7 Pitfalls to Avoid with Terraform

Table of Contents

Introduction

Infrastructure automation is the backbone of the modern DevOps ecosystem, which requires frequent and faster deliveries across dynamic platforms. This automation minimizes the human effort required to maintain a tech stack by creating repeatable scripts and functions.

Infrastructure as code (IaC) is one approach that lets DevOps teams configure a system's desired state using declarative specifications. This, in turn, helps organizations provision infrastructure rapidly while allowing for seamless integration between multiple deployment environments.

Infrastructure automation relies on various tools and processes for the automatic provisioning, deployment, and testing of infrastructure. Terraform is one of the most commonly used IT automation tools and provides a one-stop solution for IT infrastructure configuration and management issues.

In this article, we delve into Terraform's benefits as well as common misconfigurations and the best practices for avoiding them. But first, let's briefly review what Terraform is.

What Is Terraform?

Terraform is an open-source, declarative configuration management system (CMS) designed to automate infrastructure provisioning through human-readable code. The platform relies on a state file containing all of the infrastructure's configuration details. You can store this state file on any machine or server that has access to it and use it for both cloud-native or on-premises setups. Along with the capabilities of deploying and changing the configuration of your infrastructure, Terraform also offers development standards like version control and testing.

Terraform leverages a plugin-based architecture that consists of two major components: **Terraform Core** and **Terraform Plugins.**

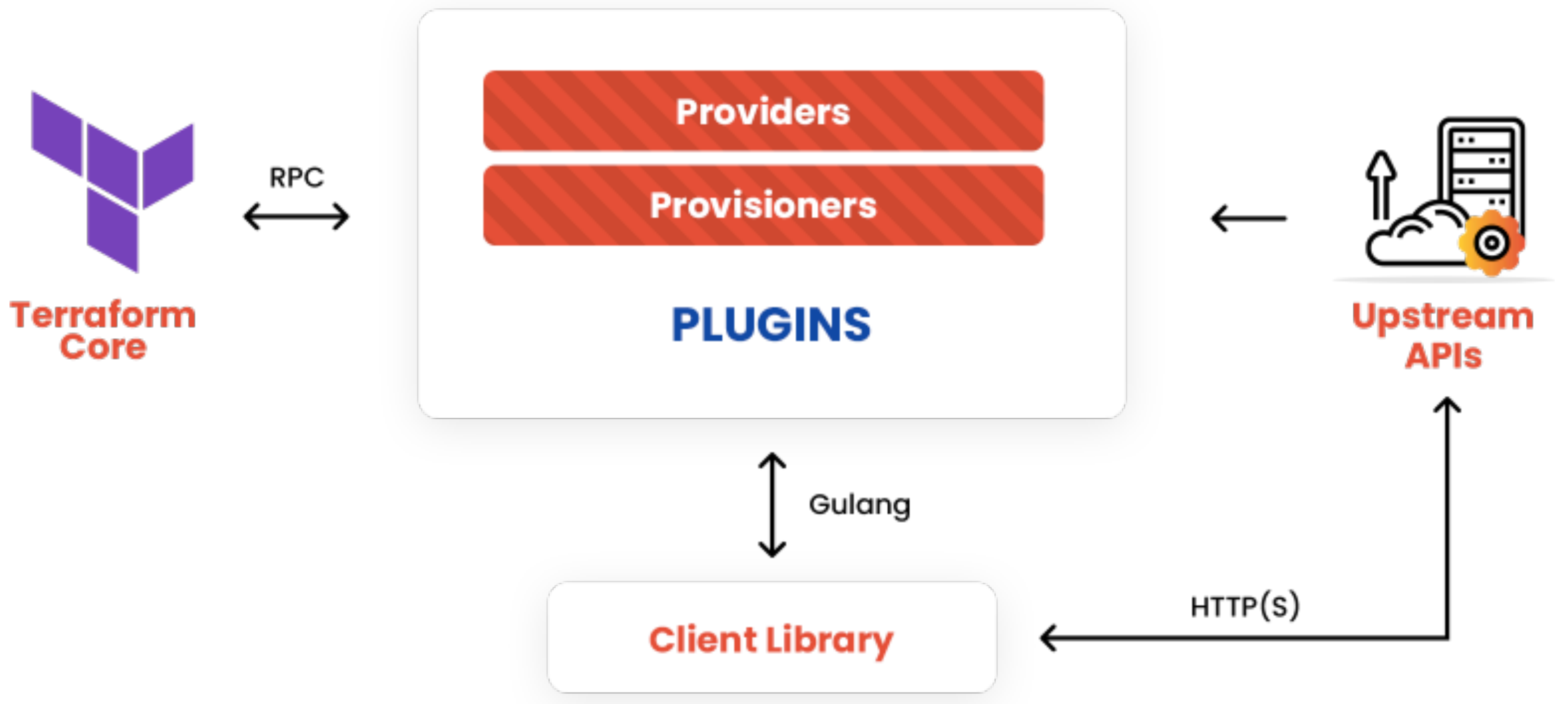


Figure 1: Terraform components (Source: HashiCorp)

- **Core** compares the Terraform configuration file with the Terraform state file to apply configuration changes. To achieve the desired state, Core communicates with Plugins over an RPC.
- **Plugins** consist of **Providers** and **Provisioners** that are executable binaries invoked by Core. Plugins allow you to integrate with various cloud platforms using external APIs.

Benefits of Using Terraform

There are numerous advantages to implementing Terraform, several of which we discuss below.

Developed by HashiCorp

Terraform was developed by HashiCorp, one of the largest known companies offering innovative open-source tools for infrastructure management. Terraform's configuration file is written in a language that was specifically designed for HashiCorp tools, the HashiCorp Configuration Language (HCL). The language is both human and machine-friendly, and although it loosely resembles JSON, its capabilities are far more advanced.

Seamless Multi-Cloud Infrastructure Deployment

Being cloud-agnostic, Terraform allows you to provision infrastructure across different cloud providers dynamically. A single configuration file can manage multiple cloud providers and resolves multi-cloud dependencies as well.

Lower Development Costs

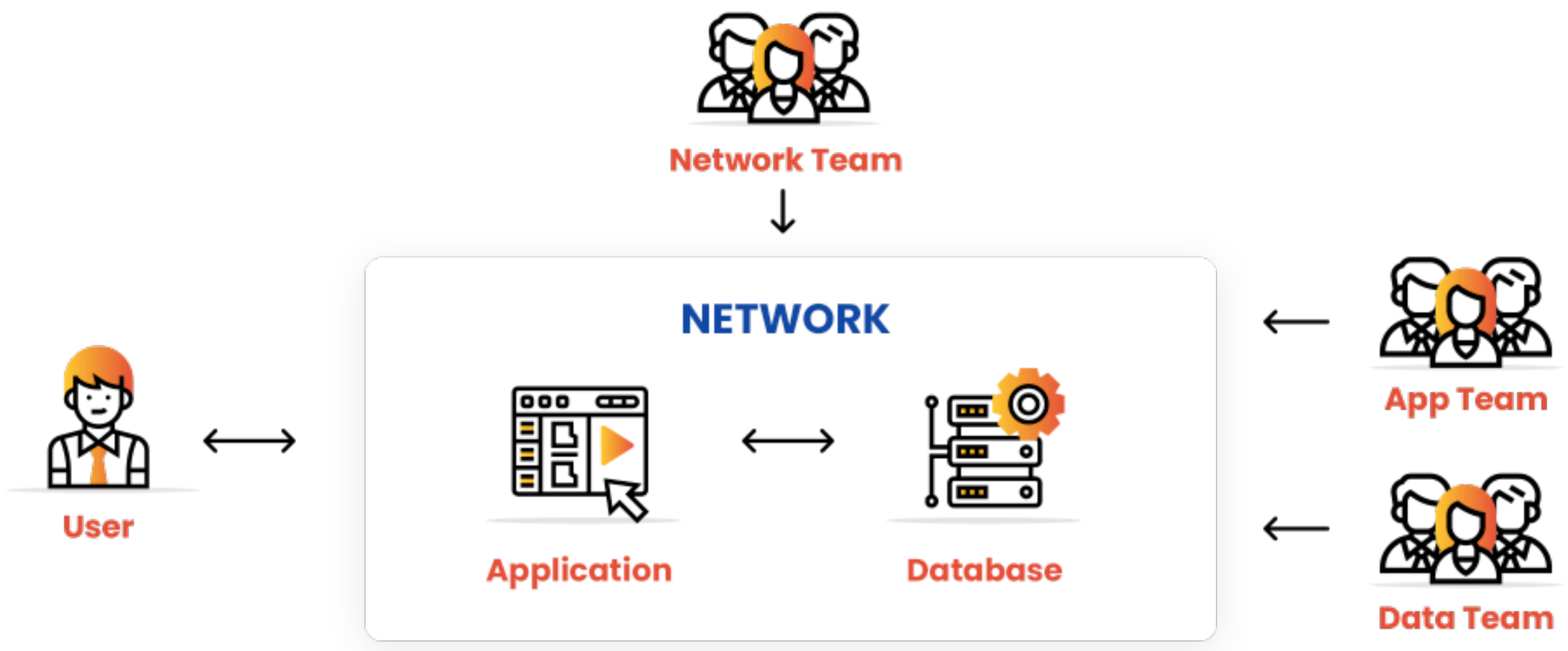
The platform reduces your cost of development by offering on-demand environments. Using API calls and templates, developers can create and decommission environments in just a few easy steps.

Quicker Deployments

Terraform automates the process of infrastructure deployment, making it faster and less error-prone. Additionally, infrastructure on demand helps provision seamless non-prod QA and load testing environments to facilitate rapid testing and delivery.

Facilitates Cross-Functional Collaboration

Terraform supports a DevOps model that facilitates seamless cross-functional collaboration among distributed teams relying on the same infrastructure and processes. This improves visibility and productivity while reducing root cause identification and resolution time.



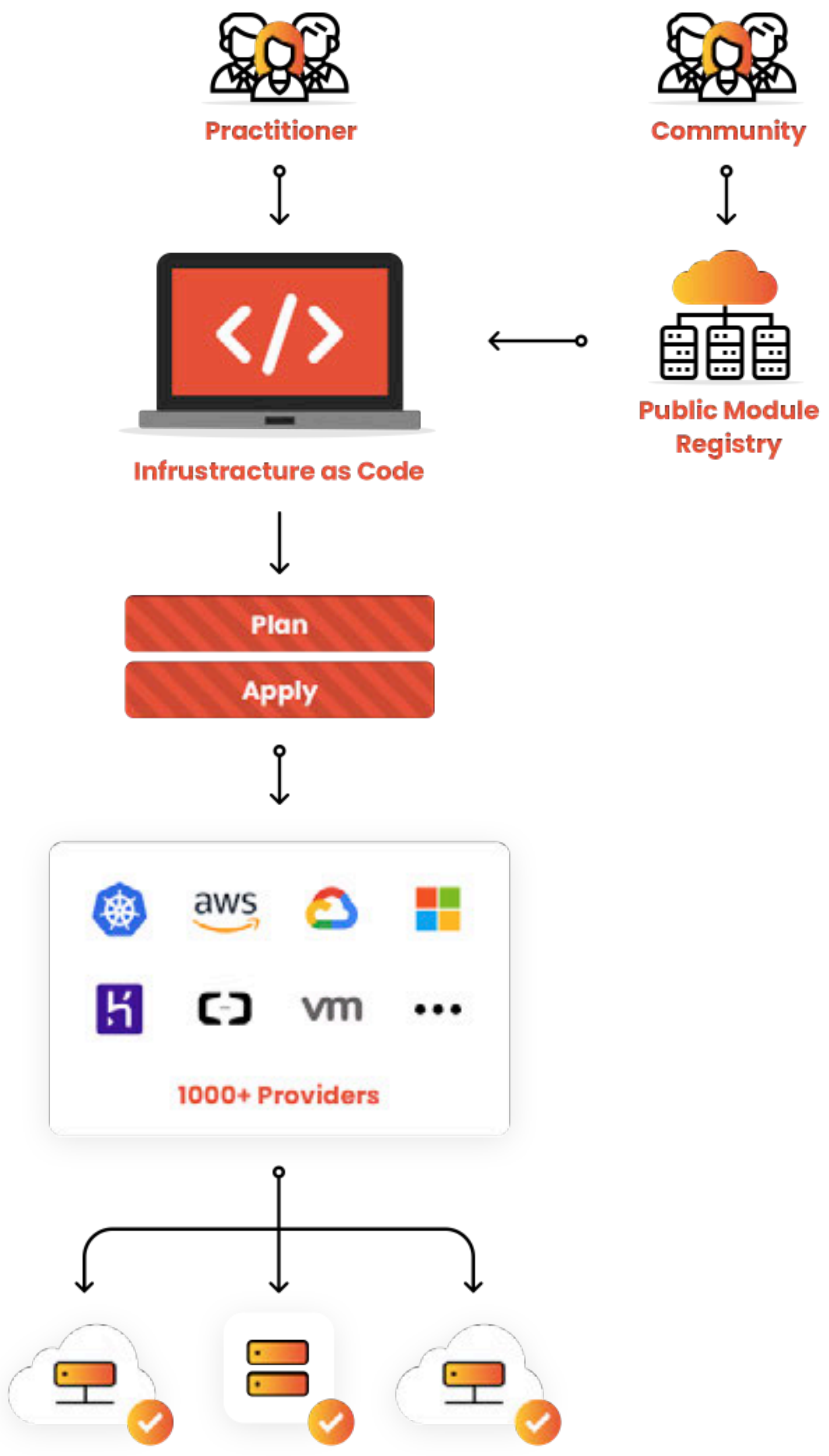
A typical collaborative workflow by Terraform (Source: HashiCorp)

Automated Infrastructure Provisioning and Documentation

Terraform facilities automated provisioning and configuration changes. The platform uses a source file to store configuration details representing the state of your infrastructure, which can be referred to across your organization.

Unified Scripting Language

Terraform uses a unified scripting language that acts as a baseline to modify scripts across multiple cloud providers.



Terraform Is Modular

The platform supports the concept of modularity and reusability through Terraform modules. These are similar to containers, allowing you to group resources together and create reusable configurations that can be called by other environments.

The 7 Pitfalls to Avoid When Using Terraform

Although Terraform allows developers to provision infrastructure through configuration files, several misconfigurations commonly occur due to human error. Since writing the configuration is a manual task, standards are often ignored or missed. Below are some common pitfalls that developers should avoid and the recommended best practices to help you do so.

1. Storing Secrets in Plain Text

Terraform files that contain declarative codes and variable declarations are usually maintained in a version control system (VCS) like Git for centralized access by cross-functional teams.

Recommended practice: To ensure high security for sensitive information, like account credentials or private data, such information should not be committed in VCS repositories to prevent exposure to attack vectors. Instead, you should store such information in an encrypted format rather than hardcode it as plaintext in unencrypted files.
You can avoid hardcoding sensitive data by:

- Excluding commit files that may contain sensitive information from tools that use VCS (such as **Gitignore**)
- Using **environment variables** to avoid using plaintext sensitive information in code
- Storing sensitive information within the VCS in an encrypted format via encryption keys in internal/third-party secret vaults

2. Storing State Locally

Once you build your infrastructure with Terraform configurations, a state file called **terraform.tfstate** is created in the local workspace directory. This state file records the details of all the infrastructure provisioned, which Terraform uses to refresh the configuration with the latest infrastructure before executing any operation. Storing files locally does not create any problems until the provisioned infrastructure is used in a silo. However, in a distributed team structure, keeping state files locally complicates Terraform usage, as each user in the team needs to ensure that they have the latest file with them and no other user is accessing it.

Recommended practice: It is best to maintain a **remote state** file and make Terraform write the state data in a remote data store called a **backend**. **As recommended by HashiCorp**, Amazon S3 is the most adopted method for a backend and is best paired with DynamoDB for **state locking**. Other commonly used remote backends include Terraform Cloud, Google Cloud Storage, and HashiCorp Consul.

3. Missing Version Control for Modules and Providers

While developing modules and specifying providers, developers often forget to pin their versions. This leads to a mismatch of local and remote versions while causing resolution overhead.

Recommended practice: To ensure seamless upgrades and reduce compatibility issues, you should explicitly mention the required provider's version in the configuration file. It is also recommended to pin the minimum provider versions each module is compatible with. Besides this, a module that is intended to be used as a root module should only provide a maximum provider version to prevent incompatible version upgrades.

Pro tip: Use an IDE plugin, such as **HashiCorp's language server** to remind you of version locks and other best practices.

4. Non-Reusable Modules

Modules are an important element that promotes reusability and structured programming. A common mistake is to use a service-level structure as suggested, but later make changes to modules, expecting them to span across other environments. Developers can also tend to focus on designing modules to solve an immediate requirement and may overlook the longer-term, more comprehensive benefits of a generic module.
In such cases, with every new requirement, developers then have to either write a new module or modify existing modules to incorporate changes, giving rise to countless versions of the same module. Apart from the issue of non-reusability, this often leads to pushing improper versions of code to production or having to deal with the challenges of backtracking issues.

Recommended practice: The suggested approach to overcome this is to create modules that are generic. Some characteristics of **generic** modules include:

- They perform only one operation.
- They do not contain environment-specific details, allowing them to be used across different environments.
- They are well-documented to make modification and adoption easier.
- For the efficient application of changes across multiple environments, generic modules implement changes on the source instead of on the downloaded code.

5. Incorrect Directory Structure

As Terraform is designed to be flexible for use across various organizational structures, a common misconfiguration occurs when developers organize their codebase based on the resource type. On account of its non-modular and non-reusable modules, a resource-type directory structure tends to not be compatible with a growing infrastructure and will later require refactoring.

Recommended practice: Use a directory structure based on the **service level.** This helps to isolate dependencies between the underlying repositories and enhance **modularity**. It is also recommended that you create separate repositories for each configuration to achieve faster module creation.

6. Missing Configuration Documentation

With a growing infrastructure and new requirements, developers are required to keep updating Terraform configuration files but often forget to document the changes due to lack of time.
Ironically, in the real world, most code complexities arise due to a lack of documentation. Without adequate documentation, debugging is often a herculean task that impacts the efficiency and timeline of an SDLC.

Recommended practice: Organizations should implement strict and clear policies on code documentation. Developers can additionally leverage features of the Terraform **HashiCorp Configuration Language (HCL)** to document variables and outputs, or add comments along with configurations.

7. Mismatching Terraform Binary Versions

Binary version mismatch occurs when the whole team works on a pinned version of a Terraform binary while one member of the team upgrades the binary to a different version and applies it locally. This breaks the entire pipeline and forces all other members of the team to upgrade their binary version to the newest one.

For any mismatch in the binary, Terraform is traditionally known to return errors that are complex to comprehend, thereby making backtracking and debugging a tedious job.

Recommended practice: A typical approach to solving version mismatches is by following **version constraints**, which help enforce an acceptable version range to avoid known incompatibilities. Setting version constraints also helps make any returned error messages less ambiguous for efficient debugging. Another recommended practice is to run Terraform builds through an automated CI pipeline. In such instances, the build is initiated with the pipeline to ensure a unified environment and avoid mismatches.

Conclusion

Gartner has predicted that the significance of automation will continue to rise to the extent that "[B]y **2025, more than 90% of enterprises will have an automation architect."** On top of this, Gartner also projected that "by 2024, **organizations will lower operational costs by 30%** by combining hyperautomation technologies with redesigned operational processes."

As the infrastructure of your organization grows, there will be challenges pertaining to increasing complexity and sub-optimal productivity. Although embracing automation supported by tools such as Terraform simplifies infrastructure operations at scale, developers need to be aware of the pitfalls often faced when implementing these tools and take proper action to avoid or manage them.

About Zesty

Zesty is the world's first AI-driven cloud management technology that auto-scales cloud resources to fit real-time application needs. As today's cloud environments become increasingly dynamic, Zesty automates cloud efficiency, improves DevOps productivity, and reduces cloud costs with zero human input. As a result, DevOps engineers no longer need to spend time on repetitive, and mundane infrastructure management tasks and can enjoy the cloud's flexibility and scalability without worrying about cost or maintenance concerns. Zesty was founded in 2019 in Tel Aviv and is used by leading organizations such as Armis, Gong, Yotpo, and others. For more information, visit **Zesty.co.**