**Day 1: Introduction to Terraform and Terraform Basics**

* **What is Terraform and how can it help you manage infrastructure as code?**

Managing [infrastructure](https://www.bmc.com/blogs/what-is-it-infrastructure-and-what-are-its-components/) is a core requirement for most modern applications. Even in [PaaS or serverless](https://www.bmc.com/blogs/serverless-paas/) environments, there will still be components that require user intervention for customization and management. With the ever-increasing complexity of software applications, more and more infrastructure modifications are required to facilitate the functionality of the software.

It is unable to keep up with the [rapid development cycles](https://www.bmc.com/blogs/sdlc-software-development-lifecycle/) with manual infrastructure management. It will create bottlenecks leading to delays in the delivery process.

Infrastructure as Code (IaC) has become the solution to this issue—allowing users to align infrastructure changes with development. It also facilitates faster automated repeatable changes by codifying all the infrastructure and configuration and managing them through the delivery pipeline.

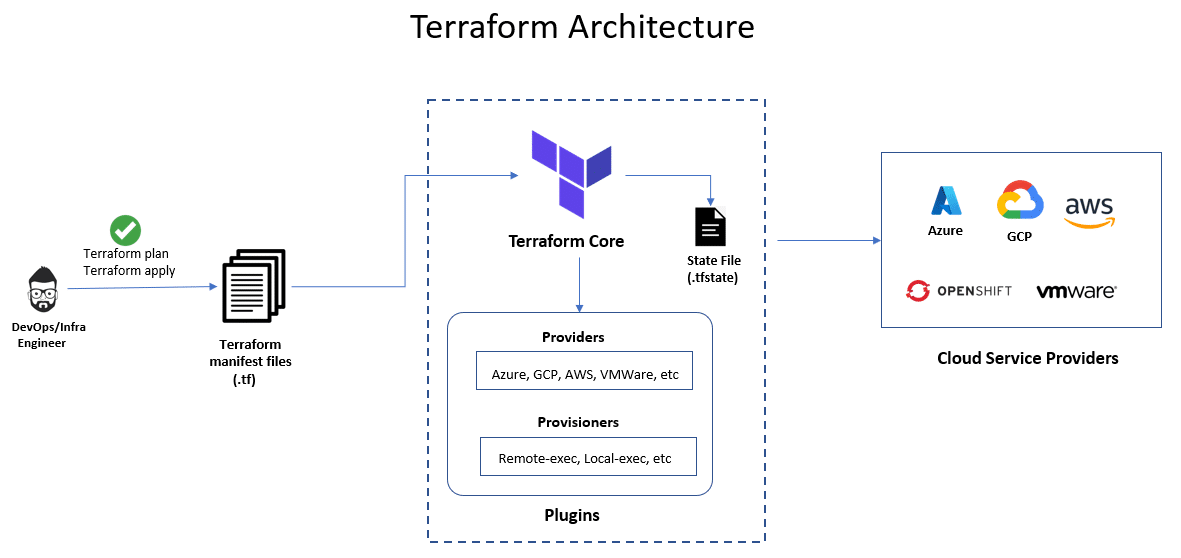
Terraform is one of the leading platform agnostic IaC tools that allow users to define and manage infrastructure as code. In this article, let’s dig into what Terraform is and how we can utilize it to manage infrastructure at scale.

**What is Infrastructure as Code?**

Infrastructure as Code (IaC) tools allow you to manage infrastructure with configuration files rather than through a graphical user interface. IaC allows you to build, change, and manage your infrastructure in a safe, consistent, and repeatable way by defining resource configurations that you can version, reuse, and share.

Terraform is HashiCorp's infrastructure as code tool. It lets you define resources and infrastructure in human-readable, declarative configuration files, and manages your infrastructure's lifecycle. Using Terraform has several advantages over manually managing your infrastructure:

* Terraform can manage infrastructure on multiple cloud platforms.
* The human-readable configuration language helps you write infrastructure code quickly.
* Terraform's state allows you to track resource changes throughout your deployments.
* You can commit your configurations to version control to safely collaborate on infrastructure.



**What is Terraform?**

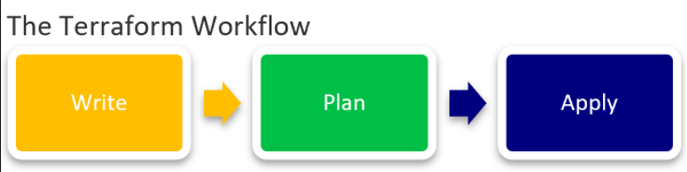
Terraform is an open-source infrastructure as a code tool from [HashiCorp](https://www.hashicorp.com/" \t "_blank). It allows users to define both on-premises and cloud resources in human-readable configuration files that can be easily versioned, reused, and shared. Terraform can be used to manage both low-level components (like compute, storage, and networking resources) as well as high-level resources (DNS, PaaS, and SaaS components).

Terraform is a declarative tool further simplifying the user experience by allowing users to specify the expected state of resources without the need to specify the exact steps to achieve the desired state of resources. Terraform manages how the infrastructure needs to be modified to achieve the desired result.

Terraform is a platform-agnostic tool, meaning that it can be used across any supported provider. Terraform accomplishes this by interacting with the [APIs](https://www.bmc.com/blogs/microservice-vs-api/) of cloud providers. When a configuration is done through Terraform, it will communicate with the necessary platform via the API and ensure the defined changes are carried out in the targeted platform. With [more than 1,700 providers](https://registry.terraform.io/browse/providers) from HasiCorp and the Terraform community available with the Terraform Registry, users can configure resources from leading cloud providers like Azure, AWS, GCP, and Oracle Cloud to more domain-specific platforms like Cloudflare, Dynatrace, elastic stack, datadog, and Kubernetes.

**The Terraform workflow**

The Terraform workflow is one of the simplest workflows only consisting of three steps to manage any type of infrastructure. It provides users the flexibility to change the workflow to support their exact implementation needs.



1. Write

The first stage of the workflow is where users create the configurations to define or modify the underlying resources. It can be as simple as provisioning a simple compute instance in a cloud provider to deploy a multi-cloud Kubernetes cluster. This writing part can be facilitated either through [HasiCorp Configuration Language (HCL)](https://www.terraform.io/language/syntax/configuration" \t "_blank), the default language to define resources or using the [Cloud Development Kit for Terraform (CDKTF)](https://github.com/hashicorp/terraform-cdk) which allows users to define resources using any supported common programming languages like Python, C#, Go, and Typescript.

2. Plan

This is the second stage of the workflow where Terraform will look at the configuration files and create an execution plan. It enables users to see the exact charges that will happen to the underlying infrastructure from what new resources are getting created, resourced, modified, and deleted.

3. Apply

This is the final stage of the workflow which takes place if the plan is satisfactory once the user has confirmed the changes. Terraform will carry out the changes to achieve the desired state in a specific order respecting all the resource dependencies. It will happen regardless of whether you have defined dependencies in the configuration. Terraform will automatically identify the resource dependencies of the platform and execute the changes without causing issues.

Terraform uses the state to keep track of all the changes to the infrastructure and detect config drifts. It will create a state file at the initial execution and subsequently update the state file with new changes. This state file can be stored locally or in a remote-backed system like an s3 bucket. Terraform always references this state file to identify the resources it manages and keep track of the changes to the infrastructure.

* **Why do we need Terraform and how does it simplify infrastructure provisioning?**

Let’s look at why so many people appreciate Terraform

* **Declarative nature.**A declarative tool allows users to specify the end state and the IaC tools will automatically carry out the necessary steps to achieve the user configuration. It is in contrast to other imperative IaC tools where users need to define the exact steps required to achieve the desired state.
* **Platform agnostics.**Most IaC tools like AWS CloudFormation and Azure Resource templates are platform specific. Yet, Terraform allows users to use a single tool to manage infrastructure across platforms with applications using many tools, platforms, and multi-cloud architectures.
* **Reusable configurations.**Terraform encourages the creation of reusable configurations where users can use the same configuration to provision multiple environments. Additionally Terraform allows creating reusable components within the configuration files with modules.
* **Managed state.**With state files keeping track of all the changes in the environment, all modifications are recorded and any unnecessary changes will not occur unless explicitly specified by the user. It can be further automated to detect any config drifts and automatically fix the drift to ensure the desired state is met at all times.
* **Easy rollsbacks.**As all configurations are version controlled and the state is managed, users can easily and safely roll back most infrastructure configurations without complicated reconfigurations.
* **Integration to CI/CD.**While IaC can be integrated into any pipeline, Terraform provides a simple three-step workflow that can be easily integrated into any CI/CD pipeline. It helps to completely automate the infrastructure management.

**How to use Terraform**

You can start using Terraform by simply installing it in your local environment. Terraform supports Windows, Linux, and macOS environments. It provides users the option to install manually using a pre-compiled binary, or use a package manager like Homebrew on Mac, Chocolatey on Windows, Apt/Yum on Linux. It offers users the flexibility to install Terraform in their environments and integrate it into their workflows.

HashiCorp also provides a [managed solution called Terraform Cloud](https://cloud.hashicorp.com/products/terraform). It provides users with a platform to manage infrastructure on all supported providers without the hassle of installing or managing Terraform itself. Terraform Cloud consists of features like;

* Remote encrypted state storage
* Direct CI/CD integrations
* Fully remote and SOC2 compliant collaborative environment
* Version Controls
* Private Registry to store module and Policy as Code support to configure security and compliance policies
* Complete auditable environment.
* Cost estimations before applying infrastructure changes in supported providers.

Additionally, Terraform Cloud is deeply integrated with other [HasiCrop Cloud Platform](https://cloud.hashicorp.com/" \t "_blank) services like Vault, Consul, and Packer to manage secrets, provide service mesh and create images. All these things allow users to manage their entire infrastructure using the HasiCorp platform.

* Explain the important terminologies of Terraform with the example at least:

Essential Components of Terraform Provider: A provider is a plugin responsible for interacting with a specific infrastructure platform or service (e.g., AWS, Azure, Google Cloud). Terraweek Day 1 5 provider "aws" { region = "us-east-1" } Resource: A resource represents a tangible infrastructure component, such as a virtual machine, database, or network. resource "aws\_instance" "example" { ami = "ami-0c55b159cbfafe1f0" instance\_type = "t2.micro" } Module A module is a reusable collection of Terraform configurations that can be used to encapsulate and abstract infrastructure components. module "web\_server" { source = "./modules/web\_server" instance\_count = 2 } Variable Variables are used to parameterize Terraform configurations, allowing you to reuse configurations with different inputs. variable "instance\_count" { description = "Number of instances to create" type = number default = 1 } Output Outputs allow you to expose specific values from your Terraform configuration for reference or use in other scripts. Terraweek Day 1 6 output "instance\_ips" { value = aws\_instance.example[\*].public\_ip } State The Terraform state is a crucial concept. It's a JSON file that stores the current state of your infrastructure as known to Terraform. It keeps track of what resources have been created and their current configuration. Terraform uses the state to understand what changes are needed to align your infrastructure with your configuration. Here’s a sample statefile: { "version": 4, "terraform\_version": "0.15.0", "serial": 3, "lineage": "1b234567-89ab-cdef-0123-456789abcdef", "modules": [ { "path": [ "root" ], "outputs": {}, "resources": { "aws\_instance.example": { "type": "aws\_instance", "depends\_on": [], "primary": { "id": "i-0123456789abcdef0", "attributes": { "ami": "ami-0c55b159cbfafe1f0", "instance\_type": "t2.micro", "private\_ip": "10.0.0.123", "public\_ip": "34.56.78.90", // ... other attributes ... }, "meta": {}, "tainted": false } } } } ] }