based on your need.

```
resource "vsphere_virtual_machine" "XXXXX" {
name             = var.vm_name
resource_pool_id = "${data.vsphere_resource_pool.pool.id}"
#resource_pool_id = "${data.vsphere_compute_cluster.cluster.
                    resource_pool_id}"
datastore_id = "${data.vsphere_datastore.datastore1.id}"
folder     = "${data.vsphere_datacenter.dc.name}/vm/${data.
             vsphere_folder.folder.path}"
num_cpus                    = var.vm_cpu
cpu_hot_add_enabled         = "true"
  memory                      = var.vm_memory
  memory_hot_add_enabled      = "true"
  wait_for_guest_net_timeout = 0
  wait_for_guest_ip_timeout  = 0
  firmware                    = "${data.vsphere_virtual_machine.
                                template.firmware}"
  tags                        = ["${data.vsphere_tag.tag1.id}",
                                "${data.vsphere_tag.tag2.id}"]
  guest_id                    = "${data.vsphere_virtual_machine.
                                template.guest_id}"

  network_interface {
 network_id   = "${data.vsphere_network.network_vlan.id}"
 adapter_type= "${data.vsphere_virtual_machine.template.
               network_interface_types[0]}"
  }
```

```
  disk {
    label            = "disk0.vmdk"
    size             = var.disk_size
 eagerly_scrub    = "${data.vsphere_virtual_machine.template.
                    disks.0.eagerly_scrub}"
 thin_provisioned="${data.vsphere_virtual_machine.template.
                  disks.0.thin_provisioned}"
  }

  clone {
    template_uuid = "${data.vsphere_virtual_machine.
                    template.id}"

    customize {
      windows_options {
        computer_name         = var.computer_name
        admin_password        = var.vm_admin_password
        auto_logon            = true
        auto_logon_count      = 4
        full_name             = "Administrator"
       run_once_command_list = var.skip_post_deploy ? local.
       start_up_nocmd : local.start_up_command
      }

      network_interface {
        ipv4_address = var.network_address
        ipv4_netmask = var.network_address_subnetmask
      }

      ipv4_gateway = var.network_address_gateway
    }
  }
}
```

The objects defined in this resource section are important and need to be used as is. The name of the objects are fixed and unique per provider; however, the value of these objects are specific to your environment. We see in the resource section that there are different subsections. For example, in the code snippet of the resource section, we can see for a VMware VM deployment, we have separate definitions to define the general settings of a VM, then network configurations, disk configurations, etc.

Variable Section

The variable section is used to define variables that can be used throughout the configuration file. Variables can include strings, numbers, and Booleans, and can be used to pass in values to resource configurations. Variables are an important way to make your Terraform configuration more flexible and reusable. Variables play a crucial role when we want to manage the environment with the help of Terraform modules. The Terraform modules are the repeatable configuration requirements written where users can just enter desired values without going into the background writing of a detailed configuration file. We will touch upon variables and modules in detail during the later part of this chapter. Here are sample Terraform `variables.tf` and module files for reference:

`Vaiables.tf file:`

```
variable "datacenter" {
  description = "VMware Vsphere datacenter name"
}
```

```
variable "resource_pool" {
  description = "Specify cluster present inside datacentre,
                example: <ClusterName>/Resources"
}

variable "vm_name" {
  description = "Name of the VM to be assigned in Vcentre"
}

variable "vm_cpu" {
  description = "Count of CPU's to be assigned to VM eg. 1 or 4
                or 8 etc."
}

variable "vm_memory" {
  description = "Amount of memory to be assigned in MB for
                example 12288 MB == 12 GB of RAM"
}

variable "datastore" {
  description = "Vsphere datastore name where VM is to be
                deployed"
}

variable "disk_size" {
  description = "Size of the disk we need to deploy in GB"
}

variable "network" {
  description = "Network name where assignment is required"
}

variable "network_address" {
  description = "Static IP address that needs to be assigned"
}
```

```
variable "network_address_subnetmask" {
  description = "Subnet Mask in CIDR notation eg. 23 or
                 24 etc."
}

variable "network_address_gateway" {
  description = "Gateway for the subnet specified"
}

variable "template_name" {
  description = "Name of the VM Template through which VM will
                 be cloned"
}

variable "computer_name" {
  description = "Name of the computer that is to be assigned"
}

variable "skip_post_deploy" {
  type        = bool
  description = "Flag to enable/disable post deployment task
                 like Salt bootstrap, DNS"
  default     = false
}

variable "environment_tag" {
  description = "Resource tags"
  #Possible values: "Development", "Production",
                    "Qualification", "Test"
}

variable "vcentre_folder" {
  description = "Vsphere folder to deploy a VM"
}
```

The following are some sample module inputs for ease in deployment. If you notice, we can define a variable for each input required to play in a module. Users just need to provide sample values in the following variables and can just play the following file for the VM deployment. They do not need to write a detailed configuration file for a VM deployment each time. Just specify the required values in a module. Thus, you can create repeatable code.

```
module "vmware_windows_vm" {
  source         = "./../"
  datacenter     = "LabXXXX"
  resource_pool = "x.x.x.x/Resources"
  instance_count              = 1
  vm_name                     = "Ubantu"
  vm_cpu                      = 4
  vm_memory                   = 12288
  datastore                   = "Local_Disk_800GB"
  disk_size                   = 120
  network                     = "VM Network"
  network_address             = "x.x.x.x"
  network_address_subnetmask = 24
  network_address_gateway     = "x.x.x.x"
  template_name               = "Win2K16"
  computer_name               = "Terraform-test"
  skip_post_deploy            = true
  catg_enironment             = "Environment"
  vcentre_folder              = "TerraformDeployments"
}
```

We are going to touch upon modules again in the later part of this chapter.

Terraform Provisioners

Terraform provisioners come to the rescue by enabling the execution of scripts and commands on target machines before or after resource creation. In this section, we will explore the fundamentals of Terraform provisioners and their use cases, and we will provide a code example illustrating their usage in VMware infrastructure.

Understanding Terraform Provisioners

Terraform provisioners are a powerful feature that allows you to run scripts or execute commands on remote machines during the infrastructure provisioning process. They facilitate various automation tasks such as configuring software, installing dependencies, initializing databases, and more.

Terraform supports two types of provisioners.

- **Inline provisioners:** Inline provisioners are defined directly within the Terraform configuration file.
- **External provisioners:** External provisioners are separate scripts or commands executed by Terraform.

Example Use of Inline Provisioners in VMware VM Deployment

Let's consider an example where we want to provision a virtual machine on VMware and configure a custom script to run after the VM creation.

Here's the `main.tf` file:

```
provider "vsphere" {
  user             = var.vsphere_username
  password         = var.vsphere_password
  vsphere_server   = var.vsphere_server
```

```
  allow_unverified_ssl = true
}

resource "vsphere_virtual_machine" "example_vm" {
  name             = "my-vm"
  resource_pool_id = data.vsphere_resource_pool.pool.id
  datastore_id     = data.vsphere_datastore.datastore.id
  template_uuid    = data.vsphere_virtual_machine.template.id
  num_cpus         = 2
  memory           = 4096
  network_interface {
    network_id = data.vsphere_network.network.id
  }

  provisioner "remote-exec" {
    inline = [
      "echo 'Hello, Provisioner!'",
      "echo 'This is an example of an inline provisioner.'",
      "echo 'You can run custom scripts or commands here.'",
    ]
  }
}
```

In this example, we provision a VMware virtual machine using the `vsphere_virtual_machine` resource. Additionally, we define an inline provisioner of type `remote-exec`. The inline provisioner executes a series of commands or scripts on the newly created virtual machine.

Example Use of External Provisioners

External provisioners allow you to execute custom scripts or actions during the provisioning process. Here's an example use case of using external provisioners with VMware VM deployment.

Let's say you need to deploy VMware virtual machines and configure them to join an Active Directory domain as part of your infrastructure setup. You can achieve this by using external provisioners in Terraform. Here's how you might structure your Terraform configuration:

Set Up Variables and Resources

First, you would define your variables and resources, including the VMware virtual machine:

Here is the HCL code:

```
variable "vm_name" {
  type    = string
  default = "my-vm"
}

resource "vsphere_virtual_machine" "my_vm" {
  name              = var.vm_name
  guest_id          = "windows9Server64Guest"
  ...
}
```

Use an External Provisioner

Now, you can use an external provisioner to run a script on the deployed VM. In this case, we're using the `local-exec` provisioner, which runs a script on the machine running Terraform. You can use an appropriate provisioner based on your use case.

```
resource "null_resource" "configure_vm" {
  triggers = {
    vm_id = vsphere_virtual_machine.my_vm.id
  }
```

```
  provisioner "local-exec" {
    command = "powershell -Command \".\\join_domain.ps1\""
  }
}
```

Create the External Script

You would also need to create the `join_domain.ps1` script in the same directory as your Terraform configuration. This script would contain the logic to join the VM to the Active Directory domain.

```
# PowerShell script to join a Windows VM to a domain
$domain = "example.com"
$username = "admin"
$password = "password"
Add-Computer -DomainName $domain -Credential (Get-Credential
-UserName $username -Password $password)
```

Depend on Script Execution

To ensure that the script is executed after the VM is deployed, set up a dependency between the `null_resource` and the VMware VM.

```
depends_on = [vsphere_virtual_machine.my_vm]
```

Apply the Configuration

Finally, apply your Terraform configuration, and Terraform will deploy the VMware VM and then execute the script to join it to the Active Directory domain.

This example demonstrates how you can use external provisioners to perform custom actions during VMware VM deployment. Depending on your specific use case, you can adapt the script and provisioner to perform other tasks such as software installation, configuration, or system updates as needed.

Output Section

The output section is used to define the outputs that will be generated by the Terraform configuration. Outputs can include resource IDs, IP addresses, and other information that may be needed by other parts of your infrastructure provisioning/management. Outputs can be used to pass information between different parts of your Terraform configuration or to other systems.

Backup Strategy for Config File

Given the importance of the configuration file, it is critical to have a reliable backup strategy in place. This can be accomplished by following a few simple steps, as shown in Figure 2-3.

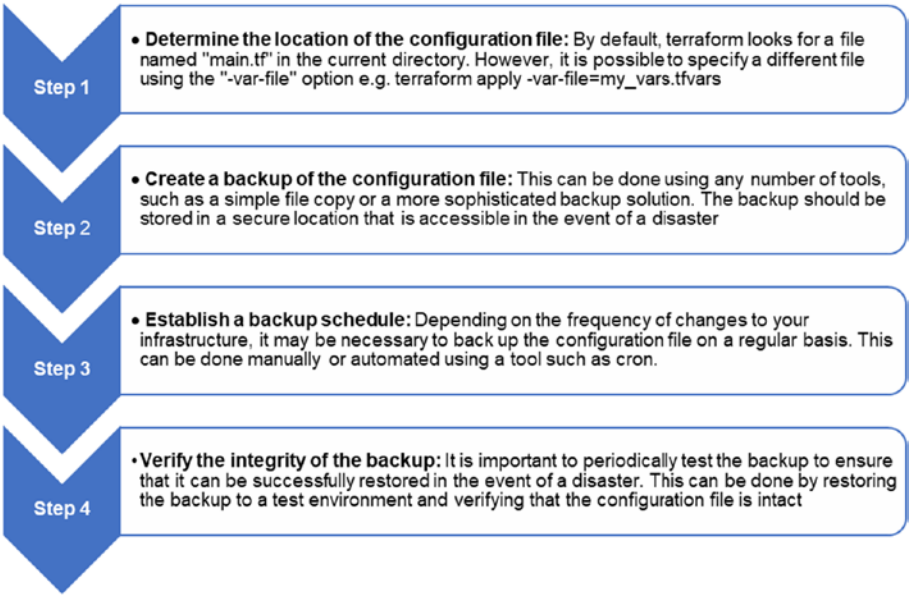


Figure 2-3. *Steps for backing up config files*

By following these steps, you can ensure that your Terraform configuration file is properly backed up and can be reliably restored in the event of a disaster. This can provide peace of mind and help to minimize the impact of unexpected events on your infrastructure.

In addition to the configuration file, it is also important to consider the security of any sensitive data that may be included in the file. This can include access keys or passwords, which should be stored securely and encrypted if possible. Additionally, it is recommended to use version control to manage changes to the configuration file, allowing for easy retrieval of previous versions if needed.

Another important consideration when dealing with Terraform configuration files is the use of variables. Variables provide a means of passing values to your configuration file, which can help to make your infrastructure more flexible and easier to manage. However, it is important to ensure that these variables are properly managed and secured, especially when dealing with sensitive data.

To ensure the security of your variables, it is recommended to use a separate file to store them, rather than including them directly in your configuration file. This file can then be encrypted and stored in a secure location, with appropriate access controls in place. Additionally, it is important to ensure that any variables passed to your configuration file are properly validated, to prevent unauthorized access or malicious modifications.

In addition to variables, it is also important to consider the use of Terraform modules. Modules provide a means of encapsulating infrastructure components and making them reusable, which can help to simplify your configuration file and reduce duplication. However, it is important to ensure that modules are properly managed and versioned, to ensure the integrity of your infrastructure.

To manage your Terraform modules, it is recommended to use a version control system such as Git, which can help to track changes and enable collaboration. Additionally, it is important to ensure that modules are properly tested before deployment, to prevent issues or conflicts with other components of your infrastructure.

Mastering Control: Utilizing Terraform Variables as Powerful Module Parameters

To streamline the deployment process and promote reusability, it's essential to parameterize our Terraform configuration. Here, we will explore how we can leverage variables, maps, and lists in Terraform, along with variable defaults and populating variables, to enhance our configuration management. We will also discuss specific examples of using these features in the context of deploying infrastructure on VMware.

Variables

Variables in Terraform provide a way to dynamically set and manage values within our configuration. By using variables, we can make our Terraform modules more flexible, allowing for customization and adaptability. For example, let's say we are deploying a VMware virtual machine (VM) using Terraform, and we want to specify the VM size as a variable. We can define a variable named `vm_size` and assign it a default value or let the user input a value during runtime. This way, we can easily change the VM size based on specific requirements or environmental constraints.

In the context of VMware infrastructure provisioning with Terraform, variables can be utilized to define and customize various aspects of the infrastructure. Let's talk about how variables can be leveraged.

Defining Variables

In your Terraform configuration file (typically with a `.tf` extension), you can declare variables in the variable block. For example, you can define variables such as `vm_name`, `cpu_count`, `memory_size`, and `disk_size` to specify the characteristics of virtual machines to be provisioned in VMware.

Assigning Variable Values

Variable values can be assigned directly in the configuration file or through other mechanisms such as environment variables or command-line arguments. Assigning variable values is at the user discretion and their strategy for adopting Terraform in their environment.

For instance, you can set the value of `vm_name` to `web-server` and `cpu_count` to 2, either in the configuration file or by passing them as command-line arguments when executing Terraform commands. These variable assignments are of different types including string, numbers, Boolean, etc.

Input Validation and Type Constraints

Terraform allows you to enforce constraints on variable values, such as required fields, allowed value ranges, or specific data types. You can specify validation rules using attributes such as default, description, type, validation, and more in the variable block.

For example, you can enforce that the `cpu_count` variable must be an integer greater than or equal to 1.

Variable Interpolation

Variables can be interpolated within your Terraform configuration to dynamically populate values in resource definitions. For instance, you can use the `vm_name` variable as part of the virtual machine resource block to create unique virtual machine names based on the provided input.

Variable Overrides

When executing Terraform commands, you can override variable values to accommodate different environments or specific use cases. This flexibility allows you to reuse the same Terraform configuration with different input values, making it adaptable to varying infrastructure requirements.

```
# Define variables
variable "vm_name" {
  description = "Name of the virtual machine"
  type        = string
}

variable "cpu_count" {
  description = "Number of CPUs for the virtual machine"
  type        = number
  default     = 2
}
```

```
variable "memory_size" {
  description = "Amount of memory in MB for the virtual
                machine"
  type        = number
  default     = 4096
}

variable "disk_size" {
  description = "Size of the virtual disk in GB"
  type        = number
  default     = 20
}

# VMware virtual machine resource
resource "vsphere_virtual_machine" "example_vm" {
  name             = var.vm_name
  cpu_number       = var.cpu_count
  memory           = var.memory_size
  # ... other configuration options for the virtual machine
}
```

In the previous example, we define four variables: `vm_name`, `cpu_count`, `memory_size`, and `disk_size`. Each variable has a description to provide clarity on its purpose. The `type` attribute specifies the data type of the variable, such as string or number.

The default attribute sets default values for the variables. If no value is provided when executing Terraform commands, these default values will be used. For example, if no value is passed for `cpu_count`, it will default to 2.

Within the `vsphere_virtual_machine` resource block, we reference the variables using the `var` prefix. This allows the values provided for the variables to be interpolated dynamically when provisioning the virtual machine. To use this configuration, you can execute Terraform commands and provide values for the variables. For example:

```
terraform plan -var="vm_name=web-server" -var="cpu_count=4"
```

In the previous command, we override the `vm_name` variable with the value `web-server` and the `cpu_count` variable with the value 4. This way, you can customize the virtual machine's attributes based on your specific needs. By utilizing variables in this manner, you can create flexible and reusable Terraform configurations for provisioning VMware infrastructure.

Maps

Maps in Terraform allow you to define a collection of key-value pairs, where each key is unique. Maps are useful for organizing and managing data in a structured manner. Let's explore how maps can be utilized in a Terraform configuration for VMware.

In the context of VMware infrastructure provisioning with Terraform, maps can be employed to define and manage various properties of virtual machines or other resources. Let's talk about how maps can be leveraged.

Defining a Map

In your Terraform configuration file, you can declare a map variable using the variable block.

For example, you can define a map called `vm_config` to hold the configuration details of VMware virtual machines:

```
variable "vm_config" {
  description = "Configuration details of VMware virtual
                machines"
  type        = map
}
```

Assigning Values to the Map

Map values can be assigned directly in the configuration file or through other mechanisms such as command-line arguments or environment variables.

For instance, you can define the map values for `vm_config` as follows:

```
vm_config = {
"web-server" = {
cpu_count   = 2
memory_size = 4096
disk_size   = 20
}
"database-server" = {
cpu_count   = 4
memory_size = 8192
disk_size   = 100
}
}
```

Accessing Map Values

You can access specific values within the map using the key associated with each value. For example, to access the `cpu_count` value for the `web-server` virtual machine, use this:

```
resource "vsphere_virtual_machine" "example_vm" {
  name             = "web-server"
  cpu_number       = var.vm_config["web-server"]["cpu_count"]
  memory           = var.vm_config["web-server"]["memory_size"]
  disk_size        = var.vm_config["web-server"]["disk_size"]
  # ... other configuration options
}
```

Dynamically Populating Maps

Maps can be dynamically populated using interpolation and other Terraform features. For instance, you can use a loop to iterate over a list and generate a map with dynamic values based on specific criteria or conditions.

Lists

Lists in Terraform allow you to define ordered collections of values. Unlike maps, lists do not have associated keys and are primarily used to store multiple related values. In the context of VMware infrastructure provisioning with Terraform, lists can be useful for defining arrays of properties or configurations for virtual machines or other resources. Let us explore how lists can be utilized.

Defining a List

In your Terraform configuration file, you can declare a list variable using the variable block.

For example, you can define a list called `datastore_clusters` to store the names of VMware datastore clusters.

```
variable "datastore_clusters" {
  description = "Names of VMware datastore clusters"
  type        = list(string)
}
```

Assigning Values to the List

List values can be assigned directly in the configuration file or through other mechanisms such as command-line arguments or environment variables. For instance, you can define the list values for `datastore_clusters` as follows:

```
datastore_clusters = ["cluster1", "cluster2", "cluster3"]
```

Accessing List Values

You can access specific values within the list using the index position. For example, to access the second element in the `datastore_clusters` list, use this:

```
resource "vsphere_virtual_machine" "example_vm" {
  name             = "web-server"
  datastore_cluster = var.datastore_clusters[1]
  # ... other configuration options
}
```

Dynamically Generating Lists

Lists can be dynamically generated using interpolation and other Terraform features. For example, you can use a loop to iterate over a map and extract specific values to populate a list.

Variable Defaults

In Terraform, we can assign default values to variables, ensuring that our configuration works even if a user does not explicitly provide a value. For example, consider a scenario where we are deploying VMware virtual machines and want to set a default network interface. By assigning a

default value to the variable representing the network interface, we ensure that the configuration remains functional even if the user does not specify a custom value.

To assign a default value to a variable, we can use the `default` parameter in the variable declaration. If a user provides a value for that variable during runtime, the user-defined value takes precedence over the default value.

Defining a Variable with Default

In your Terraform configuration file, you can define a variable with a default value using the variable block. For example, you can define a variable called `vm_cpu_count` with a default value of 2.

```
variable "vm_cpu_count" {
  description = "Number of CPUs for virtual machine"
  type        = number
  default     = 2
}
```

Using the Default Value

If a value is not explicitly provided for the variable at runtime, Terraform will use the default value. For instance, if the `vm_cpu_count` variable is not set during execution, Terraform will automatically use the default value of 2.

Overriding the Default Value

Users can override the default value by explicitly providing a value for the variable during runtime.

For example, when executing Terraform commands, users can set the value of `vm_cpu_count` to a different number, which will override the default value. Here's an example of using variable defaults in a VMware configuration:

```
variable "vm_cpu_count" {
  description = "Number of CPUs for virtual machine"
  type        = number
  default     = 2
}

resource "vsphere_virtual_machine" "example_vm" {
  name       = "web-server"
  cpu_count  = var.vm_cpu_count
  # ... other configuration options
}
```

In the previous example, if the `vm_cpu_count` variable is not provided during runtime, Terraform will use the default value of 2. However, if a value is provided explicitly, it will override the default value.

Populating Variables

When working with Terraform for VMware infrastructure provisioning, you can populate variables in multiple ways. This flexibility allows you to customize and configure your Terraform configuration based on different scenarios and requirements. Let's explore the various methods of populating variables in Terraform with a VMware example.

Command-Line Flags

You can populate variables by passing values directly through command-line flags when executing Terraform commands. For example, to set the value of a variable named datastore during runtime, you can use the `-var` flag:

```
terraform apply -var="datastore=example_datastore"
```

Environment Variables

Another approach is to populate variables using environment variables. Terraform automatically reads environment variables with a specific naming convention (`TF_VAR_variable_name`).

For example, you can set the value of a variable named network using an environment variable.

```
export TF_VAR_network="example_network"
```

Variable Files

Terraform allows you to store variable values in separate files, known as *variable files*. These files typically have a `.tfvars` or `.tfvars.json` extension. You can populate variables by creating a variable file and specifying its location when executing Terraform commands. For example, create a file named `variables.tfvars` with the following content:

```
datastore = "example_datastore"
network   = "example_network"
```

Then, when executing Terraform commands, specify the variable file using the `-var-file` flag.

Interactive Prompts

Terraform can prompt you to enter variable values interactively during runtime if a variable is not populated by any of the previous methods. For example, if the variable cluster is not populated, Terraform will prompt you to enter its value. These are some common methods of populating variables in Terraform for VMware infrastructure provisioning. You can choose the approach that best suits your workflow and requirements. Additionally, you can also combine these methods to handle complex configurations and provide flexibility in managing your variables.

Example Usage in VMware Configuration

Here's an example of populating variables using a combination of command-line flags and a variable file:

```
terraform apply -var="datastore=example_datastore"
```

```
export TF_VAR_network="example_network"
```

```
datastore = "example_datastore"
network   = "example_network"
```

```
variable "datastore" {
  description = "Name of the VMware datastore"
  type        = string
}
```

```
variable "network" {
  description = "Name of the VMware network"
  type        = string
}
```

```
resource "vsphere_virtual_machine" "example_vm" {
  name       = "web-server"
  datastore  = var.datastore
  network    = var.network
  # ... other configuration options}
```

When executing Terraform commands, you can populate the variables using the methods described earlier to customize the configuration for your specific VMware environment.

Leveraging Modularization in Terraform

One of the key features with Terraform is the ability to organize and reuse configurations using modules. In this blog, we will delve into the concept of modules in Terraform, exploring how they can improve the efficiency and scalability of infrastructure provisioning.

Introduction to Modules

Modules in Terraform allow you to encapsulate reusable infrastructure configurations. They provide a way to organize and package resources, variables, and other elements together, enabling efficient reuse across different environments and projects.

To create a module in Terraform, you need to create a separate directory with a specific structure. Let us create a module for provisioning VMware virtual machines:

- Create a directory named `vm_module`.
- Inside the `vm_module` directory, create a file named `main.tf` with the following content:

```
resource "vsphere_virtual_machine" "example_vm" {
  name             = var.vm_name
  resource_pool_id = data.vsphere_resource_pool.pool.id
  datastore_id     = data.vsphere_datastore.
  datastore.id
  template_uuid    = data.vsphere_virtual_machine.
  template.id
  num_cpus         = var.num_cpus
  memory           = var.memory
  network_interface {
    network_id = data.vsphere_network.network.id
  }
}
```

In this example, we define a module that provisions a VMware virtual machine. The module has input variables `vm_name`, `num_cpus`, and `memory` that allow customization for each instance of the module. The module uses the `vsphere_virtual_machine` resource to create the VM, with the specified variable values.

Module Structure

A Terraform module is essentially a directory containing one or more Terraform configuration files. The module directory should have a specific structure, including `main.tf`, `variables.tf`, and `outputs.tf` files, among others. The `main.tf` file defines the resources and configurations for the module, while `variables.tf` defines the input variables that can be customized for each module instance. The `outputs.tf` file specifies the values that will be exposed by the module for use in other configurations.

The structure of the module directory should resemble the following:

```
vm_module/
├── main.tf
```

This is a basic structure for a module, and you can add other files like `variables.tf` or `outputs.tf` based on your requirements.

Using a Module

To use the module we defined in the previous section, follow these steps: first, create a new Terraform configuration file (e.g., `main.tf`).

In the new configuration file, reference the module by providing the module source and any required input variables.

Here's an example:

```
module "my_vm" {
source  = "./vm_module"
vm_name = "my-vm"
num_cpus = 2
memory   = 4096
}
```

In this example, we use the module `vm_module` located in the local directory. We provide values for the input variables `vm_name`, `num_cpus`, and `memory`. Terraform will use the module configuration to provision the VMware virtual machine.

The previous was some brief insight into modules with Terraform. It should give you an idea about how the back-end complexity of defining each and every parameter in the configuration file (main.tf) can be masked and end users can simply assign the required variable values and exercise the code reusability that come from Terraform modules.

Streamlining Infrastructure Provisioning with Terraform

Here, we will focus on the steps involved in committing a configuration file, initializing Terraform, and applying the configuration to provision resources. We will take an example of a VMware resource provisioning.

Committing the Configuration File

To get started with Terraform and VMware, you need to create a configuration file that defines the desired state of your infrastructure. Let us assume we want to provision a virtual machine (VM) on VMware using Terraform.

Here's an example of `main.tf`:

```
provider "vsphere" {
  user              = var.vsphere_username
  password          = var.vsphere_password
  vsphere_server    = var.vsphere_server
  allow_unverified_ssl = true
}

resource "vsphere_virtual_machine" "example_vm" {
  name              = "my-vm"
  resource_pool_id = data.vsphere_resource_pool.pool.id
  datastore_id      = data.vsphere_datastore.datastore.id
  template_uuid     = data.vsphere_virtual_machine.template.id
  num_cpus          = 2
  memory            = 4096
  network_interface {
    network_id = data.vsphere_network.network.id
  }
}
```

In this example, we define a VMware provider configuration and a virtual machine resource using the `vsphere_virtual_machine` resource type. We provide the necessary details such as the VM name, resource pool, datastore, template, CPU, memory, and network configuration.

Initializing Terraform

Once you have your configuration file ready, the next step is to initialize Terraform. This step ensures that Terraform downloads the necessary provider plugins and sets up the working directory.

Here is the command:

```
terraform init
```

Running the Terraform `init` command initializes the Terraform working directory, fetching the required provider plugins based on the providers defined in your configuration file.

terraform plan

After Terraform initializes, the next step is to run a Terraform plan, which identifies the changes that Terraform is planning to incorporate in the infrastructure resource. You need to carefully look at the plan, and if any changes that are not desired, then you would need to adopt those changes accordingly in your configuration file and run `terraform plan` again.

Here's the command:

```
terraform plan
```

`terraform plan` does not affect the resources; it just shows you the changes that `terraform plan` brings to the infrastructure.

terraform apply

Once you approve the changes displayed by `terraform plan`, you can apply those changes, and Terraform with the help of the provider will start bringing in the desired changes in the destination platform.

Here's the command:

```
terraform apply
```

Once the changes are successfully applied on the destination platform, the Terraform state file is automatically updated to reflect the current point in time for future changes that the user may apply on the resource.

Hands-On Exercise: Generation of Config and State Files to Create a VMware VM via vCenter (Using Templates)

Here's a step-by-step guide with examples for creating a VMware VM via vCenter using templates with Terraform. This hands-on exercise assumes that the steps to install Terraform executables and VMware providers are already executed, as explained in the hands-on exercise of Chapter 1.

1. Define the configuration file (`main.tf`) with different sections like provider, data, resource, etc. You can define other files as well such as `variables.tf` and `output.tf`; however, for simplicity, we just define one complete file (`main.tf`). The following is the complete config file sample for reference.

   For an exercise, this file can be downloaded from the following GitHub repository:

   https://github.com/sumitbhatia1986/Terraform-VMware-ReverseEngineering/blob/main/Sample_Config_File

```
terraform {
  required_providers {
    vsphere = {
      source = "local/hashicorp/vsphere"
      version = "2.3.1"
    }
  }
}

provider "vsphere" {
vsphere_server = "x.x.x.x"
user = "administrator@vsphere.local"
password = "xxxxxxx"
#if you have a self-signed cert
allow_unverified_ssl = true
}

data "vsphere_datacenter" "dc" {
  name = "Your DC Name"
}

data "vsphere_compute_cluster" "cluster" {
  name          = "Your cluster name"
  datacenter_id = "${data.vsphere_datacenter.dc.id}"
}
data "vsphere_datastore" "datastore" {
name = "Datastore name needed for VM deployment"
datacenter_id = "${data.vsphere_datacenter.dc.id}"}

data "vsphere_network" "network" {
name = "VM Network"
datacenter_id = "${data.vsphere_datacenter.dc.id}"
}
```

```
data "vsphere_virtual_machine" "template" {
  name          = "Ubantu" #Your template name
  datacenter_id = "${data.vsphere_datacenter.dc.id}"
}

resource "vsphere_virtual_machine" "Ubantu"{
  name             = "UbantuTest"   #VMname for
                     deployment
  resource_pool_id = "${data.vsphere_compute_cluster.
                     cluster.resource_pool_id}"
  datastore_id     = "${data.vsphere_datastore.
                     datastore.id}"

  num_cpus = 4
  cpu_hot_add_enabled = "true"
  memory   = 12288
  memory_hot_add_enabled = "true"
  wait_for_guest_net_timeout = 0
  wait_for_guest_ip_timeout = 0
  guest_id = "${data.vsphere_virtual_machine.template.
             guest_id}"
  scsi_type = "${data.vsphere_virtual_machine.template.
              scsi_type}"

  network_interface {
    network_id = "${data.vsphere_network.network.id}"
 adapter_type = "${data.vsphere_virtual_machine.
                template.network_interface_types[0]}"
  }

  disk {
    label            = "disk0.vmdk"
    size             = "${data.vsphere_virtual_machine.
                       template.disks.0.size}"
```

```
  eagerly_scrub = ${data.vsphere_virtual_machine.
                  template.disks.0.eagerly_scrub}"
  thin_provisioned = "${data.vsphere_virtual_
                     machine.template.disks.0.thin_
                     provisioned}"}

   clone {
     template_uuid = "${data.vsphere_virtual_machine.
                     template.id}"
     customize {
       linux_options {
         host_name = "Terraform-test"
         domain = "test.internal"
       }
       network_interface {
         ipv4_address = "x.x.x.x" #IP you want to assign
         ipv4_netmask = 24
       }
     ipv4_gateway = "x.x.x.x"
}}}
```

2. Install Terraform and the VMware vSphere provider. Download and install the latest version of Terraform from the official website. https://www.Terraform.io/downloads.html

   Install the VMware vSphere provider plugin for Terraform by running the following command:

   ```
   terraform init
   ```

   The `init` command initializes the Terraform project and downloads the necessary plugins.

3. Run a Terraform plan to identify the configuration that is going to be deployed. The plan command shows a preview of the changes that Terraform will make to the infrastructure.

```
root@ubantu:/home/user/te# terraform plan
data.vsphere_datacenter.dc: Reading...
data.vsphere_datacenter.dc: Read complete after 0s
[id=datacenter-1001]
data.vsphere_network.network: Reading...
data.vsphere_datastore.datastore: Reading...
data.vsphere_compute_cluster.cluster: Reading...
data.vsphere_virtual_machine.template: Reading...
data.vsphere_network.network: Read complete after 0s
[id=network-1035]
data.vsphere_datastore.datastore: Read complete after
0s [id=datastore-1029]
data.vsphere_compute_cluster.cluster: Read complete
after 0s [id=domain-c1006]
data.vsphere_virtual_machine.template: Read complete
after 0s [id=4215b623-df65-ae56-72f6-4f7001ae46a3]
terraform used the selected providers to generate
the following execution plan. Resource actions are
indicated with the following symbols:
  + create
terraform will perform the following actions:
  # vsphere_virtual_machine.Ubantu will be created
  + resource "vsphere_virtual_machine" "Ubantu" {
      + annotation                                = (known
                                                    after
                                                    apply)
      + boot_retry_delay                          = 10000
```

```
      + change_version                          = (known
                                                  after
                                                  apply)
      + cpu_hot_add_enabled                     = true
      + cpu_limit                               = -1
      + cpu_share_count                         = (known
                                                  after
                                                  apply)
      + cpu_share_level                         = "normal"
      + datastore_id                            = "
        datastore-
        1029"
      + default_ip_address                      = (known
                                                  after
                                                  apply)
      + ept_rvi_mode                            =
"automatic"
      + extra_config_reboot_required            = true
      + firmware                                = "bios"
      + folder                                  = "vm"
      + force_power_off                         = true
      + guest_id                                =
        "ubuntu64Guest"
      + guest_ip_addresses                      = (known
                                                  after
                                                  apply)
      + hardware_version                        = (known
                                                  after
                                                  apply)
```

```
      + host_system_id                 = (known
                                         after
                                         apply)
      + hv_mode                        = "hvAuto"
      + id                             = (known
                                         after
                                         apply)
      + ide_controller_count           = 2
      + imported                       = (known
                                         after
                                         apply)
      + latency_sensitivity            = "normal"
      + memory                         = 12288
      + memory_hot_add_enabled         = true
      + memory_limit                   = -1
      + memory_share_count             = (known
                                         after
                                         apply)
      + memory_share_level             = "normal"
      + migrate_wait_timeout           = 30
      + moid                           = (known
                                         after
                                         apply)
      + name                           =
"UbantuTest"
      + num_cores_per_socket           = 1
      + num_cpus                       = 4
      + power_state                    = (known
                                         after
                                         apply)
      + poweron_timeout                = 300
```

```
      + reboot_required                          = (known
                                                   after
                                                   apply)
       + resource_pool_id                         =
         "resgroup-1007"
      + run_tools_scripts_after_power_on         = true
      + run_tools_scripts_after_resume           = true
      + run_tools_scripts_before_guest_shutdown = true
      + run_tools_scripts_before_guest_standby   = true
      + sata_controller_count                    = 0
      + scsi_bus_sharing                         = "noSharing"
      + scsi_controller_count                     = 1
      + scsi_type                                 =
"lsilogic"
      + shutdown_wait_timeout                     = 3
      + storage_policy_id                         = (known
                                                   after
                                                   apply)
      + swap_placement_policy                    = "inherit"
      + tools_upgrade_policy                      = "manual"
      + uuid                                      = (known
                                                   after
                                                   apply)
      + vapp_transport                            = (known
                                                   after
                                                   apply)
      + vmware_tools_status                       = (known
                                                   after
                                                   apply)
```

```
+ vmx_path                                = (known
                                            after
                                            apply)
+ wait_for_guest_ip_timeout               = 0
+ wait_for_guest_net_routable             = true
+ wait_for_guest_net_timeout              = 0
+ clone {
    + template_uuid = "
      4215b623-df65-ae56-72f6-
      4f7001ae46a3"
    + timeout       = 30
    + customize {
        + ipv4_gateway = "x.x.x.x"
        + timeout      = 10
         + linux_options {
             + domain       = "test.internal"
             + host_name    = "Terraform-test"
             + hw_clock_utc = true
           }
         + network_interface {
             + ipv4_address = "x.x.x.x"
             + ipv4_netmask = 24
           }
       }
  }
+ disk {
    + attach            = false
    + controller_type   = "scsi"
    + datastore_id      = "<computed>"
    + device_address    = (known after apply)
    + disk_mode         = "persistent"
```

```
            + disk_sharing      = "sharingNone"
            + eagerly_scrub     = false
            + io_limit          = -1
            + io_reservation    = 0
            + io_share_count    = 0
            + io_share_level    = "normal"
            + keep_on_remove    = false
            + key               = 0
            + label             = "disk0.vmdk"
            + path              = (known after apply)
            + size              = 16
            + storage_policy_id = (known after apply)
            + thin_provisioned  = false
            + unit_number       = 0
            + uuid              = (known after apply)
            + write_through     = false
          }
        + network_interface {
            + adapter_type          = "vmxnet3"
            + bandwidth_limit       = -1
            + bandwidth_reservation = 0
            + bandwidth_share_count = (known after apply)
            + bandwidth_share_level = "normal"
            + device_address        = (known after apply)
            + key                   = (known after apply)
            + mac_address           = (known after apply)
            + network_id            = "network-1035"
          }
      }
Plan: 1 to add, 0 to change, 0 to destroy.
```

4. Validate the Terraform plan output and see if it is as per the desired configuration; you are looking for a VM deployment. After validation, you can perform `terraform apply`.

   The `apply` command applies the changes and creates the virtual machine in the specified VMware vSphere environment.

   You would need to confirm and type `Yes` to the deployment question asked.

```
Plan: 1 to add, 0 to change, 0 to destroy.
Do you want to perform these actions?
Terraform will perform the actions described above.
Only 'yes' will be accepted to approve.
Enter a value: Yes
Plan: 1 to add, 0 to change, 0 to destroy.
Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.
  Enter a value: yes
vsphere_virtual_machine.Ubantu: Creating...
vsphere_virtual_machine.Ubantu: Still creating...
[10s elapsed]
vsphere_virtual_machine.Ubantu: Still creating...
[20s elapsed]
vsphere_virtual_machine.Ubantu: Still creating...
[30s elapsed]
vsphere_virtual_machine.Ubantu: Still creating...
[40s elapsed]
vsphere_virtual_machine.Ubantu: Still creating...
[50s elapsed]
```

```
vsphere_virtual_machine.Ubantu: Creation complete after
53s [id=4215dd8a-20f6-3d0b-d350-c1376f9621cc]
Apply complete! Resources: 1 added, 0 changed, 0
destroyed.
root@ubantu:/home/user/te#
```

5. Verify the virtual machine.

   Log in to the VMware vSphere environment to verify that the virtual machine was created successfully.

   You can also use the `terraform show` command to view the current state of the infrastructure in the Terraform state file.

Summary

In this chapter, we embarked on a comprehensive exploration of Terraform, delving into its core functionalities and advanced capabilities. We began by reinforcing the significance of infrastructure as code and the key concepts related to Terraform. Building upon this foundation, we ventured into the heart of Terraform, uncovering its configuration syntax and resource management capabilities.

The chapter further expanded its scope by addressing critical aspects such as dependencies, provisioners, and state management, crucial elements that ensure the successful orchestration of complex infrastructure deployments. We also delved into the versatility of Terraform modules, enabling readers to efficiently organize and reuse configurations, thereby streamlining their workflow. Moreover, we shed light on the importance of variables and data sources, providing the means to create dynamic and flexible configurations tailored to specific requirements. Understanding the intricacies of remote state management and various Terraform back ends facilitated collaboration and scalability, essential elements in modern IT environments.

CHAPTER 3

The Basics of Reverse Engineering

In the preceding chapters, we introduced infrastructure as code (IaC) and Terraform. Additionally, we covered related fundamental ideas, such as state files, configuration files, and Terraform plans. In this chapter, we will first discuss the Terraform workflow and its associated challenges. Then, we will focus on understanding reverse-engineering practices and how these practices can help you mitigate some common Terraform challenges.

To further strengthen your understanding of reverse engineering, we will introduce you to a sample use case in which we will build a sample model to work with existing VMware virtual machines. Finally, in the hands-on exercise, we will present some sample Python logic to programmatically fetch point-in-time VMware VM object values, which lays the foundation for reverse engineering.

Terraform Workflow Overview

Terraform offers powerful automation capabilities for almost every infrastructure technology available in the market. Say components are present in a compute technology space such as VMware, the public cloud, or a containerization technology space such as Kubernetes, etc. Terraform can enable IT administrators to define their IaC. In fact, organizations

S. Bhatia and C. Gabhane, *Reverse Engineering with Terraform*,
https://doi.org/10.1007/979-8-8688-0074-0_3

are increasingly relying on Terraform and using it as a common platform to ease their infrastructure management. Using a single platform for infrastructure management comes with the following benefits:

- Ease of management
- Easy to train the IT workforce because they need to learn only one technology
- Cost savings as organizations have to invest in only one technology
- Centralized administration saves time and effort

Let's understand the Terraform functionality for infrastructure automation at a very high level. Figure 3-1 shows how Terraform operates in different deployments and its interactions for infrastructure automation. This is followed by a more detailed analysis.

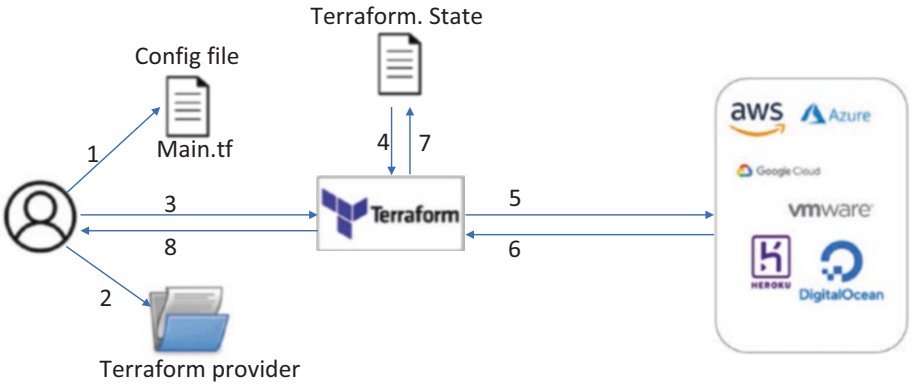


Figure 3-1. *Overview of Terraform workflow*

1. **Config file modification:** Users first need to define their infrastructure as code in HashiCorp Configuration Language (HCL) or in JSON format. This file is called a *configuration file* (or *config*

file). It is a file that is accessible to end users, and they usually play with this file to attain the desired configuration change on the resources respective to a platform. There are required mandatory and optional parameters that need to be populated in this config file. It is this configuration file that enables IT admins to define IaC. Changes in this config file determine the future state of the resource after `terraform apply`.

2. **Terraform provider:** This is an integral part of Terraform functioning that enables the Terraform core to operate with different infrastructure technologies. For example, the Google Cloud Platform (GCP) has a separate provider for VMware and so on. For interaction with the desired platform, the Terraform provider has built-in logic to facilitate the interaction of the source code. These providers are built by either open-source communities, OEM vendors, or HashiCorp.

3. **Terraform plan:** Users bring in changes to the infrastructure that provide a config file to the Terraform core. If the state file is not already there, then it's considered a new provisioning. This plan lists the user's proposed changes on the desired resource running on the destination platform.

4. **Delta with Terraform state file and input Config file:** If a state file is present, then the delta changes are identified by finding the difference in the current config file that is defined by the user and the old known state of the workload. These delta

changes identify the specific modification that is to be invoked on the workload respective to the destination infrastructure platform.

5. **Terraform apply:** Once a user approves the changes identified between the config and state file, the changes are submitted to the destination infrastructure platform. Destination platforms are the Terraform-supported platforms such as AWS, Azure, Google, VMware, etc.

6. **Tracking of changes:** The Terraform core keeps track of the changes that are invoked on the infrastructure platform. It is engaged until the given operation either succeeds or fails.

7. **State file updating:** If the changes are successful, the Terraform core updates the state file to reflect the current state of the resource. This is required to record the last known good state of the resource. The state file is ideally not to be accessed by the end users. This is an integral part of Terraform operations.

8. **User acknowledgement:** Once the changes are successfully implemented on the platform and the state file is updated, the Terraform core acknowledges to the end user that it was a successful operation.

Note that where there are no changes found between the config file and state file, then the Terraform plan shows no changes required to the platform; in that scenario, the workflow described is different because no changes are done on the infrastructure. The previous workflow is for when there are changes to be applied on the destination platform.

With an understanding of this basic workflow, you can see how Terraform depends on the state file to record the last known good state of the resource it manages. In the next section, we are going to introduce certain shortcomings associated with the fact that Terraform needs to maintain a state file for it to function.

Terraform and Its Shortcomings

Keeping Terraform as a single platform for all the infrastructure automation needs is considered valuable, but it comes with different challenges as well. Let's discuss some of these challenges in brief that are associated with the tool.

Terraform Dependence on Point-in-Time Config Files for Import Operations

The way Terraform operates is that the user first writes a config file in HCL or JSON that the Terraform tool understands. When we start with a workload life cycle, there is no state file present in the Terraform inventory already; it can provision a resource and maintain a state file in its inventory, which represents the last known state of a resource that Terraform deployed. This is the ideal scenario and usually occurs when we want to start the life cycle of a resource from scratch. However, a real problem is when we want to make use of Terraform with an already provisioned resource that has been running in our infrastructure for so long, even before Terraform was introduced. This is because Terraform does not know the current state of the resource. For Terraform to know the current state of a resource, we need to import a resource into Terraform, whereas the import operation is not a straightforward task. For you to import a resource into Terraform to manage its life cycle, you need to have an equivalent (to the current state and in HCL or JSON format)

well-defined config file, so a successful import operation can be performed. Generating a config file that matches the current state is a tedious job. If we must do this manually, it would be a nightmare for many of the IT administrators because there are several required parameters that they need to prepopulate in this config file and in defined format (HCL or Json) for a successful import of that resource. This is even more complicated when it is required to be done at scale.

This import operation makes it difficult to manage an existing resource with Terraform automation. There are requirements where existing applications and resources are required to be managed via automation.

Definition `terraform import` is a command that allows you to bring existing resources under Terraform management. It requires a config file as a parameter to help import a resource cleanly into Terraform automation.

Terraform Dependence on State File for Life-Cycle Management

Another challenge that was noticed in doing infrastructure operations with Terraform is that it is heavily dependent on the state file for managing the life cycle of a resource. When `terraform plan` is run, it basically identifies the delta between the user end file, i.e., the config file, and the last known state of the resource present in the Terraform inventory called the *state file*. The Terraform action wants to bring the state of a resource in accordance with the definitions present in a user-defined config file. If there is a difference between the config and state files, Terraform presents that as an actionable point that is to be executed on the platform.

This way of working is best when we want to have an immutable infrastructure and every aspect of the operations is handled by Terraform. However, in the real infrastructure operations world, it is imperative that administrators need log in to the platform for debugging and making direct changes on the platform that are necessary at times to fix priority issues. An example is when a VMware VM life cycle is managed by Terraform and an administrator logs into vCenter directly and changes the configuration of the VM. This action directly on the platform would invalidate the existing state file present in the Terraform inventory. As a result, when there is another change driven by a change in the config file, users would find changes that were done by the administrator, and they are forced to revert them because neither the state file nor the config file is aware of the changes that the administrator performed on the platform directly. Therefore, the state file that was maintained so far is invalidated.

To fix this, one solution is to revert the changes done by administrators directly on the platform. This is the least desired solution because changes made by administrators are for a purpose; reverting these changes is risky and can potentially reintroduce the problems fixed by administrators.

The best solution is to manually adopt the config file in such a way that it reflects the changes done by the administrator directly on the platform, so when `terraform plan` is run, the changes performed already on the platform do not reflect in the delta changes to be performed by `terraform apply`. This solution comes with a price of manually adjusting the config file every time a direct change is performed on the platform.

The challenges highlighted here may inhibit the adoption of Terraform in automating infrastructure operations with the tool.

Mitigating These Shortcomings

The solution is the reliable and automated import of the existing resource, which requires a consistent, point-in-time configuration file. If we can autogenerate a configuration file that is clean and consistent with the platform, we can then reliably import an existing resource and bring that infrastructure item under Terraform management.

As shown in Figure 3-2, to import an existing resource into Terraform, when a correct, point-in-time configuration file is presented for Terraform import, we can then successfully import existing resources and leverage Terraform automation to further manage the life cycle of the resource.

Figure 3-2. *Terraform import workflow*

This configuration file can be generated manually as well, but it should be discouraged for large environments as it is complex and error prone. Administrators looking to import existing resources into Terraform then would need to manually present each parameter value, run `terraform plan`, and identify if it is not suggesting any changes. If `terraform plan` suggests changes, then it means the import operation is not clean. Any execution with `terraform apply` can alter the existing resource configuration.

If we can automate the process of generating the configuration file, we will be able to handle the situation better. The dependency on maintaining a state file can be eliminated because we can fresh import a resource every time we anticipate any corruption in the state file.

In the next section, we are going to introduce reverse-engineering practices that help in finding the best solution to address these shortcomings.

What Is Reverse Engineering?

Before we explain how reverse engineering can help us mitigate the challenges, let's first understand more about this. Reverse engineering has been a common industry practice for a long time. Historically, it was used for analyzing hardware, for gaining military advantage, and even for learning about biological functions related to how genes work. Reverse engineering for software is typically for understanding the underlying source code for the maintenance and improvement of the current software.

The following are some popular definitions of reverse engineering:

> *"The reproduction of another manufacturer's product following detailed examination of its construction or composition."*
>
> *"Examine the construction or composition of another manufacturer's product in order to create (a duplicate or similar product)."*
>
> —Oxford dictionary

> *"The act of copying the product of another company by looking carefully at how it is made."*
>
> —Cambridge dictionary

In short, reverse engineering is a practice that enables you to determine how a product was designed so you can understand its function and based on your requirements develop solutions around it. Reverse engineering in information technology is used to address compatibility issues and make the hardware or software work with other hardware, software, or operating systems that it was not originally compatible with. Another great advantage of reverse engineering is that it helps in the

adoption of technologies and tools. This is because it empowers us to understand the fine details of the technology and tools so it is absorbed well in the proposed solution and way of working in day-to-day life.

There are multiple examples of such tools in the market that have reverse-engineering practices at their core. The famous one is computer-aided design (CAD), which basically helps re-create a manufactured part when the original blueprint is not available. The key feature of CAD is its ability to produce 3D images of the desired part so it can easily be remanufactured.

With help from reverse engineering, we can understand how Terraform works for a respective platform, how Terraform identifies the state of the resource it manages on the platform, and what the source of truth for Terraform itself is with regard to a particular platform. Answers to all of that can provide us with valuable information, and the same source of truth can be leveraged to generate the point-of-time configuration file. Before we look more into this, let's understand more about this reverse engineering process in the next section.

Reverse-Engineering Process for IT Infrastructure Tools

The process employed in reverse engineering is specific for each product. However, at a high level, all reverse-engineering processes mainly consist of three basic steps, as shown in Figure 3-3.

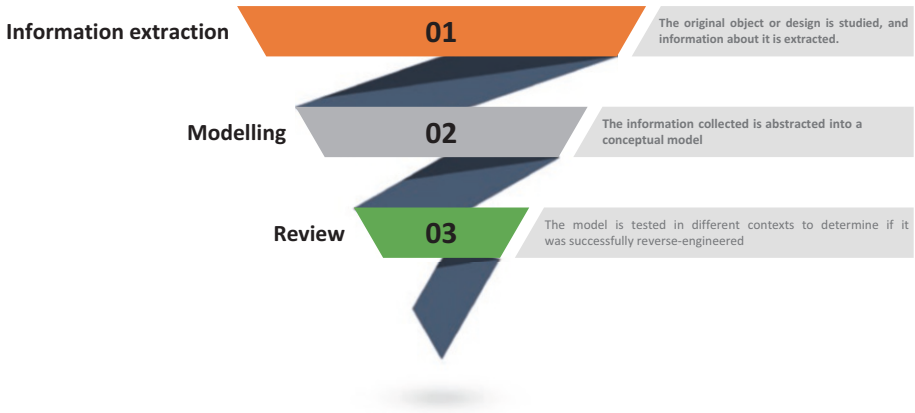


Figure 3-3. *Reverse-engineering process*

The steps are as follows:

1. **Information extraction**

   This is the first step of the reverse-engineering process and is linked to the gathering of information on how the current product performs. Information about its design, its operations, and its interactions is studied thoroughly. In software reverse engineering, a specific focus is on generating the sample source code and the design documents. The information that is extracted is important in understanding how the product and tool works, what the boundaries of its function are, and how it is implemented with different installations.

   Our case study later in this chapter will evaluate the following questions: What is the source of truth for the tool itself? How does it interact with VMware. How do config and state files play a critical role?

   The outcome of this step is a comprehensive understanding of how the product functions.

2. **Modeling**

   Now, the information that is extracted in the previous step needs to be depicted in such a way that it explains the function of the overall proposed structure. The essence of this step is to take the information specific to the original product and summarize it in a general model that can be used to design a new system that we conceptualize to fit into our big picture. In software reverse-engineering terms, this means making a data flow chart or just plain component interactions that explain the model you initially hypothesized.

   With regard to our case study, this step is where the sample config file is modeled along with its required key parameters. Formalization on the process of fetching it with the source of truth (e.g., MOB for vSphere) and finally coming up with a model that interacts with the whole system to iterate the complete process programmatically and at large scale.

3. **Review**

   Once the final resultant model is designed, in this stage it is time to test it to ensure it is a realistic abstraction of our original aspirations. In software reverse-engineering terms, repeating the suggested operations in a sample lab environment can ensure it can deal with the desired complex and unique scenarios.

   In our sample case study of VMware, this step is to review the automated import operations of a diverse set of VMware VM workloads by autogenerating the config file and importing it into the Terraform

tool. For example, with diverse configuration of the VM workloads in terms of their operating system types, storage, network configuration, etc., all these specifics relevant to the VM should be fetched and tested for variety of different configurations present in your specific environment.

These steps depict the process of doing reverse engineering with a platform. To adopt Terraform automation to existing infrastructure resources, we can perform these steps and come up with our own logic script that enables us to import existing resources into the Terraform tool, and we can do this operation at scale. In the next section, we will specifically apply what we learned in this section to autogenerating a config file required for Terraform import.

Reverse Engineering with Terraform and Its Benefits

Reverse engineering with Terraform includes employing the set of standard practices we discussed in the previous section to autogenerate a configuration file for the benefit of our use. Figure 3-4 is a high-level overview of the reverse-engineering practices that we can employ to write import logic to generate a clean and point-in-time configuration file.

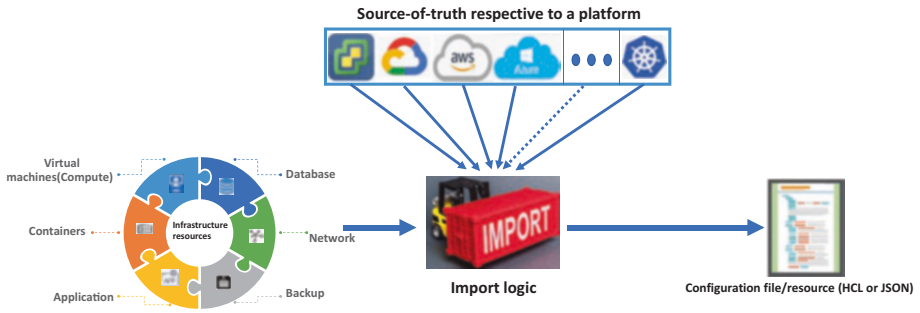


Figure 3-4. *Source of truth and Terraform import*

The focus of reverse engineering is to generate a configuration file by determining the "source of truth" relevant to a platform. Every platform, whether it's a public cloud (e.g., Amazon, Google, Azure) or a private cloud (e.g., VMware) or Kubernetes containers, maintains a resource object inventory (metadata) that is true at any point in time and exposed over an application programming interface (API). This metadata can be programmatically fetched to generate a valid configuration file required for Terraform import.

Once the source of truth is known for the platform, a logic script can be written in any known language (e.g., Python, PowerShell) to fetch the required point-in-time parameter values for the automated generation of a configuration file. A script can be created with custom logic.

Here are some examples of resource object inventory databases, which are repositories of metadata for an IT resource that is maintained by every platform:

1. **VMware:** For VMware infrastructure, the resource object inventory is called the *managed object browser* (MOB) and is accessible via `https://<vcenterIP>/mob`.
2. **Azure:** Different resource APIs offered by the platform offer access to metadata objects inventory.
3. **Google Cloud:** Resource-manager APIs and Cloud Asset inventory in Google Cloud offer access to resource object inventory.
4. **Amazon Web services (AWS):** The AWS systems manager inventory provides access to metadata for the resources provisioned in the cloud.

Once the source of truth is known, a logic script can be written to fetch the values of desired objects required for the clean import of a resource into Terraform.

As per reverse-engineering practices, once the source of truth is identified, we then need to identify the mandatory required object parameters along with optional parameters (that suits most of your deployment) for a typical resource configuration file. This can be done via a thorough study of the required Terraform provider. Take the example of a VMware vSphere Terraform provider; there are lists of mandatory objects that are required in a configuration file for typical `vsphere_virtual_machine` management. For a list of parameters, please refer to more details about the example provider objects here:

`https://registry.Terraform.io/providers/hashicorp/vsphere/latest/docs/resources/virtual_machine`

Examples of such mandatory parameters for a virtual machine are details such as `name`, `num_cpus`, and `memory`.

Optional parameters are `extra_config`, `hardware_version`, etc.

A list of these mandatory and optional parameters is essential to define the import logic and to fetch the values of these mandatory parameters automatically from the source of truth.

This is going to be a one-time exercise for the entire platform you want to manage with Terraform. Once these parameters are identified, you just need to run your import logic to fetch the values in the desired format (HCL or JSON). And you can use these files to auto-import existing resources into the Terraform automation.

In the next section, we will introduce the benefits of autogenerating configuration files.

Benefits of Autogenerating Configuration Files

Identifying the list of mandatory and optional parameters that are applicable to most of the resources deployed in your environment is a one-time exercise. Fetching a few extra objects and having them in a configuration file that represents the point-in-time state of a resource

does not do any harm. You will have problems with importing when there are fewer parameters that do not completely represent the true state of a resource.

The reverse-engineering process offers tremendous benefits to enable automation for your infrastructure resources. Let us discuss these benefits in detail.

Reliably import existing resources into Terraform and at scale: The output configuration files can be used by the IT administrators to enable the DevOps way of handling any existing application and bringing any resource under Terraform management.

Using the reverse-engineering approach, you can automatically generate the point-in-time config files, which exactly represent the current state of the workloads. Once we have a config file generated, we can import any workload into Terraform without the infrastructure being provisioned with Terraform first. In fact, this allows us to discard the current state and generate a fresh state file every time we import resources into the tool.

Remove dependence on maintaining state: With the import logic, the point-in-time state of the resource needed for the Terraform operation can be easily regenerated. Therefore, administrators can benefit from both worlds, i.e., employ Terraform to automate their L1 and L2 operations. At the same time, they can work directly on the platform without impacting Terraform functionality for L3 activities that are required to be performed directly on the platform. The reverse-engineering practice is helpful because with this approach, you are not dependent on the state file at all. Administrators can make changes directly on the platform, and at the same time for any infrastructure automation needs, fresh imports can be performed to leverage automation capabilities with Terraform.

Skip the Terraform learning curve: Creating a configuration file manually is an iterative process where you define some configuration parameters, apply them, and then verify if it reflects the same desired state of the resource. But when a configuration file is automatically generated, IT administrators do not have to spend time trying to understand each

and every configuration parameter needed to manage a resource via Terraform. The reverse-engineering logic script can be written by a subset group of Terraform specialists, and that can be shared with a wide number of administrators and application experts to allow them to use Terraform automation capabilities. These administrators and application experts can focus on their core development and tasks without having to do tedious tasks to learn detailed Terraform objects and configuration files to manage their deployments.

Better adoption of Terraform: Considering Terraform capabilities to cover a variety of infrastructure landscapes, the automated generation of configuration files makes it easy for administrators to better adapt the tool for a variety of their infrastructure automation needs. Whether the platform is modern (like a public cloud) or legacy infrastructure (like VMware VMs), this reverse-engineering practice allows you to use Terraform on practically any resource whether it was created with Terraform or already exists.

Cost savings: It allows the use of a common tool to manage the diversity of the infrastructure landscape with a common team.

You now understand the benefits of automatically generating configuration files and empowering administrators to leverage the Terraform infrastructure automation capabilities, bringing any traditional resource into Terraform management. In the next section, we will introduce a sample use case that we want to build in subsequent chapters to strengthen your understanding of reverse engineering. The methodology we are developing here can be easily replayed in any other platform where you want to do automation with help from Terraform.

Sample Use Case: Reverse Engineering a VMware VM

To strengthen the concept of reverse engineering in the upcoming chapters of this book, we are going to demonstrate how to leverage reverse-engineering practices and perform an automated import of a virtual machine running on the VMware platform.

Our lab environment is running VMware vCenter version 7.0 update 1. We have an Ubuntu machine deployed in our environment, and Terraform is not aware of this as there is no state already known to Terraform for this lab machine.

The use case focuses on the following steps. These steps are going to be covered in the upcoming chapters of this book.

1. Use the VMware MOB as a source of truth to get the point-in-time values of the VM object (`https://<Vcenter FQDN>/mob`).
2. Identify the key mandatory and optional objects for a typical VMware environment. These are key for our reverse-engineering approach.
3. Write an import logic script to programmatically fetch these object values from the MOB and create a file in a specified format (HCL or JSON) using these objects, which Terraform can understand. The file that is generated in this step is called a *configuration file*.
4. Do a `terraform import` with the configuration file and validate if the import was successful.

5. After a successful import, perform a change on the VMware VM leveraging Terraform automation by changing one of the parameters in the configuration file.

The above points cover end-to-end scenarios where any existing VMware virtual machine can be imported to leverage the automation capabilities. In this chapter, with the hands-on exercise, we are going to cover the use of the MOB to fetch the desired objects from the VMware object repository. We will write some sample Python logic that facilitates the programmatic interaction with the VMware platform.

The following are some key prerequisites before diving into hands-on exercises:

- An intermediate-level understanding of the Python language and different libraries required to perform the required automation on a platform
- An intermediate-level understanding of VMware virtualization
- Understanding of Terraform

Hands-On Exercise: Managed Object Browser in VMware (vCenter) as a Source of Reverse Engineering

To continue with our sample use case of reverse engineering a VMware VM into Terraform, we are going to discuss the MOB in this exercise and how it plays a key role.

The MOB is a web-based server application available in all ESX/ESXi and vCenter server systems. This vSphere utility allows you to view detailed information about metadata objects such as virtual machines, datastores, and resource pools.

Access to it is disabled by default. You need to enable MOB on the VMware system.

Access to MOB with the vCenter Server is typically via the following link:

```
https://<Vcenter FQDN>/mob
```

Figure 3-5 shows the landing page when we log in to the MOB.

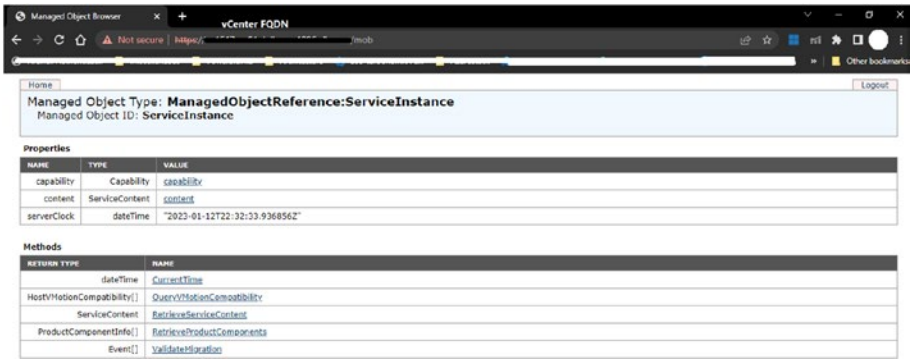


Figure 3-5. *MOB landing page*

You can browse through the desired objects you are managing in vCenter. For example, to look for a sample VM object, the following is the typical sample location: *Content ➤ rootFolder (Datacenters) ➤ childEntity (Cluster) ➤ vmFolder (VM) ➤ childEntity (Your VM)*.

For any sample VM, a managed object browser would look like Figure 3-6.

Home

Managed Object Type: **ManagedObjectReference:VirtualMachine**
Managed Object ID: **vm-3005**

Properties

NAME	TYPE	VALUE
alarmActionsEnabled	boolean	true
availableField	CustomFieldDef[]	
capability	VirtualMachineCapability	capability
config	VirtualMachineConfigInfo	config
configIssue	Event[]	
configStatus	ManagedEntityStatus	"green"
customValue	CustomFieldValue[]	
datastore	ManagedObjectReference:Datastore[]	datastore-1029
declaredAlarmState	AlarmState[]	declaredAlarmState["alarm-10.vm-3005"] AlarmState declaredAlarmState["alarm-112.vm-3005"] AlarmState declaredAlarmState["alarm-12.vm-3005"] AlarmState declaredAlarmState["alarm-128.vm-3005"] AlarmState declaredAlarmState["alarm-140.vm-3005"] AlarmState (more...)
disabledMethod	string[]	"vim.ManagedEntity.destroy" "vim.VirtualMachine.unregister" "vim.VirtualMachine.unmountToolsInstaller" "vim.VirtualMachine.answer" "vim.VirtualMachine.upgradeVirtualHardware" (more...)
effectiveRole	int[]	-1
environmentBrowser	ManagedObjectReference:EnvironmentBrowser	envbrowser-3005
guest	GuestInfo	guest
guestHeartbeatStatus	ManagedEntityStatus	"green"
layout	VirtualMachineFileLayout	layout
layoutEx	VirtualMachineFileLayoutEx	layoutEx
name	string	"UbantuTest"
network	ManagedObjectReference:Network[]	network-1035 (VM Network)

Figure 3-6. *Sample virtual machine metadata as seen via the MOB*

We can go one level deep into "config" to see further object details of a sample VM that we want to import.

Figure 3-7 shows the VM configuration of our Ubuntu VM.

Home

Data Object Type: **VirtualMachineConfigInfo**
Parent Managed Object ID: **vm-3005**
Property Path: **config**

Properties

NAME	TYPE	VALUE
alternateGuestName	string	""
annotation	string	""
bootOptions	VirtualMachineBootOptions	bootOptions
changeTrackingEnabled	boolean	false
changeVersion	string	"2023-02-18T00:35:43.697102Z"
consolePreferences	VirtualMachineConsolePreferences	Unset
contentLibItemInfo	VirtualMachineContentLibraryItemInfo	Unset
cpuAffinity	VirtualMachineAffinityInfo	Unset
cpuAllocation	ResourceAllocationInfo	cpuAllocation
cpuFeatureMask	HostCpuIdInfo[]	Unset
cpuHotAddEnabled	boolean	true
cpuHotRemoveEnabled	boolean	false
createDate	dateTime	"2023-02-15T22:20:25.372619Z"
datastoreUrl	VirtualMachineConfigInfoDatastoreUrlPair[]	datastoreUrl
defaultPowerOps	VirtualMachineDefaultPowerOpInfo	defaultPowerOps
extraConfig	OptionValue[]	extraConfig["tools.guest.desktop.autolock"] OptionValue extraConfig["nvram"] OptionValue extraConfig["pciBridge0.present"] OptionValue extraConfig["svga.present"] OptionValue extraConfig["pciBridge4.present"] OptionValue (more...)
files	VirtualMachineFileInfo	files
firmware	string	"bios"
flags	VirtualMachineFlagInfo	flags
forkConfigInfo	VirtualMachineForkConfigInfo	forkConfigInfo
ftEncryptionMode	string	"ftEncryptionOpportunistic"
ftInfo	FaultToleranceConfigInfo	Unset
guestAutoLockEnabled	boolean	false
guestFullName	string	"Ubuntu Linux (64-bit)"
guestId	string	"ubuntu64Guest"
guestIntegrityInfo	VirtualMachineGuestIntegrityInfo	guestIntegrityInfo

Figure 3-7. *Sample virtual machine configuration metadata as seen via the MOB*

Similarly, we can go to any level deep to identify the point-in-time values of almost any object managed by vCenter.

If you carefully look at the properties in Figure 3-7, many of them are mandatory parameters in a typical config file required for Terraform VMware VM management.

The idea of reverse engineering is to programmatically pick up the required live values from the MOB every time we want Terraform to manage our VM. We are going to present some sample Python code with which we can pick up the values of the following sample objects through the object browser:

- **VM name**: UbantuTest
- **VM guestFullName**: Ubuntu Linux (64-bit)
- **VM guestID**: ubuntu64Guest

Prerequisites for Python Code

The required libraries to talk to the managed object browsers of vSphere are pyVim and pyVmomi. If they are not already present, you can do a `pip` install of the required Python libraries.

```
sudo pip install pyvim
sudo pip install pyvmomi
```

Now, after installing the prerequisites, let's look at some sample Python code.

The following code can be accessed from the GitHub repository as well (https://github.com/sumitbhatia1986/Terraform-VMware-ReverseEngineering/blob/main/MOB_walkthrough.py):

```
# -*- coding: utf-8 -*-
"""
Created on Sat Feb 18 13:11:43 2023
@author: SBhatia3
"""
import requests
from requests.packages.urllib3.exceptions import
InsecureRequestWarning
from pyVim import connect
import ssl
from pyVmomi import vim
```

```
app_settings = {
    'api_pass': "xxxx",
    'api_user': "user@vsphere.local",
    'api_url': "https://<vCenterFQDN/rest/",
    'vcenter_ip': "vCenter IP",
    'VM_name': "UbantuTest" ### Desired VM for which we need to
               find details
}

## Authenticate with Vcenter to find the VM ID
def auth_vcenter(username, password):
    resp = requests.post(
        '{}/com/vmware/cis/session'.format(app_
        settings['api_url']),
        auth=(app_settings['api_user'], app_settings['api_
        pass']),
        verify=False
    )
    if resp.status_code != 200:
        print('Error! API responded with: {}'.format(resp.
        status_code))
        return
    return resp.json()['value']
def get_api_data(req_url):
    sid = auth_vcenter(app_settings['api_user'], app_
    settings['api_pass'])
    resp = requests.get(req_url, verify=False,
    headers={'vmware-api-session-id': sid})
    if resp.status_code != 200:
        print('Error! API responded with: {}'.format(resp.
        status_code))
        return
    return resp
```

```
## Getting VM details including VM ID needed to browser
through MOB
def get_vm(vm_name):
    resp = get_api_data('{}/vcenter/vm?filter.names={}'.
    format(app_settings['api_url'], vm_name))
    j = resp.json()
    return (j)

#Fetching VM ID from REST API for the VM we want to import
requests.packages.urllib3.disable_warnings(InsecureReque
stWarning)
vmdetails = get_vm(app_settings['VM_name'])
vmid = vmdetails['value'][0]['vm']

s = ssl._create_unverified_context()

service_instance = connect.SmartConnect(
    host=app_settings['vcenter_ip'], user=app_settings
    ['api_user'], pwd=app_settings['api_pass'], sslContext=s
)

content = service_instance.RetrieveContent()
container = content.rootFolder  # starting point to look into
viewType = [vim.VirtualMachine]  # object types to look for
recursive = True  # whether we should look into it recursively
containerView = content.viewManager.CreateContainerView
                (container, viewType, recursive)  # create
                container view
children = containerView.view
for child in children:  # for each statement to iterate all
names of VMs in the environment
    if (str(vmid) in str(child)):
        vm_summary = child.summary  #Summary of the desired VM
                     to import
```

```
        vm_config = child.config #Complete config data
                    hiararchy and child item values loaded in
                    the variable
        vm_resourcepool = child.resourcePool #Resource
                          pool details
        vm_network = child.network #Network details of the VM
        vm_datastore = child.datastore
        vm_parent = child.parent
        vm_name = child.name

print("VM name: ", vm_name)
print ("VM guestFullName: ", vm_config.guestFullName)
print("VM guestID: ", vm_config.guestId)
```

Here's the output:

```
runfile('C:/Users/Mob_WalkThrough')
VM name:  UbantuTest
VM guestFullName:  Ubuntu Linux (64-bit)
VM guestID:  ubuntu64Guest
```

The previous Python code is the sample representation of the reverse-engineering principle and how we can employ the VMware managed object browser to fetch the required parameters to generate a sample config file for every VM.

Here we fetched values for VM name, VM guestFullName, and VM guestID. Similarly, we can fetch values for all other mandatory and optional objects required in a typical configuration file.

Summary

The core focus in this chapter was reverse engineering. It started by explaining why reverse engineering is needed. We explained the Terraform automation workflow where an end user adjusts the infrastructure definitions in the configuration file and how the whole system integrates to reflect the changes on the destination platform. This workflow is dependent on the Terraform operations maintaining the last known state of the resource.

Further, the chapter provided details about the Terraform shortcomings associated with its implementation of automation. To solve these challenges, the chapter focused on reverse engineering and the different phases of it such as information extraction, modeling, and then review.

Toward the end, chapter explained the benefits of doing reverse engineering and automatically generating configuration files that are required for a Terraform import. The hands-on exercise walked you through a sample Python script that can fetch configuration file objects from the source of truth for VMware, also called the managed object browser (MOB). This laid a foundation for the example of reverse-engineering a VMware virtual machine.

CHAPTER 4

Terraform and Reverse Engineering

In the previous chapter, we studied the fundamentals of reverse engineering and its operation. Additionally, we discussed Terraform with you and the difficulties it presents when attempting to use it to automate infrastructure activities. In this chapter, we will cover every step required for using Terraform to accomplish this automation. We will examine each stage of the reverse-engineering process in detail, covering information extraction, modeling, and review. We'll use an import of a pre-existing VMware virtual machine as an example. We will highlight the sample reverse-engineering model with Terraform and the importance of using a "hook" to fetch the point-in-time state parameters required for successful import of a resource. We will also look at a typical import operation and build the import logic for an automated import.

In the hands-on exercise, we will provide a sample import script written in the Python language that can autogenerate a configuration file required for the successful import of a VMware virtual machine. Let's start this chapter with an explanation of the information extraction step, which is the initial stage of the reverse-engineering process.

S. Bhatia and C. Gabhane, *Reverse Engineering with Terraform*,
https://doi.org/10.1007/979-8-8688-0074-0_4

Information Extraction

In Chapter 3, we looked at the three basic steps of reverse engineering. The first step is information extraction, where we are required to gather information on how the product is designed and operated. In this section, we will dive a little more into the Terraform architecture and explain how that architecture can help model the information we need to further build a reverse-engineering solution. The architecture that will be discussed is generic and explains how the Terraform core functions and interacts with different infrastructure platforms. The outcome of this section should give you an idea of how Terraform operates with the "source of truth" we discussed in earlier chapters. See Figure 4-1.

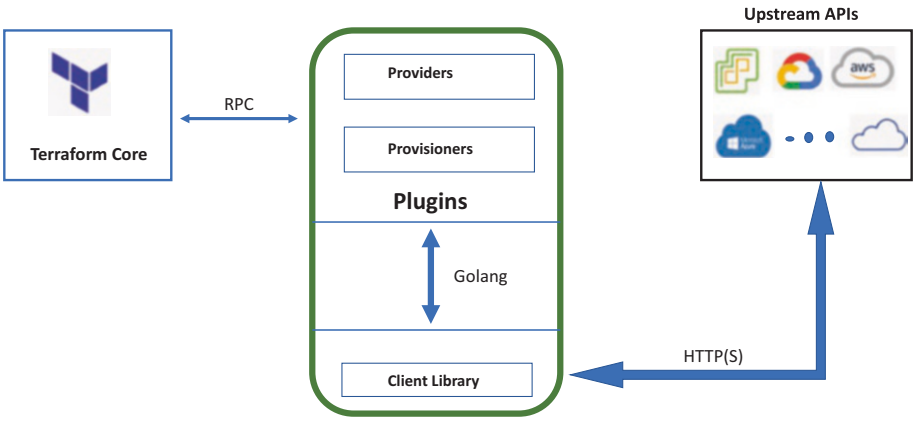


Figure 4-1. *Terraform architecture*

Terraform today supports multiple providers and more than 800 provider binaries. To manage these multiple provider binaries, HashiCorp would need to manage each of them. This would be difficult. Instead, HashiCorp has made the Terraform architecture an extensible architecture. This means the respective platforms provide support with Terraform core and maintain their own provider plugins and as well as life cycle of these provider plugins. Terraform offers the "provider SDK," which

is a software development kit that defines how the Terraform core interacts with the plugins that are to be written by a specific platform that wants to support Terraform. Further, to support the writing of providers (which are specific to each platform), Terraform offers "Terraform-provider-scaffolding," which is a code repository that provides a template defining how a provider should be written. This template contains the following:

- A resource and a data source (`internal/provider/`)
- Examples (`examples/`) and generated documentation (`docs/`)
- Miscellaneous meta files

This template provided by HashiCorp contains boilerplate code that you will need to edit to create your own Terraform provider. Once the Terraform provider is written, the platform vendor needs to publish in the Terraform registry maintained by HashiCorp so that the provider is available to a wide audience.

Since Terraform is a plugin-based architecture, these Terraform plugins enable all developers to extend Terraform usage by writing new plugins or compiling modified versions of existing plugins to build a new one if required. As depicted in Figure 4-1, there are two main components of the Terraform architecture: the Terraform core and Terraform plugins. The Terraform core interacts with plugins over remote procedure calls (RPCs) and offers multiple ways to discover and load plugins for further interaction with the destination platform. It's the Terraform plugins that enable Terraform to expose an implementation for a specific service such as support on Azure, VMware, GCP, etc.

Terraform Core

The Terraform core is open-source code and hosted at (https://github.com/hashicorp/Terraform). It is a consistently compiled list of binary files written in the Go programming language. These compiled binary files are called the Terraform CLI. The CLI uses RPCs to communicate with the Terraform plugins and offers multiple ways to discover and load plugins for use. The following are the key functions of the Terraform core:

- It simulates infrastructure as a code (IaC). It enables the reading and interpolation of configuration files and Terraform modules.
- It manages the state of Terraform managed resources.
- It builds a dependency graph from the Terraform configuration and walks this graph to generate Terraform plans, refresh state, and more.
- It executes the Terraform plan.
- It communicates with plugins via RPCs.

As shown in Figure 4-2, a few reference key commands where the Terraform core interacts with the providers over RPCs include `Diff()`, `Apply()`, `Refresh()`, etc.

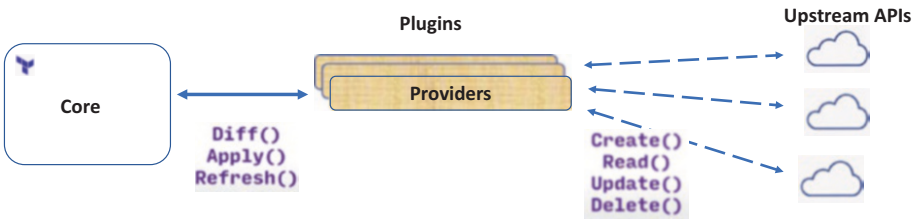


Figure 4-2. *Terraform core and plugin interaction*

Terraform Plugins

Terraform plugins contain key components known as *providers* and *provisioners*. Providers are code written by developers in the Go programming language. These are executable binaries that are invoked by the Terraform core using the RPCs. We discussed Terraform provisioners in Chapter 2; they allow the remote execution of custom code directly on the supported platform. Each plugin exposes an implementation of a specific service such as Azure, GCP, VMware, etc.

End users define the respective provider and provisioners in the configuration file they create, which further enables the interaction via the RPCs of the Terraform core to the destination platform. Terraform has many built-in provisioners, while providers are added dynamically as and when support is added by the respective platform developers. This is the beauty of the Terraform core SDK; it provides a high-level framework that abstracts away the details of plugin discovery and RPC communication.

The following are the key responsibilities of provider plugins:

- Initialization of any included client specific libraries that are used to make API calls to the Upstream API platform
- Authentication with the supported infrastructure platform
- The definition of managed resources that map to specific services

In Figure 4-2, we can see that a few of the important functions that enable interaction of providers with the upstream APIs are `create()`, `Read()`, `Delete()`, and `Update()`.

The following is the key responsibility of the provisioners present in the plugins: executing commands or scripts on the designated resource following creation or destruction.

When a user runs the `terraform init` command, the Terraform core looks for the plugins in the directories listed in Table 4-1. Please note that some of the directories' paths are static, while some are relative to the current working directory.

Table 4-1. *Terraform and Default Directories*

Directory	Uses
Location of Terraform binary (e.g., `/usr/local/bin`)	Typical Terraform installations
`.terraform/plugins/<OS>_<ARCH>`	Automatically downloaded providers
`~/.terraform.d/plugins` or `%APPDATA%%\terraform.d\plugins`	The user plugins directory

Note OS and ARCH here use the Go language standard OS and architecture names, for example, `ubuntu_amd64`.

Third-party plugins should usually be installed in the user plugins directory, which is located at `~/.terraform.d/plugins` on most operating systems and at `%APPDATA%\terraform.d\plugins` on Windows operating systems.

If you are running `terraform init` with the `-plugin-dir=<PATH>` option (with a nonempty `<PATH>`), this will override the default plugin locations and search only the path that you had specified.

After finding any installed plugins, `terraform init` compares them to the configuration's version constraints and chooses a version for each plugin as defined here:

- If there are any acceptable versions of the plugin that have already been installed, Terraform uses the newest installed version that meets the constraint (even if releases.hashicorp.com has a newer acceptable version).
- If no acceptable versions of plugins have been installed and the plugin is one of the providers distributed by HashiCorp, Terraform downloads the newest acceptable version from releases.hashicorp.com and saves it in .terraform/plugins/<OS>_<ARCH>.

This step is skipped if terraform init is run with the -plugin-dir=<PATH> or -get-plugins=false option.

We just explained the semantics of how Terraform will attempt to download the plugins, because without plugins, you will not be able to use Terraform on your respective platform.

Client Library

Most platforms these days support a set of API libraries that allows programmatic interaction with the platform. Developers who write a specific provider leverage these client (platform) libraries for automated interaction and enable the Terraform core to leverage these libraries via the providers. These client libraries interact with the platform on the HTTP(s) protocol. This means that for any platform that has API-based management capabilities, developers of that platform can offer Terraform integration as well.

The responsibility of managing client libraries lies with the developers who wrote the provider support for a particular platform. Every time there is a revision in the client libraries, developers from the platform can revise the integration and offer a new provider version that may offer new features added in the client libraries.

We covered the Terraform architecture in this section and learned about the core components including the Terraform core, Terraform plugins, and client libraries. In the next section, we will look at how we can build the next phase of reverse engineering, called modeling.

Modeling

As you now know, Terraform is dependent on client libraries to interact with the destination platform. What if we in our reverse-engineering model use the same libraries to interact with the platform to create the configuration file for our import operations? We can essentially use the same "source of truth" as those client libraries that the Terraform core is leveraging and enable interactions with the given platform. In this section, we will discuss how we can use the source of truth and do modeling.

Sample Model

Modeling in reverse engineering refers to the process of depicting the information extracted in the first step into a general model that can be used to design a new system.

Figure 4-3 shows the system we presented in Chapter 3.

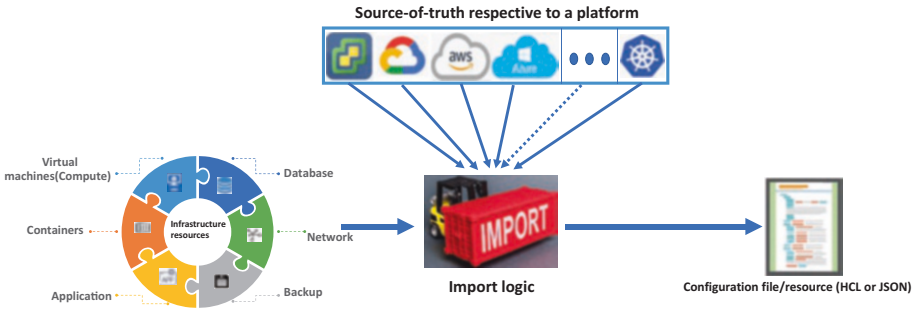


Figure 4-3. *Source of truth and Terraform import*

This represents the modeling aspect of reverse engineering with Terraform. When writing our import logic, if we can leverage the same "client library" we referred to in the previous section that is respective to a destination platform, then we can generate the point-in-time configuration file required for importing an existing resource into Terraform.

The import logic can be written in any language where there is support provided with the client library. Many vendor platforms offer their supported libraries in Python, Go, Ruby, etc. The idea is to leverage the capabilities of those libraries and generate the consistent configuration file that's needed for Terraform import.

Object Identification

Another important aspect when modeling the reverse engineering with Terraform is the definition of objects in the configuration file, meaning the list of objects that you like to fetch from the source of truth with help of those client libraries. As we described in earlier chapters, a configuration file is a user-managed file (in HCL or JSON) where users can define the infrastructure as code and do modifications to change the desired state of the resource.

Terraform experts can easily identify the common object parameters (mandatory and optional) that represent a large part of their infrastructure deployment. The import logic can be accordingly coded to fetch the values of these object parameters and can be written into a configuration file in the desired format needed for Terraform import (HCL or JSON). Every provider in Terraform offers the list of mandatory and optional objects that are to be defined in the configuration file that the user generates. The documentation published by Terraform corresponding to a provider usually has a list of objects in different sections such as "data," "resource," etc.

You need to understand these objects and collect a sample list that usually is applicable to most of the infrastructure resources you want to manage. This should be a one-time exercise if you are looking to import and manage your infrastructure with Terraform. It is not a problem if you are collecting extra sets of object variables from the source of truth; usually the more the better because it allows for a successful import and less chance that Terraform is not imposing default values if that object parameter is not specified in the configuration file.

Review

It is this stage of reverse engineering with Terraform where the resultant model is defined, the configuration file objects are identified, and an import script is written that can represent the configuration file parameters needed for a successful import of any existing resource into Terraform. This is an important step of the reverse-engineering process because it ensures the whole exercise is fruitful. The following are the steps in this phase of reverse engineering with Terraform:

1. Perform comprehensive testing of a sample import of the varied resources you want to manage with Terraform and benefit from infrastructure automation.

2. With every import that is automated, test using `terraform plan` and validate that the Terraform core is not suggesting any changes on the infrastructure. If Terraform suggests any changes, then that is not a successful import. Either there are missing object parameters that need to be coded and fetched in your configuration file or there is a possibility that after the automated configuration

file was generated that there were changes made on the resource directly on the destination platform. In that case, you would need to discard the existing file and generate a new configuration file automatically using your import logic.

3. If your reverse-engineering process is part of another, bigger workflow, then you would want to verify the functioning of your defined process in a bigger context and fix any issues per the desired outcome.

The review process with reverse engineering can be integrated into the bigger workflow, and you can do complete end-to-end automation on the infrastructure resource. Figure 4-4 shows that there is an Azure DevOps (ADO) pipeline triggering the Terraform import and running the Terraform plan. If there are no changes suggested when running the plan, it is a successful import, and we can proceed and leverage further the infrastructure automation capabilities provided by Terraform.

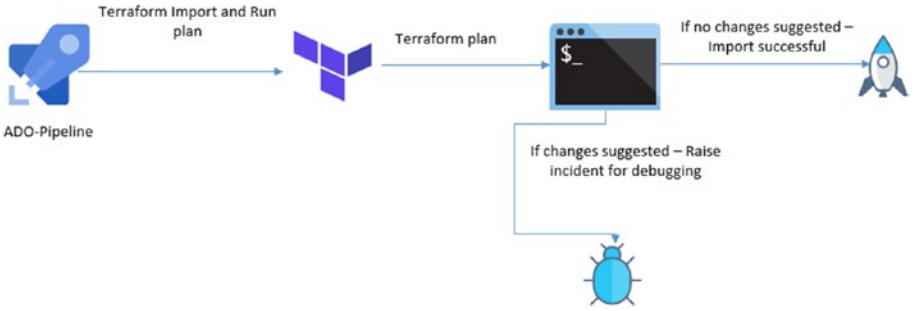


Figure 4-4. *Terraform sample workflow with ADO and incident management*

However, if there are changes suggested, you can create a workflow to create a ticket/incident for the Terraform experts to look at and fix the import logic to incorporate if anything that was newly added or is missing in the configuration file.

Understand a Sample Reverse-Engineering Model with Terraform

So far, we have learned about the reverse engineering concepts and how they help in fulfilling our purpose. The knowledge we want to build in this section can be easily augmented and implied in a scenario where we are looking for the automated management of our infrastructure resources with Terraform. The skills we want to build are in a space where organizations have dependency on a tool (Terraform) to manage their automation across diverse infrastructure platforms. The sample case study uses VMware to do an automated import of a large number of VMs already present on a VMware farm.

In Chapter 3, we covered in brief the concept of the managed object browser (MOB) in VMware. That is a hook (source of truth) for doing reverse engineering in VMware. Similarly, with other platforms, there are similar hooks that enable doing reverse engineering on respective platforms. Over the last few months, native platform providers such as Microsoft Azure and GCP have been coming up with native command-line utilities to allow the creation of a Terraform configuration file for the easy import of existing resources. We are going to touch on those in Chapter 7. Regardless, the knowledge we are building through this book allows IT administrators to build their own reverse-engineered system that allows automated management via Terraform.

Now, to further our understanding of our sample use case that consists of an automated import of a VMware VM, let's deep dive into the concepts further.

The import logic we want to write should be such that it suits the majority of the workloads in our environment and should be easy to invoke and give us the desired results. Figure 4-5 shows the sample model of the VMware import.

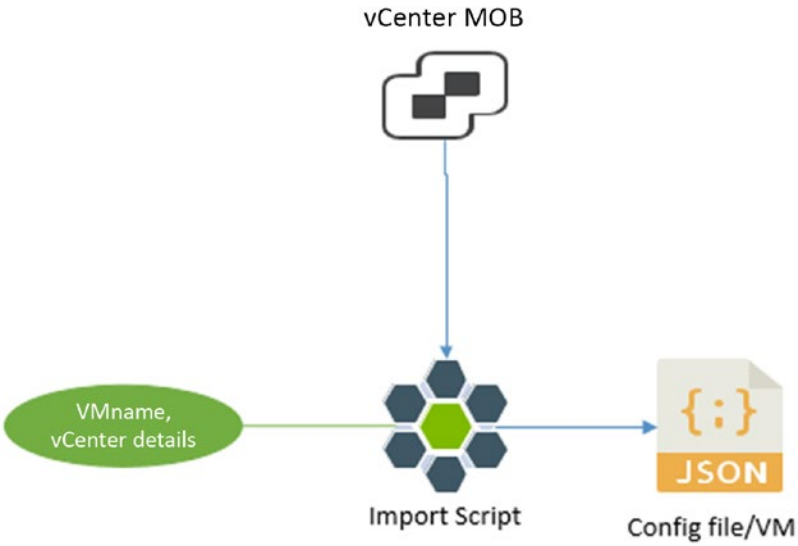


Figure 4-5. *Sample model for VMware VM import*

In this model, we are presenting just the VM name and vCenter details to a Python script called the *import script*. This script is at the heart of this model and plays a key role. This script can be written in any language and fetches the desired parameter values from the vCenter MOB. The script has prebuilt logic where it collects data automatically and in a desired format via REST calls through the vCenter object browser. The data collected is a point-in-time and accurate representation of the VM objects present in the vCenter database. The output of this script can be a file in the HashiCorp Command Language (HCL) or in JSON format; this file acts as a config file for further imports you want to do in Terraform.

The import script is considered small-logic software like any other script whose job is to fetch details from one platform and populate information in files as an output. Like any other automation logic, this import script has dependencies too. The configuration parameters for which we are fetching values from the MOB and are defined in the import script are in turn dependent on the following.

Terraform Provider Version

The Terraform provider provides a medium for the Terraform core to interact with the desired infrastructure platform. The Terraform community continues to enhance their providers, which results in certain parameter values or names being added, modified, or removed from the list of parameters that Terraform manages for the environment. As the platform providers enhance their providers, the list of objects we need to fetch to generate configuration files evolves too. For a consistent reverse-engineering experience, it is advised to keep the provider version constant. In the case of updates needed to the provider, you should follow the "review" phase of reverse engineering to ensure your import logic is still consistent with the revised provider version.

Infrastructure Platform Revisions

The source of truth is the entity that is the hook for getting the true information of the objects in that platform. In our case, the MOB is the source of truth for VMware. As with any other platform, VMware developers consistently revise/renew the objects they manage in their MOB inventory. Whenever we upgrade the vCenter version in our landscape, it potentially may affect the object and values that it stores in its database. In the case of an update with the platform (vCenter), you should follow the "review" phase steps of reverse engineering to ensure your import logic is still consistent with the revised platform upgrades.

The dependencies mentioned for the import script, i.e., the Terraform provider version and source of truth version, are key, and they should be constant when we are operating our reverse-engineering solution with our production workload. Every time we are doing an upgrade of a Terraform provider or platform, the import logic should be evaluated in a test environment first with a diverse set of workloads and configuration. Once validated, the code is then good to use for production environments.

There can be several use cases around the import script to facilitate the usage of config files for importing into Terraform. These are some sample use cases:

- The output config file from this import script can be used for manual consumption by the IT administrators. They can in turn do this to allow and enable the DevOps way of handling an application via Terraform.
- The output config file from the import script can be submitted as a check-in to a DevOps or GitHub repository for direct end-user consumption. Of course, the DevOps repository can be linked with Terraform, and other operations can be handled via that same source repository for other VM management workflows.
- Another sample use case could be feeding the output of the import script to a DevOps pipeline directly where it is consumed for further automated import of the desired resources and for allowing automated configuration management via a DevOps system.

In short, the logic we want to write can be integrated with numerous schemes that allow for the automated management of our infrastructure platform.

Automated Creation of a Point-in-Time Config File

Now that we understand the model that allows for reverse engineering of a resource into Terraform, let's take a look at what a typical configuration file looks like for a VMware VM, which has necessary and optional parameters required for import.

There are multiple sections of this configuration file required for VMware workloads. Let's discuss them in brief again.

Provider

As the name suggests, the provider is the section of the code block that specifies the infrastructure you are going to manage your resources. In our example of VMware, you can see the provider is `vSphere`.

Since Terraform continuously brings in revisions to the code that enables Terraform interaction with the infrastructure, it is mandatory to use the correct version of the provider. In the following example, the `vSphere` provider version is `2.3.1`.

The following is a sample code snippet in JSON for reference:

```
{
    "terraform": {
        "required_providers": {
            "vsphere": {
                "source": "local/hashicorp/vsphere",
                "version": "2.3.1"
            }
        }
    },
```

The statement `"source": "local/hashicorp/vsphere"` implies a local copy of the provider version (2.3.1 in our case) that is already installed on our local machine from where we are operating the Terraform core.

Provider Details to Connect to the Platform

In the config file, details of the infrastructure platform are required so Terraform can make connections and manage the desired resource on the specified platform. There are different parameters that can be defined specific to the provider. Each provider needs to authenticate with your specific platform.

The following is a sample code snippet in JSON format for reference:

```
"provider": {
    "vsphere": {
        "vsphere_server": "xx.xx.xx.xx",
        "user": "user@domain",
        "password": "xxxx",
        "allow_unverified_ssl": "true"
    }
},
```

Data Section

The `data` section of the code block defines the external additional resources on which our core resource would be built. For example, to create or modify a VM, this data code block would need the datastore details, VM network details, etc. The code block in the data section is referenced with parameters defined in the `resource` section.

The following is a sample code snippet in JSON format for reference:

```
"data": {
```

```
        "vsphere_datacenter": {
            "dc": {
                "name": "XXX-YourDatacentername-XXX"
            }
        },
        "vsphere_resource_pool": {
            "pool": {
                "name": "<Cluster>/Resources",
                "datacenter_id": "${data.vsphere_
                                  datacenter.dc.id}"
            }
        },
        "vsphere_datastore": {
            "datastore": {
                "name": "<Datastore name where VM resides>",
                "datacenter_id": "${data.vsphere_
                                  datacenter.dc.id}"
            }
        },
        "vsphere_network": {
            "network0": {
                "name": "VM Network",
                "datacenter_id": "${data.vsphere_
                                  datacenter.dc.id}"
            }
        }
    },
```

Resource Section

The `resource` section of code is the main section that defines the resource that we need to manage with Terraform. It has several mandatory and optional parameters that need to be defined. For example, to manage a VM, the resource code block consists of essential parameters such as the VM name, datastore details, CPU, memory details, network and storage details etc.

The following is a sample code snippet in JSON format for reference:

```
"resource": {
    "vsphere_virtual_machine": {
        "VMresource": {
            "network_interface": {
                "adapter_type": "vmxnet3",
                "network_id": "${data.vsphere_network.
                              network0.id}"
            },
            "name": "UbantuTest",
            "resource_pool_id": "${data.vsphere_resource_
                                 pool.pool.id}",
            "datastore_id": "${data.vsphere_datastore.
                            datastore.id}",
            "boot_retry_enabled": false,
            "enable_disk_uuid": false,
            "enable_logging": false,
            "num_cores_per_socket": 1,
            "num_cpus": 3,
            "guest_id": "ubuntu64Guest",
            "memory": 12288,
            "cpu_hot_add_enabled": "true",
            "memory_hot_add_enabled": "true",
```

```
                "firmware": "bios",
                "scsi_type": "${data.vsphere_virtual_machine.
                             template.scsi_type}",
                "lifecycle": {
                    "ignore_changes": [
                        "custom_attributes",
                        "tags"
                    ]
                },
                "disk": {
                    "label": "disk0",
                    "size": 16,
                    "unit_number": 0,
                    "thin_provisioned": "false",
                    "path": "[Datastore name] UbantuTest/
                            UbantuTest.vmdk",
                    "keep_on_remove": "true"
                }
            }
        }
    }
}
```

Note that there are several other sections in a typical configuration file such as input variables, output variables, local variables, etc. But these are not explained here because our focus is mainly on the configuration file, which is essential for automated imports. You can combine all these sections in one configuration file (called `main.tf`). This represents a sample config file needed for a typical import of a VM ware VM. This config file can be augmented to match the needs of IT administrators.

In the hands-on exercise with this chapter, we are going to provide a Python script (called an *import script*) that can be employed to generate this same config file with most of the required parameters from VMware vCenter.

Importing of a Resource with Terraform

Terraform can import existing infrastructure. This allows users to take resources that are created outside of Terraform and bring them under Terraform management. Importing a resource also means that Terraform is made aware of the current state of the resource. Terraform provides a single command that helps in importing a resource. However, as described earlier, the clean importing of a resource needs a well-defined and populated configuration file. Let's first understand the importing logic; see Figure 4-6.

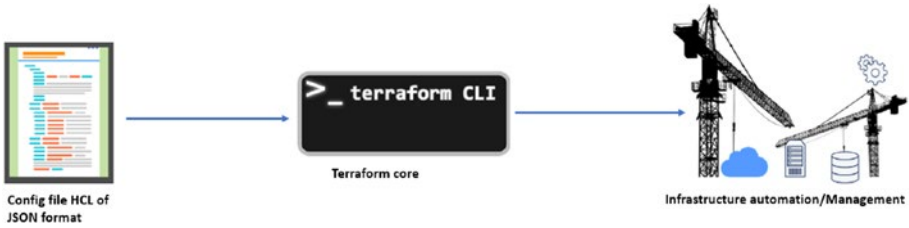


Figure 4-6. *Terraform import*

From Figure 4-6, you can see we need to have a config file with the required parameters. This configuration file can be in JSON or HCL format. This configuration file then needs to be presented to the Terraform core, which helps to import the resource, and it then generates the Terraform state file.

To explain the import operation in a more practical way, we will use a lab virtual machine running Ubuntu.

For a successful import, we need to place the fully populated configuration file as explained earlier in this chapter in the desired location. The following shows the config file named `main.tf.json` present in a folder named `import`.

```
user@ubantu:~/import$ ls -ll
total 4
-rw-rw-r-- 1 user user 2829 Feb 21 00:54 main.tf.json
user@ubantu:~/import$
```

Configuration file

The sample VM running in the lab setup is `UbantuTest`, which is to be imported with Terraform, as shown in Figure 4-7.

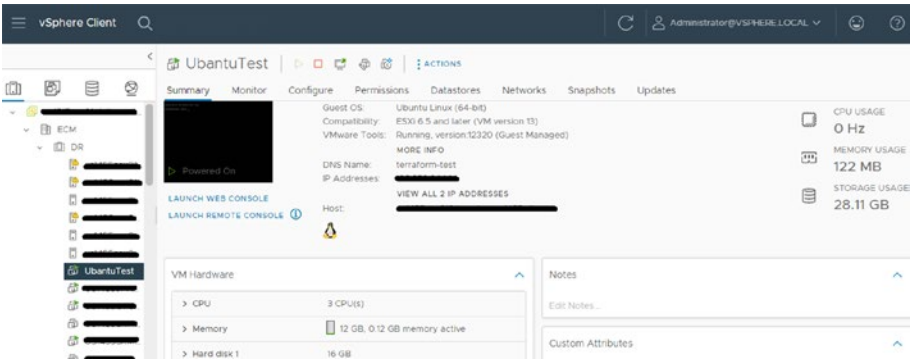


Figure 4-7. *Pre-existing VM called UbantuTest in vCenter*

The sample command for a Terraform import of a VMware VM is shown here:

```
Terraform import vsphere_virtual_machine.<ConfigFileResource
Name>  /<Cluster>/vm/<VMName>
```

If the configuration file is properly populated, it would result in a successful import. Here's an example of the output from a successful import:

- `VMresource` is the name of a resource as defined in the configuration file (`main.tf.json`).
- `ECM` is the name of the sample datacenter in vCenter.
- `UbantuTest` is the VM name running in the datacenter that is imported.

```
user@ubantu:~/import$ terraform import vsphere_virtual_machine.VMresource /ECM/vm/UbantuTest
data.vsphere_datacenter.dc: Reading...
data.vsphere_datacenter.dc: Read complete after 0s [id=datacenter-1001]
data.vsphere_network.network0: Reading...
data.vsphere_resource_pool.pool: Reading...
data.vsphere_datastore.datastore: Reading...
data.vsphere_virtual_machine.template: Reading...
data.vsphere_datastore.datastore: Read complete after 0s [id=datastore-1029]
data.vsphere_network.network0: Read complete after 0s [id=network-1035]
data.vsphere_resource_pool.pool: Read complete after 0s [id=resgroup-1007]
data.vsphere_virtual_machine.template: Read complete after 0s [id=42150859-0f1d-32dc-504f-78ec6097cdfe]
vsphere_virtual_machine.VMresource: Importing from ID "/ECM/vm/UbantuTest"...
vsphere_virtual_machine.VMresource: Import prepared!
  Prepared vsphere_virtual_machine for import
vsphere_virtual_machine.VMresource: Refreshing state... [id=42150859-0f1d-32dc-504f-78ec6097cdfe]

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.
```

After a successful import into Terraform, the following is the additional file that gets created. This is the state file that Terraform maintains for its records. This is the file against which our configuration file is differentiated when there are changes to be applied on the destination platform.

```
user@ubantu:~/import$ ls -ll
total 16
-rw-rw-r-- 1 user user  2830 Feb 21 01:09 main.tf.json
-rw-rw-r-- 1 user user 10889 Feb 21 01:09 terraform.tfstate
user@ubantu:~/import$
```

After a successful import, we should initialize Terraform with the `terraform init` command.

```
user@ubantu:~/import$ terraform init

Initializing the backend...

Initializing provider plugins...
- Reusing previous version of local/hashicorp/vsphere from the dependency lock file
- Using previously-installed local/hashicorp/vsphere v2.3.1

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
user@ubantu:~/import$
```

Validating a Successful Import

After a successful import, it is important to validate if it is correct. We define a successful import when we run `terraform plan` and it suggests no modification to our infrastructure. If `terraform plan` suggests modifications, there are certain parameters missing in the config file whose values are not fetched and Terraform has assumed the default value in the configuration file. If there are changes suggested by Terraform that are applied on the platform, then it would make changes to the resource, which may not be desired. Therefore, a successful import relies on a clean config file, which reflects the accurate, point-in-time state of a resource. To get to the stage of a clean config file, IT administrators might have to do a few iterations when building logic or fetching the values of the desired parameters. The config file generated should be tested for a variety of resources and complex configurations.

Now let's run `terraform plan` for the sample VMware import we just performed.

```
user@ubantu:~/import$ terraform plan
data.vsphere_datacenter.dc: Reading...
data.vsphere_datacenter.dc: Read complete after 0s [id=datacenter-1001]
data.vsphere_network.network0: Reading...
data.vsphere_datastore.datastore: Reading...
data.vsphere_virtual_machine.template: Reading...
data.vsphere_resource_pool.pool: Reading...
data.vsphere_datastore.datastore: Read complete after 0s [id=datastore-1029]
data.vsphere_network.network0: Read complete after 0s [id=network-1035]
data.vsphere_resource_pool.pool: Read complete after 0s [id=resgroup-1007]
data.vsphere_virtual_machine.template: Read complete after 0s [id=42150859-0f1d-32dc-504f-78ec6097cdfe]
vsphere_virtual_machine.VMresource: Refreshing state... [id=42150859-0f1d-32dc-504f-78ec6097cdfe]

No changes. Your infrastructure matches the configuration.

Terraform has compared your real infrastructure against your configuration and found no differences, so no changes are needed.
user@ubantu:~/import$
```

As you can see, this import is clean because `terraform plan` suggested no changes to the infrastructure. With our development of `import script`, we were able to identify and define the required parameters that suit most of the VM workloads deployed in our environment. Any IT administrator should be able to use the same reverse-engineering concepts to define a config file and create or modify the "custom import" script to fetch the required parameters for a successful import that matches the VM deployments in their own environment.

Hands-On Exercise: Import Script to Demonstrate Successful Autogeneration of a Config File

In this hands-on exercise, we are going to demonstrate the reverse-engineering concepts we've discussed so far and do an automated import of a sample VM in a VMware environment. This import script is fetching details from the MOB and creating a config file, which allows for a successful import of a VMware virtual machine. The most updated import script written for an import of the VMware virtual machine can be accessed from the GitHub repository we maintain, available here:

https://github.com/sumitbhatia1986/Terraform-VMware-ReverseEngineering

We are replicating the sample code here to show you how a typical import script can be written in the Python language that interacts with the MOB and autogenerates a typical configuration file.

Please note that though this exercise demonstrates a VMware import, this import configuration suits most of the workloads in our environment. There may be certain additional parameters you need to fetch based on the VM workloads running in your specific environment. Also note that the same reverse-engineering logic can be easily used on other platforms where you want automation via Terraform.

```
# -*- coding: utf-8 -*-
"""
Created on Mon Dec  7 17:42:16 2022

@author: Sbhatia3
"""
import json
import requests
```

```
import sys
from requests.packages.urllib3.exceptions import
InsecureRequestWarning
from pyVim import connect  # client libraries to interact
with MOB
import ssl
from pyVmomi import vim # client libraries to interact with MOB
import re
import collections

#Defining class for structural collection of Config file
parameters.
class OrderedConfig(collections.OrderedDict):
    pass
app_settings = {
    'api_pass': "XXXX",
    'api_user': "administrator@vsphere.local",
    'api_url': "https://<vCenter FQDN>/rest/",
    'vcenter_ip': "xx.xx.xx.xx",
    'VM_name': "UbantuTest", ### Desired VM for which we need
               to find details <Case sensitive>
    'vsphere_datacenter': "ECM"
}

## Authenticate with Vcenter to find the VM ID
def auth_vcenter(username, password):
    resp = requests.post(
        '{}/com/vmware/cis/session'.format(app_
        settings['api_url']),
        auth=(app_settings['api_user'], app_settings['api_
        pass']),
        verify=False
```

```
    )
    if resp.status_code != 200:
        print('Error! API responded with: {}'.format(resp.
                                            status_code))
        return
    return resp.json()['value']

def get_api_data(req_url):
    sid = auth_vcenter(app_settings['api_user'],
          app_settings['api_pass'])
    resp = requests.get(req_url, verify=False,
           headers={'vmware-api-session-id': sid})
    if resp.status_code != 200:
        print('Error! API responded with: {}'.format(resp.
                                            status_code))
        return
    return resp

## Getting VM details including VM ID needed to browser
through MOB
def get_vm(vm_name):
    resp = get_api_data('{}/vcenter/vm?filter.names={}'.
           format(app_settings['api_url'], vm_name))
    j = resp.json()
    return (j)
```

#Fetching VM ID from REST API for the VM we want to import

```
requests.packages.urllib3.disable_warnings(InsecureRequest
Warning)
vmdetails = get_vm(app_settings['VM_name'])
vmid = vmdetails['value'][0]['vm']

s = ssl._create_unverified_context()
```

```
service_instance = connect.SmartConnect(
    host=app_settings['vcenter_ip'], user=app_settings
    ['api_user'], pwd=app_settings['api_pass'], sslContext=s
)
content = service_instance.RetrieveContent()
container = content.rootFolder  # starting point to look into
viewType = [vim.VirtualMachine]  # object types to look for
recursive = True  # whether we should look into it recursively
containerView = content.viewManager.CreateContainerView
                (container, viewType, recursive)  # create
                container view
children = containerView.view

for child in children:  # for each statement to iterate all
names of VMs in the environment
    if (str(vmid) in str(child)):
        vm_summary = child.summary  #Summary of the desired VM
                     to import
        vm_config = child.config #Complete config data
                    hiararchy and child item values loaded in
                    the variable
        vm_resourcepool = child.resourcePool #Resource
                          pool details
        vm_network = child.network #Network details of the VM
        vm_datastore = child.datastore
        vm_parent = child.parent
```

Config file structure and populating with corresponding values fetched with MOB

```
data = OrderedConfig()
data["provider"] = OrderedConfig()
data["provider"]["vsphere"] = OrderedConfig()
```

```
data["provider"]["vsphere"]["user"] = "sampleuser"
data["provider"]["vsphere"]["password"] = "samplepassword"
data["provider"]["vsphere"]["allow_unverified_ssl"] = "true"
```

#Posting resource pool details

```
resourcepool_string = str(vm_resourcepool.owner.name) + '/' +
str(vm_resourcepool.name)
data["data"] = OrderedConfig()

data["data"]["vsphere_datacenter"] = OrderedConfig()
data["data"]["vsphere_datacenter"]["dc"] = OrderedConfig()
data["data"]["vsphere_datacenter"]["dc"]["name"] = app_
settings['vsphere_datacenter']

data["data"]["vsphere_resource_pool"] = OrderedConfig()
data["data"]["vsphere_resource_pool"]["pool"] = OrderedConfig()
data["data"]["vsphere_resource_pool"]["pool"]["name"] =
resourcepool_string
data["data"]["vsphere_resource_pool"]["pool"]["datacenter_id"]
= "${data.vsphere_datacenter.dc.id}"
```

#Posting datastore details

```
datastore_string = str(vm_datastore[0].name)
data["data"]["vsphere_datastore"] = OrderedConfig()
data["data"]["vsphere_datastore"]["datastore"] =
OrderedConfig()
data["data"]["vsphere_datastore"]["datastore"]["name"] =
datastore_string
data["data"]["vsphere_datastore"]["datastore"]["datacenter_id"]
= "${data.vsphere_datacenter.dc.id}"
```

```
    #Posting virtual machine details

datastore_string = str(vm_datastore[0].name)
data["data"]["vsphere_virtual_machine"] = OrderedConfig()
data["data"]["vsphere_virtual_machine"]["template"] =
OrderedConfig()
data["data"]["vsphere_virtual_machine"]["template"]["name"] =
str(vm_config.name)
data["data"]["vsphere_virtual_machine"]["template"]
["datacenter_id"] = "${data.vsphere_datacenter.dc.id}"

    #For all network adapter, creating a stack, adapter details in order

network_adapter = []
for item_netadapter in vm_config.hardware.device:

    if (str("Network adapter") in str(item_netadapter.
    deviceInfo.label)):
        vm_networkadapter = item_netadapter

        #Posting adapter type for Network interface details
        if (str("E1000e") in str(type(vm_networkadapter))):
            network_adapter.append("e1000e")

        if (str("Vmxnet3") in str(type(vm_networkadapter))):
            network_adapter.append("vmxnet3")

data["resource"] = OrderedConfig()
data["resource"]["vsphere_virtual_machine"] = OrderedConfig()
data["resource"]["vsphere_virtual_machine"]["jsontemplate"] =
OrderedConfig()

    #Posting network details and adding network interface details at
same order

for index, nic in enumerate(vm_network):
```

```
  network_string = str(nic.name)
  data["data"]["vsphere_network"+str(index)] = OrderedConfig()
  data["data"]["vsphere_network"+str(index)]
  ["network"+str(index)] = OrderedConfig()
  data["data"]["vsphere_network"+str(index)]
  ["network"+str(index)]["name"] = network_string
  data["data"]["vsphere_network"+str(index)]
  ["network"+str(index)]["datacenter_id"] = "${data.vsphere_
  datacenter.dc.id}"

for index, nic in reversed(list(enumerate(vm_network))):
  #adding network interface details
  data["resource"]["vsphere_virtual_machine"]["jsontemplate"]
  ["network_interface"+str(index)] = { "adapter_type" :
  network_adapter.pop() }
  data["resource"]["vsphere_virtual_machine"]["jsontemplate"]
  ["network_interface"+str(index)]["network_id"] = str("${data.
  vsphere_network."+"network"+str(index)+".id}")

####################Posting other VM generic details

    #Posting Name of the Virtual machine

data["resource"]["vsphere_virtual_machine"]["jsontemplate"]
["name"] = str(vm_config.name)
data["resource"]["vsphere_virtual_machine"]["jsontemplate"]
["folder"] = vm_parent.name
data["resource"]["vsphere_virtual_machine"]["jsontemplate"]
["resource_pool_id"
                                                          ] =
"${data.vsphere_resource_pool.pool.id}"
data["resource"]["vsphere_virtual_machine"]["jsontemplate"]
["datastore_id"] = "${data.vsphere_datastore.datastore.id}"
```

```
data["resource"]["vsphere_virtual_machine"]["jsontemplate"]
["boot_retry_enabled"

] = vm_config.bootOptions.bootRetryEnabled
data["resource"]["vsphere_virtual_machine"]["jsontemplate"]
["enable_disk_uuid"] = vm_config.flags.diskUuidEnabled
data["resource"]["vsphere_virtual_machine"]["jsontemplate"]
["enable_logging"] = vm_config.flags.enableLogging
data["resource"]["vsphere_virtual_machine"]["jsontemplate"]
["num_cores_per_socket"

] = vm_config.hardware.numCoresPerSocket

    #Posting Number of CPU's

data["resource"]["vsphere_virtual_machine"]["jsontemplate"]
["num_cpus"] = vm_config.hardware.numCPU
data["resource"]["vsphere_virtual_machine"]["jsontemplate"]
["guest_id"] = str(vm_config.guestId)

    #Posting Memory

data["resource"]["vsphere_virtual_machine"]["jsontemplate"]
["memory"] = vm_config.hardware.memoryMB

    #Posting guestid

data["resource"]["vsphere_virtual_machine"]["jsontemplate"]
["guest_id"] = str(vm_config.guestId)

    #Posting CPU Hot add enabled flag info

data["resource"]["vsphere_virtual_machine"]
["jsontemplate"]["cpu_hot_add_enabled"] = str(vm_config.
cpuHotAddEnable).lower()
```

```
    #Posting Memory hot add enabled flag info

data["resource"]["vsphere_virtual_machine"]["jsontemplate"]
["memory_hot_add_enabled"] = str(vm_config.
memoryHotAddEnabled).lower()

    #Posting firmware information

data["resource"]["vsphere_virtual_machine"]["jsontemplate"]
["firmware"] = str(vm_config.firmware).lower()

    #Posting scsi type information

data["resource"]["vsphere_virtual_machine"]["jsontemplate"]
["scsi_type"] = "${data.vsphere_virtual_machine.template.
scsi_type}"

    #Posting ignore tags and custom attribute changes

data["resource"]["vsphere_virtual_machine"]["jsontemplate"]
["lifecycle"] = { "ignore_changes": ["custom_attributes",
"tags"]}
disk_starting_name = "disk"
no_of_disk = 0

    #Posting disk information

for item_virtualdisk in vm_config.hardware.device:
    if (str("VirtualDisk") in str(type(item_virtualdisk))):
        diskname = disk_starting_name + str(no_of_disk)
        data["resource"]["vsphere_virtual_machine"]
        ["jsontemplate"][diskname] = OrderedConfig()
        data["resource"]["vsphere_virtual_machine"]
        ["jsontemplate"][diskname]["label"] = "disk" + str(
            item_virtualdisk.unitNumber
        )
```

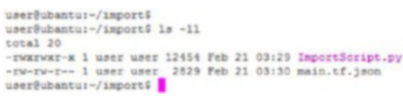
```
        data["resource"]["vsphere_virtual_machine"]
        ["jsontemplate"][diskname]["size"] = int(
            item_virtualdisk.capacityInKB / 1048576
        )
        data["resource"]["vsphere_virtual_machine"]
        ["jsontemplate"][diskname]["unit_number"] = item_
        virtualdisk.unitNumber
        data["resource"]["vsphere_virtual_machine"]
        ["jsontemplate"][diskname]["thin_provisioned"] = str(
            item_virtualdisk.backing.thinProvisioned
        ).lower()
        data["resource"]["vsphere_virtual_machine"]
        ["jsontemplate"][diskname]["path"] = str(
            item_virtualdisk.backing.fileName
        )
        data["resource"]["vsphere_virtual_machine"]
        ["jsontemplate"][diskname]["keep_on_remove"] = "true"
        no_of_disk += 1

for key in data:
    if (key == 'resource'):
        for t in data[key]:
            if (t == 'vsphere_virtual_machine'):
                data[key][t][str(vm_config.name)] = data[key]
                [t].pop('jsontemplate')

#getting json string (Order changing issue)
json_string = json.dumps(data, indent = 4)

#replacing all different diskname(eg: disk0) to "disk" itself
for i in range(no_of_disk):
    replace_string = '"disk'+ str(i) +'": {'
```

```
    json_string = re.sub(replace_string, '"disk": {',
    json_string)

#replacing all different diskname(eg: "vsphere_network0") to
"vsphere_network" itself
for i in range(len(vm_network)):
    replace_string = '"vsphere_network'+ str(i) +'": {'
    json_string = re.sub(replace_string, '"vsphere_network":
                  {', json_string)

#replacing all different diskname(eg: "network_interface0") to
"network_interface" itself
for i in range(len(vm_network)):
    replace_string = '"network_interface'+ str(i) +'": {'
    json_string = re.sub(replace_string, '"network_interface":
    {', json_string)

sys.stdout.write(json_string)

#output config JSON file
with open("main.tf.json", "w") as outfile:
    outfile.write(json_string)
```

The different sections of this script are self-explanatory and are built upon basic Python fundamentals, which are written to fetch a sample config file. For ease of understanding, the following is the sample execution and generation of the configuration file with this import script:

```
user@ubantu:~/import$ ls -ll
total 16
-rwxrwxr-x 1 user user 12454 Feb 21 03:29 ImportScript.py
user@ubantu:~/import$
user@ubantu:~/import$
user@ubantu:~/import$ python3 ImportScript.py
```

Executing the Python script fetches us the autogenerated configuration file in JSON format. The following is the sample config file for reference:



The following is the sample configuration file generated with the import script we demonstrated in this hands-on exercise:

```
Content of main.tf.json
{
    "provider": {
        "vsphere": {
            "vsphere_server": "XX.XX.XX.XX",
            "user": "administrator@vsphere.local",
            "password": "xxxxxx",
            "allow_unverified_ssl": "true"
        }
    },
    "data": {
        "vsphere_datacenter": {
            "dc": {
                "name": "ECM"
            }
        },
        "vsphere_resource_pool": {
            "pool": {
                "name": "DR/Resources",
                "datacenter_id": "${data.vsphere_
                                  datacenter.dc.id}"
            }
        },
        "vsphere_datastore": {
            "datastore": {
                "name": "aabbcc",
```

```
            "datacenter_id": "${data.vsphere_
                              datacenter.dc.id}"
        }
    },
    "vsphere_virtual_machine": {
        "template": {
            "name": "UbantuTest",
            "datacenter_id": "${data.vsphere_
                              datacenter.dc.id}"
        }
    },
    "vsphere_network": {
        "network0": {
            "name": "VM Network",
            "datacenter_id": "${data.vsphere_
                              datacenter.dc.id}"
        }
    }
},
"resource": {
    "vsphere_virtual_machine": {
        "UbantuTest": {
            "network_interface": {
                "adapter_type": "vmxnet3",
                "network_id": "${data.vsphere_network.
                               network0.id}"
            },
            "name": "UbantuTest",
            "resource_pool_id": "${data.vsphere_resource_
                                 pool.pool.id}",
```

```
"datastore_id": "${data.vsphere_datastore.
                datastore.id}",
"boot_retry_enabled": false,
"enable_disk_uuid": false,
"enable_logging": false,
"num_cores_per_socket": 1,
"num_cpus": 3,
"guest_id": "ubuntu64Guest",
"memory": 12288,
"cpu_hot_add_enabled": "true",
"memory_hot_add_enabled": "true",
"firmware": "bios",
"scsi_type": "${data.vsphere_virtual_machine.
             template.scsi_type}",
"lifecycle": {
    "ignore_changes": [
        "custom_attributes",
        "tags"
    ]
},
"disk": {
    "label": "disk0",
    "size": 16,
    "unit_number": 0,
    "thin_provisioned": "false",
    "path": "[ aabbcc] UbantuTest/
            UbantuTest.vmdk",
    "keep_on_remove": "true"
} } } } }
```

This is a typical configuration file that may suit the majority of the workloads present in a typical VMware farm.

Summary

This chapter talked about the reverse-engineering process and the three steps in more detail. Also, it explained how to create the model we want to build for Terraform using a VMware virtual machine as an example.

The chapter started by explaining the core functionality of Terraform and explained the architecture of Terraform and its interaction with providers, provisioners, and destination platforms. Further, the chapter explained the different sections of a configuration file, which is required for a sample VMware virtual machine import into Terraform. After the configuration file was generated, the chapter introduced the import operation of a sample VM into Terraform. The validation step is key to the success of the whole import operation. After importing, you run `terraform plan`, and the outcome should suggest "No Changes." This confirms a successful import.

In the hands-on exercise, we provided an import script to help you generate the point-in-time configuration file for a sample VMware virtual machine. This script can be augmented to fetch any environment-specific object parameters required to be populated in the Terraform configuration file for a successful import.

CHAPTER 5

Debugging for Import Issues and Best Practices

Terraform is a formidable tool in the world of modern infrastructure management, revolutionizing the way we handle IT infrastructure resources. It enables us to treat resources as code, ushering in a new era of control, transparency, and efficiency. This strategy has numerous benefits, most notably the use of version control and the certainty of repeatability in our infrastructure deployments.

Terraform serves as a critical component for the paradigm shift toward infrastructure as code (IaC). By encapsulating our infrastructure as code, we unlock the capacity to orchestrate and manage infrastructure resources in a systematic, programmatic fashion. This translates into the ability to define, provision, and maintain configurations just like we manage software source code, all the while granting us the power of predictability.

In this chapter, we embark on a journey where we delve into the nitty-gritty details of bringing an existing VM under the encompassing umbrella of Terraform's control. This process involves a well-orchestrated sequence of steps that set the path for a seamless transition into the Terraform ecosystem.

S. Bhatia and C. Gabhane, *Reverse Engineering with Terraform*,
https://doi.org/10.1007/979-8-8688-0074-0_5

However, it is crucial to remember that with great power comes great responsibility. Terraform is a formidable tool, but it is not without its potential pitfalls. In this chapter, we will not only explore the mechanics of VM integration with Terraform but also emphasize the utmost importance of precautionary measures. As we traverse this path, we will draw your attention to the safeguards and best practices you should adopt. This, in turn, ensures that your resources are not only efficiently managed but also well-protected from unintended consequences.

Potential Error Scope with Reverse Engineering

There are potential challenges at every stage of the import process when importing existing virtual machines into Terraform. Let's understand the import process and possible issues that can affect your import operation.

The Challenge of Evolving Features

Destination platforms such as VMware and cloud providers are continually evolving and introducing enhancements and new functionalities. These features can enhance performance, security, and management, among other benefits. However, the introduction of these new features may lead to potential conflicts or discrepancies between existing infrastructure configurations and the revised feature set.

To effectively integrate the latest features into existing infrastructure, it is crucial to update both the Terraform core and relevant providers. The Terraform core handles the primary functionalities of the Terraform tool, while providers extend its capabilities to interact with specific destination platforms.

It is suggested that whenever these specifics change for your environment, you should run an import test with a variety of your resources to reflect that the last import logic you had still is relevant. If the import test is failing, then make sure your import logic is adapted accordingly so as you are again making clean imports of your infrastructure resources with your import logic. In the next section, we will discuss the importance of continuously testing the Import logic.

Importance of Testing Import Logic

Before updating the Terraform core and providers, it is essential to assess how these changes might impact the existing infrastructure. The "import logic" refers to the process by which existing resources are imported into Terraform's state and configuration files. Any changes in the destination platform's behavior might affect this import logic, leading to discrepancies during the import process.

To mitigate potential issues, thorough testing of the import logic is necessary. This involves evaluating how existing resources are imported, verifying that all required configuration parameters are fetched correctly, and confirming that the infrastructure remains consistent throughout the process.

To ensure clean imports, follow the suggested steps mentioned in Figure 5-1 when updating the destination platform, the Terraform core, and the providers involved in your reverse-engineering process.

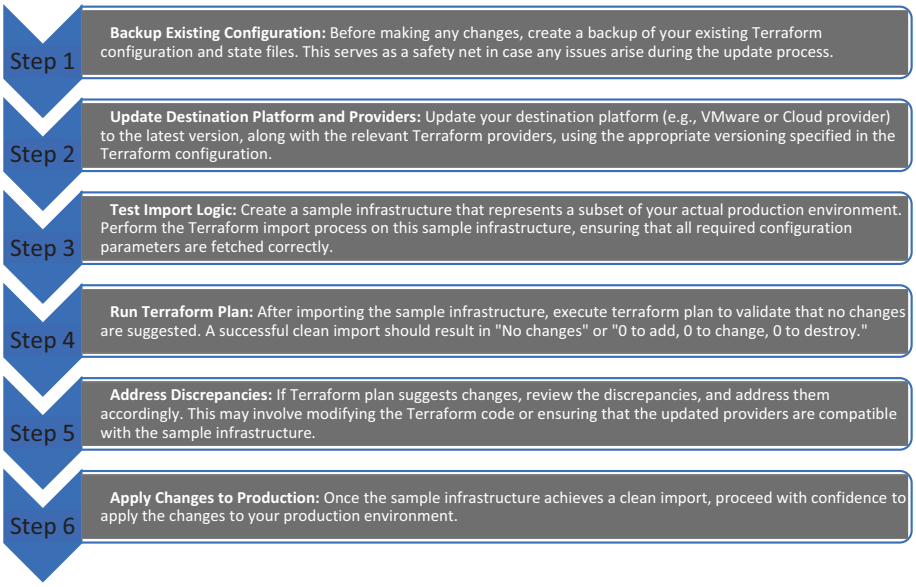


Figure 5-1. *Testing clean imports with sample infrastructure*

Remember that clean imports are not a one-time process but an ongoing practice. As infrastructure evolves, keep updating your Terraform code and fetching mechanisms to match any changes in the source of truth. With proper planning, testing, and adherence to best practices, Terraform empowers you to build a scalable, consistent, and reliable infrastructure that meets the needs of your organization.

Clean Imports: A Guide to Ensuring Accurate Configurations

To streamline the infrastructure provisioning process, many IT professionals turn to Terraform for solutions. And, when performing reverse-engineering tasks, one often encounters challenges related to

fetching missing configuration file parameters from the source of truth. Let's discuss the significance of "clean imports" and demonstrate how to ensure accurate configurations in VMware environments using Terraform.

Understanding the Challenge

During reverse engineering, the process of creating infrastructure code from an existing environment, it is common to encounter incomplete or missing configuration parameters. These parameters could be vital for a successful infrastructure deployment, and without them, the provisioning process may lead to inconsistencies or errors.

When utilizing Terraform to import existing resources into a configuration, discrepancies between the imported state and the desired state specified in Terraform code can occur. These discrepancies might arise due to incomplete data or other inconsistencies in the imported resources.

The concept of "source of truth" is critical in ensuring the accuracy of configurations. The source of truth represents the most reliable and up-to-date information about the infrastructure, usually a central configuration management system. It is essential to fetch all relevant configuration parameters from the source of truth to achieve a clean import. And example of the "source of truth" for VMware infrastructure is the managed object browser (MOB) we discussed in earlier chapters.

Importance of Clean Imports

A clean import refers to a successful Terraform import process where all the required configuration parameters are accurately fetched from the source of truth. When the import process results in no changes suggested by the Terraform plan, it indicates that the imported infrastructure is consistent with the desired state defined in the Terraform code. Clean imports play a pivotal role in the following aspects:

- **Consistency and stability:** Clean imports ensure that the provisioned infrastructure is consistent with the expected state, reducing the chances of configuration drift and unexpected behavior.
- **Reliability:** Importing resources with all the necessary parameters enhances the reliability of the provisioning process, as there are fewer chances of misconfigurations.
- **Collaboration:** Clean imports promote smoother collaboration between team members, as everyone is working with a standardized and accurate infrastructure.

Achieving Clean Imports with Terraform

Let's walk through an example scenario to illustrate how to achieve clean imports in a VMware environment using Terraform.

Scenario: Importing a VMware Virtual Machine

Suppose we have an existing VMware virtual machine that we want to import into Terraform. The virtual machine's configuration file contains vital parameters such as the VM name, CPU count, memory, disk size, and network configurations.

Step 1: Create the Terraform Configuration File Using Import logic

Start by creating a new Terraform configuration file (e.g., `main.tf`) and define the required VMware provider settings.

Step 2: Import the Virtual Machine

Next, execute the `terraform import` command to import the existing virtual machine into the Terraform state. Replace `<vm-id>` with the VMware-assigned ID for the virtual machine.

```
terraform import vsphere_virtual_machine.example <vm-id>
```

Step 3: Initialize Terraform

After importing the virtual machine, initialize Terraform from your working directory so the latest providers are fetched for the Terraform operation.

You can run `terraform init` to initialize Terraform from the working directory.

Step 4: Run terraform plan

Now, run `terraform plan` to check for discrepancies between the imported state and the desired state defined in the Terraform resource.

```
terraform plan
```

Step 5: Identify and Fetch Missing Parameters

During the `terraform plan` execution, you might receive suggestions for changes due to missing or incomplete parameters fetched from the source of truth. Make sure to update your Terraform resource block with all the required configuration parameters. You can perform the necessary changes in your import logic to fetch and reflect the desired changes you need where `terraform plan` suggests no changes to the infrastructure after you import your resource.

Step 6: Verify a Clean Import

Rerun `terraform plan` after updating the Terraform resource. If the output shows "No changes" or "0 to add, 0 to change, 0 to destroy," congratulations! You have achieved a clean import, and your VMware virtual machine is now accurately represented in Terraform.

Step 7: Dry Run and Testing on Different Resources

Before performing the actual import process, use the Terraform `import` command with the `-target` option to perform a dry run on a specific resource. This allows you to observe the expected changes and verify if all required parameters are fetched correctly. Additionally, create test environments to validate the imported infrastructure before applying changes to production environments.

Achieving a clean import is a process, and when it is performed iteratively on a variety of resources present in your infrastructure, it leads to a stage where you can cleanly identify all the required parameters specific to your environment and fetch their values automatically from the source of truth of the destination platform. All this leads to a clean import and successful reverse engineering in your environment.

Understanding the Configuration File

Before we first dive into the debugging of issues with the reverse-engineering process, let's first re-emphasize the importance of configuration files. The configuration file plays a vital role as it defines the desired state of the VM. It typically includes settings such as the VM's name, guest operating system, CPU and memory allocation, and network configurations.

As you begin reverse engineering a VM, it is crucial to pay close attention to the configuration file. Ensure that all the necessary parameters are correctly imported and captured within the file. A small oversight here can lead to involuntary consequences, such as incorrect resource allocation, misconfiguration of network settings, etc.

To demonstrate this, let's look at an example of a Terraform configuration file for a VMware VM:

```
resource "vsphere_virtual_machine" "my_vm" {
 name             = "example-vm"
  guest_id         = "centos7_64Guest"
  num_cpus         = 2
  memory           = 4096
  network_interface {
    label          = "VM Network"
    ipv4_address   = "10.1.1.1"
  }
}
```

In this example, we define a VM named `example-vm` with CentOS 7 as the guest operating system. It is allocated two CPUs and 4096 MB of memory and is connected to the VM network with the IP address 10.1.1.1. Make sure these basic parameters along with the others that are required match your specific VM's configuration. If they are different, then it can lead to an incorrect import of a VM. Make sure your configuration file is reflecting the true point-in-time state of the resource you intend to import.

Importing the Existing VM

Once we have the configuration file ready, the next step is to import the existing VM into Terraform using the `terraform import` command. This command allows us to associate an existing resource into Terraform management.

The import command requires two arguments: the Terraform resource type and the ID of the resource to be imported. For VMware VMs, the resource type is `vsphere_virtual_machine`. However, before proceeding, we need to obtain the correct ID of the VM we want to reverse engineer.

To find the ID of a VM, you can use the VMware vSphere client or managed object browser (MOB) to view the VM's properties. The ID is typically in the format of a universally unique identifier (UUID) and can be found within the VM's properties.

It is crucial to double-check and ensure that you have the correct IDs for the VMs you intend to import. Using the wrong ID can lead to unintended consequences and may impact other VMs that are not supposed to be targeted.

To import the VM, execute the following command in your Terraform project directory:

```
Terraform import vsphere_virtual_machine.my_vm <vm_id>
```

Replace `<vm_id>` with the actual ID of the VM you obtained earlier. This command associates the imported VM with the Terraform resource named `my_vm`.

Verifying the Imported Configuration

After importing the VM, it's essential to verify that the imported configuration aligns with your expectations. Compare the imported values to the original VM's settings and ensure that all the parameters have been correctly captured in the Terraform configuration file.

For example, you can use the `terraform show` command to display the current state of the imported VM:

```
terraform show
```

This shows the current state of the imported VM, including its name, guest operating system, CPU and memory allocation, and network settings. Compare this output with the original VM's properties to ensure that all the necessary information has been imported correctly.

If you notice any discrepancies or missing values, you can update the Terraform configuration file accordingly. Make the necessary adjustments to ensure that the imported VM aligns with the desired state.

Once you have verified the imported configuration, you can proceed with managing the VM using Terraform. Since the VM is now under Terraform management, you can leverage its capabilities to make changes, provision additional resources, or even destroy and re-create the VM if needed. Figure 5-2 shows the summary of the configuration file import steps.

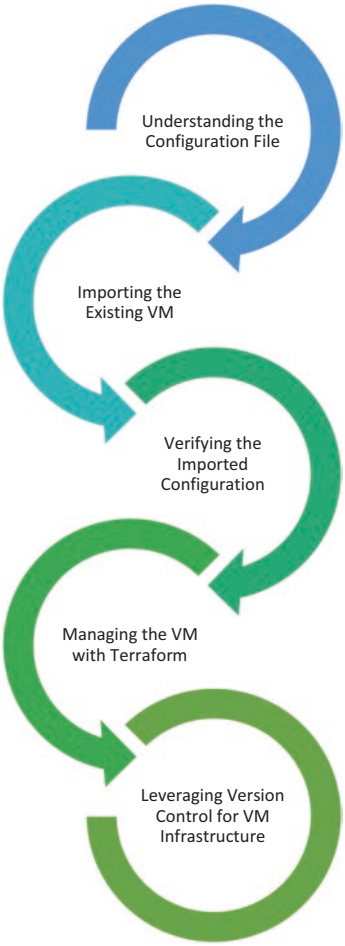


Figure 5-2. *Import steps for configuration file*

Remember, Terraform provides the advantage of version control, allowing you to track and manage changes to your VM infrastructure over time. You can commit your configuration files to a version control system such as Git, enabling collaboration, reverting to previous states, and ensuring consistency across environments.

Provider Version Compatibility for Successful Reverse Engineering

When reverse engineering an existing infrastructure using Terraform, it is essential to ensure that the Terraform provider version matches the version of the destination infrastructure being targeted. A version mismatch between the provider and the infrastructure can result in import issues and other errors.

For example, with a VMware import, if the Terraform provider version is outdated, it may not support certain resources or configurations that are present in the newer version of VMware. In this case, Terraform may fail to import or create the required resources, resulting in errors.

On the other hand, if the Terraform provider version is newer than the VMware infrastructure version, it may introduce features or configurations that are not supported by the infrastructure. This can also result in errors during the import process or when applying changes to the infrastructure.

To mitigate these issues, it is important to carefully review and validate the Terraform configuration file and ensure that it accurately reflects the current state of the VMware infrastructure. Additionally, it is recommended to test the configuration on a nonproduction environment before applying changes to the production infrastructure.

Let's explore another scenario in which we have a VMware infrastructure operating on version 6.7. It is important to highlight that version 6.7 has already reached its end of life (EOL). In this context, our objective is to reverse engineer the existing resources using Terraform.

If we use an outdated version of the Terraform provider that supports only VMware version 6.5, we may encounter import issues and other errors. For example, we may not be able to import or leverage certain features that are available only with VMware version 6.7. New features may relate to security, network configurations, advanced storage options, etc.

On the other hand, if we use a newer version of the Terraform provider that supports VMware version 7.0, which may introduce features or configurations that are not supported by the older VMware infrastructure present in our landscape (VMware version 6.7 our example). New provider may demand a configuration item to be defined in the configuration file, but our platform may not support that because it is running on an older release. This can also result in errors during the import process or when applying changes to the infrastructure.

To avoid these issues, we need to ensure that the Terraform provider version matches the version of the VMware infrastructure that we are targeting.

Terraform configuration file example in YAML format

```
terraform:
  required_providers:
    vsphere:
      source: "hashicorp/vsphere"

      version = "X.Y.Z"  # Use the latest version you found
```

In the previous example, we use a YAML representation of the Terraform configuration file to illustrate the version compatibility considerations. The required providers section specifies the version of the vSphere provider we want to use. It's advisable to use the latest compatible version for your specific use case. You can check the latest version here:

https://registry.Terraform.io/providers/hashicorp/vsphere/latest

The `resources` section defines the `example_vm` resource, which represents the virtual machine to be imported or created. It includes properties such as the name, resource pool, datastore, number of CPUs, memory, network configuration, and disk settings. By using the appropriate provider version, we ensure that these properties align with the capabilities of the targeted VMware infrastructure.

The `data` section defines the necessary data sources (`vsphere_datastore`, `vsphere_network`, `vsphere_datacenter`, and `vsphere_resource_pool`) to fetch information about existing resources in the VMware infrastructure. These data sources help Terraform accurately reflect the current state of the infrastructure during the reverse-engineering process.

By carefully reviewing the Terraform configuration file and selecting the appropriate provider version, we can mitigate version mismatches, configuration issues, and import errors. Taking these precautions will help guarantee a smooth reverse-engineering process and maintain the integrity of the VMware infrastructure.

Remember, it is crucial to test the configuration on a nonproduction environment first to ensure the compatibility and accuracy of the reverse-engineering process.

Overall, ensuring compatibility and consistency between the Terraform provider and the VMware infrastructure version is crucial for a successful reverse-engineering and import process.

Debugging and Troubleshooting Steps with Terraform

Debugging and troubleshooting are important steps when performing a reverse-engineering process using Terraform. In this section we'll show an example of how to configure debugging and troubleshoot issues during the import process.

Suppose we have a Terraform configuration file that imports an existing VMware virtual machine. We are encountering an error during the import process and want to troubleshoot the issue. Here are some of the ways defined to help you find more details about the issue you may want to debug:

1. **Set the debug log level:** We can configure the Terraform log level to debug by setting the `TF_LOG` environment variable to `DEBUG`. This will enable detailed logging and help us identify the source of the issue. For example, we can set the `TF_LOG` environment variable to debug using the following command:

   ```
   $ export TF_LOG=DEBUG
   ```

   Prevent a fork: When working on Terraform projects collaboratively, ensuring consistency across environments becomes crucial. The `TF_FORK` environment variable comes to your rescue in such situations. By setting `TF_FORK=0`, you prevent Terraform from forking and executing multiple copies of itself in parallel, ensuring a single process for the entire configuration. This helps in avoiding race conditions and enhances predictability when dealing with shared state or resources.

   Here is a Bash command example:

   ```
   export TF_FORK=0
   terraform apply
   ```

   Define logs verbosity: There are ways to define the different logging levels that help in debugging. Terraform provides detailed logging to aid developers in understanding the execution flow and identifying potential issues. The `TF_LOG` environment variable allows you to control the logging verbosity, with five possible levels: `TRACE`, `DEBUG`, `INFO`, `WARN`, or `ERROR`.

Here is a Bash command example:

```
export TF_LOG=DEBUG
terraform plan
```

Save logs at custom location: While the default logs are displayed on the console, you might want to save them for later analysis or debugging purposes. With `TF_LOG_PATH`, you can specify a custom file path to save Terraform logs.

Here is a Bash command example:

```
export TF_LOG=DEBUG
export TF_LOG_PATH="/path/to/Terraform.log"
terraform apply
```

2. **Run the Terraform import command:** We can use the Terraform import command to import the VMware virtual machine resource into our Terraform state file. For example, we can use the following command to import the resource:

   Code:

   ```
   $ terraform import vsphere_virtual_machine.my_vm /
   datacenter/vm/my_vm
   ```

3. **Review the debug logs:** After running the import command, we can review the debug logs to identify any errors or issues. The debug logs will provide detailed information about the import process, including the resource being imported, any dependencies, and any errors encountered. For example, we can use the following command to view the debug logs:

   ```
   $ terraform show -debug
   ```

4. **Troubleshoot the issue:** Based on the debug logs, we can troubleshoot the issue and make any necessary changes to the Terraform configuration file. For example, if we find that the import command failed due to a version mismatch between the Terraform provider and the VMware infrastructure, we can update the provider version to resolve the issue.

Through configuring, debugging, and troubleshooting with some of the ways defined earlier, we can identify and resolve issues during the import process and successfully reverse engineer the existing VMware infrastructure using Terraform.

When dealing with errors or issues, it is essential to protect sensitive information that might be present in your logs. Always ensure that any personally identifiable information (PII), access credentials, or sensitive data are obfuscated or removed from crash logs before sharing them with others.

Best Practice for Debugging and Troubleshooting While Performing Reverse Engineering

In addition to the previously mentioned ways of debugging, there are some useful tips and best practices that can help with your debugging and troubleshooting when performing a reverse-engineering process using Terraform.

1. **Check the Terraform logs:**

   In addition to setting the `TF_LOG` environment variable to `DEBUG`, we can also review the Terraform default logs to identify any errors or issues during the import process. The logs provide detailed

information about the import process, including any resources that were successfully imported and any errors encountered.

2. **Use the terraform plan command:**

   The `terraform plan` command can be used to preview the changes that will be made to the infrastructure before applying them. This can help to identify any potential issues or errors before they occur. For an ideal import, `terraform plan` should suggest no changes to the infrastructure when it is run first time after the import.

3. **Use the `terraform refresh` command:**

   The `terraform refresh` command can be used to update the state file with the latest information from the infrastructure. This can help to ensure that the state file accurately reflects the current state of the infrastructure and avoid any import issues due to outdated information.

4. **Check the destination platform API logs:**

   In some cases, the issue may be related to the destination platform (e.g., VMware) API itself. In such cases, reviewing the API logs can provide additional information about the issue.

5. **Check the Terraform provider documentation:**

   If the issue persists, it may be helpful to review the Terraform provider documentation for the specific resource being imported. The documentation may provide additional information about any known issues or specific configuration requirements.

Terraform Issues and Support

If you encounter a bug while working with Terraform, here are some references to help deal with it:

- **Check GitHub:**

  Before reporting a bug, search the GitHub repositories (`https://github.com/hashicorp/Terraform` for core issues and `https://github.com/Terraform-providers` for provider plugins). It is possible that someone has already reported it, and there might be an ongoing discussion or even a solution. This helps a lot to know the available workaround of the problems that others have noticed too.

  If you do not find a reference, you can report the issue here on GitHub:

  `https://github.com/hashicorp/Terraform/issues`

- **Check the provider documentation:**

  Sometimes, an issue might be specific to a particular provider's implementation. Always check the respective provider's documentation for any known limitations or bugs. The issue you are facing may already be a known limitation with the provider you are working with.

Example Bug Report

If you are required to submit a bug report with the Terraform community, here is an example bug report for your reference that includes the key areas you should cover when reporting bugs and seeking help with the community:

Title:
Description:
Steps to reproduce:
Expected behavior:
Actual behavior:
Additional information:

Here is one complete sample bug report we are pasting for your reference where the user is not able to create an AWS EC2 instance using Terraform:

Title: Unable to create AWS EC2 instance using Terraform 1.0.2

Description:

I'm facing an issue while trying to create an EC2 instance using the latest version of Terraform (v1.0.2). The error message I receive is: "Error: Error launching source instance: InvalidSubnetID.NotFound: The subnet ID 'subnet-12345678' does not exist."

Steps to Reproduce:

1. Set up AWS credentials in Terraform.
2. Use the following Terraform code snippet:

```
resource "aws_instance" "example" {
ami = "ami-0c55b159cbfafe1f0"
instance_type = "t2.micro"
subnet_id = "subnet-12345678"
}
```

3. Run `terraform init`, `terraform plan`, and `terraform apply`.

Expected Behaviour:

```
Terraform should create an EC2 instance in the specified subnet
without any errors.

Actual Behaviour:

Terraform throws an error indicating that the specified subnet
ID does not exist, even though it does.

Additional Information:
- Terraform Version: v1.0.2
- AWS Provider Version: v3.0.0
- Operating System: macOS Big Sur 11.4

Note: I've confirmed that the subnet ID 'subnet-12345678' is
valid and exists in the region I'm using.

Thank you for your attention to this matter.
```

Summarizing How Import Issues Can Be Avoided

Here is short summary of what we have discussed so far in this chapter to avoid import issues when reverse engineering an existing infrastructure using Terraform:

- Ensure that the Terraform provider version matches the version of the platform (in our case the VMware infrastructure) that you are targeting.
- Validate the Terraform configuration file to ensure that it accurately reflects the current state of the resource.
- Check for any missing or outdated resources and objects in the Terraform configuration file and update them accordingly.

- Use the `terraform plan` command to preview any changes that will be made to the infrastructure before applying them.
- Stay up-to-date with the latest releases and updates for both the Terraform provider and destination infrastructure to ensure that you are using the most current and compatible versions.
- When importing resources using Terraform, it is important to use specific resource identifiers such as resource IDs, resource names, and path names. This helps Terraform to accurately identify and import the correct resources.
- It is important to validate resource dependencies when importing resources using Terraform. Terraform may not be able to import a resource if its dependencies are not met.
- Avoid circular dependencies that occur when two or more resources depend on each other. This can result in import issues and errors. To avoid circular dependencies in Terraform and vSphere, you should carefully define resource dependencies. Here's a small Terraform code snippet as a symbolic example illustrating resource dependencies using vSphere virtual machines (VMs) and datastores:

```
resource "vsphere_datastore" "example_datastore" {
  name = "my_datastore"
  # Other attributes for defining the datastore
}
```

```
resource "vsphere_virtual_machine" "example_vm" {
  name         = "my_vm"
  datastore_id = vsphere_datastore.example_datastore.id
  # Other attributes for defining the VM
}
```

In this symbolic code, we are creating a VM resource that depends on a datastore resource. The `datastore_id` attribute of the VM is set to the ID of the datastore, establishing a clear dependency. This is an example of how you should structure your Terraform configurations to avoid circular dependencies. Ensure that resources rely on other resources in a logical and linear manner so that they don't create circular relationships.

- Test the Terraform configuration on a nonproduction environment before applying changes to the production environment. This helps to identify and resolve any import issues or errors before they affect the production environment.
- It is important to document the process of reverse engineering using Terraform. This helps to keep track of the changes made to the infrastructure and to troubleshoot any issues that may arise.

By following these best practices, you can avoid import issues and ensure a successful reverse-engineering process using Terraform.

Best Practices for Terraform State Management

Terraform state is the file that is required by Terraform for its management and for core operations on your infrastructure. It is not meant to be altered by Terraform users. Here, we will explore essential practices to manage your Terraform state effectively, along with practical examples of using the `terraform state` command, which will come handy for your day-to-day management.

Backups, Versioning, and Encryption

Backing up your Terraform state files is a fundamental step in protecting your infrastructure configurations. State files contain sensitive information and are critical for your infrastructure's integrity. Regularly backup these files and consider versioning to maintain historical states in case of rollbacks or auditing requirements.

Additionally, apply encryption to safeguard sensitive data within your state files.

Example (Linux shell script using AWS S3 for state backup):

Backup the Terraform state file to an S3 bucket with versioning enabled

```
aws s3 cp Terraform.tfstate s3://your-state-bucket/
Terraform.tfstate
```

Enable versioning on the S3 bucket

```
aws s3api put-bucket-versioning --bucket your-state-bucket --
versioning-configuration Status=Enabled
```

Do Not Edit Manually

It is crucial to avoid manually modifying your Terraform state files. Instead, rely on Terraform commands to manage your infrastructure to prevent inconsistencies and potential issues. Directly editing state files can lead to a mismatch between the actual infrastructure and the state, causing problems during future Terraform operations.

Main Keys in the Terraform State File

Understanding the main keys present in the Terraform state file provides insight into how Terraform manages your infrastructure. Use the `jq` command (a lightweight JSON processor) to extract and analyze the main keys.

Here's an example (using `jq` to extract main keys from the Terraform state file):

```
cat terraform.tfstate.backup | jq 'keys'
```

Here's how it breaks down:

- **"lineage":** A unique identifier for the state file that persists after initialization. It helps in tracking the provenance of the state file, aiding in auditing, and debugging.
- **"modules":** The main section that holds information about the configured modules and their resources.
- **"serial":** An incrementing number representing the version of the state. Terraform uses this to handle concurrent state updates and avoid conflicts.
- **"Terraform_version":** An implicit constraint specifying the minimum Terraform version required to read and apply the state.

- **"version":** The state format version. Different Terraform versions use different state formats, and this key indicates the format being used.

Utilizing the terraform state Command

The `terraform state` command provides a set of subcommands that allow you to interact with the Terraform state in various ways.

a) **Move/rename modules:**

Use `terraform state mv` to move or rename modules within your state file. This is helpful when refactoring your infrastructure without losing existing state data.

Here's an example of moving a resource within the state file:

```
terraform state mv aws_instance.example aws_instance.
new_example
```

b) **Safely remove resources from the state:**

The `terraform state rm` command is useful for safely removing resources from the state file without destroying them in the actual infrastructure. This can help when you need to remove a resource from management by Terraform.

Here's an example of removing a resource from the state:

```
terraform state rm aws_security_group.example
```

c) **Pull remote state:**

The `terraform state pull` command allows you to observe the current remote state without making any changes. It is useful for retrieving the state data stored in remote backends like S3.

Here's an example of pulling the remote state:

```
terraform state pull
```

d) **List and show resources:**

The `terraform state list` and `terraform state show` commands provide detailed information about the resources managed by Terraform. These commands help in debugging and understanding your infrastructure state.

Here's an example of listing resources and showing details:

```
terraform state list
terraform state show aws_instance.example
```

Hands-On Exercise

In this exercise we are going to demonstrate how a change in configuration file affects the resource you are importing from a VMware VM farm.

Let's say we have an existing VM named `web-server` running on a VMware vSphere infrastructure. We want to manage this VM using Terraform, so we need to reverse engineer it by importing it into Terraform.

To do this, we first need to create a Terraform configuration file that describes the `web-server` VM's configuration. The `resource` section of the configuration file could look something like this:

```
resource "vsphere_virtual_machine" "web-server" {
  name             = "web-server"
  datastore        = "datastore1"
  guest_id         = "ubuntu64Guest"
  memory           = 4096
  num_cpus         = 2
  network_interface {
    network_id = "network1"
  }
}
```

This configuration file specifies the VM's name, datastore, guest operating system, memory and CPU allocation, and network settings.

Next, we need to use the `terraform import` command to bring the `web-server` VM under Terraform management. If we have already set up the vSphere provider and authenticated with our vSphere infrastructure, we can run the following command:

```
terraform import vsphere_virtual_machine.web-server <web-
server-UUID>
```

Here, `<web-server-UUID>` is the UUID of the `web-server` VM as obtained from the vSphere client or vSphere Web Client.

Once we run this command, Terraform will import the `web-server` VM into its state file, allowing us to manage it using Terraform.

Now, suppose we want to modify the `web-server` VM's configuration to increase its memory allocation to 8192 MB. We can simply update the configuration file as follows:

```
resource "vsphere_virtual_machine" "web-server" {
  name             = "web-server"
  datastore        = "datastore1"
  guest_id         = "ubuntu64Guest"
  memory           = 8192
```

```
  num_cpus          = 2
  network_interface {
    network_id = "network1"
  }
}
```

Finally, we can apply these changes using the `terraform apply` command:

```
terraform apply
```

This will update the `web-server` VM's configuration in the vSphere infrastructure based on the new configuration file.

In this way, we have successfully reverse engineered the `web-server` VM using Terraform and can now manage it as code.

How do you use the `terraform plan` command in the previous example to cross check that things went well?

The `terraform plan` command can be used to generate an execution plan that shows what changes Terraform will make to the infrastructure when the `apply` command is run. This can be a useful tool for cross-checking that our Terraform configuration is correct and that the changes we intend to make to the infrastructure match our expectations.

In the example I provided earlier, we reverse engineered a VMware virtual machine (VM) named `web-server` using `terraform import` and a configuration file. To use the `plan` command to cross-check that everything is working correctly, navigate to the directory where the Terraform configuration file is stored.

Run the `terraform plan` command. This will generate a plan that shows the changes that Terraform will make to the infrastructure based on the current state of the configuration.

In our example, if we run `terraform plan` after importing the `web-server` VM, Terraform should show us a plan that includes only the modification we made to the VM's memory allocation. The output of the plan command might look something like this:

```
# vsphere_virtual_machine.web-server will be updated in-place
  ~ resource "vsphere_virtual_machine" "web-server" {
        cpu_hot_add_enabled    = false
        cpu_hot_remove_enabled = false
        guest_id               = "ubuntu64Guest"
        memory                 = 4096 -> 8192
        name                   = "web-server"
        num_cpus               = 2
        num_cores_per_socket   = 1
        scsi_type              = "pvscsi"
        wait_for_guest_net     = true
        wait_for_guest_net_timeout = 0
        network_interface {
            network_id = "network1"
        }
    }
Plan: 0 to add, 1 to change, 0 to destroy.
```

Here, we can see that Terraform will modify the memory allocation of the `web-server` VM from 4096 MB to 8192 MB.

By reviewing the output of the `plan` command, we can ensure that Terraform will make the changes we expect before running the `apply` command. This helps us catch errors or unexpected changes before they are applied to the infrastructure, reducing the risk of mistakes and minimizing downtime.

In addition to showing what changes Terraform will make to the infrastructure, the `plan` command can also be used to validate the Terraform configuration file and ensure that all required resources and dependencies are defined correctly.

For example, if we were to introduce an error in our Terraform configuration file, such as referencing a nonexistent datastore or network, the `plan` command would show us an error message indicating that the resource could not be created. This allows us to catch errors early in the development process, before applying changes to the infrastructure.

Furthermore, the `plan` command can be used to perform a "dry run" of changes before applying them to the infrastructure. This can be useful for testing changes to the Terraform configuration file without making any changes to the infrastructure. By running `terraform plan -detailed-exitcode`, we can receive a detailed exit code that will tell us whether any changes would be made to the infrastructure. This allows us to validate the correctness of our changes without applying them and risking unintended consequences.

Overall, the `terraform plan` command is an essential tool and can be used as debugging and troubleshooting tool for the Terraform workflow, allowing us to validate and test our Terraform configuration files, catch errors early, and ensure that our infrastructure changes are aligned with our expectations.

Summary

In this chapter, we explored the complexities of integrating current VM configurations into Terraform. We discussed the importance of the configuration file in reverse engineering and walked through the process of correctly understanding and importing configurations. The importance of provider version compatibility was highlighted, ensuring a successful reverse engineering experience. We explored debugging and troubleshooting steps to pre-emptively address potential errors and offer a comprehensive summary of how import issues can be effectively avoided. The chapter's highlight was a deep dive into obtaining clean imports in VMware using Terraform. You now have the expertise to confidently tackle import issues with real solutions and thorough testing.

CHAPTER 6

Life-Cycle Management After Import

In the previous chapter, we discussed some of the common import issues and best practices with Terraform. This chapter will cover potential Terraform integrations and resource life-cycle management following the import process. Terraform integrations allows us to explore the possible workflows for reverse engineering and to leverage the infrastructure automation capabilities to help solve complex business challenges. Considering that you are well versed at this point with the concepts of reverse engineering and the import operation with Terraform, now it is time to understand further use cases that apply after the import operation to implement infrastructure automation in routine IT activities.

This chapter starts with introducing different Terraform integrations in diverse IT infrastructure domains. Next, we will outline the most typical use cases for IT automation and offer standard procedures that can implement them. The integration of Terraform with DevOps and configuration management tools is explained in the second half of the chapter. Lastly, we will show how to integrate Terraform with SaltStack in a hands-on exercise.

S. Bhatia and C. Gabhane, *Reverse Engineering with Terraform*,
https://doi.org/10.1007/979-8-8688-0074-0_6

Terraform Integrations

The Terraform ecosystem offers a wide range of integration opportunities with diverse infrastructure use cases and environments. After importing an existing resource, the next step is to make use of the powerful automation capabilities provided by Terraform. Considering the Terraform capabilities, there are a variety of partners offering solutions to use Terraform in different ways. Broadly, the Terraform integrations are categorized into two subcategories, namely, workflow partners and infrastructure partners. Let's learn more about them.

Workflow Partners

HashiCorp's Terraform is available in several variants.

- **The Terraform cloud version** is a hosted service capable of providing Terraform runs in a consistent and reliable environment including easy access to the shared state and secret data, access controls for approving changes to infrastructure, capabilities to define policy controls governing the contents of Terraform configurations, etc.
- **The Terraform enterprise version** provides a private instance to enterprises that they can deploy on-premises, and it mostly includes all the advanced features available in the Terraform cloud.
- **The Terraform core** includes the open-source binaries, which are the backbone of Terraform's infrastructure as code. All other advanced features such as role-based access, policy controls, and REST API support are not available with the Terraform core.

The workflow partners offer integrations with the Terraform enterprise or SaaS offering of Terraform, i.e., the Terraform cloud. These workflow partners allow customers to use their existing platform within a Terraform run.

Let's discuss some of these workflow use cases where Terraform has integrations and supports workflows in collaboration with different industry partners.

- **Cost management:**

  There are partners that offer integrations with Terraform to analyze the impact of new infrastructure cost and apply cost governance. Using the Terraform configuration files as a standard definition of how an application/workload's cost is estimated, you can use the Terraform cloud and enterprise APIs to automatically analyze estimated cloud financial data or use Terraform's user interface to provide direct access to finance information to review costs. By doing this, you can help eliminate many slower oversight processes. The Terraform cloud estimates costs for many resources found in your Terraform configuration. It displays an hourly and monthly cost for each resource, as well as the monthly delta. Once enabled, when a Terraform plan is run, Terraform will reach out to the AWS, Azure, and/or GCP cost estimation APIs to present the estimated cost for that plan, which can be used accordingly within your financial workflow. You can also export this estimation report as JSON.

  Terraform can also play a role in the cost optimization after the resources are deployed. Terraform can be integrated with tools such as IBM Turbonomic, where

the tool provides optimization recommendations for resources running already in the public cloud. Users can integrate the tool with Terraform and/or your CI/CD pipeline. Further updates of the Terraform configuration are performed with the revised recommended configuration; then when the `terraform run`, `plan`, and `apply` commands are run, a newly optimized and compliant resource is provisioned on the platform.

There is a detailed blog from HashiCorp on the cost management workflows and potential use cases that Terraform offers. You can find it here:

https://www.hashicorp.com/blog/a-guide-to-cloud-cost-optimization-with-hashicorp-Terraform

- **Security:**

  There are vendors such as Prisma Cloud by Palo Alto Networks that offer integrations to detect Terraform configuration errors that do not align with the organization-defined security and compliance requirements. These vendors offer support for scanning resource configuration files before they are deployed. This is called *preplan support*. It enables you to scan the code before the plan files are even generated. This feature streamlines and accelerates development because you no longer wait to scan the plan file to identify security issues. These vendor tools allow you to assess runtime environments and IaC on a single policy, rather than correlating policy definitions across different tools. This reduces the overhead of

maintaining multiple policies and the associated rules across different tools and languages, which can easily drift apart. They offer "one policy" to meet the security and compliance requirements of the entire organization.

- **Observability and monitoring:**

  Partners that offer this integration focus on detecting infrastructure changes and ensuring optimal observability is in place. Different vendors such as Datadog offer observability as code (OaC) with help from Terraform. This means configuring the observability of any application deployment using the definitions and configurations through files. Similar to infrastructure as code, observability as code uses code to promote automation and repeatability within observability workflows, aiming to give developer teams visibility into the entire software stack. OaC helps developers drive real-time action via live insight—to ensure that teams achieve service-level objectives and optimize critical business metrics with enhanced efficiency.

- **Continuous integration/continuous deployment:**

  There are partners that focus on providing support with Terraform for continuous integration and continuous deployment. Running Terraform with CI/CD can boost your organization's performance and ensure consistent deployments. Partners that have built-in integrations with Terraform are GitLab and Visual Studio.

- **Single sign-on (SSO):**

  Partners in this space focus on providing authentication to end users for single sign-on. Organizations can utilize the Terraform cloud to set up SAML SSO, an alternative to conventional user management. SSO provides administrators with greater control over securing access to the projects, workspaces, and managed resources of your company. SSO enables your company to consolidate user management for software-as-a-service (SaaS) providers like the Terraform cloud, enhancing security and accountability for user and identity management inside an organization. Some SSO providers include Microsoft Azure AD, Okta, and SAML. Given the power of Terraform, integrating multifactor authentication (MFA) along with SSO is a best practice when implementing SSO.

- **Low code/no code:**

  Partners focus on implementing, deploying, and delivering IT, supply chain, operations management, business, and other workflows. Tools such as ServiceNow offer integration with Terraform and enable the automated management of complete industrial workflows.

Most workflow partners create "run tasks" that let them integrate their services directly into a Terraform workflow. The Terraform cloud offers custom integrations in the form of these run tasks with different technology partners where they have access to plan details in between the plan and apply phases and can display custom messages within the run pipeline as well as prevent a run from continuing to the apply phase. Run tasks let the Terraform cloud execute tasks in external systems at certain

points in the Terraform cloud's run life cycle. Specifically, Terraform cloud users can configure run tasks to execute during the pre-plan, post-plan, and pre-apply run stages. Run tasks allow the Terraform cloud to execute tasks in external systems at specific points in the Terraform cloud's run life cycle. This integration offers much more extensibility to Terraform cloud customers, enabling them to integrate third-party services into the Terraform cloud workflow. This feature allows users to add and execute these tasks during the pre-plan, post-plan, and pre-apply stages. Figure 6-1 highlights the Terraform workflow and infrastructure partners that offer diverse integration possibilities with Terraform.

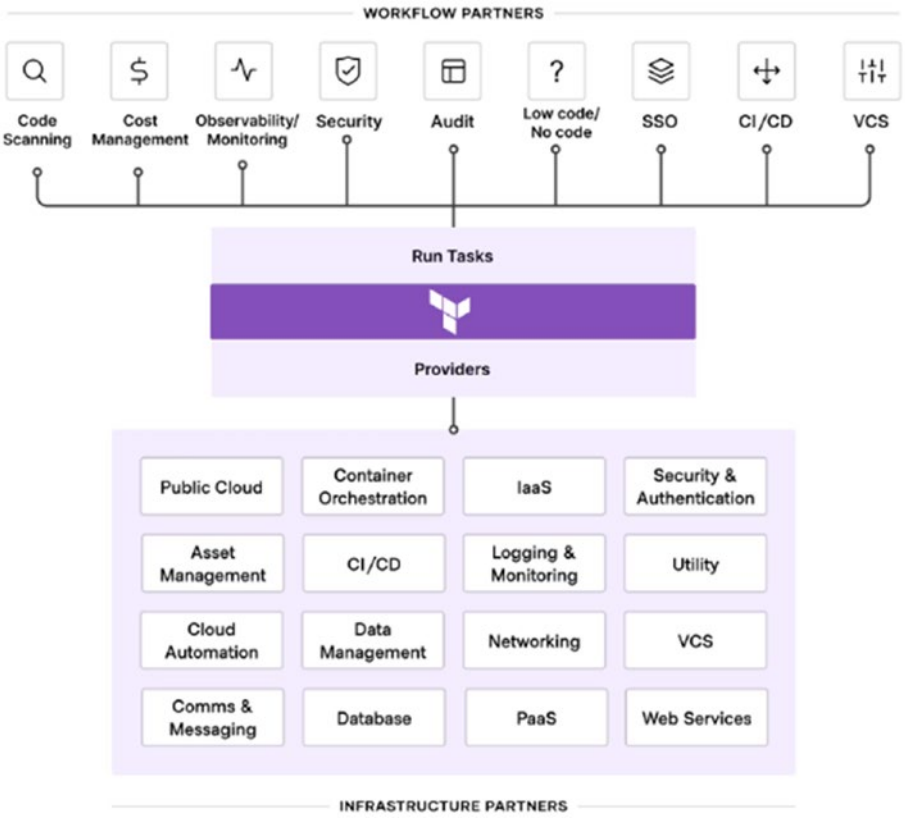


Figure 6-1. *Terraform partners and integration possibilities (source: https://developer.hashicorp.com/terraform/docs/partnership)*

Infrastructure Partners

These partners build Terraform providers and enable customers to leverage Terraform to manage resources exposed by their platform APIs. These providers are accessible to users of all Terraform editions such as the Terraform core, the enterprise version, or the Terraform cloud. Partners in the infrastructure space offer support for most of the variants available with Terraform. As depicted in Figure 6-1, there are different use cases offered by these infrastructure partners. Let us discuss some of them briefly.

- **Public cloud:**

  A large number of infrastructure partners including Google, Amazon, Microsoft, IBM, etc., offer a range of services on their platforms with Terraform including IaaS, SaaS, and PaaS management. This integration with a diverse set of services offered with different public cloud providers make Terraform a cloud-agnostic tool.

- **Container orchestration:**

  With the wide adoption of microservices, there are different infrastructure partners that offer support for container provisioning and deployment. This makes infrastructure automation possible with container platforms as well.

- **Infrastructure as a service (IaaS):**

  Typically for platforms hosted on-premises, there are infrastructure partners that offer support and solutions for the management of platforms such as storage, networking, and virtualization.

- **Asset management:**

  Asset management is an important service in the infrastructure, where the partners offer asset management of key organization and IT resources, including software licenses, hardware assets, and cloud resources.

- **Data management:**

  Partners in this space offer capabilities to manage data center storage, backup, and recovery solutions with Terraform automation.

- **Comms and messaging:**

  For notification via emails and alerting, there are partners offering integration with email and messaging platforms with Terraform.

- **Version control systems (VCS):**

  These partners focus on controlling versions of the code for projects, teams, and repositories within Terraform.

All these infrastructure partners offer integration by building and publishing a plugin called a Terraform provider. As you learned, these are binaries written in the Go language that communicate with the Terraform core over an RPC interface. These providers act as an intermediate layer for transactions of external APIs offered by the destination platforms and services we briefly highlighted in this section.

Terraform Provisioners for Integrations

Chapter 2 introduced Terraform provisioners. In this section, we will introduce how provisioners can be leveraged and integrated into the direct configuration management of a resource that we want to manage with Terraform. As part of infrastructure operations, you are at times required to directly access the resource you are managing with Terraform. For example, you might want to perform some configuration management operations inside the resource or pass some data to complete the provisioning of a resource. For that you need to interact with the remote servers over SSH or WinRM. Or you can push some controls during the boot of that resource. All this is possible with help from provisioners that can pass data by logging into the server and executing instructions directly on the server.

Provisioners mainly deal with configuration activities that happen after the resource is created. The operations involve interacting with files, executing CLI commands, or even executing a script on the resource. Once Terraform is successfully initialized, it is ready to accept connections. These connections are important because then Terraform can log into these instances and perform the required operations. Let's discuss different types and how they can be leveraged for integrations. Please note that provisioners are part of the resource code block in the configuration file.

Local-exec Provisioner

This is the simplest provisioner as it executes on the machine that hosts and executes Terraform commands. If Terraform is installed in the local machine, then local-exec provisioners would run from the same machine. For local-exec provisioners to run and do configuration management on the destination platform, the resource that is being provisioned or modified, the Terraform host should be able to reach that resource. When

Terraform configuration is applied, local-exec can run any shell script or command and execute that on the local host where Terraform is running. Figure 6-2 explains the different types of provisioners and their actions.

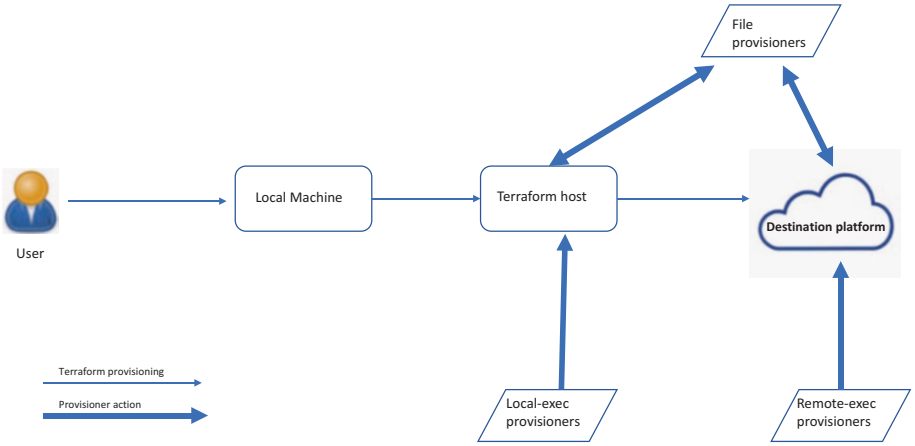


Figure 6-2. *Terraform provisioners in action*

File Provisioners

File provisioners provide a way to copy required files or artifacts from the host machine where Terraform is running to the target resources that Terraform is creating or modifying. File provisioners come in handy when you want to transport certain script files, configuration files, artifacts, etc., in the form of JAR files or binaries. These file provisioners can talk to the destination resource when the resource is created or boots for the first time. File provisioners create a connection block between the Terraform host and destination resource to transfer the required files or scripts.

Remote-exec Provisioners

Remote-exec provisioners are like local-exec provisioners except the commands or scripts are executed on the target resource on the destination platform instead of the Terraform host or local host. This is

accomplished by using the same connection block that a file provisioner uses to transfer files to the destination resource. We can use remote-exec provisioners to run single or multiple commands at the same time.

Therefore, provisioners extend into the space of configuration management and enable us to execute any command or script on the target resource. This means a lot of capabilities for infrastructure automation. It opens up great abilities and allows infrastructure admins to perform automated actions on the OS and application layers.

Typical Terraform Integration with Infrastructure Ecosystem for Automation

For the IT operations world, the real power of Terraform is mined when it is helping with the day-to-day repeatable tasks, which otherwise take quality time to complete. However, not every environment is the same. The IT ecosystem, tools, and business processes are unique for every environment. With careful designing and planning, Terraform can be integrated into your IT infrastructure and can be easily adopted. In this section, we will talk about a sample integration that was once merely a dream for many infrastructure administrators but is now possible because of Terraform. But let's first discuss in detail the self-service and ZeroOps concepts.

Self-Service

This is an approach where users are given the desired access on the platform and given resource access to enable them to perform the basic operations themselves without requiring assistance from IT support personnel. In the IT operations world, self-service is changing the way organizations assist their end users, taking the control for most common tasks from IT admins and giving it directly to end users. This has several advantages if implemented successfully.

- **Improved SLA for service requests:**

  With self-service, the service level agreement (SLA), i.e., the commitment by IT admins to complete a task in a specific amount of time, is considerably reduced. This is because self-service empowers the end users, and thus IT tickets are not going into traditional operations queue where they need to be worked on by experts. With self-service, users can use the ticketing portal (like ServiceNow, Remedy, etc.) to fill in the required details in a service request and submit the request. The control then automatically is passed to the underlying infrastructure automation receiver, and the full request is orchestrated from end to end. Once a request is fully completed with automation, control is then given back to the ticketing tool confirming the status of task execution. Therefore, when manual intervention is removed, the turnaround time is improved for the tasks.

- **Enhanced customer experience:**

  All services are mostly designed considering the end customers' experience. For IT operations, their customers are the application teams, which need resources from the infrastructure. With self-service schemes, it enhances their experience because control is in their hands, and they decide the urgency and execution needs of the task. They are not required to follow up with respective individuals in operation to get the required task done in time.

Therefore, we say that self-service provides businesses with a unique opportunity to improve customer service while reducing the cost and strain on IT support teams.

ZeroOps

ZeroOps is also called NoOps or no operations. This is the concept in an IT environment when it becomes so automated and extracted from the underlying infrastructure that there is no need for dedicated teams to provide the IT services in-house. This is possible in IT infrastructure when there is a set of practices defined in an environment focusing primarily on automation and defining their infrastructure as code. Infrastructure automation tools such as Terraform, SaltStack, Ansible, Chef, etc., have made the goal of ZeroOps practically possible.

Though it may be too ambitious for certain organizations to completely isolate the need of an IT operations team, the presence of these automation tools is a step in the right direction to bring needed automation to the IT infrastructure. These tools are a win-win for both the IT administrators and the organization. This is because IT administrators no longer would need to keep doing repetitive and redundant tasks but could spend time on managing the infrastructure as code and handling instances at scale. And for organizations, it helps in saving cost and improving employee productivity.

Now that you understand the core concepts, let's look at the possible integration that facilitates self-service and ZeroOps. Here we are using the example of VMware on-prem, but this is possible for any environment that has support for Terraform. See Figure 6-3.

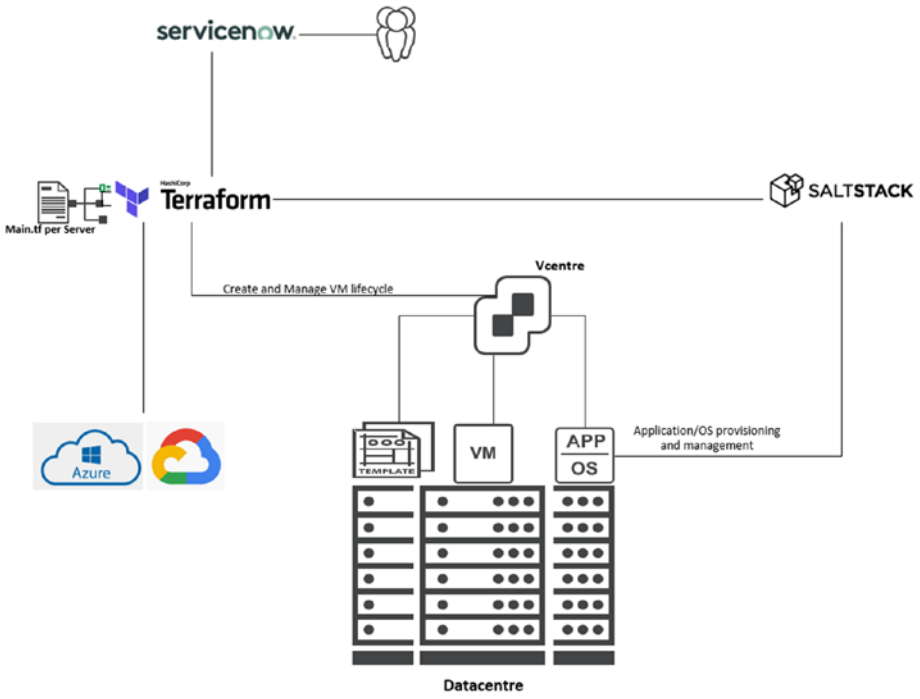


Figure 6-3. *Terraform and integration with the infrastructure ecosystem*

From Figure 6-3, you can see that Terraform is a tool that allows you to define infrastructure as code, does infrastructure automation, and provides the right interaction of different infrastructure ecosystem components. It has rich capabilities where it can be integrated with different platforms such as Microsoft Azure, Google Cloud, or VMware.

SaltStack is a configuration management tool and provides automation at the operating system and application levels. SaltStack works in a master-minion pair, where minions are the agents that get deployed on the individual servers, and they talk to a specified master over a specified TCP/IP port. The master can instruct minions to perform a series of commands and actions based on the state defined on the master.

ServiceNow allows administrators to offer self-service catalogs. These catalogs are defined based on each task that is desired to be automated. Once the catalog is defined and mapped with the user requirements and with requirements to invoke automation on the Terraform (usually inputs needed to invoke Terraform modules), users then just need to log in to the ServiceNow ticketing tool. Then they can perform defined tasks in their back-end infrastructure landscape automatically. ServiceNow offers direct integration with Terraform; it interacts with the Terraform core, enterprise, or cloud version in the background and works as an orchestrator to get the task finished. Once the task is completed, it then sends an acknowledgment to the users.

For example, consider that the user has a requirement to expand an existing disk. With Terraform and SaltStack integration, this job can be done automatically. Users can raise requests in the service catalog. Service now (or a ticketing tool) can then pass the control to Terraform and increase the disk size from the back end. Once the required disk size is increased from the platform, Terraform can then pass control to SaltStack to expand the disk at the operating system level. Finally, once the task is complete, SaltStack can notify ServiceNow and complete the entire workflow. Therefore, a complete workflow can be orchestrated.

Terraform Use Cases

We discussed Terraform integrations in the previous sections of this chapter. A discussion of Terraform integrations is not complete without discussing the most common use cases of Terraform, including where Terraform can define our infrastructure as code. This empowers and enables the smart management of our infrastructure resources. Terraform allows the use of a consistent workflow to safely and efficiently provision and manage your infrastructure throughout its life cycle. Let's understand some of the prime use cases where Terraform shines.

Multicloud Deployment

Organizations are increasingly moving to the public cloud. One common trend among organizations moving to the cloud is to leverage offerings from more than one public cloud. This is especially true when organizations want to avoid vendor lock-in situations, leverage offerings that suit their applications, and get cost advantages when they split their footprint among different public cloud providers. Provisioning infrastructure across multiple clouds also increases the fault tolerance, allowing for more graceful recovery from the cloud provider outages.

This trend imposes challenges as well. This is because each cloud provider has its own interface, tools, and workflows. To fully adapt to cloud services, leveraging automation is especially important. Therefore, a Terraform presence allows the use of the same workflows to manage multiple cloud providers and handle their cross-cloud dependencies. The use of a common language (HCL) simplifies management and orchestration for large-scale, multicloud infrastructures.

The Terraform registry offers support for thousands of publicly available providers. It has support for all major public cloud platforms. And the list keeps growing. You can find a full list of the latest available supported providers here:

`https://registry.Terraform.io/browse/providers`

This ever-growing list of providers makes Terraform even more popular, and it is increasingly making sense for many organizations to adopt Terraform capabilities in their infrastructure management.

Application Infrastructure Orchestration, Scaling, and Monitoring

When we talk about applications, it is the tiering of applications that separates them from the other infrastructure deployments. The installation and deployment of applications needs to be sequentially arranged. For

example, when we are deploying applications, the first database tier should be deployed, then the web servers, then caching server, and so on. Terraform can handle these dependencies automatically. So, it can deploy a database tier before provisioning a web server that depends on it. Also, Terraform can efficiently release, scale, and monitor the infrastructure of multitier applications. A multitier application lets you scale application components independently and provides a separation of concern.

Terraform has support to automate the monitoring of the application infrastructure with the Datadog provider. When you want to deploy Nginx applications to a Kubernetes cluster, with help from Terraform, you can also orchestrate the installation of a Datadog agent across the cluster resources. This Datadog agent reports the cluster health to the Datadog dashboard.

The Terraform capabilities can also be leveraged for blue-green and canary deployments. For blue-green deployments, for feature toggles, you can define a Terraform configuration with a list of potential deployment strategies and in parallel conduct canary tests and incrementally promote the green environment. Hence, the orchestration is made a lot simpler with the presence of infrastructure as a code.

Self-Service Model

As we discussed in previous sections, large organizations have a central operations team where they may be working on repetitive infrastructure service requests. With help from Terraform, they can build a self-serving infrastructure model that lets their customers manage their own infrastructure independently. Terraform allows the creation of modules to replicate the same standards across the different deployments, thus allowing different teams to effectively deploy services in compliance with the respective organization's standards. One such integration we discussed in the previous section is the integration with ServiceNow, i.e., a ticket

tool that can act as a self-service interface for organizations looking to implement automation and offload the repetitive tasks to Terraform and make their support group more productive.

Policy Compliance and Management

Policies are the guidelines that the Terraform cloud applies to Terraform runs. Policies may be used to ensure that the Terraform plan conforms with security guidelines and best practices. Terraform can assist you in enforcing regulations around the resources that teams can provide and utilize. Ticket-based review processes can be a barrier in development. Sentinel, a policy-as-code framework, can be used instead to automatically enforce compliance and governance requirements before Terraform performs infrastructure modifications. Terraform enterprise and Terraform cloud versions both support Sentinel policies.

Sentinel and Open Policy Agent (OPA) are two policy-as-code frameworks you can use to create logic-based, fine-grained policies. Policies might be advisory notices or strict restrictions that stop Terraform from provisioning infrastructure, depending on the settings.

Software-Defined Networking

Software-defined networks (SDNs) and Terraform can work together to autonomously configure the network to meet the requirements of the applications using it. By doing this, you may switch from a ticket-based approach to an automated one and speed up deployment.

These use cases show the rich capabilities of the technology that when leveraged smartly can bring in significant benefits for infrastructure management.

Terraform Integration with Configuration Management Tools

Configuration management tools help administrators maintain system consistency. They ensure that new machines, software packages, and updates are installed and configured according to the desired state. This helps to maintain consistency across various IT systems and sites. Popular configuration management tools include SaltStack, Chef, Puppet, and Ansible.

Most configuration management tools support popular operating systems like Linux and Windows where the system is controlled by the remote server centrally. In the case of SaltStack, there are different minions that get installed on respective systems and are centrally controlled by the master server. IT administrators can configure the desired state on the SaltStack master, which can enforce the configuration on the minions. See Figure 6-4.

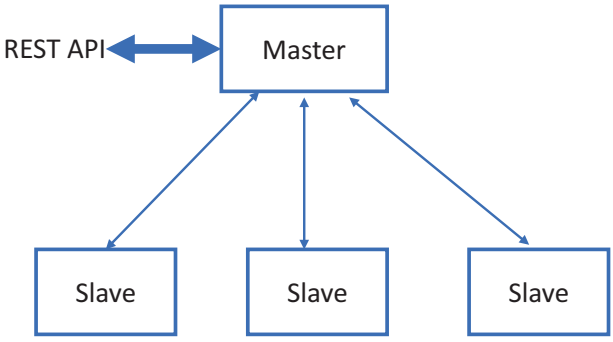


Figure 6-4. *Sample model with configuration management tools*

In Figure 6-4, slaves are agents (minions) that get installed in different systems running on Linux, Windows, etc. These slaves register themselves with a central server (called as *master*) responsible for configuring and running the same desired state on the respective slaves. Modern-day

master configurations support REST APIs, where configuration and management can be done through a RESTful API, which makes it easy to programmatically manage the master-slave integration.

SaltStack integration with Terraform is a value-add for IT administrators because they can automate infrastructure operations from end to end. Terraform controlling the automation at the platform infrastructure and SaltStack controlling automation at the systems level create the complete combination needed for typical infrastructure automation use cases. Their integration provides real value-add in doing end-to-end automation.

Terraform has different integration options for such configuration tools. We can employ Terraform to install these configuration agents every time we are creating a new VM. Furthermore, to complete the integration, we can also define ways to register the newly installed agents (minions) and accept it automatically on the master side to complete the registration process of these agents. Let's look at some sample code for such integration; the following are a few examples for your reference.

Agent Installation with Terraform VM Creation

Terraform offers a section called `locals` where you can define the `start_up_command` to run after the new resource deployment. To install SaltStack minions, you need to supply the deployment executable at the time of the VM deployment process. There are multiple ways to provide the executable for the minion installation. Two such examples in the Terraform configuration file as shown here:

- The agent executable is already present at a specified location in the VM template image, and we can invoke the executable from that specified location present locally on the VM image template. This is the most common method of invoking the installation of any agent with Terraform. The following is the sample code in the Terraform `main.tf` file for such an install:

```
...

...

Locals {

        Start_up_command = [

                                "powershell.exe
                                -command \"<File
                                drive location>\
                                sample.exe\""]

Resource "vsphere_virtual_machine" "xxx" {

...

...

...

Customize {

        Windows_options {

                        Computer_name = "xyz"

                        ...

                        ...

                        Run_once_command_list = local.
                        start_up_command

                        }

                }

        }
```

- The second method could be that the agent executables are placed in a central repository. The location should be easily accessible from the resource we are deploying. Then the desired agent executables are downloaded after IP assignment to the new VM under consideration. This implementation is a little complicated because here we would need to write the logic to first download the agent executable and then invoke the agent installation. The following is the sample code for reference:

```
...
...
locals {
  start_up_command = [
    "powershell.exe -command \"'$webclient.
    DownloadFile(\\\"https://<repo>/sample.exe\\\",
    \\\"$env:TEMP\\minion.exe\\\")' \"",
  "powershell.exe -command \"$env:TEMP\\minion.exe\""]
}
Resource "vsphere_virtual_machine" "xxx" {
...
...
...
Customize {
        Windows_options {
                        Computer_name = "xyz"
                        ...
                        ...
                        Run_once_command_list = local.
                        start_up_command
                        }
                }
        }
```

Please note that the SaltStack minion requires the master IP to be present when we invoke the minion installation. The previous examples are sample representations of invoking an executable file with different methods. For SaltStack minion installation particularly, we can always tweak the method; for example, instead of an executable download, we can have a script created to first download the minion and then install it by presenting the master IP in the previous code.

Operational Uses Cases for VM Management

When we talk about IT operations, there are many tasks that are repetitive in nature. IT infrastructure administrators who support compute, storage, and applications get the following sample requests on a day-to-day basis:

- Create a virtual server
- Decommission a virtual server
- Change the size of a virtual server
- Add a new disk to a virtual server
- Expand an existing disk on a virtual server
- Add a network interface to a virtual server

There are several infrastructure aspects where there are routine requests related to network, storage, compute, application, etc. Here we are keeping the focus simple and going to discuss only compute-related use cases so you can understand how Terraform can help bring automation to these common use cases.

Let's understand these use cases a bit more in detail.

Virtual Server: Create

This is the most common task that administrators must deal with on a daily basis. The creation of a virtual machine is at times very complex because administrators have to deal with multiple dependencies, follow a lot of post-provisioning processes, install some tools, configure machines, etc. However, with help from infrastructure automation tools such as Terraform and configuration management tools (e.g., SaltStack), end-to-end provisioning can be automated without manual intervention from infrastructure admins. Figure 6-5 shows a sample workflow.

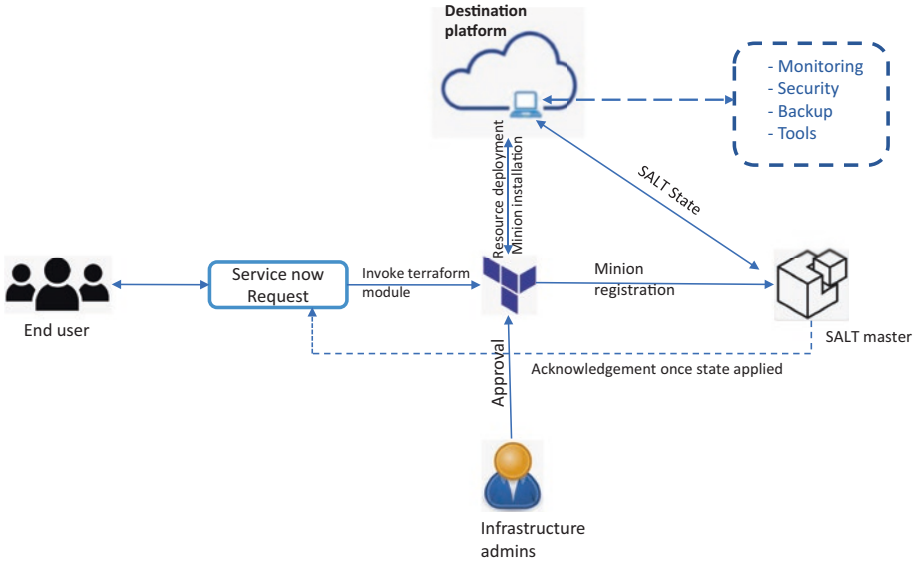


Figure 6-5. *VM provisioning sample automated workflow*

In Figure 6-5, the VM provisioning workflow has the following main steps:

1. Users who need a VM for their application can invoke a catalog in ServiceNow to create a VM. A catalog provides self-service to the end users. They just need to provide detailed information as per their VM requirements. These requirements can be environment specific, but typically they are VM size, storage, and network requirements. Users can also select a backup and monitoring policy that they need to configure. Also, they can select security and firewall ports that need to be allowed on the new resource for its interaction with other infrastructure deployments they may have. All such requirements are gathered and submitted by end users into the service request.

2. Once the requirements are submitted, they can then be mapped to the Terraform module for the resource deployment on the destination platform. These platforms can be any that are supported by Terraform.

3. Terraform can run plans and seek approvals from the infrastructure administrators. Administrators can verify the desired capacity, cost, and VM requirements and can approve the provisioning with Terraform. The Terraform enterprise and Terraform cloud versions offer the approval workflows integration by infrastructure admins. Please note this is an optional step, because we can configure requests in Terraform to "auto-approve," and in that

case Terraform would directly deploy the resource on the destination platform. Approvals can be performed in ServiceNow as well.

4. Terraform can invoke the SaltStack minion installation on the VM. This installation can be done via Terraform provisioners. SaltStack is required for the configuration management of the VM. Please note that there can be multiple ways for doing the configuration management such as applying VM policies in the cloud, etc. Having a dedicated configuration management tool such as SaltStack can make life easier for administrators to manage the complete life cycle of the resource.

5. Terraform can register the minion with the SaltStack master by passing the minion keys and invoking the accept request for that minion on the master server. We will explain this in more detail during the hands-on exercise of this chapter.

6. Once the SaltStack minion is accepted, the SaltStack master can trigger the state file for the VM. Meaning, administrators can define the SaltStack state to apply common organization policies to all the resources deployed in their environment. An example of such a state configuration could be that all VMs require a domain join; other examples include applying the latest patches, imposing security standards, using a backup agent, etc. SaltStack state can make sure such tools are properly installed on the new VM and are deployed and are configured correctly.

7. Once the SaltStack state is successfully applied, the SaltStack master can provide acknowledgment to ServiceNow that the VM is now complying with the desired state and is ready for end-user consumption.

8. ServiceNow then can confirm to the end user and can complete the VM provision request workflow.

This workflow can help automate the day-to-day VM provision requests by end users. It offloads quite a lot of burden from the administrators, and they can focus on other improvements in their infrastructure rather than doing the same tasks every day.

Virtual Server: Decommission

Decommissioning a virtual machine is another daunting task that at times is very time-consuming and redundant. This is because administrators need to follow the proper process to decommission a VM. The following are sample tasks needed in any typical IT environment:

1. Remove the server from monitoring.

2. Take a final backup of the server as per the desired retention period.

3. Delete the firewall rules that may be associated with the VM IP address when it was running.

4. Free up the IP address.

5. Delete all the associated resources with that virtual machine.

Decommissioning is an important part of the life-cycle management of a resource. This process can be automated from end to end using Terraform and Terraform provisioners. If the VM is not already under

Terraform management, with the help of reverse engineering, we can bring the VM under Terraform management first and then leverage the automation and integration capabilities of Terraform.

Once a VM is under Terraform management, then Terraform provisioners can be invoked with "decommission" requests and can play the associated scripts to perform activities such as removing servers from monitoring, invoking a final backup, etc. Terraform can deal with the infrastructure portion of it to make sure all the resources associated with the VM are removed completely once the provisioners complete their job. Figure 6-6 shows the sample VM decommission workflow.

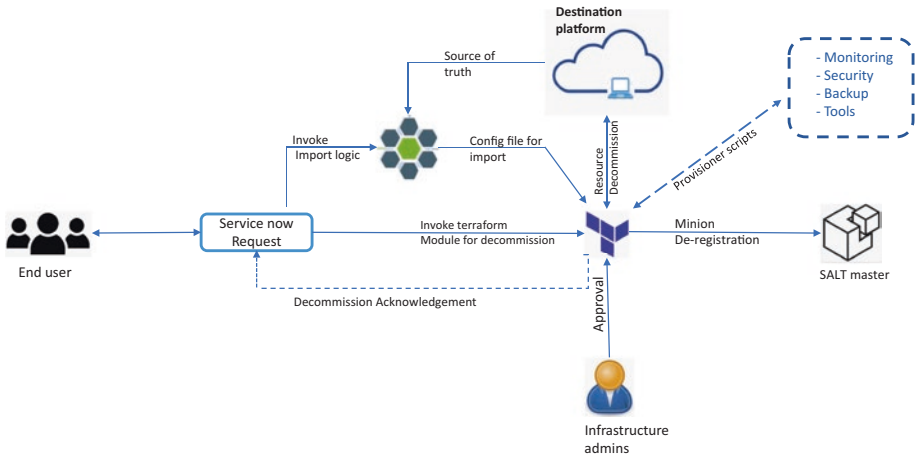


Figure 6-6. *VM decommission sample automated workflow*

Let's discuss the sample decommission workflow.

1. The end user is required to raise a request in their ticketing tool (ServiceNow in our example). The service catalog needs to capture all the required information needed for a decommission job to run. Information includes VM details, final backup retention of the resource, etc.

2. If a VM is not already under Terraform management, then in the workflow you can place the import logic to first neatly import the existing resources into Terraform management. With the import logic, we have built the capabilities to generate the point-in-time configuration files, thus removing the dependency on the Terraform state. Fresh imports allow consistency and efficiency with the Terraform-managed automation. After the successful import, the VM is under Terraform control.

3. After the VM import, the VM decommission module is invoked, taking information from the ServiceNow catalog.

4. Terraform can run plans and seek approvals from the infrastructure administrators, where administrators can verify the decommission request and further approve the decommissioning with Terraform. The Terraform enterprise and Terraform cloud versions offer the approval workflows integration by infrastructure admins. Note that this is an optional step because we can configure requests in Terraform to "auto-approve" as well, and in that case Terraform would directly decommission the resource on the destination platform.

5. Before Terraform can start the actual decommission on the platform, provisioners can be invoked first as per the organization process defined. An example provisioner workflow is as follows:

   a. Provisioners can invoke individual scripts to remove the desired VM from the central monitoring tool.

   b. Take a final backup via the REST API supported by the backup platform.

   c. Release the firewall ports previously configured centrally at the firewall.

6. Once provisioners complete their respective tasks, Terraform decommissions and removes all the associated VM resources and fully deletes them from the destination platform. Finally, after decommissioning, the provisioners can invoke commands to release the IP addresses associated with the VM from the IP management tool.

7. Terraform acknowledges the service request about the status of the decommission request.

8. ServiceNow provides information to the end user, and upon successful completion of the request, it closes the service request raised by the end user.

This workflow can help automate the day-to-day VM decommission requests by end users. It relieves the administrators of a significant amount of work, allowing them to concentrate on other infrastructure upgrades rather than doing the same activities every day.

Virtual Server: Change

If you need to make a change in a virtual server, you need to make changes on a live running server. These changes could be related to VM configuration, adding new resources such as disks or network cards or modifying existing resources such as increasing the existing disk size,

adding more vCPUs or memory, etc. These types of requests are another set of repetitive tasks frequently asked by end users to be performed on the virtual servers. Once again, Terraform provides infrastructure automation capabilities and allows for the full end-to-end automation of such tasks too.

With help from reverse engineering, any existing resource can be imported into the Terraform automation, and thus administrators need not maintain a state with them. Administrators can get the benefit of working directly on the destination platform and can employ Terraform to automate the redundant tasks. Refer to Figure 6-7 for the sample automated workflow.

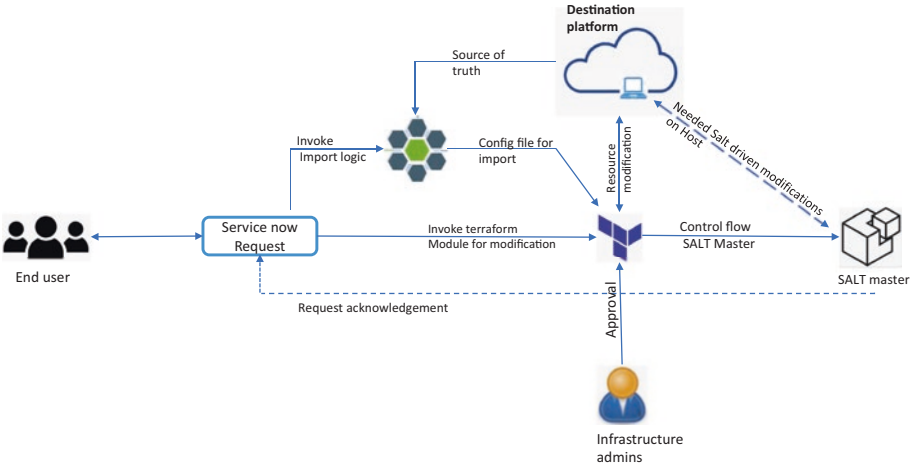


Figure 6-7. *VM modification sample automated workflow*

Let's discuss the sample VM modification workflow.

1. Users who need a VM modification invoke the desired catalog in ServiceNow. The catalog provides self-service to the end users. Here, they need to provide detailed information as per their VM modification requirements. These requirements can

be environment specific, but typically they are VM size, additional storage, and network requirements. All such requirements are gathered and submitted by end users into the service request.

2. If a VM is not already under Terraform management, then in the workflow we can place the import logic to first import the existing resources into Terraform management. With import logic we have built the capabilities to generate the point-in-time configuration files, thus removing the dependency on the Terraform state. Fresh imports allow consistency and efficiency with the Terraform managed automation. After a successful import, the VM is not brought under Terraform control.

3. Once the requirements are submitted, they can then be mapped to the desired Terraform module for the existing resource modification on the destination platform. These platforms can be any that are supported by Terraform.

4. Terraform can run plans and seek approvals from the infrastructure administrators, and administrators can verify the desired capacity, cost, and VM modifications planned and can approve the changes with Terraform. The Terraform enterprise and Terraform cloud versions offer the approval workflows integration by infrastructure admins. Please note that this is an optional step, because we can configure requests in Terraform to "auto-approve" as well, and in that case Terraform would directly deploy the resource on the destination platform.

5. Once the desired modifications are completed by Terraform directly on the infrastructure, then control can be given to the SaltStack master to further the process and make the desired amendments (if needed) directly on the VM host operating system. Since SaltStack is a configuration management tool, it has better control over the changes that are to be performed inside the virtual machines and on the operating system as well as on the applications deployed on the virtual machine.
6. Once the SaltStack master completes the changes desired by the end user, an acknowledgment can be given back by the SaltStack master to the ServiceNow catalog that is keeping track of the entire workflow.
7. ServiceNow confirms the process has completed, and the VM modification request workflow ends.

With help from reverse engineering, the most common use cases can be simply automated. Once the Terraform state is imported, the life-cycle management of the VM can be easily automated with just a small tweak in the configuration file, and dedicated Terraform modules can be created mapped to the ServiceNow catalog to better handle this entire workflow automation.

Terraform Integration with DevOps

DevOps stands for development (Dev) plus operations (Ops). It is a methodology in the IT industry that uses a set of practices and tools to automate the work of software development and in turn shortens the software development life cycle. DevOps complements the agile

methodology of software development where the focus is to produce a functional prototype quickly and add features to the software in an iterative fashion.

The reverse-engineering process we introduced in this book is not complete without explaining the possible integration of Terraform with a popular DevOps tool. In this section, we will briefly explain the possible integration of Terraform with Azure DevOps. See Figure 6-8.

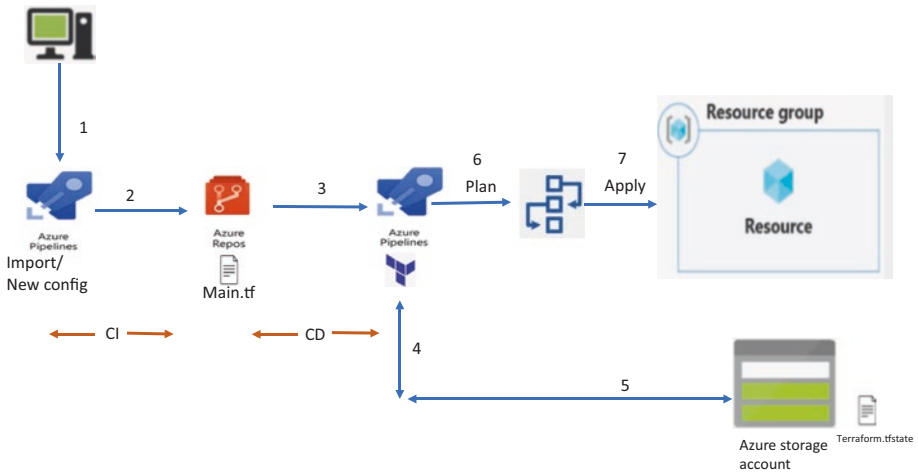


Figure 6-8. *Terraform integration with DevOps*

Before we explain the integration process, let's understand the new terms we are introducing here.

- **Azure Repos**

  For DevOps practices, you need to keep track of changes in the code that are made by different developers working on the software development. Azure Repos is a version control tool that is used to track the changes any developer makes in the code over time. As developers edit the code, you can have the version control system take snapshots of your files,

and these snapshots are saved permanently so you can recall them later if needed. This helps coordinate code changes across the development teams.

For example, the Terraform configuration file (`main.tf`) can reside in Azure Repos, and any changes to this file can be tracked through the version control practices implemented with Azure Repos.

- **Azure Pipelines**

  Azure Pipelines combines continuous delivery and continuous integration to carry out testing on the build of the code and ship that same code onto a specified target. Pipelines has agents running that are capable of executing some code and offers integration with different platforms such as Azure or GCP or VMware. Pipelines support almost all languages including Python, PHP, C/C++, etc.

  With some basic understanding on the terminologies, let us look more closely at what that integration of Terraform and Azure DevOps looks like. Refer to Figure 6-8 to understand the following steps of the DevOps integration.

Step 1: Generate a Config File

The first step of the integration requires a config file to be present. We can generate this config file because of our reverse-engineering process or generate it fresh every time from the user. If we are doing reverse engineering to generate a configuration file, the import logic of reverse engineering can be integrated into the pipeline at this stage, which can run the desired logic (import script) of generating a config file. Since Azure

Pipelines allows integration with different platforms, the import logic can fetch the desired parameters from the platform and populate them into a Terraform configuration file. In the case of a fresh resource creation using Terraform, users can simply use Visual Studio or any other editor to check in the configuration file to the Azure DevOps repository.

Step 2: Check-In the Configuration File to Azure Repository

After our reverse engineering, Azure Pipelines can post the config file generated to an Azure repository. This is the native integration that is offered by Microsoft Azure DevOps. This is also called *continuous integration* (CI) because it enables the automated check-in of configuration files to the Azure DevOps repository.

Step 3: Continuous Delivery Pipeline - For Safe Storing State File

This is the most critical part of the whole DevOps integration process. The Terraform configuration file is pulled from the Azure DevOps repository for continuous deployment. However, we need to store the Terraform state file in a safe place. In this model, we are storing the state file in an Azure storage account. Therefore, the Pipelines code can provide the logic for integrating with the Azure storage account and for uploading the Terraform state file.

Steps 4 and 5: Integrate Azure Pipelines and Azure Storage

Terraform needs to refer to a state file every time it needs to bring in changes to the infrastructure. Azure Pipelines code can refer to the Azure storage account keys, which allows easy access of the storage account

resources. When we want to import an existing resource, Azure Pipelines can import a resource, and the state file that is generated can be uploaded to the storage account for future references.

Step 6: Install Terraform, Initiate the Azure Suite, and Run a Terraform Plan

Azure Pipelines are introduced in step 3. A requirement needs to be defined to install the Terraform executables to the pipeline agents and initiate the SDK for integration with the desired destination platform. In our example, the pipeline agents need to install an Azure kit for the Azure subscription and resource group, i.e., where the resource is to be created or modified. This is an important step because CD agents in the pipeline need to have Terraform executables installed, which can then be employed to run further automated deployment and management of the destination resource. After the successful installation of Azure Pipelines and initiation on the pipeline agents, Azure Pipelines can run a Terraform plan, which refers to a state file stored in the Azure storage account. The plan can find the changes that are required on the infrastructure.

Step 7: Run terraform apply in the Pipeline

The execution steps in the pipeline can apply the delta changes identified in step 6 and post them on the desired resource in the Azure platform. Please note that step 3 to step 7 are the coding or programs written in Azure Pipelines to carry out the desired integration and do the job automatically from end to end. After `terraform apply`, the desired configuration changes can be achieved on the resource.

Thus, the DevOps pipelines and repositories provide a lot of convenience to orchestrate the entire automation workflow with Terraform.

Hands-On Exercise: Terraform Integration with SaltStack and Invocation of SaltStack Install After Terraform Completes VM Provisioning

In this hands-on exercise, we will provide sample code that can be employed for integration with SaltStack. We are installing SaltStack on a Windows virtual machine that we are going to create with Terraform on the VMware platform.

We will use the open-source SaltStack master and manually log in to it to accept the keys for the SaltStack minion we are deploying. Please note that the SaltStack enterprise provides integration support via RESTful APIs, thus making integrations a lot simpler when we want to accept keys without logging in to the SaltStack master.

The following PowerShell script (`Install_Minion.ps1`) is transported to the new VMware VM (via the `local-exec` Terraform provisioner) that we are creating with Terraform. This script downloads the respective minion executables from the SaltStack website and registers with the SaltStack master.

Here is the SaltStack website to download the Windows minion executables:

https://docs.saltproject.io/salt/install-guide/en/latest/topics/install-by-operating-system/windows.html

If needed, you can refer to `Install_Minion.ps1` from the GitHub repository here:

https://github.com/sumitbhatia1986/Terraform-VMware-ReverseEngineering/blob/main/Install_Minion.ps1

Please note the following:

- The assumption for running the `Install_Minion.ps1` script is that your new VMWare VM has connectivity to the Internet. Immediately after the deployment via Terraform, the VM can download the SaltStack minion executables from the SaltStack website.

- Your new VMware VM can talk to the SaltStack master on the required ports. The default ports for connectivity with the SaltStack master are 4505 and 4506.

Let's look at some sample scripts. Please note that these scripts are available on our GitHub repository as well.

https://github.com/sumitbhatia1986/Terraform-VMware-ReverseEngineering

Here is `Install_Minion.ps1` script:

```
$SALTMASTER = 'x.x.x.x' #Required field
$MINIONNAME = $env:COMPUTERNAME
echo $MINIONNAME

# Do not Deploy if Salt is already on the system
if ([System.IO.File]::Exists("c:\salt\bin\python.exe")) {
    Write-output "nothing to do: Salt is already installed"
    #exit 0
}

Write-output "Salt is not installed, Starting Salt
Deployment script"

#This downloads the Salt install to Temp
[Net.ServicePointManager]::SecurityProtocol = [Net.
SecurityProtocolType]::Tls12
[System.Net.ServicePointManager]::ServerCertificateValidation
Callback = { $true }
$webclient = New-Object system.net.webclient
$tempfolder = $env:TEMP
Write-output "Downloading Salt Minion to $tempfolder"

$webclient.DownloadFile("https://repo.saltproject.io/salt/
py3/windows/latest/Salt-Minion-3006.3-Py3-AMD64-Setup.exe",
```

```
"$tempfolder\saltminion.exe") #Downloading minion from SALT
website.

if (![System.IO.File]::Exists("$tempfolder\saltminion.exe")) {
    Write-output "FAILED - Failed to find $tempfolder\
    saltminion.exe , was supposed to download from https://
    repo.saltproject.io/salt/py3/windows/latest/Salt-
    Minion-3006.3-Py3-AMD64-Setup.exe, please investigate,
    exiting script."
    exit 1
}

if ([System.IO.File]::Exists("$tempfolder\saltminion.exe")) {
    Write-output " SALT executable download successful "
}

$MINIONCONF = @"
id: $MINIONNAME
master: $SALTMASTER
tcp_keepalive: True
tcp_keepalive_idle: 60
"@

new-item -Path "C:\ProgramData\Salt Project\Salt\conf"
-itemtype directory
new-item -Path "C:\ProgramData\Salt Project\Salt\conf\minion.d"
-itemtype directory

Set-Content "C:\ProgramData\Salt Project\Salt\conf\minion.d\
minion.conf" $MINIONCONF

Start-Process -FilePath $tempfolder'\saltminion.
exe' -ArgumentList "/S /master=$SALTMASTER /minion-
name=$MINIONNAME", /install-dir="C:\salt" -Wait  #Minion
installation
```

```
New-Item -ItemType SymbolicLink -Path "C:\salt\conf" -Target
"C:\ProgramData\Salt Project\Salt\conf"
New-Item -ItemType SymbolicLink -Path "C:\salt\var" -Target
"C:\ProgramData\Salt Project\Salt\var"

sleep 5
if (-not (Get-Service 'salt-minion'  -ErrorAction
SilentlyContinue)) {
    Write-output "Did not find salt-minion service , sleeping
    30 seonds and retrying."
    sleep 30
    if (-not (Get-Service 'salt-minion'  -ErrorAction
    SilentlyContinue)) {
        Write-output "FAILED - Did not find salt-minion
        service."
        Write-output "removing c:\salt because salt
        installed failed"
        Remove-Item -Path c:\salt -Force -Recurse
        Write-output "FAILED - Salt Minion Failed to Install,
        please investigate, exiting script."
        exit 1
    }
}
if( Get-Service 'salt-minion') {
        write-output "Salt Service is running"
}

Write-output "----------------------------------------"
Write-output "Installation of SaltMinion was successful!"
Write-output "----------------------------------------"
exit 0
```

With an understanding of the `Install_Minion.ps1` script, you will now learn how to integrate the Terraform core and SaltStack.

The following is a snippet of the configuration file (`main.tf`) that is required to invoke the `Install_Minion.ps1` script in Terraform provisioners.

We are assuming the `Install_Minion` script is present in the local machine directory from where you are running the Terraform code.

The following code can be accessed from the GitHub repository with the filename `VMCreationWithTerraformProvisioner.tf`:

https://github.com/sumitbhatia1986/Terraform-VMware-ReverseEngineering/blob/main/VMCreationWithTerraformProvisioner.tf

Here is the code:

```
provider "vsphere" {
vsphere_server = "vcslab01.dc.com"
user = "administrator@vsphere.local"
password = "XXXXX"
#if youn have a self-signed cert
allow_unverified_ssl = true
}

data "vsphere_datacenter" "dc" {
  name = "Lab"
}

data "vsphere_resource_pool" "pool" {
  name          = "vcslab01.dc.com/Resources"
  datacenter_id = "${data.vsphere_datacenter.dc.id}"
}

data "vsphere_datastore" "datastore" {
name = "XYZ"
datacenter_id = "${data.vsphere_datacenter.dc.id}"
}
```

```
data "vsphere_network" "network" {
name = "VM Network"
datacenter_id = "${data.vsphere_datacenter.dc.id}"
}

data "vsphere_virtual_machine" "template" {
  name          = "Win2K16"
  datacenter_id = "${data.vsphere_datacenter.dc.id}"
}

resource "vsphere_virtual_machine" "vm"{
  name             = "Windows2016_terraform"
  resource_pool_id = "${data.vsphere_resource_pool.pool.id}"
  datastore_id     = "${data.vsphere_datastore.datastore1.id}"

  num_cpus = 4
  cpu_hot_add_enabled = "true"
  memory   = 12288
  memory_hot_add_enabled = "true"
  wait_for_guest_net_timeout = 0
  wait_for_guest_ip_timeout = 0
  firmware = "efi"
  guest_id = "${data.vsphere_virtual_machine.template.
              guest_id}"
  scsi_type = "${data.vsphere_virtual_machine.template.
               scsi_type}"

  network_interface {
    network_id = "${data.vsphere_network.network.id}"
    adapter_type = "${data.vsphere_virtual_machine.template.
                    network_interface_types[0]}"
  }

  disk {
```

```
    label            = "disk0.vmdk"
    size             = 120
    eagerly_scrub    = "${data.vsphere_virtual_machine.
                       template.disks.0.eagerly_scrub}"
    thin_provisioned = "${data.vsphere_virtual_machine.
                       template.disks.0.thin_provisioned}"
  }

  clone {
    template_uuid = "${data.vsphere_virtual_machine.
                    template.id}"

    customize {
      windows_options {
        computer_name = "terraform-test"
        admin_password = "XXXX"
        auto_logon = true
        auto_logon_count = 1
        full_name = "Administrator"
      }

      network_interface {
        ipv4_address = "x.x.x.x" #IP of the new VM we are
                       deploying
        ipv4_netmask = 24
      }

    ipv4_gateway = "x.x.x.x"
}
}
```

```
  provisioner "local-exec" {
    command = "copy-item C:\\terraform\\Test\\Install_Minion.ps1
              -destination C:\\ -ToSession (New-PSSession
              -ComputerName x.x.x.x -Credential (new-object
              -typename System.Management.Automation.PSCredential
              -argumentlist local\\Administrator, (convertto-
              securestring -AsPlainText -Force -String XXXX)))"
    interpreter = ["PowerShell", "-Command"]
  }

  provisioner "local-exec" {
    command = "Invoke-Command -ComputerName x.x.x.x -Credential
              (new-object -typename System.Management.
              Automation.PSCredential -argumentlist local\\
              Administrator, (convertto-securestring
              -AsPlainText -Force -String XXXX)) -ScriptBlock
              { C:\\Install_Minion.ps1 }" ## Copying and
              installing from the C drive of new VM we deploying
    interpreter = ["PowerShell", "-Command"]
  }

}
```

When we run `terraform apply`, the execution logs show the following:

- The logs show the VM deployment and assignment of the IP address as specified in the previous configuration file.
- The logs show the execution of the Terraform provisioner (`local-exec`) to copy the script `Install_Minion.ps1` to the new VM that is deployed. The copying of files happens over the WINRM as the Terraform host first creates a session on the new IP address that is assigned on the Windows VM that is deployed.

- Then this `Install_Minion.ps1` script is executed on the VM (the same PowerShell session ID) deployed by the Terraform provisioner (`local-exec`) via the WINRM session.
- The execution of `Install_Minion.ps1` on the new Windows VM installs and downloads the SaltStack minion executable from the SaltStack website and points it to the desired master present in the infrastructure.
- Finally, once the installation of a minion is successful, we will go to the SaltStack master to accept the minion manually and complete the registration.

This completes the workflow for the SaltStack and Terraform integration. Once the SaltStack minion is installed and registered, the SaltStack master has full control over the new Windows VM to perform any configuration management required by administrators on this newly deployed virtual machine.

The following are the logs captured for the Terraform and SaltStack integration deployment for your reference:

```
Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value: yes

vsphere_virtual_machine.vm: Creating...
vsphere_virtual_machine.vm: Still creating... [9m51s elapsed]
vsphere_virtual_machine.vm: Still creating... [10m1s elapsed]
vsphere_virtual_machine.vm: Provisioning with 'local-exec'...
vsphere_virtual_machine.vm (local-exec): Executing:
["PowerShell" "-Command" "copy-item C:\\terraform\\Test\\
```

```
Install_Minion.ps1 -destination C:\\ -ToSession (New-PSSession
-ComputerName x.x.x.x -Cr
edential (new-object -typename System.Management.Automation.
PSCredential -argumentlist local\\Administrator, (convertto-
securestring -AsPlainText -Force -String XXXX)))"]
vsphere_virtual_machine.vm: Still creating... [10m21s elapsed]
vsphere_virtual_machine.vm: Provisioning with 'local-exec'...
vsphere_virtual_machine.vm (local-exec): Executing:
["PowerShell" "-Command" "Invoke-Command -ComputerName
x.x.x.x -Credential (new-object -typename System.Management.
Automation.PSCrede
ntial -argumentlist local\\Administrator, (convertto-
securestring -AsPlainText -Force -String XXXX)) -ScriptBlock {
C:\\Install_Minion.ps1 }"]
vsphere_virtual_machine.vm: Still creating... [10m31s elapsed]

vsphere_virtual_machine.vm (local-exec):     Directory: C:\
Users\Public\Documents

vsphere_virtual_machine.vm (local-exec):
Mode                 LastWriteTime         Length Name
PSComputerName
vsphere_virtual_machine.vm (local-exec):
----                 -------------         ------ ----
--------------

vsphere_virtual_machine.vm (local-exec): terraform-test
vsphere_virtual_machine.vm (local-exec): Salt is not installed,
Starting Salt Deployment script
vsphere_virtual_machine.vm (local-exec): Downloading Salt
Minion to C:\Users\AZUREU~1\AppData\Local\Temp\2 vsphere_
virtual_machine.vm (local-exec): SALT executable download
successful
```

```
vsphere_virtual_machine.vm: Still creating... [10m41s elapsed]
vsphere_virtual_machine.vm (local-exec):     C:\ProgramData\
Salt Project\Salt

vsphere_virtual_machine.vm (local-exec):
Mode                 LastWriteTime         Length Name
PSComputerName
vsphere_virtual_machine.vm (local-exec):
----                 -------------         ------ ----
vsphere_virtual_machine.vm (local-exec): d
-----         7/9/2020   6:30 AM                conf

vsphere_virtual_machine.vm (local-exec):     Directory: C:\
ProgramData\Salt Project\Salt\conf

vsphere_virtual_machine.vm (local-exec):
Mode                 LastWriteTime         Length Name
PSComputerName
vsphere_virtual_machine.vm (local-exec):
----                 -------------         ------ ----
                                                                    --
vsphere_virtual_machine.vm (local-exec): d
-----         7/9/2020   6:30 AM                minion.d
vsphere_virtual_machine.vm: Still creating... [10m51s elapsed]
vsphere_virtual_machine.vm: Still creating... [11m1s elapsed]
vsphere_virtual_machine.vm: Still creating... [11m11s elapsed]
vsphere_virtual_machine.vm (local-exec): Salt Service
is running
vsphere_virtual_machine.vm (local-exec):
----------------------------------------
vsphere_virtual_machine.vm (local-exec): Installation of
SaltMinion was successful!
```

```
vsphere_virtual_machine.vm (local-exec):
----------------------------------------

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
PS C:\terraform>
```

You can see that SaltStack is successful after the VM deployment by Terraform.

We can SSH to the SaltStack master. Here is the command to list the respective minions:

```
root@SALTMasterLab:/# salt-key
Accepted Keys:
Web1
Web2
Denied Keys:
Unaccepted Keys:
Terraform-test    <--- This is the new VM we just deployed.
Rejected Keys:
```

To accept the keys on the SaltStack master, you can run the following command:

```
root@SALTMasterLab:/# salt-key -a Terraform-test
The following keys are going to be accepted:
Unaccepted Keys:
Terraform-test
Proceed? [n/Y] Y
Key for minion Terraform-test accepted.
```

This completes the demonstration of integrating Terraform with SaltStack.

Summary

This chapter on life-cycle management after import explained the key use cases of Terraform for infrastructure automation when used in IT operations. Once we have the import capability built with help from reverse engineering, we want to implement Terraform and automate the monotonous day-to-day infrastructure tasks.

This chapter started by explaining the Terraform integrations where automation support is offered in a variety of IT infrastructure spaces such as security, monitoring, cost management, etc. The chapter then introduced the Terraform provisioners, which are an excellent method for the configuration management of resources using Terraform. The most popular buzzword these days is "ZeroOps," which is now possible with the Terraform IaC and reverse engineering.

The chapter also focused on the most common operational use cases such as VM deployment, VM modifications, and VM decommission. Terraform integration with SaltStack and DevOps was also briefly explained in this chapter since SaltStack and DevOps are of growing interest in today's infrastructure management field. We also explained how to embed the reverse-engineering logic in these operational use cases to make IT administrators' lives easier.

The chapter concluded with the hands-on exercise that demonstrated how to integrate SaltStack (a configuration management tool) and was a complete virtual machine post-provisioning exercise.

CHAPTER 7

Terraform and Import Support on Other Platforms

Our main focus up until this point has been to understand Terraform, reverse engineer it, and effectively import an existing VMware virtual machine into Terraform. Because Terraform is a single platform that can assist in defining our infrastructure as code (IaC) and manage numerous platforms, it is becoming more and more popular.

As an illustration, consider Terraform's automation capabilities across a variety of platforms such as Google Cloud and Microsoft Azure. These platforms encounter the difficulties that we have previously covered; for example, when managing resources that were developed outside of Terraform, Terraform will not know the state of the resources. Manually importing each cloud resource into Terraform is another challenge. Because of the growing popularity of Terraform, developers have been creating solutions in the DevOps world to automatically import existing infrastructure resources into Terraform.

In this chapter, we will talk about tools that are currently under development and show how they can help you leverage Terraform capabilities to set up automation in your infrastructure.

S. Bhatia and C. Gabhane, *Reverse Engineering with Terraform*,
https://doi.org/10.1007/979-8-8688-0074-0_7

We will start the chapter by explaining the challenges of adopting IaC in the public cloud and how you can mitigate those challenges. Next, we will go over the beta feature that Google currently offers, which allows you to import resources that are already running on GCP projects into Terraform. We will also suggest a few more solutions for the Amazon AWS and Microsoft Azure platforms. During the hands-on exercise, we will showcase a tool named `aztfexport` that facilitates the creation of Terraform configuration files for resources that are currently operating on the Microsoft Azure platform.

While this chapter contains references to a number of different import tools, please be aware that some of these tools are still under development. We hope they are made formally available for use in production soon. Still, the ideas in this chapter should enable automatic Terraform imports and assist you in reverse engineering across a variety of supported platforms.

Overview of Challenges with a Public Cloud

Microsoft Azure, Amazon AWS, Google GCP, etc., are the major public cloud providers. These cloud providers offer tools to define IaC. IaC tooling allows business infrastructure to be repeatable, understandable, and programmatic. Examples of such IaC-supported tools for the cloud are ARM templates, Ansible, Terraform, etc.

As covered in this book, Terraform can integrate with different platforms, has an easy-to-understand configuration file, and can help with management. But as Terraform has challenges, there are challenges with the cloud platforms. Let's discuss them briefly now.

Learning Curve

Even if you are familiar with Terraform and also on the public cloud platform, it is still challenging to determine how to author a Terraform

configuration file manually that reflects the exact infrastructure you want to manage. Creating a configuration file is an iterative process where you define some configuration parameters, apply them, and then verify them to see that they are producing the desired configuration on the public cloud portal. Overall, the process of generating a configuration file is time-consuming and complex. It would be great if we could just create resources on the platform directly and then these cloud platforms would allow us to import them into Terraform. That way, administrators do not have to spend time understanding each configuration parameter that they want to implement.

Already Provisioned Resources Outside Terraform

Another challenge for VMware that we discussed is when a resource was provisioned outside of Terraform and administrators do not have a record of the exact settings of the resource. When organizations adopt IaC tools, they want to implement IaC automation across infrastructures, and even for minor changes, the infrastructure requires a fully-fledged configuration file. Without a clean configuration file, changing the infrastructure is risky. Therefore, without importing a resource into Terraform, managing a cloud resource is difficult if it is created outside Terraform.

Cloud Preview Functionality

Cloud providers are continually adding new features to make the life of cloud administrators easier. These new features are released in phases, starting with private, then public preview, and, finally, general availability (GA). Terraform does not usually support these cloud features until the final GA stage. With the automated generation of configuration files, whenever a new feature makes it to the GA stage, administrators can

easily adapt the feature to their configuration file if they are able to import existing resources. Thus, this allows administrators to implement IaC workarounds and make full use of the tools without worrying about new features and respective continuous adoptions they would need to do with the tool.

Escape Hatch

Cloud providers have a vast offering of services and features that are continually expanding and changing. Terraform supports the majority of these services and features. However, when you need to manage a feature or service that is not yet supported, administrators end up having to fall back to imperfect workarounds, or "escape hatches," that do not take advantage of all the good features you expect from Terraform. This is not desired, but there are mechanisms available that allow the generation of configuration files that match the infrastructure feature. With automated import support for cloud platforms, community developers take care of offering import support for these escape hatches.

Removal of Escape Hatches

Escape hatches allow temporary workarounds. But when the "official" support is added to Terraform, administrators want to remove these escape hatches in the implementation. Without the presence of a mechanism that generates the configuration file automatically, administrators would have to destroy the infrastructure and redeploy it. This is not an efficient way to manage things; having a tool that allows for an automated import is appreciated by infrastructure admins.

Now that you understand the challenges with the public cloud and how reverse engineering can help mitigate them, we will explain some of the recent releases by the major cloud providers or open-source communities that allow the automated generation of the Terraform

supported configuration file through a small utility. In other words, the following utilities mitigate all the challenges previously discussed. These support utilities are very helpful in properly adopting Terraform in any environment.

Google Cloud Utility for Terraform Import

With the increasing attractiveness of Terraform in the DevOps world, there has been increase in Terraform adoption as well on Google. At the same time, different platforms including the Google Cloud are coming up with native tools that support the automated import of an existing resource into Terraform. One such utility recently announced by Google is called `gcloud beta resource-config`. This readymade tool already has reverse-engineering logic built in, which in turn talks to the Google Cloud Platform and fetches the required Terraform configuration of the resources running inside a GCP project. This allows IT administrators to use Terraform in their day-to-day IT automation.

Here is a link to the detailed GCP documentation:

`https://cloud.google.com/sdk/gcloud/reference/beta/resource-config`

Note that this utility is currently in beta and can be changed by Google without prior notice.

The utility allows you to bulk export project resources into a `*.tf` file, which can be readily used for further import via the Terraform tool. Figure 7-1 shows a sample import workflow with GCP projects.

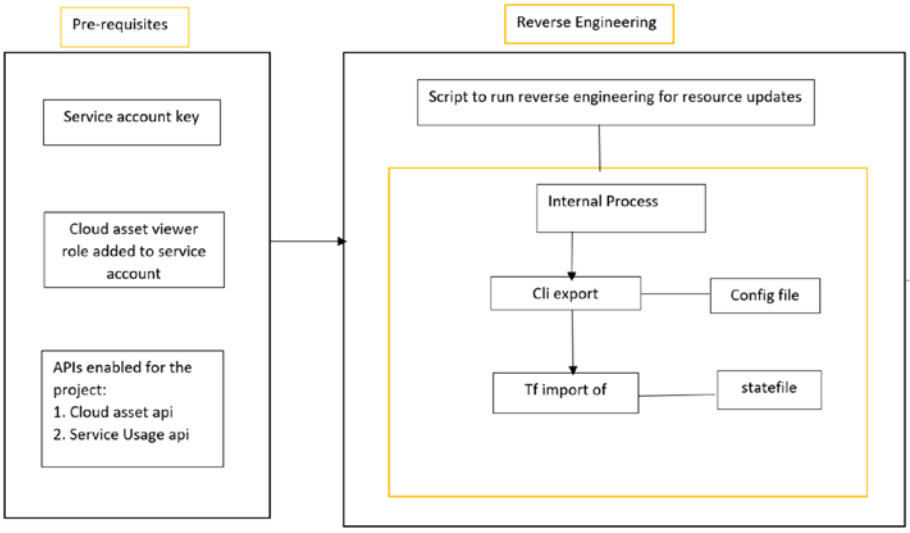


Figure 7-1. *Sample Terraform import workflow with GCP*

The following are the high-level steps of importing an existing GCP resource with Terraform.

These are the prerequisites:

1. Get the service account key and activate it in the SDK for a project in the GCP.
2. Make sure the service account has a Cloud Asset Viewer role.
3. Enable the following APIs with your GCP project:

   cloud asset api

   service usage api

Here are the importing steps:

1. Define the execution environment and set the GCP project, region, and zone.
2. Here is the command for the bulk export of the Terraform configuration files for resources in a GCP project:
   ```
   gcloud beta resource-config bulk-export
   --project=<Project Name> --path=./instances
   --resource-format=Terraform --resource-
   types=storage.cnrm.cloud.google.com/
   ComputeInstance -q
   ```
3. In this bulk export, copy the required resource `*.tf` file to another directory where you want to run `import`.
4. Run `terraform init`.
5. Import your required resource with this command:
   ```
   terraform import google_compute_instance.
   <VMname> <ProjectName>/<Region>/<VMName>
   ```
6. Run `terraform plan` and verify that no changes are suggested. If no changes are suggested, that signifies a successful import.

The steps defined here are a brief representation of the import logic. Obviously, the tool supports different command-line options, which can be referred to by visiting the link we provided.

As you learned with the VMware import in earlier chapters, a successful import facilitates the automation of many redundant tasks and completes the dots of the infrastructure automation. After a successful import of a GCP resource, all other common use cases and infrastructure integrations can be similarly leveraged, which reduces the manual effort of IT administrators.

Microsoft Azure Cloud Utility for Terraform Import

To allow the automated generation of Terraform configuration files, there is a community of developers working on doing reverse engineering with the Azure platform to create configuration files and import existing resources into Terraform. There are open-source and Microsoft-supported tools available that facilitate this job for administrators. The following are a few tools that allow the easy import of existing resources into Terraform:

- **Azure Terrafy (aztfy)**

  With Azure Terrafy, you can quickly and easily turn the existing Azure infrastructure into Terraform HCL and import it to a Terraform state. After you have completed importing your infrastructure, you can manage it with your standard IaC processes. Learn more here:

  https://techcommunity.microsoft.com/t5/azure-tools-blog/announcing-azure-terrafy-and-azapi-Terraform-provider-previews/ba-p/3270937

  Here is the link to the GitHub source code link of the tools:

  https://github.com/Azure/Terraform

  Here is a sample CLI command that imports a resource by the resource ID:

```
aztfy resource <resource id>`
Test:~$ aztfy resource /subscriptions/XXXXX-XXXX-
XXXX-XXXX-XXXXXXX/resourcegroups/test_group/
providers/Microsoft.Compute/virtualMachines/test
Test:~$ ls
main.tf provider.tf Terraform.tfstate
```

- **Azure Export:**

  Another tool that is offered by Microsoft is called Azure Export for Terraform. This tool is designed to help reduce friction when translating between Azure and Terraform concepts. For scenarios related to escape hatches or the removal of escape hatches, Azure Export for Terraform allows the use of Azure preview features with Terraform. To learn more about the tool, please visit the following link:

  https://learn.microsoft.com/en-us/azure/developer/Terraform/azure-export-for-Terraform/export-Terraform-overview

  Here is the Azure Export GitHub page:

  https://github.com/Azure/aztfexport/releases

Azure Export offers the following benefits:

Easy adoption of Terraform: Azure Export for Terraform allows users to easily import existing resources into Terraform using a single command. Users can first export the resource configuration file in HCL format and then can further import it into Terraform to generate a fresh Terraform state, thus making it easy to import any existing Azure resource into Terraform.

Export user-specified sets: Azure Export for Terraform allows the predetermined scope to export. Users can define scope that is granular as a single resource, a resource group, and its nested resources or an entire subscription. The export provides the configuration file in HCL format that can be readily used to import a resource with Terraform.

Inspection of pre-existing infrastructure: The Azure Export utility offers a read-only export with the option to expose all the configurable resource properties. Thus, it is helpful for learning about a newly released Azure resource or investigating any issue in production.

The links given contain information about installing the binaries and making use of the source to help generate a Terraform-supported configuration file of a desired resource.

Here is a sample CLI command to import a resource by the resource ID:

```
aztfexport [command] [option] <scope>
```

The scope can change depending on the command the user wants to run. It depends on whether the user is trying to export a single resource or a resource group. See Table 7-1.

Table 7-1. *AZTFEXPORT Command Options*

Task	Explanation	Command
Export a single resource.	To export a single resource, specify the Azure resource ID associated with the resource.	`aztfexport resource [option] <resource id>`
Export a resource group.	To export a resource group (and its nested resources), specify the resource group name, not the ID.	`aztfexport resource-group [option] <resource group name>`
Export using a query.	The tool supports exporting with an Azure Resource Graph query.	`aztfexport query [option] <ARG where predicate>`

By default, Azure Export for Terraform collects the telemetry data to improve the user experience. Azure Export for Terraform does not collect any private or personal data. However, you can still easily disable this process with the following command:

```
aztfexport config set telemetry_enabled false
```

We are going to show how to use Azure Export for Terraform in the hands-on exercise of this chapter.

Amazon AWS Cloud Utility for Terraform Import

There are open-source tools available that allow reverse engineering on the AWS platform as well. One example of such a tool is Terraformer. This is a tool developed by Waze, a subsidiary of Google; however, it is not an official product of Google. Terraformer is an open-source tool that can be modified and used across all major platforms including Amazon, Google, AWS, IBM Cloud, and Alibaba Cloud.

Terraformer uses Terraform providers and is designed to easily support newly added resources. To upgrade resources with new fields, all you need to do is upgrade the relevant Terraform providers.

Here is the link to the GitHub source code for the tool:
https://github.com/GoogleCloudPlatform/Terraformer
Here are the steps:

1. Install Terraformer following the instructions at the provided GitHub link.
2. Clone the GitHub repository and go to Terraformer.
3. Build the modules with the provider you choose.
4. Run the `import` command to start importing.

Here is the sample CLI command to import all EC2 instances in region us-east-1 for AWS:

```
Terraformer import aws --resources=ec2_instance --regions=ap-
southeast-2
```

Terraformer imports the existing AWS resource directly into Terraform and creates the state file.

Here are the capabilities of the Terraformer open-source tool:

- It can generate the TF/JSON + tfstate file from the existing infrastructure for the supported objects of each resource on the respective platform.
- The state file it generates can be uploaded to the cloud bucket directly.
- The import is supported by the resource name and its type.
- Users can save TF/JSON files using a custom folder tree pattern.

Note If you don't specify the region, it will import the resources from the default region.

```
test: $ Terraformer import aws --resources=ec2_instance
--regions=ap-southeast-2
2022/09/20 16:56:26 aws importing region ap-southeast-2
2022/09/20 16:56:29 aws importing... ec2_instance
2022/09/20 16:56:30 aws done importing ec2_instance
2022/09/20 16:56:30 Number of resources for service ec2_
instance: 0
2022/09/20 16:56:30 aws Connecting....
```

```
2022/09/20 16:56:30 aws save ec2_instance
2022/09/20 16:56:30 aws save tfstate for ec2_instance
test: $ cd generated/aws/ec2_instance/
test: $ ls
provider.tf Terraform.tfstate
```

After Terraformer imports the existing resource, you can run `terraform plan` to confirm that the tool is not suggesting any changes.

Word of Caution

We have provided references to different open-source and native tools; however, please note that you need to pick the right tool for your situation and use a systematic approach to import cloud resources into a Terraform configuration. There may be tweaks needed based on your specific infrastructure needs. While all the tools provide an option to import, they may not necessarily be adequate for your specific needs. Performing `terraform plan` to ensure there are no changes in the infrastructure is an important step in doing the automated import. Although the tools do an impressive job at importing, they may not be mature enough to match the current practices of creating Terraform configuration files specific to your environment.

Following a well-structured approach, putting in some work to restructure configuration files as per best practices, and maintaining the accuracy of the state and configuration files will ensure a successful transition to using Terraform to maintain your cloud resources.

Hands-On Exercise: Using Azure Export for Terraform to Autogenerate a Configuration File and Import an Azure VM

In this hands-on exercise, we will demonstrate how to use Azure Export for Terraform to autogenerate a configuration file for an Azure VM and auto-import an Azure VM into Terraform. Furthermore, we will run `terraform plan` to demonstrate a clean import of an existing resource where Terraform suggests no changes on the platform.

As a prerequisite, you should have the Azure CLI installed and Terraform executables deployed on the `%PATH%` of your local machine where you are doing this exercise.

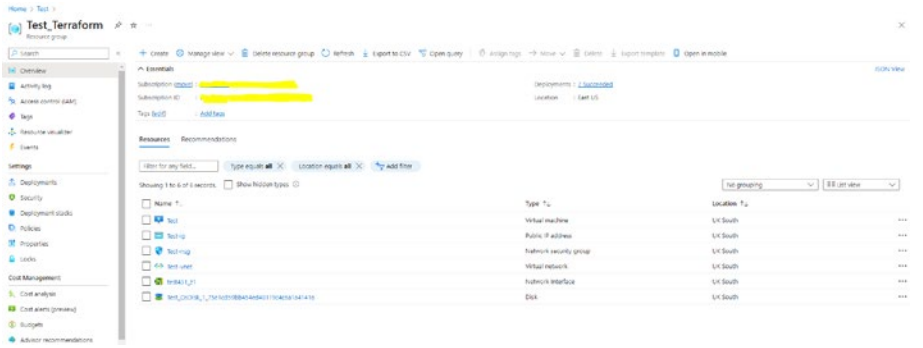
You can find detail information about Azure Export for Terraform here:

https://learn.microsoft.com/en-us/azure/developer/Terraform/azure-export-for-Terraform/export-Terraform-overview

To demonstrate the functionality of Azure Export for Terraform, we have already created a resource group named `Test_Terraform` in an Azure subscription.

Also, we have created a `Test` virtual machine inside the resource group `Test_Terraform`.

Please refer to the following to see the resources present inside the resource group:



The following is the outcome of the command:

```
Az resource list –name <VMname>
PS C:\Users\Desktop> az resource list --name Test
[
  {
    "id": "/subscriptions/XXXXXXXXX/resourceGroups/Test_
    Terraform/providers/Microsoft.Compute/virtualMachines/Test",
    "identity": null,
    "kind": null,
    "location": "uksouth",
    "managedBy": null,
    "name": "Test",
    "plan": {
      "name": "servercore-2019",
      "product": "servercore-2019",
      "promotionCode": null,
      "publisher": "cloud-infrastructure-services",
      "version": null
    },
    "properties": null,
    "resourceGroup": "Test_Terraform",
```

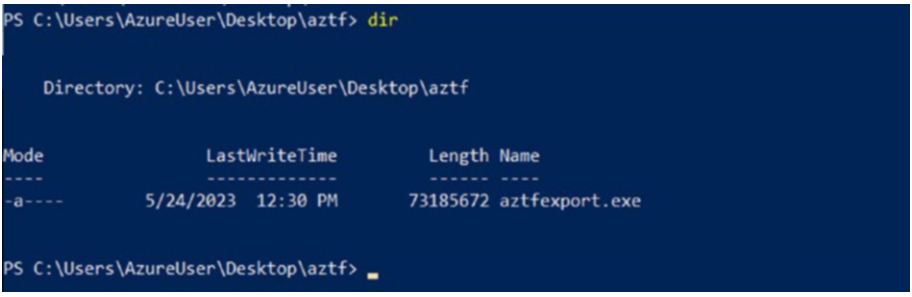
```
    "sku": null,
    "tags": null,
    "type": "Microsoft.Compute/virtualMachines",
    "zones": [
      "1"
    ]
  }
]
PS C:\Users\Desktop>
```

Here is the step-by-step flow of the process:

1. Download the Azure Export for Terraform tool from the following link and extract the `.exe` file to a specified folder:

   https://github.com/Azure/aztfexport/releases

   In this case, we extracted it to the `aztf` folder on the desktop.



2. Now from the same directory, you can run the following command:

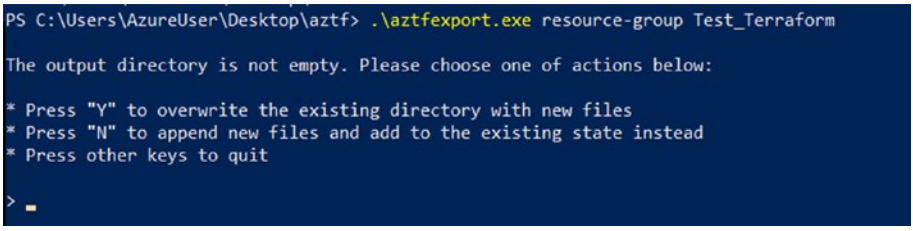
   ```
   .\aztfexport.exe resource-group
   <resource-group name>
   ```

Note Since we are running this command from PowerShell and using `az` commands, you must already be logged in to the desired Azure subscription. You can do so using the command `az login` and setting the subscription with `az account set - -subscription <desired subscription>`.

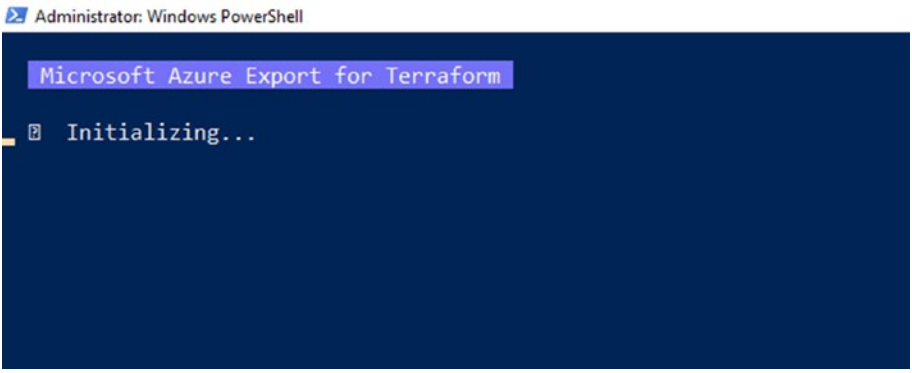
Make sure your `AZ cli` is also up-to-date on your desktop. You can install the latest version by downloading the executables from the following website:

https://learn.microsoft.com/en-us/cli/azure/install-azure-cli-windows?tabs=azure-cli

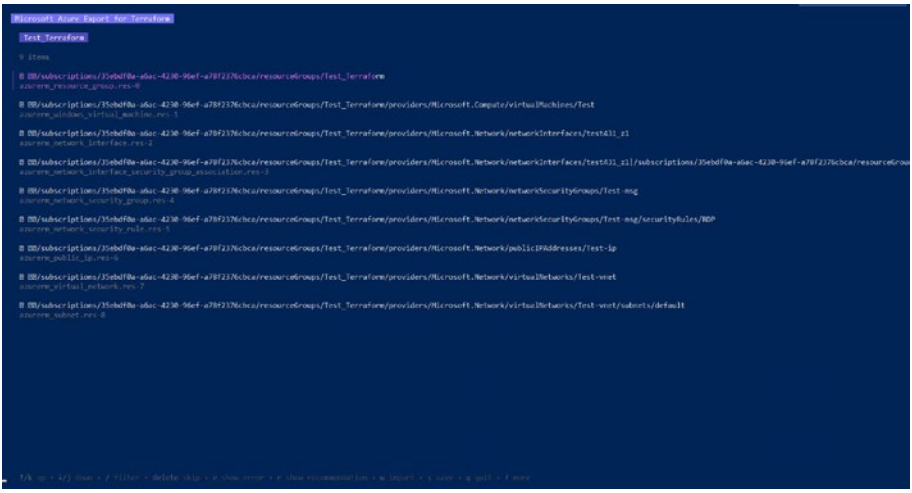
3. After running the previous command, you can select the desired option to proceed.



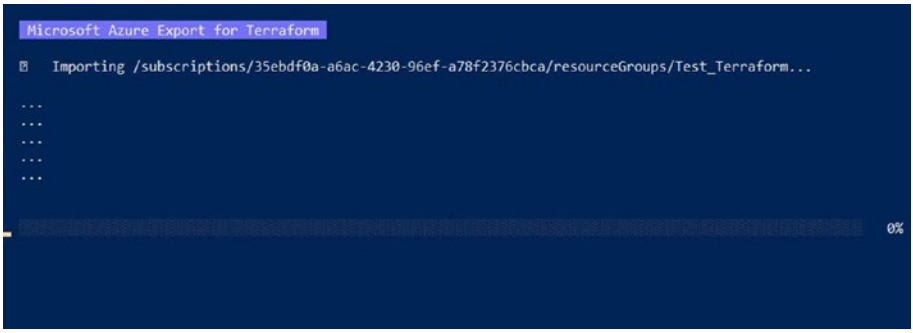
4. It will then start to initialize Microsoft Azure Export for Terraform. It takes a while to initialize.

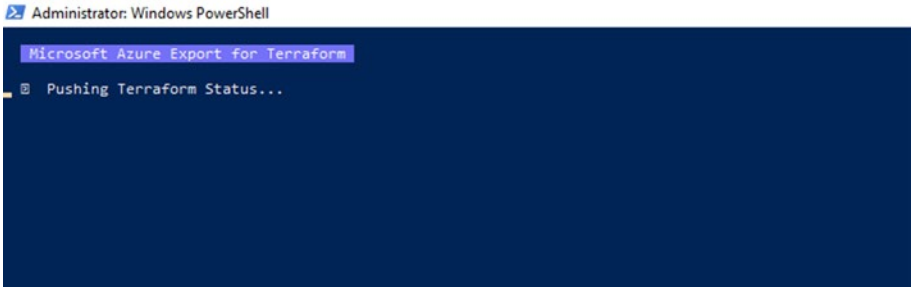


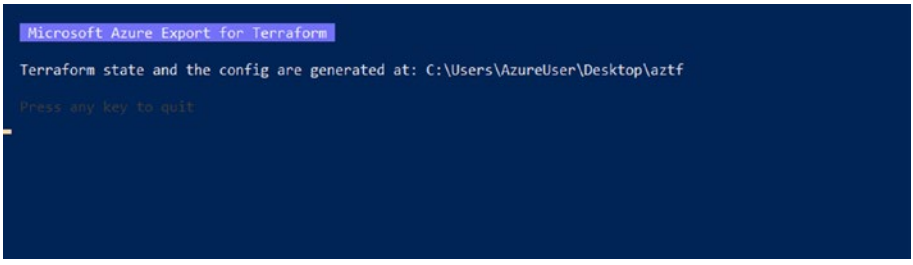
5. After initialization, a list of the resources to be exported is displayed. Each line has an Azure resource ID matched to the corresponding AzureRM resource type. The list of available commands displays at the bottom of the screen. Using one of the commands, scroll to the bottom and verify that the expected Azure resources are properly mapped to their respective Terraform resource types.



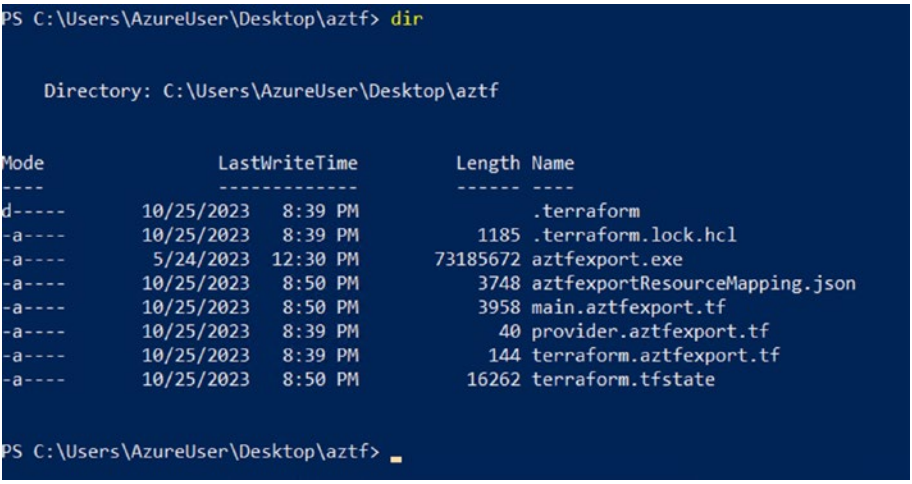
6. Press W to run the auto-import of the Azure resource into Terraform.







7. After the Azure export of Terraform is completed, you will see that the folder is populated with the desired Terraform configuration files.



`Main.aztfexport.tf` is the autogenerated configuration file from the tool, as shown here:

```
resource "azurerm_resource_group" "res-0" {
  location = "eastus"
  name     = "Test_Terraform"
}
resource "azurerm_windows_virtual_machine"
"res-1s" {
  admin_password        = "ignored-as-imported"
  admin_username        = "cloudadmin"
  location              = "uksouth"
  name                  = "Test"
  network_interface_ids = ["/subscriptions/
                          XXXXXX/resourceGroups/
                          Test_Terraform/providers/
                          Microsoft.Network/
                          networkInterfaces/
                          test431_z1"]
```

```
  resource_group_name   = "Test_Terraform"
  size                  = "Standard_DS1_v2"
  zone                  = "1"
  boot_diagnostics {
  }
  os_disk {
    caching              = "ReadWrite"
    storage_account_type = "Premium_LRS"
  }
  plan {
    name      = "servercore-2019"
    product   = "servercore-2019"
    publisher = "cloud-infrastructure-services"
  }
  source_image_reference {
    offer     = "servercore-2019"
    publisher = "cloud-infrastructure-services"
    sku       = "servercore-2019"
    version   = "latest"
  }
  depends_on = [
    azurerm_network_interface.res-2,
  ]
}
resource "azurerm_network_interface" "res-2" {
  enable_accelerated_networking = true
  location                      = "uksouth"
  name                          = "test431_z1"
  resource_group_name           = "Test_Terraform"
  ip_configuration {
    name                          = "ipconfig1"
```

```
    private_ip_address_allocation = "Dynamic"
    public_ip_address_id          = "/subscriptions/
                                    XXXXXX/
                                    resourceGroups/
                                    Test_Terraform/
                                    providers/
                                    Microsoft.Network/
                                    publicIPAddresses/
                                    Test-ip"
    subnet_id                     = "/subscriptions/
                                    XXXXXX/
                                    resourceGroups/
                                    Test_Terraform/
                                    providers/
                                    Microsoft.Network/
                                    virtualNetworks/
                                    Test-vnet/subnets/
                                    default"
  }
  depends_on = [
    azurerm_public_ip.res-6,
    azurerm_subnet.res-8,
  ]
}
resource "azurerm_network_interface_security_group_
association" "res-3" {
  network_interface_id      = "/subscriptions/XXXXXX/
                              resourceGroups/Test_
                              Terraform/providers/
                              Microsoft.Network/
                              networkInterfaces/
                              test431_z1"
```

```
  network_security_group_id = "/subscriptions/XXXXXX/
                              resourceGroups/Test_
                              Terraform/providers/
                              Microsoft.Network/
                              networkSecurityGroups/
                              Test-nsg"
  depends_on = [
    azurerm_network_interface.res-2,
    azurerm_network_security_group.res-4,
  ]
}
resource "azurerm_network_security_group" "res-4" {
  location            = "uksouth"
  name                = "Test-nsg"
  resource_group_name = "Test_Terraform"
  depends_on = [
    azurerm_resource_group.res-0,
  ]
}
resource "azurerm_network_security_rule" "res-5" {
  access                      = "Allow"
  destination_address_prefix  = "*"
  destination_port_range      = "3389"
  direction                   = "Inbound"
  name                        = "RDP"
  network_security_group_name = "Test-nsg"
  priority                    = 300
  protocol                    = "Tcp"
  resource_group_name         = "Test_Terraform"
  source_address_prefix       = "*"
  source_port_range           = "*"
  depends_on = [
```

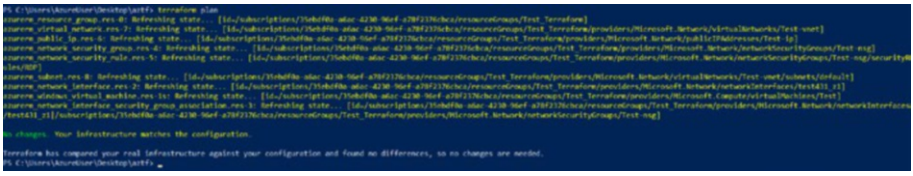
```
    azurerm_network_security_group.res-4,
  ]
}
resource "azurerm_public_ip" "res-6" {
  allocation_method   = "Static"
  location            = "uksouth"
  name                = "Test-ip"
  resource_group_name = "Test_Terraform"
  sku                 = "Standard"
  zones               = ["1"]
  depends_on = [
    azurerm_resource_group.res-0,
  ]
}
resource "azurerm_virtual_network" "res-7" {
  address_space       = ["10.0.0.0/16"]
  location            = "uksouth"
  name                = "Test-vnet"
  resource_group_name = "Test_Terraform"
  depends_on = [
    azurerm_resource_group.res-0,
  ]
}
resource "azurerm_subnet" "res-8" {
  address_prefixes     = ["10.0.0.0/24"]
  name                 = "default"
  resource_group_name  = "Test_Terraform"
  virtual_network_name = "Test-vnet"
  depends_on = [
    azurerm_virtual_network.res-7,
  ]
}
```

This configuration file accurately defines the Azure VM in terms of Terraform IaC.

8. To further validate whether your import is successful, run `terraform init`.

```
PS C:\Users\AzureUser\Desktop\aztf> terraform init

Initializing the backend...

Initializing provider plugins...
- Reusing previous version of hashicorp/azurerm from the dependency lock file
- Using previously-installed hashicorp/azurerm v3.56.0

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
PS C:\Users\AzureUser\Desktop\aztf>
```

9. Run `terraform plan` to verify that no changes are suggested by Terraform.



In the previous example, we demonstrated the autogeneration and import of configuration files into a Terraform state. However, Azure Export for Terraform also offers the generation of Terraform configuration files on its own. The following is a sample command to demonstrate that:

```
C:\Users\AzureUser\Desktop\test\aztfexport.exe resource-group
- -non-interactive - -hcl-only <resource-group name>
```

Users can use the configuration files generated from the previous command and can import the resource manually into Terraform.

Summary

This chapter discussed the Terraform import support of different public cloud platforms. Development communities are increasingly working on developing utilities that offer Terraform import support of pre-existing resources on respective platforms. In this chapter, we mainly talked about the utilities offered by cloud providers such as Google Cloud Platform (GCP), Microsoft Azure, and Amazon AWS.

Also, in the hands-on exercise, we demonstrated how to use Azure Export for Terraform to import a pre-existing Azure virtual machine into Terraform automation. These tools are of tremendous help and make it possible to use Terraform in the popular cloud platforms.

Throughout this book we covered the reverse engineering process and how it provides tremendous benefits for adopting infrastructure automation with our IT infrastructure, which was at once merely a dream for many IT administrators. Reverse engineering a system is surely a challenging task. When practicing it in any deployment, it is worth considering the benefits versus effort in implementing the process. This assessment is important to justify the benefits of reverse engineering. Considering the variety of infrastructure technologies available these days, eventually administrators will need to look for solutions to enable automation in their respective infrastructure.

We hope you enjoyed reading this book as much as we loved writing it. We sincerely hope the explanations and examples provided in this book enable you to think out of the box and help you achieve your personal and professional goals.

Index

S. Bhatia and C. Gabhane, *Reverse Engineering with Terraform*,
https://doi.org/10.1007/979-8-8688-0074-0

E

F

G

H

I

J, K

L

M

N

O

P, Q

R

S

T

U

V, W, X, Y

Z