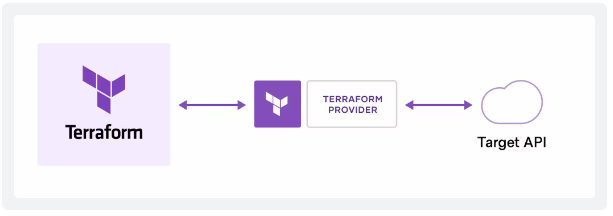
**What is Terraform?**

Terraform is an infrastructure as code tool that lets you build, change, and version cloud and on-prem resources safely and efficiently.

HashiCorp Terraform is an infrastructure as code tool that lets you define both cloud and on-prem resources in human-readable configuration files that you can version, reuse, and share. You can then use a consistent workflow to provision and manage all of your infrastructure throughout its lifecycle. Terraform can manage low-level components like compute, storage, and networking resources, as well as high-level components like DNS entries and SaaS features.

**How does Terraform work?**

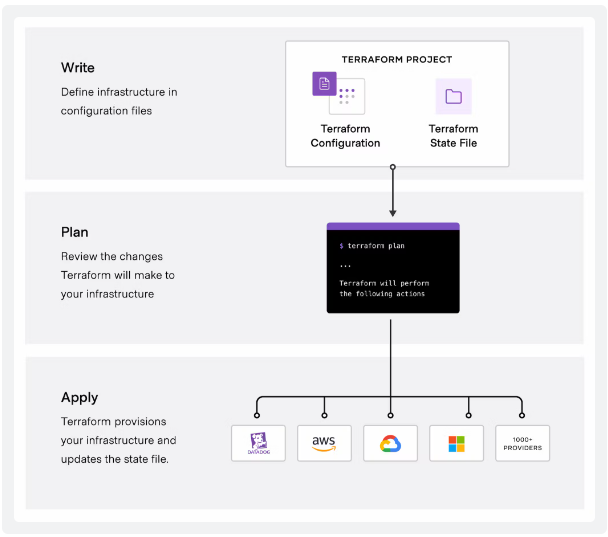
Terraform creates and manages resources on cloud platforms and other services through their application programming interfaces (APIs). Providers enable Terraform to work with virtually any platform or service with an accessible API.



HashiCorp and the Terraform community have already written **thousands of providers** to manage many different types of resources and services. You can find all publicly available providers on the [Terraform Registry](https://registry.terraform.io/), including Amazon Web Services (AWS), Azure, Google Cloud Platform (GCP), Kubernetes, Helm, GitHub, Splunk, DataDog, and many more.

**The core Terraform workflow consists of three stages:**

* **Write:** You define resources, which may be across multiple cloud providers and services. For example, you might create a configuration to deploy an application on virtual machines in a Virtual Private Cloud (VPC) network with security groups and a load balancer.
* **Plan:** Terraform creates an execution plan describing the infrastructure it will create, update, or destroy based on the existing infrastructure and your configuration.
* **Apply:** On approval, Terraform performs the proposed operations in the correct order, respecting any resource dependencies. For example, if you update the properties of a VPC and change the number of virtual machines in that VPC, Terraform will recreate the VPC before scaling the virtual machines.



**Why Terraform?**

HashiCorp co-founder and CTO Armon Dadgar explains how Terraform solves infrastructure challenges.

**Manage any infrastructure**

Find providers for many of the platforms and services you already use in the [Terraform Registry](https://registry.terraform.io/). You can also [write your own](https://developer.hashicorp.com/terraform/plugin). Terraform takes an [immutable approach to infrastructure](https://www.hashicorp.com/resources/what-is-mutable-vs-immutable-infrastructure), reducing the complexity of upgrading or modifying your services and infrastructure.

**Track your infrastructure**

Terraform generates a plan and prompts you for your approval before modifying your infrastructure. It also keeps track of your real infrastructure in a [state file](https://developer.hashicorp.com/terraform/language/state), which acts as a source of truth for your environment. Terraform uses the state file to determine the changes to make to your infrastructure so that it will match your configuration.

**Automate changes**

Terraform configuration files are declarative, meaning that they describe the end state of your infrastructure. You do not need to write step-by-step instructions to create resources because Terraform handles the underlying logic. Terraform builds a resource graph to determine resource dependencies and creates or modifies non-dependent resources in parallel. This allows Terraform to provision resources efficiently.

**Standardize configurations**

Terraform supports reusable configuration components called [modules](https://developer.hashicorp.com/terraform/language/modules) that define configurable collections of infrastructures, saving time and encouraging best practices. You can use publicly available modules from the Terraform Registry, or write your own.

**Collaborate**

Since your configuration is written in a file, you can commit it to a Version Control System (VCS) and use [HCP Terraform](https://developer.hashicorp.com/terraform/intro/terraform-editions#hcp-terraform) to efficiently manage Terraform workflows across teams. HCP Terraform runs Terraform in a consistent, reliable environment and provides secure access to shared state and secret data, role-based access controls, a private registry for sharing both modules and providers, and more.

**What Are the Providers in Terraform?**

A provider in Terraform is a plugin that enables interaction with an API. Providers allow Terraform to interact with cloud providers, SaaS providers, and other APIs.

The providers are specified in the Terraform configuration code. They tell Terraform which services it needs to interact with.

**Here are key aspects of a Terraform provider:**

1. **Resource Management**: Providers define a set of resources (such as virtual machines, databases, networks) and manage their lifecycle (creation, update, deletion).
2. **API Interactions**: Each provider interacts with the API of a specific platform (like AWS, Azure, Google Cloud, etc.) to perform CRUD (Create, Read, Update, Delete) operations on resources.
3. **Configuration**: Providers are configured in Terraform configuration files (\*.tf files) using a specific syntax. They require authentication credentials or tokens to authorize actions on the platform.
4. **Extensibility**: Terraform is extensible, allowing developers to create custom providers for proprietary or niche platforms that aren't supported by default.
5. **Versioning**: Providers are versioned separately from Terraform itself, allowing updates to be managed independently to support new features or APIs of the underlying platforms.

[**Some of the basic commands for Terraform are**](https://www.bing.com/ck/a?!&&p=f8f3d6f0e716acf3JmltdHM9MTcxOTI3MzYwMCZpZ3VpZD0wODBmMjlhZC0zZWExLTYwNWYtMTFjOC0zZDgwM2YzYTYxMTYmaW5zaWQ9NTg5NA&ptn=3&ver=2&hsh=3&fclid=080f29ad-3ea1-605f-11c8-3d803f3a6116&psq=terraform+basic+commands&u=a1aHR0cHM6Ly90ZWNhZG1pbi5uZXQvdGVycmFmb3JtLWJhc2ljLWNvbW1hbmRzLw&ntb=1)[**:**](https://www.bing.com/ck/a?!&&p=b3bbeff46cabab91JmltdHM9MTcxOTI3MzYwMCZpZ3VpZD0wODBmMjlhZC0zZWExLTYwNWYtMTFjOC0zZDgwM2YzYTYxMTYmaW5zaWQ9NTg5NQ&ptn=3&ver=2&hsh=3&fclid=080f29ad-3ea1-605f-11c8-3d803f3a6116&psq=terraform+basic+commands&u=a1aHR0cHM6Ly90ZWNhZG1pbi5uZXQvdGVycmFmb3JtLWJhc2ljLWNvbW1hbmRzLw&ntb=1)

* **terraform init:** This command initializes the configuration and downloads any required plugins or modules.
* **terraform plan:** This command creates an execution plan and shows what actions Terraform will take to achieve the desired state.
* **terraform apply:** This command applies the changes in the execution plan to the actual infrastructure.
* **terraform destroy:** This command destroys the resources that are managed by Terraform.
* **terraform validate:** This command checks the syntax and validity of the configuration files.

The code snippet you provided is a Terraform configuration block specifying a required provider for Azure resources using the azurerm provider plugin from HashiCorp.

Let's break down what each part of the configuration means:

terraform {

required\_providers {

azurerm = {

source = "hashicorp/azurerm"

version = "3.109.0"

}

}

}

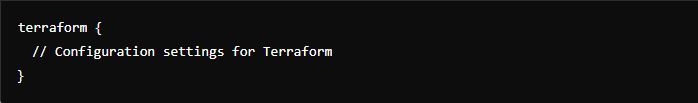
provider "azurerm" {

# Configuration options

}

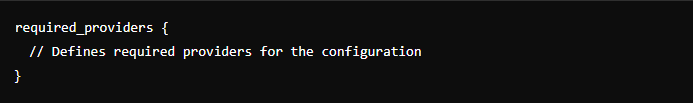
**Explanation:**

1. **‘terraform’ block:**



This block defines the version and required providers for your Terraform configuration.

1. **‘required\_providers’ Block:**



Within the terraform block, the required\_providers block specifies which providers are needed for the Terraform configuration.

In this case, it specifies that the configuration requires the ‘**azurerm’** provider.

1. **azurerm Provider Configuration:**



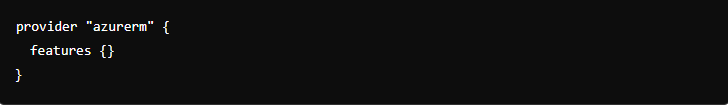
* **azurerm:** This is the name of the provider. It refers to the Terraform Azure provider, which allows Terraform to interact with resources in Microsoft Azure.
* **source**: Specifies the source of the provider plugin. In this case, hashicorp/azurerm indicates that Terraform should download the azurerm provider plugin from the HashiCorp Terraform Registry, specifically from the namespace hashicorp.
* **version**: Specifies the version of the azurerm provider plugin that Terraform should use. Here, it is set to "3.109.0", meaning Terraform will use version 3.109.0 of the azurerm provider.

### **Usage:**

* Once you have this configuration in your .tf file, when you run terraform init, Terraform will download the specified version of the azurerm provider plugin from the Terraform Registry and make it available for use in your Terraform configuration.
* You can then start defining Azure resources in your Terraform files using the resources and data sources provided by the azurerm provider, using the alias azurerm.

**4. ‘provider "azurerm"’ block:**





**Explanation:**

* ‘provider "azurerm"’: This declares that you are configuring the Azure provider (azurerm) for Terraform.
* Configuration Options:
* Inside the braces {}, you would typically specify configuration options such as authentication details, subscription ID, region, etc. These options depend on what your Terraform setup requires to interact with Azure.
* Configuration options are specified using key-value pairs, and they can vary depending on the provider and the resources you intend to manage.

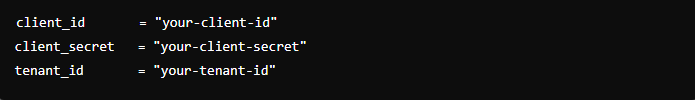
**Example Configuration Options:**

Here are some common configuration options you might include inside the provider block:

* **Subscription ID**: Specifies the Azure subscription ID to use.



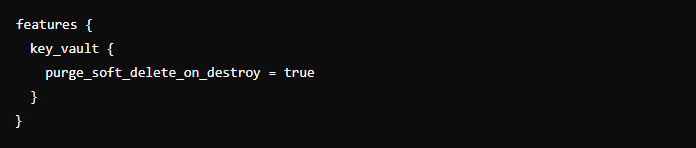
* **Client ID, Client Secret, and Tenant ID**: Used for service principal authentication.



* **Version**: Specifies the version of the Azure provider to use.

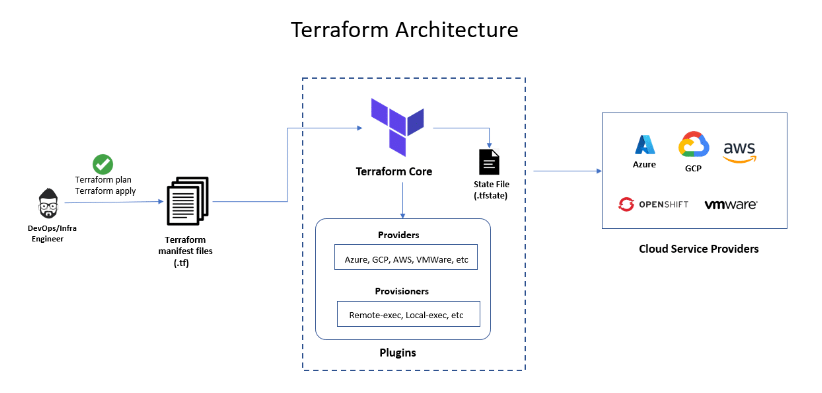


* **Features**: Allows enabling or disabling specific features of the provider.



**Additional Notes:**

* **Authentication**: For Azure, you typically authenticate using either a service principal (client ID, client secret, tenant ID) or managed identity.
* **Resource Configuration**: After configuring the provider, you would define resources (e.g., virtual machines, storage accounts) that Terraform should manage within the same configuration files.



**Explanation:**

* **Purpose:** This block ensures that when you run terraform init, Terraform will download and install the AzureRM provider version 3.109.0 from the HashiCorp Terraform Registry.
* **Version Constraint:** The version constraint (3.109.0) ensures that Terraform will use exactly this version of the provider. This is important for ensuring consistency and compatibility with the Terraform configuration and the Azure resources you plan to manage.
* **Provider Source:** By specifying the source, you declare where Terraform can find the provider. The HashiCorp Terraform Registry (hashicorp/azurerm) is the default registry for many providers.

This configuration block is essential for ensuring that your Terraform configuration can successfully interact with Azure resources using the specified provider and version.

**Summary:**

The terraform block with required\_providers ensures that Terraform knows which provider to use (azurerm) and the exact version (3.109.0) required for managing Azure resources effectively. It helps maintain consistency and reliability in your infrastructure provisioning and management workflows.

**Usage:**

**To use this Terraform configuration:**

Ensure you have Terraform installed on your machine.

Create a .tf file (e.g., main.tf) and paste the provided configuration block into it.

Run terraform init in the directory containing your .tf file. This initializes Terraform and downloads the specified provider plugin (azurerm in this case) to your local system.

Proceed with writing the rest of your Terraform configuration (resource, variable, output blocks, etc.) that defines the Azure infrastructure you want to manage.

Use terraform plan and terraform apply commands to plan and apply changes to your Azure infrastructure based on your Terraform configuration.

In summary, the terraform block with required\_providers is a crucial part of any Terraform configuration, defining the providers and their versions necessary for managing infrastructure resources.

**Terraform state:**

The terraform state file, by default, is named terraform. tfstate and is held in the same directory where Terraform is run. It is created after running terraform apply.

Terraform stores information about your infrastructure in a state file. This state file keeps track of resources created by your configuration and maps them to real-world resources.

Terraform compares your configuration with the state file and your existing infrastructure to create plans and make changes to your infrastructure. When you run terraform apply or terraform destroy against your initialized configuration, Terraform writes metadata about your configuration to the state file and updates your infrastructure resources accordingly.

In Terraform, the state file is created as soon as you run the terraform apply command for the first time in a directory where you have defined your infrastructure using Terraform configuration files (usually with .tf extension).

Here’s a breakdown of when the state file is created:

1. **First terraform apply:** When you run terraform apply for the first time in a directory where you have your Terraform configuration files (main.tf, variables.tf, etc.), Terraform will initialize and create a state file (terraform.tfstate by default) in that directory. This state file keeps track of the current state of your infrastructure managed by Terraform
2. **Purpose of the state file:** The state file is crucial because it helps Terraform understand what infrastructure it manages and what changes need to be made when you run commands like terraform apply or terraform destroy. It stores mappings between your resources in the configuration files and the actual resources created in your cloud provider (AWS, Azure, Google Cloud, etc.).
3. **Location of the state file:** By default, the state file (terraform.tfstate) is created in the current working directory where you run Terraform commands. However, it’s recommended to use remote state storage (like Amazon S3, Azure Blob Storage, or HashiCorp Consul) in a production environment for better collaboration and durability.
4. **Subsequent runs:** After the initial terraform apply, subsequent runs of terraform apply or terraform plan will update this state file with any changes made to your infrastructure configuration. It keeps track of resource attributes, dependencies, and metadata required by Terraform.

To summarize, the state file in Terraform is created the first time you apply your infrastructure configuration (terraform apply). It is essential for Terraform to maintain and manage your infrastructure effectively across multiple executions of Terraform commands.

**Terraform drift:**

In Terraform, "zero drift" state refers to the concept where the actual infrastructure state matches exactly what is defined in your Terraform configuration files (\*.tf files). This means that there are no differences or discrepancies between what Terraform expects to be provisioned (as per your configuration) and what actually exists in your cloud provider (AWS, Azure, Google Cloud, etc.).

Here are the key aspects of zero drift state in Terraform:

**Ideal State:** When you first apply your Terraform configuration (terraform apply), Terraform creates a state file (terraform.tfstate by default) that records the current state of your infrastructure. Ideally, this state file should accurately reflect what you have defined in your Terraform configuration files.

**Drift:** Drift occurs when there are differences between the state described in your Terraform configuration and the actual state of resources in your cloud provider. This can happen due to manual changes made directly in the cloud console or through other automation tools outside of Terraform.

**Zero Drift:** Zero drift state means that Terraform manages all resources exactly as specified in your configuration files. It implies that no manual changes have been made outside of Terraform that would alter the intended state of your infrastructure.

In summary, zero drift state in Terraform refers to the state where the actual infrastructure matches exactly what is defined in your Terraform configuration files, ensuring consistency and predictability in your infrastructure management practices.