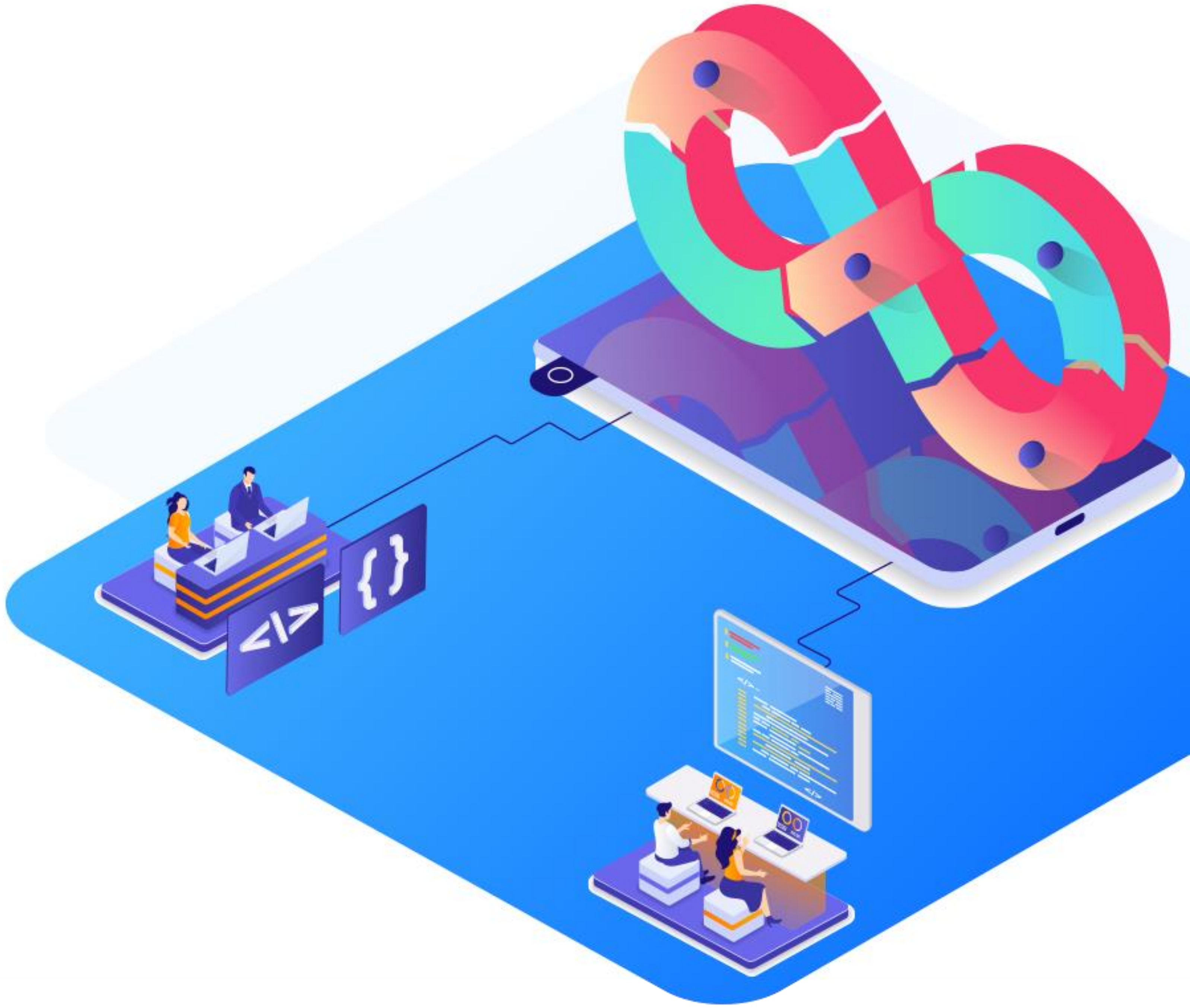
Configuration Management with Ansible and Terraform



Terraform Basics and Workflow

Learning Objectives

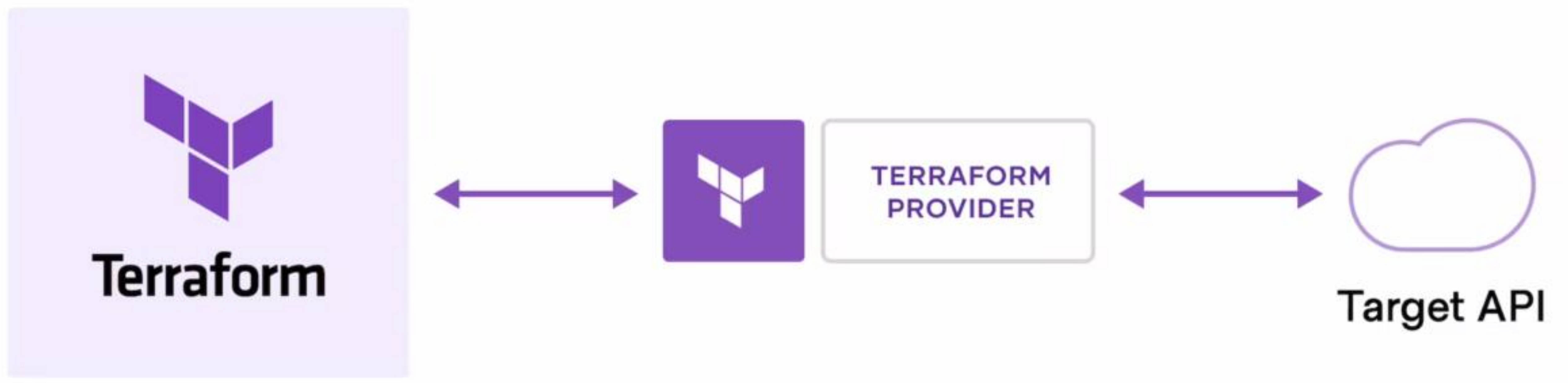
By the end of this lesson, you will be able to:

- Utilize multiple Terraform providers and provisioners for infrastructure automation and management
- Develop a custom Terraform module for AWS EC2 instances, including configuration and TLS certificates
- Implement local-exec and remote-exec provisioners in Terraform configurations to automate post-deployment tasks
- Debug Terraform configurations to resolve errors
- Execute the core workflow of Terraform from initialization to destruction, identifying key steps and best practices

Terraform Providers

What Are Terraform Providers?

They are plugins that enable Terraform to manage resources across various platforms, including the cloud, container tools, databases, and other APIs.



Source: https://developer.hashicorp.com/terraform/language/providers

Terraform Providers

Terraform supports a wide range of providers. Some examples include:

Azure

AWS

Kubernetes

Google Cloud

Alibaba Cloud

Oracle Cloud Infrastructure

VMware vSphere

What Providers Do

The functions of Terraform providers are given below:

01 They provide resource types and data sources for managing infrastructure.

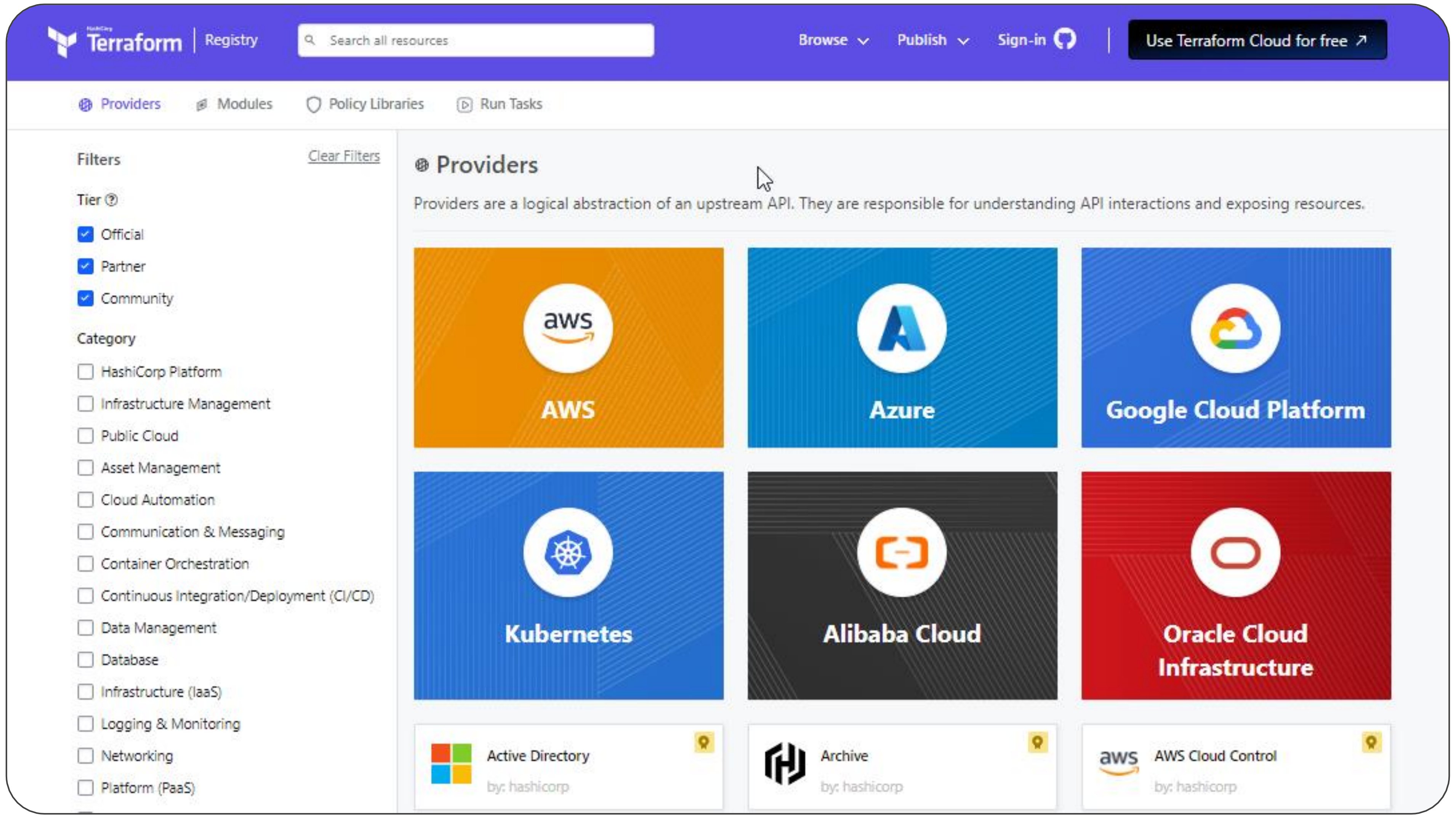
02 They implement each resource type, making them essential for Terraform's functionality.

03 They are typically tailored to specific platforms, accommodating various cloud or self-hosted environments.

04 They can also include local utilities, such as generating random numbers for unique resource names.

Terraform Providers Registry

The Terraform registry serves as the primary repository for publicly accessible Terraform providers, offering support for most infrastructure platforms.



Providers

Providers are a logical abstraction of an upstream API. They are responsible for understanding API interactions and exposing resources.

Providers are released separately from Terraform, each with its own schedule and version number.

Configuring Terraform Providers

Terraform uses providers to interact with remote systems. Configurations must declare required providers in a block for installation and use.

The command in the following example installs the Terraform AWS provider:

```
terraform {
  required_version = ">= 1.0.0"
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "~> 3.0"
    }
  }
}
```

Here, **version = ">= 1.0.0“** is a version constraint string that includes one or more conditions, each with an operator and version number, separated by commas.

Initializing Terraform Providers

After providing the Terraform provider installation configuration, execute the **init** command to recognize the configuration update and download the specified provider.

Example:

```
terraform init
Initializing the backend...


Initializing provider plugins...
- Reusing previous version of hashicorp/random from the dependency lock file
- Reusing previous version of hashicorp/aws from the dependency lock file
- Using previously-installed hashicorp/aws v3.62.0
- Using previously-installed hashicorp/random v3.1.0

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
```

Running Multiple Terraform Providers

Running multiple Terraform providers in a single project allows you to interact with different types of infrastructure within the same configuration. Some of the advantages are given below:

- Improved reliability and fault tolerance
- Enhanced security and compliance
- Cost optimization
- Innovation and flexibility
- Performance optimization
- Strategic benefits like market expansion

Using Multiple Terraform Providers

Terraform's plug-in architecture makes it easy to use multiple providers in one configuration. The following steps demonstrate how to install multiple providers within a single Terraform configuration file:

01 Install Terraform HTTP provider

02 Verify the installation of the Terraform HTTP provider

03 Install Terraform Random provider

04 Verify the installation of the Terraform Random provider

05 Install Terraform Local provider

Installing Terraform HTTP Provider

The HTTP provider interacts with HTTP servers in a Terraform configuration for sending requests and retrieving data, enabling automation with web-based APIs.

Step 1

Add the following script to configure the HTTP provider in the **terraform.tf** file:

```
terraform {
  required_providers {
    http = {
      source = "hashicorp/http"
      version = "2.1.0"
    }
  }
}
```

Installing Terraform HTTP Provider.

The Terraform **init** command initializes the working directory and downloads the necessary plugins or providers specified in the configuration.

Step 2

To install the HTTP provider, execute the given command:

```
terraform init
```

Verify Terraform HTTP Provider

Step 3 Verify the installation of the HTTP provider using the following command:

```
terraform version
Terraform v1.0.8
on linux_amd64
+ provider registry.terraform.io/hashicorp/aws v3.62.0
+ provider registry.terraform.io/hashicorp/http v2.1.0
+ provider registry.terraform.io/hashicorp/random v3.0.0
```

Installing Terraform Random Provider

The Terraform Random provider generates random values, such as strings and integers, which can be used in Terraform configurations to help with unique resource identification and testing.

Step 4

Add the following script to configure the Random provider in the **terraform.tf** file:

```
terraform {
  required_version = ">= 1.0.0"
  required_providers {
  random = {
     source = "hashicorp/random"
     version = "3.1.0"
    }
  }
}
```

Verify Terraform Random Provider

Step 5

Execute the following commands to initialize and verify the Terraform Random provider installation:

```
terraform init –upgrade
terraform version
Terraform v1.0.8
on linux_amd64
+ provider registry.terraform.io/hashicorp/aws v3.62.0
+ provider registry.terraform.io/hashicorp/http v2.1.0
+ provider registry.terraform.io/hashicorp/random v3.1.0
```

Installing Terraform Local Provider

Step 6

The Local provider is utilized for managing local resources, such as files. The command for its installation is given below:

```
terraform {
  required_version = ">= 1.0.0"
  required_providers {
    aws = {
      source = "hashicorp/aws"
    }
    http = {
      source  = "hashicorp/http"
      version = "2.1.0"
    }
    random = {
      source  = "hashicorp/random"
      version = "3.1.0"
    }
    local = {
      source  = "hashicorp/local"
      version = "2.1.0"
    }
  }
}
```

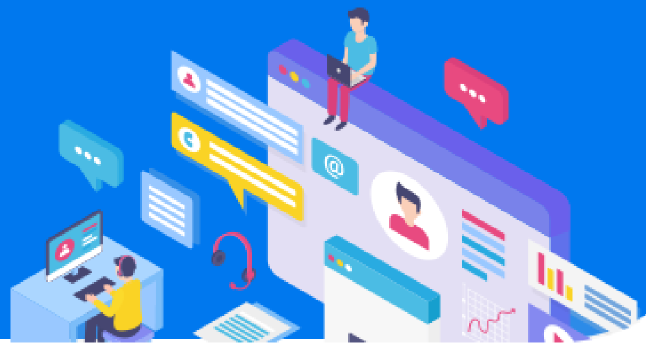
Verify Terraform Local Provider

Step 7

Execute the following commands to initialize and verify the Terraform Local provider installation:

```
terraform init –upgrade
terraform version
Terraform v1.0.8
on linux_amd64
+ provider registry.terraform.io/hashicorp/aws v3.62.0
+ provider registry.terraform.io/hashicorp/http v2.1.0
+ provider registry.terraform.io/hashicorp/local v2.1.0
+ provider registry.terraform.io/hashicorp/random v3.1.0
```

Assisted Practice



Configuring AWS and Random providers in Terraform **Duration: 10 Min.**

Problem statement:

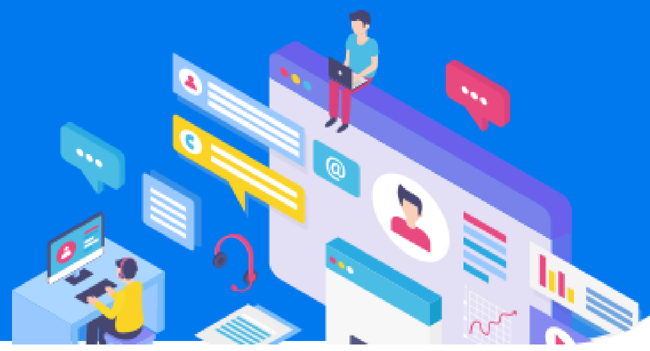
You have been assigned a task to configure the AWS and random Terraform providers in the Terraform configuration file for managing cloud infrastructure efficiently.

Outcome:

By completing this demo, you will have a Terraform configuration file that integrates AWS and random providers, enabling efficient and consistent management of cloud infrastructure.

Note: Refer to the demo document for detailed steps: 01_Configuring_AWS_and_Random_Providers_in_Terraform

Assisted Practice: Guidelines



Steps to be followed:

1. Install the Terraform extension in VS Code
2. Create the main Terraform configuration file
3. Configure the Terraform providers
4. Initialize the Terraform configuration file

Running Multiple Cloud Provider Configurations in Terraform

Terraform allows the use of multiple configurations for the same provider by selecting a provider configuration on a per-resource or per-module basis.

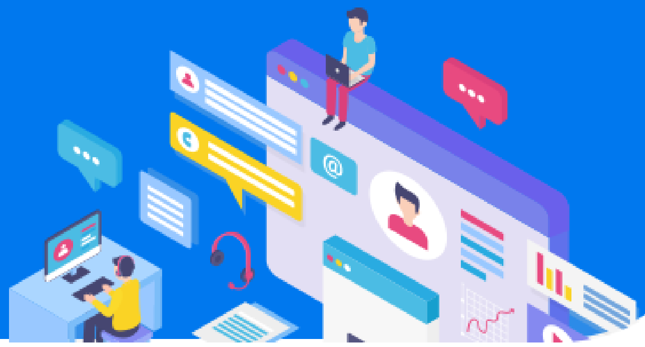
Example:

```
# The default provider configuration; resources that begin with `aws_` will use
# it as the default, and it can be referenced as `aws`.
provider "aws" {
  region = "us-east-1"
}

# Additional provider configuration for west coast region; resources can
# reference this as `aws.west`.
provider "aws" {
  alias  = "west"
  region = "us-west-2"
}
```

The main use is to support multiple regions for a cloud platform; other uses include targeting multiple Docker hosts, multiple Consul hosts, and more.

Assisted Practice



Running Multiple Cloud Provider Configurations **Duration: 10 Min.**

Problem statement:

You have been assigned a task to demonstrate the use of multiple AWS provider configurations in Terraform for managing resources across different regions, showcasing the ability to define, configure, and deploy resources in various regions within a single Terraform configuration.

Outcome:

By completing this task, you will demonstrate the use of multiple AWS provider configurations in Terraform to manage resources across different regions. You will showcase the ability to define, configure, and deploy resources in various regions within a single Terraform configuration.

Note: Refer to the demo document for detailed steps:
02_Running_Multiple_Cloud_Provider_Configurations

Assisted Practice: Guidelines

Steps to be followed:

1. Define providers with aliases in the main.tf file
2. Specify provider configurations for each resource in the main.tf file
3. Initialize and apply the configuration

What Is the TLS Provider?

TLS

- It stands for Transport Layer Security, which offers tools for managing security keys and certificates.
- It includes resources for creating private keys, certificates, and certificate requests as part of a Terraform deployment.

TLS vs SSL

- Another name for Transport Layer Security (TLS) is Secure Sockets Layer (SSL). TLS and SSL are equivalent regarding the resources managed by this provider.

Generating an SSH Key Using the TLS Provider

The steps to generate an SSH Key using the Terraform TLS Provider are given below:

> Check Terraform version 01

> Install Terraform TLS Provider 02

> Create a self-signed certificate with TLS Provider 03

Check Terraform Version

Step 1

Execute the following command to check the Terraform version:

```
terraform -version
```

The output is as shown below:

```
Terraform v1.0.10
```

Install Terraform TLS Provider

Step 2 Edit the file titled **terraform.tf** to add the TLS provider configuration:

```
terraform {
  required_version = ">= 1.0.0"
  required_providers {
  tls = {
     source  = "hashicorp/tls"
     version = "3.1.0"
    }
  }
}
```

Install Terraform TLS Provider

Step 3 Execute the following command to initialize the Terraform file:

```
terraform init
```

Step 4 Validate the installation by running a Terraform version command:

```
terraform version
Terraform v1.0.10
on linux_amd64
+ provider registry.terraform.io/hashicorp/aws v3.62.0
+ provider registry.terraform.io/hashicorp/http v2.1.0
+ provider registry.terraform.io/hashicorp/local v2.1.0
+ provider registry.terraform.io/hashicorp/random v3.1.0
+ provider registry.terraform.io/hashicorp/tls v3.1.0
```

Create a Self-Signed Certificate with TLS Provider

Step 5

Update the **main.tf** file to generate a **TLS** self-signed certificate and save the private key locally:

```
resource "tls_private_key" "generated" {
  algorithm = "RSA"
}

resource "local_file" "private_key_pem" {
  content  = tls_private_key.generated.private_key_pem
  filename = "MyAWSKey.pem"
}
```

Create a self-signed certificate with TLS Provider

Step 6

Create the Keypair via Terraform by using the following command:

```
terraform apply
```

The output is as shown below:

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

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value: yes
Apply complete! Resources: 2 added, 0 changed, 0 destroyed.
```

Assisted Practice

Generating SSH Key with Terraform TLS Provider **Duration: 10 Min.**

Problem statement:

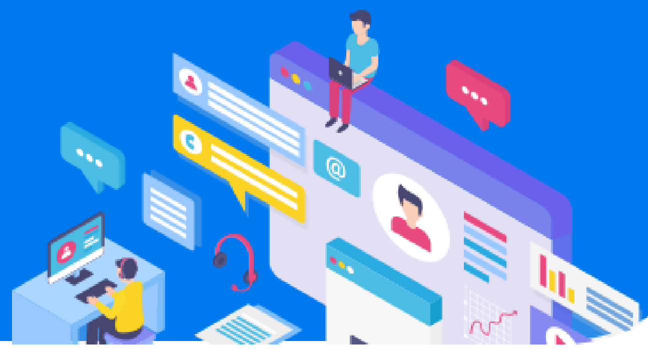
You have been assigned a task to generate an SSH key with the Terraform TLS provider using the latest version and configurations for secure access to your cloud infrastructure.

Outcome:

By completing this task, you will generate an SSH key using the Terraform TLS provider with the latest version and configurations, ensuring secure access to your cloud infrastructure.

Note: Refer to the demo document for detailed steps: 03_Generating_SSH_Key_with_Terraform_TLS_Provider

Assisted Practice: Guidelines



Steps to be followed:

1. Check the version of the installed providers
2. Add the Terraform TLS provider configuration in the main.tf file
3. Initialize the Terraform configuration file
4. Update the main.tf file with a TLS certificate configuration block
5. Execute the apply command and validate the SSH key

Upgrading Terraform Providers

It refers to the process of updating provider plugins to their latest versions to maintain compatibility with cloud services and improve infrastructure reliability and security.

There are two ways to upgrade Terraform providers:

Manual

You can manually update the provider versions in the Terraform configuration file.

Upgrade command

You can do this by running the command:

terraform init -upgrade

Upgrading Terraform Providers Manually

Use the following steps to upgrade Terraform providers manually:

01 Check the existing version of the Terraform providers

02 Specify the upgraded Terraform providers version in the Terraform configuration file

03 Initialize the Terraform configuration file to upgrade the provider version

By upgrading the provider's version manually you can upgrade any specific provider to the required version.

Upgrading Provider Versions Using a Command

The command to upgrade the provider version is given below:

```
terraform init -upgrade
```

Run **terraform version** to ensure the new provider version is installed and ready:

```
terraform version
Terraform v1.0.10

+ provider registry.terraform.io/hashicorp/aws v3.61.0
+ provider registry.terraform.io/hashicorp/random v3.1.0
```

Quick Check

You manage a multi-cloud infrastructure with Terraform. You recently learned about the TLS provider and how to generate SSH keys with it. Which command would you use to generate a new SSH key pair with the TLS provider in Terraform?

A. terraform tls -generate-ssh-key

B. terraform apply tls_key_pair

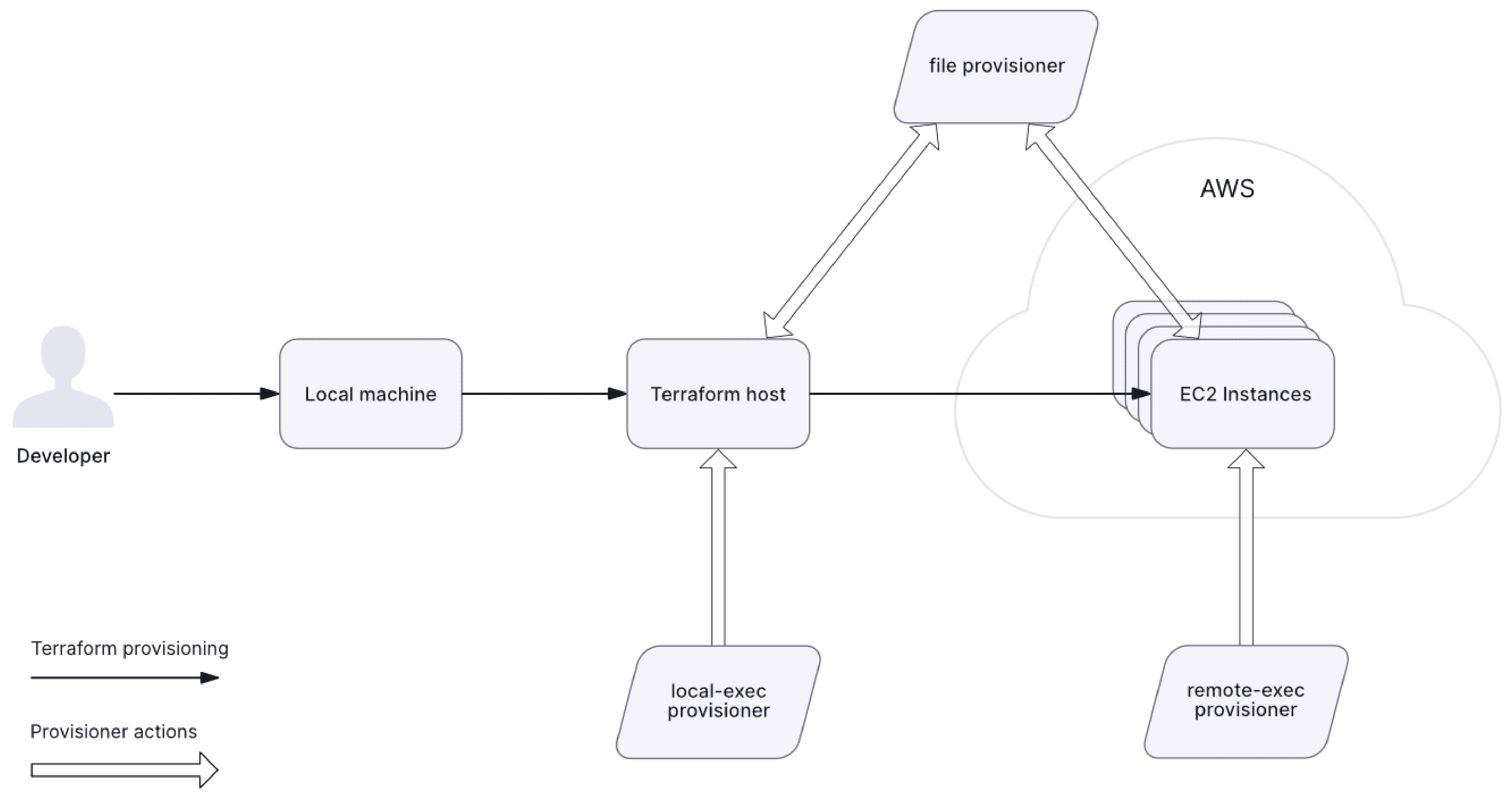
C. tls_private_key resource

D. resource "tls_private_key"

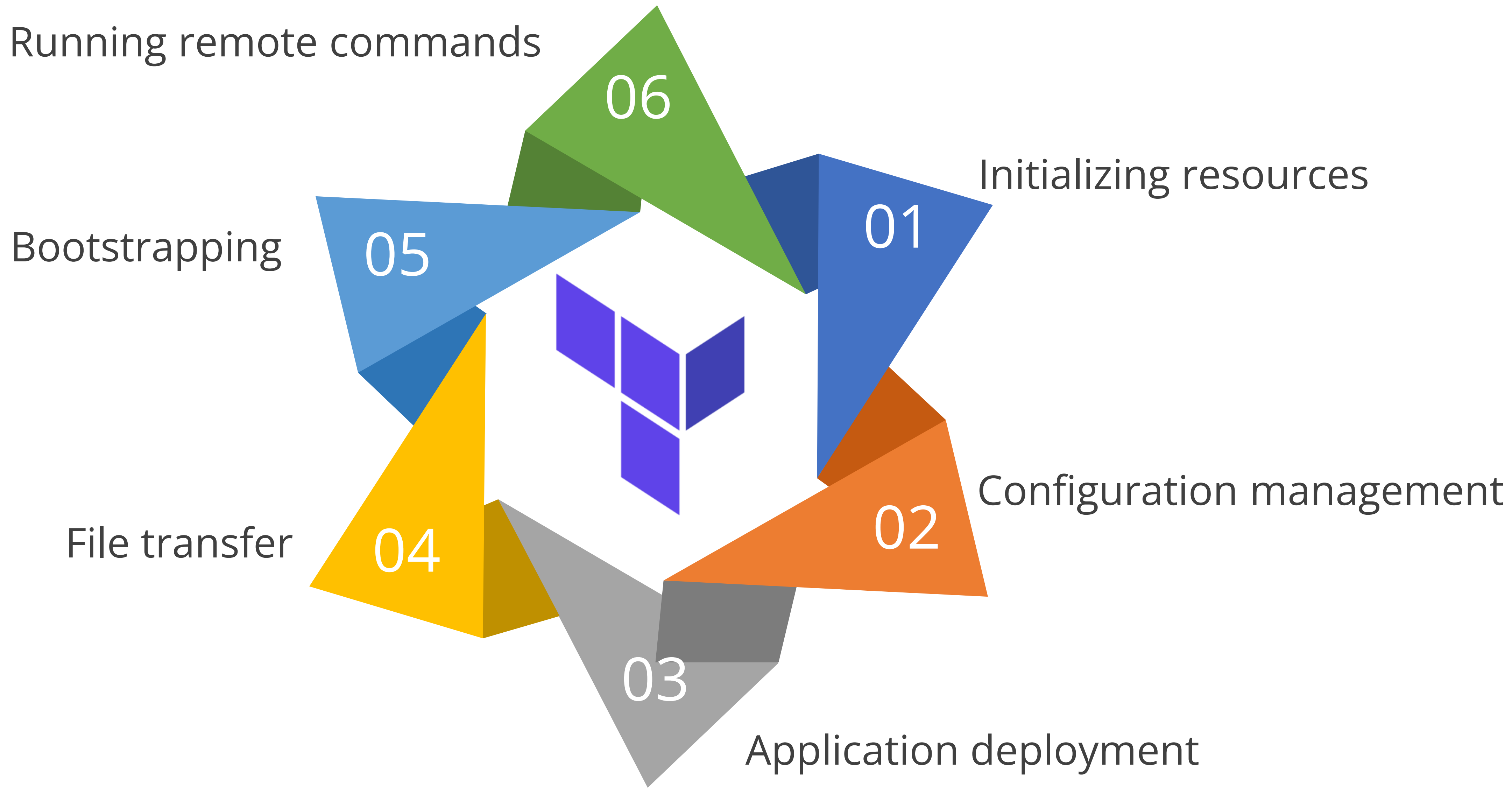
Terraform Provisioners

What Are Terraform Provisioners?

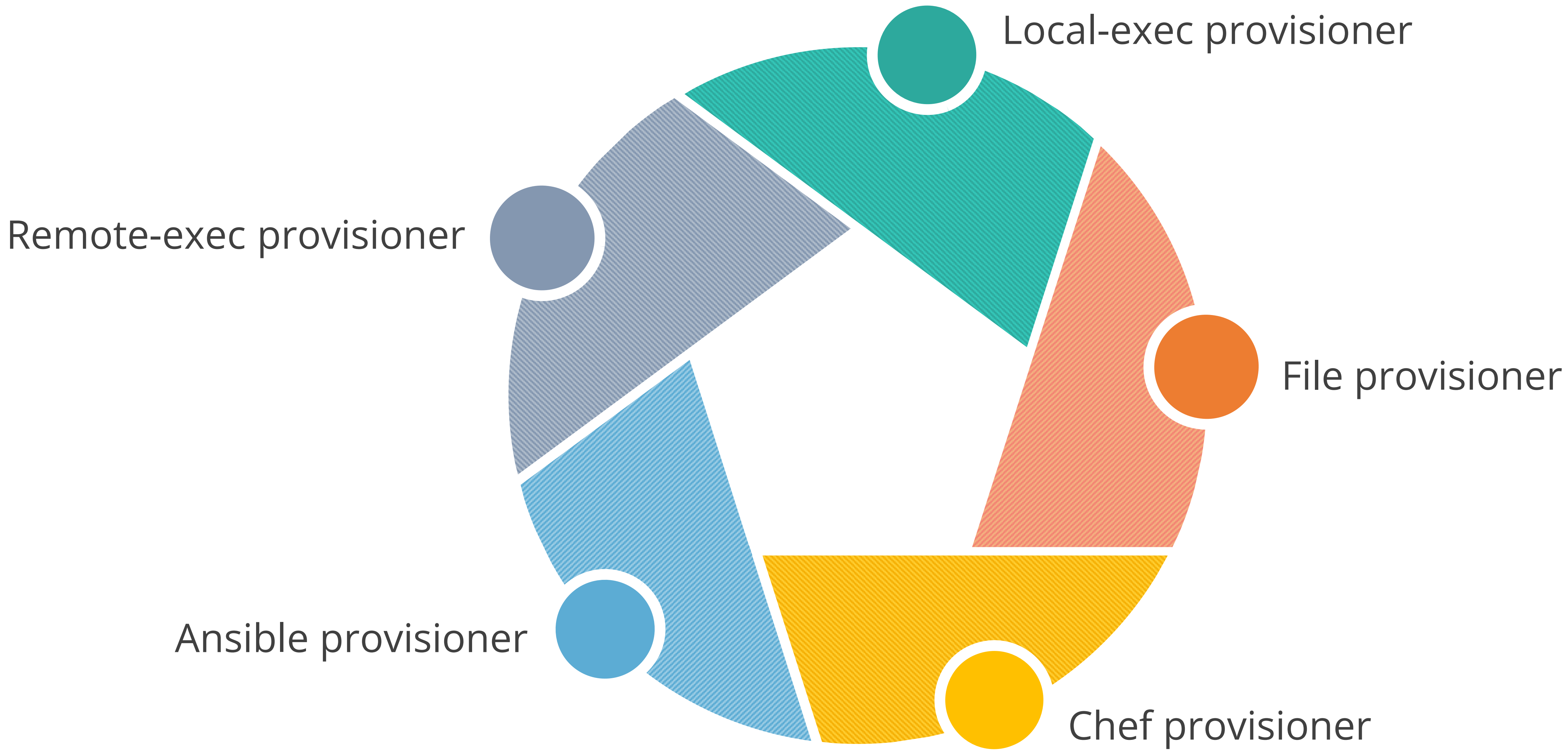
They are components within **HashiCorp Terraform** that enable the execution of scripts or commands on a local or remote machine at certain stages of the resource lifecycle.



Uses of Terraform Provisioners



Types of Provisioners in Terraform



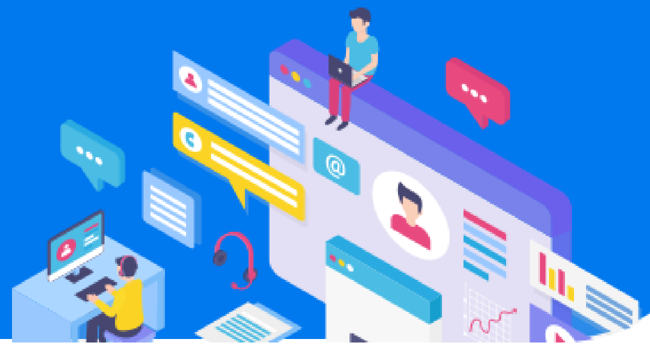
Local-Exec Provisioner

It executes a command on the machine running Terraform. It is useful for tasks like setting up local configuration files with scripts.

The code snippet below uses the **local-exec** provisioner to execute the command **echo 'Hello, Terraform!'** on the local machine after the associated resource is created:

```
provisioner "local-exec" {
  command = "echo 'Hello, Terraform!'"
}
```

Assisted Practice



Implementing Local Exec Provisioners **Duration: 10 Min.**

Problem statement:

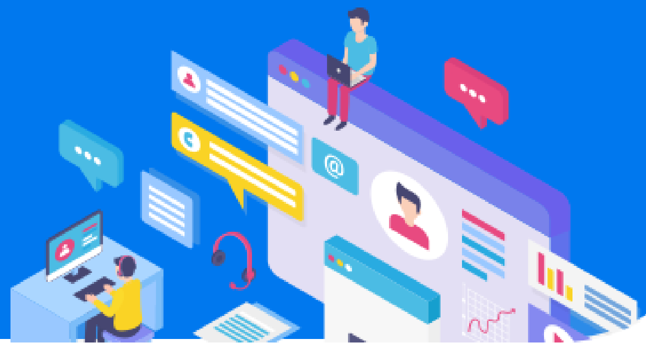
You have been assigned a task to implement local-exec provisioners for executing commands on the machine running Terraform during resource provisioning.

Outcome:

By completing this task, you will implement local-exec provisioners in Terraform, enabling the execution of commands on the machine running Terraform during resource provisioning. This will facilitate automation and enhance the provisioning process.

Note: Refer to the demo document for detailed steps:
04_Implementing_Local_Exec_Provisioners

Assisted Practice: Guidelines



Steps to be followed:

1. Implement local-exec provisioners
2. Create the local script
3. Initialize and verify the configuration

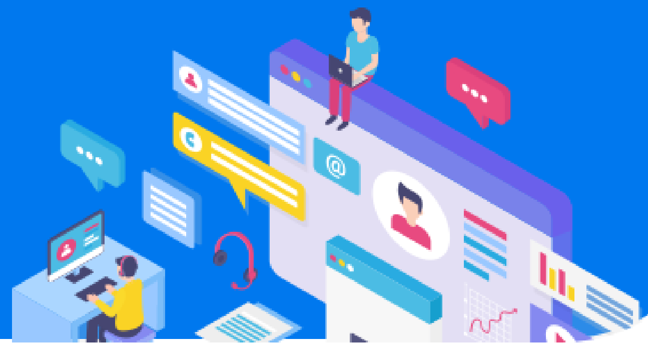
Remote-Exec Provisioner

It runs commands on a remote machine via SSH or WinRM. It is useful for tasks like installing software or configuring services on new resources.

The below code snippet uses the **remote-exec** provisioner to run commands on a remote machine:

```
provisioner "remote-exec" {
  inline = [
} "sudo apt-get update",
    "sudo apt-get install -y nginx"
  ]
```

Assisted Practice



Implementing Remote Exec Provisioners **Duration: 10 Min.**

Problem statement:

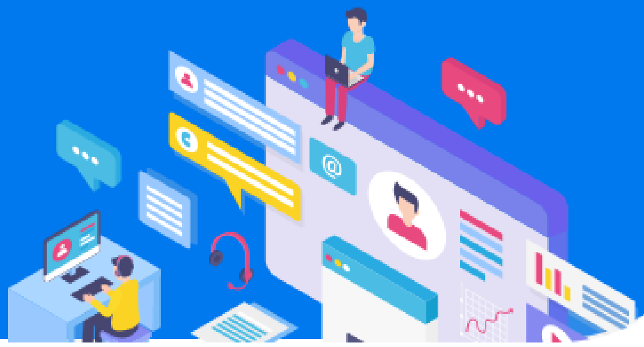
You have been assigned a task to implement the remote-exec provisioners in Terraform to automate the setup of an AWS EC2 instance for efficient and consistent deployment across multiple environments

Outcome:

By completing this task, you will implement remote-exec provisioners in Terraform to automate the setup of AWS EC2 instances, ensuring efficient and consistent deployment across multiple environments.

Note: Refer to the demo document for detailed steps:
05_Implementing_Remote_Exec_Provisioners

Assisted Practice: Guidelines



Steps to be followed:

1. Implement remote-exec provisioners in Terraform
2. Deploy EC2 instances from AWS console

File Provisioner

It copies files or directories from the Terraform machine to a new resource. It is commonly used for transferring configuration files or scripts.

The following code snippet uses the **file** provisioner to copy a local file named **config.ini** to the remote destination **/etc/config.ini** on the target resource:

```
provisioner "file" {
  source      = "config.ini"
  destination = "/etc/config.ini"
}
```

Chef Provisioner

It is used to automate the configuration management of the infrastructure using Chef. Terraform integrates with Chef to run recipes or cookbooks on managed instances.

The given code snippet uses the **Chef** provisioner in Terraform to configure a remote resource:

```
provisioner "chef" {
  version = "~> 15.0"
  server_url = "https://chef.example.com"
  node_name = "node-01"
  run_list = ["recipe[my_cookbook]"]
}
```

Ansible Provisioner

It automates the configuration management of the infrastructure with Ansible playbooks.

The given code snippet uses the **Ansible** provisioner in Terraform to apply an Ansible playbook to configure a remote resource:

```
provisioner "ansible" {
  playbook_file = "playbook.yml"
}
```

Quick Check

You are deploying an application that requires placing configuration files, running local and remote scripts, and using Chef and Ansible for management. Which provisioner should you use to run a script on the target machine without communicating with other machines?

A. File provisioner

B. Chef provisioner

C. Local-Exec provisioner

D. Ansible provisioner

Formatting and Taint

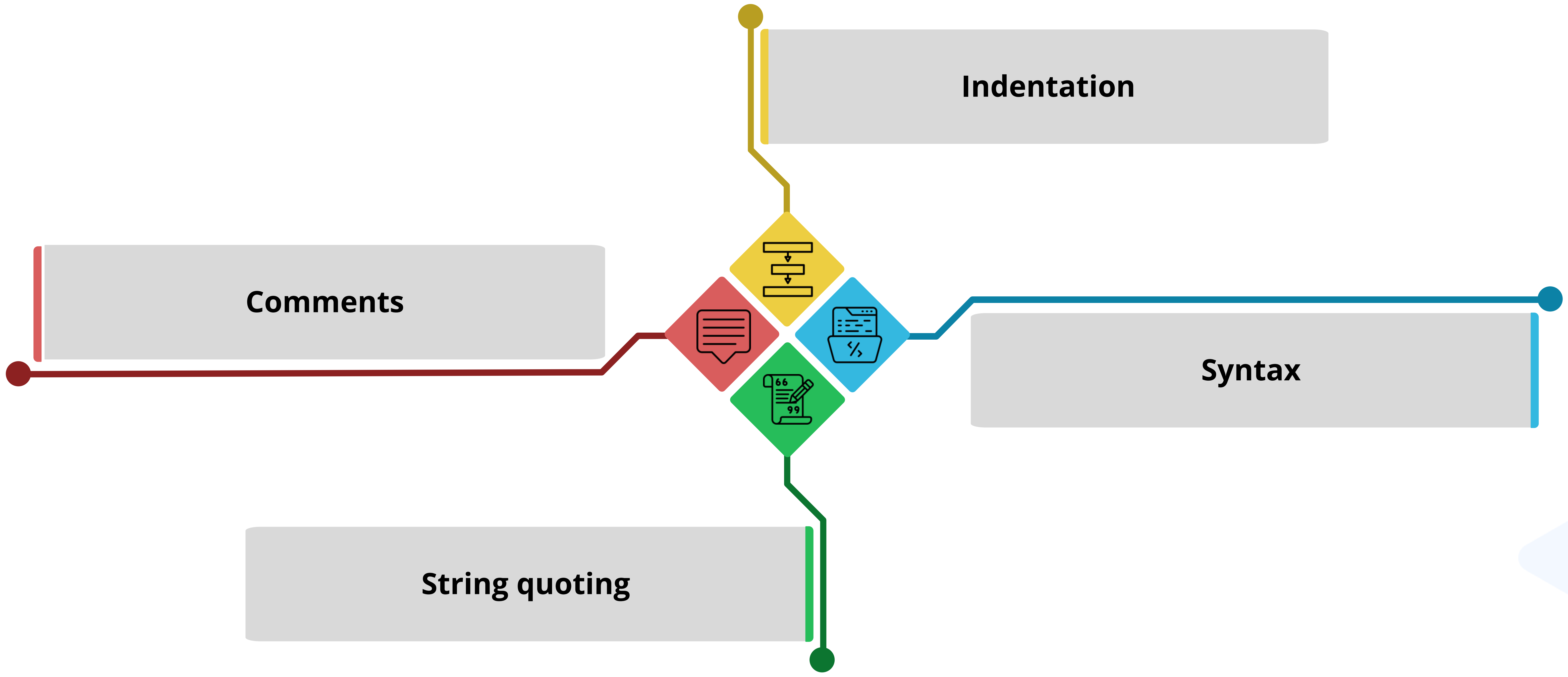
What Is Formatting in Terraform?

It involves structuring your configuration files using **HashiCorp Configuration Language (HCL)** to ensure they are readable, maintainable, and correctly understood by Terraform.

HashiCorp

Formatting in Terraform

The following aspects are checked while formatting a Terraform file:



Taint

The **terraform taint** command marks a specific object as tainted in the Terraform state, indicating it is degraded or damaged. This prompts Terraform to propose its replacement in the next plan.

You can taint a resource using the terraform taint command followed by the resource identifier. For example:

```
terraform taint aws_instance.my_instance
```

Warning: This command is deprecated. For Terraform **v0.15.2** and later, use the **-replace** option with **terraform apply** instead.

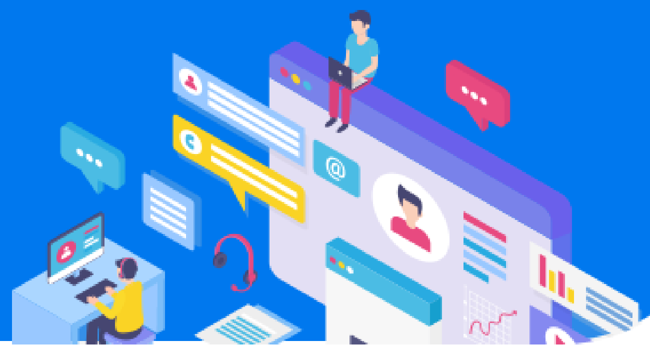
Recommended Alternative

For Terraform **v0.15.2** and later, use the **-replace** option with **terraform apply** to force object replacement without configuration changes.

The command to apply this is given below:

```
terraform apply -replace="aws_instance.example[0]"
```

Assisted Practice



Formatting Terraform Code **Duration: 10 Min.**

Problem statement:

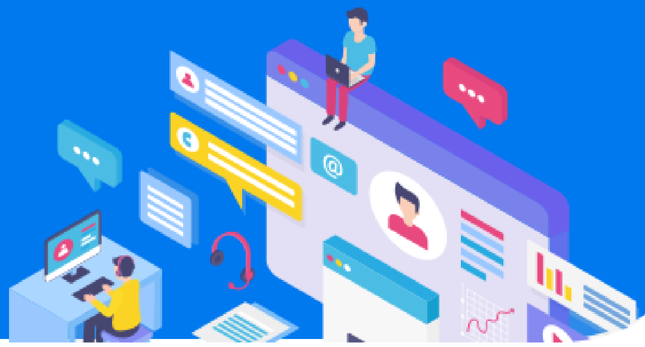
You have been assigned a task to format Terraform code in canonical format and style based on a subset of Terraform language style conventions for improved readability, maintainability, and consistency across your infrastructure as code projects.

Outcome:

By completing this task, you will format Terraform code in the canonical format and style based on a subset of Terraform language style conventions, improving readability, maintainability, and consistency across your infrastructure as code projects.

Note: Refer to the demo document for detailed steps:
06_Formatting_Terraform_Code

Assisted Practice: Guidelines



Steps to be followed:

1. Update the unformatted code in the main.tf file
2. Format the provided code using the fmt command

Quick Check

You have noticed that the formatting of your Terraform configuration files is inconsistent, making them hard to read and maintain. Additionally, you need to force Terraform to recreate a specific resource due to a detected issue. Which commands would you use to address these needs?

A. terraform apply and terraform plan

B. terraform fmt and terraform taint

C. terraform validate and terraform destroy

D. terraform init and terraform refresh

Workspaces and CLI (Command Line Interface)

Terraform Workspaces

Terraform workspaces help organize infrastructure by environments and variables within a single directory. Given below is an overview of how Terraform workspace function:

- **Environment management:** Manage multiple environments (for example, dev, staging, production) within the same configuration
- **State isolation:** Maintain isolated states, preventing changes in one from affecting another
- **Easy switching:** Facilitate easy switching between workspaces
- **Consistent configuration:** Share configuration files across environments for consistent setups

Command Structure of Terraform Workspaces

Key commands for managing workspaces include:

terraform workspace list: Lists all available workspaces

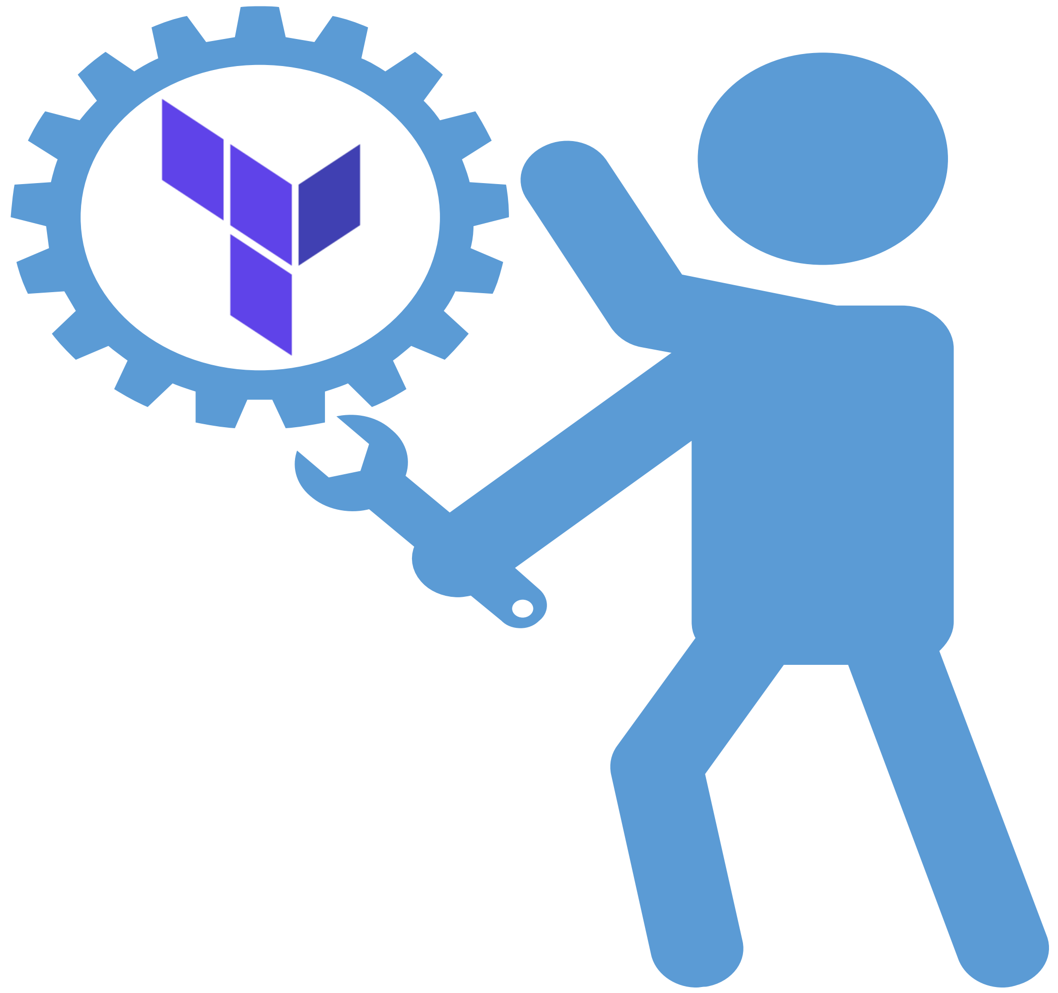
terraform workspace new <workspace_name>: Creates a new workspace

terraform workspace delete <workspace_name>: Deletes an existing workspace

terraform workspace show: Displays the current workspace

Terraform CLI

Terraform CLI is a tool that allows you to manage Infrastructure as Code (IaC) using HashiCorp Configuration Language (HCL).



It enables defining, provisioning, and managing of resources in a readable, versionable, reusable, and shareable format.

Terraform State Command

The terraform state command is used for advanced state management, allowing you to modify the state indirectly.

These commands safely manage Terraform state, reducing errors and inconsistencies, and enhancing reliability and maintainability.

Enable Logging

To enable detailed Terraform logs, set the TF_LOG environment variable to any value. This will display detailed logs on stderr. Here are the commands for different operating systems:

Linux

```
export TF_LOG=TRACE
```

PowerShell

```
$env:TF_LOG="TRACE"
```

You can run any command after this command to start debugging the terminal.

Terraform Import

Terraform import helps add the existing resources to the configuration and bring them into the Terraform state with minimal coding and effort.

The steps to add resources to our configuration and bring them into the state are:

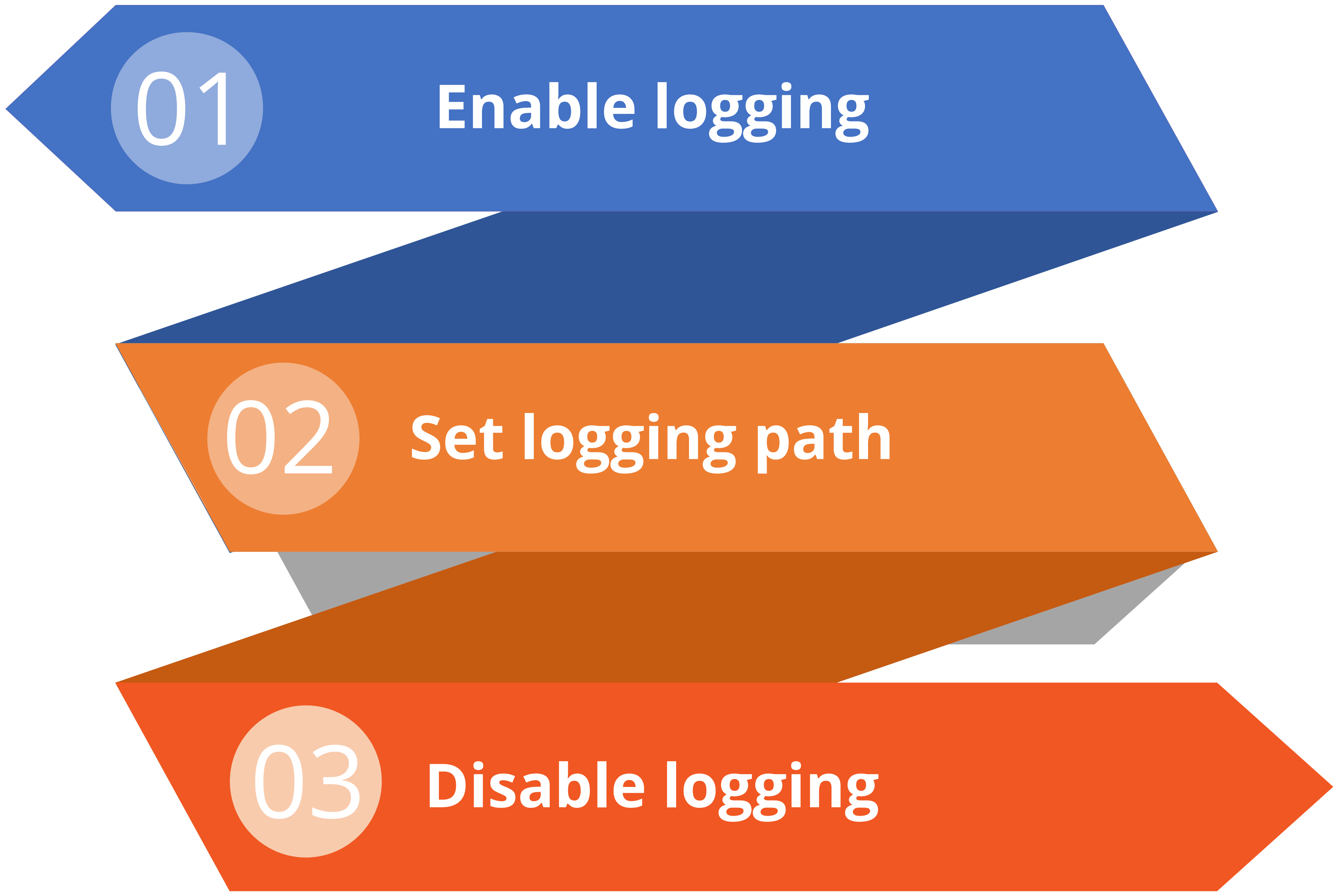
01 > Manually create resource (not with terraform)

02 > Prepare for a terraform import

03 > Import the resource in terraform

Debugging in Terraform

Debugging Terraform involves enabling detailed logging to help diagnose issues and understand the internal workings of Terraform. Here are the steps to debug Terraform configurations:



Set Logging Path

To save logged output, you can set **TF_LOG_PATH** to ensure the log is always appended to a specific file when logging is enabled. The commands on different OS are:

Linux

```
export TF_LOG_PATH="terraform_log.txt"
```

PowerShell

```
$env:TF_LOG_PATH="terraform_log.txt"
```

Open the **terraform_log.txt** to see the contents of the debug trace for any terraform command executed after the logging.

Disable Logging

Terraform logging can be disabled by clearing the appropriate environment variable. The commands on different OS are:

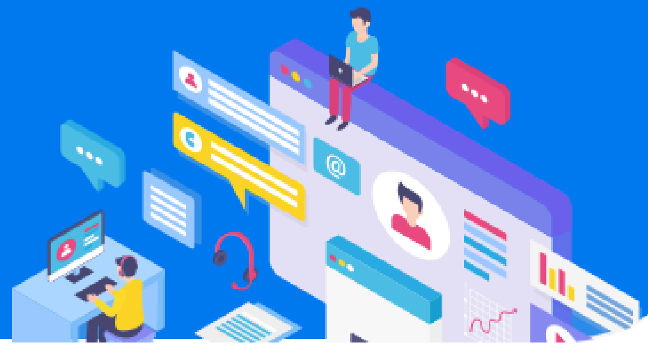
Linux

```
export TF_LOG=""
```

PowerShell

```
$env:TF_LOG=""
```

Assisted Practice



Generating Workspaces in Terraform **Duration: 10 Min.**

Problem statement:

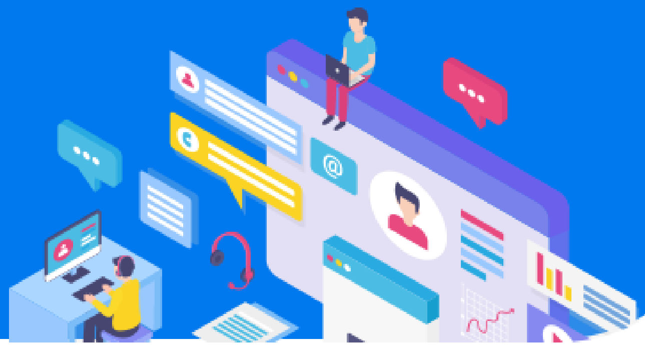
You have been assigned a task to effectively utilize Terraform workspaces for environment separation, create, and manage a development workspace, deploy infrastructure within it, and switch between different workspaces.

Outcome:

By completing this task, you will effectively utilize Terraform workspaces for environment separation, create and manage a development workspace, deploy infrastructure within it, and seamlessly switch between different workspaces.

Note: Refer to the demo document for detailed steps:
07_Generating_Workspaces_in_Terraform

Assisted Practice: Guidelines



Steps to be followed:

1. Update the main.tf file
2. Check the Terraform workspaces
3. Generate a new Terraform workspace for the development state
4. Modify main.tf file
5. Navigate between workspaces

Quick Check

You are tasked with integrating existing infrastructure into Terraform for better management. Your team is discussing the steps to accomplish this. What command should you use to import existing infrastructure into Terraform?

A. Using the **terraform import** command

B. Editing the Terraform state file directly

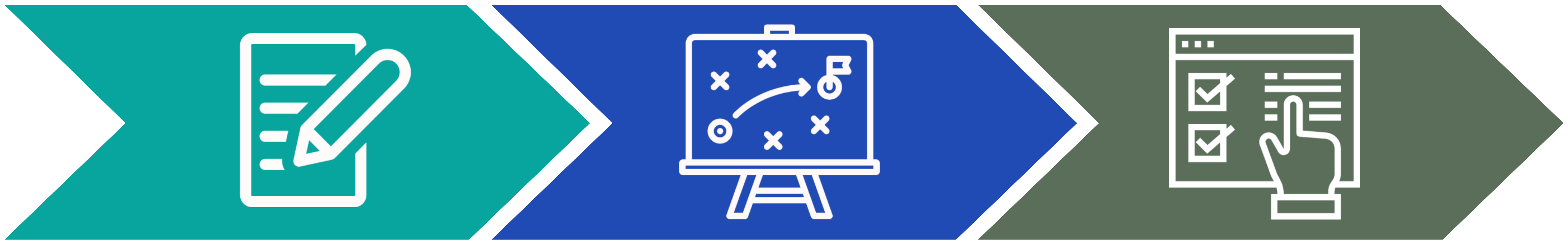
C. Running **terraform apply** with the import flag

D. Using the **terraform sync** command

Terraform Core Workflow

Understanding the Terraform Workflow

The core Terraform workflow has three steps:



Write

Develop your infrastructure as code

Plan

Preview changes before execution

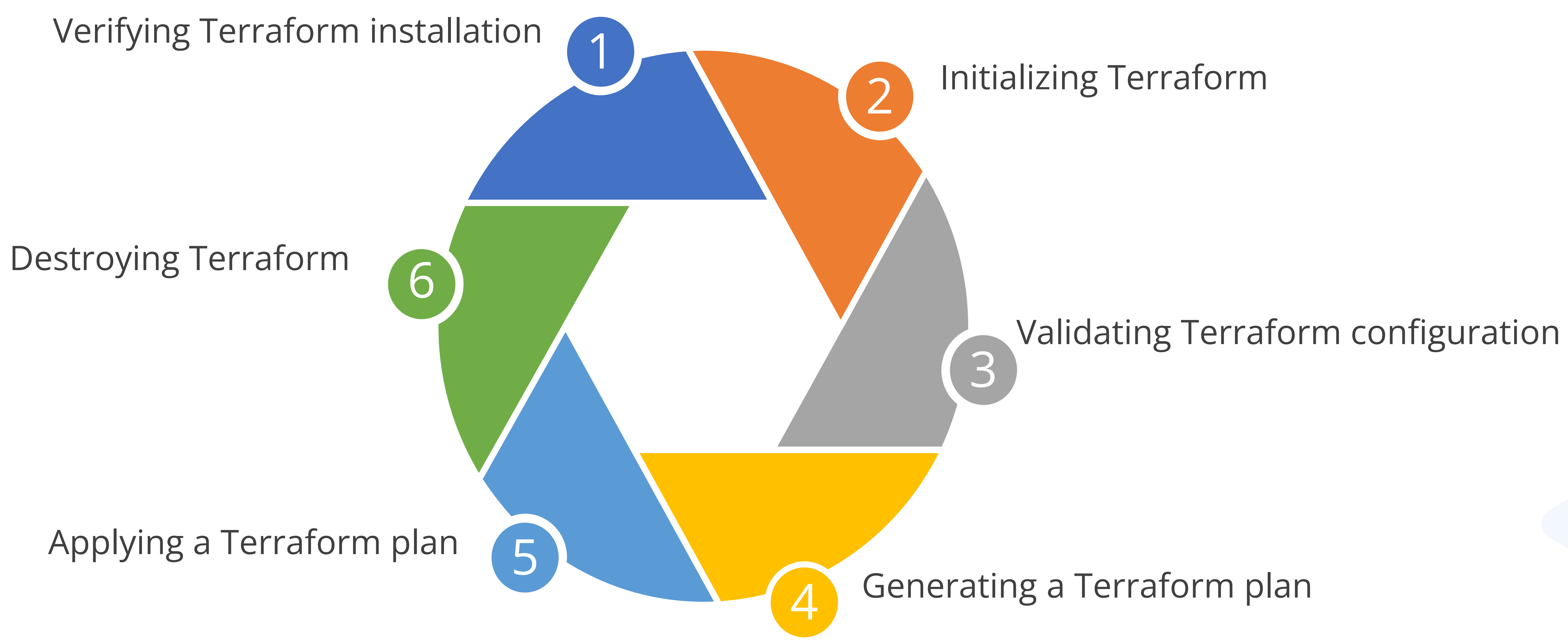
Apply

Deploy reproducible infrastructure

These high-level steps include several sub-steps for setting up the Terraform workflow.

Terraform Workflow

The Terraform workflow includes the following sub-steps to manage and automate infrastructure:


Verifying Terraform installation
1
2
Initializing Terraform
Destroying Terraform
6
Validating Terraform configuration
3
Applying a Terraform plan
5
4
Generating a Terraform plan

Verifying Terraform Installation

It is used to verify that Terraform is installed correctly.

Step 1 Run the following command in your terminal or command prompt:

```
terraform --version
```

or

```
terraform -v
```

Initializing Terraform

Initializing your workspace is used to initialize a working directory.

Step 2 Copy the code snippet below into the file called main.tf

```
resource "random_string" "random" {
  length = 16
}
```

Step 3 Run the init command shown below:

```
terraform init
```

Validating Terraform Configuration

It checks the syntax and structure of Terraform configuration files for errors before applying them, ensuring reliability and consistency.

Step 4 Run the following command in the terminal to validate the Terraform:

```
terraform validate
```

Terraform will display errors or warnings in your configuration, allowing you to correct them before applying. Ensure Terraform is installed and configured correctly before running **terraform validate**.

Generating a Terraform Plan

Terraform offers a dry run mode, allowing you to preview changes without affecting your infrastructure.

Step 5 This mode is activated by running the below command:

```
terraform plan
```

This will create a plan file that Terraform can use during an **apply** command.

Applying a Terraform Plan

Step 6 Follow the given command to create the resources specified in your plan file :

```
terraform apply myplan
```

Destroying Terraform

This command removes all remote objects managed by a specific Terraform configuration. It does not delete your configuration files (such as **main.tf**) but eliminates the resources created by your Terraform code.

Step 7

Use the command shown below to execute a planned destruction:

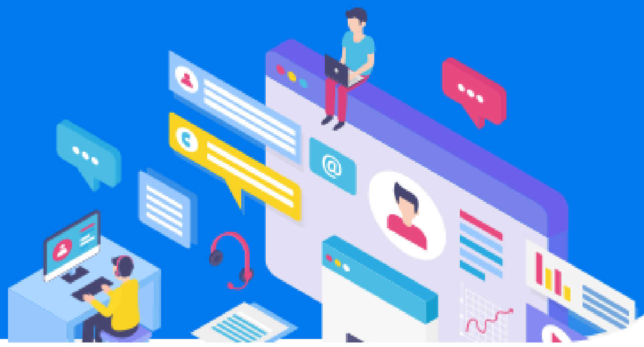
```
terraform plan -destroy
```

Step 8

To permanently delete the random string you created, execute the destroy command shown below:

```
terraform destroy
```

Assisted Practice



Validating applying and destroying a Terraform configuration file **Duration: 10 Min.**

Problem statement:

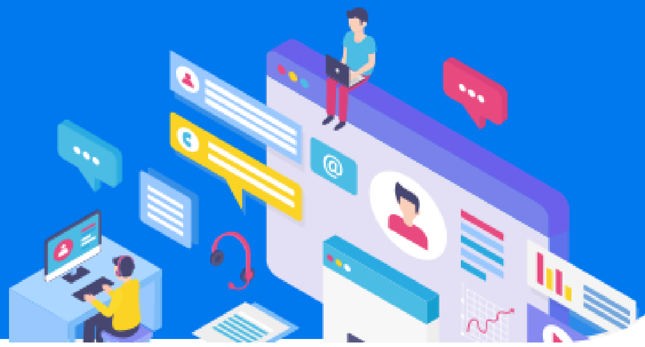
You have been assigned a task to validate, apply, and destroy the Terraform configuration file for efficient and reliable management of your infrastructure lifecycle.

Outcome:

By completing this task, you will be able to validate, apply, and destroy the Terraform configuration file, ensuring efficient and reliable management of your infrastructure lifecycle.

Note: Refer to the demo document for detailed steps:
08_Validating_applying_and_destroying_a_Terraform_configuration_file

Assisted Practice: Guidelines



Steps to be followed:

1. Modify the main.tf file with the provider configuration code
2. Validate the Terraform configuration file
3. Apply the validated Terraform configuration file
4. Destroy the applied Terraform configurations

Quick Check

You are starting a new project and plan to use Terraform for infrastructure provisioning. Your team is discussing the initial steps in the Terraform workflow. What is the first step in the Terraform core workflow?

A. Writing Terraform configuration files

B. Running terraform init

C. Running terraform plan

D. Running terraform apply

Key Takeaways

- Understand the role of providers in managing resources across various platforms, including cloud providers like AWS, Azure, and Google Cloud.
- Learn to develop custom Terraform modules for AWS EC2 instances, incorporating configuration and TLS certificates for secure communication.
- Implement local-exec and remote-exec provisioners in Terraform to automate post-deployment tasks on local and remote machines.
- Master the core Terraform workflow, including steps from initialization to destruction, ensuring best practices are followed.
- Apply version constraints to maintain compatibility and stability in Terraform configurations, ensuring reliable deployments.

Key Takeaways

- Utilize Terraform state and workspaces to manage infrastructure efficiently, supporting multiple environments within a single configuration.
- Develop methods for debugging Terraform configurations using tools like terraform validate, logs, and other commands to resolve errors.
- Configure and manage multiple providers in a single Terraform project to handle resources across different platforms and regions.
- Use the Terraform TLS provider to generate SSH keys, ensuring secure access to cloud infrastructure.
- Format Terraform code for readability and maintainability and use the taint command to manage and replace degraded resources.

Building and Testing a Terraform Module

Duration: 25 Min.

Project agenda: To build and test a Terraform module for efficient and streamlined infrastructure management using AWS

Description: As a DevOps engineer at a tech company, you are tasked with building and testing a Terraform module to standardize cloud resource deployment across projects. Your objectives include creating a reusable module that follows best practices, ensures security compliance, and is flexible for different requirements. Additionally, you will develop automated tests to validate the module's functionality, aiming to streamline provisioning, reduce errors, and accelerate delivery.

Building and Testing a Terraform Module

Duration: 25 Min.

Perform the following:

1. Configure the AWS CLI from the terminal
2. Create the directory structure for the Terraform project
3. Write the Terraform VPC module code
4. Write the main Terraform project code
5. Deploy the code and test the module

Expected deliverables: A step-by-step guide to writing the Terraform VPC module code, writing the main Terraform project code, and deploying the code to test the module.

Thank You