DevOps

Caltech | Center for Technology & Management Education

Post Graduate Program in DevOps

©Simplilearn. All rights reserved.

simplilearn

Ansible on Cloud with Terraform

©Simplilearn. All rights reserved.

Learning Objectives

By the end of this lesson, you will be able to:

- Explain the working of Terraform
- Illustrate and deploy playbooks on AWS
- Create deployment pipeline with Jenkins and Ansible
- Work with weave using DHCP or host-local

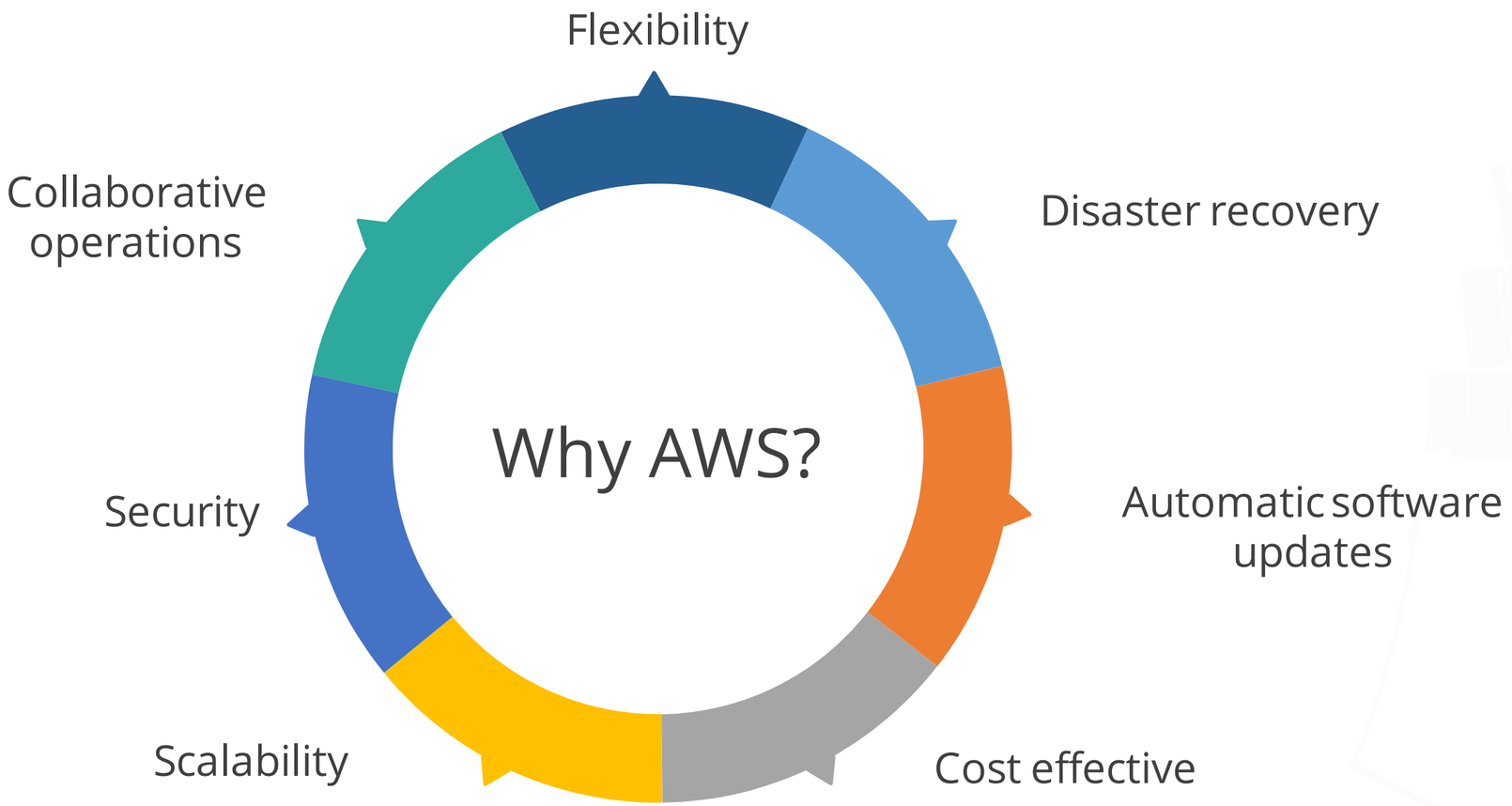
©Simplilearn. All rights reserved.

AWS Support with Ansible

©Simplilearn. All rights reserved.

Why AWS?

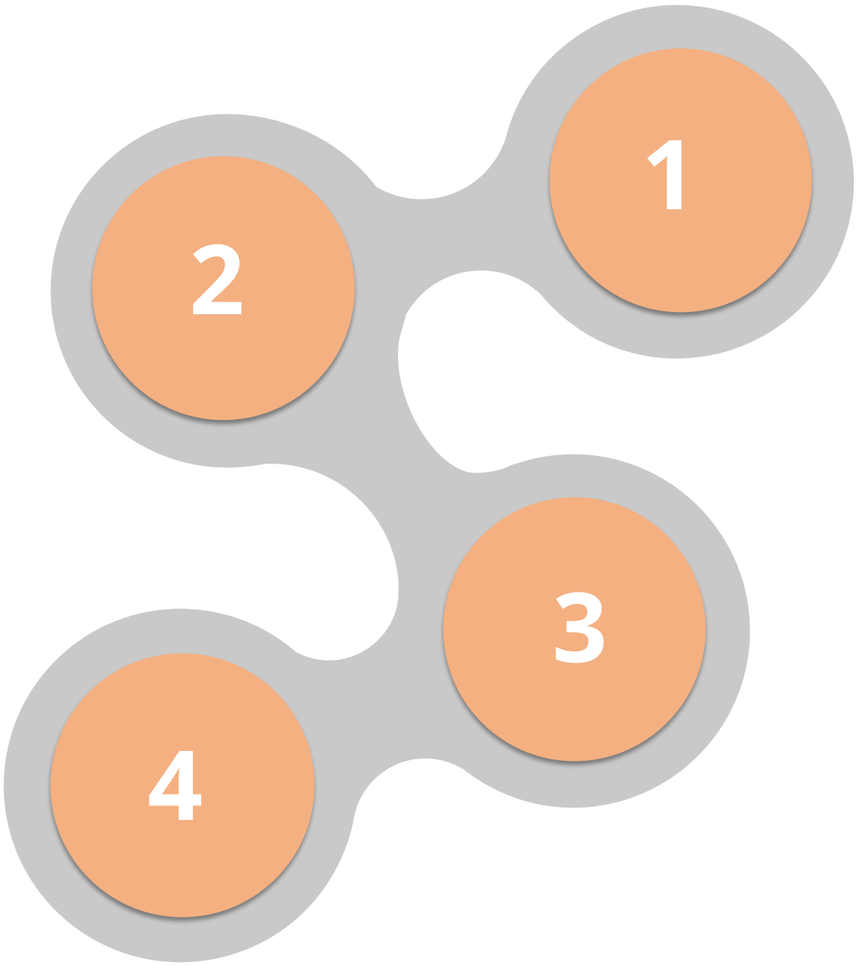
AWS is one of the most stable and reliable platforms to perform the cloud operations.



©Simplilearn. All rights reserved.

Why AWS?

Automatic software updates
It is a cloud-based service which makes it easier for developers to set periodic checks to keep all the tools up-to-date.



Flexibility
AWS is a cloud-based service that is suitable for fluctuating business demands.

Cost effective
It is a cloud-based service which reduces the hardware and software costs.

Disaster recovery
It has disaster recovery solutions for the customers to develop robust and cost-effective plans.

©Simplilearn. All rights reserved.

Why AWS?

Scalability
AWS allows you to purchase additional servers whenever you want to scale up.

Security
AWS provides the ability to remotely transfer data to another server making it safe and intact.

Collaborative operations
AWS allows the team to access, edit, and share documents which increases the collaboration.

©Simplilearn. All rights reserved.

Why Ansible with AWS?

Both Ansible and AWS have their own set of benefits which help users to secure and automate the infrastructure. Below are the reasons why Ansible integration with AWS is a boon:

©Simplilearn. All rights reserved.

Why Ansible with AWS?

AWS as group of services

- There are many services available that help in deployment and scaling of an application.
- Ansible manages AWS environment as a group of services rather than using them as a group of servers.

Dynamic inventory

- In a development environment, hosts keep spinning up and shutting down with diverse business requirements.
- Dynamic inventory automatically maps hosts based on groups specified in inventory scripts.

©Simplilearn. All rights reserved.

Why Ansible with AWS?

Ansible cloud modules

- Ansible consists of multiple AWS support modules which help in creating server-host connection and playbook execution. Some of the modules are shown below:

1. Elastic Cloud Compute (EC2)
1. Identity Access Manager (IAM)
1. Lambda
1. Simple Storage Service

Safe automation

- With the help of Ansible Tower, you can provide limited access to different users.
- This makes the automation process secure and hazel-free.

©Simplilearn. All rights reserved.

Assisted Practice
Ansible with AWS

Problem Statement:

You are given a project to create s3 bucket in AWS account.

©Simplilearn. All rights reserved.

Assisted Practice: Guidelines

Steps to perform:

1. Establish connection between both nodes
2. Generate access key
3. Create and run the playbook

©Simplilearn. All rights reserved.

Terraform

©Simplilearn. All rights reserved.

What Is Terraform?

Terraform is a tool for building and versioning infrastructure efficiently. Terraform can manage existing service provider solutions.

- Terraform generates a plan describing what it will do to build the described infrastructure.
- When configuration changes, Terraform determines the changes and creates an incremental execution plan.
- Terraform can manage low-level components such as compute instances and storage along with high-level components such as DNS entries and SaaS features.

©Simplilearn. All rights reserved.

Key Features of Terraform

Infrastructure as a code
This allows a blueprint of the datacenter to be versioned.

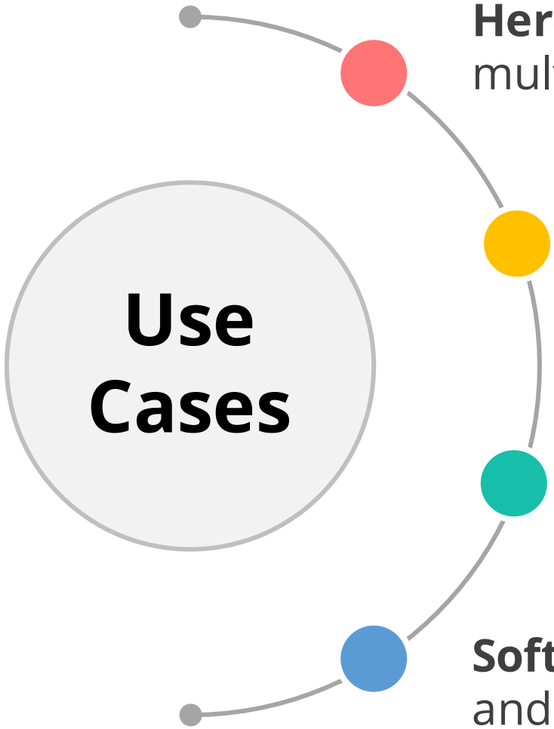
Change automation
Change sheets in Terraform provide information and sequence of what changes will be made.

Execution plans
It shows what Terraform will do when you call apply.

Resource graph
It is a graph of all resources which paralyzes any non-dependent resource.

©Simplilearn. All rights reserved.

Terraform Use Cases



Heroku app setup: It is PaaS for hosting applications. It is scalable with the help of multiple dynos or workers.

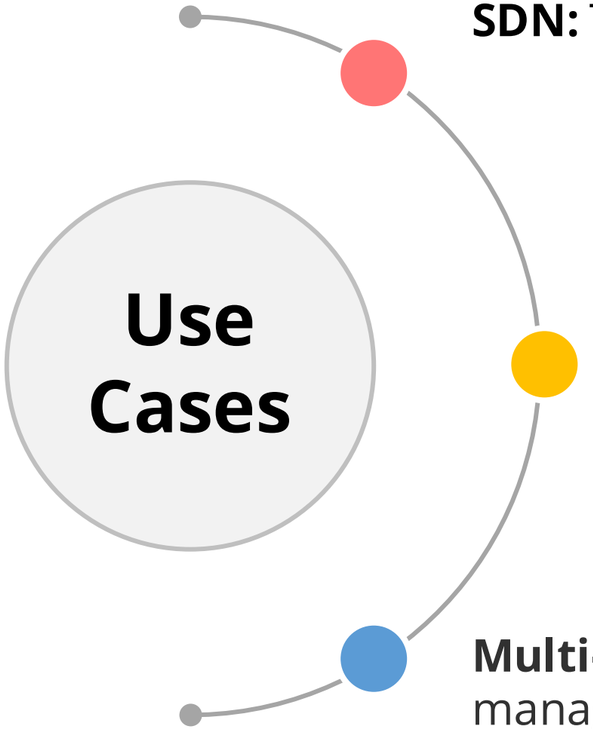
Multi-tier applications: Terraform handles multi-tier applications as a group of resources and all the dependencies are handled automatically.

Self-service cluster: Scaling and building of a service can be converted into code and the service is handled automatically.

Software demos: You can provide a Terraform configuration to create, provision, and bootstrap a demo on cloud providers like AWS.

©Simplilearn. All rights reserved.

Terraform Use Cases



SDN: Terraform can be used to codify the configuration for SDNs.

Resource schedulers: Resource schedulers allow Terraform to be used to set up infrastructure and provision the schedulers.

Multi-cloud deployment: Terraform allows single configuration to be used to manage multiple providers and cloud dependencies.

©Simplilearn. All rights reserved.

HCL

HCL stands for HashiCorp Configuration Language developed by HashiCorp. Terraform is developed using HCL.

- HCL is a structured configuration language.
- HCL is JSON compatible.
- JSON code can be a valid input in HCL.
- HCL is very similar to **libucl** and **nginx** configuration.

©Simplilearn. All rights reserved.

HCL Syntax

A high-level overview of the HCL syntax is given below:

- Single-line comments start with **#** or **//**
- Multi-line comments are wrapped in **/*** and ***/**
- Values are assigned with the syntax **key = value**.
- The value can be a string, number, boolean, object, or list.
- Strings are double-quoted.
- Multi-line strings start with **<<EOF** at the end of a line and end with **EOF** on its own line.

©Simplilearn. All rights reserved.

HCL Syntax

An example of a block written in HCL is given below:.

```
resource "aws_vpc" "main" {
  cidr_block = var.base_cidr_block
}

<BLOCK TYPE> "<BLOCK LABEL>" "<BLOCK LABEL>" {
  # Block body
  <IDENTIFIER> = <EXPRESSION> # Argument
}
```

- Identifiers can contain letters, digits, underscores, and hyphens.
- The first character of an identifier should not be a digit.
- Boolean contains either true or false.
- List is a sequence of values, like **[“abc", “xyz"]** with an index starting from 0.
- Map is a group of values identified by labels like **{name = "Mabel", age = 52}**.

©Simplilearn. All rights reserved.

Providers

©Simplilearn. All rights reserved.

Introduction to Provider

Providers are plugins in Terraform used to interact with remote systems.

- Every Terraform configuration must have a provider definition.
- It’s a good practice to have a tree of providers when using a lot of providers/plugins in a Terraform definition resource.
- Each provider adds a set of resource types and/or data sources that Terraform can manage.
- Every resource type is implemented by a provider and without them Terraform can't manage any kind of infrastructure.

©Simplilearn. All rights reserved.

Requiring Provider

Providers must be declared and nested inside the top-level **terraform** block within a **required providers** sub-block.

- A provider requirement consists of a **local name**, a **source location**, and a **version constraint.**
- Each argument in the **required providers** block enables one provider.
- The key determines the provider's unique local name.
- The value is an object with the following elements:

1. **source**: the global source address for the provider to be used.

1. **version**: the subset of available provider versions compatible with the module.

©Simplilearn. All rights reserved.

Requiring Provider

- Syntax: Provider Declaration for Terraform v0.13 -

```
terraform {
  required_providers {
    PROVIDER_NAME = {
      source = “PROVIDER_SOURCE”
      version = “PROVIDER_VERSION”
    }
  }
}
```

- Example: Provider declaration for Terraform v0.13 -

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 1.0"
    }
  }
}
```

©Simplilearn. All rights reserved.

Provider Configuration

Expressions used in the configuration arguments can only reference values known before the configuration is applied.

A provider configuration is created using a **provider** block.

The local name of the provider is in block header, also included in the **required_providers** block to configure.

Provider configurations belong in the root module of a Terraform configuration.

The body of the block (between **{** and **}**) contains configuration arguments for the provider.

©Simplilearn. All rights reserved.

Provider Configuration

- There are two **meta-arguments** which are defined by Terraform and are available for all provider blocks:

    1. **alias**: It is required for using the same provider with different configuration.

    1. **region**: It is used to set the region where AWS operations will take place.

- When Terraform needs the name of a provider configuration, it expects a reference of the form **<PROVIDER NAME>** or **<PROVIDER NAME>.<ALIAS>**

- In the next example, **aws** would refer to the provider with the **us-east-1** region, whereas, the **aws.west** would refer to the provider with the **us-west-2** region.

©Simplilearn. All rights reserved.

Provider Configuration

- Syntax: Provider Configuration for Terraform v0.13:

```
provider “LOCAL_PROVIDER_NAME” {
  region = “PROVIDER_REGION”
}

provider “LOCAL_PROVIDER_NAME” {
  alias = “ALIAS_FOR_SAME_PROVIDER_HAVING_DIFFERENT_CONFIG”
  region = “OTHER_PROVIDER_REGION”
}
```

- Example: Multi-provider configuration for Terraform v0.13:

```
# The default provider configuration
provider "aws" {
  region = "us-east-1"
}

# Additional provider configuration for west coast region
provider "aws"
  alias  = "west"
  region = "us-west-2"
}
```

©Simplilearn. All rights reserved.

Variations and Interpolation Syntax

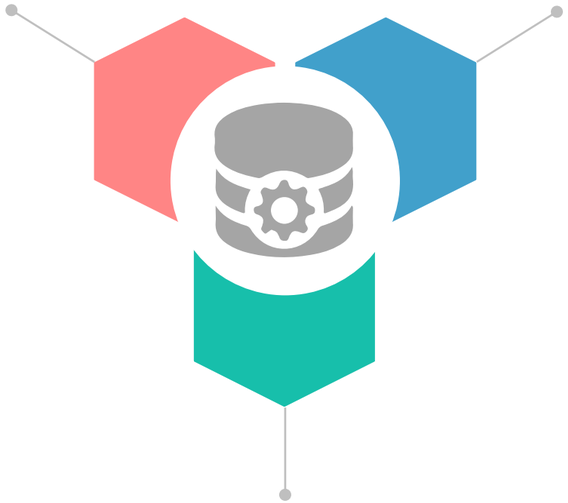
©Simplilearn. All rights reserved.

Variables

Input Variable

Serves as parameter for a Terraform module and can be customized without editing the source.

They are like function arguments.



Output Variable

Returns values for a Terraform module.

They are like function return values.

Local Variable

Convenience feature for assigning a short name to an expression.

They are like a function's temporary local variables.

©Simplilearn. All rights reserved.

Declaring Variables

- Each input variable accepted by a module must be declared using a **variable** block.
- The variable name must be unique among all variables in the same module.
- The name of a variable can be any valid identifier except **source**, **version**, **providers**, **count**, **for_each**, **lifecycle**, **depends_on**, and **locals**.
- The above names are reserved for meta-arguments in **module configuration** blocks, and cannot be declared as variable names.

©Simplilearn. All rights reserved.

Declaring Variables

- Terraform CLI defines the following optional arguments for variable declarations:
  - **default**: Default value which then makes the variable optional.
  - **type**: Specifies what value types are accepted for the variable.
  - **description**: Specifies the input variable's documentation.
  - **validation**: Block to define validation rules, usually in addition to type constraints.
  - **sensitive**: Limits Terraform UI output when the variable is used in configuration.

©Simplilearn. All rights reserved.

Declaring Variables

- When defining the variables without any default value, the Terraform CLI will ask the value in run-time. However, in production environment we create a **tfvars** file.
- The **tfvars** files are basically files specific to defining values to terraform variables.

©Simplilearn. All rights reserved.

Defining Variable Data type

- The keyword **any** may be used to indicate that any type is acceptable.

- The **type** constraints are created from a mixture of **type** keywords and **type** constructors. The supported type keywords are:

1. string
2. number
3. bool

- The **type** constructors also allow you to specify complex types such as collections :

1. list(<TYPE>)
2. set(<TYPE>)
3. map(<TYPE>)
4. object({<ATTR NAME> = <TYPE>, ... })
5. tuple([<TYPE>, ...])

©Simplilearn. All rights reserved.

Defining Variable Data type

Example: Creating different types of variables in Terraform v0.13 :

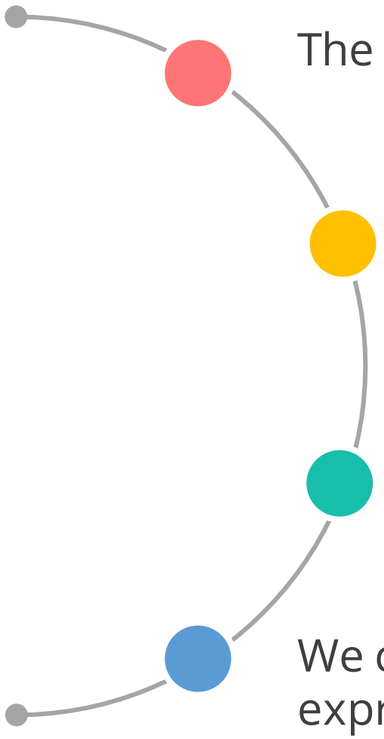
```
variable "student_name" {
  type = string
}

variable "bucket_name" {
  type = string
  default = "random-bucket-var-simpli"
  description = "Bucket name for S3"
}

variable "students_in_class" {
  type    = list(string)
  default = ["Ron", "Henry", "Adam"]
}
```

©Simplilearn. All rights reserved.

Interpolation Syntax



The expressions in terraform basically constitute the interpolation syntax.

Interpolation Syntax is the way how we interact with other terraform data structure definitions.

Terraform does the type conversation and evaluates expressions and other elements that are defined in the interpolation syntax.

We can define a combination of constants and variable names to evaluate expressions in terraform.

©Simplilearn. All rights reserved.

Interpolation Syntax

- Example 1: Adding a prefix to a bucket name -

```
resource "aws_s3_bucket" "variable_s3_bucket" {
  bucket = "${var.bucket_name}-testing"
}
```

- Example 2: Check if a variable is empty or has a default value to assign the name using ternary operator:

```
resource "aws_s3_bucket" "variable_s3_bucket" {
  bucket = var.bucket_name == "" ? "testing-simpli-s3-bucket" : var.bucket_name
}
```

©Simplilearn. All rights reserved.

Terraform Commands

Command	Description
init	Initialize a working directory containing terraform configuration file
validate	Check if the configuration is valid
plan	Show changes required
apply	Create or update infrastructure
destroy	Destroy previously created infrastructure

Option	Description
-chdir=DIR	Switch to a different working directory before executing the given subcommand
-help	Show this help output, or the help for a specified subcommand.
-version	An alias for the "version" subcommand

- For more commands used in Terraform, refer the link given below:
  https://www.terraform.io/docs/cli/commands/index.html

©Simplilearn. All rights reserved.

Terraform Workflow

Workflow of Terraform deployment is given below:

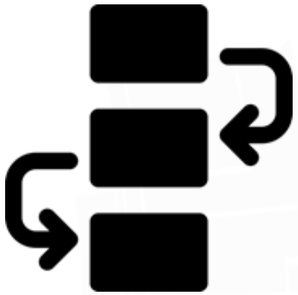
1. Scope – Confirm the resources to be created

1. Author - Create the configuration file in HCL

1. Initialize - Run **terraform init** in the project directory with the configuration files to download providers and plugins

1. Plan - Run **terraform plan** to verify creation process

1. Apply – Run **terraform apply** to create resources and state file. This command also compares the changes made in configuration files to the current configurations of the deployment environment.



©Simplilearn. All rights reserved.

Advantages of Terraform

1

Platform agnostic
With Terraform, you can manage a hybrid environment with the same workflow by creating a configuration file.

2

State management
Terraform refers the state file while making any changes and decides what changes need to be made in the current configuration.

3

Operator confidence
Upon **terraform apply**, the user can review the proposed changes and only after the user's confirmation, Terraform will apply them.

©Simplilearn. All rights reserved.

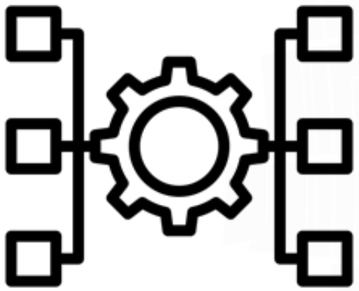
Terraform Service Providers

A provider is responsible for understanding API interactions and exposing resources.

- Providers generally are an IaaS service such as AWS, PaaS service such as Heroku, or SaaS services such as Terraform Cloud and Cloudflare.

- Any infrastructure type can be represented as a resource in Terraform. Some of the popular providers in Terraform are shown below:

1. GCP
1. Oracle cloud
1. AWS
1. Azure
1. PostgreSQL
1. Chef
1. Kubernetes



©Simplilearn. All rights reserved.

Assisted Practice
Set up Terraform

Problem Statement:

You are given a project to configure Terraform in Linux.

©Simplilearn. All rights reserved.

Assisted Practice: Guidelines

Steps to perform:

1. Download the appropriate package from Terraform website
2. Add the binary file into the **bin** directory

©Simplilearn. All rights reserved.

Assisted Practice
First Terraform Deployment

Problem Statement:

You are given a project to create an S3 bucket using Terraform.

©Simplilearn. All rights reserved.

Assisted Practice: Guidelines

Steps to perform:

1. Set up Git repo
2. Create Terraform execution plan

©Simplilearn. All rights reserved.

Assisted Practice

S3 Bucket Creation Using Variables

Problem Statement:

You are given a project to create multiple S3 buckets using different Variables.

©Simplilearn. All rights reserved.

Assisted Practice: Guidelines

Steps to perform:

1. Create credential file
2. Create variables file
3. Create **terraform.tfvars** file
4. Create **demo.tf** scripts
5. Execute Terraform code

©Simplilearn. All rights reserved.

Terraform with Jenkins

©Simplilearn. All rights reserved.

Assisted Practice
Terraform with Jenkins

Problem Statement:

You are given a project to integrate Terraform with Jenkins.

©Simplilearn. All rights reserved.

Assisted Practice: Guidelines

Steps to perform:

1. Set up Jenkins
2. Install and configure Terraform plugin
3. Create and run Terraform project

©Simplilearn. All rights reserved.

Assisted Practice
Provisioning EC2 with Ansible

Problem Statement:

You are given a project to provision EC2 instance using Ansible.

©Simplilearn. All rights reserved.

Assisted Practice: Guidelines

Steps to perform:

1. Establish connection between both nodes
2. Generate access key
3. Create and run the playbook

©Simplilearn. All rights reserved.

Assisted Practice
Ansible with Jenkins

Problem Statement:

You are given a project to run Jenkins job using Ansible playbook

©Simplilearn. All rights reserved.

Assisted Practice: Guidelines

Steps to perform:

1. Logging into Jenkins and getting the API key
2. Creating and running an Ansible playbook

©Simplilearn. All rights reserved.

Key Takeaways

- Dynamic inventory automatically maps hosts based on groups specified in inventory scripts.
- Terraform generates a plan describing what it will do to build the described infrastructure.
- Infrastructure as a code allows a blueprint of the data center to be versioned.
- Majority of the infrastructure type can be represented as code in Terraform.

©Simplilearn. All rights reserved.