Sr.No.	Topics	Duration(Mins)	Session No(2 Hours)	Session No.(4 Hours)
1	Introduction To Terraform	20	1	1
2	Terraform Vs Ansible	20	1	1
3	Terraform Architecture	30	1	1
4	Terraform Configuration	30	1	1
5	Terraform Commands	20	1	1
6	Managing Terraform Resources	30	2	1
7	Terraform End to End Project	30	2	1
8	Practical	60	2	1

Terraform Overview

1. Introduction to Terraform

   Information

- Terraform is an open-source infrastructure as code (IaC) tool developed by HashiCorp.
- It allows developers to define, manage, and provision infrastructure resources in a declarative manner using a simple and human-readable configuration language.
- With Terraform, you can describe your entire infrastructure stack, including virtual machines, networks, storage, and other cloud resources, as code.

- Key Features of Terraform:

a. Infrastructure as Code (IaC):

- Terraform allows you to define your infrastructure in code, providing a version-controlled and repeatable way to create and manage resources.
- This eliminates manual configuration and reduces the chances of human errors in the deployment process.

b. Multi-Cloud Support:

- Terraform is cloud-agnostic and supports multiple cloud providers, including Amazon Web Services (AWS), Microsoft Azure, Google Cloud Platform (GCP), and others.
- This flexibility allows you to manage resources across different cloud environments using a single set of configurations.

c. Declarative Language:

- Terraform uses a declarative configuration language, called HashiCorp Configuration Language (HCL), to define infrastructure resources.
- In HCL, you specify what you want your infrastructure to look like, and Terraform handles the how and when of resource creation and management.

d. Resource Graph:

- Terraform creates a dependency graph of your infrastructure resources based on the configurations.
- It automatically determines the order in which resources should be created or updated to avoid any conflicts or inconsistencies.

e. Plan and Apply:

- Terraform provides a two-step process of planning and applying changes.
- During the plan phase, Terraform generates an execution plan that shows what resources will be created, modified, or deleted.
- In the apply phase, Terraform makes the necessary changes to bring your infrastructure to the desired state.

f. State Management:

- Terraform maintains a state file that keeps track of the actual infrastructure resources deployed.
- This state file is used to compare the desired state (as specified in the configurations) with the actual state and determine what changes are required.

g. Reusable Modules:

- Terraform supports modularization, allowing you to create reusable modules to define common sets of resources or patterns.
- Modules can be shared across different projects, promoting code reusability and standardization.

h. Collaboration:

- Terraform enables collaboration among team members by storing the configuration files in version control systems like Git.
- Multiple developers can work on the same infrastructure code simultaneously and merge changes as needed.

i. Continuous Integration and Deployment (CI/CD) Integration:

- Terraform can be integrated with CI/CD pipelines to automate the process of infrastructure deployment and management, ensuring consistent and reliable infrastructure changes as part of your application deployment process.

- Terraform simplifies the process of managing cloud infrastructure, making it easier for teams to create, update, and scale resources in a consistent and efficient manner.
- By treating infrastructure as code, Terraform empowers organizations to adopt infrastructure changes as part of their software development lifecycle, leading to better collaboration, faster deployments, and improved scalability.

Overview of infrastructure provisioning :

- Infrastructure provisioning refers to the process of setting up and preparing the necessary hardware, software, and networking resources to support an application or service.
- In the context of cloud computing, infrastructure provisioning involves creating and configuring virtual machines, storage, networks, and other cloud resources to meet the requirements of a specific application or workload.

Benefits of Terraform:

a. Infrastructure as Code (IaC):

- Terraform allows you to define your entire infrastructure stack as code using a declarative configuration language (HCL).
- This enables version control, collaboration, and repeatability, making it easier to manage and update infrastructure.

b. Multi-Cloud Support:

- Terraform is cloud-agnostic and supports multiple cloud providers, including AWS, Azure, GCP, and more.

- This flexibility allows you to use the same code to provision and manage resources across different cloud environments.

c. Automation and Consistency:

- Terraform automates the process of provisioning and managing infrastructure resources, ensuring consistent and reliable deployments.
- This reduces the chances of human errors and increases operational efficiency.

d. Scalability:

- With Terraform, you can easily scale resources up or down to meet changing demands.
- It supports auto-scaling features, allowing your infrastructure to adapt dynamically to workload fluctuations.

e. Resource Dependencies and Graph:

- Terraform automatically handles resource dependencies and creates a resource graph based on your configuration.
- This ensures that resources are provisioned in the correct order, avoiding conflicts and ensuring a successful deployment.

f. State Management:

- Terraform maintains a state file that tracks the current state of your infrastructure.
- This state file is used to understand what resources are currently deployed and to plan and apply changes incrementally.

g. Modularity and Reusability:

- Terraform supports modularization, allowing you to create reusable modules for common infrastructure patterns.
- This promotes code reusability and standardization across projects.

h. Change Management:

- Terraform provides a plan and apply workflow, which shows you the proposed changes before applying them to the infrastructure.
- This allows you to review and approve changes before they are executed.

i. Auditability and Compliance:

- The use of code-based configurations in Terraform enhances auditability and compliance.

- You can track changes, review past configurations, and maintain an audit trail of infrastructure updates.

Features of Terraform:

a. Declarative Syntax:

- Terraform uses a declarative syntax in the HashiCorp Configuration Language (HCL) to define infrastructure resources.
- You describe what you want the infrastructure to look like, and Terraform takes care of the underlying details.

b. Resource Providers:

- Terraform supports various resource providers, which are plugins that interact with cloud APIs.
- These providers enable you to manage resources on different cloud platforms, as well as other infrastructure providers like Docker, Kubernetes, and more.

c. Terraform CLI:

- The Terraform Command Line Interface (CLI) provides a set of commands for initializing, planning, applying, and managing infrastructure changes.
- It is the primary tool for interacting with Terraform.

d. Provisioners:

- Terraform includes provisioners that allow you to run scripts or configuration management tools on newly created resources.
- This enables you to customize and configure resources during the provisioning process.

e. Data Sources:

- Terraform supports data sources, which allow you to fetch information from external sources (e.g., existing resources) and use that data in your configuration.
- Data sources help you incorporate existing infrastructure into your Terraform setup.

f. Remote State:

- Terraform supports storing the state file remotely, allowing for collaboration among team members and facilitating the management of state in a shared environment.

g. Modules:

- Terraform modules encapsulate reusable sets of resources, enabling you to create abstractions and share common infrastructure patterns across projects.

- Modules promote code modularity and simplify infrastructure configurations.

h. Backends:

- Terraform backends provide state storage and retrieval for Terraform configurations.
- Different backends, like Amazon S3 or Azure Storage, allow you to choose where to store the state file.

- Terraform's combination of flexibility, automation, and cloud-agnosticism makes it a popular choice for managing infrastructure across various cloud providers and on-premises environments.
- It empowers organizations to adopt best practices for infrastructure management, such as version control, automation, and infrastructure-as-code, leading to more efficient and reliable deployments.

2. Terraform Architecture

Information

- Terraform follows a client-server architecture, where the client interacts with the Terraform CLI (Command

Line Interface) and communicates with various cloud providers' APIs, while the server manages the state and the resources. Let's dive into the key components of Terraform architecture:

a. Terraform CLI (Client):

- The Terraform CLI is the main interface that users interact with to manage their infrastructure as code.
- It is a command-line tool that allows users to execute Terraform commands, such as terraform init, terraform plan, terraform apply, and more.
- The CLI reads the Terraform configuration files (.tf files) and translates them into API requests to cloud providers or other infrastructure providers.

b. Terraform Configuration:

- The Terraform configuration files are written in the HashiCorp Configuration Language (HCL) or JSON format.
- These files define the desired state of the infrastructure resources, including the providers, resources, variables, data sources, and outputs.
- The configuration files specify what infrastructure needs to be created, modified, or destroyed.

c. Terraform Providers:

- Providers are responsible for managing the lifecycle of resources and are responsible for translating the Terraform configuration into API calls for the corresponding cloud or infrastructure provider.
- Each provider is a separate plugin responsible for interacting with a specific cloud platform, such as AWS, Azure, GCP, or other infrastructure platforms like Docker, Kubernetes, etc.

d. Terraform State:

- The Terraform state is a crucial component that tracks the current state of the managed infrastructure.
- The state file records the mapping between resources defined in the configuration files and the real-world resources that were created in the cloud or infrastructure provider.
- Terraform uses the state to identify any changes to the infrastructure and to plan and apply updates correctly.

e. Terraform Backend:

- The backend is responsible for storing the state file securely and making it available for team collaboration.
- Terraform supports various backends, including local file storage, Amazon S3, Azure Blob Storage, HashiCorp Consul, and more.

- By default, the state file is stored locally, but it is recommended to use remote backends for team-based or production environments.

f. Terraform Plan and Apply:

- The terraform plan command generates an execution plan by comparing the current state of the infrastructure with the desired state defined in the configuration files.
- It shows what changes will be made before actually applying them. The terraform apply command applies the changes to the infrastructure and provisions or modifies resources as necessary.

g. Terraform Modules:

- Modules are reusable blocks of Terraform configuration that encapsulate infrastructure resources and logic.
- They allow users to create abstractions and share common infrastructure patterns across different projects.
- Modules promote code reusability and maintainability.

h. Terraform Provisioners:

- Provisioners are used to run scripts or configuration management tools on the newly created resources.

- Provisioners allow users to customize and configure the resources after creation.

- Terraform follows the Infrastructure as Code (IaC) philosophy, enabling users to define and manage infrastructure using code.
- The architecture of Terraform ensures that infrastructure deployments are repeatable, version-controlled, and can be managed efficiently at scale.
- It simplifies the process of provisioning and managing infrastructure across different cloud providers and infrastructure platforms.

3. Terraform Configuration

   Information

- Terraform configuration is written in the HashiCorp Configuration Language (HCL) or JSON format.
- It is a declarative language used to define the desired state of your infrastructure and resources.
- Terraform configuration files have a '.tf' extension and contain the following key elements:

a. Provider Block:

- The provider block specifies the cloud or infrastructure provider you want to use.
- It defines the connection details and credentials required to authenticate with the provider's API.
- For example, to use AWS as a provider, you would have a block like this:

```
provider "aws" {
  region = "us-east-1"
}
```

b. Resource Blocks:

- Resource blocks define the individual infrastructure resources that Terraform will manage.

- They represent the resources you want to create, modify, or delete. Each resource block has a unique type and a set of arguments that configure the resource.
- For example, to create an AWS EC2 instance, you would have a block like this:

```
resource "aws_instance" "example" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
}
```


c. Data Blocks:

- Data blocks allow you to fetch information from external sources, like existing resources or APIs, and use that data in your configuration.
- For example, you can use a data block to retrieve the ID of an existing AWS security group and use it in your resource block:

```
data "aws_security_group" "existing_group" {
  name = "my-existing-security-group"
}
```

d. Variable Blocks:

- Variables allow you to parameterize your Terraform configuration, making it more flexible and reusable.
- You can define variables in a separate .tf file or within the same configuration file.
- For example:

```
variable "instance_type" {
  description = "The type of EC2 instance to launch"
  default     = "t2.micro"
}
```

e. Output Blocks:

- Output blocks define values that are exposed to the user after the Terraform execution.

- They allow you to view useful information, such as IP addresses or resource IDs, that you may need after the deployment is complete:

```
output "public_ip" {
  value = aws_instance.example.public_ip
}
```

f. Modules:

- Modules are reusable blocks of Terraform configuration that can be used to encapsulate and abstract infrastructure resources.
- They promote code reusability and allow you to create abstractions for common infrastructure patterns.

- Terraform uses these configuration files to create a dependency graph of resources and their relationships.
- When you run 'terraform apply', Terraform will create or update resources to match the configuration defined in your files.

- It is important to version control your Terraform configuration files and follow best practices for managing infrastructure as code.
- This ensures that changes are tracked, reviewed, and applied consistently, making your infrastructure more manageable, repeatable, and reliable.

4. Terraform Commands

Information

- Terraform provides a set of commands to interact with the infrastructure and manage the lifecycle of resources.

a. terraform init:

- This command initializes a new or existing Terraform working directory.
- It downloads and installs the necessary provider plugins and modules mentioned in the configuration files.
- It is the first command you should run in a new Terraform project or when the configuration changes.

b. terraform plan:

- The 'plan' command creates an execution plan that shows what Terraform will do when you apply the configuration.
- It compares the desired state in the configuration with the current state of the infrastructure and displays any changes that will be made.
- It does not actually apply any changes; it only shows you what will happen.

c. terraform apply:

- The ‘apply’ command is used to create or update infrastructure according to the configuration.
- It reads the configuration, compares it to the current state, and makes the necessary changes to reach the desired state.
- It prompts for confirmation before proceeding, and once confirmed, it creates or updates resources.

d. terraform destroy:

- The ‘destroy’ command is used to delete all the resources defined in the configuration.
- It reads the current state and destroys all the resources associated with the project.
- It is useful when you want to tear down your infrastructure completely.

e. terraform validate:

- The ‘validate’ command validates the syntax and configuration of the Terraform files.
- It checks for errors and issues in the configuration and ensures that all resources and variables are correctly defined.

f. terraform state:

- This command allows you to manage the Terraform state file directly.
- You can perform operations like listing resources, moving resources, or removing resources from the state.

- terraform fmt:
- The ‘fmt’ command is used to format the Terraform configuration files in a canonical format.
- It ensures consistent formatting and makes the code easier to read and maintain.

g. terraform get:

- The ‘get’ command is used to download and install any modules or plugins defined in the Terraform configuration.
- It ensures that all required dependencies are available for use.

h. terraform workspace:

- Terraform workspaces allow you to manage multiple sets of infrastructure resources for different environments (e.g., development, staging, production).
- The ‘workspace’ command is used to manage these workspaces, like switching between them, creating new ones, or deleting existing ones.

- These are some of the common Terraform commands.
- There are other commands and options available for more advanced use cases and specific operations.
- You can get a complete list of Terraform commands and their usage by running ‘terraform –help’.

5. Managing Terraform Resources

Information

- Managing Terraform resources involves creating, modifying, and deleting infrastructure resources using Terraform configuration files.

a. Creating Resources:

- To create resources, you define resource blocks in your Terraform configuration files.
- Resource blocks specify the type of resource you want to create and its attributes.
- For example, to create an AWS EC2 instance, you would define a resource block like this:

```
resource "aws_instance" "example" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
}
```

b. Modifying Resources:

- To modify resources, you can make changes directly in the Terraform configuration files.
- For example, if you want to change the instance type of the EC2 instance, you can modify the 'instance_type' attribute in the resource block and run 'terraform apply' to apply the changes.

c. Dependent Resources:

- Terraform automatically manages the dependencies between resources.
- If you have a resource that depends on another resource, Terraform ensures that the dependencies are created or updated before the dependent resource.

d. State Management:

- Terraform keeps track of the state of the managed resources in a state file.
- The state file records the relationship between the resource in your configuration and the actual resources in the cloud provider.
- Terraform uses this state to manage resources and keep track of changes.

e. Tainting Resources:

- If you want to recreate a resource without actually modifying it, you can use the ‘terraform taint’ command.
- Tainting marks a resource as "needing recreation," and the next ‘terraform apply’ will destroy and recreate the tainted resource.

f. Destroying Resources:

- To destroy resources and remove them from your cloud provider, you use the ‘terraform destroy’ command.
- This command reads the state file and destroys all the resources defined in your configuration.
- It will prompt for confirmation before destroying the resources.

g. Data Sources:

- In addition to creating resources, you can use data sources to fetch information from the cloud provider.
- Data sources allow you to query existing resources and use their attributes in your configuration.

- Remember to follow best practices when managing Terraform resources.
- Always version control your Terraform configuration files, use Terraform workspaces for managing multiple

environments, and make use of modules for code reusability and maintainability.

- Regularly apply ‘terraform plan’ before applying changes to ensure that you understand the impact of your actions on the infrastructure.

6. Terraform End to End Project

Information

- Creating an end-to-end project with Terraform involves provisioning and managing infrastructure resources on a cloud provider using Terraform configuration files.
- Let's outline the steps to create an end-to-end Terraform project:

a. Project Setup:

- Create a new directory for your Terraform project.
- Initialize the project by running terraform init to download required plugins and initialize the working directory.

b. Provider Configuration:

- Choose the cloud provider you want to use (e.g., AWS, Azure, GCP, etc.).
- Configure the provider in your Terraform configuration file using the appropriate provider block and authentication credentials.

c. Resource Definition:

- Define the infrastructure resources you want to create, such as virtual machines, storage, networking components, etc.
- Use resource blocks in your configuration file to describe the resources and their attributes.

d. Variables and Input:
- Parameterize your Terraform configuration using variables.
- Create a variables file or use environment variables to pass input values to your configuration.

e. Modules (Optional):
- Consider using modules to organize and reuse Terraform code.
- Create custom modules for specific resource types or complex infrastructure components.

f. Terraform Plan:
- Run ‘terraform plan’ to generate an execution plan.
- Review the plan to ensure it matches your expectations and that there are no unintended changes.

g. Terraform Apply:
- Apply the changes to create the infrastructure resources by running ‘terraform apply’.

- Review the proposed changes, and confirm by typing "yes" when prompted.

h. Resource Management:

- Make modifications to the resources as needed by updating your Terraform configuration files.
- Re-run ‘terraform plan’ and ‘terraform apply’ to apply the changes.

i. Terraform State Management:

- Terraform maintains the state of your infrastructure in a state file.
- Consider using a remote backend to store the state file securely and allow for collaboration with a team.

j. Terraform Destroy:

- When you are done with the resources, use terraform destroy to remove the infrastructure.
- Be cautious as this will delete all resources created by Terraform.

Practical :

1. Installation of Terraform on AWS EC2 Instance

- To install Terraform on an AWS EC2 instance, you can use the following steps:

a. Connect to the EC2 Instance:

- Launch an EC2 instance on AWS using your preferred Linux distribution (e.g., Amazon Linux, Ubuntu, CentOS).
- Connect to the instance via SSH using the private key associated with the key pair used during instance creation.

b. Update the Package Manager:

- Update the package manager on your EC2 instance to ensure you have the latest package information.
- For Amazon Linux or CentOS:

```
sudo yum update -y
```

- For Ubuntu:

```
sudo apt update
```

c. Install Unzip (if not installed):

- Terraform binaries are distributed as ZIP archives, so ensure the unzip utility is installed on your instance.
- For Amazon Linux or CentOS:

```
sudo yum install unzip -y
```

For Ubuntu:

```
sudo apt install unzip -y
```

d. Download Terraform:

- Visit the official Terraform website (https://www.terraform.io/downloads.html) to find the URL for the latest version of Terraform.
- Use the wget command to download the Terraform ZIP archive.
- For example:

```
wget
https://releases.hashicorp.com/terraform/<version>
/terraform_<version>_linux_amd64.zip
```

- Replace <version> with the desired version of Terraform (e.g., 0.15.4).

e. Unzip and Move Terraform Binary:

- Unzip the downloaded Terraform archive using the unzip command:

```
unzip terraform_<version>_linux_amd64.zip
```

- Move the Terraform binary to a directory in your system's PATH.
- For example:

```
sudo mv terraform /usr/local/bin/
```

f. Verify Installation:

- Verify that Terraform is installed and accessible by running the following command:

```
terraform version
```

g

- You can remove the downloaded ZIP file to save space on the instance:

```
rm terraform_<version>_linux_amd64.zip
```

- That's it! You have successfully installed Terraform on your AWS EC2 instance.

- You can now use Terraform to manage your infrastructure and resources on AWS.
- Remember to keep Terraform updated to take advantage of the latest features and bug fixes.

2. Writing Terraform Configuration

- Writing Terraform configuration involves describing the desired state of your infrastructure using the HashiCorp Configuration Language (HCL).
- The HCL syntax is human-readable and easy to understand.
- Below are the steps to write a basic Terraform configuration for creating an AWS EC2 instance:

a. Create a New Directory:

- Create a new directory for your Terraform project. This will contain all your Terraform configuration files.

b. Initialize Terraform:

- In the project directory, run terraform init. This will initialize the working directory and download the necessary plugins.

c. Create a Configuration File:

- Create a new file named main.tf in the project directory. This is where you'll define your infrastructure resources.

d. Provider Configuration:

- Define the AWS provider and specify the AWS region in the 'main.tf file':

```
provider "aws" {
  region = "us-west-2"  # Update with your desired region
}
```

e. Define an EC2 Instance:

- Add a resource block to create an EC2 instance:

```
resource "aws_instance" "example" {
  ami           = "ami-0c55b159cbfafe1f0"  # Amazon Linux 2 AMI
  instance_type = "t2.micro"
}
```

f. Variable Definitions (Optional):

- You can use variables to make your configuration more flexible and reusable.

- Define variables in a separate file (e.g., variables.tf), or use inline variables directly in your ‘main.tf’ file.

```
variable "region" {
  description = "AWS region to deploy resources."
  default     = "us-west-2"
}
```

g. Output (Optional):

- You can add an output block to display information about the resources created.

```
output "instance_ip" {
  value = aws_instance.example.public_ip
}
```

h. Plan and Apply:

- Run terraform plan to see what resources Terraform will create.

- This step is optional but recommended to review the changes before applying them.
- Run terraform apply to create the resources defined in your configuration.

- That's it! You have now written a basic Terraform configuration for creating an AWS EC2 instance.
- Remember to run terraform destroy when you're done with the resources to clean up the infrastructure.
- Terraform will ensure that the actual state of your infrastructure matches the state defined in your configuration.

3. Creation of AWS EC2 instance using terraform
- To create an AWS EC2 instance using Terraform, follow the steps below:

a. Install Terraform:
- Make sure you have Terraform installed on your local machine.
- You can download and install it from the official website: https://www.terraform.io/downloads.html

b. Set Up AWS Credentials:

- Before proceeding, ensure you have AWS credentials (Access Key ID and Secret Access Key) configured on your local machine.
- You can set them up using the AWS CLI or environment variables.

c. Create a New Directory:

- Create a new directory for your Terraform project.

d. Initialize Terraform:

- In the project directory, run terraform init.
- This will initialize the working directory and download the necessary providers.

e. Create a Configuration File:

- Create a new file named main.tf in the project directory.
- This is where you'll define your infrastructure resources.

f. Provider Configuration:

- Define the AWS provider and specify the AWS region in the ‘main.tf’ file:

```
provider "aws" {
  region = "us-west-2"  # Replace with your desired region
}
```

g. Define an EC2 Instance:

- Add a resource block to create an EC2 instance:

```
resource "aws_instance" "example" {
  ami           = "ami-0c55b159cbfafe1f0"  # Amazon Linux 2 AMI (Replace with your desired AMI)
  instance_type = "t2.micro"               # Replace with your desired instance type
}
```

- Run terraform plan to see what resources Terraform will create.
- This step is optional but recommended to review the changes before applying them.
- Run terraform apply to create the EC2 instance.

i. Clean Up:

- When you're done with the resources, you can run terraform destroy to remove the EC2 instance.

- That's it! You have now created an AWS EC2 instance using Terraform.
- Terraform will ensure that the state of your infrastructure matches the state defined in your configuration, and it will manage the lifecycle of the resources for you.
- Remember to handle sensitive information such as access keys and secret keys securely, and to always review and understand the changes Terraform will make before applying them to your infrastructure.

4. Managing AWS resources using terraform

- Managing AWS resources using Terraform involves defining infrastructure as code in Terraform configuration files and using Terraform commands to create, modify, and delete resources in your AWS account.
- Here's an overview of how to manage AWS resources with Terraform:

a. Provider Configuration:

- Start by configuring the AWS provider in your Terraform configuration file (main.tf or any other .tf file).
- This tells Terraform that you'll be working with AWS resources.

```
provider "aws" {
  region = "us-west-2"  # Replace with your desired AWS region
}
```

b. Define AWS Resources:

- In the same Terraform configuration file, define the AWS resources you want to create.
- For example, an EC2 instance, S3 bucket, or VPC.

```
resource "aws_instance" "example" {
  ami           = "ami-0c55b159cbfafe1f0"  # Amazon Linux 2 AMI (Replace with your desired AMI)
  instance_type = "t2.micro"               # Replace with your desired instance type
}

resource "aws_s3_bucket" "example" {
  bucket = "example-bucket"  # Replace with your desired bucket name
```

c. Variable Definitions (Optional):

- You can use variables to make your configuration more flexible and reusable.
- Define variables in a separate file (e.g., variables.tf), or use inline variables directly in your .tf file.

```
variable "aws_region" {
  description = "AWS region to deploy resources."
  default     = "us-west-2"
}
```

- Use Terraform commands to manage resources:
- terraform init: Initialize the working directory.
- terraform plan: Show the execution plan for creating/modifying resources.
- terraform apply: Apply the changes and create/modify resources.
- terraform destroy: Destroy the resources defined in the configuration.

e. State Management:

- Terraform uses state files to keep track of the actual state of your infrastructure.
- By default, state files are stored locally, but it is recommended to use remote state storage (e.g., S3) for collaboration and durability.

f. Reusability:

- You can reuse modules in Terraform to encapsulate sets of resources and configurations, making it easier to manage complex infrastructures.

g. Data Sources and Outputs (Optional):

- Use data sources to reference existing resources in your Terraform configuration.
- Use outputs to display information about resources created in the terraform apply output.

h. Terraform Workspaces (Optional):

- Workspaces allow you to manage multiple sets of infrastructure configurations for different environments

(e.g., dev, staging, production) within the same Terraform configuration.

- Remember that Terraform is a powerful tool for managing infrastructure as code, and it's essential to review changes before applying them to avoid unexpected modifications to your infrastructure.
- Regularly run terraform plan to preview changes and terraform destroy to clean up resources that are no longer needed.
- Additionally, follow best practices for securely managing AWS credentials and state files to ensure the safety and integrity of your infrastructure.

5. End to End Infrastructure Creation Project

- Creating an end-to-end infrastructure project involves defining and managing various cloud resources using infrastructure as code tools like Terraform.
- Below is an outline of an example end-to-end infrastructure creation project:

- Objective: Create a simple web application infrastructure on AWS using Terraform.
- Steps:

a. Project Setup:
- Set up a new directory for your project and initialize Terraform in the directory using terraform init.

b. Provider Configuration:
- Configure the AWS provider in your main Terraform configuration file (main.tf).

c. VPC (Virtual Private Cloud):
- Create a new VPC with public and private subnets.

d. Internet Gateway and Route Table:
- Create an internet gateway and associate it with the public subnet.
- Set up a route table to route traffic to the internet gateway.

e. Security Groups:

- Create security groups to control inbound and outbound traffic for the instances.

f. EC2 Instances:
- Launch EC2 instances in the public and private subnets, using the Amazon Linux 2 AMI.
- Associate an Elastic IP address with the public instance.

g. Load Balancer (Optional):

- Create an Application Load Balancer (ALB) and configure it to distribute traffic to the EC2 instances.

h. Database (Optional):

- Create a database instance using Amazon RDS (MySQL or PostgreSQL).

i. Auto Scaling (Optional):

- Set up an auto-scaling group to automatically add or remove EC2 instances based on traffic load.

j. SSH Key Pair:

- Create an SSH key pair to access the EC2 instances securely.

k. Output (Optional):

- Add an output block to display the public IP address or load balancer DNS name.

l. Terraform Plan and Apply:

- Run terraform plan to see what resources Terraform will create.
- Run terraform apply to create the infrastructure.

m. Testing:

- Test the web application by accessing the public IP or load balancer DNS.

n. Cleanup:

- When you're done testing, run terraform destroy to clean up the infrastructure.

- Remember to follow best practices, such as securing credentials, using variables for reusability, and storing the state remotely for collaboration and version control.
- This is a simplified example; in a real-world scenario, you may need to configure additional resources, networking settings, IAM roles, and more depending on your application's requirements.

StarAgile

StarAgile