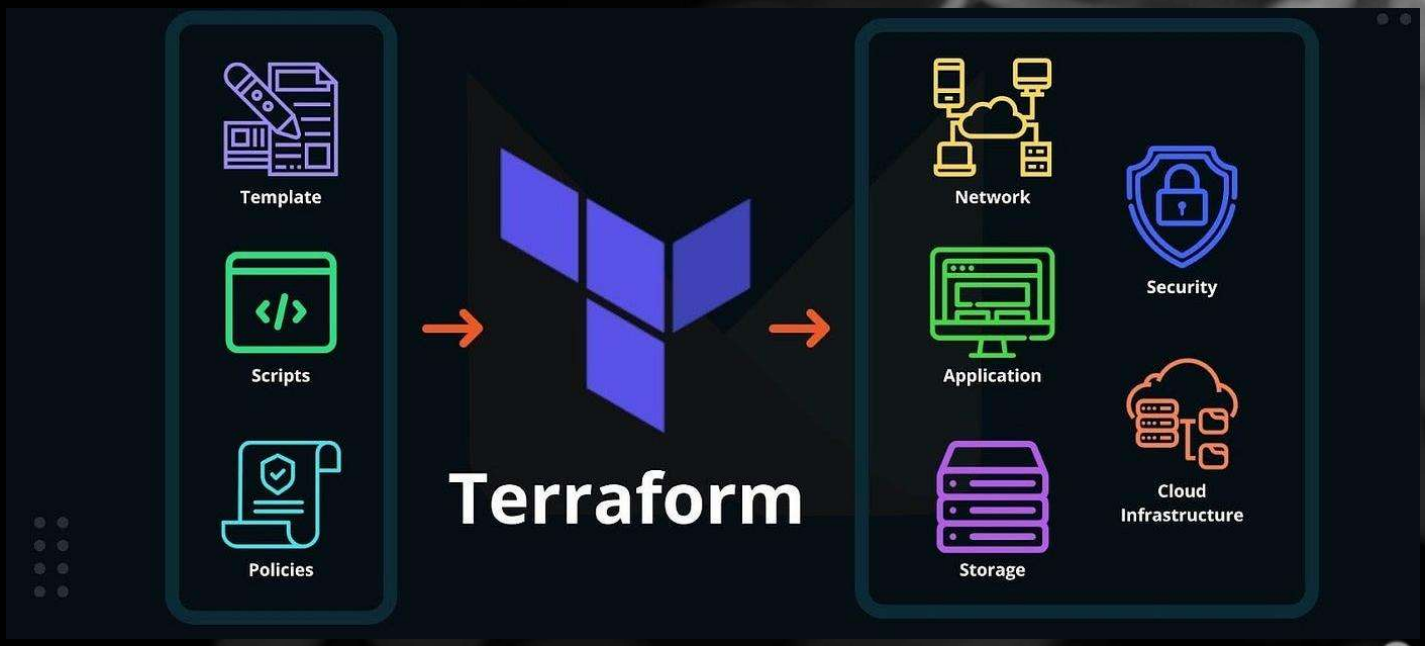
TERRAFORM

MASTERING INFRASTRUCTURE AS CODE



SUBHABRATA PANDA

ABOUT ME

I am **Subhabrata Panda**, a passionate **DevOps and Cloud Enthusiast**, with a strong foundation in **Terraform**, **AWS**, and cutting-edge cloud technologies.I have consistently demonstrated expertise in deploying scalable, secure, and efficient infrastructure solutions.

With hands-on experience implementing **Infrastructure as Code (IaC)** using Terraform, I have successfully worked on projects that include building secure CI/CD pipelines, managing multi-tier applications, and optimizing cloud deployments.

This e-book reflects my commitment to making complex infrastructure concepts accessible and actionable. Whether you are a beginner or an experienced professional, my goal is to provide you with the tools and insights needed to excel in modern cloud-based DevOps practices.

Feel free to connect with me at **panda.subhabrata2003@gmail.com** to discuss ideas, share feedback, or collaborate on exciting projects!

INDEX

PREREQUISITES

- Must have knowledge of **AWS** Cloud.
- Must know to handle **VS code** editor well.

INTRODUCTION

In this book/pdf we will learn all the **basic to advance** of the TERRAFORM tool that is mainly used in Devops with the help of **AWS cloud** and **VS Code editor** or **Cursor**.

Terraform is an open-source **Infrastructure as Code (Iac)** tool developed by **HashiCorp.**

TERRAFORM

Terraform is an **open-source** infrastructure as a code **(IAC)** tool, written in **HCL** (**Hashicorp Config Language**) format, which has State Management feature (*terraform.tfstate*)where it maintained a detailed recodes of current state of managed resources.

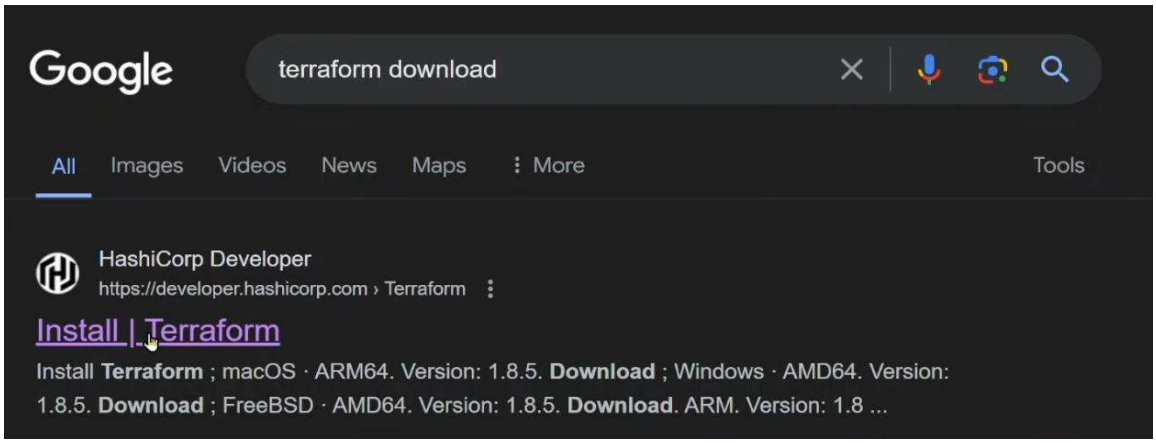
3. IAC

Tools allow you to manage infrastructure with configuration files rather than through a graphical user interface.

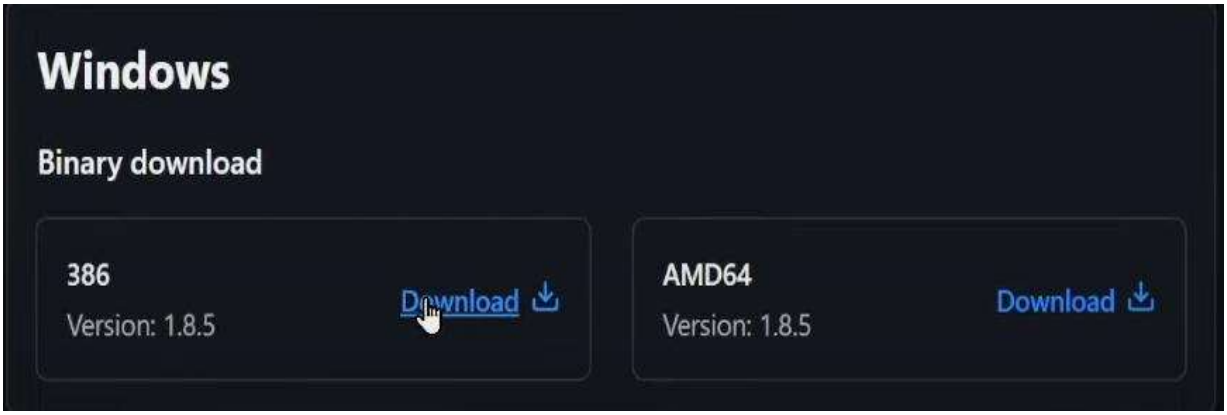
E.g.
→ You want to host the EC2 instance in AWS cloud, normally what we will do is to go to
→ dashboard
→setup up the instance step by step graphically

Rather than doing the graphically, write the simply configuration file that contains the details like
→ to host the instance
→ which os is required
→ what all resources is required for the instance
Then will simply execute the file, then IAC will automatically do all the things you asked for.

4. INSTALLATION ON TERRAFORM ON WINDOWS

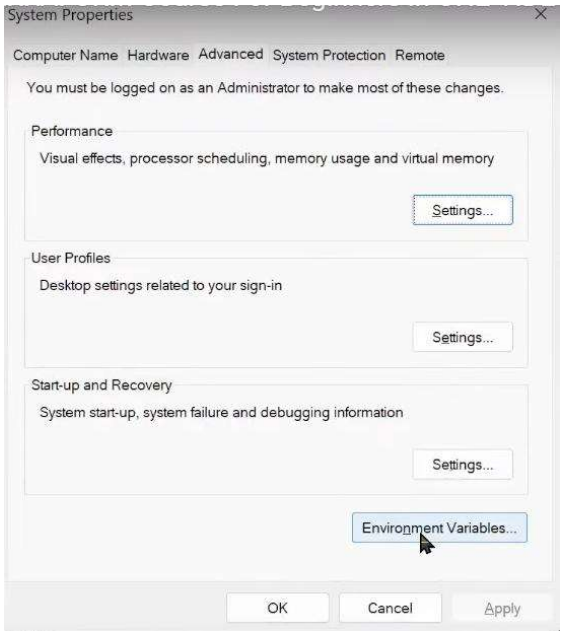


Scroll Down and search for windows section and download binary of terraform

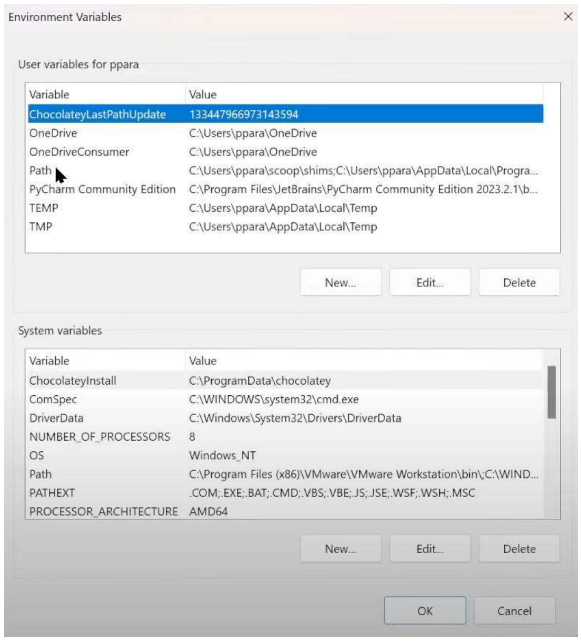


Extract the files that you have download

Search for “Edit the system environment variable” on windows search bar it will show something like



Click on “Environment Variables”



Click on “Path” where the arrow is showing
→then Select New option
→ add the terraform extracted file location then click ok
➔Then ok → ok

Under windows cmd terminal

```
C:\Users\subha>terraform version
Terraform v1.9.4
on windows_amd64
```

5. INSTALLATION ON TERRAFORM ON MAC/UBUNTU

MAC

brew tap hashicorp/tap brew install hashicorp/tap/terraform
Or download binary of Terraform

LINUX

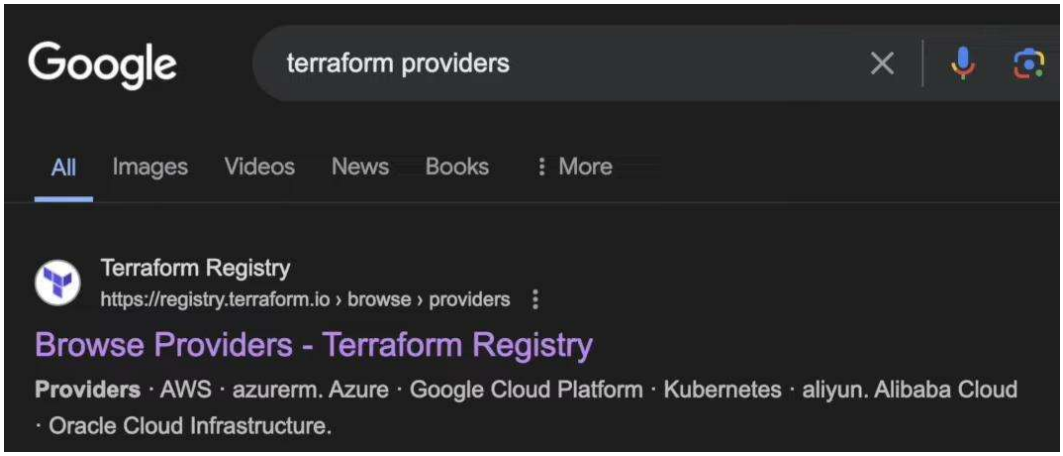
Ubuntu/Debian
wget -O - https://apt.releases.hashicorp.com/gpg | sudo gpg --dearmor -o /usr/share/keyrings/hashicorp-archive-keyring.gpg echo "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/hashicorp-archive-keyring.gpg] https://apt.releases.hashicorp.com $(lsb_release -cs) main" | sudo tee /etc/apt/sources.list.d/hashicorp.list sudo apt update && sudo apt install terraform

CentOS/RHEL
sudo yum install -y yum-utils sudo yum-config-manager --add-repo https://rpm.releases.hashicorp.com/RHEL/hashicorp.repo sudo yum -y install terraform

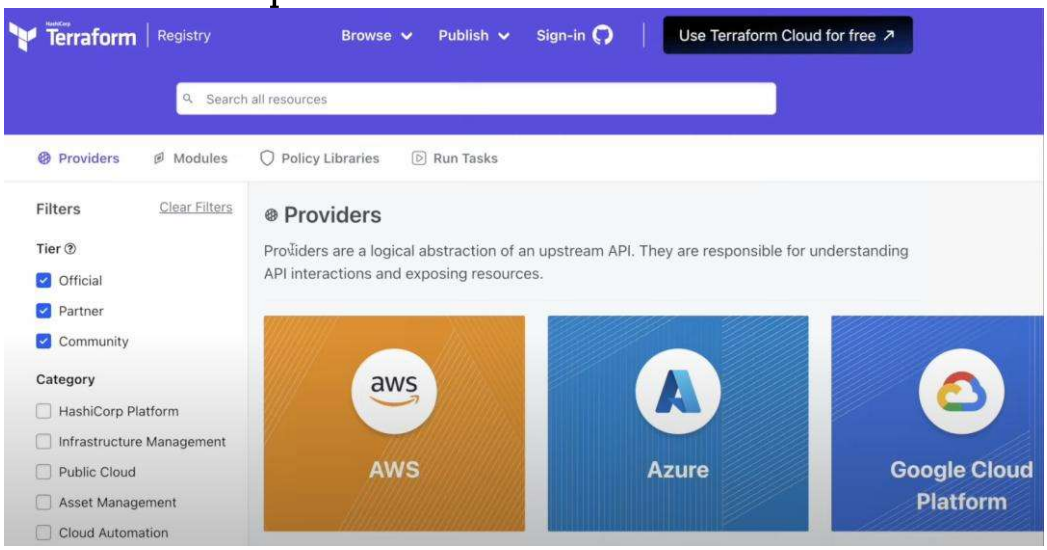
Amazon Linux
sudo yum install -y yum-utils shadow-utils sudo yum-config-manager --add-repo https://rpm.releases.hashicorp.com/AmazonLinux/hashicorp.repo sudo yum -y install terraform

6. CHOOSE THE PROVIDER

After the terraform is done now we need to choose which provider we will be working on to see which all provider is available, we need to go to –



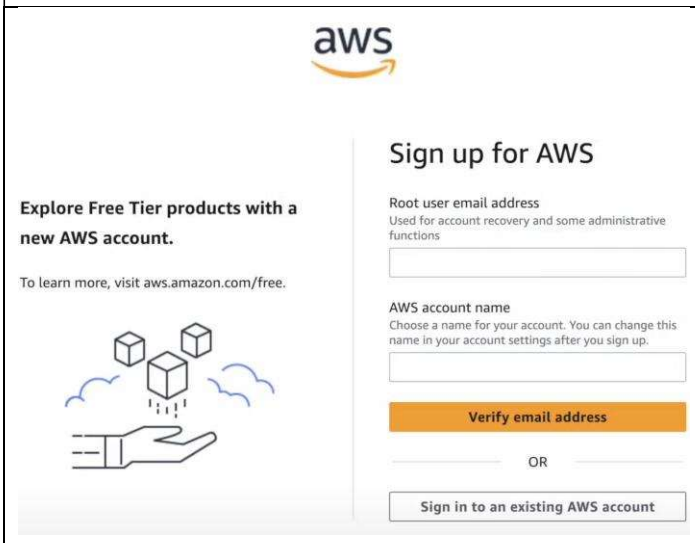
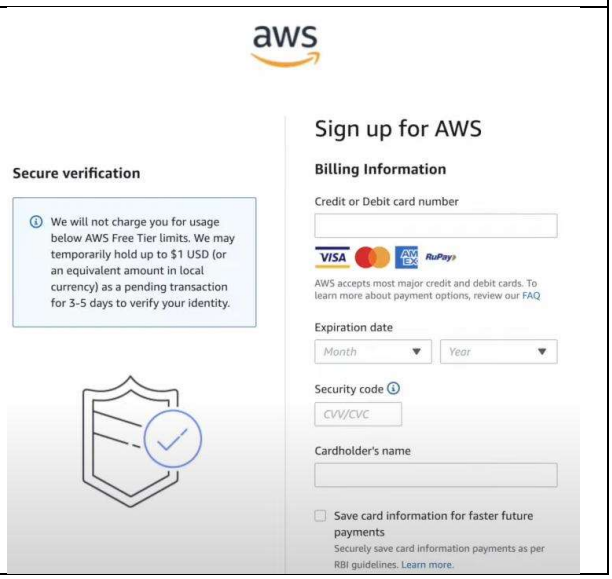
Search for the provider

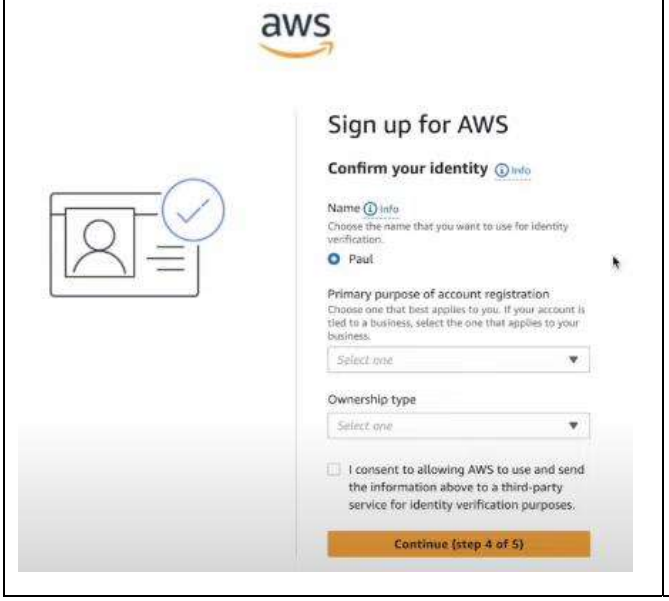
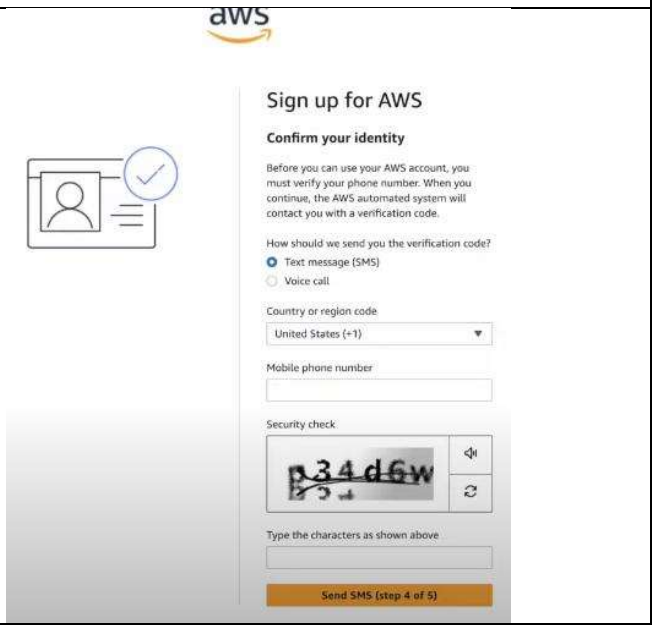


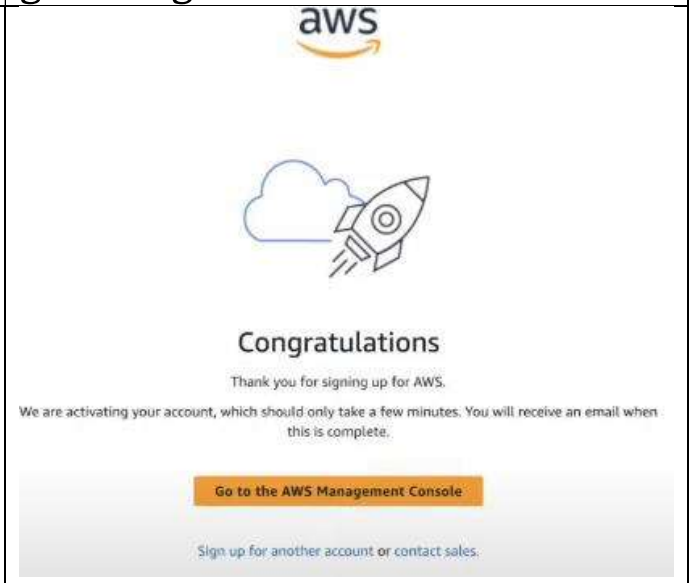
7. SETUP AWS ACCOUNT

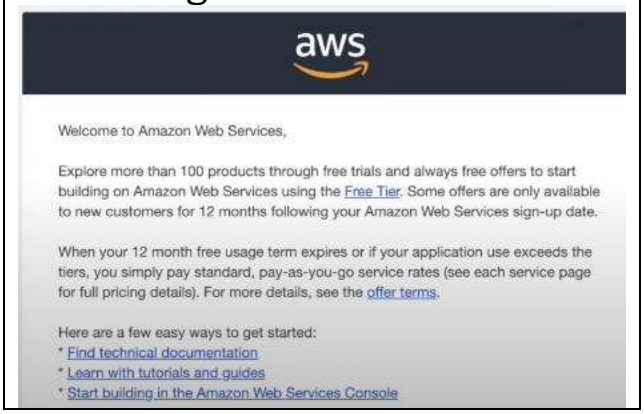
Go to this account aws.amazon.com

Fill all the details that the page is asking, all the details must be legal And at the end it will cost **1USD** for verification which will be returned automatically.

STEP-1	STEP-2
	

STEP-3	STEP-4
	

STEP-5	STEP-6
Select the basic support	If this page shows, then you are good to go
	

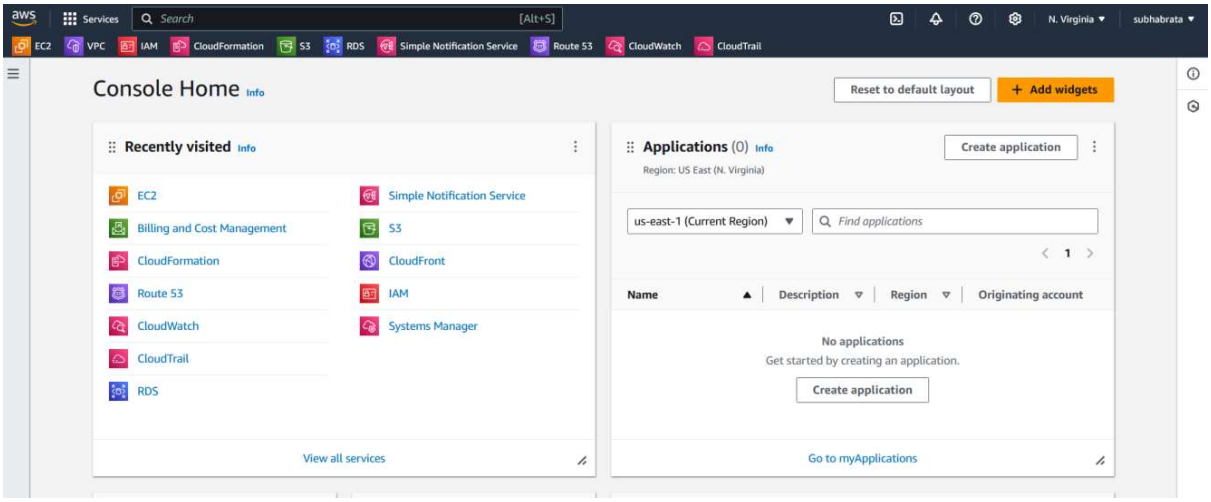
STEP-7
And you will also get the mail something like this


8. AWS DASHBOARD

The AWS Management Console Dashboard is a user-friendly, web-based interface for managing AWS services and resources. Key features include:

- Customizable Interface: Pin frequently used services.
- Resource Monitoring: View EC2, S3, and other resources immediately.
- Billing Management: Track usage, view bills, and set cost alerts.
- Integrated Tools: Access CloudWatch, CloudFormation, and IAM.
- Global Management: Manage resources across regions and zones.

It provides real-time insights, centralized management, and tools for monitoring, security, and cost optimization, simplifying cloud operations for all users.



9. AWS USER SETUP

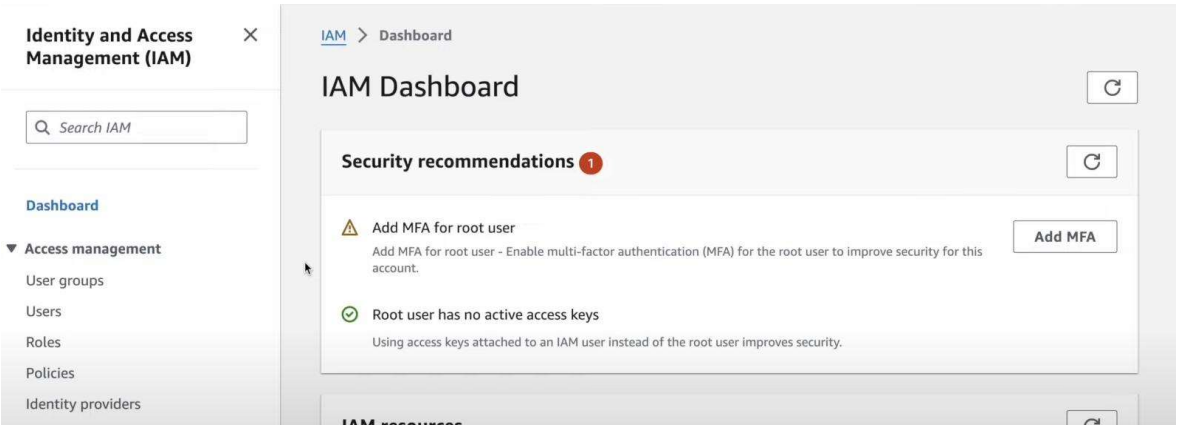
Now we need to do user setup because we need accessibility from user as we will be performing action on the local machine for AWS.

So, we will be using service known as **IAM (Identity and Access Manager)**

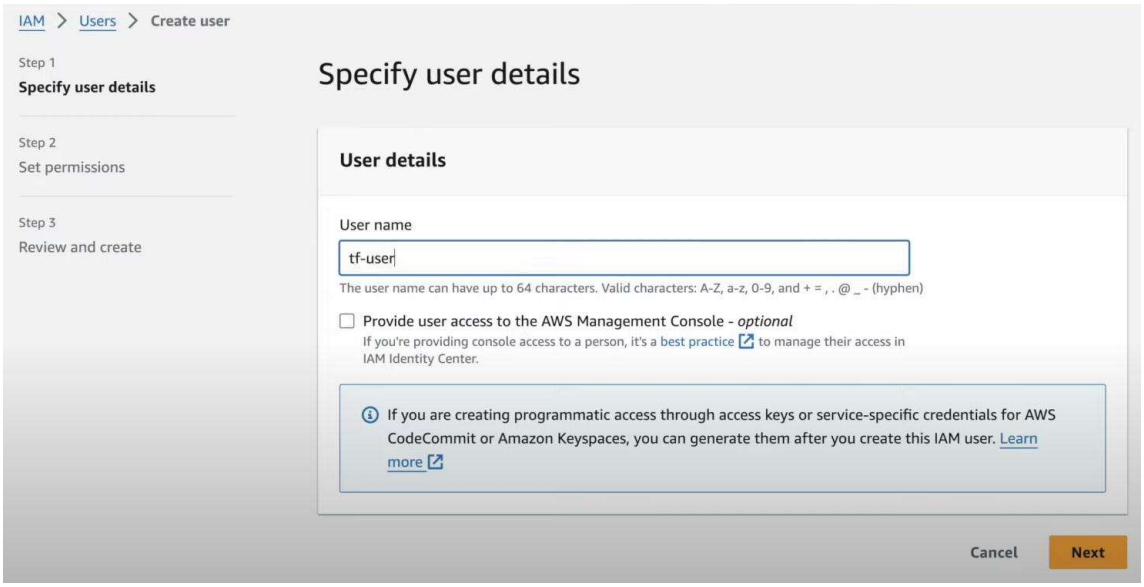
IAM → AWS Identity and Access Management (IAM) is a critical service that enables you to securely control access to AWS resources. It provides fine-grained access management for AWS services and resources, ensuring that only authorized users and applications can interact with your cloud infrastructure.

IAM is a foundational security service used across all AWS environments, including services like Amazon EKS, EC2, S3, RDS, and others.

Go to search bar of AWS and type IAM you will see the dashboard something like this

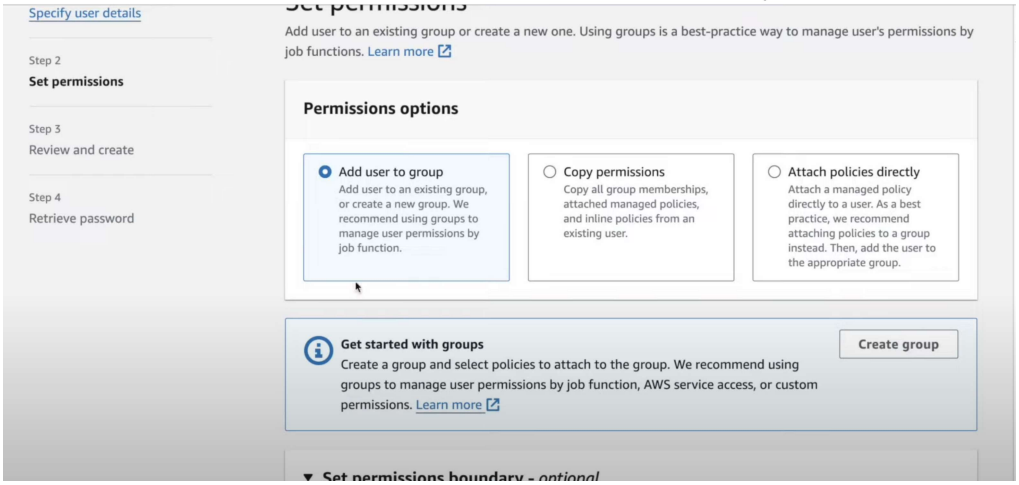


Now click on “User” → Give the User name → And Click on “Next”



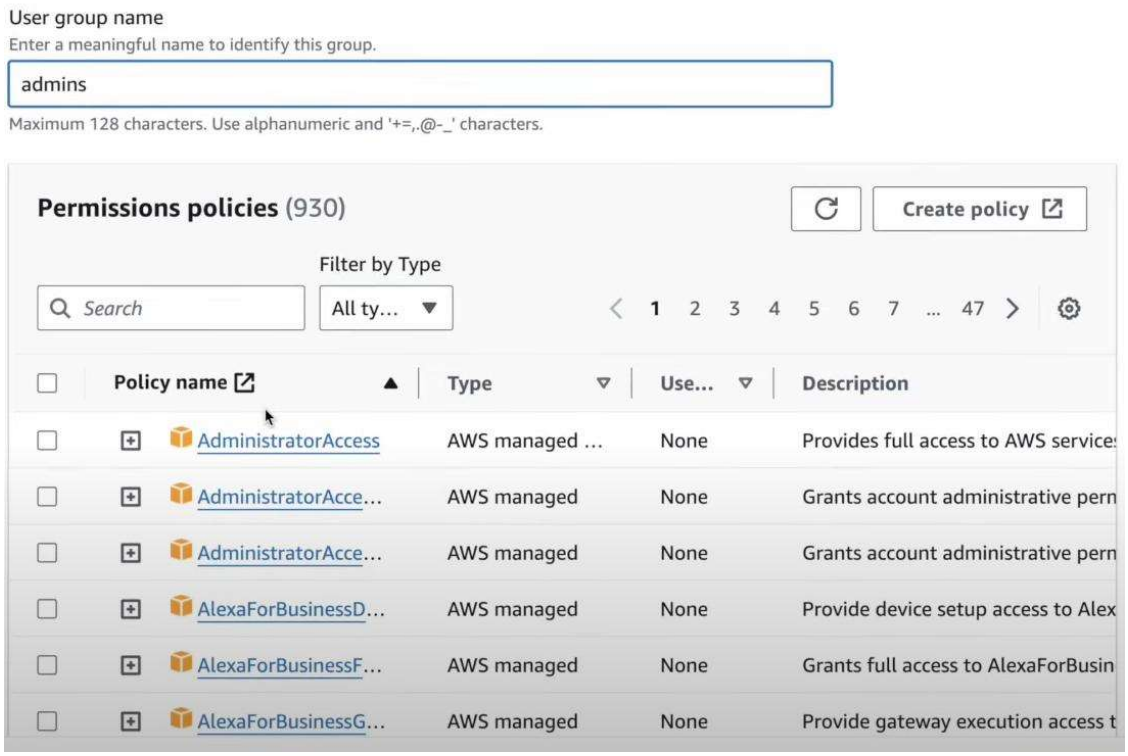
Tick on “Provide user access AWS Management console”
→Then tick on “I want to create an IAM user”
→Now Tick and Give the “Custom password”
→For now, untick “User must create a new password at next sign-in”
→ And click on “Next”

Under “Set Permission”



→Select “Add user to group”.
→Click on “Create group”.

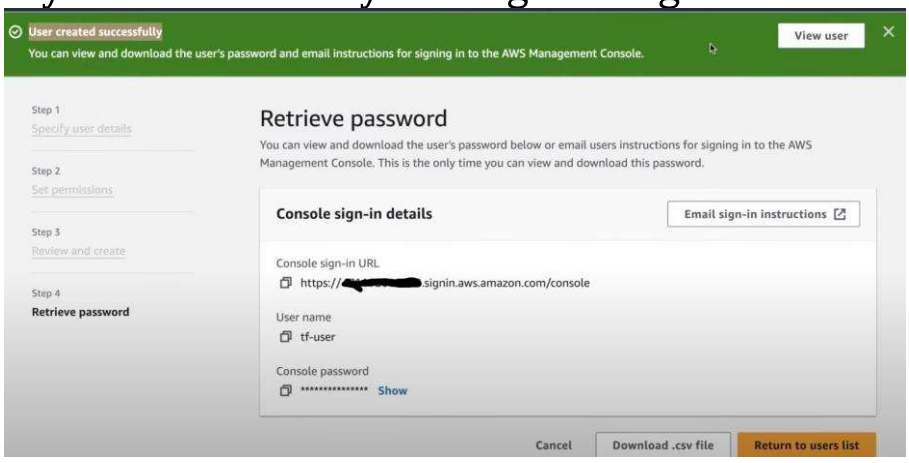
Next this page will appear



→For now, Tick “AdministratiorAccess”
→Click “Create user group”

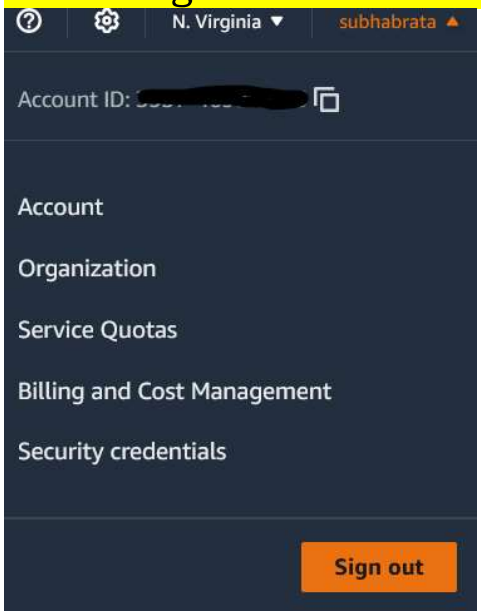
→Then click on “Next”
→If you want to give tag, then give or click on create user.

If you see this then you are good to go

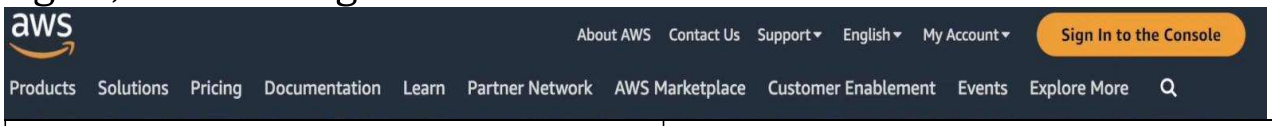


→remember or copy the “User name”, and “sign-in URL”, “Console Password”
→Then click on “Return to user list”

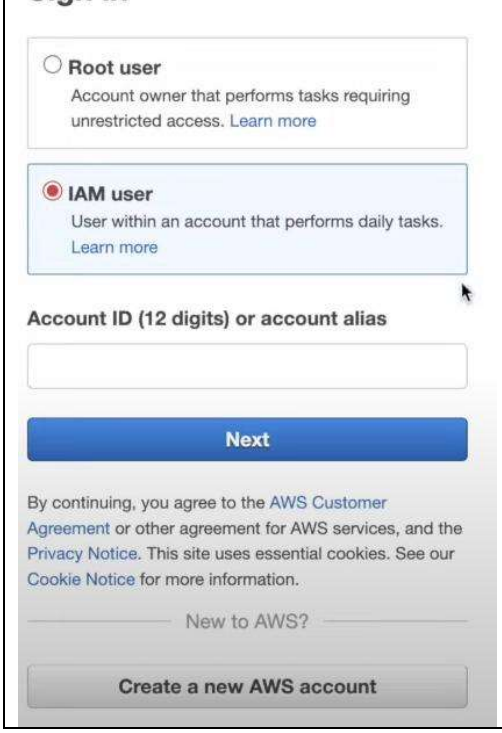
→Now sign-out from the “Root id”

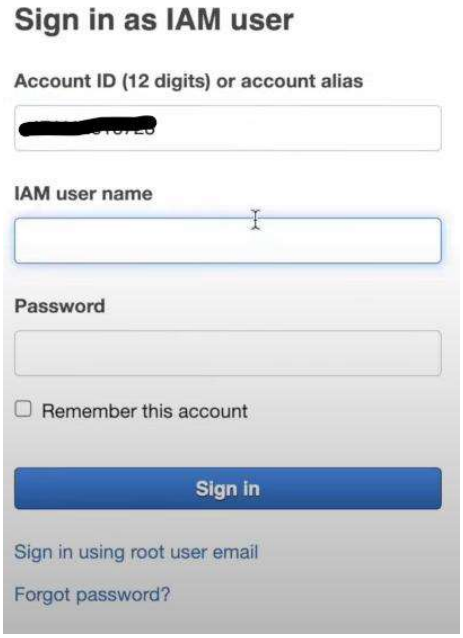


Again, click on “Sign-in to the Console”



STEP -1	STEP-2
This icon will appear	→Give the 12digit account

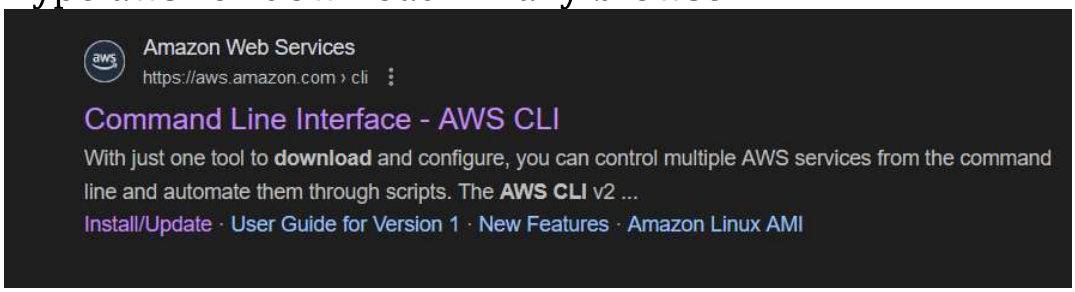




→Give the details and sign in.
→It will sign as a user not a root user

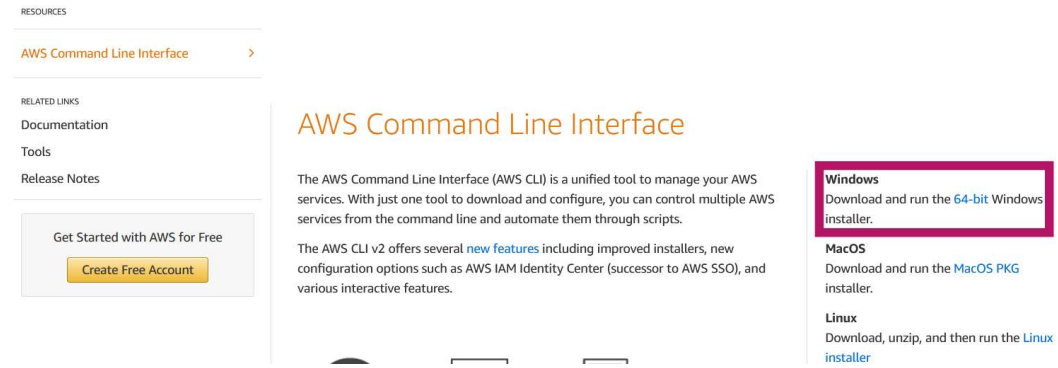
10. DOWNLOADING AWS CLI ON WINDOWS

Type aws “cli download” in any browser

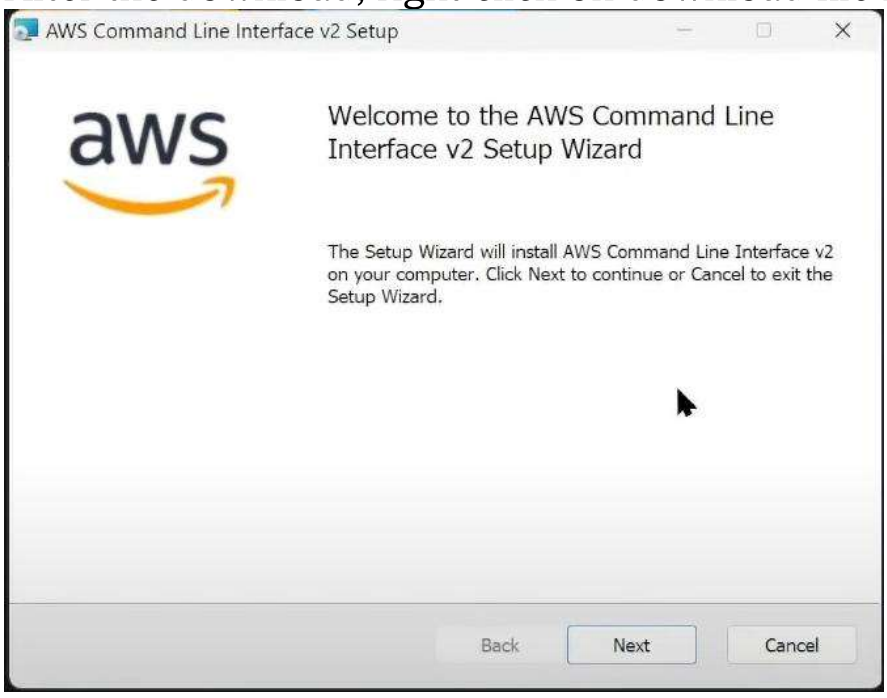


Click on the given link

Click on 64-bit and it will automatically download it

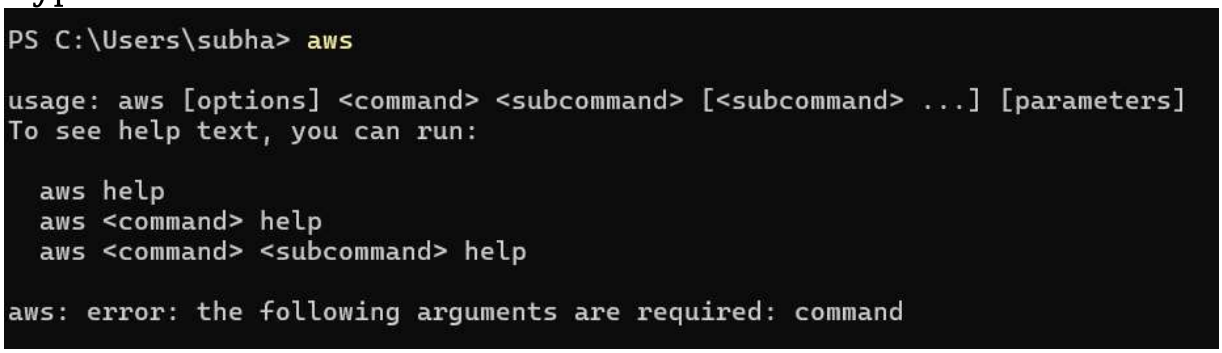


After the download, right click on download file it will show



Just do next→next→ Accept the License Agreement→next→ install

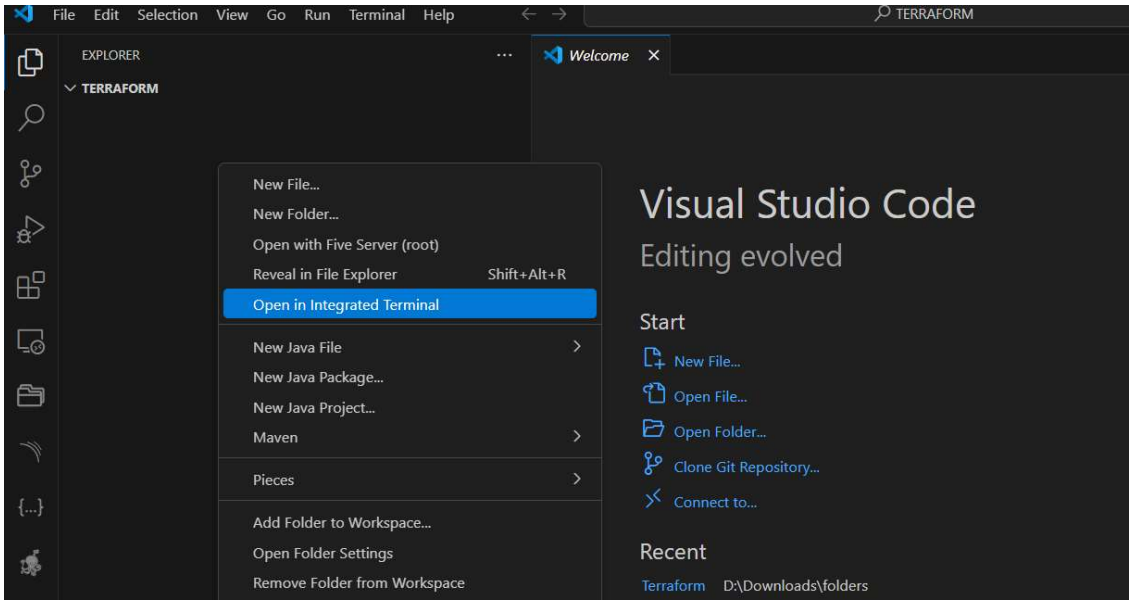
Type “aws” on the windows cmd



If you see this screen means you have successfully installed the aws cli. Now, we can access AWS cloud platform from a local environment.

11. AWS CLI CONFIGURE IN VS CODE

All the configuration we will made is in VS code
→Create an empty folder name “Terraform”
→Right click inside Terraform box. To open integrated terminal in VS code.



Support to check something from the VS code terminal i.e. what all users that have created we type

→> *aws iam list-users*

By default, it will show
`Unable to locate credentials. You can configure credentials by running "aws configure".`

But in my case, it shows

`PS D:\Downloads\TERRAFORM> aws iam list-users`
 	
`An error occurred (InvalidClientTokenId) when calling the ListUsers operation: The security token included in the request is invalid.`
`PS D:\Downloads\TERRAFORM>`

NOTE:: - It's not an error it's because I have deleted my user manually from AWS IAM service

Other reasons

→Invalid or Expired Credentials

→Incorrect AWS Profile

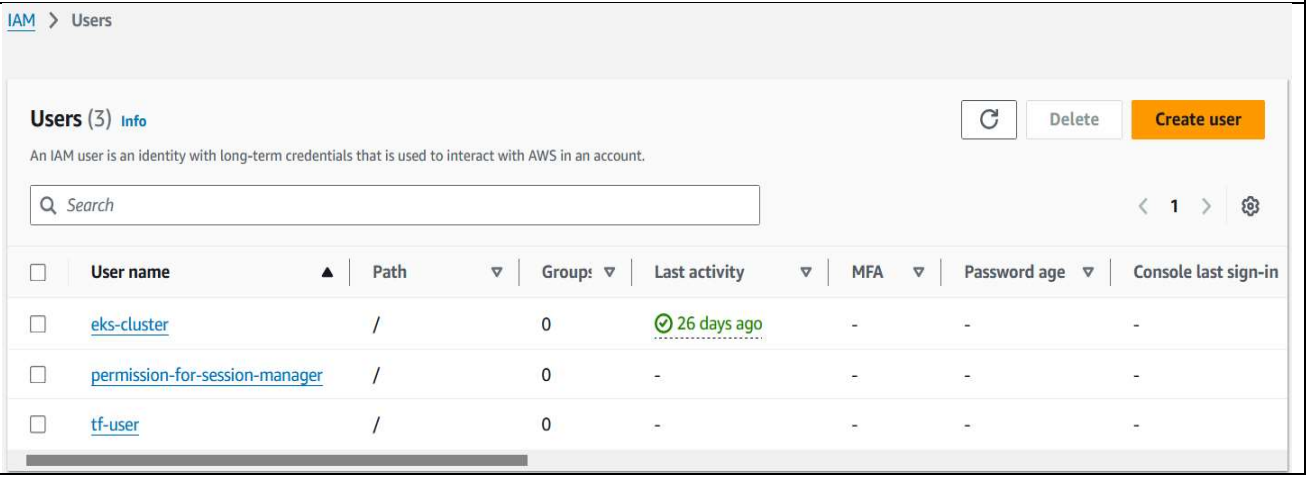
→Session Token Expiration (If using MFA)

In either way, you need to type "aws config"

```
PS D:\Downloads\TERRAFORM> aws configure
AWS Access Key ID [None]:
AWS Secret Access Key [None]:
Default region name [None]:
Default output format [None]:
```

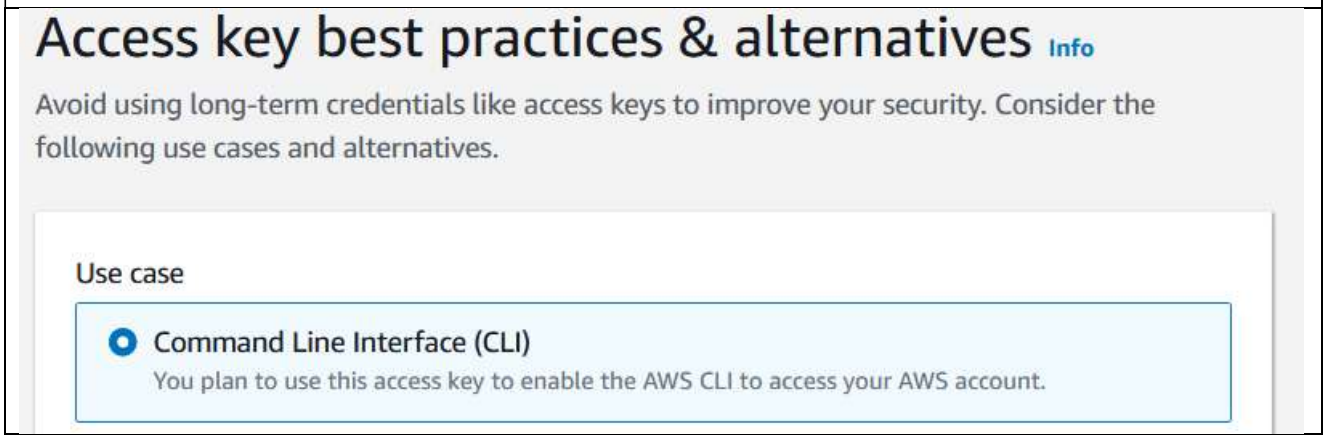
HOW TO GET ALL THE INFORMATION

1.Go to aws cloud and search for IAM service and left-click on tf-user



2.Click on "Create access key"

Tick on



Tick the confirmation-> click on Next→give the tag name →Create he access key

➔Copy access key & Secret access key

Now go to terminal of vs code and type “aws config”

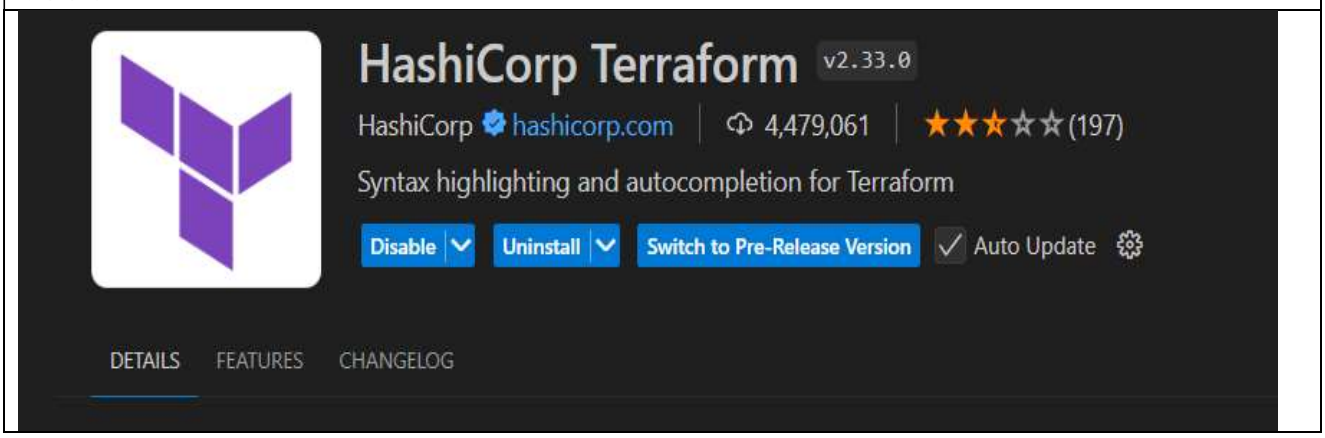
```
PS D:\Downloads\TERRAFORM> aws configure
AWS Access Key ID [None]:
AWS Secret Access Key [None]:
Default region name [None]:
Default output format [None]:
```

Fill up only “Access key & Secret Access key” and rest all skip by doing “enter”

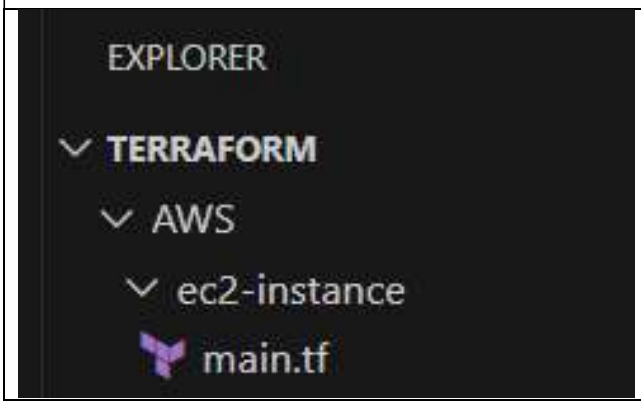
Finally, your AWS cloud and local machine has related to each other with the help of vs code.

12. AWS EC2 WITH TERRAFORM

For Starters you need to install “HashiCorp Terraform” extension in vs code



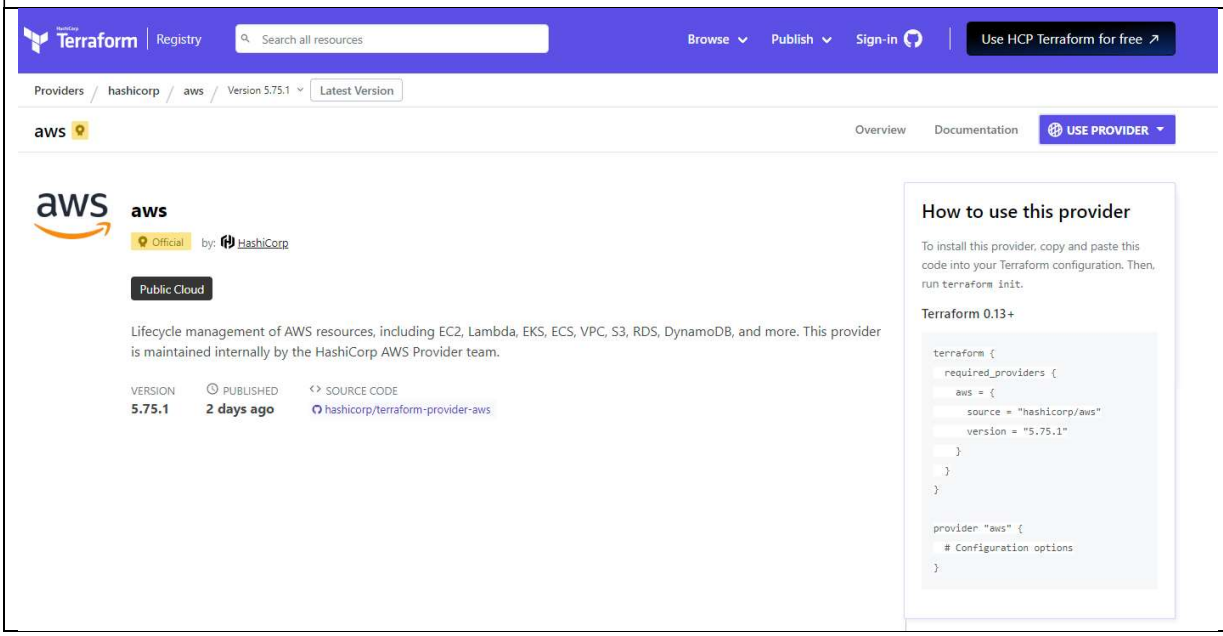
Now create the Folder name “AWS” inside then create ec2-instance folder then create main.tf file



Now decide which provider you will be working on as for me it’s “AWS”

Copy this link
https://registry.terraform.io/providers/hashicorp/aws/latest

You will see



For reference purposes. go to "documentation" that is present before "use provider"

Now copy all the code that is present under "How to use the provider" on "main.tf"

```
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "5.75.1"
    }
  }
}

provider "aws" {
  # Configuration options
}
```

You can change any name with aws like zxd but for readable purpose we use aws, the only thing that is important is

```
source = "hashicorp/aws"
    version = "5.75.1"
```

under "# Configuration options" you will now write the configuration

First, you need to specify the region where you will be working.

```
region= "us-east-1"
```


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

Second, as per the requirement we need to create EC2 instance & EC2 instance is a resource

```
resource "aws_instance" "name" {
}
```

Give the name as per your desire for me myec2 in "name"

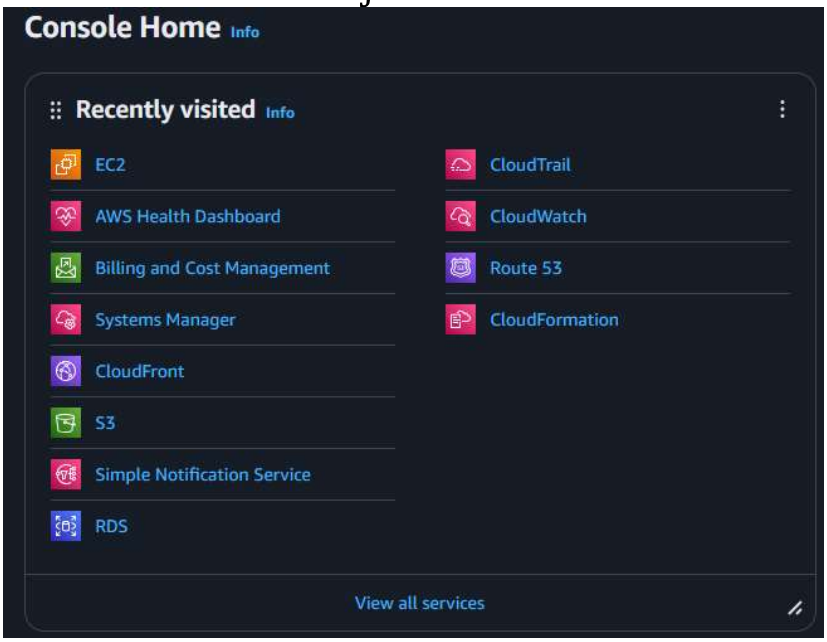
```
resource "aws_instance" "myec2" {
}
```

Now you need to add all the necessary requirements to start the instance like
"ami", "instance_type"

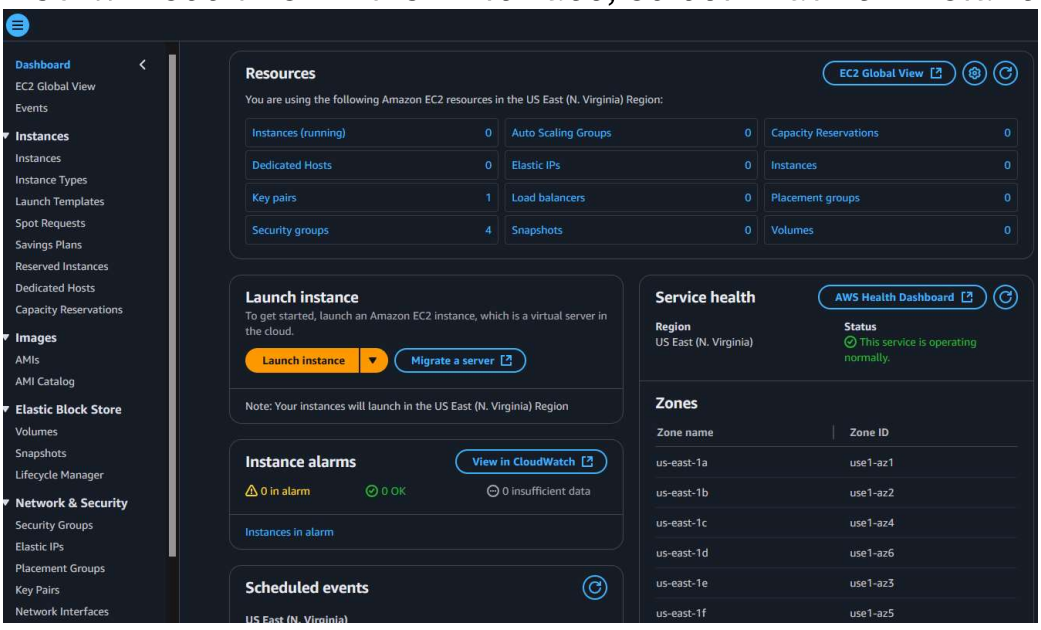
Ami → Amazon Machine Image (AMI) is a pre-configured template that contains the operating system, application server, and applications needed to launch an instance in **Amazon Elastic Compute Cloud (EC2)**.

Steps to get ami id for your desire operating system

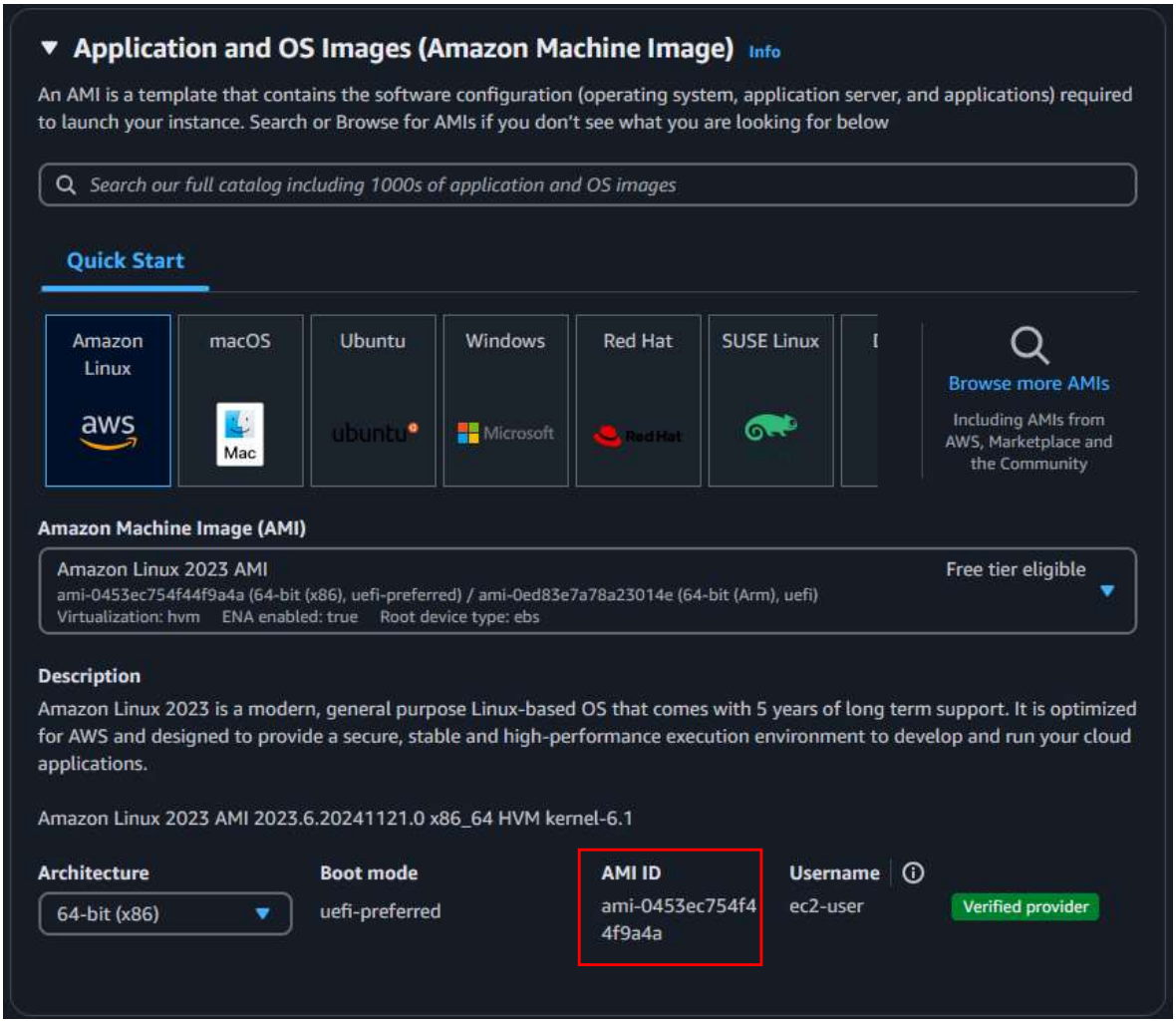
Go to aws dashboard and select EC2 or search got EC2 service on search bar which is just above the “Console Home”



You will see this kind of interface, select “Launch instance”



You will see this



Select the desired operating system copy the ami id that is marked with rectangle to add it in terraform.

Instance Type → The hardware configuration of an Amazon EC2 instance, determining its compute, memory, storage, and networking capacity.

Instance types are categorized into families based on their use cases, such as general-purpose, computer-optimized, memory-optimized, and storage-optimized workloads.

Steps to check which instance did you need for your project

To find that just go below "**Application and OS Images (Amazon Machine Image)**"

You will notice Instance type

Now that you have learned both about AMI ID and Instance Type implement it on terraform in resources

"tag" referred to giving name to ec2 instances

```
resource "aws_instance" "myec2" {
    ami = "ami-0453ec754f44f9a4a"
    instance_type = "t2.micro"

    tags = {
        Name = "Myec2"
    }
}
```

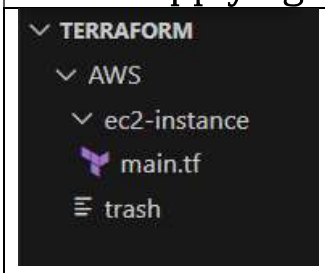
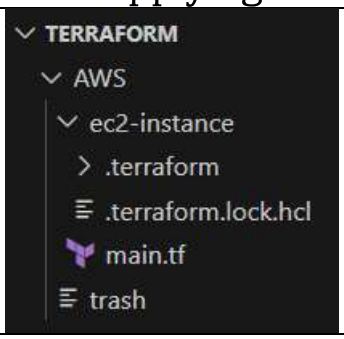
Save the configuration
Now on the terminal of the vs code type the following

IMPORTANT

terraform init → Prepares your working directory for use with Terraform by downloading necessary plugins and setting up the environment.

```
PS D:\Downloads\TERRAFORM\AWS> cd ec2-instance
PS D:\Downloads\TERRAFORM\AWS\ec2-instance> terraform init
```

terraform init → it must be applied where .tf file is present

Before applying terraform init	After applying terraform init
	

terraform plan → Generates and shows an execution plan, detailing the actions Terraform will take to achieve the desired state. Execution plan will show on the terminal

terraform apply → Executes the actions from the plan to create, update, or delete resources in your infrastructure. In this command *Terraform* will ask "Do you want to perform this action" you need to answer the given question either with yes or no only.

if you have done yes, then it will create your based on given instances, to check go to aws console to verify that

terraform destroy → Deletes all the infrastructure resources defined in your Terraform configuration. In this command *Terraform* will ask "Do you want to perform this action" you need to answer the given question either with yes or no only.

terraform destroy –auto-approve → Deletes all the infrastructure resources defined in your Terraform configuration without asking any final permission to delete it.

terraform validate →Checks the syntax and validity of your Terraform configuration files without interacting with remote resources.

13. RESOURCE CHANGE

If you want to make changes to an AWS resource, such as updating the instance type of an EC2 instance (e.g., from t2.micro to t3.micro), you need to modify the configuration file first.

Run ***terraform plan***: This will show you the changes Terraform intends to make, including the update to the EC2 instance type.

Run ***terraform apply***: This will apply the changes and update the EC2 instance to the new instance type.

After running terraform apply, you can **check the AWS EC2 Console** to verify that the instance type has been updated from t2.micro to t3.micro.

Note – if all work is done you should use command ***terraform destroy*** *to delete all the services you have used while doing project.*

14. VARIABLES IN TERRAFORM

In Terraform, ***variables*** let you reuse and customize your configurations by allowing you to pass values into them when running Terraform.

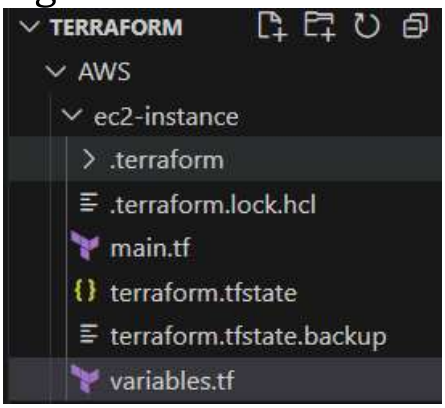
Syntax:

```
variable "variable_name" {
  type        = string   # Type: string, number, bool, list, map, etc.
  default     = "value"  # Optional default value
  description = "Description of the variable"
}
```

"variable_name" can be your any desire name

It's the best practice to create another files name "variables.tf" to store all the variables in one place

e.g.



main.tf	variables.tf

```hcl
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "5.75.1"
    }
  }
}

provider "aws" {
  # Configuration options
  region = var.region
}

resource "aws_instance" "myec2" {
    ami = var.ami
    instance_type = "t2.nano"

    tags = {
        Name = "Myec2"
    }
}
```

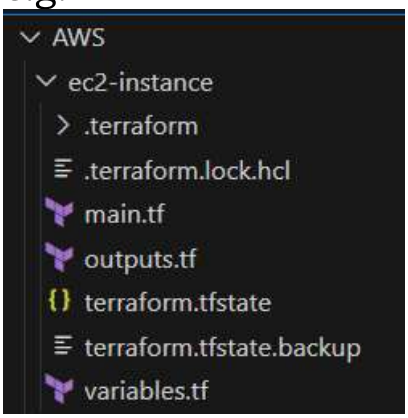
```hcl
variable "region" {
  type = string
  default = "us-east-1"
  description = "the region"
}

variable "ami" {
  type = string
  default = "ami-0453ec754f44f9a4a"
  description = "the ami"

}
```

```
PS D:\Downloads\TERRAFORM\AWS\ec2-instance> terraform validate
Success! The configuration is valid.

PS D:\Downloads\TERRAFORM\AWS\ec2-instance>
```

15. OUTPUT IN TERRAFORM

In Terraform, **output variables** are used to display information about your resources or configuration after running terraform apply. They help you extract and use values, such as resource IDs or IP addresses, for other tasks.

It's the best practice to create another files name "outputs.tf" to store all the outputs in one place
e.g.



Outputs.tf

```
output "instance_public_ip" {
  value = aws_instance.myec2.public_dns
  description = "public DNS of the EC2 instance"
}
```

```
PS D:\Downloads\TERRAFORM\AWS\ec2-instance> terraform validate
Success! The configuration is valid.

PS D:\Downloads\TERRAFORM\AWS\ec2-instance>
```

You will see something like this

```
Outputs:

instance_public_ip = "ec2-18-212-97-149.compute-1.amazonaws.com"
PS D:\Downloads\TERRAFORM\AWS\ec2-instance>
```

16. IMPLEMENT S3 BUCKET WITH THE HELP OF TERRAFORM

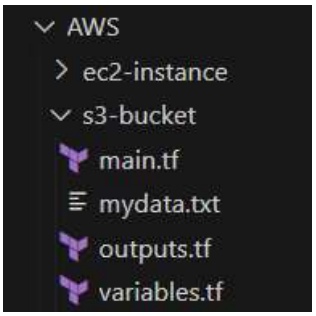
S3 Bucket is a logical container used to store and organize objects (files) in **Amazon Simple Storage Service (Amazon S3)**. It acts as a root directory where data is stored securely and can be accessed, managed, and retrieved.

Key features:

- Stores virtually unlimited amounts of data.
- Stores data as **objects** that consist of key, value, metadata.
- Must have unique bucket's name across all AWS accounts globally.
- Store and retrieve backup data
- Region-Specific to reduce latency
- Versioning, object locking, and MFA delete enhance security.

First create one folder inside AWS name "s3-bucket" inside it create blank files names

- main.tf
- variables.tf
- outputs.tf
- mydata.txt →to upload data in s3 bucket



variables.tf

```
variable "region" {
  type = string
  default = "us-east-1"
  description = "the region"
}
```

main.tf

```
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "5.75.1"
    }
  }
}

provider "aws" {
  # Configuration options
  region = var.region
}

resource "aws_s3_bucket" "demo-bucket" {
  bucket = "trial-bucket-123"
}

resource "aws_s3_object" "bucket-data" {
  bucket = aws_s3_bucket.demo-bucket
  source = "./mydata.txt"
  key = "introduction-of-project.txt"
}
```

outputs.tf

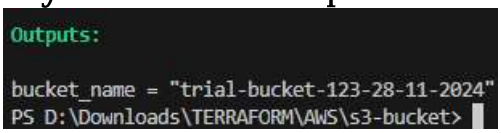
```
output "bucket_name" {
  value = aws_s3_bucket.demo-bucket.id
}
```

Now type "***terraform init***" cmd in vscode terminal to initial terraform in s3-bucket folder

```
PS D:\Downloads\TERRAFORM\AWS\s3-bucket> terraform init
Initializing the backend...
Initializing provider plugins...
- Finding hashicorp/aws versions matching "5.75.1"...
- Installing hashicorp/aws v5.75.1...
```

Type “***terraform plan***” cmd in vscode terminal to check what all configurations terraform will do.
Type “***terraform plan***” cmd in vscode terminal

If you see this output



Then you good to go
To verify whether its created or not, go to S3 service in AWS console

Note – if all work is done you should use command ***terraform destroy*** *to delete all the services you have used while doing project.*

17. RANDOM PROVIDER

The **Random provider** in Terraform is used to generate random values, such as strings, numbers, or pet names, which can be used in configurations. This is helpful when you need unique identifiers for resources.

In this demo we will use s3-bucket folder to learn “Random provider”

For reference
https://registry.terraform.io/providers/hashicorp/random/latest/docs

The provider syntax is

```
terraform {
  required_providers {
    random = {
      source = "hashicorp/random"
      version = "3.6.3"
    }
  }
}
```

Changes in main.tf

Without the use of random provider	With the use of random provider

```
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "5.75.1"
    }
  }
}

provider "aws" {
  # Configuration options
  region = var.region
}

resource "aws_s3_bucket" "demo-bucket" {
  bucket = "trial-bucket-123"
}

resource "aws_s3_object" "bucket-data" {
  bucket = aws_s3_bucket.demo-bucket
  source = "./mydata.txt"
  key = "introduction-of-project.txt"
}
```

```
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "5.75.1"
    }
    random = {
      source = "hashicorp/random"
      version = "3.6.3"
    }
  }
}

provider "aws" {
  # Configuration options
  region = var.region
}

resource "random_id" "rand_id" {
  byte_length = 10
}

resource "aws_s3_bucket" "demo-bucket" {
  bucket = "trial-bucket-${random_id.rand_id.dec}"
}

resource "aws_s3_object" "bucket-data" {
  bucket = aws_s3_bucket.demo-bucket.bucket
  source = "./mydata.txt"
  key = "introduction-of-project.txt"
}
```

Here you will notice resource with ‘rand_id” name

```
resource "random_id" "rand_id" {
  byte_length = 10
}
```

In place of “random_id” it can be

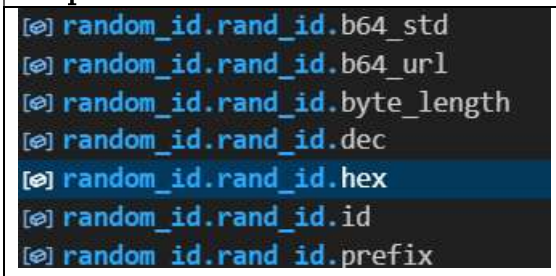
```
random_bytes
random_id
random_integer
random_password
random_pet
random_shuffle
random_string
random_uuid
```

“byte_length” can be any number depending of your choice

```
resource "aws_s3_bucket" "demo-bucket" {
  bucket = "trial-bucket-${random_id.rand_id.dec}"
}
```

Here “${…}” sign is used because we are referring from somewhere

In place of “dec” we can use the following



There will be no change in “variables.tf” and “outputs.tf” configurations

terraform init → terraform plan → terraform apply

```
bucket_name = "trial-bucket-23858315711029169259930O"
PS D:\Downloads\TERRAFORM\AWS\s3-bucket>
```

Then you good to go
To verify whether its created or not, go to S3 service in AWS console

*Note – if all work is done you should use command **terraform destroy** to delete all the services you have used while doing project.*

18. TERRAFORM REMOTE STATE MANAGEMENT

As we all know terraform maintain current state of the infrastructure in the file “terraform.tfstate”

Key Points:

→ **Purpose**: It helps Terraform determine what changes need to be applied by comparing the actual state of resources with the desired state in your configuration.

→ **Format**: The file is in JSON format and includes resource metadata, attributes, and dependencies.

→ **Location**: By default, it is saved in the working directory where Terraform is run.

Remote State: To collaborate securely, the state file can be stored remotely (e.g., in S3 or a Terraform Cloud workspace).

In this we will store “terraform.tfstate” in created S3 bucket for collaboration, security, and consistency in infrastructure management when working with Terraform.

Syntax

```
terraform {
  backend "s3" {
    bucket         = "my-terraform-state"
    key            = "state/terraform.tfstate"
    region         = "us-east-1"
    dynamodb_table = "terraform-lock-table"  # For locking
    encrypt        = true                    # Encrypt state file
  }
}
```

In this we will create a new folder name “terraform-backend” where we will create “main.tf”

Main reason to use S3 bucket is that if changes are made in the "terraform.tfstate" it automatically makes changes in S3 bucket

```
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "5.75.1"
    }
  }
  backend "s3" {
    bucket         = "trial-bucket-1129312763638267012337224"
    key            = "terraform.tfstate"
    region         = "us-east-2"
  }
}

provider "aws" {
  # Configuration options
  region = "us-east-1"
}

resource "aws_instance" "myec2" {
    ami = "ami-0453ec754f44f9a4a"
    instance_type = "t2.nano"

    tags = {
        Name = "Myec2"
    }
}
```

*Note – if all work is done you should use command **terraform destroy** to delete all the services you have used while doing project.*

19. PROJECT: DEPLOY STATIC WEBSITE ON AWS USING S3

To make this project successful we need a simple project with

- index.html
- style.css
- script.js

and terraform files like

- provider.tf
- main.tf
- variables.tf
- outputs.tf

For reference:

- aws_s3_bucket_public_access_block --Terraform
- Setting permissions for website access --AWS
- aws_s3_bucket_policy --Terraform
- aws_s3_bucket_versioning --Terraform
- aws_s3_bucket_website_configuration --Terraform
-

To host the static website in S3 bucket you must
"untick" Block all public access

```
resource "aws_s3_bucket_public_access_block" "example" {
  bucket = aws_s3_bucket.S3_bucket.id

  block_public_acls       = false
  block_public_policy     = false
  ignore_public_acls      = false
  restrict_public_buckets = false
}
```

To “add a bucket policy” →To make the objects in your bucket publicly readable, you must write a bucket policy that grants everyone s3:GetObject permission.

```
resource "aws_s3_bucket_policy"
"allow_access_from_another_account" {
  bucket = aws_s3_bucket.S3_bucket.id
  policy = jsonencode(
    {
       Version = "2012-10-17",
       Statement = [
          {
             Sid = "PublicReadGetObject",
             Effect = "Allow",
             Principal = "*",
             Action = "s3:GetObject",
             Resource = "arn:aws:s3:::${aws_s3_bucket.S3_bucket.id}/*"
          }
       ]
    }
  )
}
```

The ***jsonencode*** function in Terraform is used to convert a data structure written in HCL (HashiCorp Configuration Language) into a JSON-formatted string.

“Unable” Bucket Versioning

```
resource "aws_s3_bucket_versioning" "versioning" {
  bucket = aws_s3_bucket.S3_bucket.id
  versioning_configuration {
    status = "Enabled"
  }
}
```

The **aws_s3_bucket_website_configuration** resource in Terraform is used to configure static website hosting for an S3 bucket. This resource specifies the settings like the index document, error document, and routing rules for the website.

```
rresource "aws_s3_bucket_website_configuration" "site_hosting" {
  bucket = aws_s3_bucket.S3_bucket.id

  index_document {
    suffix = "index.html"
  }
}
```

*Note – if all work is done you should use command **terraform destroy** to delete all the services you have used while doing project.*

20. UNDERSTANDING VPC FOR TERRAFORM IMPLEMENTATION

A **Virtual Private Cloud (VPC)** is a logically isolated network in AWS, where you can launch resources securely.

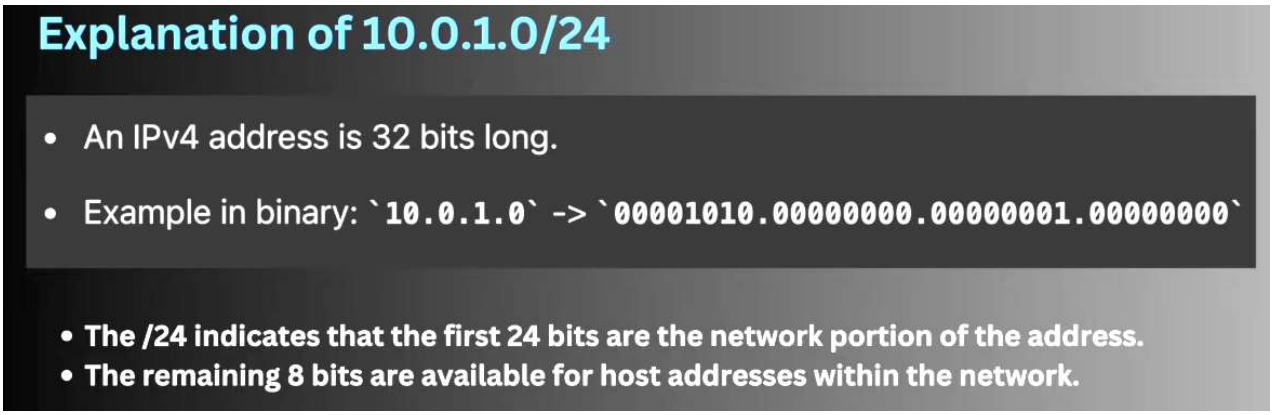
VPC CIDR BLOCK

When you create a VPC, you specify a CIDR block that defines the IP address range for the entire VPC.

CIDR: (Classless Inter-Domain Routing) is a method for allocating address and routing Internet Protocol (IP) packets.

CIDR Block Allocation:
You specify a range of Ip address (CIDR block) within the VPC's IP address range for the subnet.

This determines the pool of IP addresses available for instances in the subnet .



For e.g. 10.0.0.0/16
This block or IP allows for 65,536 IO address (but 65,531 usable address)

KEYWORDS

NAT gateway, Internet Gateway, subnets (private or public), Load Balancer, NACL, Security group, Route table, VPC Peering

NAT – Network Address Translation
NACL- Network Access Control List
ICMP- Internet Control Message Protocol

INTERNET GATEWAY

A gate that allows you to connect VPC to the Internet, only applicable to the instance data present in public subnet.

LOAD BALANCER

Forward the request depending upon the load. Basically, it is connected to the public subnet.

OR

Distributes incoming traffic across multiple targets (e.g., EC2 instances) to improve availability and reliability.

ROUTER TABLE

A path that connects the load balancer of the public subnet to the application or instance of the private subnet.
A set of rules that determine where network traffic is directed.
Each subnet in a VPC has its own route table that controls traffic flow between subnets.

SECURITY GROUPS

The first layer of security of any EC2 instances i.e. attached to the instances, which tell the instance with Ip address or port will be used to give access to the instance.

Types of Security group

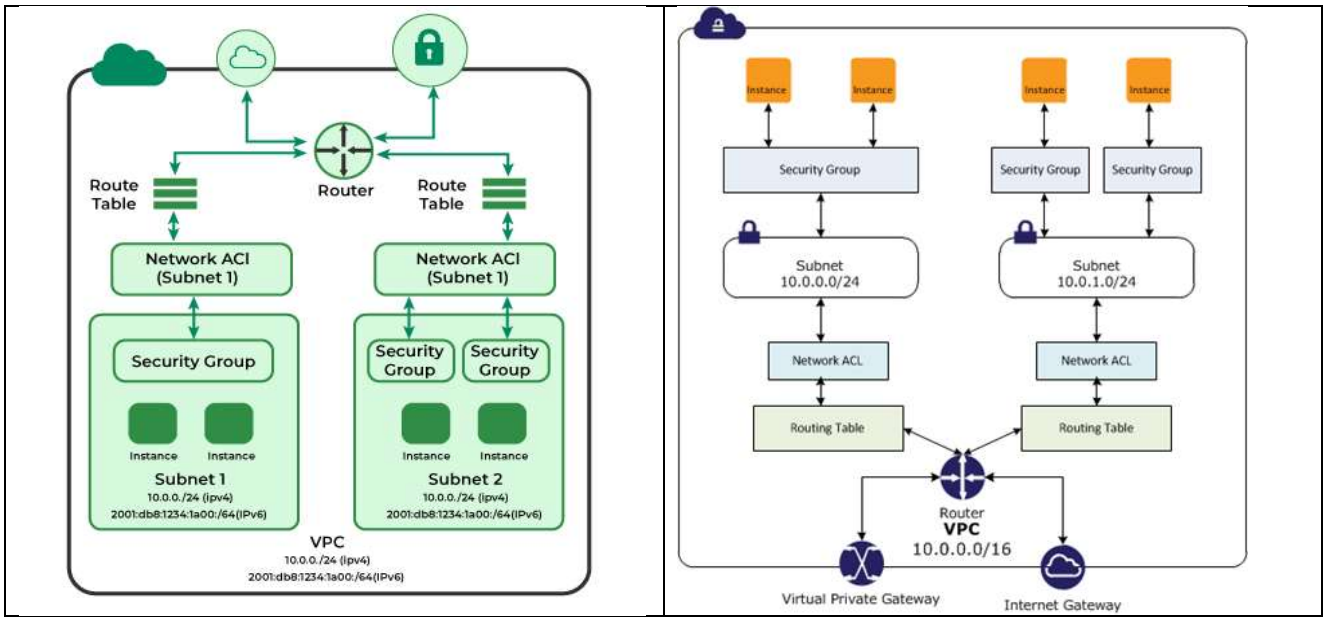
Inbound traffic	Internet to instances
Outbound traffic	Instance to internet

NACL- NETWORK ADDRESS TRANSLATION

The second layer of security that comes after security groups that are attached to the subnets
Difference between

Security group	NACL

Serves at the instance level	Serves at subnet level
Permit rules only	Permit and deny rules
All rules examined first	Rules processed until matched



SUBNETS

The VPC is created with Ip address range, splitting the Ip address for sub projects.

Inside the subnet the ec2 instances are created.

There are two types of subnets i.e.

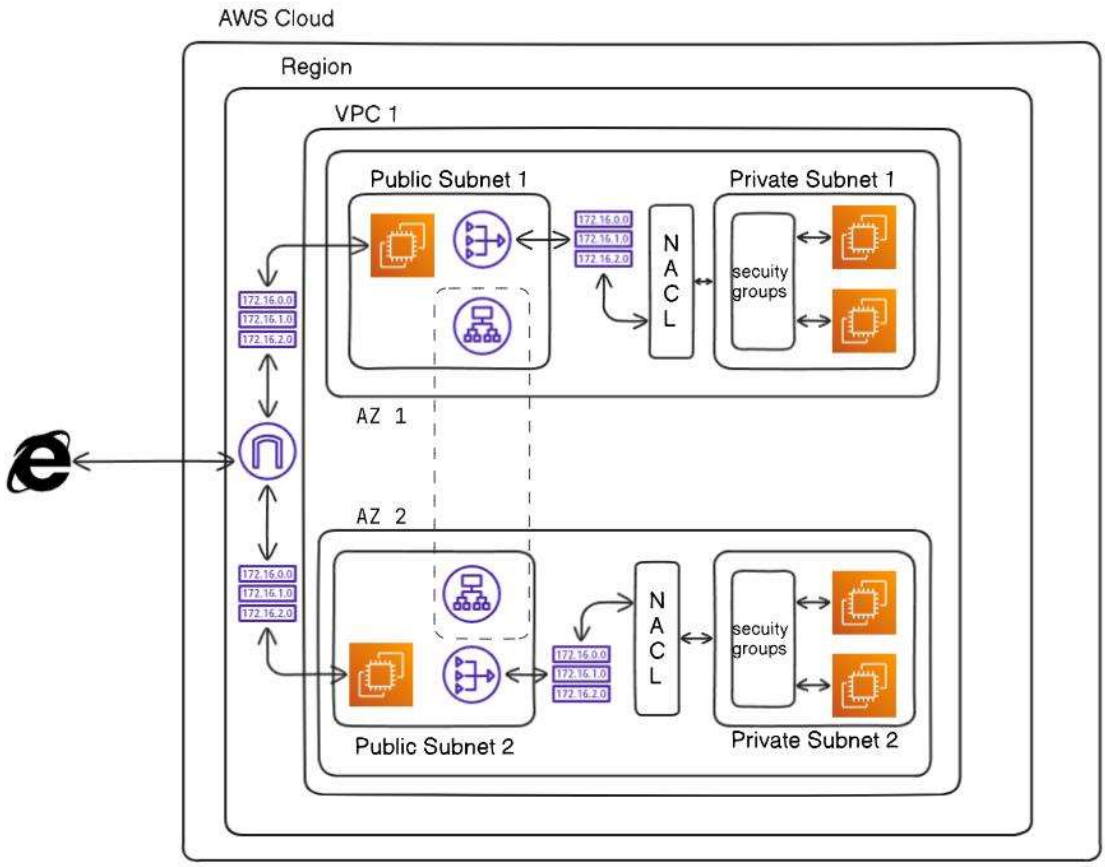
Private subnet →Instances cannot be connected to internet-by-internet gateway, instead they user NAT gateway.

Public subnet →Instances that is present inside this subnet is accessible to internet through internet gateway by route table.

NAT GATEWAY

- Used for private subnet.
- Helps to mask Ip address.
- Helps to download some resources from internet while doing that it will mask or change the Ip address with the public Ip address

either from the load balancer (SNAT) or from the router (NAT gateway)



AWS PEERING

-Stable connection between 2 VPC either on same account or two different accounts.

-Check if the peering is working or not use ping (ICMP)

Steps for Route Table Updates:

Create a VPC Peering Connection (if not already done) between the two VPCs, as explained earlier.

Update Public Subnet Route Tables: Go to the **Route Tables** section in the AWS VPC console.

- For each **public subnet** in VPC 1:
- Add a route:

- **Destination**: The CIDR block of VPC 2 (e.g., *10.2.0.0/16*).
- **Target**: The VPC Peering Connection ID.

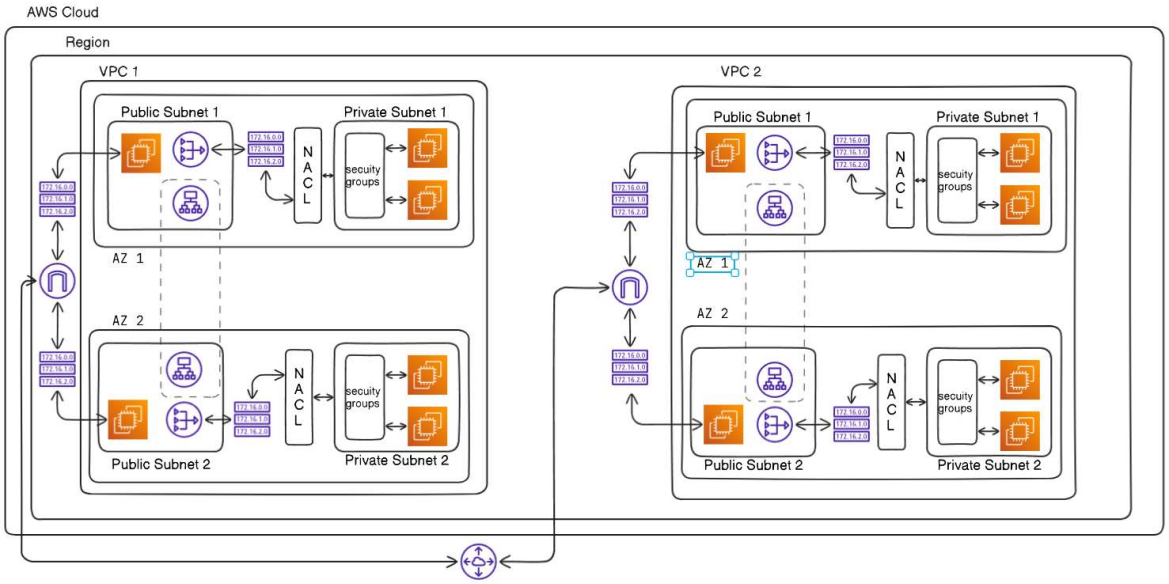
- Repeat the same for the public subnets in VPC 2, adding routes to the CIDR block of VPC 1.

Security Group Updates:

- Modify the security groups of instances in both VPCs to allow traffic from the other VPC's CIDR block (e.g., *10.1.0.0/16* and *10.2.0.0/16*).
- Add rules for required protocols (e.g., *SSH, HTTP, or custom*).

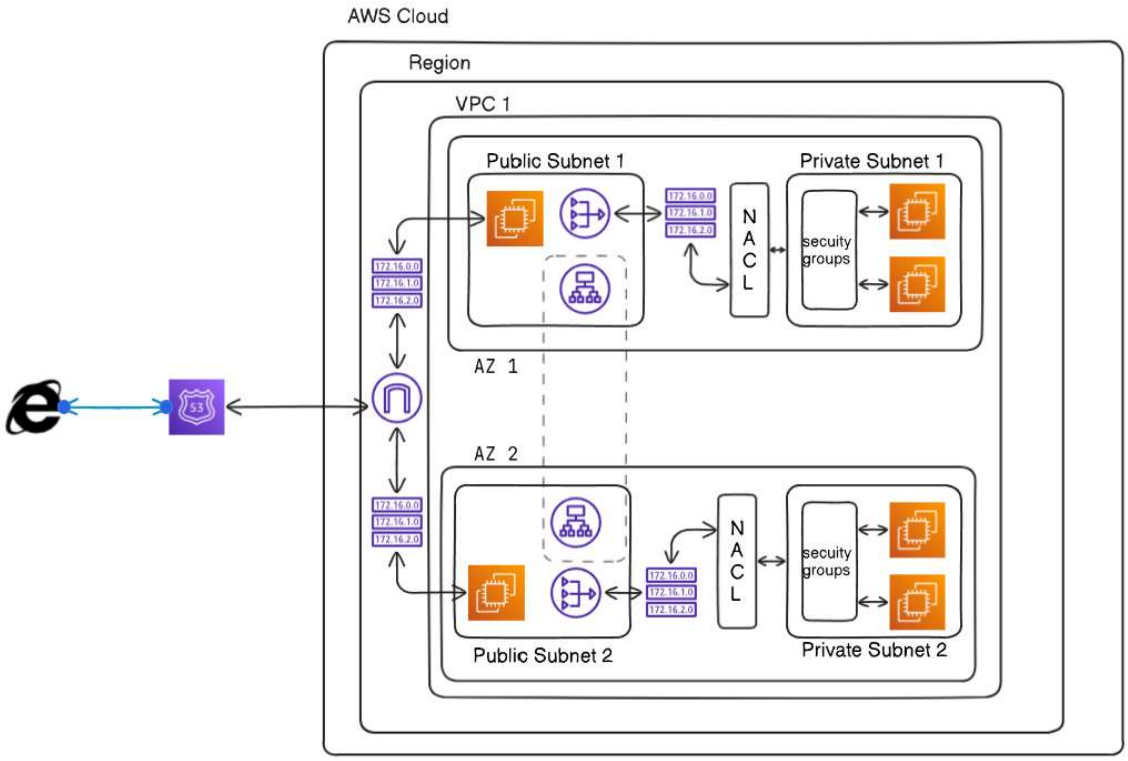
Test Connectivity:

- Launch instances in the respective subnets (public/private) of both VPCs and verify connectivity using tools like ping or curl.



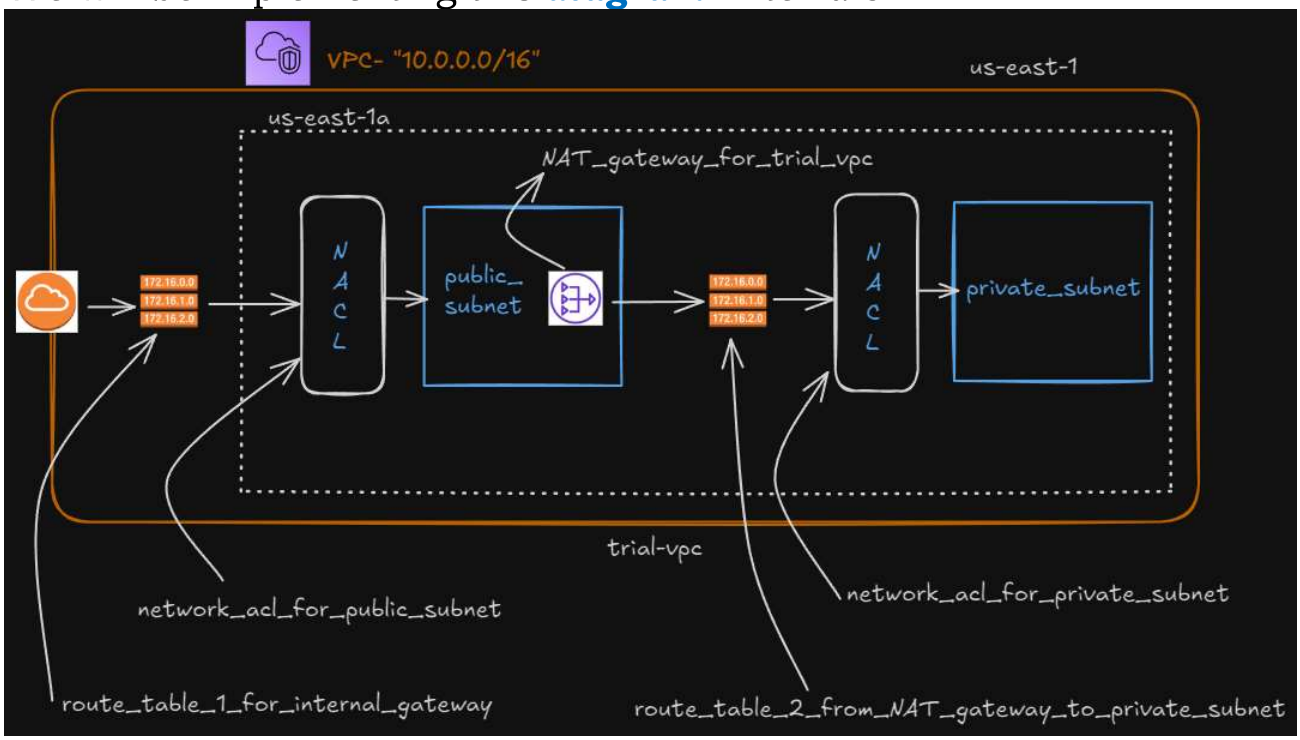
Route53

- `Provides DNS as a service.`
- `DNS – Domain Name Services`
- `Performs health check on web server.`
- `Domain Registration → hosted zones`


AWS Cloud
Region
VPC 1
Public Subnet 1
Private Subnet 1
172.16.0.0
172.16.1.0
172.16.2.0
N
A
C
L
secuity
groups
AZ 1
AZ 2
Public Subnet 2
Private Subnet 2

21. IMPLEMENTING VPC USING TERRAFORM

We will be implementing this ***diagram*** in terraform



We will be using services like

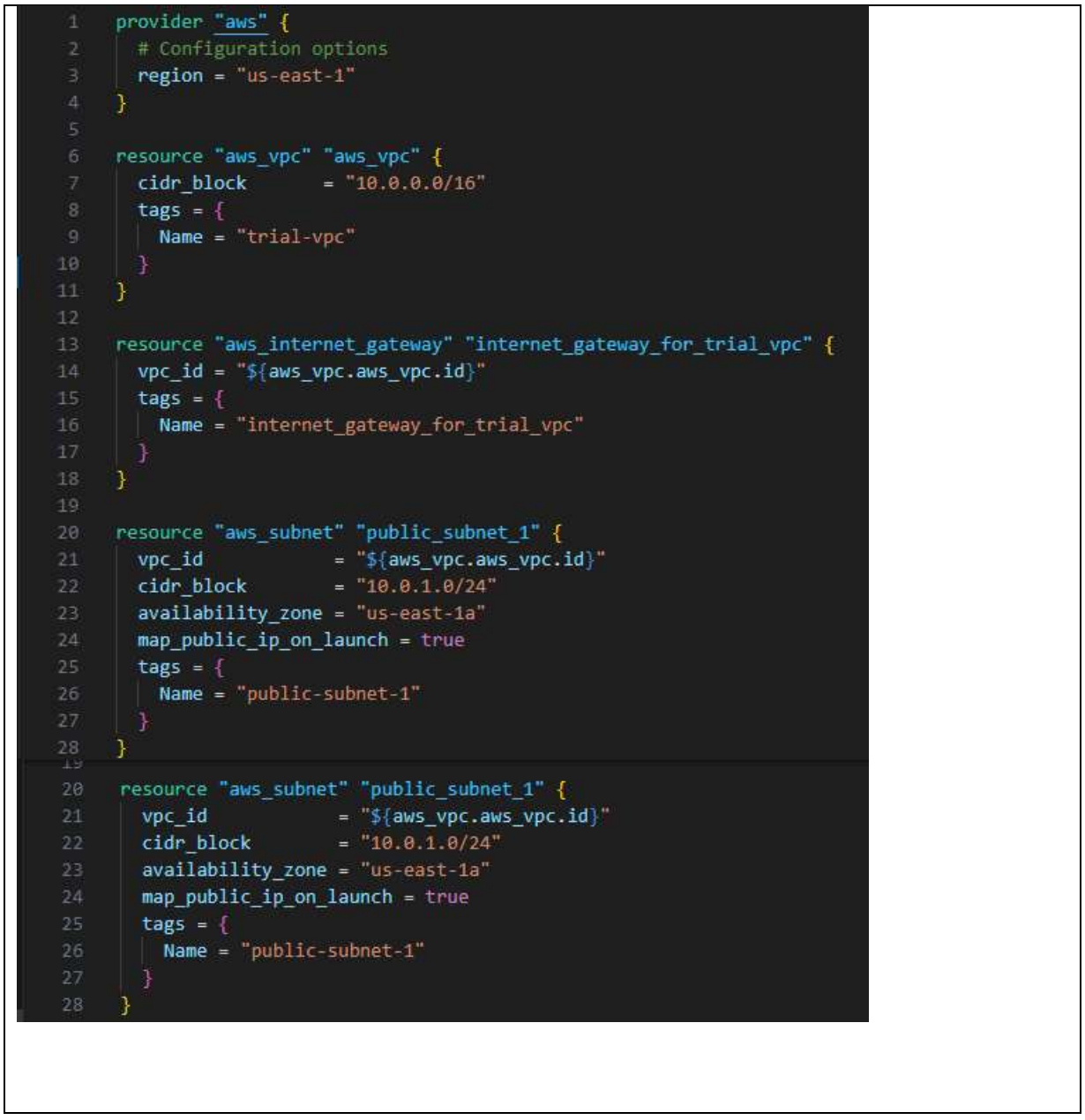
- VPC
- Availability zone
- Subnets
- Internet gateway
- Network ACL
- Network ACL association
- Route table
- Route table association
- NAT gateway
- Elastic IP NAT gateway

For this implementation we will create new folder name “vpc” that contains: -

- main.tf
- outputs.tf
- provider.tf

provider.tf

main.tf



```
resource "aws_subnet" "private_subnet_2" {
  vpc_id            = "${aws_vpc.aws_vpc.id}"
  cidr_block        = "10.0.2.0/24"
  availability_zone = "us-east-1a"
  map_public_ip_on_launch = false
  tags = {
    Name = "private-subnet-2"
  }
}

# Create a Default Network ACL for private subnet
resource "aws_network_acl" "network_acl_for_private_subnet" {
  vpc_id = aws_vpc.aws_vpc.id

  ingress {
    rule_no    = 100
    protocol   = "-1"
    action = "allow"
    cidr_block = "0.0.0.0/0"
    from_port  = 0
    to_port    = 0
  }

  egress {
    rule_no    = 100
    protocol   = "-1"
    action = "allow"
    cidr_block = "0.0.0.0/0"
    from_port  = 0
    to_port    = 0
  }

  tags = {
    Name = "network-acl-for-subnet-2"
  }
}

# Associate Network ACL with  pricate Subnet
resource "aws_network_acl_association" "acl_association_for_private_subnet" {
  subnet_id       = aws_subnet.private_subnet_2.id
  network_acl_id   = aws_network_acl.network_acl_for_private_subnet.id
}

 # Create a Default Network ACL for public subnet
resource "aws_network_acl" "network_acl_for_public_subnet" {
  vpc_id = aws_vpc.aws_vpc.id

  # Allow all inbound traffic
  ingress {
    rule_no    = 100
    protocol   = "-1"
    action = "allow"
    cidr_block = "0.0.0.0/0"
    from_port  = 0
    to_port    = 0
  }
```

```
86
87    # Allow all outbound traffic
88    egress {
89      rule_no    = 100
90      protocol   = "-1"
91      action = "allow"
92      cidr_block = "0.0.0.0/0"
93      from_port  = 0
94      to_port    = 0
95    }
96
97    tags = {
98      Name = "network-acl-for-subnet-1"
99    }
100 }
101
102 # Associate Network ACL with  public Subnet
103 resource "aws_network_acl_association" "acl_association_for_public_subnet" {
104   subnet_id        = aws_subnet.public_subnet_1.id
105   network_acl_id   = aws_network_acl.network_acl_for_public_subnet.id
106 }
107
108 resource "aws_route_table" "route_table_1_for_internal_gateway" {
109   vpc_id = "${aws_vpc.aws_vpc.id}"
110   route {
111     cidr_block = "0.0.0.0/0"
112     gateway_id = aws_internet_gateway.internet_gateway_for_trial_vpc.id
113   }
114
115   tags = {
116     Name = "route_table_1_for_internet_gateway"
117   }
118 }
119
120 resource "aws_route_table_association" "aws_route_table_association_1" {
121   subnet_id      = aws_subnet.public_subnet_1.id
122   route_table_id = aws_route_table.route_table_1_for_internal_gateway.id
123 }
124
125 #elastic ip in vpc
126 resource "aws_eip" "Elastic_IP_for_NAT_Gateway" {
127   domain = "vpc"
128
129   tags = {
130     Name = "Elastic_IP_for_NAT_Gateway"
131   }
132 }
133
134
135 resource "aws_nat_gateway" "NAT_gateway_for_trial_vpc" {
136   allocation_id = aws_eip.Elastic_IP_for_NAT_Gateway.id
137   subnet_id     = aws_subnet.public_subnet_1.id
138
139   tags = {
140     Name = "NAT_gateway_for_trial_vpc"
141   }
142 }
143
```

```
143
144  resource "aws_route_table" "route_table_2_from_NAT_gateway_to_private_subnet" {
145    vpc_id = "${aws_vpc.aws_vpc.id}"
146
147    route {
148      cidr_block = "10.0.1.0/24"
149      nat_gateway_id = aws_nat_gateway.NAT_gateway_for_trial_vpc.id
150    }
151
152    tags = {
153      Name = "route_table_2_from_NAT_gateway_to_private_subnet"
154    }
155  }
156
157  resource "aws_route_table_association" "aws_route_table_association_2" {
158    subnet_id      = aws_subnet.private_subnet_2.id
159    route_table_id = aws_route_table.route_table_2_from_NAT_gateway_to_private_subnet.id
160  }
161
```

Outputs.tf

```
AWS > vpc > outputs.tf > ...
1   output "aws_vpc_id" {
2     value = aws_vpc.aws_vpc.id
3   }
4
5   output "aws_internet_gateway_id" {
6     value = aws_internet_gateway.internet_gateway_for_trial_vpc.id
7   }
8
9   output "aws_nat_gateway_id" {
10    value = aws_nat_gateway.NAT_gateway_for_trial_vpc.id
11  }
```

Output

*Note – if all work is done you should use command **terraform destroy** to delete all the services you have used while doing project.*

22. DATA SOURCE

It allows to fetch and use information from

- external sources or
- existing resources within your cloud infrastructure
- They are read-only and useful for referencing or integrating with existing infrastructure.

Useful for obtaining dynamic data that you need for your configurations.

Real-Life Scenario:
Your company already has a production VPC and subnets set up, and you want to deploy a new application into an existing subnet without modifying the existing infrastructure. Instead of hardcoding VPC and subnet IDs, you use Terraform data sources to dynamically fetch this information.

NOTE: we are only doing "terraform plan" not "terraform apply" because it may cost you, do it at your own risk.

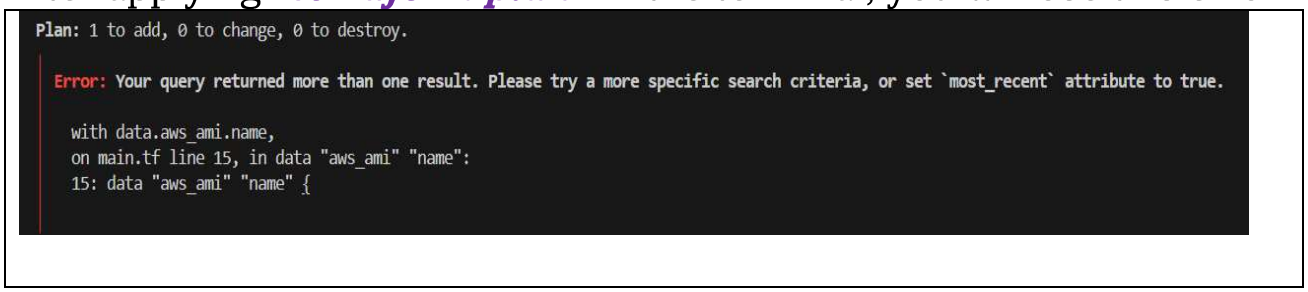
Create a new folder in "AWS" i.e. "tf-data-resource" inside create another file name "main.tf"

AWS > tf-data-resources > main.tf > output "aws_ami" > value

```
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "5.75.1"
    }
  }
}

provider "aws" {
  # Configuration options
  region = "us-east-1"
}

data "aws_ami" "name" {

}

```

```
output "aws_ami" {
  value = data.aws_ami.name
}

resource "aws_instance" "myec2" {
    ami = "ami-0453ec754f44f9a4a"
    instance_type = "t2.nano"

    tags = {
        Name = "Myec2"
    }
}
```

After applying "***terraform plan***" in the terminal, you will see this error



It explains:

- The AMI query is too broad, leading to multiple AMIs matching the criteria.
- Terraform doesn't know which AMI to select because no specific filtering or sorting is provided.

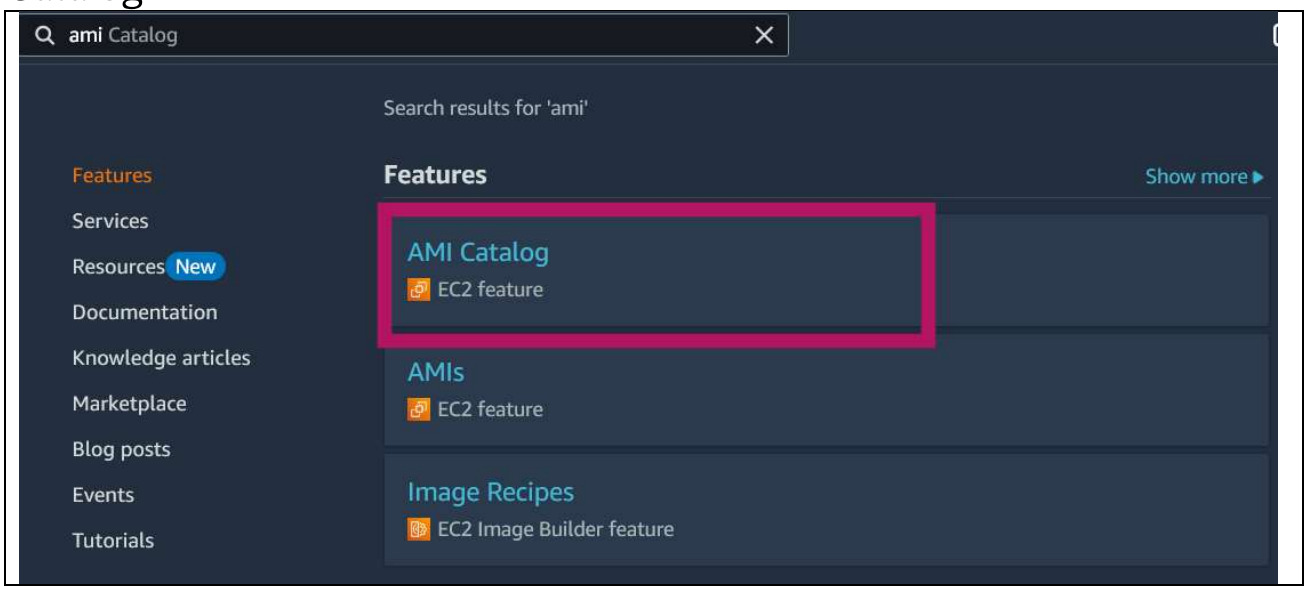
To fix the error we just need to make changes in

```
data "aws_ami" "name" {
    most_recent = true
    owners = [ "amazon" ]

}

output "aws_ami" {
  value = data.aws_ami.name.id
}
```

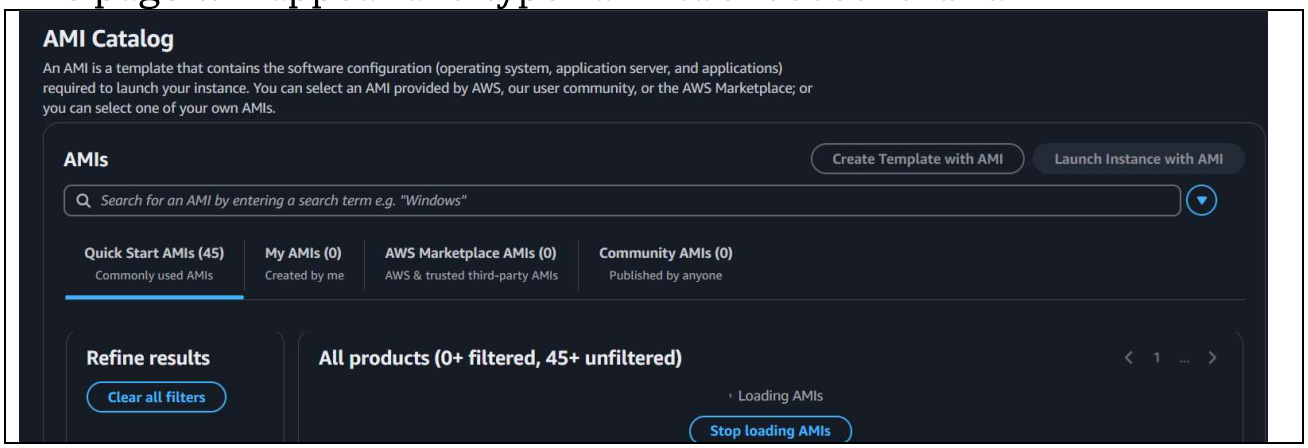
After appling "terraform plan" in the terminal,

```
Changes to Outputs:
  + aws_ami = "ami-0b5268083787b7af7"
```

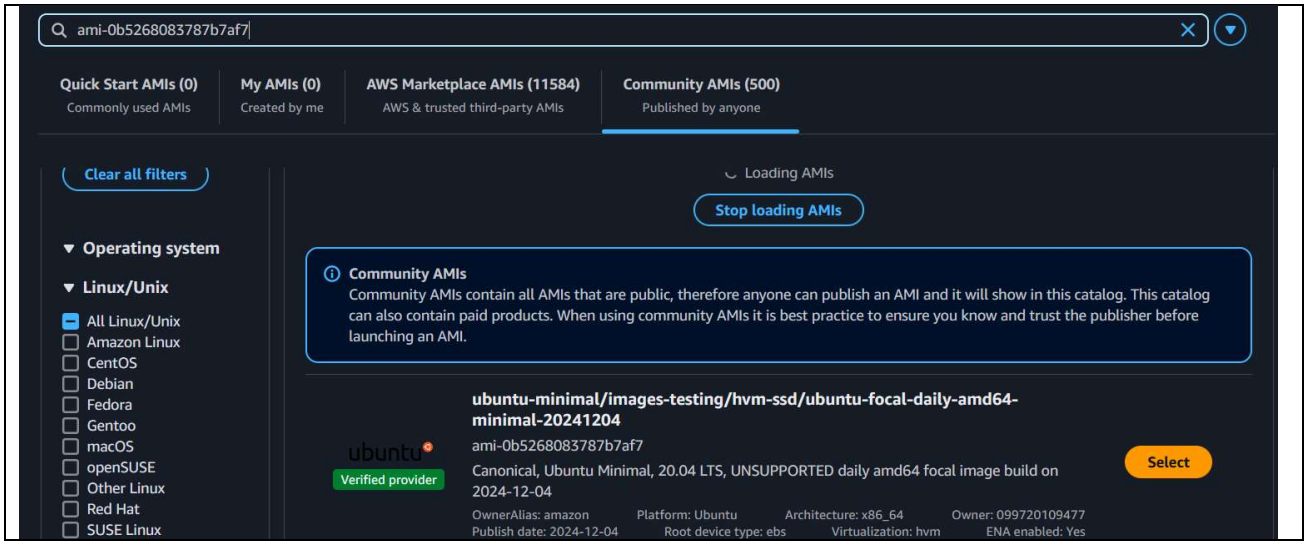
If you want to check if it's correct or not we will verify it in AWS console then go to AWS console → Type AMI in search bar → click on "AMI Catalog"



This page will appear and type "ami-0b5268083787b7af7"



You will see



To see the availability zones of particular zone

```
#AZ
data "aws_availability_zones" "names" {
  state = "available"
}
output "aws_zones" {
  value = data.aws_availability_zones.names
}
```

To get the account details

```
#To get the account details
data "aws_caller_identity" "name" {
}
output "caller_info" {
  value = data.aws_caller_identity.name
}
```

23. CREATE EC2 USING EXISTING VPC

To create an EC Instance with an existing

- VPC
- Private subnet
- Security-group

main.tf

STEP-1

```
AWS > tf-data-resources > main.tf > ...
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "5.75.1"
    }
  }
}

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

STEP-2

```
#ami-id
data "aws_ami" "name" {
    most_recent = true
    owners = [ "amazon" ]
}

#ami-id output
output "aws_ami" {
  value = data.aws_ami.name.id
}

# vpc-id
data "aws_vpc" "name" {
  tags = {
    Name = "trial-vpc"
  }
}

#vpc-id output
output "aws_vpc" {
  value = data.aws_vpc.name.id

}
```

STEP-3

```
#subnet-id
data "aws_subnet" "example" {
  filter {
    name   = "vpc-id"
    values = [data.aws_vpc.name.id]
  }
  tags = {
    Name= "public-subnet-1"
  }
}

#subnet-id output
output "aws_subnet" {
  value = data.aws_subnet.example.id
}

#security-group
data "aws_security_group" "name" {
  tags = {
    Name = "Security_Group_1"
  }
}
```

STEP-4

```
#security-group output
output "aws_security_group" {
  value = data.aws_security_group.name.id
}

#instance-creation
resource "aws_instance" "myec2" {
    ami = "ami-0453ec754f44f9a4a"
    instance_type = "t2.nano"
    subnet_id = data.aws_subnet.example.id
    security_groups = [data.aws_security_group.name.id]
    tags = {
        Name = "Myec2"
    }
}
```

Output

```
Outputs:

aws_ami = "ami-0d5d11b36133682d6"
aws_security_group = "sg-0488910c98264ee39"
aws_subnet = "subnet-071f71057ee2b670a"
aws_vpc = "vpc-0823d33924950fc64"
```

24. TERRAFORM VARIABLES

In this we will create a new folder name “tf-variables” with files “main.tf” and “variables.tf”

main.tf

```
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "5.75.1"
    }
  }
}

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

resource "aws_instance" "ec2_instance" {
  ami = "ami-0453ec754f44f9a4a"
  instance_type = "t2.micro"

  root_block_device {
    volume_size = 8
    volume_type = "gp2"
    delete_on_termination = true
  }

  tags = {
    Name = "ec2-instance"
  }
}
```

Real-World Scenario: Multi-Environment Deployment

Suppose you are tasked with deploying EC2 instances in multiple environments like development, staging, and production. Each environment requires:

Different instance types (t2.micro for dev, t2.medium for staging, m5.large for production).

Unique tags to identify the environment ("Name = dev-instance", "Name = staging-instance", etc.).

Different AMI IDs based on the environment.

If you hardcode these values in your main.tf file for each environment, you will need to duplicate and edit the file multiple times, leading to redundancy and error-prone configurations

main.tf

```
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "5.75.1"
    }
  }
}

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

resource "aws_instance" "ec2_instance" {
  ami = var.aws_ami
  instance_type = var.aws_instance_type

  root_block_device {
    volume_size = 8
    volume_type = "gp2"
    delete_on_termination = true
  }

  tags = {
    Name = "ec2-instance"
  }
}
```

variables.tf

```
variable "aws_ami" {
  description = "value of ami-id"
  type = string
  default = "ami-0453ec754f44f9a4a"
}

variable "aws_instance_type" {
    description = "Give me te instance id"
    type = string
}
```

```
variable "aws_ami" {
  description = "value of ami-id"
  type = string
  default = "ami-0453ec754f44f9a4a"
}
```

In this variable we are giving ami a default value that is "ami-0453ec754f44f9a4a"

```
variable "aws_instance_type" {
    description = "Give me te instance id"
    type = string
}
```

In this variable we need to give information in the terminal which looks like this

```
PS D:\Downloads\TERRAFORM\AWS\tf-variables> terraform apply
var.aws_instance_type
  Give me te instance id

  Enter a value:
```

But if you try to give the wrong answer something like t2.mic which is not part of ec2 instance type, terminal will take the output but when you try to do “terraform apply” it will show error.

```
Error: creating EC2 Instance: operation error EC2: RunInstances, https
response error StatusCode: 400, RequestID: f7bb7155-4bfd-407e-92d7-3171d6
074b50, api error InvalidParameterValue: Invalid value 't2.mic' for Insta
nceType.

   with aws_instance.ec2_instance,
   on main.tf line 14, in resource "aws_instance" "ec2_instance":
   14: resource "aws_instance" "ec2_instance" {
```

Problem Without Validation:

If a team member accidentally sets an unsupported instance type like m5.large, Terraform will proceed and deploy the instance. This can result in:

- ***Increased Costs:*** Deploying an expensive instance unnecessarily.
- ***Deployment Failures:*** If the specified instance type is not supported in the target AWS region.

AWS > tf-variables > variables.tf > variable "aws_instance_type"

```
variable "aws_ami" {
  description = "value of ami-id"
  type = string
  default = "ami-0453ec754f44f9a4a"
}

variable "aws_instance_type" {
    description = "Give me te instance id"
    type = string
    validation {
        condition = var.aws_instance_type == "t2.micro" || var.aws_instance_type == "t2.nano"
        error_message = "The instance type must be t2.micro or t2.nano"
    }
}
```

If you use validation in this case, you only option will be to choose either “t2.nano” or “t2.micro” if you type anything except these two it will show

```
Error: Invalid value for variable

  on variables.tf line 7:
   7: variable "aws_instance_type" {
     ├────────
     │ var.aws_instance_type is "t3.large"

The instance type must be t2.micro or t2.nano
```

If block have multiple variables in main.tf

```
14 resource "aws_instance" "ec2_instance" {
15   ami = var.aws_ami
16   instance_type = var.aws_instance_type
17
18   root_block_device {
19     volume_size = var.root_block_device_size
20     volume_type = var.root_block_device_type
21     delete_on_termination = true
22   }
23
24   tags = {
25     Name = "ec2-instance"
26   }
27 }
```

Then in spite of writing in this in variables.tf

```
16 variable "root_block_device_size" {
17   description = "Volume size"
18   type = number
19   default = 20
20 }
21
22 variable "root_block_device_type" {
23   description = "Volume type"
24   type = string
25   default = "gp2"
26 }
```

We can write

variables.tf

```
variable "ec2_config" {
  type = object({
    v_size = number
    v_type = string
  })
  default = {
    v_size = 20
    v_type = "gp2"
  }

}
```

main.tf

```
14 resource "aws_instance" "ec2_instance" {
15   ami = var.aws_ami
16   instance_type = var.aws_instance_type
17
18   root_block_device {
19     volume_size = var.ec2_config.v_size
20     volume_type = var.ec2_config.v_type
21     delete_on_termination = true
22   }
23
24   tags = {
25     Name = "ec2-instance"
26   }
27 }
```

Use of map in Terraform:

In Terraform, a map is a data structure that allows you to define and access related data efficiently. It's particularly useful for organizing and managing configurations when you need to group related values.

variables.tf

```
variable "additional_tags" {
  type = map(string)
  default = {          #can give any number of tags
    "name" = "value"
  }
}
```

main.tf

```
resource "aws_instance" "ec2_instance" {
  ami = var.aws_ami
  instance_type = var.aws_instance_type

  root_block_device {
    volume_size = var.ec2_config.v_size
    volume_type = var.ec2_config.v_type
    delete_on_termination = true
  }

  tags = merge(var.additional_tags,{
    Name = "ec2-instance"
  })
}
```

In this including “e2-instance” tags there can be any number of tags as we are using map.

Use of flatten in Terraform:

flatten combines **nested lists** into a **single flat list**.
It removes any nested layers.

Use Case: Simplify a list of lists.

Example:

hcl Copy code

```
variable "nested_list" {
  default = [[1, 2], [3, 4], [5, [6, 7]]]
}

output "flat_list" {
  value = flatten(var.nested_list)
}
```

Output:

plaintext Copy code

```
[1, 2, 3, 4, 5, 6, 7]
```

- The nested lists `[[1, 2], [3, 4], [5, [6, 7]]]` become `[1, 2, 3, 4, 5, 6, 7]`.
- `flatten` removes **all layers** of nesting.

Use of lookup in Terraform:

lookup retrieves a **value** from a **map** based on a key.
If the key is not found, you can provide a default value.

Use Case: Access values from a map safely.

Example:

hcl Copy code

```
variable "instance_types" {
  type    = map
  default = {
    "dev"  = "t2.micro"
    "prod" = "t2.large"
  }
}

output "dev_instance" {
  value = lookup(var.instance_types, "dev", "t2.nano")
}

output "test_instance" {
  value = lookup(var.instance_types, "test", "t2.nano")
}
```

Output:

plaintext Copy code

```
dev_instance = "t2.micro"
test_instance = "t2.nano"
```

- `lookup(var.instance_types, "dev", "t2.nano")` : Finds the value `"t2.micro"`.
- `lookup(var.instance_types, "test", "t2.nano")` : Since `"test"` is not in the map, the **default value** `"t2.nano"` is returned.

ENVIROMENT VARIABLES

Whenever you do “terraform apply” it always asks “ `Give me te instance id` ” despite typing input everytime we will be setting environment variables by writing given command in the terminal

Syntax: -
“export TF_VAR_key=value”

Example:-
“***export TF_VAR_aws_instance_type=t3.micro***”

If we again try to do either “terraform plan” or “terraform apply” it will directly execute without asking the input.

We can also change t3.micro to t3.nano we can rewrite the command “***export TF_VAR_aws_instance_type=t3.mano***”

terraform.tfvars

The **terraform.tfvars** file is used to assign values to the variables declared in **variables.tf**.

It helps separate configuration from implementation, making the code reusable and modular.

variables.tf	terraform.tfvars
Defines the variables Terraform expects.	Provides values for the defined variables.
Declarative (defines variable schema).	Assigns actual values to variables.
Declares variable **"region" {}.**	Assigns **region = "us-west-2"**

We have to create a new file name “terraform.tfvars”

variables.tf

```hcl
variable "aws_ami" {
  description = "value of ami-id"
  type = string
  default = "ami-0453ec754f44f9a4a"
}

variable "aws_instance_type" {
    description = "Give me te instance id"
    type = string
    validation {
        condition = var.aws_instance_type == "t2.micro" || var.aws_instance_type == "t2.nano"
        error_message = "The instance type must be t2.micro or t2.nano"
    }
}

variable "ec2_config" {
  type = object({
    v_size = number
    v_type = string
  })
  default = {
    v_size = 20
    v_type = "gp2"
  }
}

variable "additional_tags" {
  type = map(string)
  default = {          #can give any number of tags
    "name" = "value"
  }
}
```

terraform.tfvars

```hcl
aws_ami = "ami-0453ec754f44f9a4a"

aws_instance_type = "t2.nano"

ec2_config = {
  v_size = 20
  v_type = "gp2"
}

additional_tags = {
  project = "terrafrom"
  cloud = "aws"
}
```

terraform.auto.tfvars

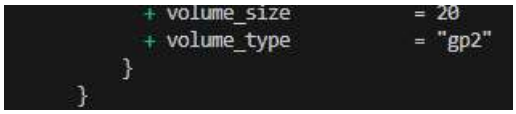
It is a special file in Terraform used to automatically assign values to variables.
Terraform automatically loads this file if it exists in the working directory.
It works similarly to **terraform.tfvars**, but with a key difference: Terraform does not require you to explicitly specify this file during execution.

terraform.tfvars

```hcl
aws_ami = "ami-0453ec754f44f9a4a"

aws_instance_type = "t2.nano"

ec2_config = {
  v_size = 20
  v_type = "gp2"
}

additional_tags = {
  project = "terrafrom"
  cloud = "aws"
}
```

Terraform.auto.tfvars

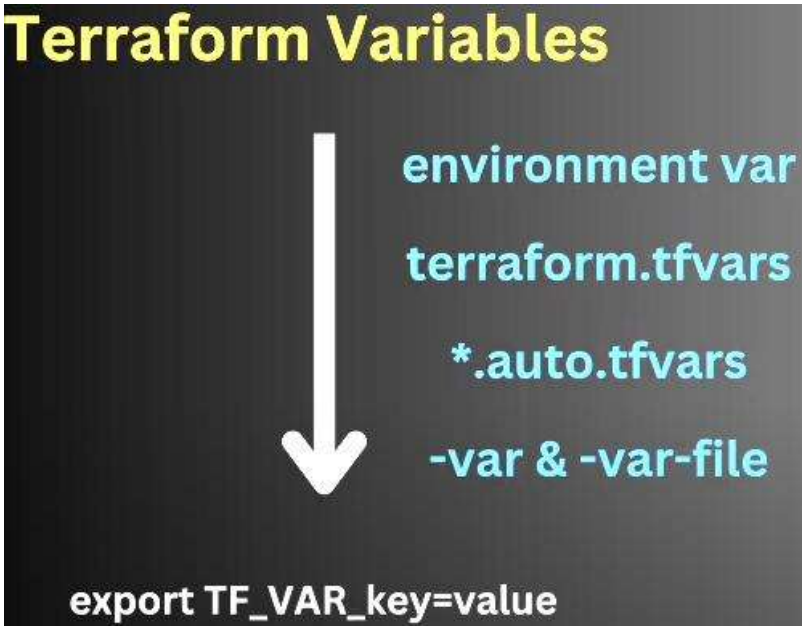
```hcl
ec2_config = {
  v_size = 50
  v_type = "gp2"
}
```

Output before having “terraform.auto.tf.vars” after applying “terraform plan”	Output after having “terraform.auto.tf.vars” after applying “terraform plan”
	

```
      + volume_size           = 20
      + volume_type           = "gp2"
    }
}
```

```
    + volume_size           = 50
    + volume_type           = "gp2"
}
```

It means “terraform.auto.tfvars” have more parity than “terraform.tfvars” it means that when we apply “terraform plan” in the terminal it first check the configurations of “terraform.auto.tfvars” then “”terraform.tf.vars



We also write `tf-variables % terraform plan -var='ec2_config={v_size=50, v_type="` in the terminal

25. LOCAL VARIABLES IN TERRAFORM

Used to simplify and organize complex configurations by defining intermediate values within a module. They allow you to create reusable and readable logic without polluting your input variables or hardcoding values.

Key Features of Local Variables

Defined with **locals** Block:

- Local variables are declared within a locals block.
- Example:

```
locals {
  environment = "staging"
  instance_type = "t3.medium"
}
```

Accessed Using **local.<name>**:

- You can reference local variables using the local namespace.
- Example:

```
resource "aws_instance" "example" {
  instance_type = local.instance_type
  tags = {
    Environment = local.environment
  }
}
```

Evaluated Dynamically:

- Local variables can be used to compute values dynamically, often combining other inputs, resources, or expressions.

Scoped to the Module:

- They are only available within the module where they are defined, making them ideal for encapsulating logic.

26. TERRAFORM: OPERATIONS & EXPRESSIONS

To explain Operations and Expressions we will create another folder “tf-operators-expressions” inside that we will create another file “main.tf”

```
terraform {
}
```

```
# Number List
variable "num_list" {
  type = list(number)
  default = [1,2,3]
}
```

```
# Object List of number
variable "person_list" {
  type = list(object({
    name = string
    age = number
  }))
```

```
# Object List of number
variable "person_list" {
  type = list(object({
    name = string
    age = number
  }))

  default = [
    {
      name = "John"
      age = 30
    },
    {
      name = "Jane"
      age = 25
    },
    {
      name = "Jony"
      age = 40
    }
  ]
}
```

```
# map, based on key and value pair
# jist like dictionary in python
variable "map_list" {
  type = map(number)
  default = {
    "name" = 1
    "age" = 2
  }
}
```

```
#calculation
locals {
    mul =2 * 2
    add = 2 + 2
    sub = 2 - 2
    div = 2 / 2
    not_equal = 2 != 3

    #to get the double
    double= [for i in var.num_list : i * 2]

    #to get the odd number
    odd= [for i in var.num_list : i if i % 2 != 0]

    #to get the names of the person from the above map list
    name = [for i in var.person_list : i.name]

    # work with map
    map_info= [for key, value in var.map_list : value]

    #double map
    double_map_info= [for key, value in var.map_list : value * 2]
}
```

```
69  output "output_1" {
70    value = local.mul
71  }
72
73  output "output_2" {
74    value = local.not_equal
75  }
76
77  output "output_3" {
78    value = var.num_list
79  }
80
81  output "output_4" {
82    value = local.double
83  }
84
85  output "output_5" {
86    value = local.odd
87  }
88
89  output "output_6" {
90    value = local.name
91  }
```

```
Changes to Outputs:
  + output_1 = 4
  + output_2 = true
  + output_3 = [
      + 1,
      + 2,
      + 3,
    ]
  + output_4 = [
      + 2,
      + 4,
      + 6,
    ]
  + output_5 = [
      + 1,
      + 3,
    ]
  + output_6 = [
      + "John",
      + "Jane",
      + "Jony",
    ]
  + output_7 = [
      + 2,
      + 1,
    ]
  + output_8 = [
      + 4,
      + 2,
    ]
```

```
93  output "output_7" {
94    value = local.map_info
95  }
96
97  output "output_8" {
98    value = local.double_map_info
99  }
```

27. TERRAFORM: FUNCTIONS

To explain Operations and Expressions we will create another folder “tf-operators-expressions” inside that we will create another file “main.tf”

```
terraform {

}
```

```
locals {
  value = "hello , world"
}
# convert the string to upper case that is present in the local.value
output "output_1" {
  value = upper(local.value)
}
```

```
+ output_1  = "HELLO , WORLD"
```

```
locals {
  value = "hello , world"
}
# convert the string to lower case that is present in the local.value
output "output_2" {
  value = lower(local.value)
}
```

```
+ output_2  = "hello , world"
```

```
locals {
  value = "hello , world"
}
# returns boolean value true, if local.value starts with hello
output "output_3" {
  value = startswith(local.value, "hello")
}
```

```
+ output_3  = true
```

```
locals {
  value = "hello , world"
}
```

```
24  # return boolean value true, if local.value ends with world
25  output "output_4" {
26    value = endswith(local.value, "world")
27  }
```

```
+ output_4  = true
```

```
5   locals {
6     value = "hello , world"
7   }
29  # return the first 4 characters of local.value
30  #including the 0th index and excluding the 4th index
31  output "output_5" {
32    value = substr(local.value, 0, 4)
33  }
```

```
+ output_5  = "hell"
```

```
5   locals {
6     value = "hello , world"
7   }
35  # return the length of local.value
36  output "output_6" {
37    value = length(local.value)
38  }
```

```
+ output_6  = 13
```

```
5   locals {
6     value = "hello , world"
7   }
40  # splits a string into a list
41  output "output_7" {
42    value = split(" ", local.value)
43  }
```

```
+ output_7  = [
    + "hello",
    + ",",
    + "world",
  ]
```

```
5   locals {
6     value = "hello , world"
7   }
45  # returns the maximum value
46  output "output_8" {
47    value = max(1,8,2,3,9,0,10)
48  }
```

```
+ output_8  = 10
```

```
locals {
 value = "hello , world"
}
# returns the minimum value
output "output_9" {
  value = min(1,8,2,3,9,0,10)
}
```

```
+ output_9  = 0
```

```
variable "string_list" {
  type = list(string)
  default = [ "server1", "server2","server3","server1" ]
}
# returns the length of the list
output "output_10" {
  value = length(var.string_list)
}
```

```
 + output_10 = 4
```

```
variable "string_list" {
  type = list(string)
  default = [ "server1", "server2","server3","server1" ]
}
# joins the elements of the list with -
output "output_11" {
  value = join("-", var.string_list)
}
```

```
+ output_11 = "server1-server2-server3-server1"
```

```
variable "string_list" {
  type = list(string)
  default = [ "server1", "server2","server3","server1" ]
}
# returns true if the string is present in the list
output "output_12" {
  value = contains(var.string_list, "server1")
}
```

```
+ output_12 = true
```

```
variable "string_list" {
  type = list(string)
  default = [ "server1", "server2","server3","server1" ]
}
```

```
76   # returns the unique elements of the list
77   output "output_13" {
78     value = var.string_list
79   }
```

```
+ output_13 = [
    + "server1",
    + "server2",
    + "server3",
    + "server1",
  ]
```

```
56   variable "string_list" {
57     type = list(string)
58     default = [ "server1", "server2","server3","server1" ]
59   }
```

```
81   # convert the list to set to remove duplicates
82   output "output_14" {
83     value = toset(var.string_list)
84   }
```

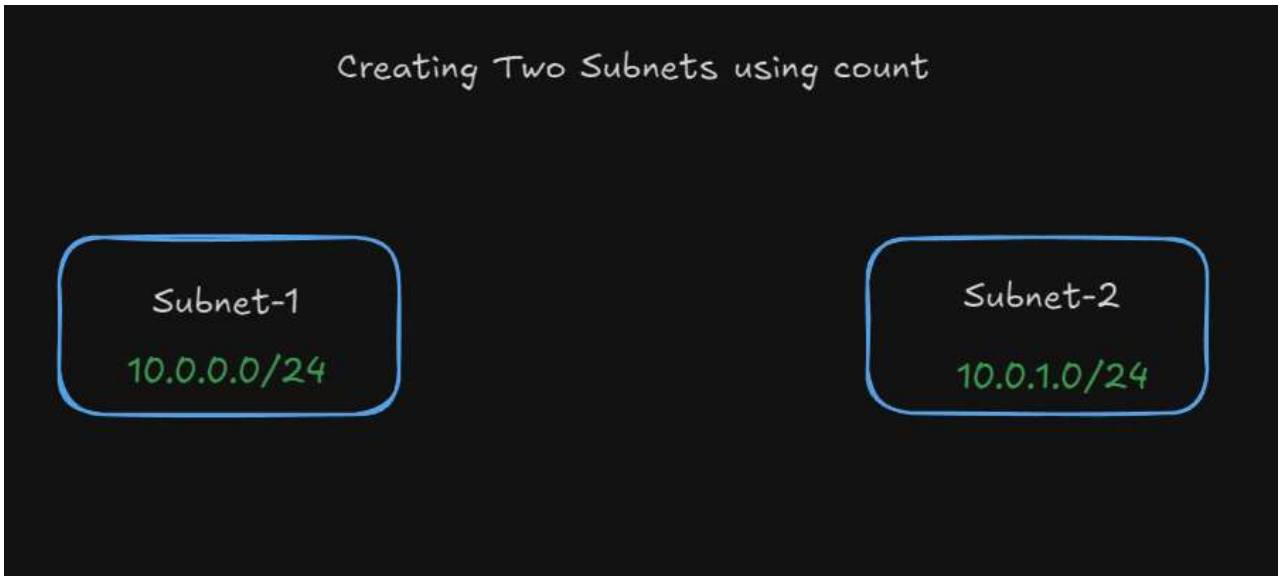
```
+ output_14 = [
    + "server1",
    + "server2",
    + "server3",
  ]
```

28. TERRAFORM: MULTIPLE RESOURCES

In this we will learn about the use of "***count***"

The ***count*** parameter in Terraform allows you to create multiple instances of a resource or module by specifying the desired number of instances dynamically.
It provides a simple way to scale resources programmatically and reduces code duplication.

For the first demo we will be creating a VPC with two subnets using count in terraform



Int order create this demo we will create new folder name "tf-multiple-resources" in AWS folder inside that we will be creating "main.tf" in "main.tf" file there are subdivision

main.tf

```
#required provider
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "5.75.1"
    }
  }
}
```

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

```
#locals variables
locals {
  project_name = "tf_vpc"
}
```

```
#creating resource for vpc
resource "aws_vpc" "tf_vpc" {
  cidr_block = "10.0.0.0/16"
  tags = {
    Name = "${local.project_name}"
  }
}
```

```
#creating resource for subnet for the vpc
resource "aws_subnet" "tf_vpc_subnet" {
  vpc_id = "${aws_vpc.tf_vpc.id}"
  cidr_block= "10.0.${count.index}.0/24"
  count = 2
  tags = {
    Name = "${local.project_name}-subnet-${count.index}"
  }
}
```

count

- The **count** parameter is set to **2**, which means Terraform will create **two instances** of the **aws_subnet** resource.
- Each instance will have unique properties based on the use of **count.index**.

count.index

- The **count.index** is a **zero-based index** representing the instance number of the resource being created.
- Since **count = 2**, the value of **count.index** will be:
  - **0** for the first subnet instance.
  - **1** for the second subnet instance.

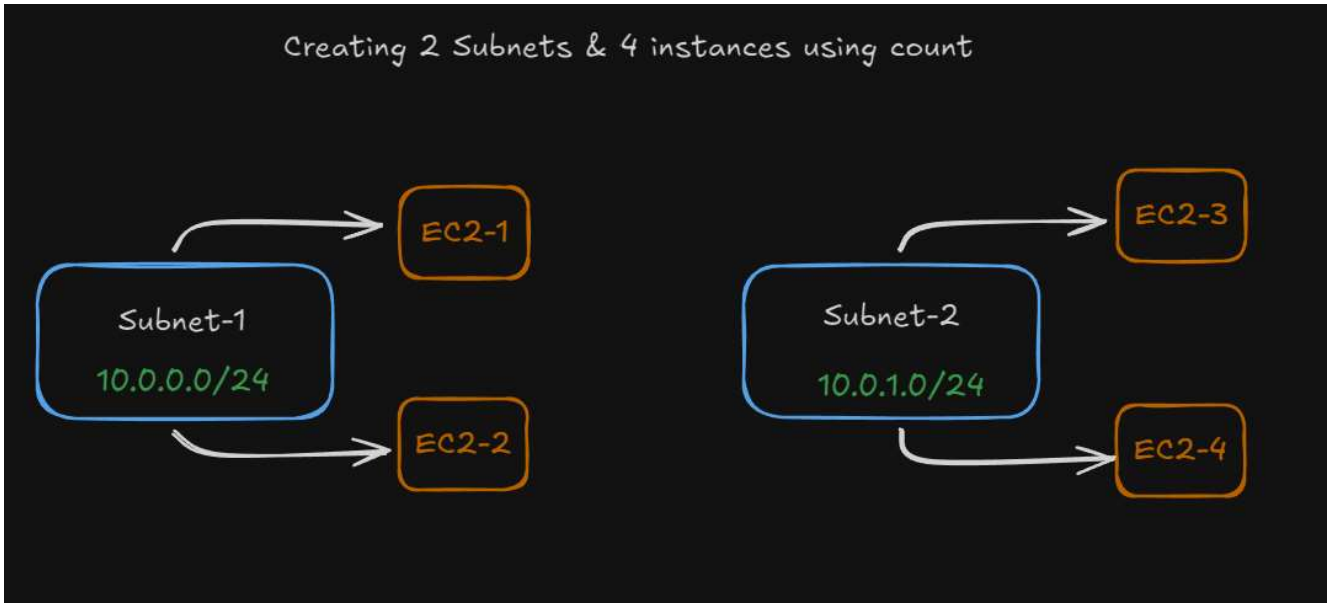
```
#output of one subnet id
output "subnet_ids" {
  value = aws_subnet.tf_vpc_subnet[0].id
}

#output of all subnet ids
output "all_subnet_ids" {
  value = aws_subnet.tf_vpc_subnet[*].id
}
```

Now type ***"terraform init" → "terraform plan" → "terraform apply"***

```
aws_vpc.tf_vpc: Creating...
aws_vpc.tf_vpc: Creation complete after 5s [id=vpc-04af14419a8a2f434]
aws_subnet.tf_vpc_subnet[0]: Creating...
aws_subnet.tf_vpc_subnet[1]: Creating...
aws_subnet.tf_vpc_subnet[0]: Creation complete after 3s [id=subnet-0213b8f9a2df80d9c]
aws_subnet.tf_vpc_subnet[1]: Creation complete after 3s [id=subnet-04a9b3aea5582dad9]

Apply complete! Resources: 3 added, 0 changed, 0 destroyed.

Outputs:

all_subnet_ids = [
  "subnet-0213b8f9a2df80d9c",
  "subnet-04a9b3aea5582dad9",
]
subnet_ids = "subnet-0213b8f9a2df80d9c"
PS D:\Downloads\TERRAFORM\aws\tf-multiple-resources>
```

Now go to aws console to see the result

For the second demo we will be creating a VPC with two subnets and four instances using count in terraform



Continuations of above infrastructure of the demo 1 project

```
#creating resources of instances for subnets
resource "aws_instance" "ec2_instance" {
  count = 4
  ami= "ami-0453ec754f44f9a4a"
  instance_type = "t2.micro"
  subnet_id = element(aws_subnet.tf_vpc_subnet[*].id, count.index % length(aws_subnet.tf_vpc_subnet))
  # 0 % 2 = 0
  # 1 % 2 = 1
  # 2 % 2 = 0
  # 3 % 2 = 1
  tags = {
    Name = "${local.project_name}-instance-${count.index}"
  }
}
```

```
subnet_id = element(aws_subnet.tf_vpc_subnet[*].id, count.index % length(aws_subnet.tf_vpc_subnet))
```

Key Components:

- **aws_subnet.tf_vpc_subnet[*].id**:
  - Retrieves the list of all subnet IDs created by the aws_subnet.tf_vpc_subnet resource.

- **length(aws_subnet.tf_vpc_subnet)**:
  - Returns the total number of subnets (e.g., 2 subnets in the earlier example).

- **count.index % length(aws_subnet.tf_vpc_subnet)**:
  - Distributes the instances evenly across the available subnets using modulo operation.

- This ensures that the subnet index cycles through the available subnets (e.g., 0, 1, 0, 1 for 2 subnets).

- **element(...)**:
  - Retrieves the subnet ID at the calculated index.

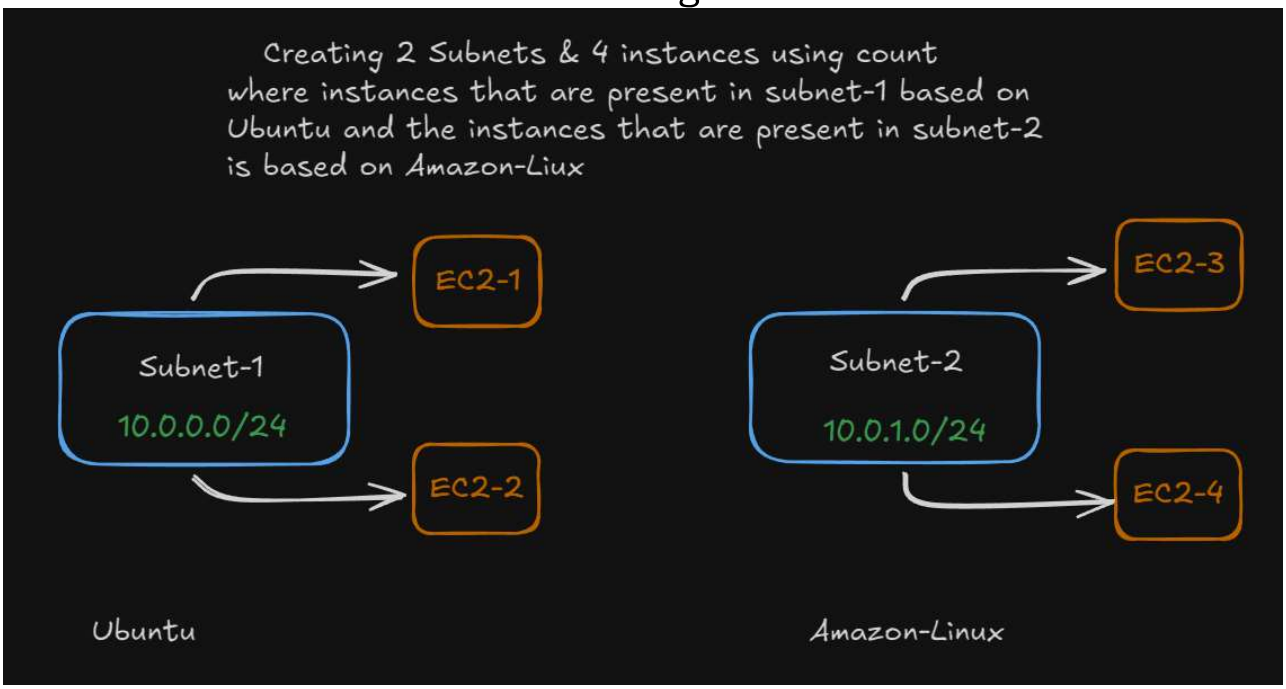
Now type “terraform plan” → ***“terraform apply”***

```
aws_vpc.tf_vpc: Creating...
aws_vpc.tf_vpc: Creation complete after 5s [id=vpc-03428ccec23dd1d36]
aws_subnet.tf_vpc_subnet[0]: Creating...
aws_subnet.tf_vpc_subnet[1]: Creating...
aws_subnet.tf_vpc_subnet[0]: Creation complete after 2s [id=subnet-01152e2f4be905bfc]
aws_subnet.tf_vpc_subnet[1]: Creation complete after 2s [id=subnet-0acb21115d40a4b6a]
aws_instance.ec2_instance[3]: Creating...
aws_instance.ec2_instance[1]: Creating...
aws_instance.ec2_instance[0]: Creating...
aws_instance.ec2_instance[2]: Creating...
aws_instance.ec2_instance[3]: Still creating... [10s elapsed]
aws_instance.ec2_instance[1]: Still creating... [10s elapsed]
aws_instance.ec2_instance[2]: Still creating... [10s elapsed]
aws_instance.ec2_instance[0]: Still creating... [10s elapsed]
aws_instance.ec2_instance[3]: Creation complete after 16s [id=i-0122fd743b77ae3c7]
aws_instance.ec2_instance[1]: Creation complete after 16s [id=i-048974bf381b862c9]
aws_instance.ec2_instance[0]: Creation complete after 16s [id=i-07928169bd5e480bf]
aws_instance.ec2_instance[2]: Creation complete after 17s [id=i-0e9df9e2418a98a00]

Apply complete! Resources: 7 added, 0 changed, 0 destroyed.

Outputs:

all_subnet_ids = [
  "subnet-01152e2f4be905bfc",
  "subnet-0acb21115d40a4b6a",
]
```

Now go to aws console to see the result

Name	Subnet IDs
tf_vpc-instance-0	subnet-01152e2f4be905bfc
tf_vpc-instance-1	subnet-0acb21115d40a4b6a
tf_vpc-instance-2	subnet-01152e2f4be905bfc
tf_vpc-instance-3	subnet-0acb21115d40a4b6a

For the third demo we will be creating



First create ***variables.tf***

```
variable "ec2_instance_ami" {
  type = list(object({
    ami = string
  }))

}
```

Second create ***terraform.tfvars***

```
ec2_instance_ami = [{
  ami = "ami-0453ec754f44f9a4a"   #amazon-linux
  },{
  ami = "ami-0e2c8caa4b6378d8c" #ubuntu
  }
]
```

Third continuations of above infrastructure of the demo 2 project but slight in ***resource "aws_instance"***

```hcl
#creating resources of instances for subnets
resource "aws_instance" "ec2_instance" {
  count = 4
  ami= var.ec2_instance_ami[count.index % length(aws_subnet.tf_vpc_subnet)].ami
  instance_type = "t2.micro"
  subnet_id = element(aws_subnet.tf_vpc_subnet[*].id, count.index % length(aws_subnet.tf_vpc_subnet))
  # 0 % 2 = 0
  # 1 % 2 = 1
  # 2 % 2 = 0
  # 3 % 2 = 1
  tags = {
    Name = "${local.project_name}-instance-${count.index}"
  }
}
```

Now type ***"terraform plan" → "terraform apply"***

```
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value: yes

aws_vpc.tf_vpc: Creating...
aws_vpc.tf_vpc: Creation complete after 6s [id=vpc-0525130de23e90f5d]
aws_subnet.tf_vpc_subnet[0]: Creating...
aws_subnet.tf_vpc_subnet[1]: Creating...
aws_subnet.tf_vpc_subnet[1]: Creation complete after 2s [id=subnet-0679e5b84de8c7bf6]
aws_subnet.tf_vpc_subnet[0]: Creation complete after 2s [id=subnet-0e500985a86971b87]
aws_instance.ec2_instance[2]: Creating...
aws_instance.ec2_instance[3]: Creating...
aws_instance.ec2_instance[1]: Creating...
aws_instance.ec2_instance[0]: Creating...
aws_instance.ec2_instance[2]: Still creating... [10s elapsed]
aws_instance.ec2_instance[3]: Still creating... [10s elapsed]
aws_instance.ec2_instance[0]: Still creating... [10s elapsed]
aws_instance.ec2_instance[1]: Still creating... [10s elapsed]
aws_instance.ec2_instance[3]: Creation complete after 16s [id=i-0c0da483c5e6d1347]
aws_instance.ec2_instance[1]: Still creating... [20s elapsed]
aws_instance.ec2_instance[0]: Still creating... [20s elapsed]
aws_instance.ec2_instance[2]: Still creating... [20s elapsed]
aws_instance.ec2_instance[2]: Creation complete after 26s [id=i-010bf9117caa4c8c0]
aws_instance.ec2_instance[0]: Still creating... [30s elapsed]
aws_instance.ec2_instance[1]: Still creating... [30s elapsed]
aws_instance.ec2_instance[0]: Creation complete after 37s [id=i-0502c64770029f59d]
aws_instance.ec2_instance[1]: Creation complete after 37s [id=i-01f01b9d832f50fc6]

Apply complete! Resources: 7 added, 0 changed, 0 destroyed.

Outputs:

all_subnet_ids = [
  "subnet-0e500985a86971b87",
  "subnet-0679e5b84de8c7bf6",
]
```

Now go to aws console to see the result

Name	Subnet IDs	Image ID
tf_vpc-instance-0	subnet-0e500985a86971b87	ami-0453ec754f44f9a4a
tf_vpc-instance-1	subnet-0679e5b84de8c7bf6	ami-0e2c8caa4b6378d8c
tf_vpc-instance-2	subnet-0e500985a86971b87	ami-0453ec754f44f9a4a
tf_vpc-instance-3	subnet-0679e5b84de8c7bf6	ami-0e2c8caa4b6378d8c

*Note – if all work is done you should use command **terraform destroy** to delete all the services you have used while doing project.*

Use of "for_each".

"for_each" → only accepts "sets" and "map"
In order to learn the use of "for_each" we will have to do a slight changes in "***terraform.vars***" , "***terraform.tfvars***" and "***main.tf***" specially in ***resource "aws_instance"*** rest all will remain same.

We will do changes in demo 3 project

terraform.vars

```
variable "ec2_instance_ami_map" {
  # key-value pair
  type = map(object({
    ami = string
  }))
}
```

terraform.tfvars

```
ec2_instance_ami_map = {
  "amazon-1" = {
    ami = "ami-0453ec754f44f9a4a"   #amazon-linux
  },

  "ubuntu-1" = {
    ami = "ami-0e2c8caa4b6378d8c" #ubuntu
  }

  "amazon-2" = {
    ami = "ami-0453ec754f44f9a4a"   #amazon-linux
  },

  "ubuntu-2" = {
    ami = "ami-0e2c8caa4b6378d8c" #ubuntu
  }
}
```

main.tf

```hcl
#creating resources of instances for subnets
resource "aws_instance" "ec2_instance" {
  for_each = var.ec2_instance_ami_map
  # we will get each.key and each.value from the map
  ami= each.value.ami
  instance_type = "t2.micro"
  subnet_id = element(aws_subnet.tf_vpc_subnet[*].id,index(keys(var.ec2_instance_ami_map),each.key) % length(aws_subnet.tf_vpc_subnet))

  tags = {
    Name = "${local.project_name}-instance-${each.key}"
  }
}
```

subnet_id = element(aws_subnet.tf_vpc_subnet[*].id, index(keys(var.ec2_instance_ami_map), each.key) % length(aws_subnet.tf_vpc_subnet))

Explanation of subnet_id:

aws_subnet.tf_vpc_subnet[*].id:

- This retrieves a list of all subnet IDs associated with the aws_subnet.tf_vpc_subnet resource.

keys(var.ec2_instance_ami_map):

- Extracts all the keys from the variable ec2_instance_ami_map, which is a map defining the AMI for each instance.

each.key:

- Refers to the current key being processed in the for_each block, which corresponds to one EC2 instance in the ec2_instance_ami_map.

index(keys(var.ec2_instance_ami_map), each.key):

- Determines the index of the current key (each.key) in the list of keys extracted from the map.

length(aws_subnet.tf_vpc_subnet):

- Calculates the total number of subnets available in the aws_subnet.tf_vpc_subnet resource.

index(...) % length(...):

- This modulus operation ensures that the index cycles through the list of subnets. For example:

- If there are 3 subnets and 5 instances, the instance-to-subnet mapping would rotate (e.g., 0 -> Subnet1, 1 -> Subnet2, 2 -> Subnet3, 3 -> Subnet1, 4 -> Subnet2).
-

element(...):

- Picks the subnet ID from the list of subnet IDs (aws_subnet.tf_vpc_subnet[*].id) based on the calculated index.

Now type "terraform plan" → ***"terraform apply"***

```
Apply complete! Resources: 7 added, 0 changed, 0 destroyed.

Outputs:


all_subnet_ids = [
all_subnet_ids = [
  "subnet-09c73fd695f0f2a17",
  "subnet-09f858049e9f90b19",
]
subnet_ids = "subnet-09c73fd695f0f2a17"
```

Now go to aws console to see the result

Name	Image ID	Subnet IDs
tf_vpc-instance-amazon-1	ami-0453ec754f44f9a4a	subnet-09c73fd695f0f2a17
tf_vpc-instance-amazon-2	ami-0453ec754f44f9a4a	subnet-09f858049e9f90b19
tf_vpc-instance-ubuntu-1	ami-0e2c8caa4b6378d8c	subnet-09c73fd695f0f2a17
tf_vpc-instance-ubuntu-2	ami-0e2c8caa4b6378d8c	subnet-09f858049e9f90b19

*Note – if all work is done you should use command **terraform destroy** to delete all the services you have used while doing project.*

29. PROJECT: AWS IAM MANAGEMENT

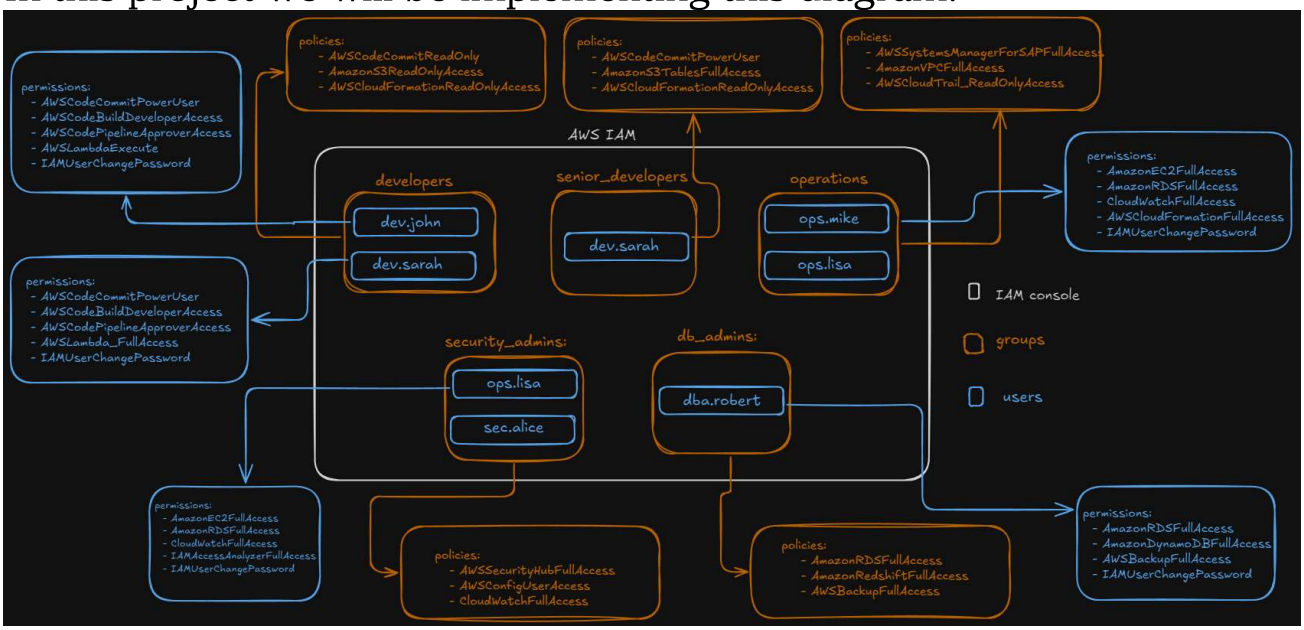
In this project you must have the knowledge of

- IAM services (users, policy, groups, Roles)
- Terraform
- Yaml configuration

In this project, we will use Terraform to provision AWS infrastructure and assign IAM policies to a group of users using AWS Identity and Access Management (IAM) service.

- Provide user and roles info via YAML file.
- Read the YAML file and process data.
- Create IAM user.
- Generate Passwords for the users
- Attract policy/roles to each user

In this project we will be implementing this diagram:



Create a new folder in “AWS” name “iam-management”, inside that folder create files like

- “***user.yaml***” for storing details of the users,groups, policies,
- “***main.tf***” for writing the infrastructure for IAM AWS services
- “***output.tf***” for storing output details

In "***user.yaml***"

STEP-1

AWS > iam-management > ! users.yaml

```yaml
# IAM Users and Roles Configuration

# Developer Team
developers:
- username: dev.john
  full_name: "John Smith"
  groups:
  - developers
  permissions:
  - AWSCodeCommitPowerUser
  - AWSCodeBuildDeveloperAccess
  - AWSCodePipelineApproverAccess
  - AWSLambdaExecute
  - IAMUserChangePassword

- username: dev.sarah
  full_name: "Sarah Johnson"
  groups:
  - developers
  - senior_developers
  permissions:
  - AWSCodeCommitPowerUser
  - AWSCodeBuildDeveloperAccess
  - AWSCodePipelineApproverAccess
  - AWSLambda_FullAccess
  - IAMUserChangePassword

# Operations Team
operations:
- username: ops.mike
  full_name: "Mike Anderson"
  groups:
  - operations
  permissions:
  - AmazonEC2FullAccess
  - AmazonRDSFullAccess
  - CloudWatchFullAccess
  - AWSCloudFormationFullAccess
  - IAMUserChangePassword

- username: ops.lisa
  full_name: "Lisa Chen"
  groups:
  - operations
  - security_admins
  permissions:
  - AmazonEC2FullAccess
  - AmazonRDSFullAccess
  - CloudWatchFullAccess
  - IAMAccessAnalyzerFullAccess
  - IAMUserChangePassword
```

STEP-2

```yaml
# Database Administrators
db_admins:
- username: dba.robert
  full_name: "Robert Wilson"
  groups:
  - db_admins
  permissions:
  - AmazonRDSFullAccess
  - AmazonDynamoDBFullAccess
  - AWSBackupFullAccess
  - IAMUserChangePassword

# Security Team
security:
- username: sec.alice
  full_name: "Alice Brown"
  groups:
  - security_admins
  permissions:
  - IAMFullAccess
  - SecurityAudit
  - AWSSecurityHubFullAccess
  - AWSConfigUserAccess
  - IAMUserChangePassword

# Groups Configuration
groups:
  developers:
    policies:
    - AWSCodeCommitReadOnly
    - AmazonS3ReadOnlyAccess
    - AWSCloudFormationReadOnlyAccess

  senior_developers:
    policies:
    - AWSCodeCommitPowerUser
    - AmazonS3TablesFullAccess
    - AWSCloudFormationReadOnlyAccess

  operations:
    policies:
    - AWSSystemsManagerForSAPFullAccess
    - AmazonVPCFullAccess
    - AWSCloudTrail_ReadOnlyAccess

  security_admins:
    policies:
    - AWSSecurityHubFullAccess
    - AWSConfigUserAccess
    - CloudWatchFullAccess

  db_admins:
    policies:
    - AmazonRDSFullAccess
    - AmazonRedshiftFullAccess
    - AWSBackupFullAccess
```

STEP-3

```
110   # Password Policy
111   password_policy:
112     minimum_length: 8
113     require_uppercase: true
114     require_lowercase: true
115     require_numbers: true
116     require_symbols: true
117     max_age_days: 90
118     prevent_reuse: 24
119     Ctrl+L to chat, Ctrl+K to generate
```

In "***main.tf***"

Provider

```
1   terraform {
2     required_providers {
3       aws = {
4         source = "hashicorp/aws"
5         version = "5.75.1"
6       }
7       random = {
8         source = "hashicorp/random"
9         version = "3.6.3"
10      }
11    }
12  }
13
14  provider "aws" {
15    # Configuration options
16    region = "us-east-1"
17  }
```

locals

{

```
#users = yamldecode(file("${path.module}/users.yaml"))
iml_data = file("./users.yaml")
```

```
+ iml-data-without-decoding    = <<-EOT
    # IAM Users and Roles Configuration

    # Developer Team
    developers:
    - username: dev.john
      full_name: "John Smith"
      groups:
      - developers
      permissions:
      - AWSCodeCommitPowerUser
      - AWSCodeBuildDeveloperAccess
      - AWSCodePipelineApproverAccess
      - AWSLambdaExecute
      - IAMUserChangePassword
```

...

```
Ctrl+L to chat, Ctrl+K to generate
iml_data_decoded = yamldecode(local.iml_data)
```

```
+ iml-data-with-decoding      = {
  + db_admins       = [
      + {
          + full_name   = "Robert Wilson"
          + groups      = [
              + "db_admins",
            ]
          + permissions = [
              + "AmazonRDSFullAccess",
              + "AmazonDynamoDBFullAccess",
              + "AWSBackupFullAccess",
              + "IAMUserChangePassword",
            ]
          + username    = "dba.robert"
        },
    ]
  + developers      = [
```

...

```
# Get all user names across different teams
all_users_names= toset(concat(
  local.iml_data_decoded.developers[*].username,
  local.iml_data_decoded.operations[*].username,
  local.iml_data_decoded.db_admins[*].username,
  local.iml_data_decoded.security[*].username
))
```

```
+ users-names                 = [
    + "dba.robert",
    + "dev.john",
    + "dev.sarah",
    + "ops.lisa",
    + "ops.mike",
    + "sec.alice",
  ]
```

```
# To get all the group names
all_groups_name=toset(flatten([
  yamldecode(local.iml_data).developers[*].groups,
  yamldecode(local.iml_data).operations[*].groups,
  yamldecode(local.iml_data).db_admins[*].groups,
  yamldecode(local.iml_data).security[*].groups
]))
```

```
+ groups-names                = [
    + "db_admins",
    + "developers",
    + "operations",
    + "security_admins",
    + "senior_developers",
  ]
```

```
# List of all teams from YAML
teams = [
    local.iml_data_decoded.developers,
    local.iml_data_decoded.operations,
    local.iml_data_decoded.db_admins,
    local.iml_data_decoded.security
  ]
```

```
+ teams-names                = [
    + [
        + {
            + full_name   = "John Smith"
            + groups      = [
                + "developers",
              ]
            + permissions = [
                + "AWSCodeCommitPowerUser",
                + "AWSCodeBuildDeveloperAccess",
                + "AWSCodePipelineApproverAccess",
                + "AWSLambdaExecute",
                + "IAMUserChangePassword",
              ]
            + username    = "dev.john"
          },
        + {
```

...

```
# Extract users and their groups as pairs
pair_users_groups = {
  for pair in flatten([
    for team in local.teams : [
      for user in team : [
        for group in user.groups : {
          username = user.username,
          group    = group
        }
      ]
    ]
  ]) : "${pair.username}-${pair.group}" => pair
}
```

```
+ pairing-users-with-groups    = {
    + "dba.robert-db_admins"          = {
        + group    = "db_admins"
        + username = "dba.robert"
      }
    + "dev.john-developers"           = {
        + group    = "developers"
        + username = "dev.john"
      }
```

...

```
# Create pairs of users and their directly assigned permissions
pair_user_permission= {
  for pair in flatten([
    for team in local.teams:[
      for user in team:[
        for permission in user.permissions :{
          username = user.username,
          permission = permission
        }
      ]
    ]
  ]) : "${pair.username}-${pair.permission}" => pair
}
```

```
+ giving-permissions-to-users = {
    + "dba.robert-AWSBackupFullAccess"          = {
        + permission = "AWSBackupFullAccess"
        + username   = "dba.robert"
      }
    + "dba.robert-AmazonDynamoDBFullAccess"     = {
        + permission = "AmazonDynamoDBFullAccess"
        + username   = "dba.robert"
      }
    + "dba.robert-AmazonRDSFullAccess"          = {
        + permission = "AmazonRDSFullAccess"
        + username   = "dba.robert"
      }
```

...

```
# Get groups and their policies from YAML
groups_policies = local.iml_data_decoded.groups
```

```
+ group-groups_policies       = {
    + db_admins         = {
        + policies = [
            + "AmazonRDSFullAccess",
            + "AmazonRedshiftFullAccess",
            + "AWSBackupFullAccess",
          ]
      }
    + developers        = {
        + policies = [
            + "AWSCodeCommitReadOnly",
            + "AmazonS3ReadOnlyAccess",
            + "AWSCloudFormationReadOnlyAccess",
          ]
      }
```

...

```
# Create pairs of groups and their assigned policies
pair_group_policy = {
  for pair in flatten([
    for group_name, group_obj in local.groups_policies : [
      for policy in lookup(group_obj, "policies", []) : {
        group_name = group_name
        policy     = policy
      }
    ]
  ]) : "${pair.group_name}-${pair.policy}" => pair
}
```

```
+ pairing-policies-with-group = {
    + db_admins-AWSBackupFullAccess                = {
        + group_name = "db_admins"
        + policy     = "AWSBackupFullAccess"
      }
    + db_admins-AmazonRDSFullAccess                = {
        + group_name = "db_admins"
        + policy     = "AmazonRDSFullAccess"
      }
```

...

```
# Get password policy configuration from YAML
password_policy = local.iml_data_decoded.password_policy
```

```
+ password_policy = {
    + max_age_days      = 90
    + minimum_length    = 8
    + prevent_reuse     = 24
    + require_lowercase = true
    + require_numbers   = true
    + require_symbols   = true
    + require_uppercase = true
  }
```

}

 # Create IAM users for all team members

```
resource "aws_iam_user" "users" {
  for_each = local.all_users_names
  name = each.value
}
```

 # Generate AWS access keys for all users

```
resource "aws_iam_access_key" "user_keys" {
  for_each = local.all_users_names
  user = each.key
}
```

 # Set up console access for users with initial password requirements

```
resource "aws_iam_user_login_profile" "user_login_profile" {
  for_each = local.all_users_names
  user     = each.key
  password_length = 20
  password_reset_required = true
}
```

Configure organization-wide password policy settings

```
resource "aws_iam_account_password_policy" "password_policy" {
  minimum_password_length        = local.password_policy.minimum_length
  require_uppercase_characters   = local.password_policy.require_uppercase
  require_lowercase_characters   = local.password_policy.require_lowercase
  require_numbers                = local.password_policy.require_numbers
  require_symbols                = local.password_policy.require_symbols
  max_password_age               = local.password_policy.max_age_days
  password_reuse_prevention      = local.password_policy.prevent_reuse

  # Optional: Allow users to change their own password
  allow_users_to_change_password = true
}
```

Create IAM groups for different team roles

```
resource "aws_iam_group" "groups" {
  for_each = local.all_groups_name
  name = each.value
}
```

Assign users to their respective IAM groups

```
resource "aws_iam_user_group_membership" "pairing_users_groups" {
  for_each = local.pair_users_groups

  user   = each.value.username
  groups = [each.value.group]
}
```

Attach AWS managed policies to IAM groups

```
resource "aws_iam_group_policy_attachment" "group_policy_attachment" {
  for_each = local.pair_group_policy
  group    = each.value.group_name
  policy_arn = "arn:aws:iam::aws:policy/${each.value.policy}"
}
```

Attach AWS managed policies directly to users

```
resource "aws_iam_user_policy_attachment" "user_policy_attachment" {
  for_each = local.pair_user_permission
  user       = each.value.username
  policy_arn = "arn:aws:iam::aws:policy/${each.value.permission}"
}
```

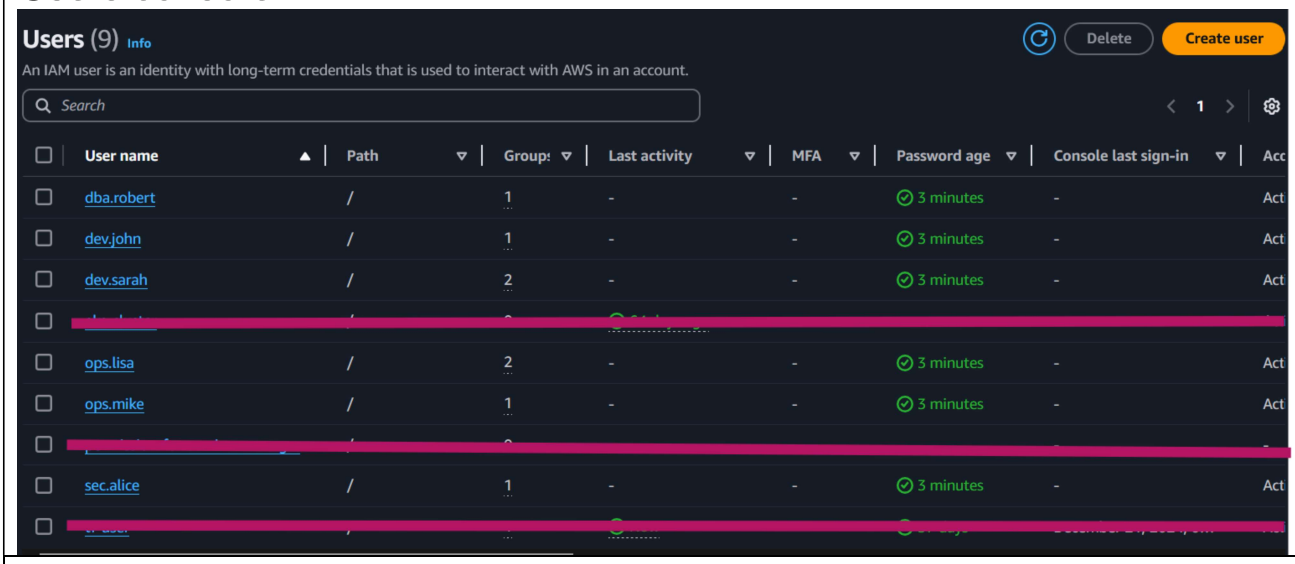
In "***output.tf***"

```
output "iml-data-without-decoding" {
  value = local.iml_data
}

Pieces: Comment | Pieces: Explain
output "iml-data-with-decoding" {
  value = local.iml_data_decoded
}

Pieces: Comment | Pieces: Explain
output "users-names" {
  value = local.all_users_names
}

Pieces: Comment | Pieces: Explain
output "groups-names" {
  value = local.all_groups_name
}

Pieces: Comment | Pieces: Explain
output "teams-names" {
  value = local.teams
}

Pieces: Comment | Pieces: Explain
output "policy-names" {
  value = toset(flatten([
    yamldecode(local.iml_data).developers[*].permissions,
    yamldecode(local.iml_data).operations[*].permissions,
    yamldecode(local.iml_data).db_admins[*].permissions,
    yamldecode(local.iml_data).security[*].permissions
  ]))
}
Pieces: Comment | Pieces: Explain
output "pairing-users-with-groups" {
  value = local.pair_users_groups
}
```

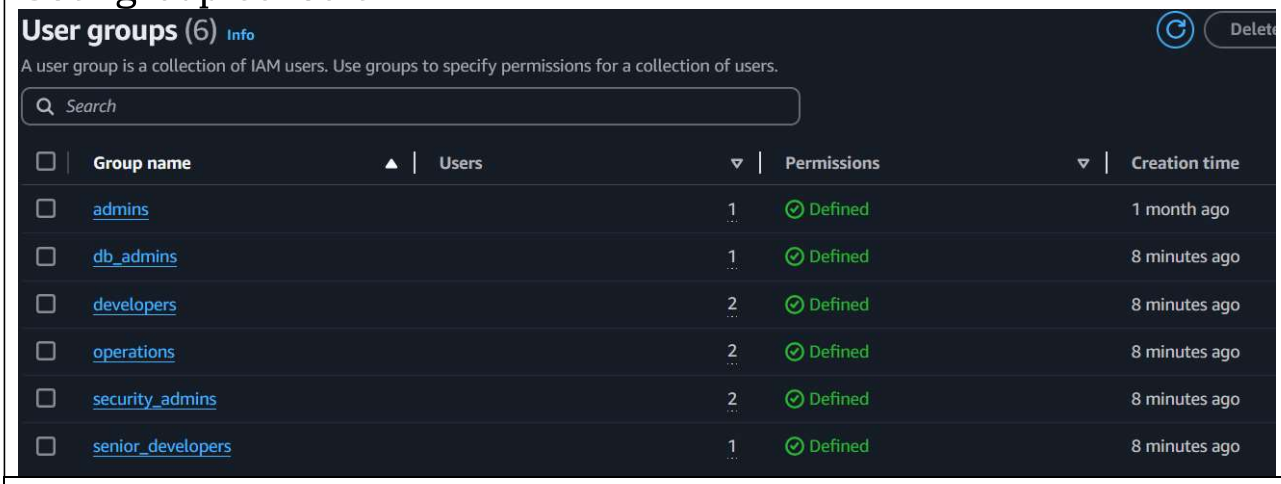
```
Pieces: Comment | Pieces: Explain
output "giving-permissions-to-users" {
  value = local.pair_user_permission
}

Pieces: Comment | Pieces: Explain
output "group-groups_policies" {
  value = local.groups_policies
}

Pieces: Comment | Pieces: Explain
output "pairing-policies-with-group" {
  value = local.pair_group_policy
}

Pieces: Comment | Pieces: Explain
output "password-policies" {
  value = local.password_policy
}
```

After the implementation of "***terraform apply***" we will see these outputs in the ***AWS IAM console***

Users console



User group console



You can log in to any user account to verify its functionality. However, it is the admin's responsibility to assign the account number, username, and password.
After logging in with the password provided by the admin, AWS will prompt you to change the password to one of your choices. If you prefer not to change it, you can skip this step.

*Note – if all work is done you should use command **terraform destroy** to delete all the services you have used while doing project.*

30. TERRAFORM MODULES

- Modules are containers for multiple resources that are used together.
- A module consists of the collection of ***.tf*** and/or ***.tf.json*** files kept together in a directory.
- Modules are the main way to package and reuse resource configurations with terraform.

Real-Life Scenario: Deploying an AWS VPC

Imagine you need to deploy **multiple VPCs** (for dev, staging, and prod environments) with the same structure but different configurations (e.g., CIDR blocks, subnets).

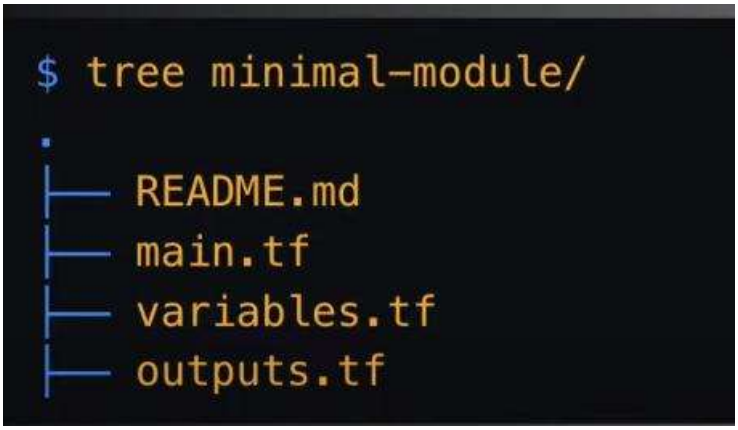
Steps Without Modules

You would have to write the same Terraform code repeatedly for each environment, making it harder to maintain and prone to errors.

Solution With Modules

You can create a **VPC module** with reusable code and call it for each environment with different parameters.

Structure of the minimal module



README.md:
A text file that provides **documentation** for the module.
Explains the module's purpose, usage, required inputs, and outputs.

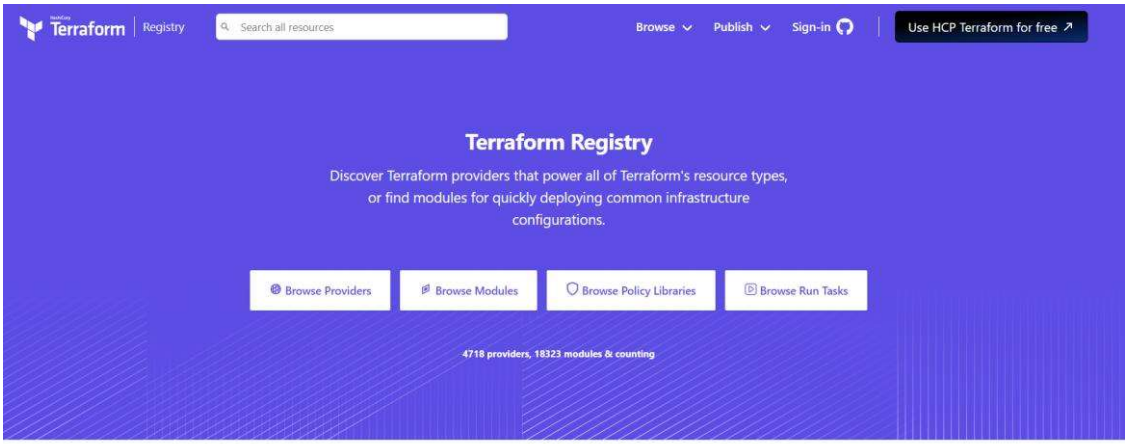
main.tf:
The core Terraform configuration file.
Defines the **resources**, **data sources**, and **logic** for the module.

variables.tf:
Specifies the **inputs** the module needs.
Defines variables with types, descriptions, and default values.

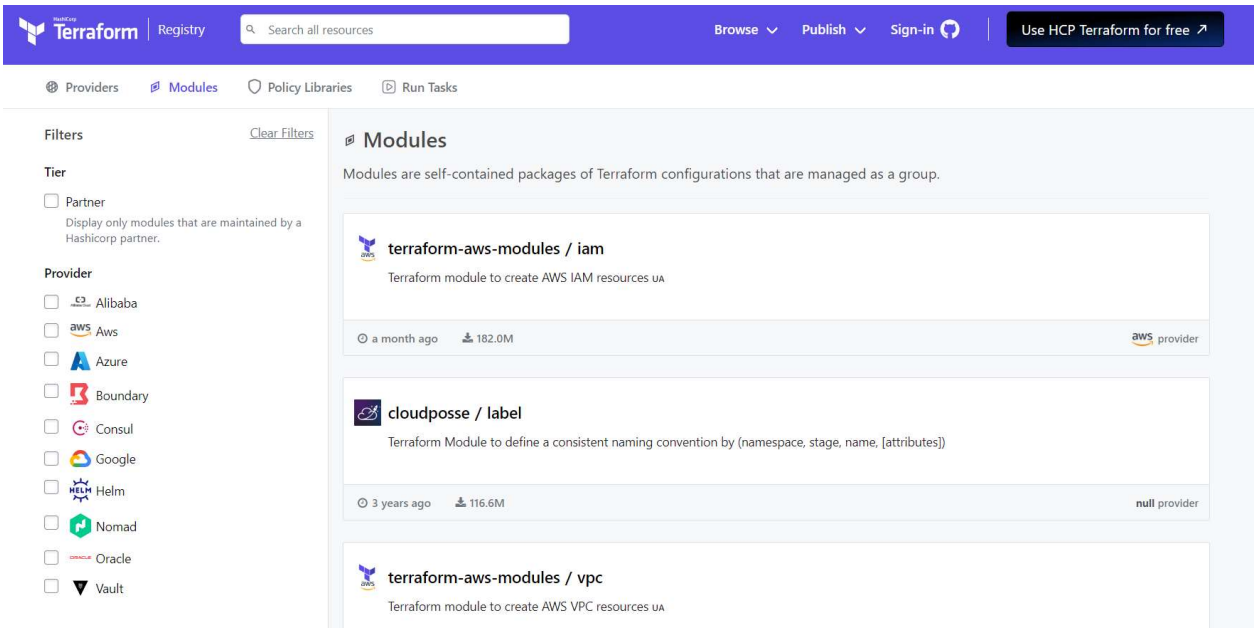
outputs.tf:
Declares the **outputs** from the module.
Shares resource information (like IDs, IPs) with the root module or other modules.

This structure ensures that the module is **self-contained**, reusable, and easy to understand. Let me know if you'd like a detailed explanation for any of these files!

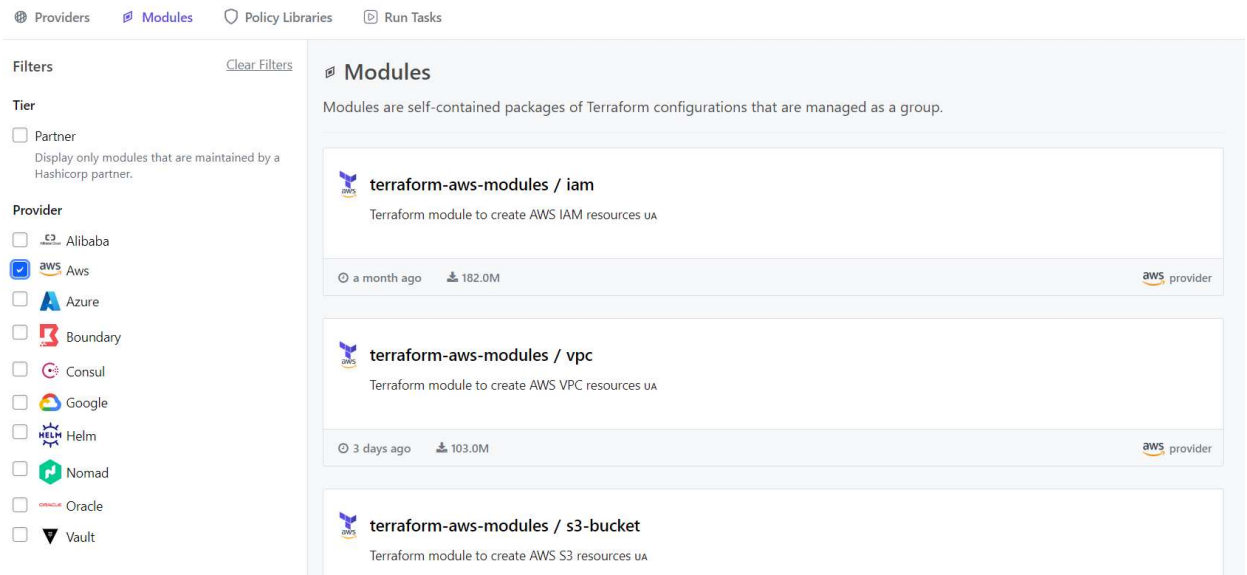
To get the proper modules go to "https://registry.terraform.io" you will see the something like this:



Now click on modules you will get to see something like this:

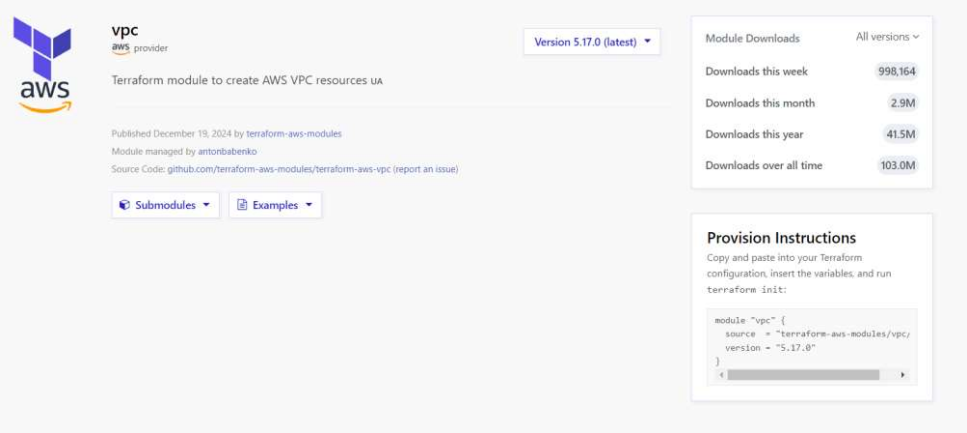


By selecting the AWS provider, you can access a wide range of modules tailored to various AWS services. These modules are organized with user-friendly names, making it easier to identify and choose the module that best fits your requirements.

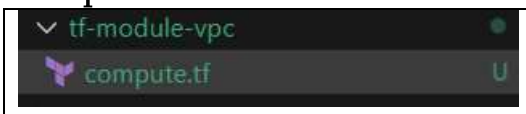


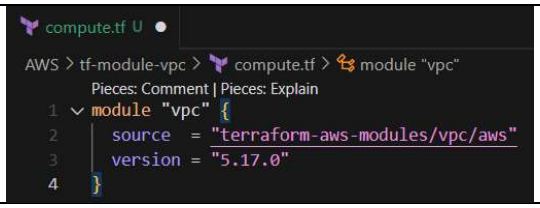
Implementation of VPC using terraform modules

Let's implement a VPC using Terraform modules. We'll create a new directory named "**tf-module-vpc"**. Inside this directory, we'll create a file called **compute.tf**. Next, search for the **AWS VPC module** in the Terraform Registry. You'll find a module that looks something like this:

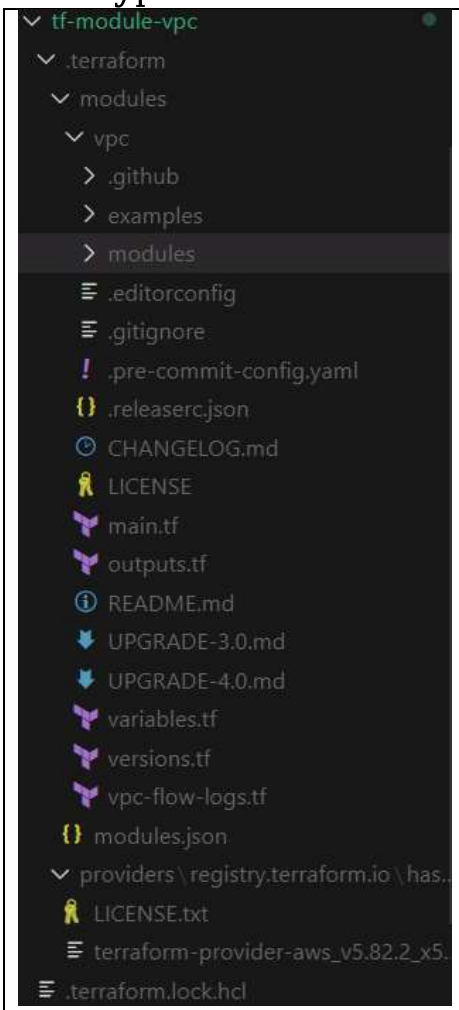


Now copy the instruction that is given in “Provision instructions” in compute.tf



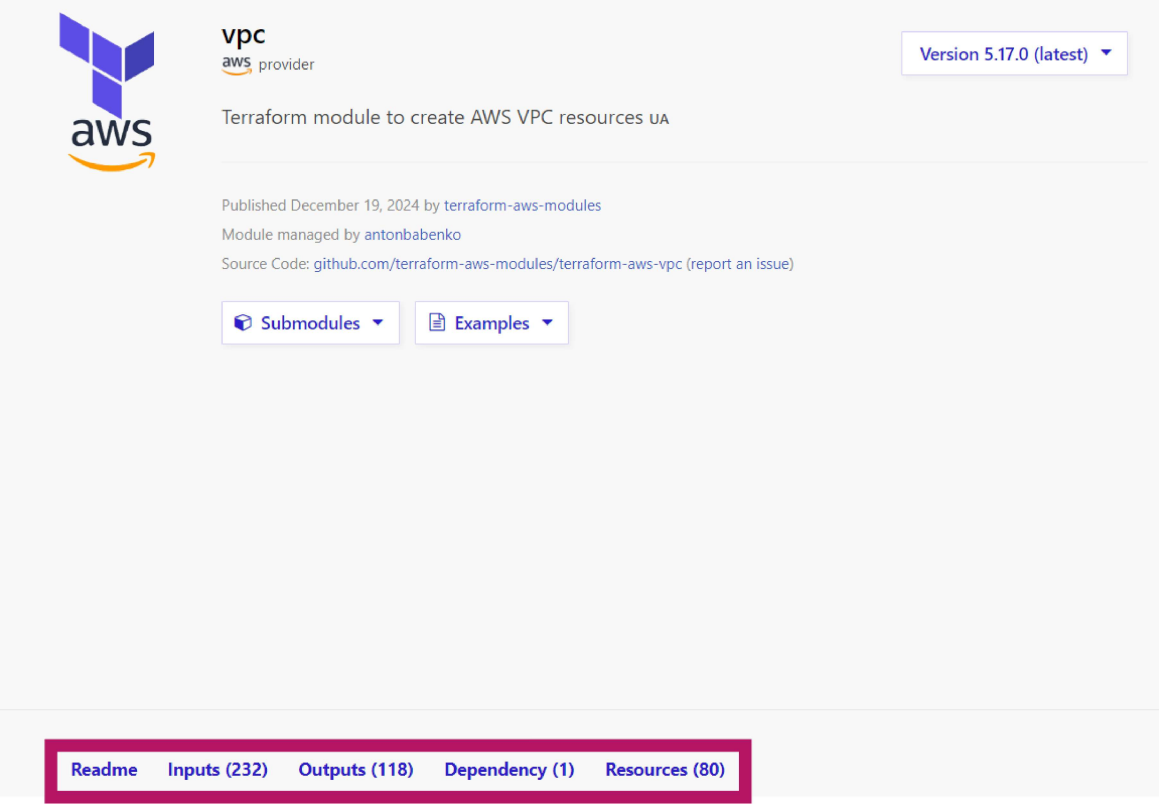


Now type “terraform init” in termial, then you will notice something like:



Go through “main.tf”, “outputs.tf”, “variables.tf”, “versions.tf”, “vpc-flow-logs.tf”

To understand how to use the module effectively, read the documentation thoroughly. Navigate to the required module in the Terraform Registry and scroll down. You'll find key sections like **"Readme"**, **"Inputs"**, **"Outputs"**, **"Dependencies"**, and **"Resources"**. These sections provide detailed information about the module's usage, configuration, required variables, outputs, and dependencies.



n the **"Inputs"** section of the Terraform Registry, you'll see all the variables defined in the module's variables.tf file. These variables automatically appear there for easy reference.

If you want to explore the infrastructure code directly, click on the **"Source Code"** link in the documentation. This will take you to the module's GitHub repository, where you can view all the details related to the AWS VPC provider.

When a module is implemented, it needs to know which AWS region to deploy the infrastructure in. To specify this, you provide the AWS region in the provider block or as a variable in your Terraform configuration.

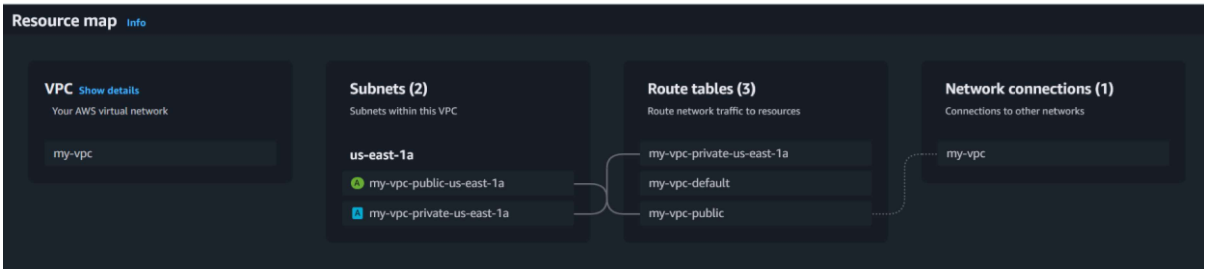
```
provider "aws" {
  region = "us-east-1"
}
Pieces: Comment | Pieces: Explain
module "vpc" {
  source  = "terraform-aws-modules/vpc/aws"
  version = "5.17.0"
}
```

For further configuration do follow readme file of the module

Final configuration of “compute.tf”

```
Pieces: Comment | Pieces: Explain
provider "aws" {
  region = "us-east-1"
}
Pieces: Comment | Pieces: Explain
module "vpc" {
  source  = "terraform-aws-modules/vpc/aws"
  version = "5.17.0"

  name = "my-vpc"
  cidr = "10.0.0.0/16"

  azs             = ["us-east-1a"]
  private_subnets = ["10.0.1.0/24"]
  public_subnets  = ["10.0.101.0/24"]



  tags = {
    Terraform = "true"
    Environment = "dev"
  }
}

```

output



Implementation of EC2 instance using terraform modules

Now, we will create a new file named **instance.tf** inside the **tf-module-vpc** directory.

This file will be used to define EC2 instances.
To find the module for deploying EC2 instances, search for **terraform-aws-modules/ec2-instance** in the Terraform Registry.
Use the same process you followed to find the VPC module:

1. Go to the **Terraform Registry**.

2. Search for **terraform-aws-modules/ec2-instance**.

3. Review the module documentation for **"Readme"**, **"Inputs"**, **"Outputs"**, **"Dependencies"**, and **"Resources"** sections to understand its configuration and usage.

In "***instance.tf***"

```
module "ec2-instance" {
  source  = "terraform-aws-modules/ec2-instance/aws"
  version = "5.7.1"

  name = "single-instance"

  ami = "ami-0453ec754f44f9a4a"

  instance_type          = "t2.micro"
  vpc_security_group_ids = [module.vpc.default_security_group_id]
  subnet_id              = module.vpc.public_subnets[0]

  tags = {
    Name    = "trial-instance"
    Environment = "dev"
  }

}
```

VPC Security Group and Subnet:

- vpc_security_group_ids: Associates the instance with the default security group from the **VPC module** (defined as module.vpc.default_security_group_id).

- subnet_id: Places the EC2 instance in the first public subnet created by the **VPC module** (module.vpc.public_subnets[0]).

Output

NOTE: we are only doing "terraform plan" not "terraform apply" because it may cost you, do it at your own risk.

Building own terraform modules

Building your own Terraform modules involves creating reusable configurations to manage infrastructure more efficiently.
These modules group related resources together, making your code modular, scalable, and easy to maintain.
Custom **modules** are especially useful when standardizing infrastructure across multiple environments (e.g., dev, staging, prod).

A **modules** typically includes:

- **main.tf**: Defines resources.
- **variables.tf**: Specifies input variables.
- **outputs.tf**: Declares output values.

README.md: Helps users understand the module's purpose, how to use it.

Real-Life Scenario:

Scenario: A company needs to deploy VPCs with consistent configurations, so we need this all things in “tf-own-module”

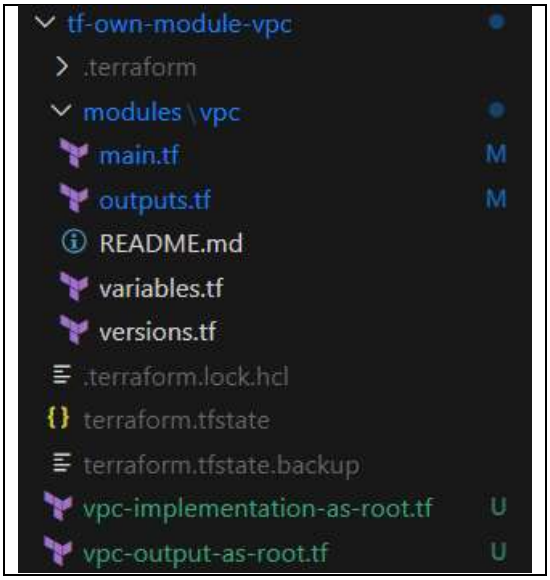
Requirements:

- Accept cidr_block from user to create VPC
- User can create multiple subnets
  → Get CIDR block for subnets from user
  → Get AZS (availability zone)
  → User can mark a subnet as public (default is private)
  If public, create IGW
  Associate public subnet with Routing table

We will create the following VPC configuration by developing a custom Terraform module.

Where all the subnets i.e. public or private is attached to Route table (rtb-001538132f80805c1) and this route table is attached to internet gateway

The file structure will be as follows:



In tf-own-module-vpc / modules / vpc / version.tf

```
terraform {
  required_version = ">= 1.0.0"

  Pieces: Comment | Pieces: Explain
  required_providers {
    object = {
      source  = "hashicorp/aws"
      version = ">= 4.0.0"
    }
  }
}
```

In tf-own-module-vpc / modules / vpc / variables.tf

```
variable "vpc_config" {
  description = "To get the CIDR and the name of the vpc from user"
  type = object({
    cidr_block = string
    vpc_name = string
  })

  Pieces: Comment | Pieces: Explain
  validation {
    condition= can(cidrnetmask(var.vpc_config.cidr_block))
    error_message = "Each cidr_block must be a valid CIDR format."
  }
}
```

```
Pieces: Comment | Pieces: Explain
variable "aws_subnets" {
  # sub1={cidr_block, availability_zone}, sub2={}, sub3={}
  description = "To get the CIDR and availability zone from user"
  type = map(object({
    cidr_block = string
    availability_zone = string
    public = optional(bool, false)
  }))

  Pieces: Comment | Pieces: Explain
  validation {
    #sub1={cidr=...}, sub2={cidr=...}, sub3={cidr=...} , [true, true, true]
    condition= alltrue([for config in var.aws_subnets : can(cidrnetmask(config.cidr_block))])
    error_message = "Each cidr_block must be a valid CIDR format."
  }
}
```

In tf-own-module-vpc/modules/vpc/main.tf

```
resource "aws_vpc" "vpc-main" {
  cidr_block = var.vpc_config.cidr_block

  tags = {
    Name = var.vpc_config.vpc_name
  }
}

Pieces: Comment | Pieces: Explain
resource "aws_subnet" "subnets_main" {
  vpc_id = aws_vpc.vpc-main.id
  for_each = var.aws_subnets

  cidr_block = each.value.cidr_block
  availability_zone = each.value.availability_zone

  tags = {
    Name = each.key
  }
}

Pieces: Comment | Pieces: Explain
locals {
  public_subnets = {
    #public_sub1 = {cidr_block = "10.0.1.0/24", availability_zone = "us-east-1a", public = true}
    for key, config  in var.aws_subnets : key => config if config.public
  }

  private_subnets = {
    for key, config  in var.aws_subnets : key => config if !config.public
  }
}
```

```
31
32  #internet gateway, if there is atleast one public subnet
33  #internet gateway will be created only once
    Pieces: Comment | Pieces: Explain
34  resource "aws_internet_gateway" "igw_main" {
35    vpc_id =  aws_vpc.vpc-main.id
36    count = length(local.public_subnets) > 0 ? 1 : 0
37  }
    Pieces: Comment | Pieces: Explain
39  resource "aws_route_table" "route_main" {
40    vpc_id = aws_vpc.vpc-main.id
41    count = length(local.public_subnets) > 0 ? 1 : 0
42
      Pieces: Comment | Pieces: Explain
43    route {
44      cidr_block = "0.0.0.0/0"
45      gateway_id = aws_internet_gateway.igw_main[0].id
46    }
47  }
49  resource "aws_route_table_association" "route_table_association_main" {
50    for_each = local.public_subnets
51    subnet_id = aws_subnet.subnets_main[each.key].id
52    route_table_id = aws_route_table.route_main[0].id
53  }
54
```

In tf-own-module-vpc/modules/vpc/output.tf

```
1   output "vpc_id" {
2     description = "diplay the vpc id"
3     value = aws_vpc.vpc-main.id
4   }
5
6   locals {
7   #output format of subnets ids is subnet_name = {subnet_id, availability_zone}
8    public_subnet_output={
9        for key , config in local.public_subnets : key => {
10         subnet_id = aws_subnet.subnets_main[key].id
11         availability_zone = aws_subnet.subnets_main[key].availability_zone
12       }
13   }
14
15   private_subnet_output={
16       for key , config in local.public_subnets : key => {
17         subnet_id = aws_subnet.subnets_main[key].id
18         availability_zone = aws_subnet.subnets_main[key].availability_zone
19       }
20   }
21  }
```

```
output "public-subnet-ids" {
  description = "list of public subnet ids"
  value = local.public_subnet_output
}

output "private-subnet-ids" {
  description = "list of private subnet ids"
  value = local.private_subnet_output
}
```

In tf-own-module-vpc / vpc-implementation-as-root.tf

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

module "vpc" {
  source = "./modules/vpc"

  vpc_config = {
    cidr_block = "10.0.0.0/16"
    vpc_name = "implementing-vpc-using-own-module"
  }

  aws_subnets = {
    public_sub1 = {
      cidr_block = "10.0.1.0/24"
      availability_zone = "us-east-1a"
      public = true
    }
    public_sub2 = {
      cidr_block = "10.0.2.0/24"
      availability_zone = "us-east-1b"
      public = true
    }
    private_sub3 = {
      cidr_block = "10.0.3.0/24"
      availability_zone = "us-east-1c"
      public = false
    }
  }
}
```

In tf-own-module-vpc / vpc-output-as-root.tf

```
output "vpc_id" {
  value =module.vpc.vpc_id
}
```

```
output "public-subnet-ids" {
  value = module.vpc.public-subnet-ids
}

output "private-subnet-ids" {
  value = module.vpc.private-subnet-ids
}
```

When you do *"terraform init"→ "terraform plan" → "terraform apply"*
output

```
Apply complete! Resources: 8 added, 0 changed, 0 destroyed.

Outputs:

private-subnet-ids = {
  "public_sub1" = {
    "availability_zone" = "us-east-1a"
    "subnet_id" = "subnet-0ad6830dd993231b7"
  }
  "public_sub2" = {
    "availability_zone" = "us-east-1b"
    "subnet_id" = "subnet-014f0a99399cc0c94"
  }
}
public-subnet-ids = {
  "public_sub1" = {
    "availability_zone" = "us-east-1a"
    "subnet_id" = "subnet-0ad6830dd993231b7"
  }
  "public_sub2" = {
    "availability_zone" = "us-east-1b"
    "subnet_id" = "subnet-014f0a99399cc0c94"
  }
}
vpc_id = "vpc-0dfffdeb4b1b7e13c"
```

NOTE: we are only doing "terraform plan" not "terraform apply" because it may cost you, do it at your own risk.

31. PREPARE MODULE FOR PUBLISH

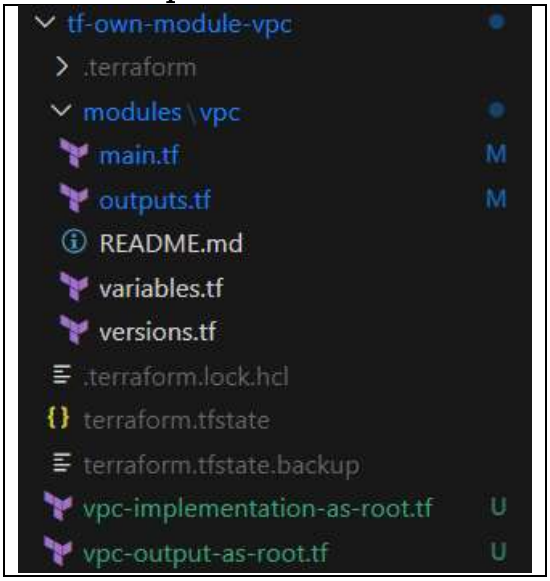
Creating your own Terraform module simplifies infrastructure management by enabling reusability, consistency, and scalability.

It allows you to define resources once and reuse them across projects, ensuring uniform configurations, reducing duplication, and making collaboration easier.

- Readme.md file
- LICENSE
- Example
- Push the code in GitHub
- Terraform Registry

To make this section successful we will take “tf-own-module” and add some extra files and directory.

This is a previous folder structure



In ***In tf-own-module-vpc/modules/vpc/ README.md***

````
#terraform-aws-vpc
| Ctrl+L to chat, Ctrl+K to generate
## Overview

This Terraform module creates an AWS VPC with a given CIDR block.
It also creates multiple subnets (public and private), and for
public subnets, it sets up an Internet Gateway (IGW)
and appropriate route tables.

## Features

- Creates a VPC with a specified CIDR block
- Creates public and private subnets
- Creates an Internet Gateway (IGW) for public subnets
- Sets up route tables for public subnets
## Usage

```
 module "vpc" {
   source = "./modules/vpc"

   vpc_config = {
     cidr_block = "10.0.0.0/16"
     vpc_name = "test_vpc"
   }

   aws_subnets = {
     public_sub1 = {
       cidr_block = "10.0.1.0/24"
       availability_zone = "us-east-1a"
       # To set the subnet as public, default is private
       public = true
     }
     public_sub2 = {
       cidr_block = "10.0.2.0/24"
       availability_zone = "us-east-1b"
       public = true
     }
     private_sub3 = {
       cidr_block = "10.0.3.0/24"
       availability_zone = "us-east-1c"
       public = false
     }
   }
 }
 ```
````

The code provided in the "Usage" section is extracted from ***tf-own-module-vpc/vpc-implementation-as-root.tf,*** excluding the provider section.

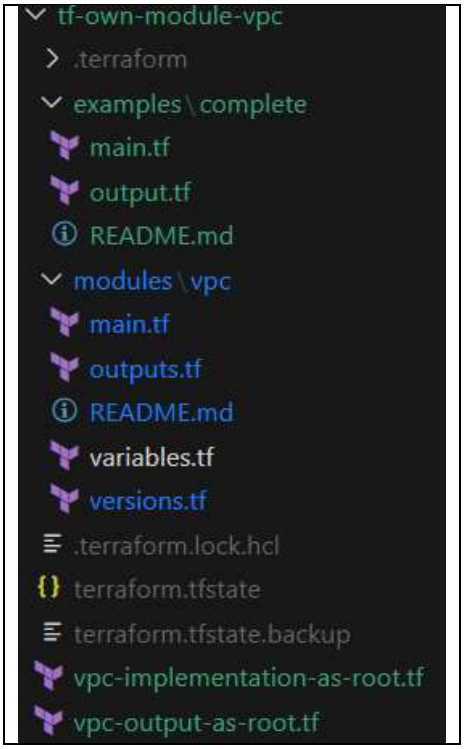
Add the necessary LICENSE file, which you can generate while creating the repository on GitHub.

Now, create a new folder named examples inside the tf-own-module-vpc directory. Within the examples folder, create another subfolder named complete. Inside the complete folder, replicate all the root-level files, such as README.md, main.tf and output.tf.

Copy the contents of ***tf-own-module-vpc/vpc-implementation-as-root.tf*** to ***tf-own-module-vpc/examples/complete/main.tf***

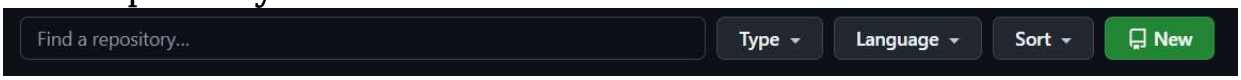
Copy the contents of ***tf-own-module-vpc/ vpc-output-as-root.tf*** to ***tf-own-module-vpc/examples/complete/outputs.tf***

Now this is the final folder structure:

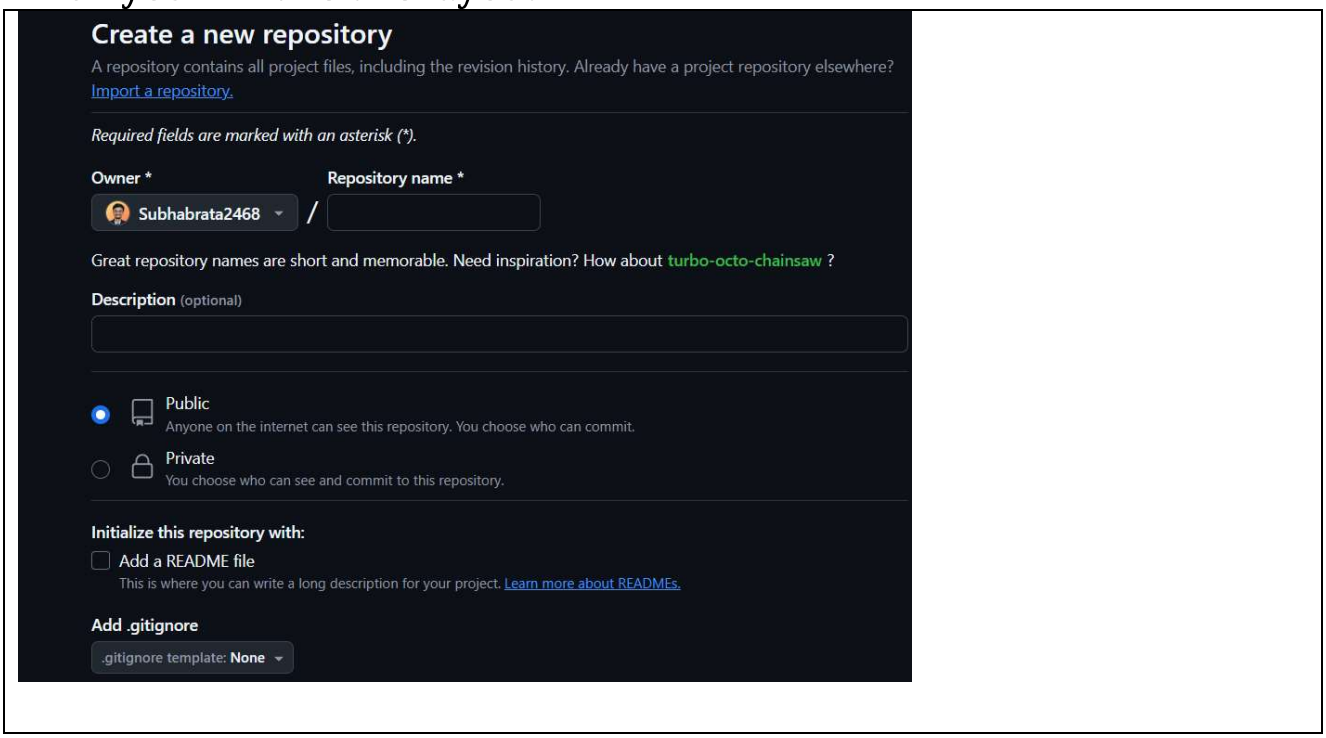


Create a new folder named GitHub-Code-Publisher and copy only the modules and examples directories from tf-own-module-vpc into it.

Log in to your GitHub account, click on the **New** button, and create a new repository.

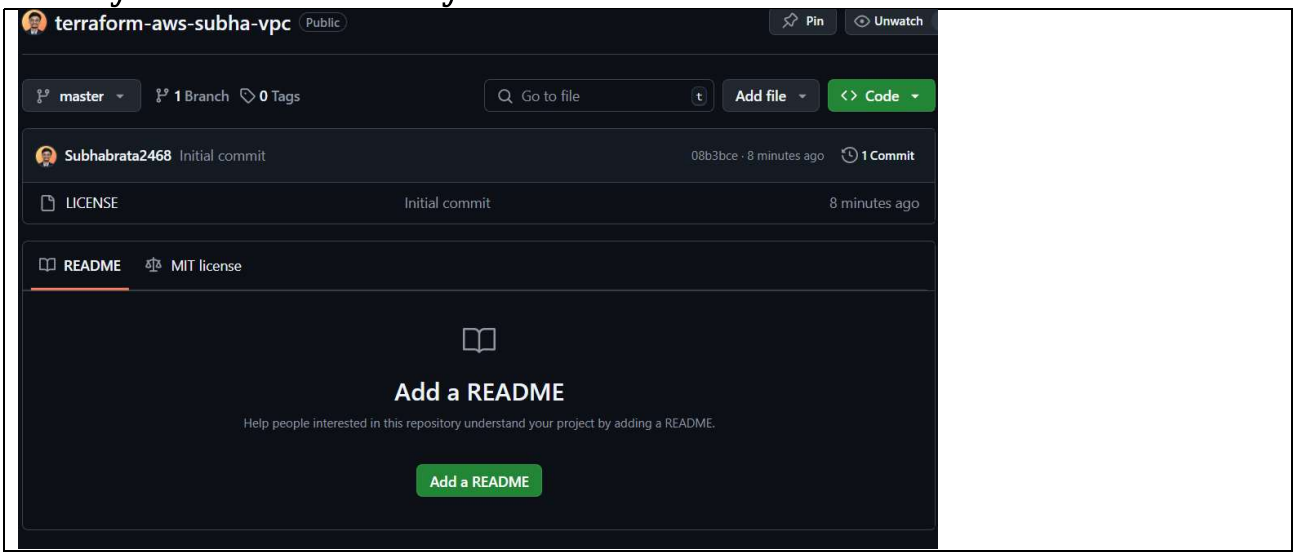


Then you will this this layout



Give the repo name in proper way something like “terraform-aws-subha-vpc” and give the MIT license then click on “create repository”

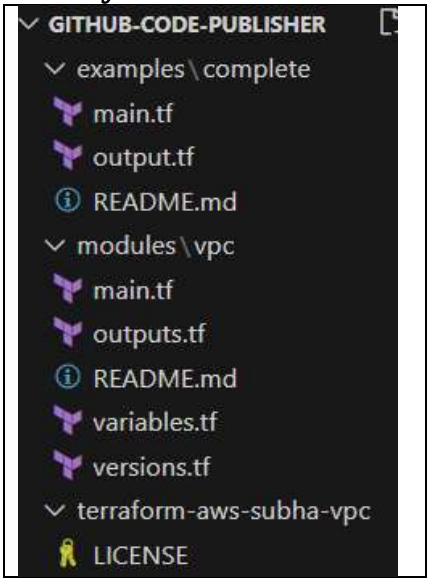
Then you will see this layout



Then click on code and copy the like on the vs code terminal by adding git clone before it

```
PS D:\Downloads\GitHub-Code-Publisher> git clone https://github.com/Subhabrata2468/terraform-aws-su
bha-vpc.git
Cloning into 'terraform-aws-subha-vpc'...
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
Receiving objects: 100% (3/3), done.
PS D:\Downloads\GitHub-Code-Publisher>
```

Now you will see this file structure

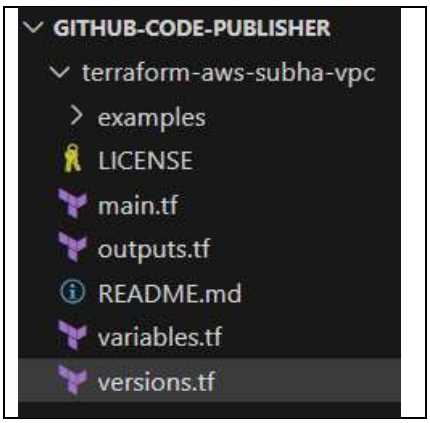


Move all files from the modules/vpc folder (e.g., main.tf, outputs.tf, variables.tf, versions.tf) to the root level of the repository, just outside the modules/vpc directory.

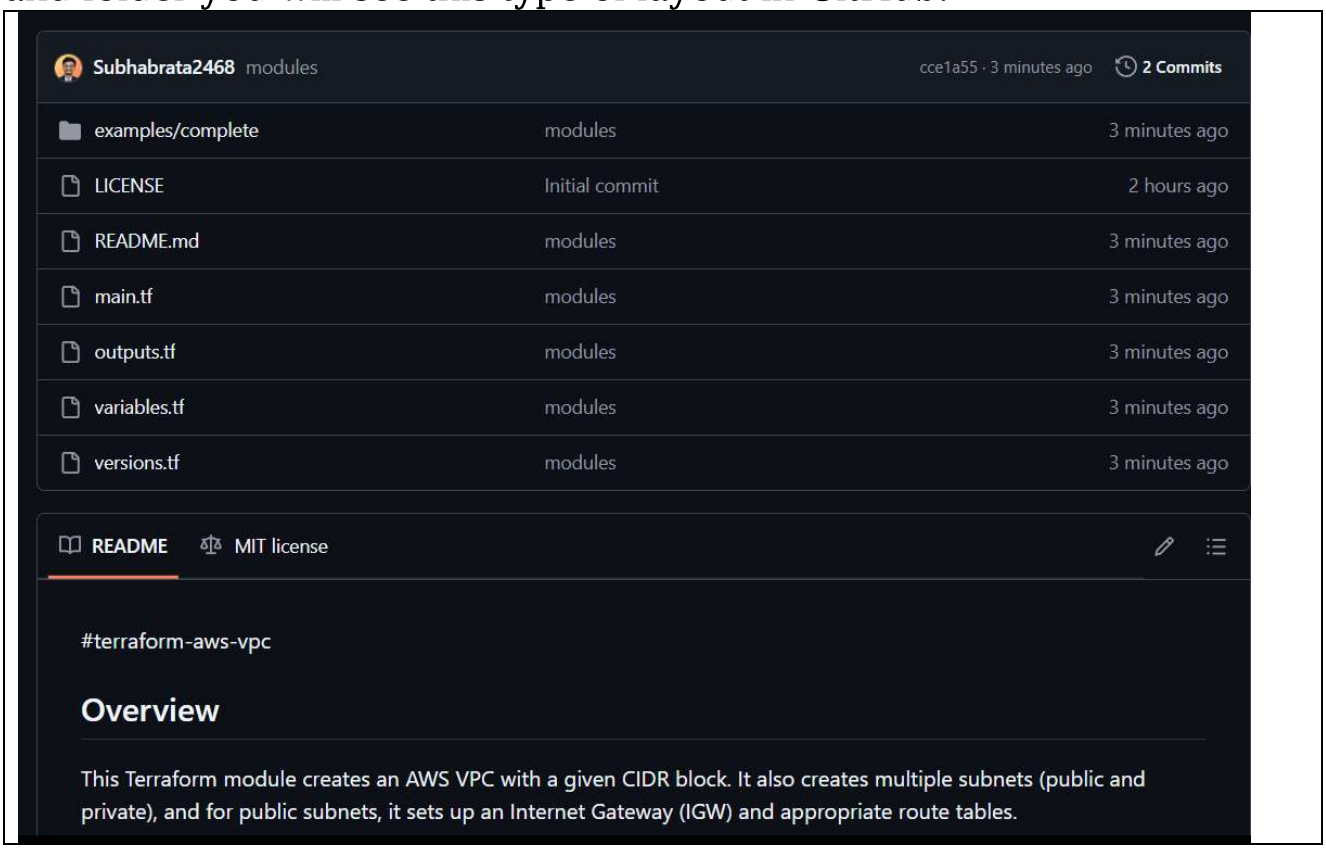
Delete the now-empty modules/vpc folder.

Copy both the files and the examples folder into the terraform-aws-subha-vpc directory.

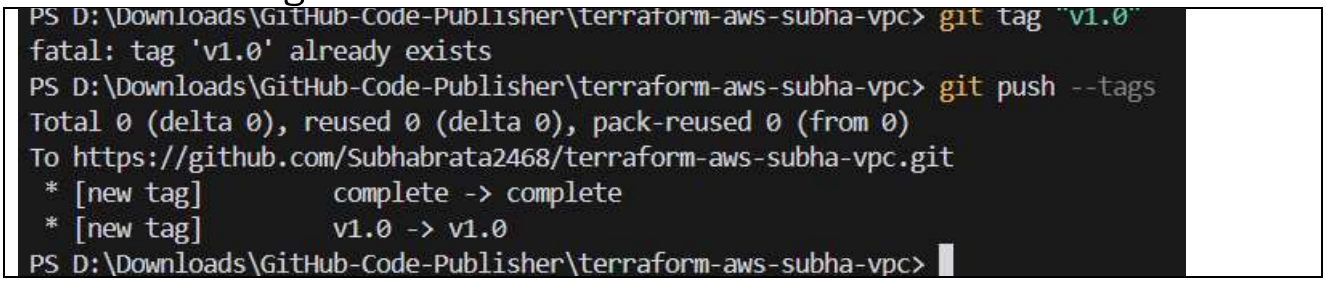
This is the new file structure after editing



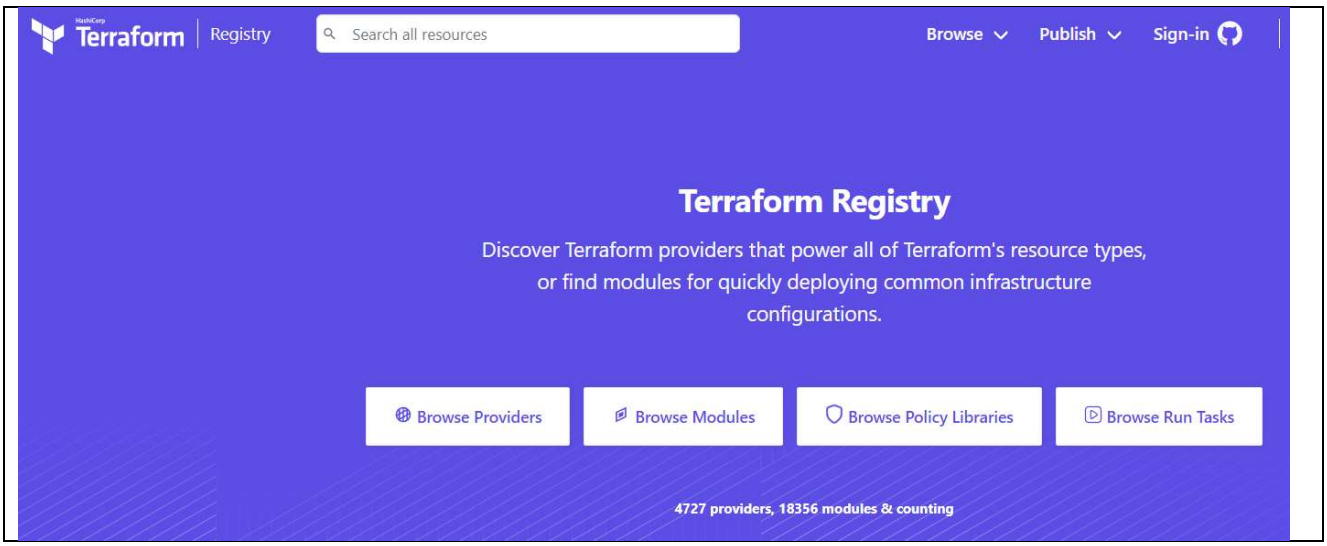
Now add the files and folder to GitHub. After you have added the files and folder you will see this type of layout in GitHub.



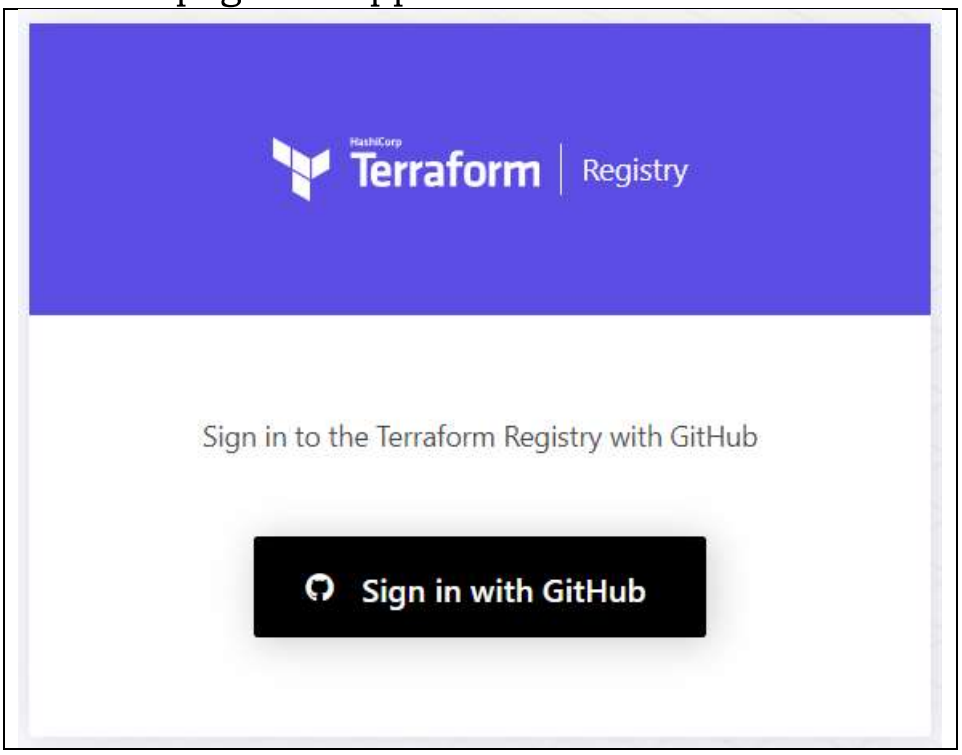
Now add the tag



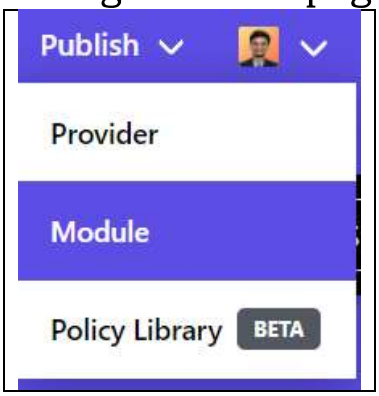
Now go to terraform registry click on sign in option.



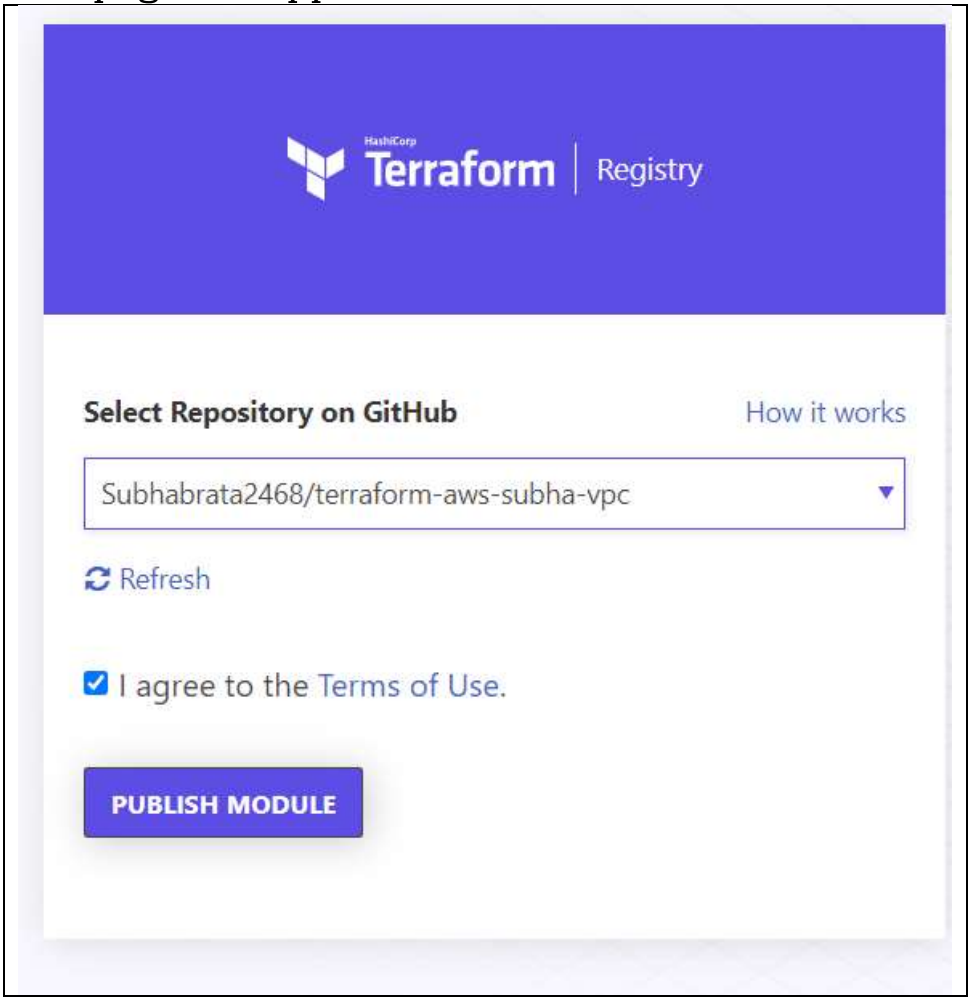
Now this page will appear



Now go to main page, select “publish” and click on module



This page will appear



Agree to terms and conditions, then “PUBLISH MODULE”

If all the above steps are correct, then you will see this output in terraform module

If you want to use this modules as configuration then please follow “***Implementation of EC2 instance using terraform modules***”

32. TERRAFORM DEPENDENCY

Terraform dependencies ensure resources are created, modified, or destroyed in the correct order by defining relationships between them. Dependencies help manage resource interconnections, avoiding errors during provisioning or destruction.

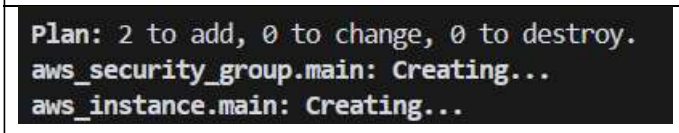
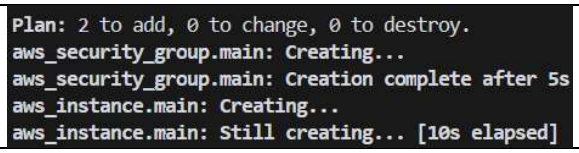
Key Uses:

1. **Automatic Ordering:** Terraform understands which resources need to exist before others.
    - Example: A subnet must be created before launching an EC2 instance in it.

2. **Error Prevention:** Ensures dependent resources are not destroyed or modified prematurely.
    - Example: Deleting an S3 bucket ensures objects within it are removed first.

3. **Custom Dependencies:** Explicitly define dependencies using depends_on when Terraform cannot infer relationships.

We will now create a new folder called **tf-dependencies**. Inside this folder name "**AWS**", we will add a file named **main.tf**, which will serve as the entry point for defining the required Terraform configurations related to module dependencies.

Before using “depends_on” in aws_instance resources	After using “depends_on” in aws_instance resources
	
Output: → after Appling terraform init→ terraform plan →terrafrom apply	Output: → after Appling terraform init→ terraform plan →terrafrom apply
	
In this both “aws_security_group.main” and “aws_instance.main” are being created simuntaneously	In this first it will create “aws_security_group.main” then it will create “aws_instance.main”

33. RESOURCES LIFECYCLE

The **lifecycle block** in Terraform is used to manage specific behaviors of resources, such as preventing their accidental deletion, ignoring certain changes, or customizing resource creation and destruction processes.

Key Arguments in the **lifecycle** block

create_before_destroy

Ensures that when a resource is replaced (due to changes in configuration), the new resource is created **before** the old one is destroyed.

Useful for resources that cannot have downtime, such as load balancers or production systems.

```
resource "aws_instance" "example" {
  ami           = "ami-123456"
  instance_type = "t2.micro"

  lifecycle {
    create_before_destroy = true
  }
}
```

Aspect	*create_before_destroy = true*	*create_before_destroy = false*
Order of Actions	Create first, destroy later	Destroy first, create later
Downtime	No downtime	Downtime possible
Resource Dependency	May require sufficient resources (e.g., IPs, capacity) for two resources to coexist temporarily.	Fewer resource requirements.
Usage	Production-critical resources	Non-critical resources where downtime is acceptable.

Things to Consider

- **Cost:** Creating a new resource before destroying the old one might temporarily increase costs (e.g., running two EC2 instances simultaneously).

- **Resource Limits:** Some AWS services have quotas (e.g., a limited number of EC2 instances or IP addresses per region). Ensure your account can support two instances temporarily.

- **Dependencies:** If other resources depend on the old instance, they might need to be updated after the replacement process.

prevent_destroy

Prevents a resource from being destroyed, even if a terraform destroy command is run or the resource is removed from the configuration.

Useful for critical resources like databases or production VPCs.

```
resource "aws_s3_bucket" "example" {
  bucket = "my-important-bucket"

  lifecycle {
    prevent_destroy = true
  }
}
```

If you try to destroy the resource, Terraform will throw an error unless you explicitly disable this protection.

Aspect	*prevent_destroy = true*	*prevent_destroy = false*
Behavior on Destroy	Prevents resource from being destroyed	Allows resource to be destroyed
Error on Destroy	Terraform fails with an error	Terraform proceeds without error
Use Case	Critical resources (databases, VPCs)	Non-critical or temporary resources
Override Requirement	Must explicitly disable to destroy	No override needed

Things to Consider

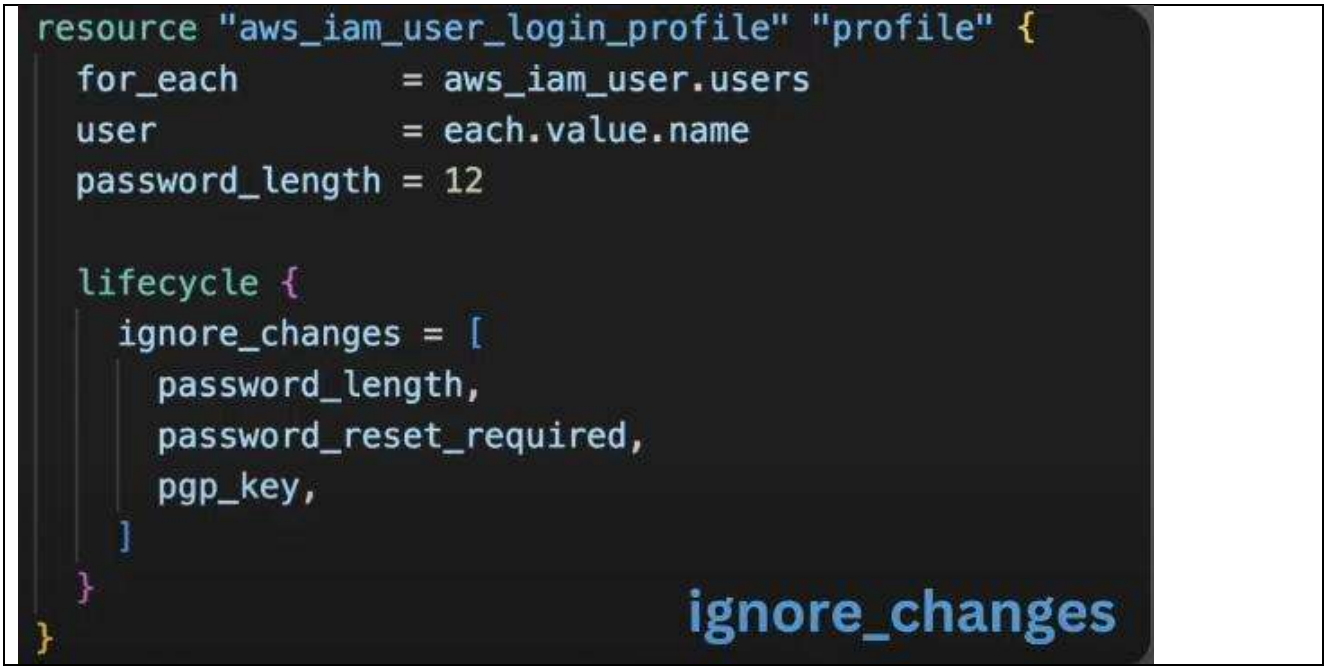
1. **Critical Resources:** Use prevent_destroy for critical resources like:
    - Databases (to prevent data loss).
    - VPCs or subnets (to avoid breaking the network).
    - Persistent storage like S3 buckets or EBS volumes.

2. **Testing Environments:** Avoid using prevent_destroy in non-critical environments where resources are often created and destroyed (e.g., development or testing).

3. **Force Destroy:** To override prevent_destroy, you can temporarily remove the argument, apply the changes, and then re-add it.

ignore_changes

Tells Terraform to ignore updates to specific attributes of a resource, even if they are changed outside of Terraform (e.g., manually or by another process).



Aspect	*With ignore_changes*	*Without ignore_changes*
Manual Changes	Ignored for specified attributes	Overwritten by Terraform
Configuration Updates	Ignored for specified attributes	Applied as part of the next plan/apply
Use Case	When certain attributes are managed externally	When Terraform should fully control the resource

Things to Consider

1. **Dynamic Values:** Use ignore_changes for attributes that are frequently updated dynamically by external systems (e.g., tags, user_data, or IAM policies).
2. **Critical Updates:** Avoid using ignore_changes for critical attributes (e.g., instance_type, cidr_block) to ensure Terraform manages them properly.
3. **Fine-Tuning:** You can ignore specific attributes rather than ignoring all changes, making it a granular control mechanism.

replace_triggered_by

The **replace_triggered_by** argument in the Terraform **lifecycle** block is used to trigger the replacement of a resource when changes occur to specified dependencies or attributes outside of its direct configuration.

How It Works:

- **Purpose:** It defines specific dependencies that, when changed, will cause the resource to be destroyed and recreated.
- **Use Case:** This is useful when a resource must be replaced if a related resource or attribute changes, even if the resource itself hasn't directly changed in the configuration.

```
resource "aws_instance" "example" {
  ami           = "ami-123456"
  instance_type = "t2.micro"

  lifecycle {
    replace_triggered_by = [
      aws_ami.my_custom_ami.id, # Dependency on a specific resource
      var.instance_type       # Dependency on a variable
    ]
  }
}
```

34. PRE & POST CONDITION – RESOURCE VALIDATIONS

In this topic we will cover

- preconditions
- postconditions

Allows to define checks that must be true before a resource is created (preconditions) and after a resource is created (postcondition).

We will be using “preconditions” and “postcondtions” in lifecycle{…} (block)

Feature	***Preconditions***	***Postconditions***
Validation Time	Before resource creation/update/destruction	After resource creation/update/destruction
Purpose	Ensure input/configuration validity	Validate resource properties or outcomes
Error Message	Displays if the condition evaluates to false	Displays if the condition evaluates to false
Use Cases	AMI ID format validation, region checks, variable checks	Instance state, S3 bucket policy validation, output checks

Syntax

preconditions

```
resource "aws_instance" "example" {
  ami           = "ami-123456"
  instance_type = "t2.micro"

  lifecycle {
    precondition {
      condition     = <expression>
      error_message = "Description of the error if the condition fails."
    }
  }
}
```

postconditions

```
resource "aws_instance" "example" {
  ami           = "ami-123456"
  instance_type = "t2.micro"

  lifecycle {
    postcondition {
      condition     = <expression>
      error_message = "Description of the error if the condition fails."
    }
  }
}
```

Examples

preconditions

Ensure the selected AMI starts with ami-:

```
resource "aws_instance" "example" {
  ami           = var.ami_id
  instance_type = "t2.micro"

  lifecycle {
    precondition {
      condition     = startswith(var.ami_id, "ami-")
      error_message = "AMI ID must start with 'ami-'."
    }
  }
}
```

postconditions

Ensure the EC2 instance has the desired state of "running":

```
resource "aws_instance" "example" {
  ami           = var.ami_id
  instance_type = "t2.micro"

  lifecycle {
    postcondition {
      condition     = aws_instance.example.state == "running"
      error_message = "The EC2 instance must be in a running state."
    }
  }
}
```

Combined Example (Preconditions and Postconditions)

```
resource "aws_instance" "example" {
  ami           = var.ami_id
  instance_type = var.instance_type

  lifecycle {
    precondition {
      condition     = startswith(var.ami_id, "ami-")
      error_message = "AMI ID must start with 'ami-'."
    }

    postcondition {
      condition     = aws_instance.example.state == "running"
      error_message = "The EC2 instance must be in a running state."
    }
  }
}
```

35. TERRAFORM STATE MODIFICATION

Terraform state modification is necessary to manage changes in your infrastructure that are not automatically reflected in the state file. Common use cases include:

- **Drift Management:** When manual changes are made to resources, and Terraform's state file no longer matches the actual infrastructure.
- **Refactoring:** Moving resources to a new module or address without destroying and recreating them.
- **Removing Orphaned Resources:** Deleting resources from state without impacting actual infrastructure.
- **Backend Migration:** Updating the backend configuration or moving state files to a new backend.

List all resources in the state

```
terraform state list
```

Purpose: Displays a list of all resources currently managed by Terraform in the state file.

Usage: Useful for auditing the state file or verifying that resources are being tracked.

Example Output:

Copy code

```
aws_instance.example
aws_s3_bucket.example
```

Show details of a specific resource

```
terraform state show <resource_address>
```

Purpose: Provides detailed information about a specific resource from the state file, such as its attributes and metadata.

Usage: Helps in debugging or reviewing the current state of a particular resource.

Example: To view details of an EC2 instance:

bash

```
terraform state show aws_instance.example
```

Move a resource to a different address

```
terraform state mv <source_address> <destination_address>
```

Purpose: Changes the address of a resource in the state file. This is needed when resource names are updated in the code without recreating them.

Usage: Prevents the destruction and recreation of resources when refactoring Terraform configurations.

Example: Rename an S3 bucket resource:

```
terraform state mv aws_s3_bucket.old_name aws_s3_bucket.new_name
```

Remove a resource from the state

```
terraform state rm <resource_address>
```

Purpose: Deletes a resource from the state file without affecting the actual infrastructure.

Usage: Useful for removing resources that are no longer managed by Terraform or were manually created outside of Terraform.

Example: Remove a DynamoDB table:

bash

```
terraform state rm aws_dynamodb_table.example
```

Pull the current state

```
terraform state pull
```

Purpose: Downloads the latest Terraform state from the remote backend to view or modify locally.

Usage: Helps in analyzing or troubleshooting the current state file.

Example:

```
bash

terraform state pull > state.json
```

Push a local state file to the remote backend

`terraform state push <state_file>`

Purpose: Updates the remote backend with a local state file, ensuring consistency across environments.

Usage: Use this cautiously to avoid overwriting valid state data. Typically used for recovery or migration.

Example:

```
bash

terraform state push terraform.tfstate
```

List all state commands

`terraform state`

Purpose: Displays all available state management commands in Terraform.

Usage: Reference this to explore state-related operations and their options.

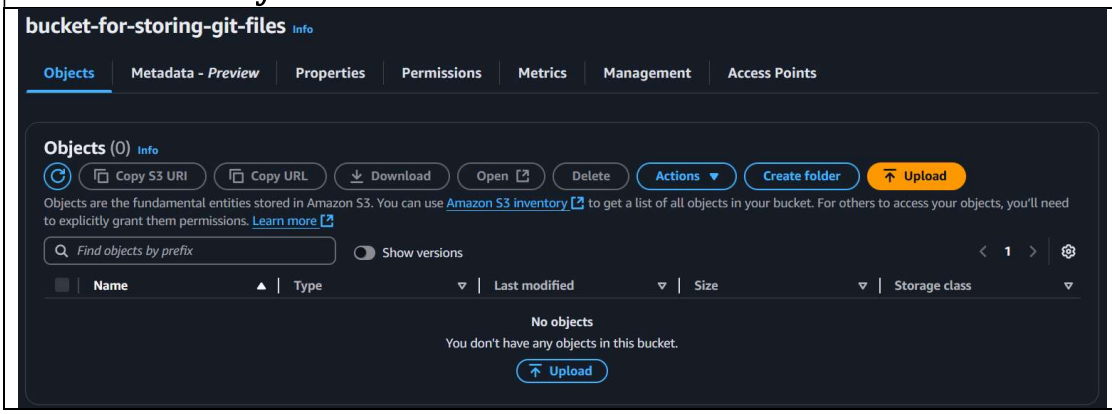
36. TERRAFORM IMPORT COMMAND

Terraform import is a command in terraform that allows you to import existing infrastructure resources into your Terraform state.

Real-Life Scenario:

You've manually created resources in AWS, such as an **S3 bucket** or an **EC2 instance**, but now you want Terraform to manage those resources without recreating them. Terraform's import command allows you to bring these existing resources into the Terraform state.

I have already created S3 bucket in mu AWS console



We will now create a new folder called **tf-import-s3**. Inside this folder name "**AWS**", we will add a file named **main.tf**, which will serve as the entry point for defining the required Terraform configurations related to module dependencies.

In "main.tf"

```
terraform {
  Pieces: Comment | Pieces: Explain
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "5.82.2"
    }
  }
}

Pieces: Comment | Pieces: Explain
provider "aws" {
  # Configuration options
  region = "us-east-1"
}

Pieces: Comment | Pieces: Explain
resource "aws_s3_bucket" "s3_bucket" {
}
```

Use the terraform import command to import the existing bucket into the Terraform state.

```
AWS\tf-import-s3> terraform import aws_s3_bucket.s3_bucket bucket-for-storing-git-files
```

If you see this output that means, it's successfully imported

```
aws_s3_bucket.s3_bucket: Importing from ID "bucket-for-storing-git-files"...
aws_s3_bucket.s3_bucket: Import prepared!
  Prepared aws_s3_bucket for import
aws_s3_bucket.s3_bucket: Refreshing state... [id=bucket-for-storing-git-files]

Import successful!

The resources that were imported are shown above. These resources are now in
your Terraform state and will henceforth be managed by Terraform.
```

Now type "terraform state list"

```
PS D:\Downloads\TERRAFORM\AWS\tf-import-s3> terraform state list
aws_s3_bucket.s3_bucket
```

Now type "terraform state show aws_s3_bucket.s3_bucket" to get all the information of that particular resources

```
PS D:\Downloads\TERRAFORM\AWS\tf-import-s3> terraform state show aws_s3_bucket.s3_bucket
# aws_s3_bucket.s3_bucket:
resource "aws_s3_bucket" "s3_bucket" {
    acceleration_status         = null
    arn                         = "arn:aws:s3:::bucket-for-storing-git-files"
    bucket                      = "bucket-for-storing-git-files"
    bucket_domain_name          = "bucket-for-storing-git-files.s3.amazonaws.com"
    bucket_prefix               = null
    bucket_regional_domain_name = "bucket-for-storing-git-files.s3.us-east-1.amazonaws.com"
    hosted_zone_id              = "Z3AQBSTGFYJSTF"
    id                          = "bucket-for-storing-git-files"
    object_lock_enabled         = false
    policy                      = null
    region                      = "us-east-1"
```

Now update the config file of resource "aws_s3_bucket"

```
resource "aws_s3_bucket" "s3_bucket" {
    bucket = "bucket-for-storing-git-files"
}
```

Now type ***"terraform apply"***

```
PS D:\Downloads\TERRAFORM\AWS\tf-import-s3> terraform apply
aws_s3_bucket.s3_bucket: Refreshing state... [id=bucket-for-storing-git-files]

No changes. Your infrastructure matches the configuration.
```

37. TERRAFORM WORKSPACE

A **Terraform Workspace** is an isolated environment within a single Terraform configuration, allowing you to manage multiple instances of the same infrastructure without duplicating code.
Each workspace has its own **state file**, which keeps track of the resources managed by Terraform in that specific workspace.

Uses of Terraform Workspaces:

1. **Environment Isolation:**
   Use workspaces to manage different environments (e.g., dev, staging, prod) with the same configuration.

2. **Multi-Tenant Applications:**
   Manage separate infrastructure for different clients or tenants.

3. **Avoid State File Conflicts:**
   Keep state files isolated for better organization and avoid conflicts when managing resources.

How Workspaces Work:

1. The default workspace is called default.
2. You can create, switch, and manage workspaces using Terraform commands.
3. Each workspace has its own state file stored in the backend.

Key Commands:
List all workspaces: `terraform workspace list`
Create a new workspace: `terraform workspace new <workspace_name>`

Switch to a workspace:

```
terraform workspace select <workspace_name>
```

Delete a workspace (if empty):

```
terraform workspace delete <workspace_name>
```

Diagram: Terraform Workspaces Example

Here's a simple diagram explaining workspaces:

Terraform Workspaces for Environment Isolation:

```
Terraform Configuration (main.tf)
        |
        |------------------ Workspaces ------------------
        |                   |                   |
    dev (state file)    staging (state file)    prod (state file)
        |                   |                   |
    AWS Resources     AWS Resources       AWS Resources
```

- **dev:** Contains resources for development (e.g., small EC2 instances).
- **staging:** Contains resources for testing.
- **prod:** Contains production-grade resources.

Each workspace manages its own **state file** and resources independently, ensuring no overlap.

Command Flow Example:

```
terraform workspace new dev
terraform apply -var-file=dev.tfvars

terraform workspace new staging
terraform apply -var-file=staging.tfvars

terraform workspace new prod
terraform apply -var-file=prod.tfvars
```

For this demo we will be using the previous terraform folder named “tf-import-s3”

```
PS D:\Downloads\TERRAFORM\AWS\tf-import-s3> terraform workspace list
* default
```

```
PS D:\Downloads\TERRAFORM\AWS\tf-import-s3> 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:\Downloads\TERRAFORM\AWS\tf-import-s3>
```

When you apply a Terraform configuration across multiple workspaces, creating resources with the same name (like an S3 bucket) in each workspace will cause an error, as most resources (e.g., AWS S3 buckets) require unique names globally.

To resolve this issue, you can use the terraform.workspace interpolation to include the workspace name in the resource's name, ensuring it is unique for each environment.

Something like this

```
resource "aws_s3_bucket" "example" {
  bucket = "my-app-bucket-${terraform.workspace}"
  acl    = "private"
}
```

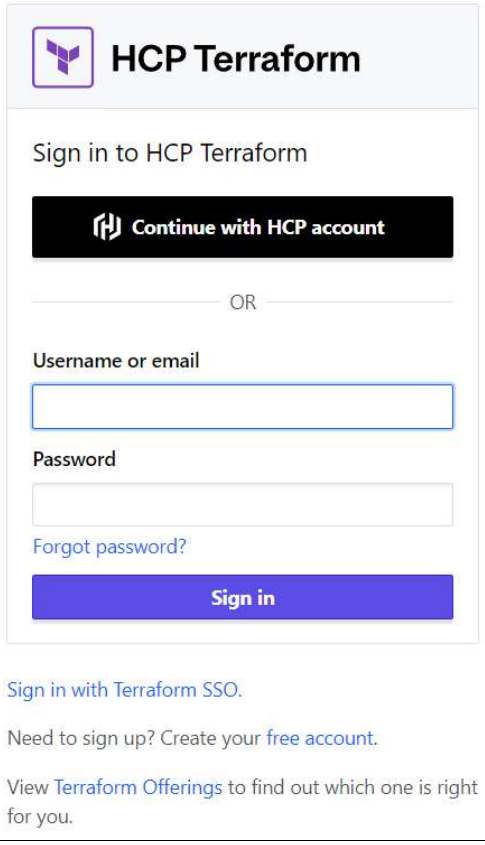
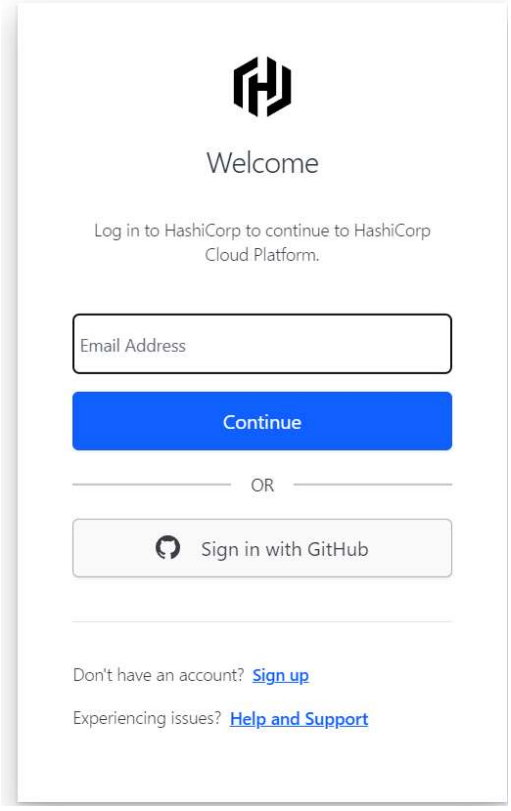
Sample of fi

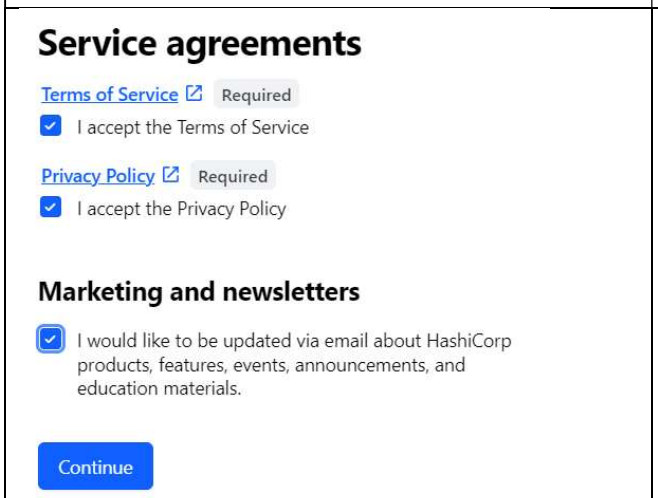
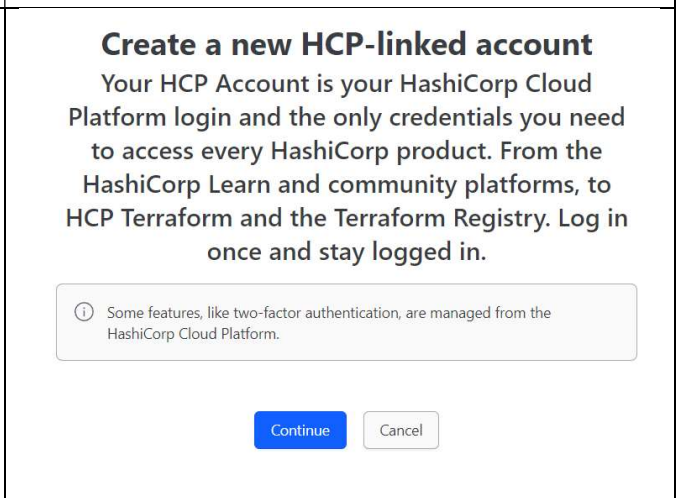
38. TERRAFORM CLOUD WITH GITHUB

Terraform Cloud is a managed service provided by HashiCrop that facilitates collaboration on Terraform configurations

Providing features like

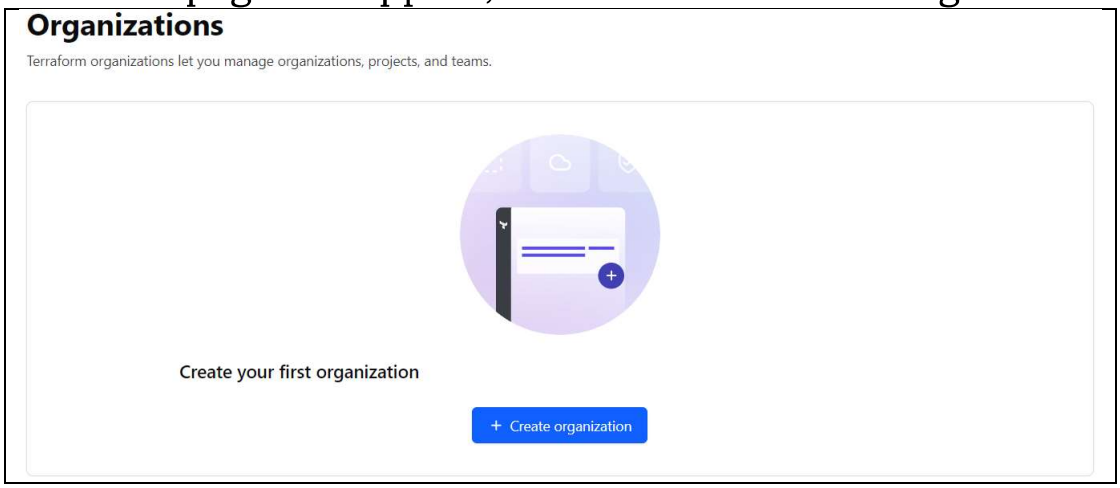
- remote state management,
- version control system (VCS) integration,
- automated runs, and
- secure variable management.

STEP-1	STEP-2
Now go to any browser and type “app.terraform.io” then this page will appear	Since you have an HCP account and are logged in, click on **"Continue with HCP account"**
	

STEP-3	STEP-4
Now click on “Sign in with GitHub”. Then tick on all the terms and conditions then click on “Continue”	Now again click on “Continue”.
	

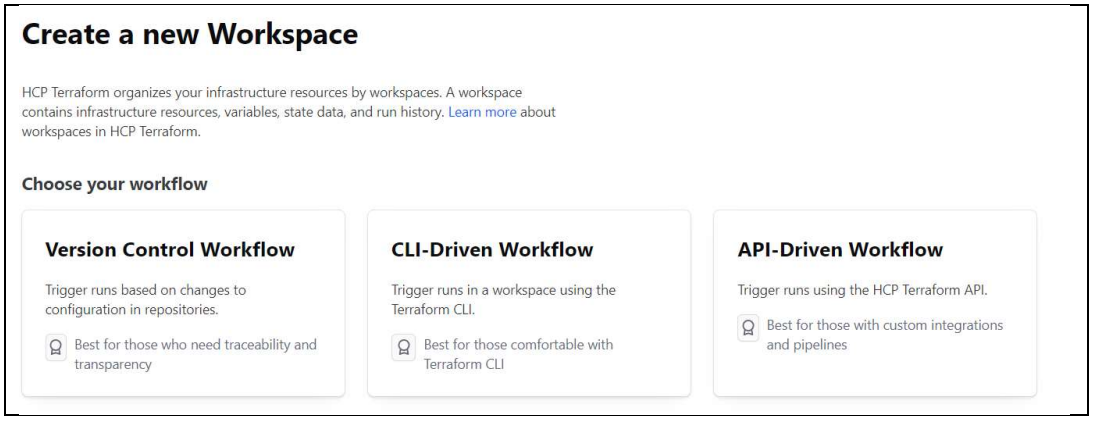
STEP-5

Then this page will appear, then click on “Create organization”



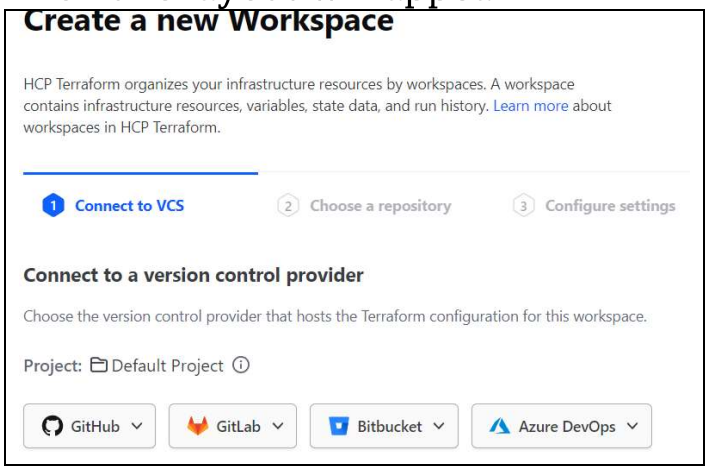
STEP-6

Then give the “Organization name” then “Create organization”. Then this layout will appear



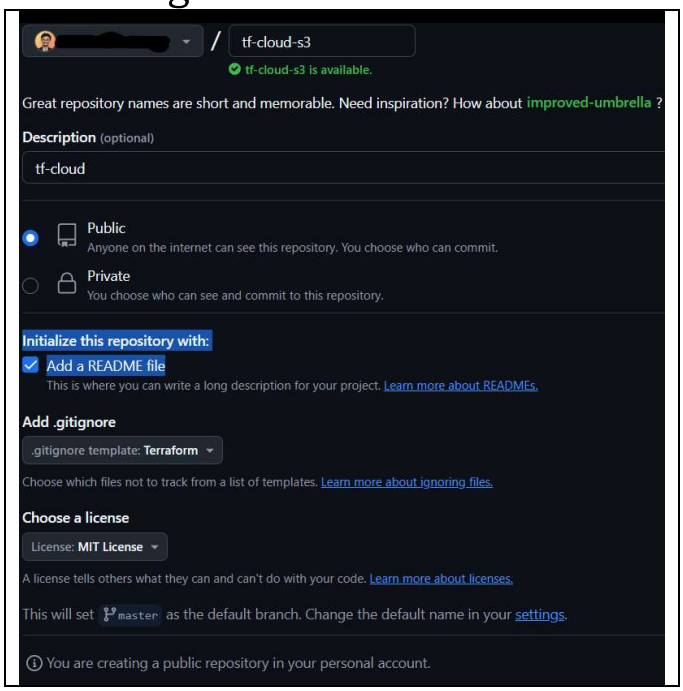
STEP-7

For the current demo we will be using “Version a new Workspace”. Then this layout will appear



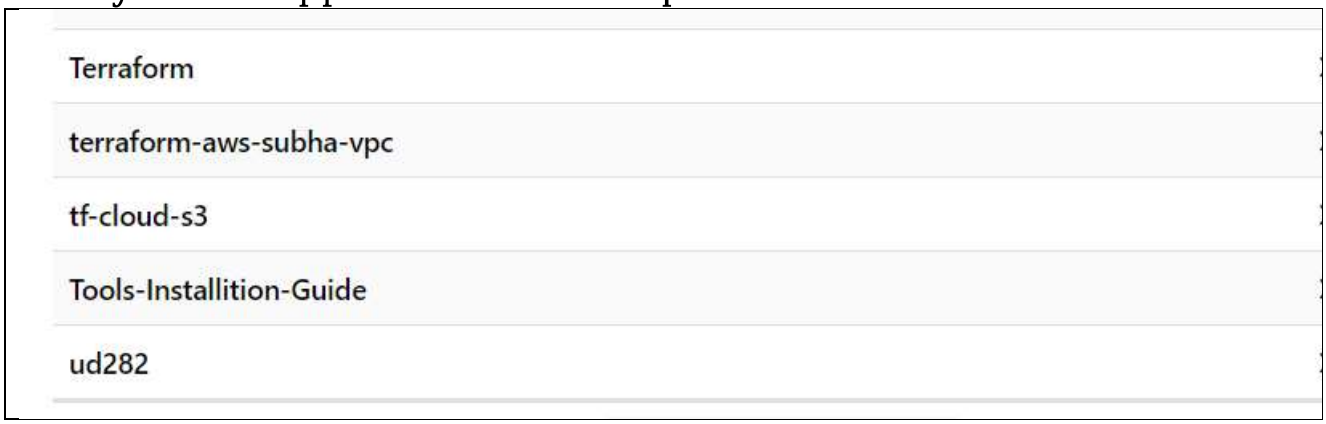
STEP-8

For now go to the GitHub and create a new repo like this



STEP-9

Now go to terraform cloud workspace and click on GitHub under that select "GitHub.com" then allow HCL to have access with GitHub, then this layout will appear with all the repos.



Then select "tf-cloud-s3" then give the description
Finally click on "create" for creating the workspace

Now clone the repo in local directory. Then this is the file structure by adding "main.tf"

In main.tf

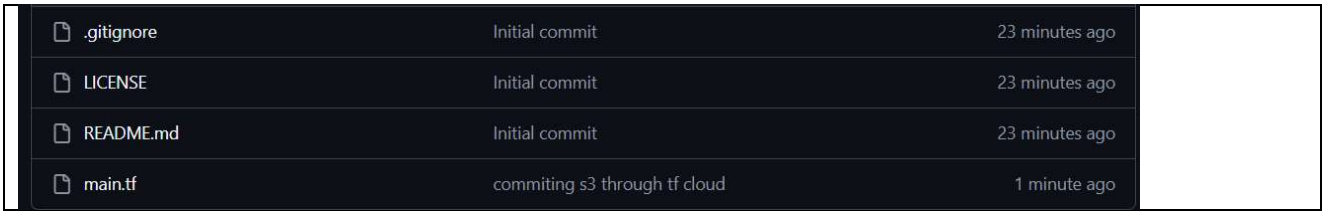
```
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "5.75.1"
    }
    random = {
      source = "hashicorp/random"
      version = "3.6.3"
    }
  }
}
provider "aws" {
  region = "us-east-1"
}
resource "random_id" "main" {
  byte_length = 12
}
resource "aws_s3_bucket" "demo-bucket" {
  bucket = "tf-cloud-s3-${random_id.main.hex}"
}
```

In terminal

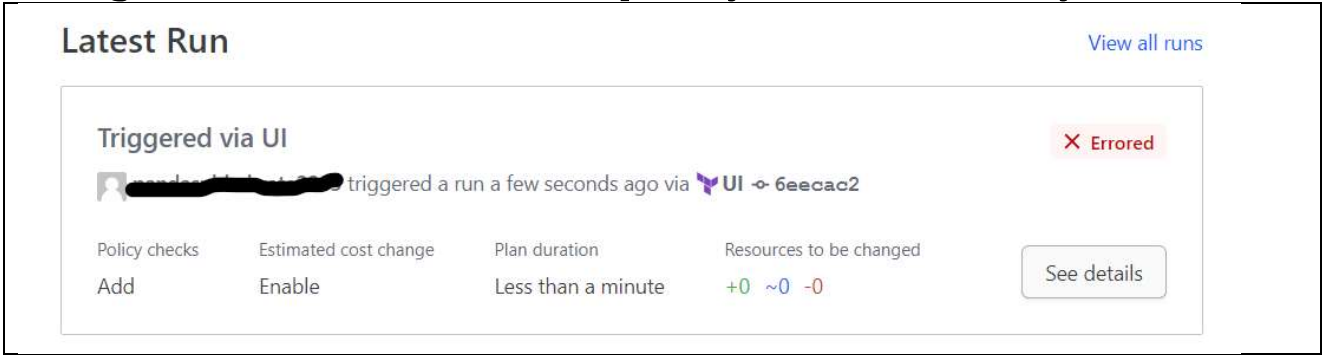
```
PS D:\Downloads\TERRAFORM\AWS\tf-cloud-s3> git status
On branch master
Your branch is up to date with 'origin/master'.

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        main.tf

nothing added to commit but untracked files present (use "git add" to track)
PS D:\Downloads\TERRAFORM\AWS\tf-cloud-s3>
```

```
PS D:\Downloads\TERRAFORM\AWS\tf-cloud-s3> git add .
PS D:\Downloads\TERRAFORM\AWS\tf-cloud-s3> git commit -m "commiting s3 through tf cloud"
[master 6eecac2] commiting s3 through tf cloud
 1 file changed, 24 insertions(+)
 create mode 100644 main.tf
PS D:\Downloads\TERRAFORM\AWS\tf-cloud-s3> git push -u origin master
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 12 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 481 bytes | 481.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0), pack-reused 0 (from 0)
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To https://github.com/Subhabrata2468/tf-cloud-s3.git
   1589dca..6eecac2  master -> master
branch 'master' set up to track 'origin/master'.
PS D:\Downloads\TERRAFORM\AWS\tf-cloud-s3>
```

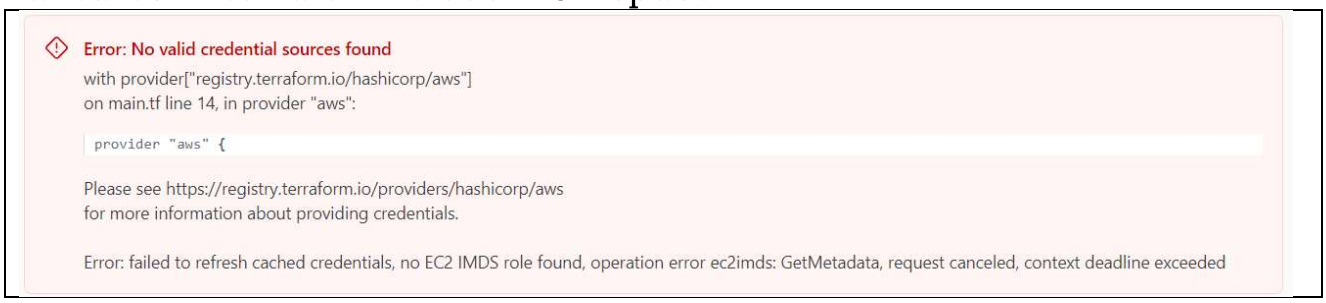
For now in GitHub



Then go to terraform cloud workspace, you will see this layout



Then you will find this error, as you have not set the variables in the variables in terraform cloud workspace



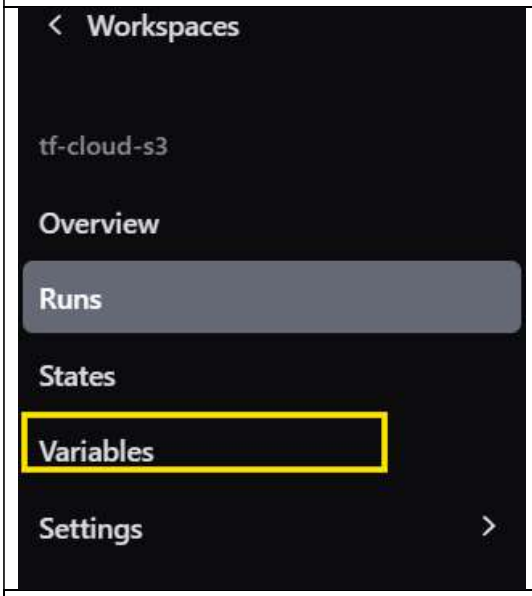
To resolve this error

First go to AWS

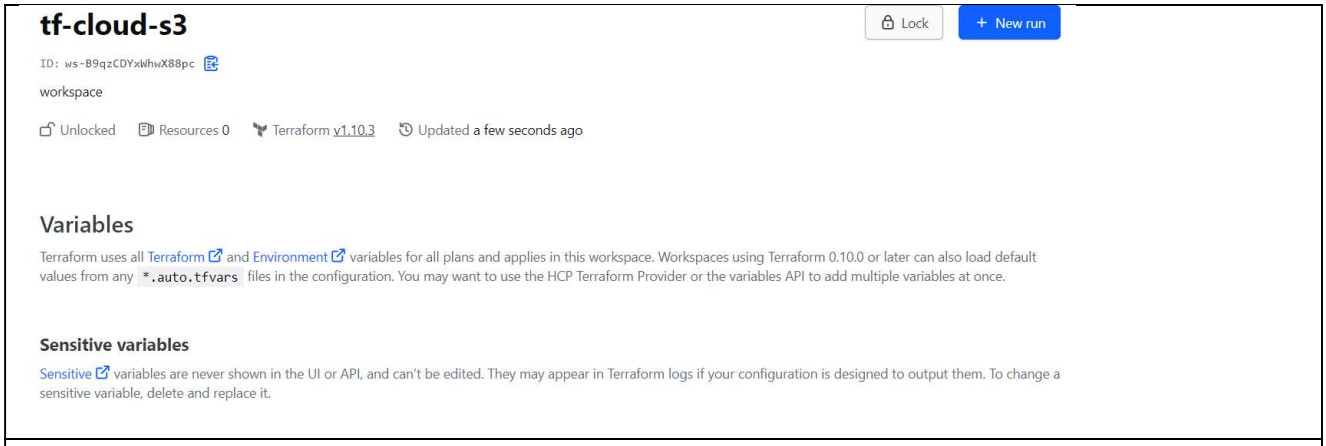
- ➔then IAM
- ➔then go for users
- ➔select “tf-user” as you are working in terraform as this users
- ➔either create the access key id or reuse the id that you are using in your local directory
- ➔form their copy the “Access key” and “Secret Access key”

Second go to terraform cloud workspace

Search for Variables



Then this layout will appear



In this either you can set the

Workspace variables

Variables defined within a workspace always overwrite variables from variable sets that have the same type and the same key

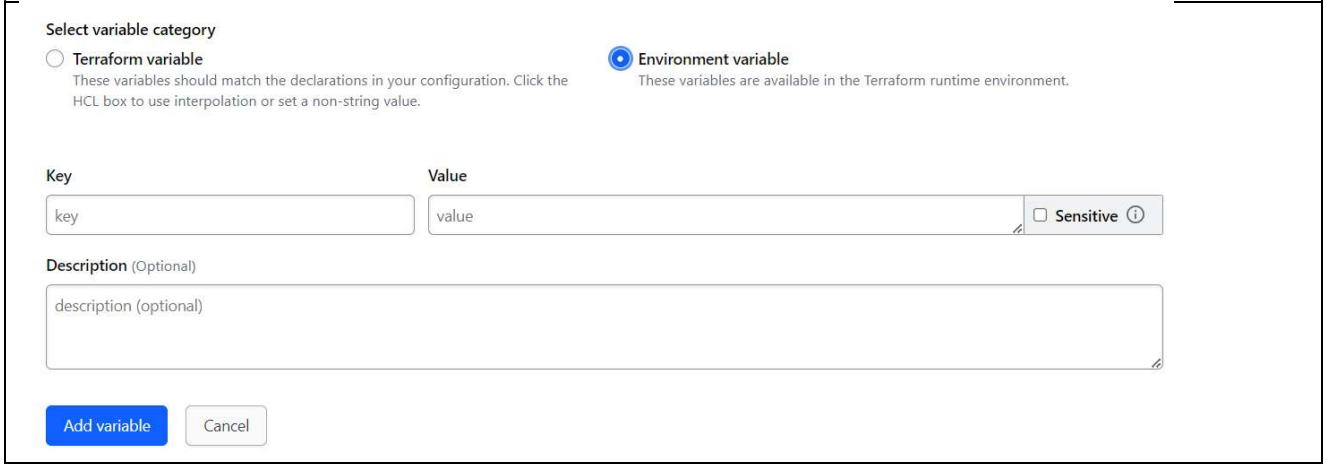
Or

Variable sets (0)

Allow you to reuse variables across multiple workspaces within your organization, recommend for creating a variable set for variables used in more than one workspace.

For now we will be creating “Workspace variables”. To do that click in “+Add variables”. As it is a environment variables click on “Environment variables”

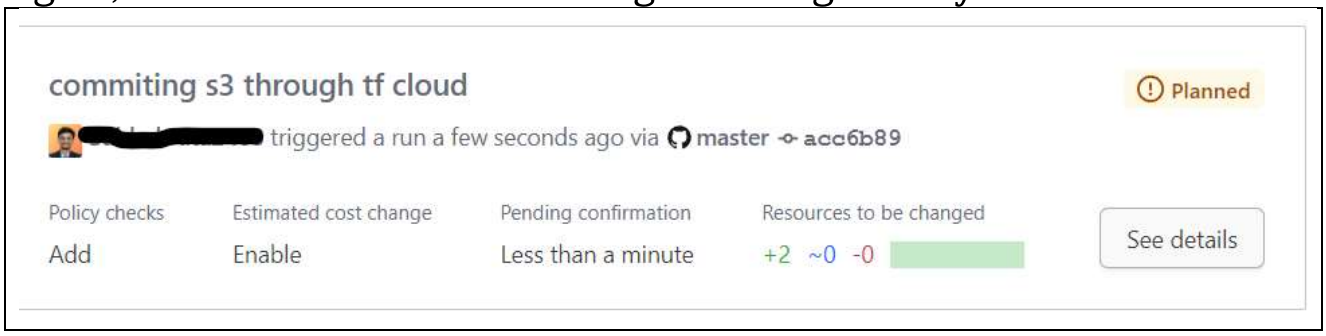
First add access key with the name “AWS_ACCESS_KEY_ID”
Then for aws secret access key we will be naming it as “AWS_SECRET_ACCESS_KEY”



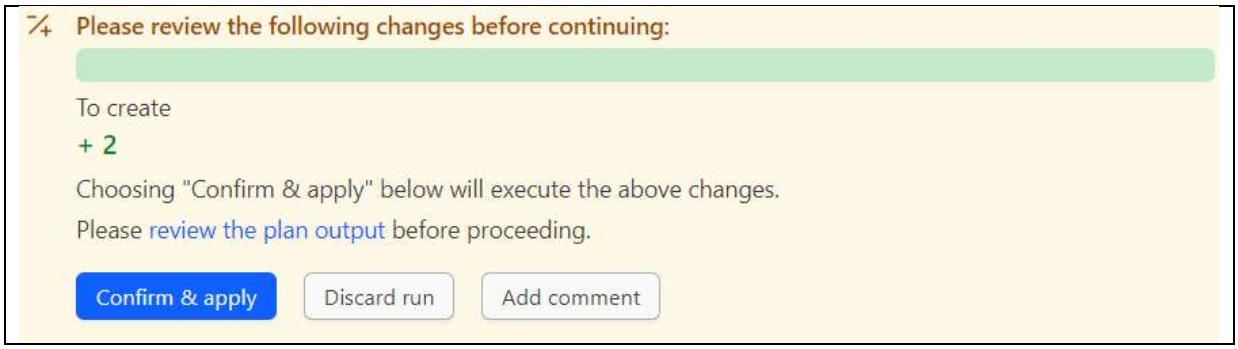
Give the key and value and also tick the “Sensitive option”
Then click on “Add variables”

Now go to code that is present in local directory now just do the minor change something like change byte_length =12 to byte_length = 13

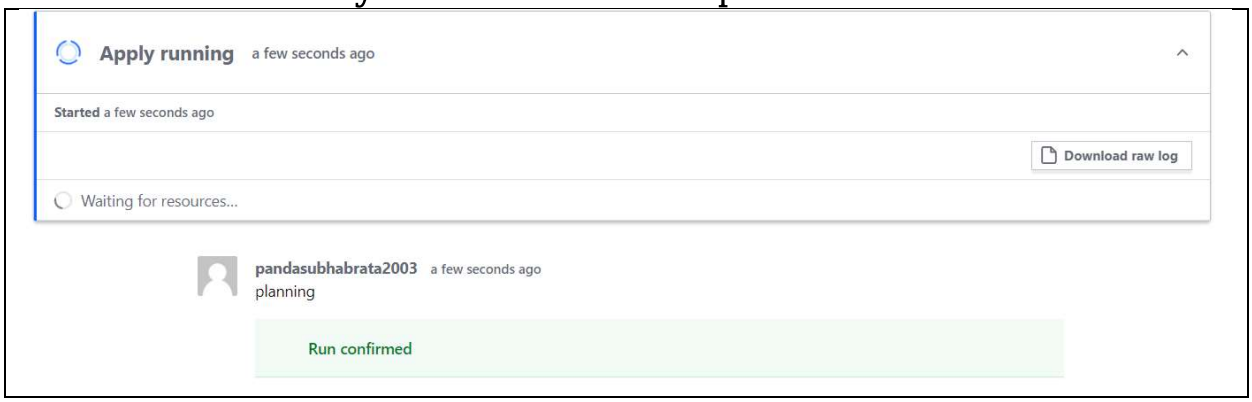
Again, commit it now see the change iin the given layout



Click on “See details” then you will see the layout which is asking for next information



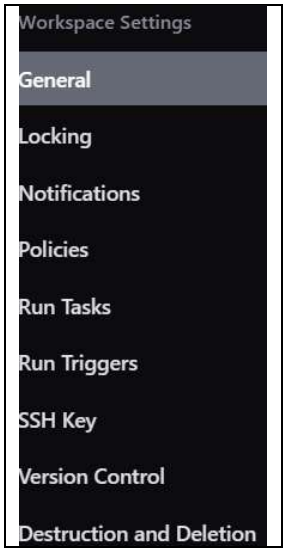
For now click on “Confirm & apply”, then it will ask for giving the comment and finally click on “Confirm plan”



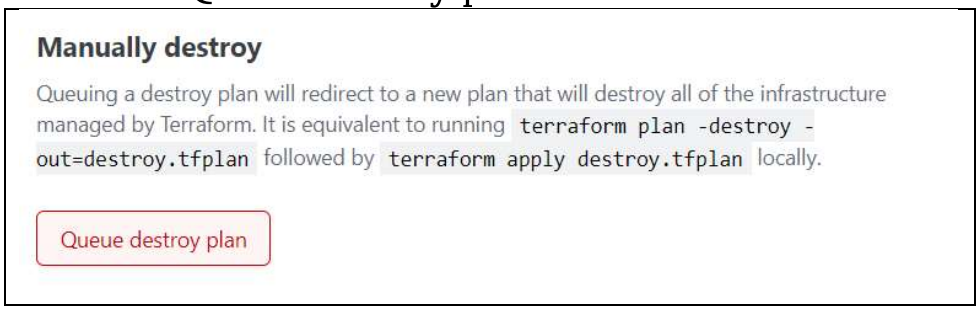
Go to S3 bucket in AWS console for verification

For seeing the states go to “States” option for seeing beautiful representation

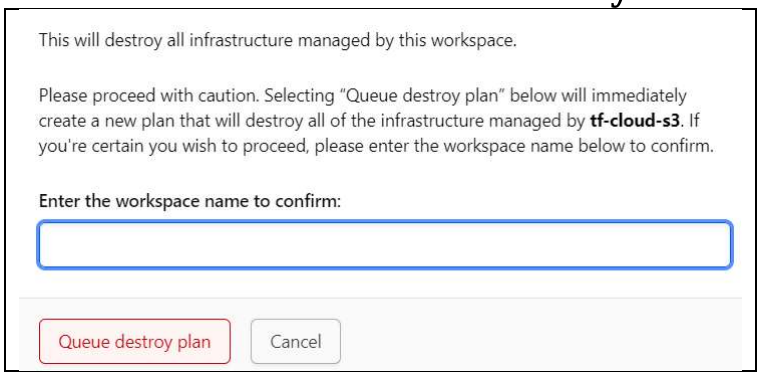
For destruction of s3-bucket.
Click on “Setting” you will see “Destruction and Deletion”



Click on “Queue destroy plan”



Then read the instruction carefully and fill in the blank



Then click on “Confirm and apply” → give the comment → click on confirm plan

Resources will be deleted.