Task 15

Terraform Workspaces

Umar Satti

Table of Contents

1. Task Description

This task focuses on understanding and implementing Terraform workspaces to manage multiple environments such as development, staging, and production using a single Terraform codebase. The objective is to demonstrate how workspaces isolate state files, enable environment-specific configurations, and simplify infrastructure lifecycle management. Through hands-on implementation, this task covers workspace creation, switching, variable management, remote backends, modular infrastructure deployment, validation, destruction, and workspace cleanup.

2. Terraform Workspaces

2.1 Problem Statement

When managing infrastructure with Terraform, it's common to work with multiple environments such as Development, Staging, and Production. Each of these environments often requires its own set of resources and configurations. To keep things organized and maintain a clean infrastructure codebase, Terraform provides a powerful feature called Workspaces.

2.2 Workspace

A Terraform Workspace allows you to create and manage separate environments within a single Terraform configuration. Each workspace is associated with its own **state file**, which means the resources for one environment are isolated from another, even though they share the same configuration files. This makes it much easier to manage multiple deployments from a single codebase.

- Workspaces should have **unique** names.
- Share the same configured backend.
- Each workspace has an **individual state file** within that backend.

The following steps check **the current workspace** to see Terraform's default behavior.

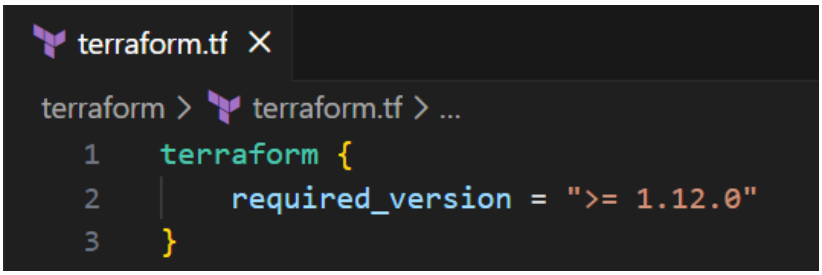
Step 1: Open Terminal

Navigate to any empty folder where the terraform project will be kept. This is where the terraform configuration files will be created.

Step 2: Initialize Terraform

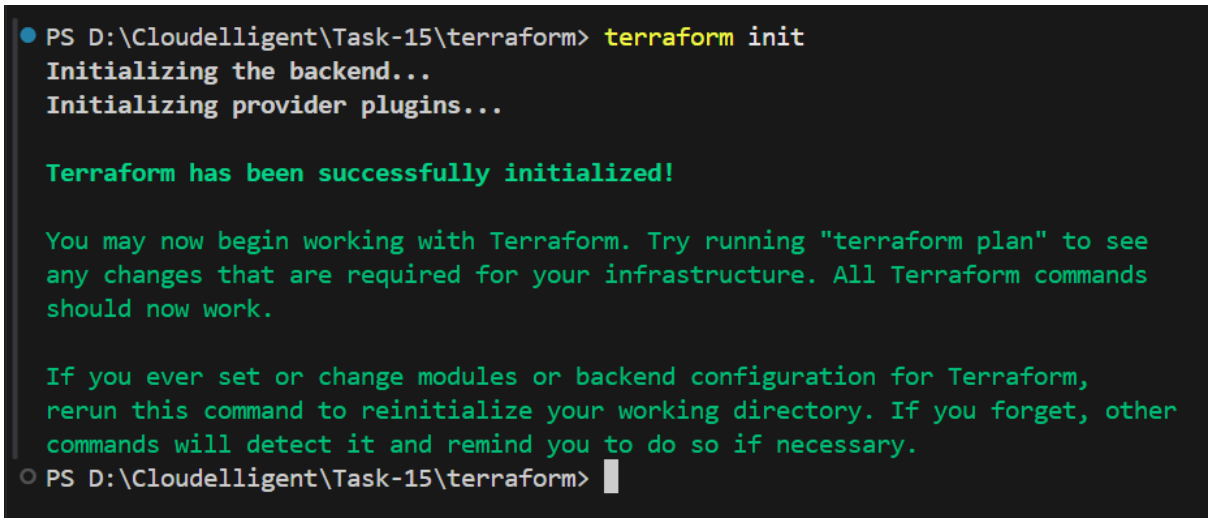
Create a single minimal terraform file like **terraform.tf** and add the terraform block containing the required version.

- **terraform.tf** file inside the project
- required_version argument for version constraint
- The minimum terraform version used is 1.12.0



Initialize Terraform by running the following command.

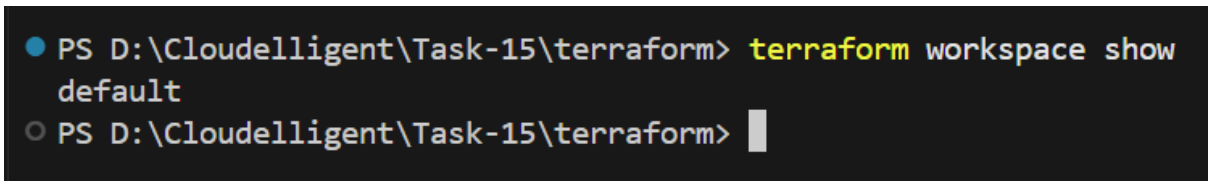
- *terraform init*



Step 3: Check Current Workspace

View the current workspace by running the following command

- *terraform workspace show*
- The expected output should display “default” workspace only.



3. Create a New Terraform Workspace

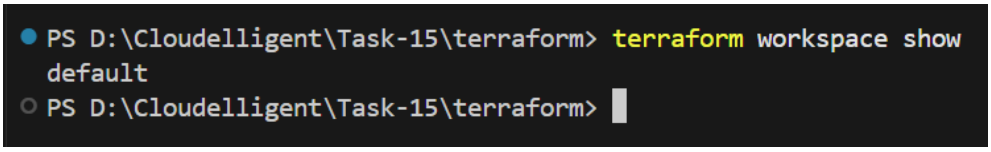
Terraform allows you to create multiple workspaces to represent different environments. When a new workspace is created:

- Terraform creates a new, isolated state (state file)
- The configuration files remain the same
- Terraform automatically switches to the new workspace

Step 1: Verify Current Workspace

Before creating a new workspace, check the current one using the following command.

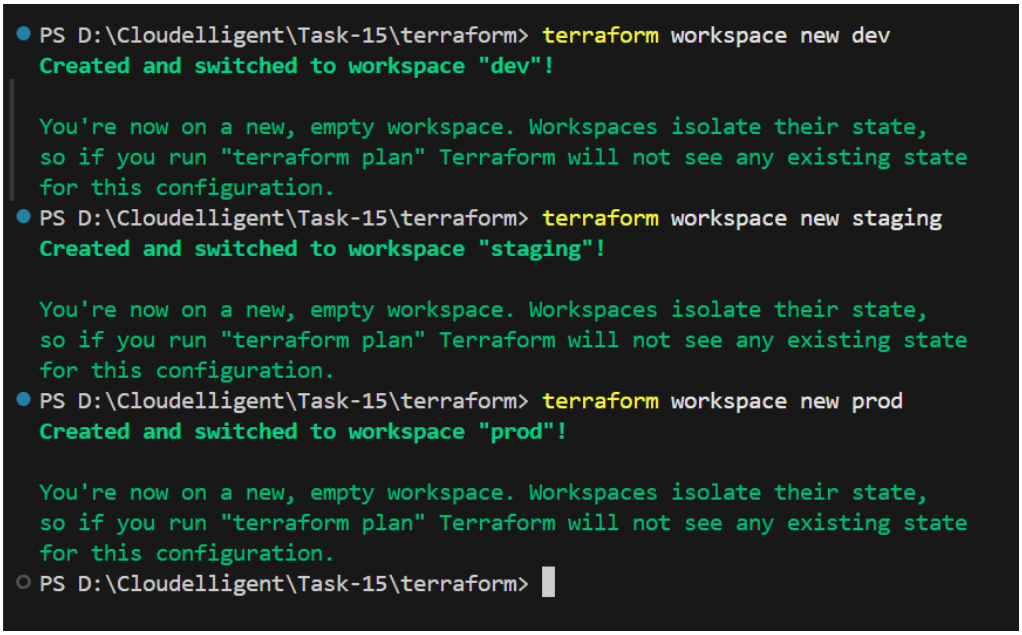
- *terraform workspace show*
- Should display "default" workspace



Step 2: Create Multiple New Workspaces

Use the following commands to create new workspaces

- *terraform workspace new dev*
- *terraform workspace new staging*
- *terraform workspace new prod*



New Terraform workspaces named dev, staging, and prod were created using the **terraform workspace new <NAME>** command. Terraform automatically switches to the newly created workspaces and creates separate state file for each of these workspaces.

4. Switch Between Terraform Workspaces

Terraform allows you to switch between environments (workspaces) without changing configuration files. When you select a workspace:

- Terraform starts using that workspace's **state file**
- Any **plan** or **apply** workflow affects only that environment
- Useful when managing multiple environments like dev, staging, and prod

Step 1: List Available Workspaces

Verify that the workspaces exist using ***terraform workspace list*** command. Expected output should be the list of workspaces created in previous section including default. The asterisk (*) indicates the currently active workspace.

```
PS D:\Cloudelligent\Task-15\terraform> terraform workspace list
  * default
    dev
    prod
    staging
```

Step 2: Switch to the Default Workspace

i. Switch from **default** to **dev** using:

- ***terraform workspace select dev***

ii. Switch from **default** to **staging** using:

- ***terraform workspace select staging***

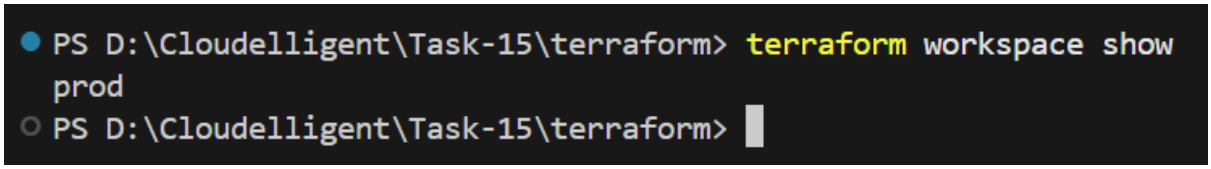
iii. Switch from **staging** to **prod** using:

- ***terraform workspace select prod***

```
PS D:\Cloudelligent\Task-15\terraform> terraform workspace select dev
Switched to workspace "dev".
PS D:\Cloudelligent\Task-15\terraform> terraform workspace select staging
Switched to workspace "staging".
PS D:\Cloudelligent\Task-15\terraform> terraform workspace select prod
Switched to workspace "prod".
PS D:\Cloudelligent\Task-15\terraform>
```

Step 3: Verify Current Workspace

Confirm the active workspace using ***terraform workspace show*** command.



Terraform allows switching between workspaces using the terraform workspace select command. This enables working in different environments using the same Terraform configuration while keeping their state files isolated.

5. List All Available Terraform Workspaces

Terraform maintains a list of all workspaces associated with a configuration.
The **terraform workspace list** command:

- Displays all available workspaces
- Highlights the currently active workspace
- Helps confirm environment availability before switching
- The active workspace is marked with an asterisk (*)

Run the Workspace List Command

In the project directory, run ***terraform workspace list*.** It should display all the workspaces created in the previous section.

```
PS D:\Cloudelligent\Task-15\terraform> terraform workspace list
  default
  dev
* prod
  staging
```

The terraform workspace list command was used to display all available workspaces. The currently active workspace is indicated by an asterisk (*), helping identify which environment Terraform is operating in.

Note: *** prod** indicates that **prod** is the current workspace.

6. Workspaces To Manage Multiple Environments

Terraform workspaces allow a single Terraform configuration to manage multiple environments by maintaining separate state files for each workspace.

- The same code is reused
- Each environment is isolated
- Changes in one workspace do **not** affect others

Step 1: Add main.tf file

The main.tf file detects the active workspace and then outputs the current environment name using **locals** and **output** block.

Step 2: Terraform Apply in different Workspaces

1. Switch to **dev** workspace and perform apply.
   - ***terraform workspace select dev***
   - ***terraform apply***

```
PS D:\Cloudelligent\Task-15\terraform> terraform workspace select dev
Switched to workspace "dev".
PS D:\Cloudelligent\Task-15\terraform> terraform apply

Changes to Outputs:
  + current_environment = "dev"

You can apply this plan to save these new output values to the Terraform state, without changing any real infrastructure.

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

  Enter a value: yes


Apply complete! Resources: 0 added, 0 changed, 0 destroyed.

Outputs:

current_environment = "dev"
PS D:\Cloudelligent\Task-15\terraform>
```

2. Switch to **staging** workspace and perform apply.

- ***terraform workspace select staging***
- ***terraform apply***

```
PS D:\Cloudelligent\Task-15\terraform> terraform workspace select staging
Switched to workspace "staging".
PS D:\Cloudelligent\Task-15\terraform> terraform apply

Changes to Outputs:
  + current_environment = "staging"

You can apply this plan to save these new output values to the Terraform state, without changing any real infrastructure.

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

  Enter a value: yes


Apply complete! Resources: 0 added, 0 changed, 0 destroyed.

Outputs:

current_environment = "staging"
PS D:\Cloudelligent\Task-15\terraform>
```

3. Switch to **prod** workspace and perform apply.

- ***terraform workspace select prod***
- ***terraform apply***

```
PS D:\Cloudelligent\Task-15\terraform> terraform workspace select prod
Switched to workspace "prod".
PS D:\Cloudelligent\Task-15\terraform> terraform apply

Changes to Outputs:
  + current_environment = "prod"

You can apply this plan to save these new output values to the Terraform state, without changing any real infrastructure.

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

  Enter a value: yes


Apply complete! Resources: 0 added, 0 changed, 0 destroyed.

Outputs:

current_environment = "prod"
PS D:\Cloudelligent\Task-15\terraform>
```

This highlights how Terraform workspaces eliminate code duplication while maintaining environment isolation through separate state files.

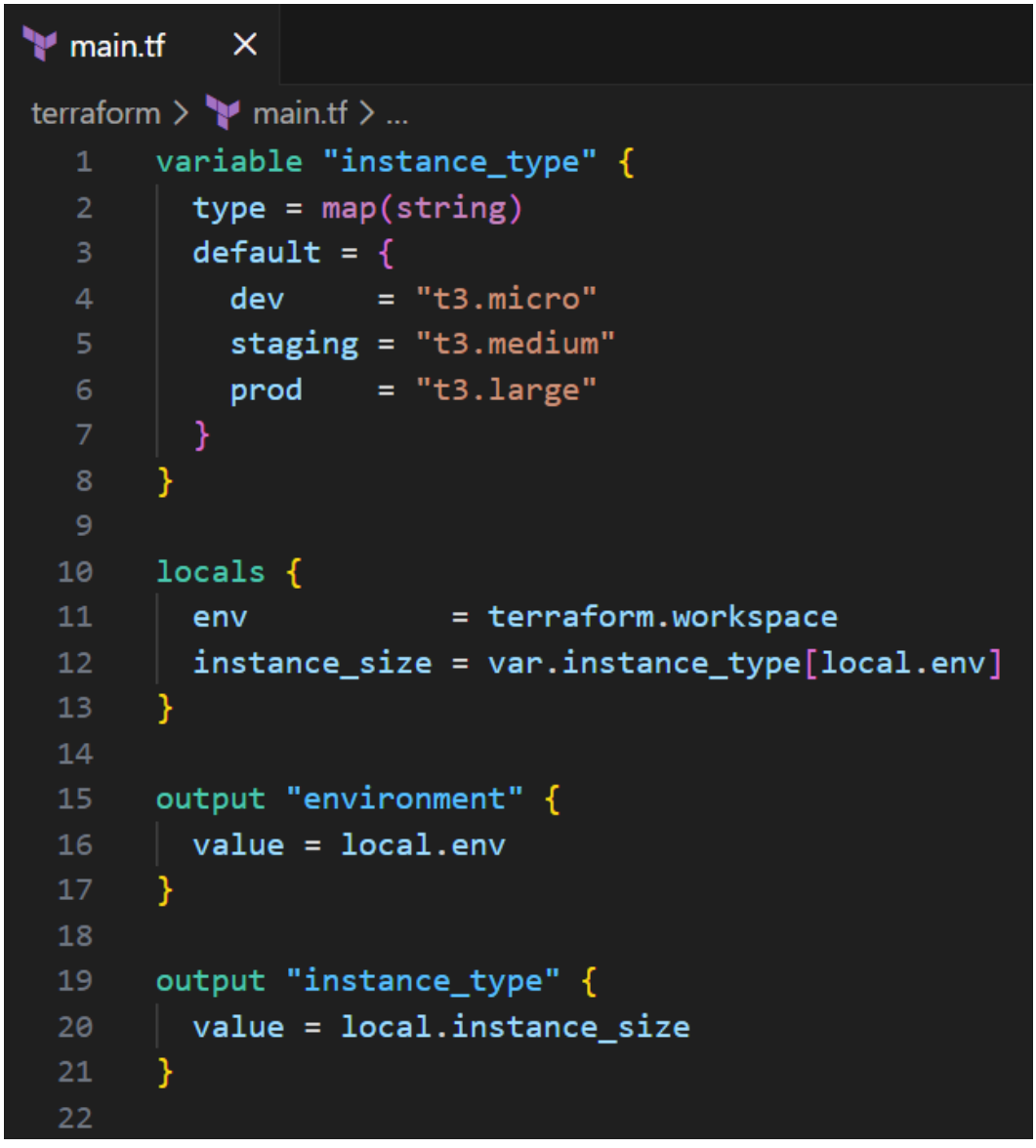
7. Workspace-Specific Variables and State Files

Terraform workspaces allow configurations to behave differently based on the **active workspace**. This can be achieved using:

- terraform.workspace
- Conditional logic
- Workspace-based variable maps

Each workspace continues to maintain its **own state file**, ensuring full isolation. The following steps utilize workspace-specific using the **terraform.workspace**.

Step 1: Update main.tf file



This configuration uses workspace-specific variables by detecting the active workspace and assigning values based on the workspace name.

Step 2: Apply in dev Workspace

The output shows **dev** workspace and its specific instance type variable **t3.micro**.

```
PS D:\Cloudelligent\Task-15\terraform> terraform workspace select dev
Switched to workspace "dev".
PS D:\Cloudelligent\Task-15\terraform> terraform apply -auto-approve

No changes. Your infrastructure matches the configuration.

Terraform has compared your real infrastructure against your configuration and found no differences, so no changes are needed.

Apply complete! Resources: 0 added, 0 changed, 0 destroyed.

Outputs:

environment = "dev"
instance_type = "t3.micro"
PS D:\Cloudelligent\Task-15\terraform>
```

Step 3: Apply in staging Workspace

The output shows **staging** workspace and its specific instance type variable **t3.medium**.

```
PS D:\Cloudelligent\Task-15\terraform> terraform workspace select staging
Switched to workspace "staging".
PS D:\Cloudelligent\Task-15\terraform> terraform apply -auto-approve

No changes. Your infrastructure matches the configuration.

Terraform has compared your real infrastructure against your configuration and found no differences, so no changes are needed.

Apply complete! Resources: 0 added, 0 changed, 0 destroyed.

Outputs:

environment = "staging"
instance_type = "t3.medium"
PS D:\Cloudelligent\Task-15\terraform>
```

Step 4: Apply in prod Workspace

The output shows **prod** workspace and its specific instance type variable **t3.large**.

```
PS D:\Cloudelligent\Task-15\terraform> terraform workspace select prod
Switched to workspace "prod".
PS D:\Cloudelligent\Task-15\terraform> terraform apply -auto-approve

Changes to Outputs:
  - current_environment = "prod" -> null
  + environment         = "prod"
  + instance_type       = "t3.large"

You can apply this plan to save these new output values to the Terraform state, without changing any real infrastructure.

Apply complete! Resources: 0 added, 0 changed, 0 destroyed.

Outputs:

environment = "prod"
instance_type = "t3.large"
PS D:\Cloudelligent\Task-15\terraform>
```

Step 5: Verification of Workspace-Specific State Files

After applying the Terraform configuration in the dev, staging, and prod workspaces, the Terraform root directory was inspected to verify how state files are managed internally.

Terraform automatically creates a directory named **terraform.tfstate.d** when multiple workspaces are used with a local backend. Each workspace is assigned its own subdirectory inside this folder.

Using the **tree /f** command, the directory structure was displayed as shown below:

```
PS D:\Cloudelligent\Task-15\terraform\terraform.tfstate.d> tree /f
Folder PATH listing for volume New Volume
Volume serial number is 5A5C-D3FC
D:.
├───dev
        terraform.tfstate
├───prod
        terraform.tfstate
└───staging
        terraform.tfstate
```

This confirms that:

- Each workspace (dev, staging, prod) maintains its **own independent terraform.tfstate file**
- Changes applied in one workspace do **not affect** the state of another workspace
- The same Terraform configuration can safely manage multiple environments without state conflicts

This structure demonstrates how Terraform workspaces provide environment isolation by separating state files while reusing the same infrastructure code.

8. Workspaces with Remote Backends and State Lock

8.1 Backends in Terraform

Backends in Terraform are responsible for managing the storage and state of infrastructure deployments. They define where and how Terraform's state data is stored, how it can be accessed, and who can access it, ensuring the state is preserved across multiple runs.

8.2 Key Features

1. State storage

The main role of Terraform backends is to store your Terraform state file safely in a place where Terraform can access, store, update, and delete it (if necessary). Backends determine how the state data is loaded and how the state is updated.

2. State locking

State locking allows your IaC runner to lock the state file while running your Terraform code, so it cannot be updated until it completes its run, successfully or not. This guarantees that nothing can edit your state file and cause conflicts.

8.3 Types of Terraform Backends

1. Local Backend

A local backend stores the state file on the machine where Terraform is running. This is the default backend that is used if you don't specify a backend in your Terraform configuration. The local backend is useful for testing and development environments, but it's not recommended for production environments since it can lead to inconsistencies if the state file is lost or corrupted.

2. Remote Backend

A remote backend stores the state file in a centralized location, such as a cloud object storage service or a database. Remote backends provide several benefits, such as enabling collaboration between team members, versioning state files, and providing a history of changes. There are several remote backend providers available, such as ***Amazon S3, Azure Storage, Google Cloud Storage, and HashiCorp*** Consul.

How Workspaces Work with Remote Backends

- Each workspace maintains a **separate state file** in the remote backend
- Terraform automatically prefixes the workspace name to the remote state
- Switching workspaces changes which state file Terraform reads and writes

Example with S3 backend

- Dev: ***s3://my-terraform-bucket/dev/terraform.tfstate***
- Staging: ***s3://my-terraform-bucket/staging/terraform.tfstate***
- Prod: ***s3://my-terraform-bucket/prod/terraform.tfstate***

This ensures safe separation of environment states.

Note: The actual S3 key prefix depends on the backend configuration. Terraform automatically manages workspace-specific paths internally.

Terraform workspaces integrate seamlessly with remote backends. Each workspace maintains its own state file in the remote backend, enabling safe multi-environment management. Remote state locking prevents concurrent modifications, ensuring team collaboration does not corrupt Terraform state.

9. Multi-Environment VPC with EC2 Using Modules and Workspaces

9.1 Overview

In this section, Terraform workspaces are used to provision and manage multiple isolated environments i.e. **dev**, **staging**, and **prod**, using a single Terraform codebase. Each workspace maintains its own state file, allowing the same infrastructure definitions to create independent AWS resources per environment.

Each workspace provisions its own VPC, Internet Gateway, subnet, route table, security group, and EC2 instance. Resource naming and tagging dynamically reference the active workspace using **terraform.workspace**, ensuring clear identification and isolation of all environments.

The following AWS resources are created:

- VPC and Internet gateway
- Public Subnet (different CIDR and AZ per environment)
- Route Table
- Security Group
- EC2 Instance (different instance sizes and storage)

Each environment was fully isolated at the network and compute level while being deployed from the same Terraform configuration.

9.2 Workspace Creation

Three new Terraform workspaces were created to represent different environments:

- dev
- staging
- prod

Terraform automatically isolated state for each workspace, ensuring that resources created in one environment do not affect the others.

```
PS D:\Cloudelligent\Task-15\terraform> terraform workspace list
* default

PS D:\Cloudelligent\Task-15\terraform> terraform workspace new dev
Created and switched to workspace "dev"!

You're now on a new, empty workspace. Workspaces isolate their state,
so if you run "terraform plan" Terraform will not see any existing state
for this configuration.
PS D:\Cloudelligent\Task-15\terraform> terraform workspace new staging
Created and switched to workspace "staging"!

You're now on a new, empty workspace. Workspaces isolate their state,
so if you run "terraform plan" Terraform will not see any existing state
for this configuration.
PS D:\Cloudelligent\Task-15\terraform> terraform workspace new prod
Created and switched to workspace "prod"!

You're now on a new, empty workspace. Workspaces isolate their state,
so if you run "terraform plan" Terraform will not see any existing state
for this configuration.
PS D:\Cloudelligent\Task-15\terraform> terraform workspace list
  default
  dev
* prod
  staging

PS D:\Cloudelligent\Task-15\terraform>
```

9.3 Environment-Specific Deployment

Each environment was deployed by selecting the appropriate workspace and applying a corresponding **.tfvars** file. The outputs include VPC ID, Subnet ID, EC2 Instance ID, EC2 Public IP, and Environment.

Dev Environment

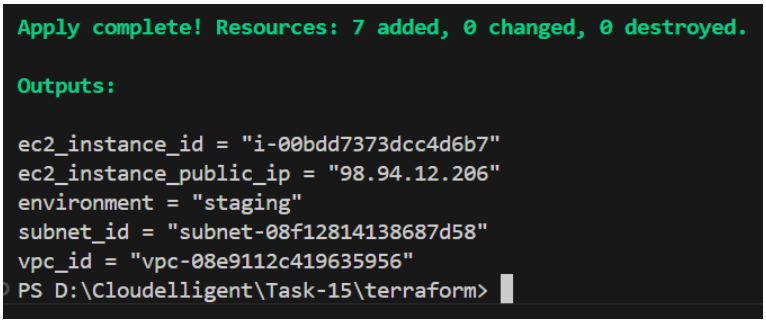
- Workspace name: dev
- Commands used:
  - *terraform workspace select dev*
  - *terraform plan -var-file="dev.tfvars"*
  - *terraform apply -auto-approve -var-file="dev.tfvars"*
- The outputs after terraform apply are shown below

```
Outputs:

ec2_instance_id = "i-03c2a75d87d2fecc3"
ec2_instance_public_ip = "3.82.201.219"
environment = "dev"
subnet_id = "subnet-0244e11c117eca549"
vpc_id = "vpc-0ea9ffa55698bee8e"
PS D:\Cloudelligent\Task-15\terraform>
```

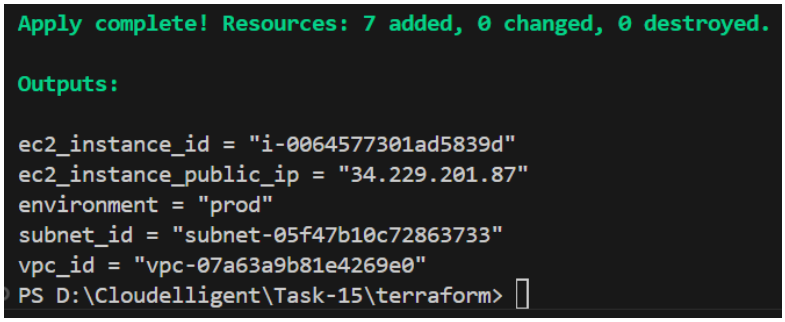
Staging Environment

- Workspace name: staging
- Command used:
  - `terraform workspace select staging`
  - `terraform plan -var-file="staging.tfvars"`
  - `terraform apply -auto-approve -var-file="staging.tfvars"`
- The outputs after terraform apply are shown below



Production Environment

- Workspace name: prod
- Command used:
  - `terraform workspace select prod`
  - `terraform plan -var-file="prod.tfvars"`
  - `terraform apply -var-file="prod.tfvars"`
- The outputs after terraform apply are shown below

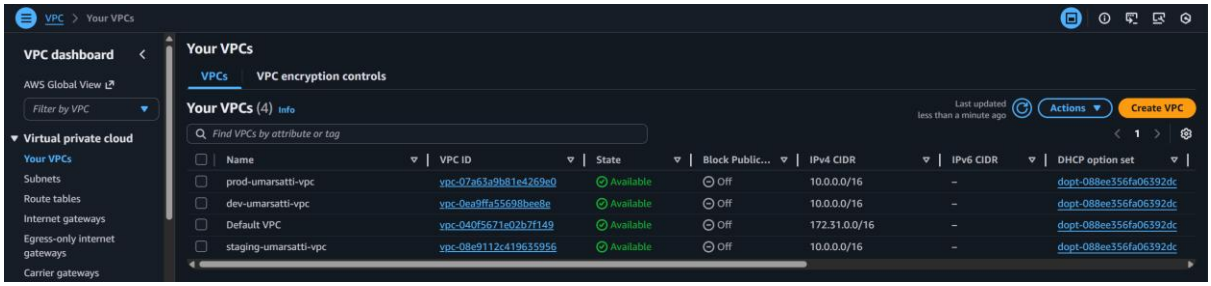


9.4 Infrastructure Verification in AWS Console

The following AWS resources were verified through the AWS Management Console.

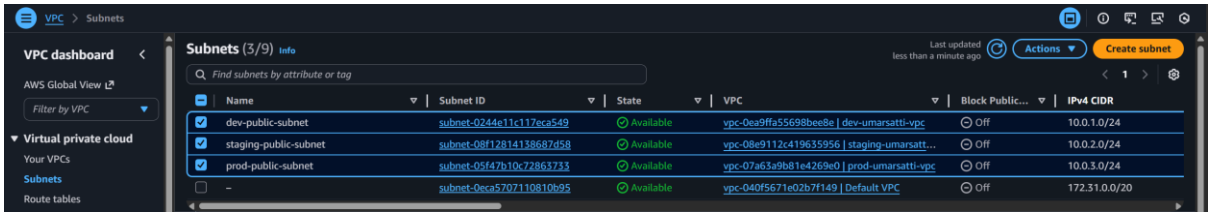
VPCs

- dev-umarsatti-vpc
- staging-umarsatti-vpc
- prod-umarsatti-vpc



Subnets

- dev-public-subnet
- staging-public-subnet
- prod-public-subnet

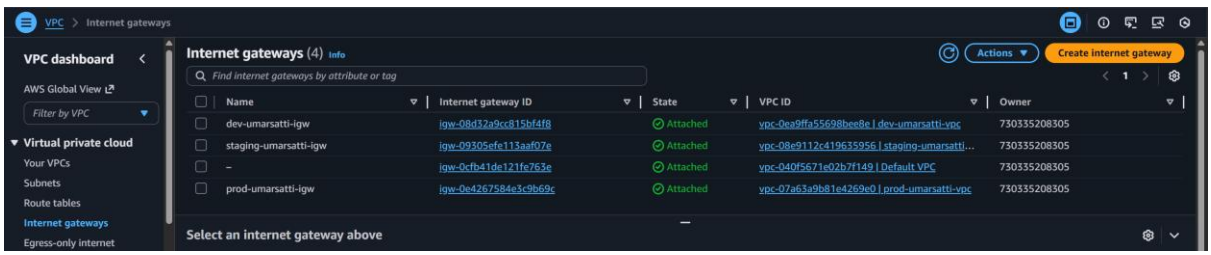


Route Tables

- dev-public-rt
- staging-public-rt
- prod-public-rt

Internet Gateways

- dev-umarsatti-igw
- staging-umarsatti-igw
- prod-umarsatti-igw

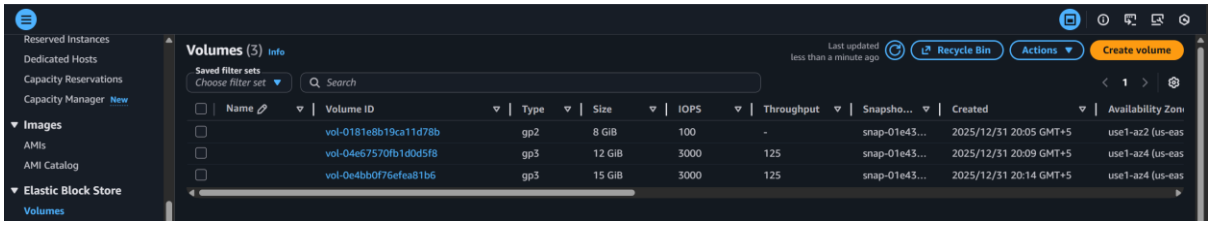


Security Groups

- dev-sg
- staging-sg
- prod-sg

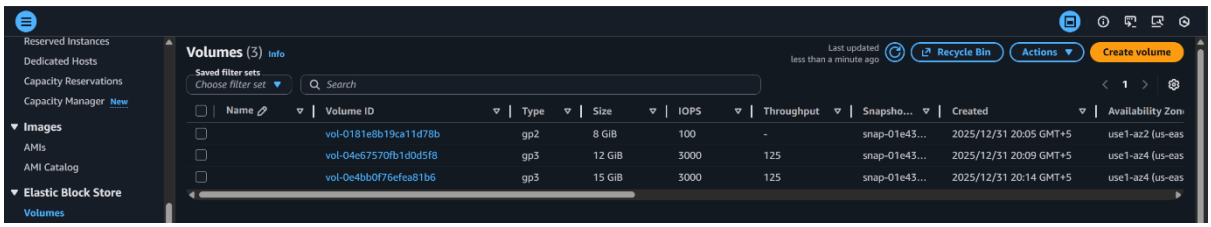
EC2 Instances

- dev-ec2
- staging-ec2
- prod-ec2



EBS Volumes

- Dev: 8 GiB (gp2)
- Staging: 12 GiB (gp3)
- Prod: 15 GiB (gp3)

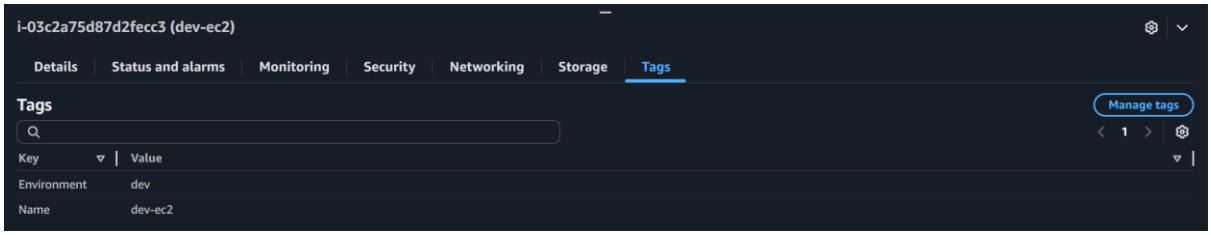


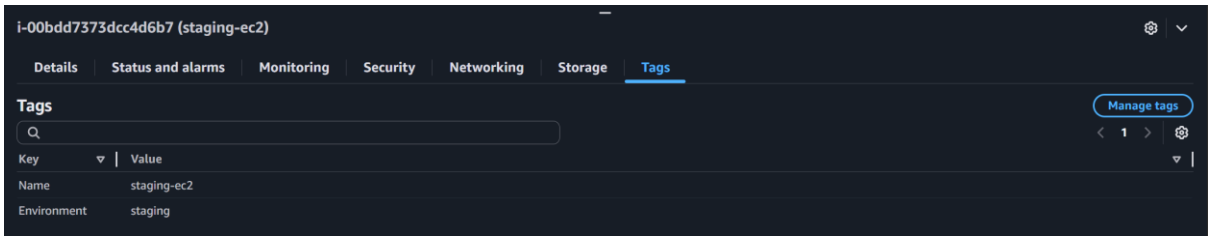
9.5 Workspace-Based Tagging

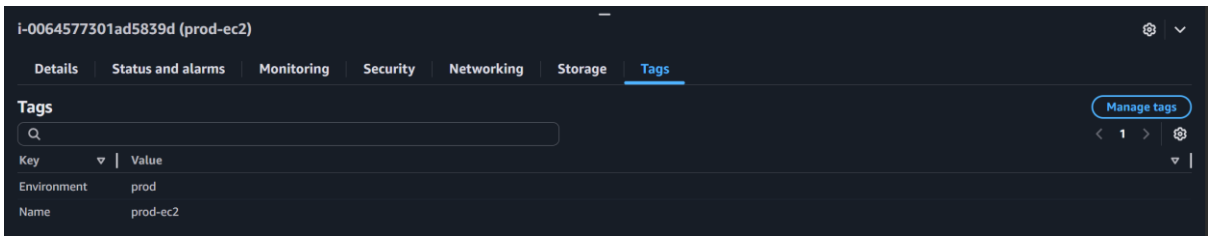
Each EC2 instance was tagged dynamically using the active workspace as shown by the terraform code below.

```
tags = {
  Name        = var.instance_name
  Environment = terraform.workspace
}
```

This ensured all resources were clearly labeled with their respective environment.







9.6 Application Validation Using User Data

A simple Apache web server was installed on each EC2 instance using Terraform user_data. The active workspace was interpolated into the web page content using local variables.

Local variables

```
locals {
  env = terraform.workspace
}
```

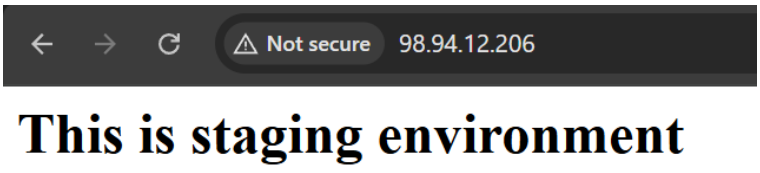
User Data Script

```
user_data = <<-EOF
            #!/bin/bash
            apt-get update -y
            apt-get install -y apache2
            systemctl start apache2
            systemctl enable apache2
            echo "<h1>This is ${local.env} environment</h1>" > /var/www/html/index.html
            EOF
```

When accessing each EC2 instance via its public IP address, the following environment-specific messages were displayed.

Dev workspace

Staging workspace



Prod workspace

This task successfully demonstrated how Terraform workspaces can be used to manage multiple environments with a single codebase. By isolating state per workspace and dynamically referencing **terraform.workspace**, environment-specific infrastructure was created, tagged, and validated without code duplication. This approach simplifies environment management while maintaining clear separation, consistency, and scalability.

10. Terraform Destroy

The infrastructure was destroyed using terraform destroy command. Since Terraform workspaces maintain separate state files, the terraform destroy command was executed individually per workspace to ensure only the resources belonging to that specific environment were deleted.

Dev Environment

```
PS D:\Cloudelligent\Task-15\terraform> terraform workspace select dev
Switched to workspace "dev".
PS D:\Cloudelligent\Task-15\terraform> terraform destroy -auto-approve -var-file="dev.tfvars"
```

Staging Environment

```
PS D:\Cloudelligent\Task-15\terraform> terraform workspace select staging
Switched to workspace "staging".
PS D:\Cloudelligent\Task-15\terraform> terraform destroy -auto-approve -var-file="staging.tfvars"
```

Prod Environment

```
PS D:\Cloudelligent\Task-15\terraform> terraform workspace select prod
Switched to workspace "prod".
PS D:\Cloudelligent\Task-15\terraform> terraform destroy -auto-approve -var-file="prod.tfvars"
```

Each command removed all AWS resources associated with the selected workspace, including the VPC, subnet, route table, Internet Gateway, security group, EC2 instance, and EBS volumes.

11. Terraform Workspace Deletion

After completing the infrastructure deployment and validation for all environments, Terraform workspaces were cleaned up using the terraform workspace delete command. This command permanently removes a workspace and its associated Terraform state, ensuring no unused environments remain.

Note: A workspace cannot be deleted while it is currently selected.

Steps to Delete Workspaces

Switch to a different workspace, ideally the default workspace. Then perform deletion commands on each workspace. The steps are shown below.

1. Switch to **default** workspace:
   - *terraform workspace select default*
2. Delete the **dev** workspace:
   - *terraform workspace delete dev*
3. Delete the **staging** workspace:
   - *terraform workspace delete staging*
4. Delete the **prod** workspace:
   - *terraform workspace delete prod*

```
PS D:\Cloudelligent\Task-15\terraform> terraform workspace select default
Switched to workspace "default".
PS D:\Cloudelligent\Task-15\terraform> terraform workspace delete dev
Releasing state lock. This may take a few moments...
Deleted workspace "dev"!
PS D:\Cloudelligent\Task-15\terraform> terraform workspace delete staging
Releasing state lock. This may take a few moments...
Deleted workspace "staging"!
PS D:\Cloudelligent\Task-15\terraform> terraform workspace delete prod
Releasing state lock. This may take a few moments...
Deleted workspace "prod"!
PS D:\Cloudelligent\Task-15\terraform>
```

Once deleted, the workspaces and their associated state files are removed. This confirms proper workspace lifecycle management and ensures that no unused Terraform environments remain after project completion.

References:

1. https://developer.hashicorp.com/terraform/cli/workspaces
2. https://developer.hashicorp.com/terraform/language/backend/s3
3. https://spacelift.io/blog/terraform-workspaces
4. https://dev.to/aws-builders/terraform-workspaces-and-multi-environment-deployments-12gb
5. https://medium.com/@b0ld8/terraform-manage-multiple-environments-63939f41c454#f409
6. https://medium.com/@surangajayalath299/what-is-terraform-backend-how-used-it-ea5b36f08396