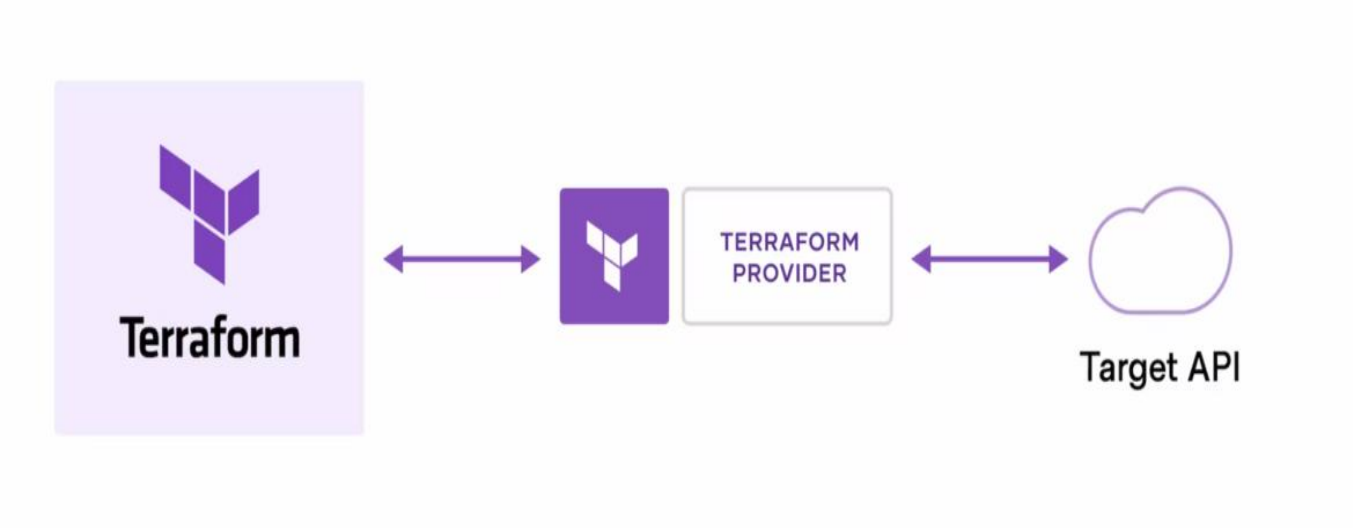
Terraform: Automating Cloud Infrastructure with Infrastructure as Code

Introduction to Terraform and Cloud Automation

In the evolving landscape of cloud computing, **automation** is no longer a luxury—it's a necessity. Manual provisioning and configuration of infrastructure are time-consuming and error-prone. This is where **Terraform**, an open-source **Infrastructure as Code (IaC)** tool developed by **HashiCorp**, becomes essential.

Terraform allows developers and DevOps teams to **define, provision, and manage infrastructure** across multiple cloud platforms—including **AWS, Azure, Google Cloud**, and even on-premises systems—**using declarative configuration files**. It helps automate the entire infrastructure lifecycle with **predictability**, **scalability**, and **version control**.



Infrastructure as Code (IaC) Overview

Infrastructure as Code is the practice of managing and provisioning computing infrastructure through **machine-readable configuration files**, rather than using interactive configuration tools or manual processes.

Benefits of IaC:

- **Consistency**: Same configuration yields same infrastructure every time.
- **Version Control**: Code can be tracked, reviewed, and rolled back using Git.
- **Reusability**: Use modules and templates across multiple environments.
- **Automation**: Speeds up provisioning, improves reliability, and minimizes manual effort.

Terraform stands at the forefront of IaC tools due to its:

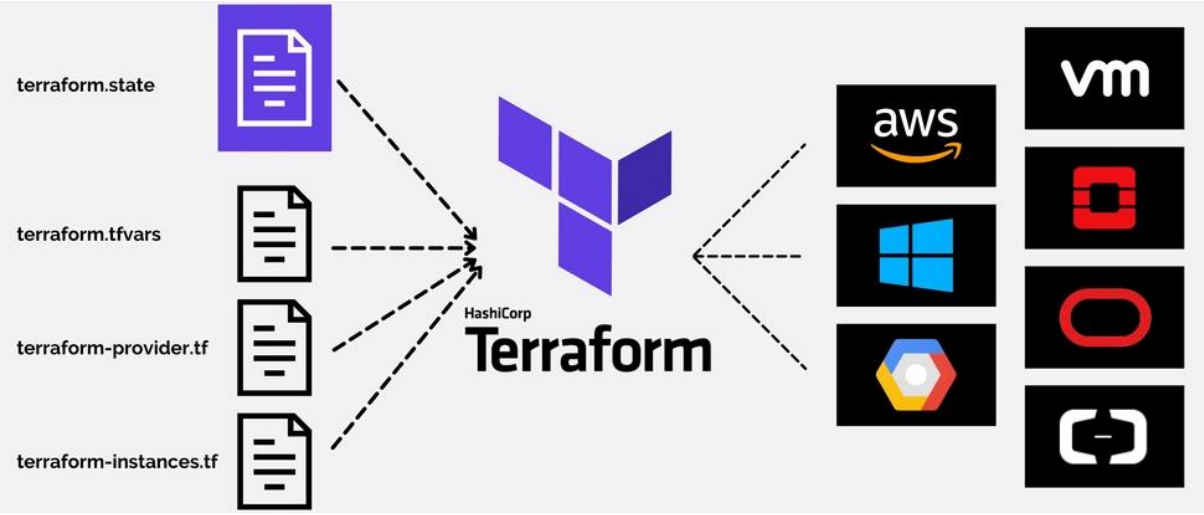
- Cloud-agnostic nature
- Strong community support
- Modular design
- State management system

Key Terraform Concepts

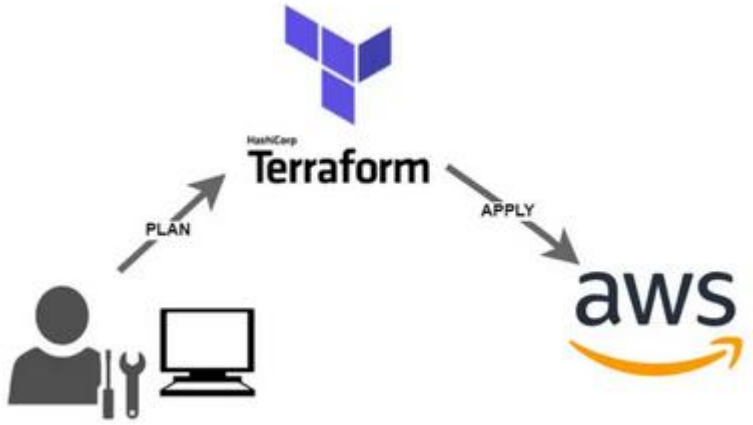
main.tf: The Core Configuration File

The main.tf file is the **primary Terraform configuration file**, written in **HCL (HashiCorp Configuration Language)**. It contains the resource definitions and specifies:

- **Providers** (like AWS)
- **Resources** (like EC2, VPC, S3, etc.)
- **Input variables**
- **Resource dependencies**



Terraform on AWS:



Terraform Overview – Building Cloud Environments Like Terraforming Planets

When we hear the word "terraform", most people think of terraforming planets—a fascinating idea from science fiction and real-world thinkers like Elon Musk, who dreams of making Mars a place where humans could live one day. In that context, *terraforming* means turning a lifeless planet into a livable one. It includes creating an atmosphere, adding water, and growing plants so that eventually, life can survive and thrive there.

Now, imagine doing something similar—not in space, but in the world of cloud computing. This is where HashiCorp's Terraform comes in. While it doesn't build planets, it builds digital environments. Terraform helps developers and system engineers transform raw cloud resources—like virtual machines, storage, and networking—into well-organized, secure, and repeatable cloud infrastructure.

Just as a lifeless planet needs structure and resources to support life, cloud platforms like AWS, Google Cloud (GCP), Microsoft Azure, and others need structured resources to support applications. Terraform allows you to define that structure using simple code. This idea is known as Infrastructure as Code (IaC). Instead of setting up servers or databases manually, you write configuration files in Terraform to tell the cloud what you want to build.

Let's say you want to create:

- A web server
- A database
- A network to connect them

In the past, you'd have to log in to your cloud provider, click through menus, and create each part manually. This is slow and error-prone. But with Terraform, you just write some code once and use it to deploy the same setup again and again—across multiple environments like development, testing, and production.

This means you can:

- Save time
- Avoid mistakes
- Keep track of changes (like with Git)
- Automate everything

In a way, Terraform becomes your blueprint. Just like engineers use blueprints to build houses, you use Terraform to build digital homes for your applications.

Terraform supports not just AWS or GCP, but dozens of cloud and infrastructure providers. You can even use it to manage on-premise environments using tools like VMware vSphere, Docker, or Kubernetes.

In summary, Terraform is a powerful tool that helps you build and manage cloud environments efficiently and automatically. It's inspired by the idea of transforming barren space into something alive—and in our case, it transforms empty cloud space into functional, reliable environments where applications can live, grow, and succeed.

What is Terraform?

Terraform is an Infrastructure as Code (IaC) tool developed by HashiCorp. It allows users to define cloud and on-premises infrastructure using human-readable configuration files. The tool provides a consistent workflow to provision, manage, and automate infrastructure across its lifecycle.

Why Use Terraform?

1. Simplicity – All infrastructure is defined in a single file, making it easy to track changes.
2. Collaboration – Code can be stored in version control systems like GitHub for team collaboration.
3. Reproducibility – Configurations can be reused for different environments (e.g., development and production).
4. Resource Cleanup – Ensures unused resources are properly destroyed to avoid unnecessary costs.

What Terraform is NOT

1. Not a Software Deployment Tool – It doesn’t manage or update software on existing infrastructure.

2. Cannot Modify Immutable Resources – Some changes (e.g., VM type) require destroying and recreating the resource.
3. Does Not Manage External Resources – Terraform only manages what is explicitly defined in its configuration files.

Terraform Workflow

1. Terraform Installed Locally – The CLI runs on a user's machine.
2. Uses Providers – These connect Terraform to cloud services (AWS, Azure, GCP, etc.).
3. Authentication Required – API keys or service accounts authenticate access to cloud platforms.

Terraform Installation

Key Terraform Commands

1. terraform init – Downloads provider plugins and initializes the working directory.
2. terraform plan – Shows what changes Terraform will make before applying them.
3. terraform apply – Provisions the defined infrastructure.
4. terraform destroy – Removes all resources defined in the Terraform configuration.

Terraform Files and Their Generation

Terraform uses several files to manage your infrastructure state, configuration, and dependencies. Below is an overview of the key files, what they represent, and which Terraform command triggers their creation or update:

1. **Configuration Files (*.tf files):**
    - **Purpose:** These files (such as main.tf, variables.tf, outputs.tf, etc.) are written by you to define your infrastructure. They describe the resources you wish to provision and how they interrelate.
    - **When They Are Created:** You create these manually. They form the blueprint for Terraform to understand and manage your environment.
2. **Terraform State File (terraform.tfstate):**
    - **Purpose:** This file tracks the current state of your infrastructure. It maps your configuration to the real-world resources, ensuring that Terraform can determine what changes need to be made.
    - **When It Is Generated/Updated:**
        - **After terraform apply:** When you run terraform apply, Terraform provisions your infrastructure based on your configuration. During this

process, it creates or updates the terraform.tfstate file with the current state of the resources.

- **After terraform destroy:** Similarly, when you destroy resources, the state file is updated to reflect that the resources no longer exist.

3. **Terraform Lock File (.terraform.lock.hcl):**
    - **Purpose:** This file locks the versions of the provider plugins used in your configuration to ensure consistency and prevent unexpected changes from newer versions.
    - **When It Is Generated/Updated:**
        - **After terraform init:** Running terraform init downloads the required provider plugins and creates the .terraform.lock.hcl file. This ensures that every team member or CI/CD pipeline uses the same provider versions.
4. **Terraform Directory (.terraform):**
    - **Purpose:** This hidden directory stores downloaded provider plugins, module sources, and backend configuration. It is essential for Terraform's operation.
    - **When It Is Generated:**
        - **After terraform init:** The .terraform directory is automatically created when you initialize your Terraform working directory using terraform init.
5. **Plan Output File (optional):**
    - **Purpose:** If you choose to save the execution plan to a file (using the -out flag with terraform plan), this binary file captures the set of changes Terraform intends to make.
    - **When It Is Generated:**
        - **After terraform plan -out=<filename>:** Running this command generates a plan file that can later be applied using terraform apply <filename>. This is useful for reviewing changes or automating deployment workflows.

Configure aws credentials locally

This is important if you want to be able to access your aws account and resources on your terminal or inside your code using an sdk such as boto3.
There might be other ways to do this but I will list 2 here:
aws configure
Export the credentials as environment variables

1. aws configure

I found that **sudo apt install aws-cli** or **pip3 install aws-cli** worked just as well for me.

You can confirm that you have installed it by checking it's version as below:

```
aws --version
```

To configure your credentials locally, you will need to create an IAM user and give them some permissions.

1. Log onto your aws console
2. Navigate to the IAM section
3. Create a new user and grant them the required permissions
4. Download the access key and secret key as csv of that user and store it securely. I prefer this rather than just copying them from the console. Just make sure not to commit them publicly. For example, if you're working on a repository that has a remote version in github/gitlab/bitbucket etc then consider adding the csv to your git ignore before adding, committing and pushing changes

Now, run the following command to configure your credentials locally.

```
aws configure
```

You will be prompted to enter the access key id, secret access key, region, and output format (choose json, text, or table, default is json). Once you fill them, it creates 2 files in the ~/.aws directory: credentials and config. The credentials file contains the access key and secret key. The config file contains the region and output format.

To test if you have access to your aws account from your local terminal, create a dummy s3 bucket then run the following command to list your buckets:

```
aws s3 ls
```

2. Export the credentials as environment variables

Run the following command on your terminal, replacing the stringed text with your actual values:

```
export AWS_ACCESS_KEY_ID="your-access-key-id"
export AWS_SECRET_ACCESS_KEY="your-secret-access-key"
export AWS_DEFAULT_REGION="your-region"
```

Terraform, Boto3, and AWS CLI will automatically use these from the environment variables or the files in the ~/.aws directory

***Managing resources on terraform ***

Create a main.tf file in the folder you are working in and save the following code in the file:

```
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "5.85.0"
    }
  }
}

provider "aws" {
  # Configuration options
  region = "your-region"
}

# Variable definitions
variable "aws_region" {
  description = "AWS region for resources"
  type        = string
  default     = "your-region"
}

resource "aws_s3_bucket" "example" {
  bucket = "my-tf-test-bucket-${random_id.bucket_suffix.hex}" # Make bucket name unique

  tags = {
    Name        = "My bucket"
    Environment = "Dev"
  }
```

```
}


# Add a random suffix to ensure bucket name uniqueness
resource "random_id" "bucket_suffix" {
  byte_length = 4
}
```

Replace “your-region” with your actual region.

Now we can run the terraform commands.

1. Initialize terraform in your folder

Run the following command:

```
terraform init
```

It initializes the backend and provider plugins. It also creates a lock file .terraform.lock.hcl to record the provider selections. It also creates a .terraform folder. Include this file in your version control repository so that Terraform can guarantee to make the same selections by default when
you run "terraform init" in the future.

2. Plan

Run the following command to see any changes that are required for your infrastructure:

```
terraform plan
```

It generates an execution plan based on the code in main.tf. At the end of the output there is a summary that looks like the below:

Plan: 2 to add, 0 to change, 0 to destroy.

Changes will be suggested which you can agree to by running the apply command explained below. You can save the plan by using the out flag:

```
terraform plan -out=filepath-to-save-file
```

3. Apply

To apply the changes suggested run the command:

```
terraform apply
```

You will be prompted to confirm by typing yes. Be careful as this does create the resources thus you will incur costs on your aws account unless you have cloud credits or are using the free tier.

After typing yes, go to the console and navigate to the s3 section. Check if you have a new bucket created. Alternatively, you can run the following command to list your buckets:

```
aws s3 ls
```

If you had previously saved your plan to a file, please run the following command to apply that plan:

```
terraform apply "filename"
```

4. Delete

Run the following command to delete the resources provisioned by terraform

Your resources marked for deletion will be listed and you will again be prompted for confirmation. Confirm by typing yes.

Summary

This guide walks you through setting up AWS credentials locally using both the AWS CLI configuration and environment variables, and demonstrates how to manage AWS resources with Terraform. You learned how to write a basic Terraform configuration to create an S3 bucket, initialize your project, preview changes with a plan, apply those changes, and ultimately destroy the resources when they are no longer needed. This systematic approach to infrastructure management not only ensures consistency and repeatability but also aligns with modern DevOps best practices.