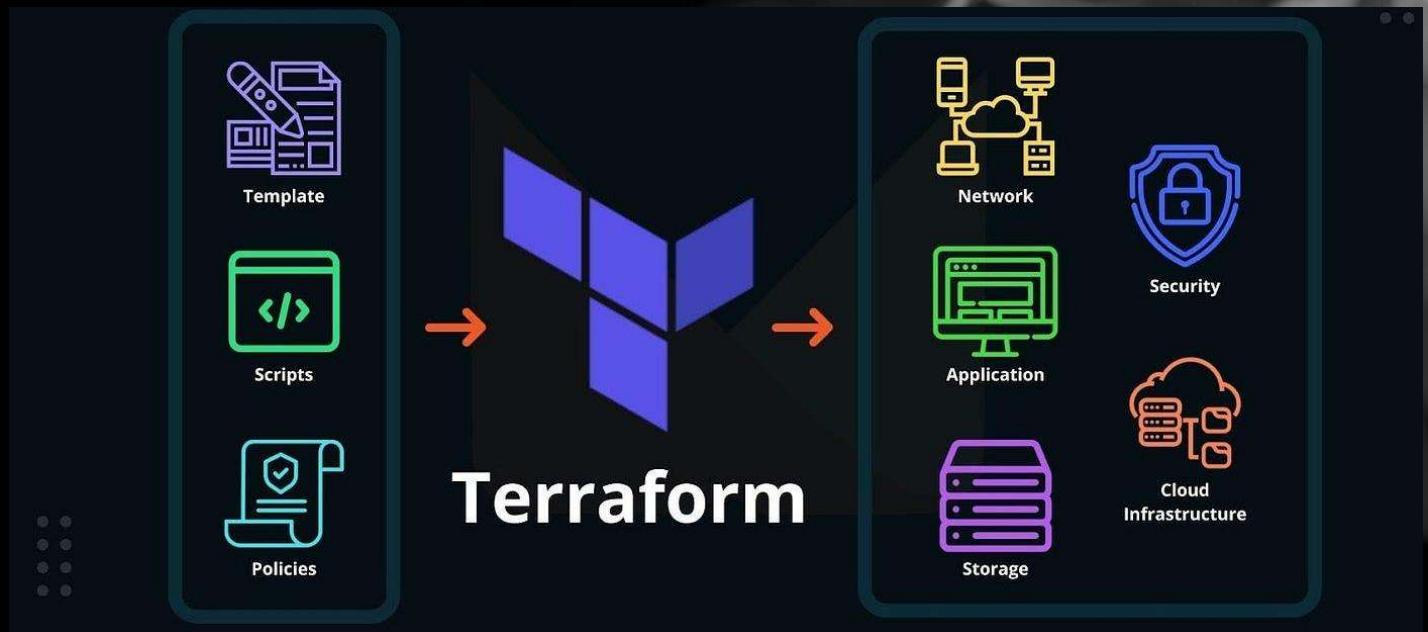


# TERRAFORM

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

1. PREREQUISITES	7
2. INTRODUCTION	7
3. IAC – INFRASTRUCTURE AS CODE	9
4. INSTALL TERRAFORM ON WINDOWS	10
5. INSTALL TERRAFORM ON MAC/UBUNTU	12
6. CHOOSE THE PROVIDER	13
7. SETUP AWS ACCOUNT	14
8. AWS DASHBOARD	16
9. AWS USER SETUP	17
10. DOWNLOAD AWS ON WINDOWS	21
11. AWS CLI CONFIGURATION FOR VS CODE	23
12. AWS EC2 WITH TERRAFORM	26
11.1 AMI	
11.2 terraform plan	
11.3 terraform apply	
11.4 terraform destroy	
11.5 terraform destroy -auto-approve	
11.6 terraform validate	
13. RESOURCE CHANGE	33
14. VARIABLES IN TERRAFORM	34
13.1 Syntax	
15. OUTPUT IN TERRAFORM	36
16. IMPLEMENT S3 BUCKET WITH THE HELP OF TERRAFORM	37
15.1 Introduction of S3 Bucket	

17. RANDOM PROVIDER	40
16.1 Syntax	
18. TERRAFORM REMOTE STATE MANAGEMENT	43
17.1 Keypoints	
17.2 Syntax	
19. PROJECT 1 -DEPLOY STATIC WEBSITE ON AWS USING S3 BUCKET	45
18.1 Reference	
18.2 Working	
20. UNDERSTAND VPC FOR TERRAFORM IMPLEMENTATION	48
19.1 VPC CIDR Block	
19.2 Internet Gateway	
19.3 Route Tables	
19.4 Security Target Groups	
19.5 NACL	
19.6 Subnets	
19.7 NAT Gateway	
19.8 AWS Peering	
19.9 Route 53	
21. IMPLEMENTING VPC USING TERRAFORM	54
20.1 Introduction	
22. DATA RESOURCE	59
21.1 Real Life Scenario	
23. CREATE EC2 USING EXISTING VPC	63

24. TERRAFORM VARIABLES	64
23.1 Real Life Scenario	
23.2 Problem without Validation	
23.3 Use of map	
23.4 Use of flatten	
23.5 Use of lookup	
23.6 Environment Variables	
23.7 <code>terraform.tfvars</code>	
23.8 <code>terraform.auto.tfvars</code>	
23.9 Diagram	
25. LOCAL VARIABLES IN TERRAFORM	73
24.1 Feature	
26. TERRAFORM: OPERATIONS & EXPRESSIONS	74
27. TERRAFORM: FUNCTIONS	75
28. TERRAFORM: MULTIPLE RESOURCES	81
27.1 <code>count</code>	
27.2 <code>count.index</code>	
27.2 <code>for_each</code>	
29. PROJECT 2 – AWS IAM MANAGEMENT	91
28.1 Introduction	
30. TERRAFORM MODULES	101
29.1 Real Life Scenario	
29.2 Without the use of modules	
29.3 With the use of module	
29.4 Implementing VPC using Terraform Module	
29.5 Implementing EC2 using Terraform Module	
29.6 Building Terraform Module	
31. PREPARE MODULES TO PUBLISH	115

32. TERRAFORM DEPENDENCIES	123
33. TERRAFORM LIFECYCLE	125
32.1 <code>create_before_destroy</code>	
32.2 <code>prevent_destroy</code>	
32.3 <code>ignore_destroy</code>	
32.4 <code>replace_triggered_by</code>	
34. PRE & POST CONDITION RESOURCES VALIDATIONS	130
33.1 Syntax for precondition	
33.2 Syntax for postcondition	
33.3 Examples	
33.4 Combined example of precondition and postcondiotn	
35. TERRAFORM STATE MODIFICATIONS	133
34.1 <code>terraform state list</code>	
34.2 <code>terraform state show</code>	
34.3 <code>terraform state mv</code>	
34.4 <code>terraform state rm</code>	
34.5 <code>terraform state pull</code>	
34.6 <code>terraform state push</code>	
34.7 <code>terraform state</code>	
36. TERRAFORM IMPORT COMMANDS	136
37. TERRAFORM WORKSPACES	138
35.1 Uses	
35.2 Working	
35.3 Diagram	
38. TERRAFORM CLOUD WITH GITHUB	141



# 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 maintains a detailed record of current state of managed resources.



### 3. I A C

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

E.g.

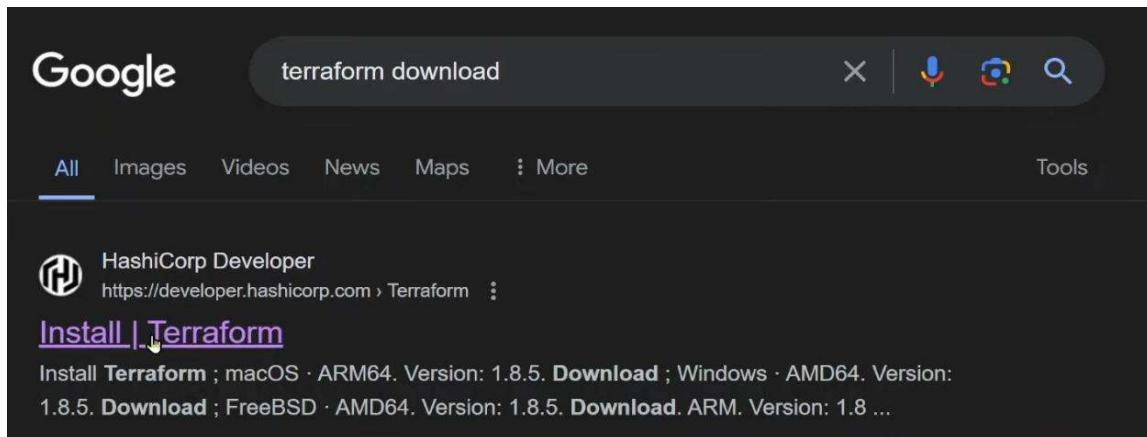
- You want to host the EC2 instance in AWS cloud, normally what we will do is to go to
  - dashboard
  - setup up the instance step by step graphically

Rather than doing the graphically, write the simply configuration file that contains the details like

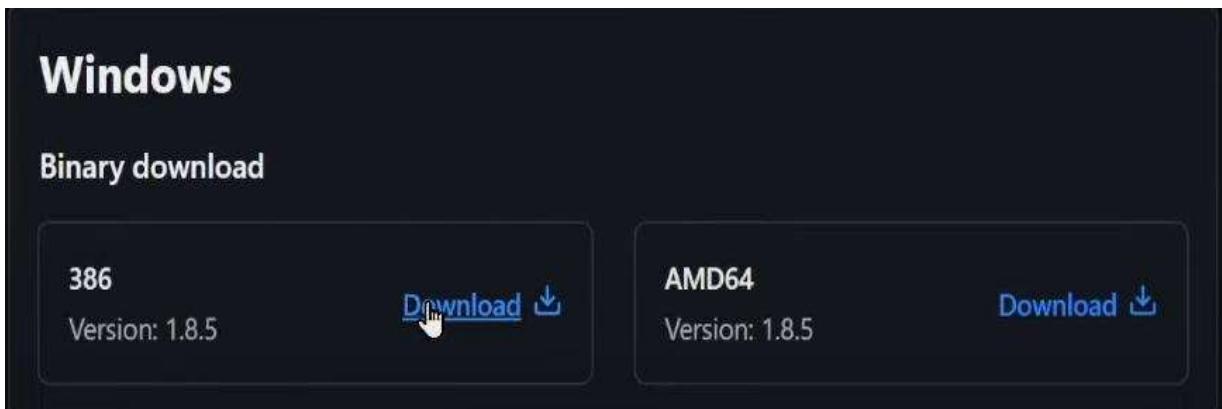
- to host the instance
- which os is required
- what all resources is required for the instance

Then will simply execute the file, then IAC will automatically do all the things you asked for.

## 4. INSTALLATION ON TERRAFORM ON WINDOWS

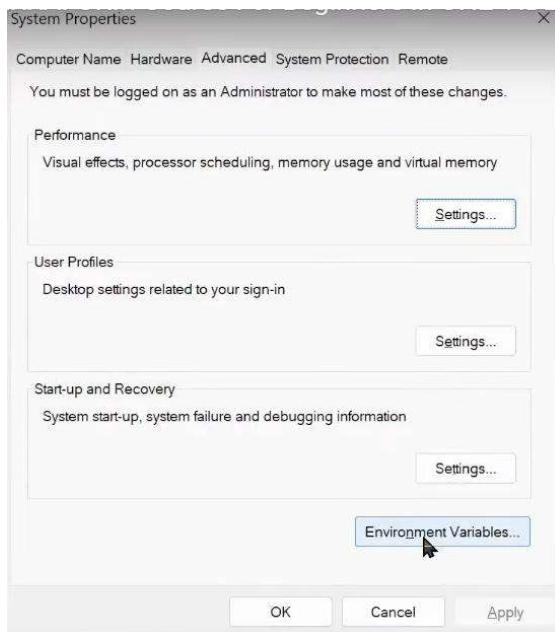


Scroll Down and search for windows section and download binary of terraform

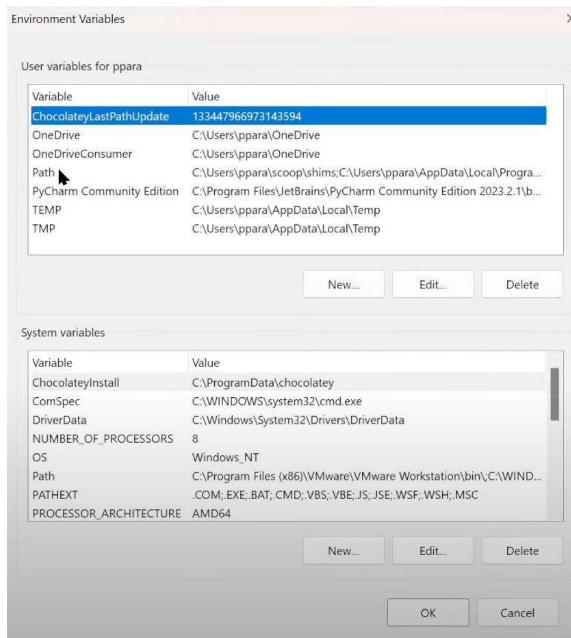


Extract the files that you have download

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

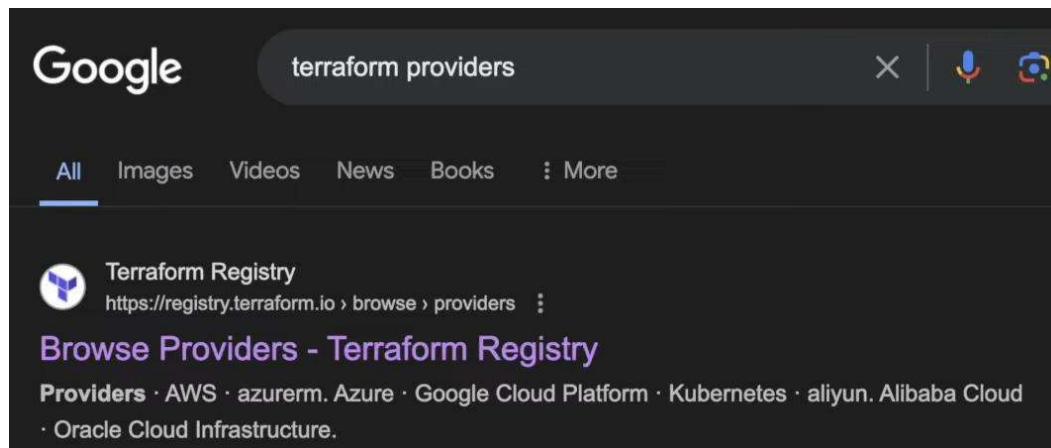
```
sudo yum install -y yum-utils shadow-utils
```

```
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

A screenshot of the HashiCorp Terraform Registry interface. At the top, there is a navigation bar with links for "Terraform", "Registry", "Browse", "Publish", "Sign-in", and "Use Terraform Cloud for free". Below the navigation bar is a search bar with the placeholder "Search all resources". The main content area has tabs for "Providers", "Modules", "Policy Libraries", and "Run Tasks". On the left side, there are "Filters" for "Tier" (with "Official" checked), "Category" (with "HashiCorp Platform", "Infrastructure Management", "Public Cloud", "Asset Management", and "Cloud Automation" listed), and "Clear Filters". The right side features a section titled "Providers" with a description: "Providers are a logical abstraction of an upstream API. They are responsible for understanding API interactions and exposing resources." It shows three cards for "AWS" (orange background, white logo), "Azure" (blue background, blue logo), and "Google Cloud Platform" (blue background, white logo).

## 7. SETUP AWS ACCOUNT

Go to this account [aws.amazon.com](https://aws.amazon.com)

The screenshot shows the top navigation bar of the AWS website. It includes the AWS logo, a search bar, and links for About AWS, Contact Us, Support, English, My Account, and Sign In to the Console. Below the navigation bar, there's a horizontal menu with links for Products, Solutions, Pricing, Documentation, Learn, Partner Network, AWS Marketplace, Customer Enablement, Events, Explore More, and a search icon.

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
<p>Explore Free Tier products with a new AWS account. To learn more, visit <a href="https://aws.amazon.com/free">aws.amazon.com/free</a>.</p>	<p>Sign up for AWS</p> <p>Secure verification</p> <p>We will not charge you for usage below AWS Free Tier limits. We may temporarily hold up to \$1 USD (or an equivalent amount in local currency) as a pending transaction for 3-5 days to verify your identity.</p>

STEP-3	STEP-4
<p>Sign up for AWS</p> <p>Confirm your identity <small>(info)</small></p> <p>Name <small>(info)</small> Choose the name that you want to use for identity verification. <input checked="" type="radio"/> Paul</p> <p>Primary purpose of account registration Choose one that best applies to you. If your account is tied to a business, select the one that applies to your business. <input type="button" value="Select one"/></p> <p>Ownership type <input type="button" value="Select one"/></p> <p><input type="checkbox"/> I consent to allowing AWS to use and send the information above to a third-party service for identity verification purposes.</p>	<p>Sign up for AWS</p> <p>Confirm your identity</p> <p>Before you can use your AWS account, you must verify your phone number. When you continue, the AWS automated system will contact you with a verification code.</p> <p>How should we send you the verification code? <input checked="" type="radio"/> Text message (SMS) <input type="radio"/> Voice call</p> <p>Country or region code United States (+1)</p> <p>Mobile phone number</p> <p>Security check </p> <p>Type the characters as shown above</p>

STEP-5	STEP-6			
<p>Select the basic support</p>  <p><b>Sign up for AWS</b></p> <p><b>Select a support plan</b></p> <p>Choose a support plan for your business or personal account. Compare plans and pricing examples.</p> <p>You can change your plan anytime in the AWS Management Console.</p> <table border="1"> <tr> <td><input checked="" type="radio"/> <b>Basic support - Free</b> Recommended for new users just getting started with AWS 24x7 self-service access to AWS resources For account and billing issues only Access to Personal Health Dashboard &amp; Trusted Advisor</td> <td><input type="radio"/> <b>Developer support - From \$29/month</b> Recommended for developers experimenting with AWS Email access to AWS Support during business hours 12 (business)-hour response times</td> <td><input type="radio"/> <b>Business support - From \$100/month</b> Recommended for running production workloads on AWS 24x7 tech support via email, phone, and chat 1-hour response times Full set of Trusted Advisor best-practice recommendations</td> </tr> </table> <p><b>Need Enterprise level support?</b> From \$15,000 a month you will receive 15-minute response times and concierge-style experience with an assigned Technical Account Manager. <a href="#">Learn more</a></p>	<input checked="" type="radio"/> <b>Basic support - Free</b> Recommended for new users just getting started with AWS 24x7 self-service access to AWS resources For account and billing issues only Access to Personal Health Dashboard & Trusted Advisor	<input type="radio"/> <b>Developer support - From \$29/month</b> Recommended for developers experimenting with AWS Email access to AWS Support during business hours 12 (business)-hour response times	<input type="radio"/> <b>Business support - From \$100/month</b> Recommended for running production workloads on AWS 24x7 tech support via email, phone, and chat 1-hour response times Full set of Trusted Advisor best-practice recommendations	<p>If this page shows, then you are good to go</p>   <p><b>Congratulations</b></p> <p>Thank you for signing up for AWS.</p> <p>We are activating your account, which should only take a few minutes. You will receive an email when this is complete.</p> <p><a href="#">Go to the AWS Management Console</a></p> <p>Sign up for another account or contact sales.</p>
<input checked="" type="radio"/> <b>Basic support - Free</b> Recommended for new users just getting started with AWS 24x7 self-service access to AWS resources For account and billing issues only Access to Personal Health Dashboard & Trusted Advisor	<input type="radio"/> <b>Developer support - From \$29/month</b> Recommended for developers experimenting with AWS Email access to AWS Support during business hours 12 (business)-hour response times	<input type="radio"/> <b>Business support - From \$100/month</b> Recommended for running production workloads on AWS 24x7 tech support via email, phone, and chat 1-hour response times Full set of Trusted Advisor best-practice recommendations		

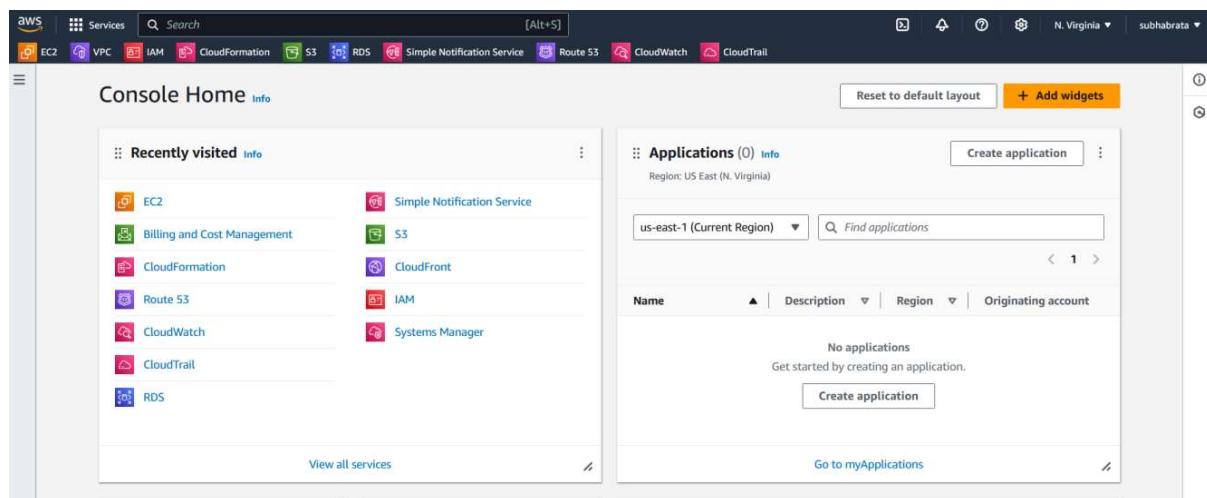
STEP-7
<p>And you will also get the mail something like this</p>  <p>Welcome to Amazon Web Services.</p> <p>Explore more than 100 products through free trials and always free offers to start building on Amazon Web Services using the <a href="#">Free Tier</a>. Some offers are only available to new customers for 12 months following your Amazon Web Services sign-up date.</p> <p>When your 12 month free usage term expires or if your application use exceeds the tiers, you simply pay standard, pay-as-you-go service rates (see each service page for full pricing details). For more details, see the <a href="#">offer terms</a>.</p> <p>Here are a few easy ways to get started:</p> <ul style="list-style-type: none"> <li>* <a href="#">Find technical documentation</a></li> <li>* <a href="#">Learn with tutorials and guides</a></li> <li>* <a href="#">Start building in the Amazon Web Services Console</a></li> </ul>

## 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

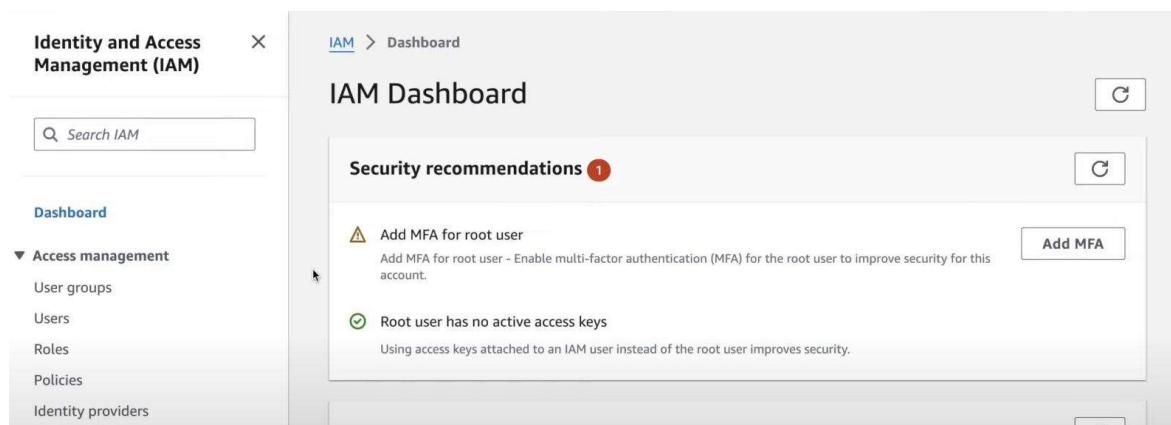
Now we need to do user setup because we need accessibility from user as we will be performing action on the local machine for AWS.

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

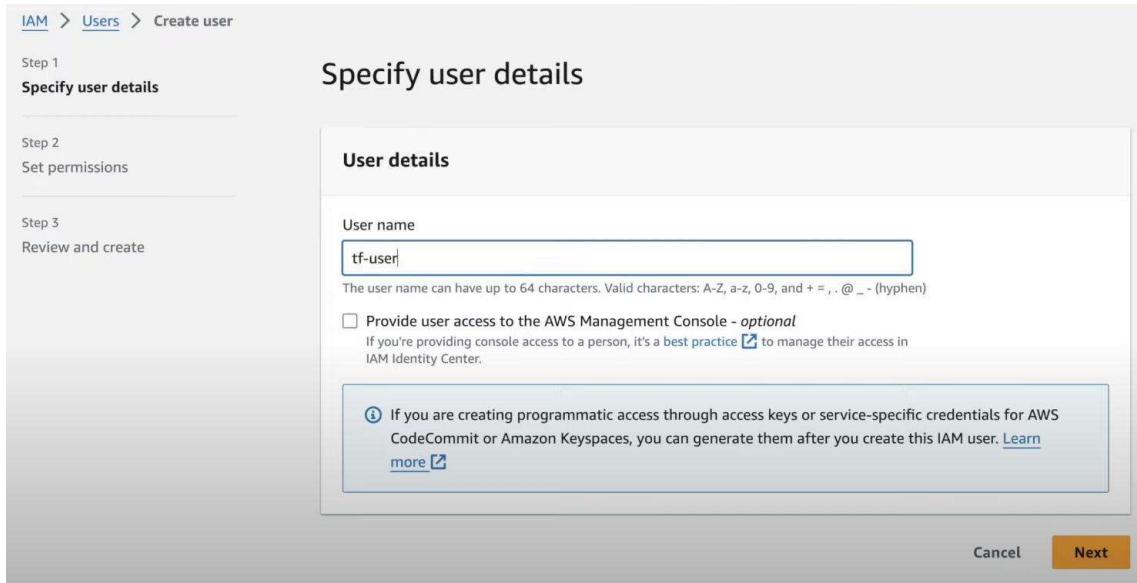
**IAM → AWS Identity and Access Management (IAM)** is a critical service that enables you to securely control access to AWS resources. It provides fine-grained access management for AWS services and resources, ensuring that only authorized users and applications can interact with your cloud infrastructure.

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

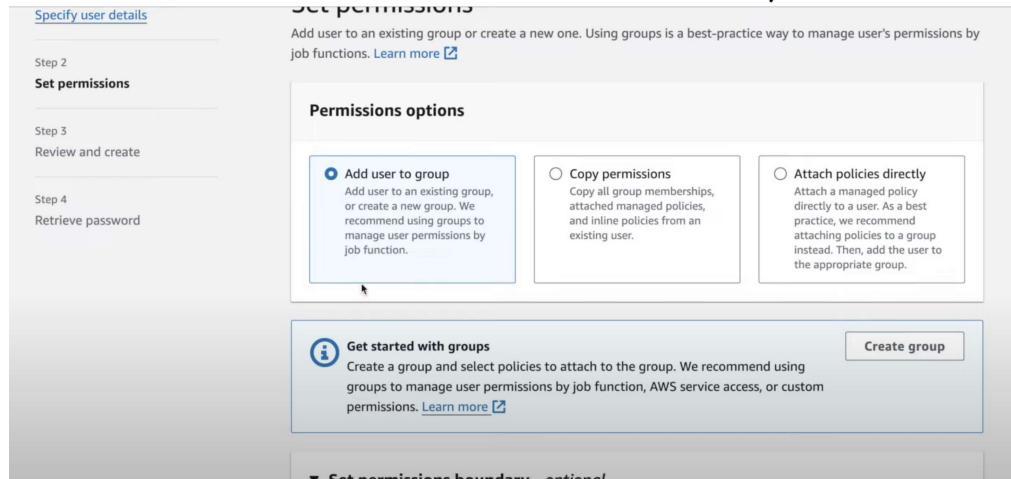


Now click on “User” → Give the User name → And Click on “Next”



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

User group name  
Enter a meaningful name to identify this group.

Maximum 128 characters. Use alphanumeric and '+,-,\_' characters.

### Permissions policies (930)

Filter by Type

Policy name	Type	Use...	Description
AdministratorAccess	AWS managed	None	Provides full access to AWS services
AdministratorAccess	AWS managed	None	Grants account administrative permission
AdministratorAccess	AWS managed	None	Grants account administrative permission
AlexaForBusinessDeviceSetup	AWS managed	None	Provides device setup access to Alexa
AlexaForBusinessFullAccess	AWS managed	None	Grants full access to AlexaForBusiness
AlexaForBusinessGatewayExecution	AWS managed	None	Provides gateway execution access to Alexa

→ For now, Tick “AdministratorAccess”

→ Click “Create user group”

AmazonAPIGateway

AWS managed      None      Provides full access to invoke APIs in the AWS Lambda service

→ 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

User created successfully  
You can view and download the user's password and email instructions for signing in to the AWS Management Console.

Step 1: Specify user details

Step 2: Set permissions

Step 3: Review and create

Step 4: Retrieve password

### Retrieve password

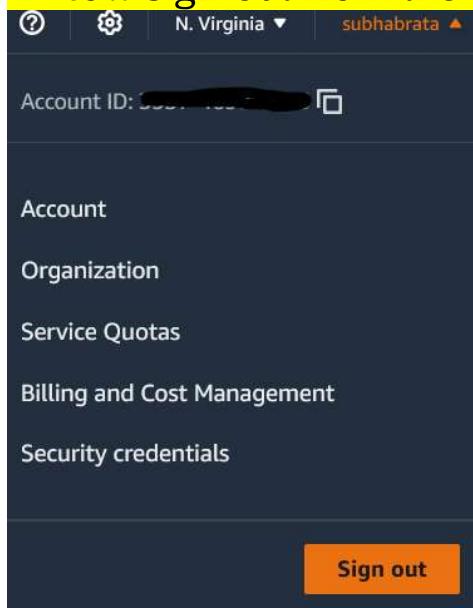
You can view and download the user's password below or email users instructions for signing in to the AWS Management Console. This is the only time you can view and download this password.

Console sign-in details	Email sign-in instructions
Console sign-in URL <a href="https://[REDACTED].signin.aws.amazon.com/console">https://[REDACTED].signin.aws.amazon.com/console</a>	<input type="button" value="Email sign-in instructions"/>
User name tf-user	
Console password ***** <input type="button" value="Show"/>	

→ 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
<p>This icon will appear</p> <p><b>Sign in</b></p> <p><input type="radio"/> <b>Root user</b> Account owner that performs tasks requiring unrestricted access. <a href="#">Learn more</a></p> <p><input checked="" type="radio"/> <b>IAM user</b> User within an account that performs daily tasks. <a href="#">Learn more</a></p> <p>Account ID (12 digits) or account alias</p> <p><input type="text"/></p> <p><b>Next</b></p> <p>By continuing, you agree to the <a href="#">AWS Customer Agreement</a> or other agreement for AWS services, and the <a href="#">Privacy Notice</a>. This site uses essential cookies. See our <a href="#">Cookie Notice</a> for more information.</p> <p>New to AWS?</p> <p><a href="#">Create a new AWS account</a></p>	<p>→Give the 12digit account</p> <p><b>Sign in as IAM user</b></p> <p>Account ID (12 digits) or account alias</p> <p><input type="text"/> 123456789012</p> <p>IAM user name</p> <p><input type="text"/></p> <p>Password</p> <p><input type="password"/></p> <p><input type="checkbox"/> Remember this account</p> <p><b>Sign in</b></p> <p><a href="#">Sign in using root user email</a></p> <p><a href="#">Forgot password?</a></p>

→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

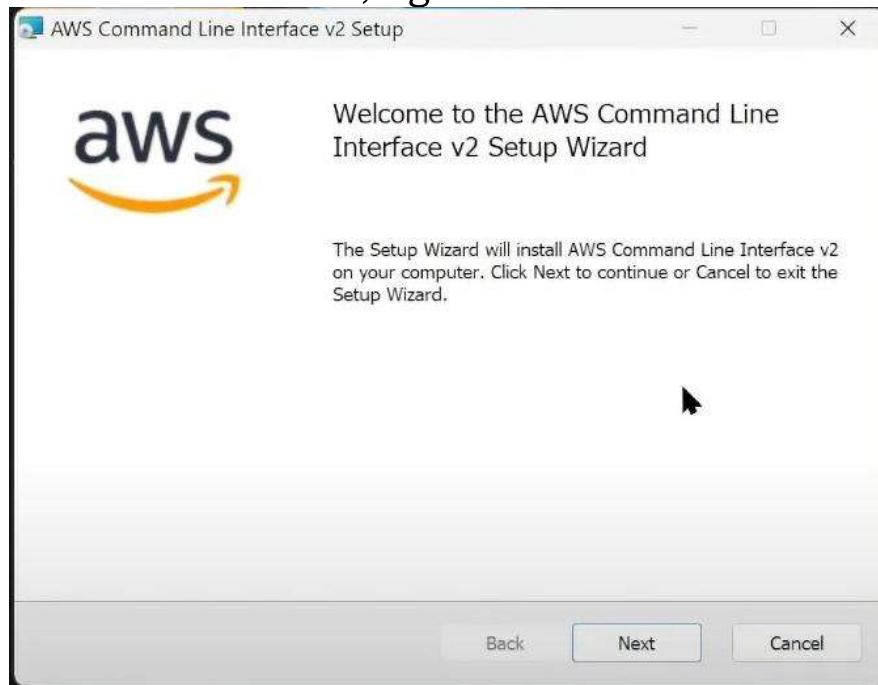
The screenshot shows the AWS Command Line Interface (CLI) page. At the top, there's a navigation bar with the AWS logo and the URL https://aws.amazon.com/cli. Below the navigation, the title "Command Line Interface - AWS CLI" is displayed in purple. A descriptive paragraph follows, stating: "With just one tool to download and configure, you can control multiple AWS services from the command line and automate them through scripts. The AWS CLI v2 ...". Below this, there are links for "Install/Update", "User Guide for Version 1", "New Features", and "Amazon Linux AMI".

Click on the given link

Click on 64-bit and it will automatically download it

The screenshot shows the main AWS Command Line Interface landing page. On the left, there's a sidebar with links to "AWS Command Line Interface", "Documentation", "Tools", and "Release Notes". Below that, there are buttons for "Get Started with AWS for Free" and "Create Free Account". The main content area has a heading "AWS Command Line Interface" in orange. It describes the AWS CLI as a unified tool to manage AWS services. It then lists three download options: "Windows" (highlighted with a red box), "MacOS", and "Linux". Each option includes a brief description and a link to the download page.

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

```
PS C:\Users\subha> aws
usage: aws [options] <command> <subcommand> [<subcommand> ...] [parameters]
To see help text, you can run:

aws help
aws <command> help
aws <command> <subcommand> help

aws: error: the following arguments are required: command
```

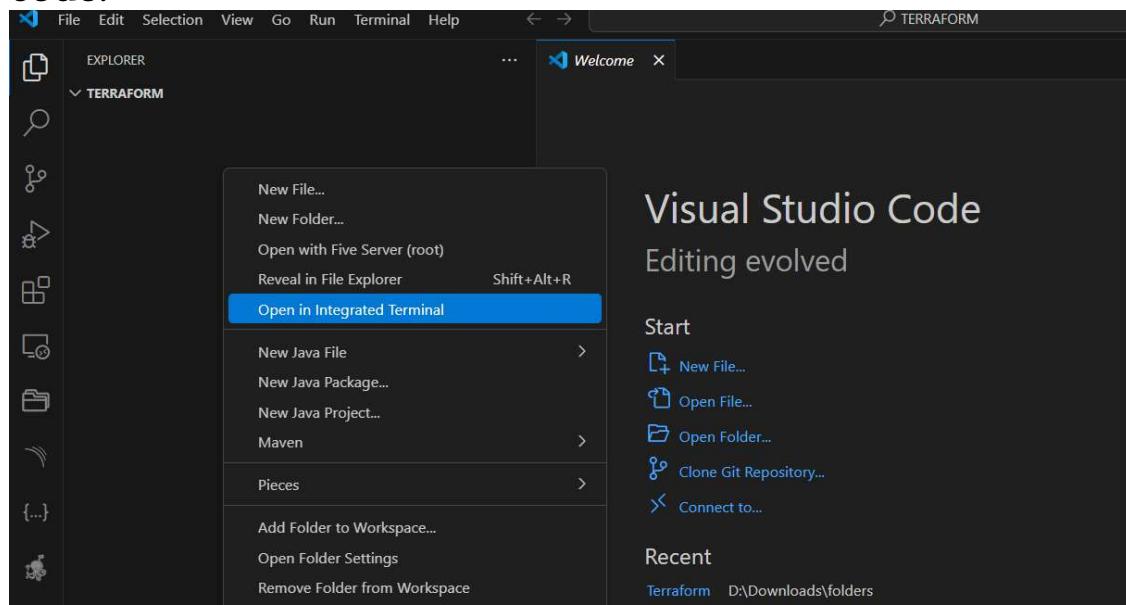
If you see this screen means you have successfully installed the aws cli.  
Now, we can access AWS cloud platform from a local environment.

## 11. AWS CLI CONFIGURE IN VS CODE

All the configuration we will make 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

The screenshot shows the AWS IAM 'Users' page. At the top, there is a search bar and navigation icons. Below the header, a table lists three IAM users:

User name	Path	Groups	Last activity	MFA	Password age	Console last sign-in
eks-cluster	/	0	26 days ago	-	-	-
permission-for-session-manager	/	0	-	-	-	-
tf-user	/	0	-	-	-	-

2. Click on “Create access key”

The screenshot shows the AWS IAM 'User Details' page for 'tf-user'. The 'Summary' section displays the following information:

ARN	Console access	Access key 1
arn:aws:iam::393746980563:user/tf-user	Enabled without MFA	Create access key
Created November 12, 2024, 22:40 (UTC+05:30)	Last console sign-in Never	

Tick on

## Access key best practices & alternatives Info

Avoid using long-term credentials like access keys to improve your security. Consider the following use cases and alternatives.

### Use case

#### ● Command Line Interface (CLI)

You plan to use this access key to enable the AWS CLI to access your AWS account.

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



HashiCorp Terraform v2.33.0

HashiCorp hashicorp.com | 4,479,061 | ★★★★★ (197)

Syntax highlighting and autocompletion for Terraform

Disable

Uninstall

Switch to Pre-Release Version

Auto Update

DETAILS

FEATURES

CHANGELOG

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

EXPLORER

▼ TERRAFORM

  ▼ AWS

    ▼ ec2-instance

      main.tf

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

The screenshot shows the HashiCorp Terraform Registry interface. At the top, there's a search bar and navigation links for 'Browse', 'Publish', 'Sign-in', and a button to 'Use HCP Terraform for free'. Below the header, the URL shows 'Providers / hashicorp / aws / Version 5.75.1 / Latest Version'. The main content area displays the 'aws' provider details. It includes the AWS logo, a 'Public Cloud' badge, and a note from HashiCorp stating: 'Lifecycle management of AWS resources, including EC2, Lambda, EKS, ECS, VPC, S3, RDS, DynamoDB, and more. This provider is maintained internally by the HashiCorp AWS Provider team.' Below this, it shows the provider version as '5.75.1' published '2 days ago' with a link to 'SOURCE CODE' at 'hashicorp/terraform-provider-aws'. To the right, a 'How to use this provider' section contains Terraform code examples:

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

*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”

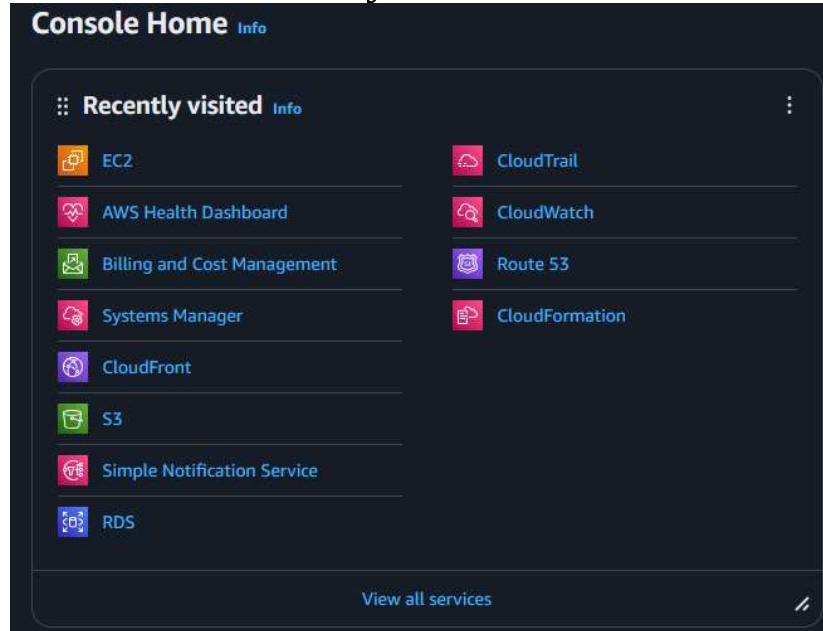
```
resource "aws_instance" "myec2" {  
}
```

Now you need to add all the necessary requirements to start the instance like “ami”, “instance\_type”

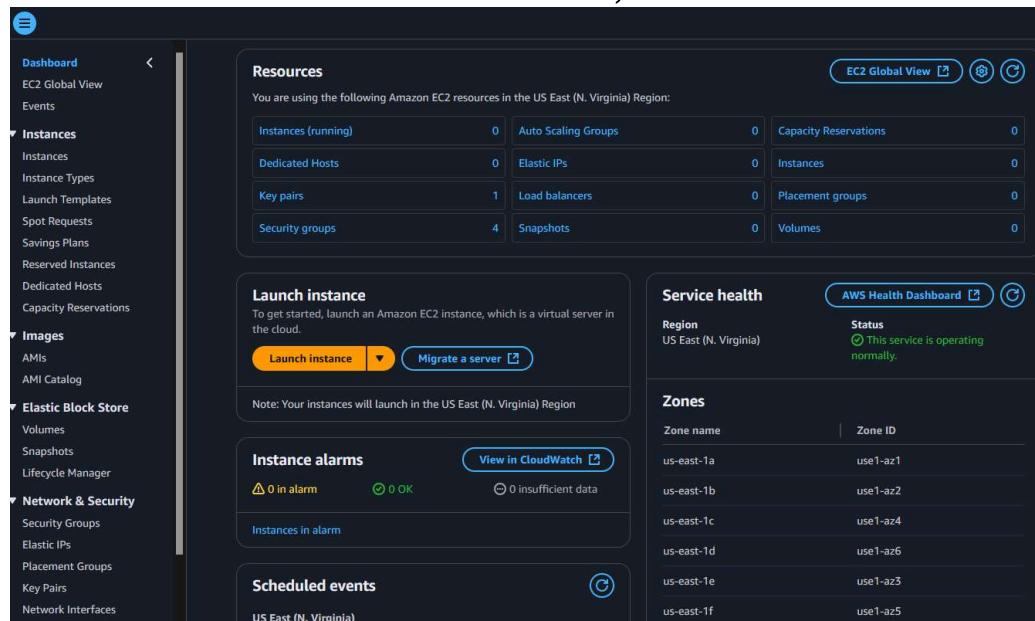
**Ami → Amazon Machine Image (AMI)** is a pre-configured template that contains the operating system, application server, and applications needed to launch an instance in **Amazon Elastic Compute Cloud (EC2)**.

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

The screenshot shows the AWS Application and OS Images (Amazon Machine Image) catalog. At the top, there's a search bar with placeholder text "Search our full catalog including 1000s of application and OS images". Below the search bar is a "Quick Start" section featuring icons for various operating systems: Amazon Linux, macOS, Ubuntu, Windows, Red Hat, SUSE Linux, and others. To the right of this section is a search icon and a link to "Browse more AMIs", with a note that it includes AMIs from AWS, Marketplace, and the Community. The main content area displays the "Amazon Machine Image (AMI)" for "Amazon Linux 2023 AMI". It shows the AMI ID as ami-0453ec754f44f9a4a, boot mode as uefi-preferred, and architecture as 64-bit (x86). The "AMI ID" field is highlighted with a red rectangle. To the right of the AMI details, there's a "Verified provider" badge. The "Description" section below the AMI details states: "Amazon Linux 2023 is a modern, general purpose Linux-based OS that comes with 5 years of long term support. It is optimized for AWS and designed to provide a secure, stable and high-performance execution environment to develop and run your cloud applications." At the bottom of the catalog page, there's a note: "Free tier eligible" with a dropdown arrow.

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

The screenshot shows the AWS Instance Type selection interface. At the top, there's a dropdown menu labeled "Instance type" with "Info" and "Get advice" links. Below it, the "Instance type" section lists the "t2.micro" option. The "t2.micro" entry includes details such as Family: t2, 1 vCPU, 1 GiB Memory, Current generation: true, and various On-Demand base pricing options. To the right of the instance type list, there are filters for "Free tier eligible" (selected), "All generations" (radio button), and a "Compare instance types" link. A note at the bottom states "Additional costs apply for AMIs with pre-installed software".

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
<pre>         ✓ TERRAFORM           ✓ AWS             ✓ ec2-instance               main.tf             trash       </pre>	<pre>         ✓ TERRAFORM           ✓ AWS             ✓ ec2-instance               .terraform               .terraform.lock.hcl               main.tf             trash       </pre>

***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.

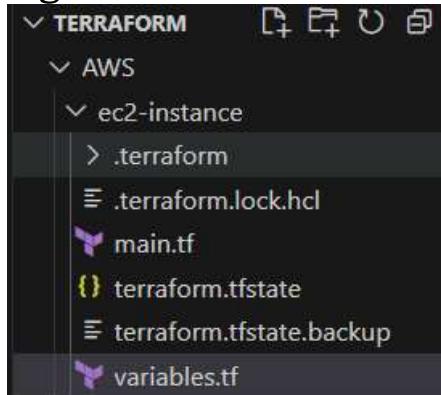
### Syntax:

```
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
<pre> 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"   } } </pre>	<pre> variable "region" {   type = string   default = "us-east-1"   description = "the region" }  variable "ami" [   type = string   default = "ami-0453ec754f44f9a4a"   description = "the ami" ] </pre>

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

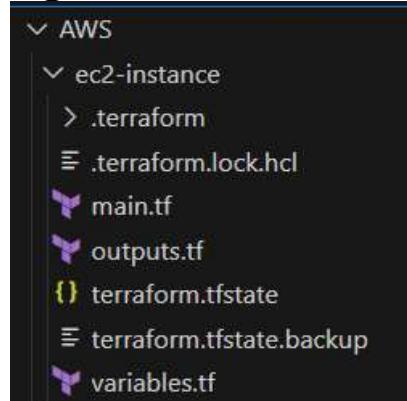
```
PS D:\Downloads\TERRAFORM\AWS\ec2-instance> []
```

## 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

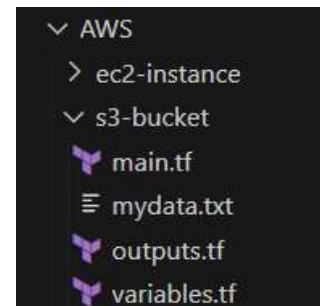
**S3 Bucket** is a logical container used to store and organize objects (files) in **Amazon Simple Storage Service (Amazon S3)**. It acts as a root directory where data is stored securely and can be accessed, managed, and retrieved.

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

```
Outputs:  
bucket_name = "trial-bucket-123-28-11-2024"  
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.*

## 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"  
        }  
    }  
}
```

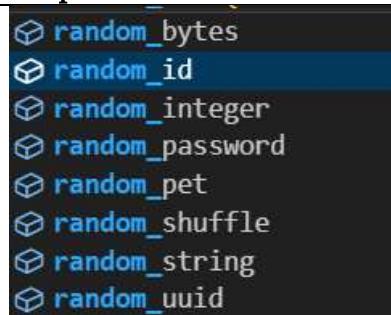
## Changes in main.tf

Without the use of random provider	With the use of random provider
<pre>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"   } }</pre>	<pre>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"   } }</pre>

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



“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

```
[?] random_id.rand_id.b64_std  
[?] random_id.rand_id.b64_url  
[?] random_id.rand_id.byte_length  
[?] random_id.rand_id.dec  
[?] random_id.rand_id.hex  
[?] random_id.rand_id.id  
[?] random_id.rand_id.prefix
```

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

*terraform init → terraform plan → terraform apply*

```
bucket_name = "trial-bucket-238583157110291692599300"  
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.

```
resource "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 .

### Explanation of 10.0.1.0/24

- An IPv4 address is 32 bits long.
- Example in binary: `10.0.1.0` -> `00001010.00000000.00000001.00000000`
  - The /24 indicates that the first 24 bits are the network portion of the address.
  - The remaining 8 bits are available for host addresses within the network.

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

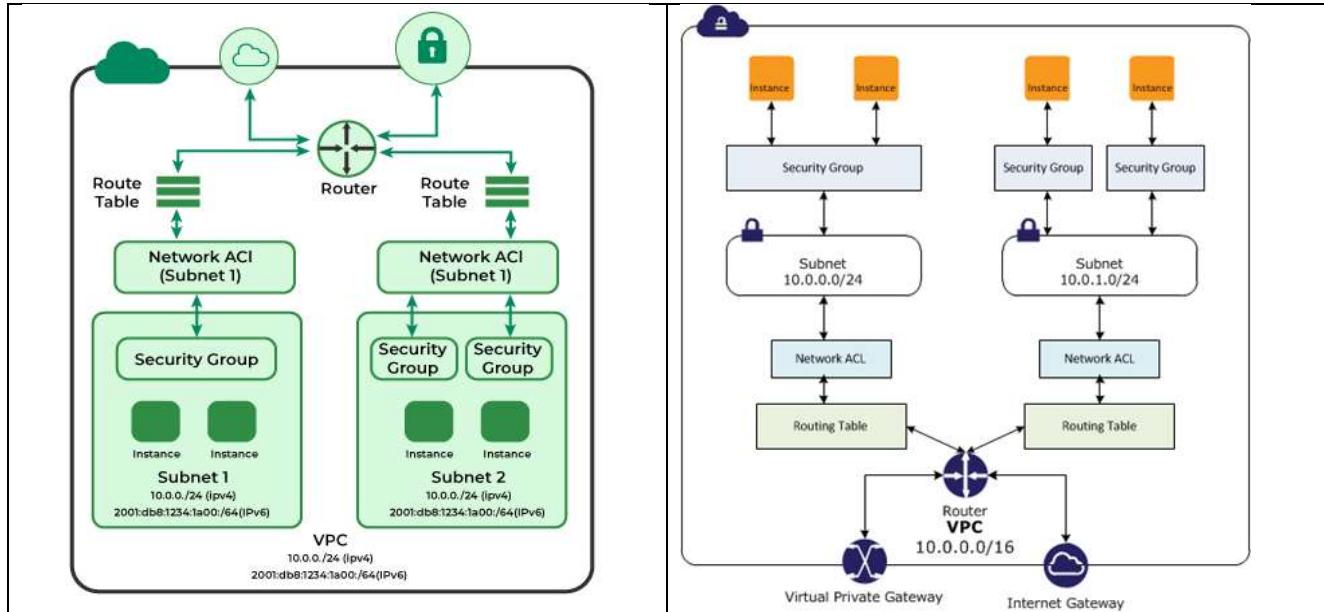
### ***NACL- NETWORK ADDRESS TRANSLATION***

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.

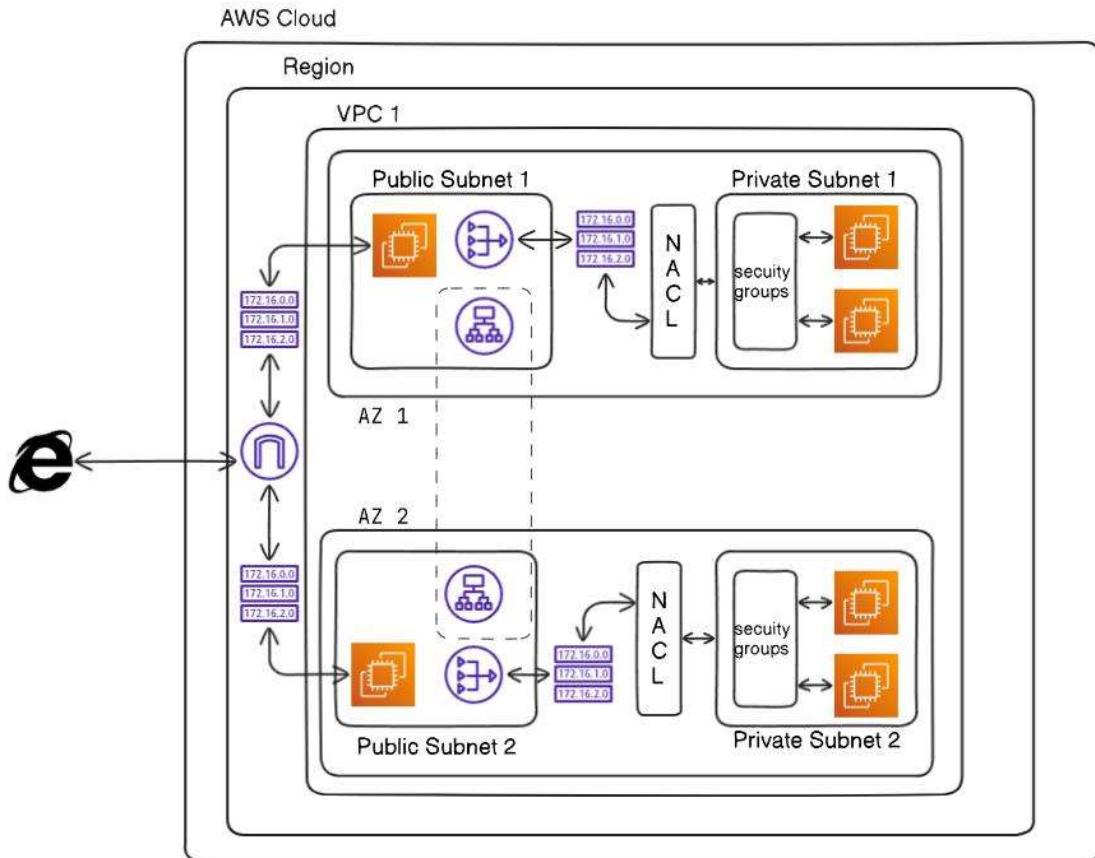
*Private subnet* → Instances cannot be connected to internet-by-internet gateway, instead they user NAT gateway.

*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:

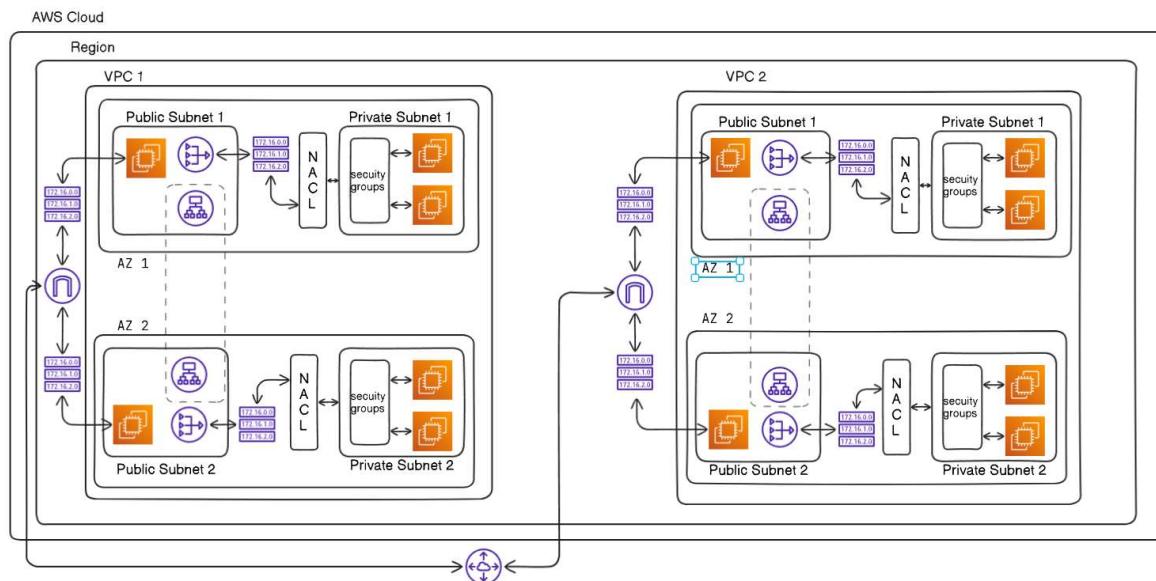
- **Destination:** The CIDR block of VPC 2 (e.g.,  $10.2.0.0/16$ ).
  - **Target:** The VPC Peering Connection ID.
- Repeat the same for the public subnets in VPC 2, adding routes to the CIDR block of VPC 1.

## 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*).

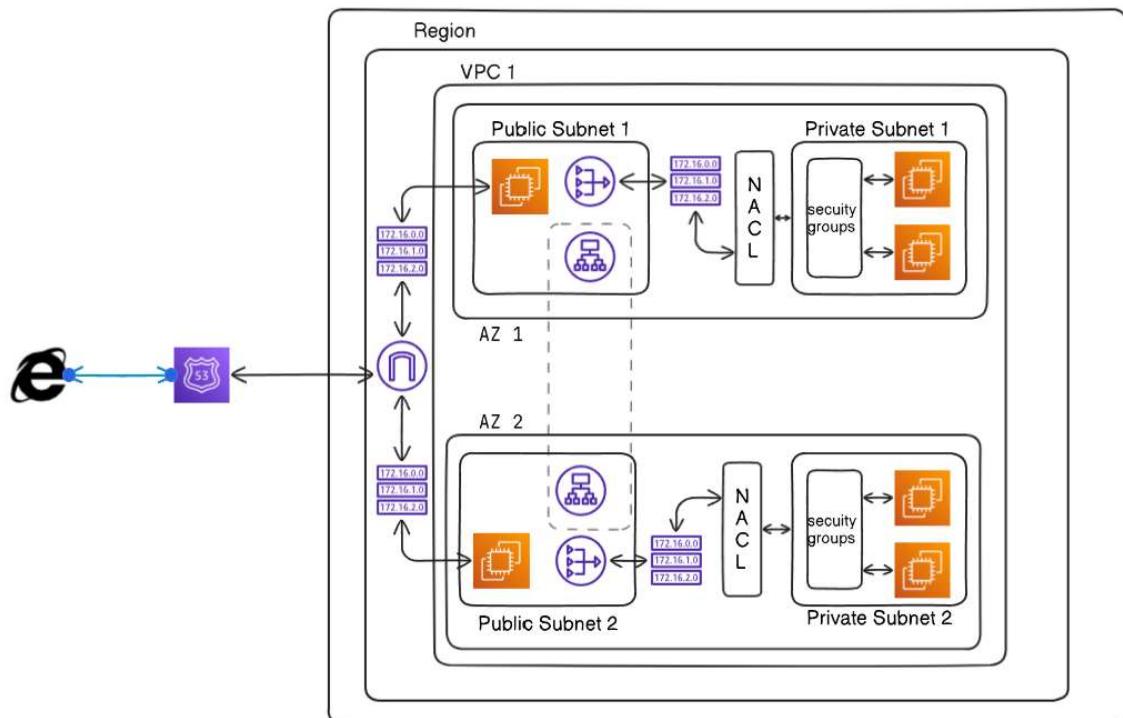
## Test Connectivity:

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



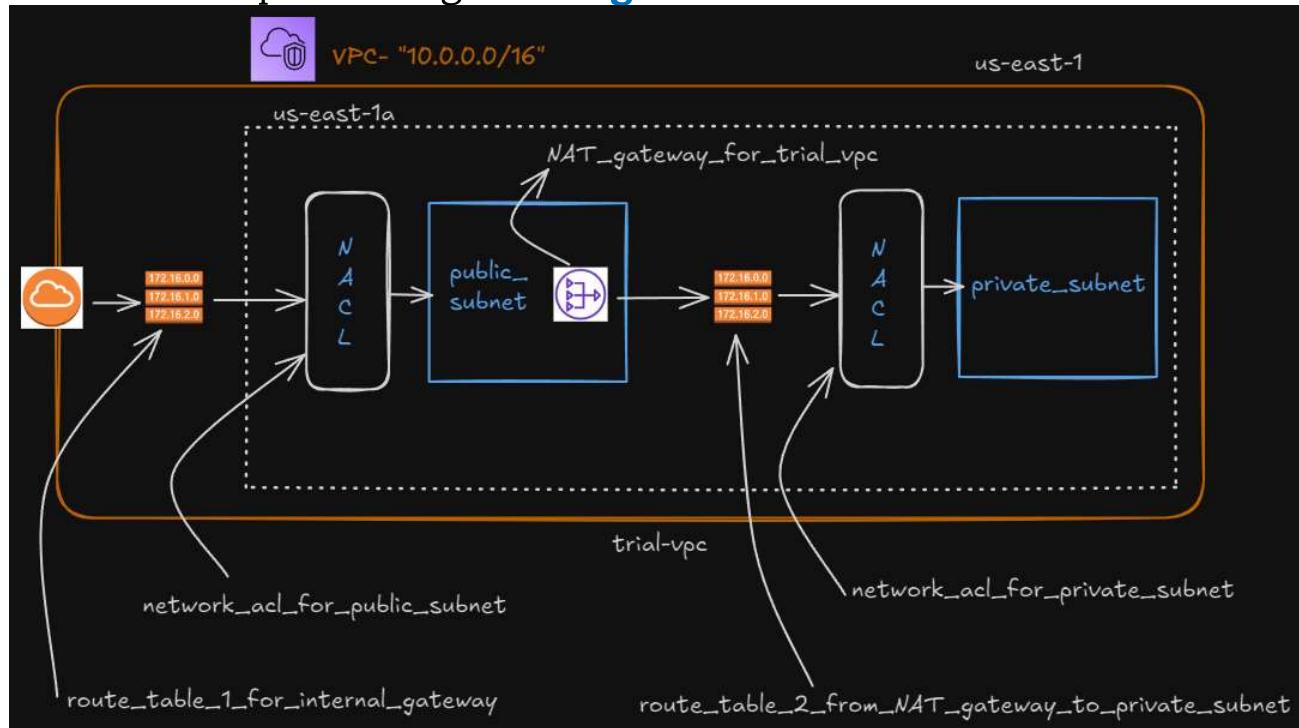
## Route53

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



## 21. IMPLEMENTING VPC USING TERRAFORM

We will be implementing this *diagram* in terraform



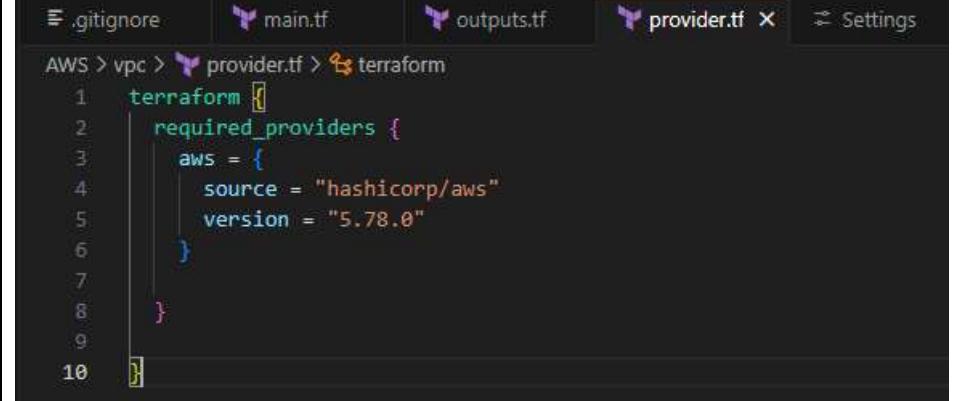
We will be using services like

- 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



```
AWS > vpc > provider.tf > terraform
1 terraform {
2   required_providers {
3     aws = {
4       source = "hashicorp/aws"
5       version = "5.78.0"
6     }
7   }
8 }
9
10 }
```

## main.tf



```
1 provider "aws" {
2   # Configuration options
3   region = "us-east-1"
4 }
5
6 resource "aws_vpc" "aws_vpc" {
7   cidr_block      = "10.0.0.0/16"
8   tags = {
9     Name = "trial-vpc"
10   }
11 }
12
13 resource "aws_internet_gateway" "internet_gateway_for_trial_vpc" {
14   vpc_id = "${aws_vpc.aws_vpc.id}"
15   tags = {
16     Name = "internet_gateway_for_trial_vpc"
17   }
18 }
19
20 resource "aws_subnet" "public_subnet_1" {
21   vpc_id          = "${aws_vpc.aws_vpc.id}"
22   cidr_block      = "10.0.1.0/24"
23   availability_zone = "us-east-1a"
24   map_public_ip_on_launch = true
25   tags = {
26     Name = "public-subnet-1"
27   }
28 }
29
30 resource "aws_subnet" "public_subnet_1" {
31   vpc_id          = "${aws_vpc.aws_vpc.id}"
32   cidr_block      = "10.0.1.0/24"
33   availability_zone = "us-east-1a"
34   map_public_ip_on_launch = true
35   tags = {
36     Name = "public-subnet-1"
37   }
38 }
```

```

30   resource "aws_subnet" "private_subnet_2" {
31     vpc_id          = "${aws_vpc.aws_vpc.id}"
32     cidr_block      = "10.0.2.0/24"
33     availability_zone = "us-east-1a"
34     map_public_ip_on_launch = false
35     tags = [
36       { Name = "private-subnet-2" }
37     ]
38   }
39
40 # Create a Default Network ACL for private subnet
41 resource "aws_network_acl" "network_acl_for_private_subnet" {
42   vpc_id = aws_vpc.aws_vpc.id
43
44   ingress {
45     rule_no    = 100
46     protocol   = "-1"
47     action     = "allow"
48     cidr_block = "0.0.0.0/0"
49     from_port  = 0
50     to_port    = 0
51   }
52
53   egress {
54     rule_no    = 100
55     protocol   = "-1"
56     action     = "allow"
57     cidr_block = "0.0.0.0/0"
58     from_port  = 0
59     to_port    = 0
60   }
61
62   tags = [
63     { Name = "network-acl-for-subnet-2" }
64   ]
65 }
66
67 # Associate Network ACL with private Subnet
68 resource "aws_network_acl_association" "acl_association_for_private_subnet" {
69   subnet_id      = aws_subnet.private_subnet_2.id
70   network_acl_id = aws_network_acl.network_acl_for_private_subnet.id
71 }
72
73 # Create a Default Network ACL for public subnet
74 resource "aws_network_acl" "network_acl_for_public_subnet" {
75   vpc_id = aws_vpc.aws_vpc.id
76
77   # Allow all inbound traffic
78   ingress {
79     rule_no    = 100
80     protocol   = "-1"
81     action     = "allow"
82     cidr_block = "0.0.0.0/0"
83     from_port  = 0
84     to_port    = 0
85   }

```

```

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
1  terraform {
2    required_providers {
3      aws = {
4        source = "hashicorp/aws"
5        version = "5.75.1"
6      }
7    }
8  }
9
10 provider "aws" {
11   # Configuration options
12   region = "us-east-1"
13 }
14
15 data "aws_ami" "name" {
16
17 }
18 }
```

```

19  output "aws_ami" {
20    value = data.aws_ami.name
21  }
22
23  resource "aws_instance" "myec2" {
24    ami = "ami-0453ec754f44f9a4a"
25    instance_type = "t2.nano"
26
27    tags = {
28      Name = "Myec2"
29    }
30  }

```

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

```

Plan: 1 to add, 0 to change, 0 to destroy.

Error: Your query returned more than one result. Please try a more specific search criteria, or set `most_recent` attribute to true.

with data.aws_ami.name,
on main.tf line 15, in data "aws_ami" "name":
15: data "aws_ami" "name" {

```

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

```

15  data "aws_ami" "name" {
16    most_recent = true
17    owners = [ "amazon" ]
18  }
19
20
21  output "aws_ami" {
22    value = data.aws_ami.name.id
23  }

```

After applying “**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”

The screenshot shows the AWS search interface with the search term 'ami' entered. The results page has a sidebar on the left with links like 'Features', 'Services', 'Resources New', 'Documentation', 'Knowledge articles', 'Marketplace', 'Blog posts', 'Events', and 'Tutorials'. The main content area displays three sections: 'Features' (with 'AMI Catalog' highlighted), 'AMIs' (with 'EC2 feature'), and 'Image Recipes' (with 'EC2 Image Builder feature'). The 'AMI Catalog' section is specifically highlighted with a pink rectangular box.

This page will appear and type “ami-0b5268083787b7af7”

The screenshot shows the 'AMI Catalog' page. At the top, there is a search bar with the placeholder 'Search for an AMI by entering a search term e.g. "Windows"'. Below the search bar, there are four filter categories: 'Quick Start AMIs (45)' (Commonly used AMIs), 'My AMIs (0)' (Created by me), 'AWS Marketplace AMIs (0)' (AWS & trusted third-party AMIs), and 'Community AMIs (0)' (Published by anyone). The 'Quick Start AMIs' category is currently selected and highlighted with a blue border. In the center, the search results are displayed under the heading 'All products (0+ filtered, 45+ unfiltered)'. A progress indicator shows 'Loading AMIs' and a button to 'Stop loading AMIs'. On the right side, there are navigation arrows for the search results.

You will see

The screenshot shows the 'AMI Catalog' page with the search term 'ami-0b5268083787b7af7' entered. The 'Community AMIs (500)' category is selected. On the left, there is a sidebar with a 'Clear all filters' button and a tree view for 'Operating system' under 'Linux/Unix'. The tree view includes nodes for All Linux/Unix, Amazon Linux, CentOS, Debian, Fedora, Gentoo, macOS, openSUSE, Other Linux, Red Hat, and SUSE Linux. On the right, a specific AMI entry for 'ubuntu-minimal/images-testing/hvm-ssd/ubuntu-focal-daily-amd64-minimal-20241204' is shown. The entry includes the AMI ID 'ami-0b5268083787b7af7', provider information ('Canonical, Ubuntu Minimal, 20.04 LTS, UNSUPPORTED daily amd64 focal image build on 2024-12-04'), and a 'Verified provider' badge. A yellow 'Select' button is located at the bottom right of the entry card.

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	STEP-2
<pre>AWS &gt; tf-data-resources &gt; main.tf &gt; ... 1  terraform { 2    required_providers { 3      aws = { 4        source  = "hashicorp/aws" 5        version = "5.75.1" 6      } 7    } 8  } 9 10 provider "aws" { 11   region = "us-east-1" 12 }</pre>	<pre>14 #ami-id 15 data "aws_ami" "name" { 16   most_recent = true 17   owners      = [ "amazon" ] 18 } 19 20 #ami-id output 21 output "aws_ami" { 22   value = data.aws_ami.name.id 23 } 24 25 # vpc-id 26 data "aws_vpc" "name" { 27   tags = { 28     Name = "trial-vpc" 29   } 30 } 31 32 #vpc-id output 33 output "aws_vpc" { 34   value = data.aws_vpc.name.id 35 } 36 }</pre>

STEP-3	STEP-4
<pre>38 #subnet-id 39 data "aws_subnet" "example" { 40   filter { 41     name  = "vpc-id" 42     values = [data.aws_vpc.name.id] 43   } 44   tags = { 45     Name = "public-subnet-1" 46   } 47 } 48 49 #subnet-id output 50 output "aws_subnet" { 51   value = data.aws_subnet.example.id 52 } 53 54 #security-group 55 data "aws_security_group" "name" { 56   tags = { 57     Name = "Security_Group_1" 58   } 59 }</pre>	<pre>61 #security-group output 62 output "aws_security_group" { 63   value = data.aws_security_group.name.id 64 } 65 66 #instance-creation 67 resource "aws_instance" "myec2" { 68   ami           = "ami-0453ec754f44f9a4a" 69   instance_type = "t2.nano" 70   subnet_id    = data.aws_subnet.example.id 71   security_groups = [data.aws_security_group.name.id] 72   tags = { 73     Name = "Myec2" 74   } 75 }</pre>

Output

Outputs:
<pre>aws_ami = "ami-0d5d11b36133682d6" aws_security_group = "sg-0488910c98264ee39" aws_subnet = "subnet-071f71057ee2b670a" aws_vpc = "vpc-0823d33924950fc64"</pre>

## 24. TERRAFORM VARIABLES

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

main.tf

```
1  terraform {
2    required_providers {
3      aws = {
4        source = "hashicorp/aws"
5        version = "5.75.1"
6      }
7    }
8  }
9
10 provider "aws" {
11   region = "us-east-1"
12 }
13
14 resource "aws_instance" "ec2_instance" {
15   ami = "ami-0453ec754f44f9a4a"
16   instance_type = "t2.micro"
17
18   root_block_device {
19     volume_size = 8
20     volume_type = "gp2"
21     delete_on_termination = true
22   }
23
24   tags = {
25     Name = "ec2-instance"
26   }
27 }
```

### 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

```
1 terraform {  
2   required_providers {  
3     aws = {  
4       source = "hashicorp/aws"  
5       version = "5.75.1"  
6     }  
7   }  
8 }  
9  
10 provider "aws" {  
11   region = "us-east-1"  
12 }  
13  
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 = 8  
20     volume_type = "gp2"  
21     delete_on_termination = true  
22   }  
23  
24   tags = {  
25     Name = "ec2-instance"  
26   }  
27 }
```

variables.tf

```
1 variable "aws_ami" {  
2   description = "value of ami-id"  
3   type = string  
4   default = "ami-0453ec754f44f9a4a"  
5 }  
6  
7 variable "aws_instance_type" {  
8   description = "Give me te instance id"  
9   type = string  
10 }
```

```
1 variable "aws_ami" {  
2   description = "value of ami-id"  
3   type = string  
4   default = "ami-0453ec754f44f9a4a"  
5 }
```

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

```
7 variable "aws_instance_type" {  
8   description = "Give me te instance id"  
9   type = string  
10 }
```

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"  
1  variable "aws_ami" {  
2    description = "value of ami-id"  
3    type = string  
4    default = "ami-0453ec754f44f9a4a"  
5  }  
6  
7  variable "aws_instance_type" {  
8    description = "Give me the instance id"  
9    type = string  
10   validation {  
11     condition = var.aws_instance_type == "t2.micro" || var.aws_instance_type == "t2.nano"  
12     error_message = "The instance type must be t2.micro or t2.nano"  
13   }  
14 }
```

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

```
6 ✓ variable "root_block_device_size" {
7   description = "Volume size"
8   type = number
9   default = 20
10  }
11
12 ✓ variable "root_block_device_type" {
13   description = "Volume type"
14   type = string
15   default = "gp2"
16 }
```

We can write

variables.tf	main.tf
<pre>✓ variable "ec2_config" {   ✓ type = object({     v_size = number     v_type = string   })   ✓ default = {     v_size = 20     v_type = "gp2"   } }</pre>	<pre>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 ]</pre>

## **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	main.tf
<pre>39   }&gt; variable "additional_tags" { 40     type = map(string) 41     default = { 42       "name" = "value" 43     } 44   }</pre>	<pre>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 = merge(var.additional_tags,{ 25       Name = "ec2-instance" 26     }) 27   }</pre>

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  
  
variable "nested_list" {  
  default = [[1, 2], [3, 4], [5, [6, 7]]]  
}  
  
output "flat_list" {  
  value = flatten(var.nested_list)  
}
```

 Copy code

### 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

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.

## ENVIRONMENT VARIABLES

Whenever you do “terraform apply” it always asks “`Give me the 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.nano`**”

## 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 " <b>region</b> " {}.	Assigns <b>region = "us-west-2"</b>

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

## variables.tf

```
1 variable "aws_ami" {
2   description = "value of ami-id"
3   type = string
4   default = "ami-0453ec754f44f9a4a"
5 }
6
7 variable "aws_instance_type" {
8   description = "Give me the instance id"
9   type = string
10  validation {
11    condition = var.aws_instance_type == "t2.micro" || var.aws_instance_type == "t2.nano"
12    error_message = "The instance type must be t2.micro or t2.nano"
13  }
14 }
15
16 variable "ec2_config" {
17   type = object{
18     v_size = number
19     v_type = string
20   })
21   default = {
22     v_size = 20
23     v_type = "gp2"
24   }
25 }
26
27 variable "additional_tags" {
28   type = map(string)
29   default = {           #can give any number of tags
30     "name" = "value"
31   }
32 }
```

## terraform.tfvars

```
1 aws_ami = "ami-0453ec754f44f9a4a"
2
3 aws_instance_type = "t2.nano"
4
5 ec2_config = {
6   v_size = 20
7   v_type = "gp2"
8 }
9
10 additional_tags = [
11   project = "terraform"
12   cloud = "aws"
13 ]
```

## 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

```
aws_ami = "ami-0453ec754f44f9a4a"

aws_instance_type = "t2.nano"

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

additional_tags = [
  project = "terraform"
  cloud = "aws"
]
```

## Terraform.auto.tfvars

```
1 ec2_config = []
2   v_size = 50
3   v_type = "gp2"
4 ]
```

Output before having  
“terraform.auto.tfvars” after  
applying “terraform plan”

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

Output after having  
“terraform.auto.tfvars” after  
applying “terraform plan”

```
+ 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.tfvars”

## Terraform Variables



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”

```
1  terraform {  
2    }  
3
```

```
4  # Number List  
5  variable "num_list" {  
6    type = list(number)  
7    default = [1,2,3]  
8  }
```

```
10 # Object List of number  
11 variable "person_list" {  
12   type = list(object({  
13     name = string  
14     age = number  
15   }))  
16 }
```

```
10 # Object List of number  
11 variable "person_list" {  
12   type = list(object({  
13     name = string  
14     age = number  
15   }))  
16  
17   default = [  
18     {  
19       name = "John"  
20       age = 30  
21     },  
22     {  
23       name = "Jane"  
24       age = 25  
25     },  
26     {  
27       name = "Jony"  
28       age = 40  
29     }  
30   ]  
31 }
```

```

33 # map, based on key and value pair
34 # jist like dictionary in python
35 variable "map_list" {
36   type = map(number)
37   default = {
38     "name" = 1
39     "age" = 2
40   }
41 }

45 #calculation
46 locals {
47   mul = 2 * 2
48   add = 2 + 2
49   sub = 2 - 2
50   div = 2 / 2
51   not_equal = 2 != 3
52
53   #to get the double
54   double= [for i in var.num_list : i * 2]
55
56   #to get the odd number
57   odd= [for i in var.num_list : i if i % 2 != 0]
58
59   #to get the names of the person from the above map list
60   name = [for i in var.person_list : i.name]
61
62   # work with map
63   map_info= [for key, value in var.map_list : value]
64
65   #double map
66   double_map_info= [for key, value in var.map_list : value * 2]
67 }
```

```
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 }
```

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

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,
```

## 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”

```
1  terraform {  
2  
3    }
```

```
5  locals {  
6    value = "hello , world"  
7  }  
9  # convert the string to upper case that is present in the local.value  
10 output "output_1" {  
11   value = upper(local.value)  
12 }
```

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

```
5  locals {  
6    value = "hello , world"  
7  }  
14 # convert the string to lower case that is present in the local.value  
15 output "output_2" {  
16   value = lower(local.value)  
17 }
```

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

```
5  locals {  
6    value = "hello , world"  
7  }  
19 # returns boolean value true, if local.value starts with hello  
20 output "output_3" {  
21   value = startswith(local.value, "hello")  
22 }
```

```
+ output_3  = true
```

```
5  locals {  
6    value = "hello , world"  
7  }
```

```
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
```

```
5  locals {
6  | value = "hello , world"
7  }
50 # returns the minimum value
51 output "output_9" {
52 |   value = min(1,8,2,3,9,0,10)
53 }
```

```
+ output_9 = 0
```

```
56 variable "string_list" {
57   type = list(string)
58   default = [ "server1", "server2", "server3", "server1" ]
59 }
61 # returns the length of the list
62 output "output_10" {
63 |   value = length(var.string_list)
64 }
```

```
+ output_10 = 4
```

```
56 variable "string_list" {
57   type = list(string)
58   default = [ "server1", "server2", "server3", "server1" ]
59 }
66 # joins the elements of the list with -
67 output "output_11" {
68 |   value = join("-", var.string_list)
69 }
```

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

```
56 variable "string_list" {
57   type = list(string)
58   default = [ "server1", "server2", "server3", "server1" ]
59 }
71 # returns true if the string is present in the list
72 output "output_12" {
73 |   value = contains(var.string_list, "server1")
74 }
```

```
+ output_12 = true
```

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

```
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",
]
]
```

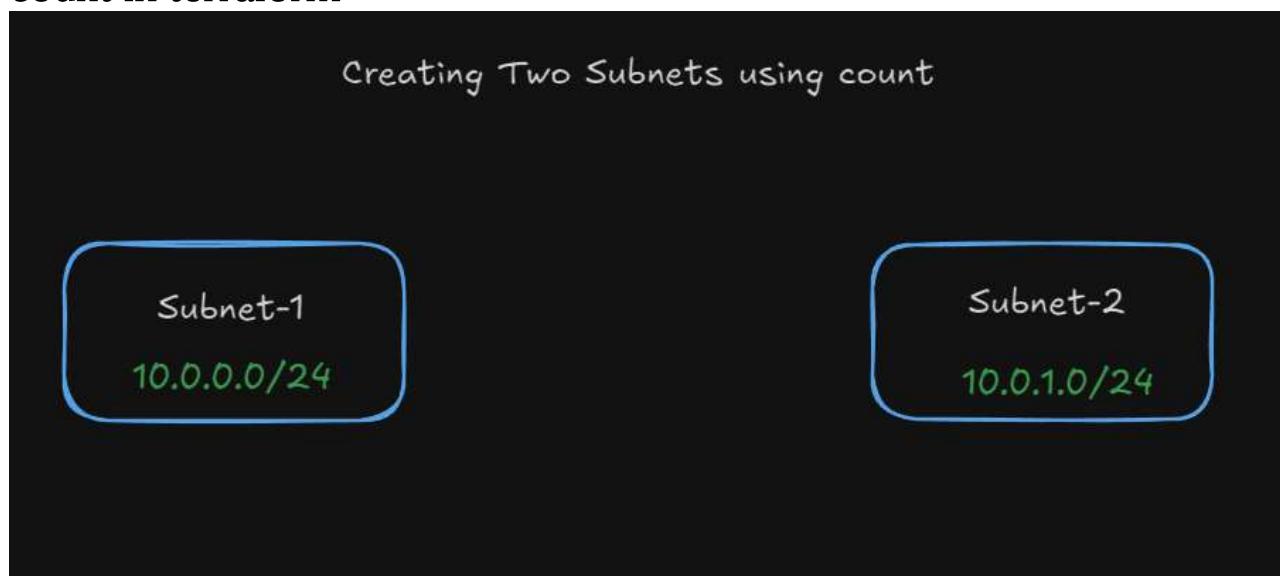
## 28. TERRAFORM: MULTIPLE RESOURCES

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



In order to create this demo we will create a new folder name “tf-multiple-resources” in AWS folder inside that we will be creating “main.tf” in “main.tf” file there are subdivisions

### main.tf

```
1 #required provider
2 ✓ terraform {
3   ✓ required_providers {
4     ✓ aws = {
5       source = "hashicorp/aws"
6       version = "5.75.1"
7     }
8   }
9 }
```

```

11 #provider
12 provider "aws" {
13   region = "us-east-1"
14 }

16 #locals variables
17 locals {
18   project_name = "tf_vpc"
19 }

21 #creating resource for vpc
22 resource "aws_vpc" "tf_vpc" {
23   cidr_block = "10.0.0.0/16"
24   tags = {
25     Name = "${local.project_name}"
26   }
27 }
```

```
#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.

```

39 #output of one subnet id
40 output "subnet_ids" {
41   value = aws_subnet.tf_vpc_subnet[0].id
42 }
43
44 #output of all subnet ids
45 output "all_subnet_ids" {
46   value = aws_subnet.tf_vpc_subnet[*].id
47 }

```

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

Creating 2 Subnets & 4 instances using count



Continuations of above infrastructure of the demo 1 project

```
39 #creating resources of instances for subnets
40 ✓ resource "aws_instance" "ec2_instance" {
41   count = 4
42   ami= "ami-0453ec754f44f9a4a"
43   instance_type = "t2.micro"
44   subnet_id = element(aws_subnet.tf_vpc_subnet[*].id, count.index % length(aws_subnet.tf_vpc_subnet))
45   # 0 % 2 = 0
46   # 1 % 2 = 1
47   # 2 % 2 = 0
48   # 3 % 2 = 1
49   ✓ tags = {
50     | Name = "${local.project_name}-instance-${count.index}"
51   }
52 }
```

```
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-03428ccce23dd1d36]
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.**

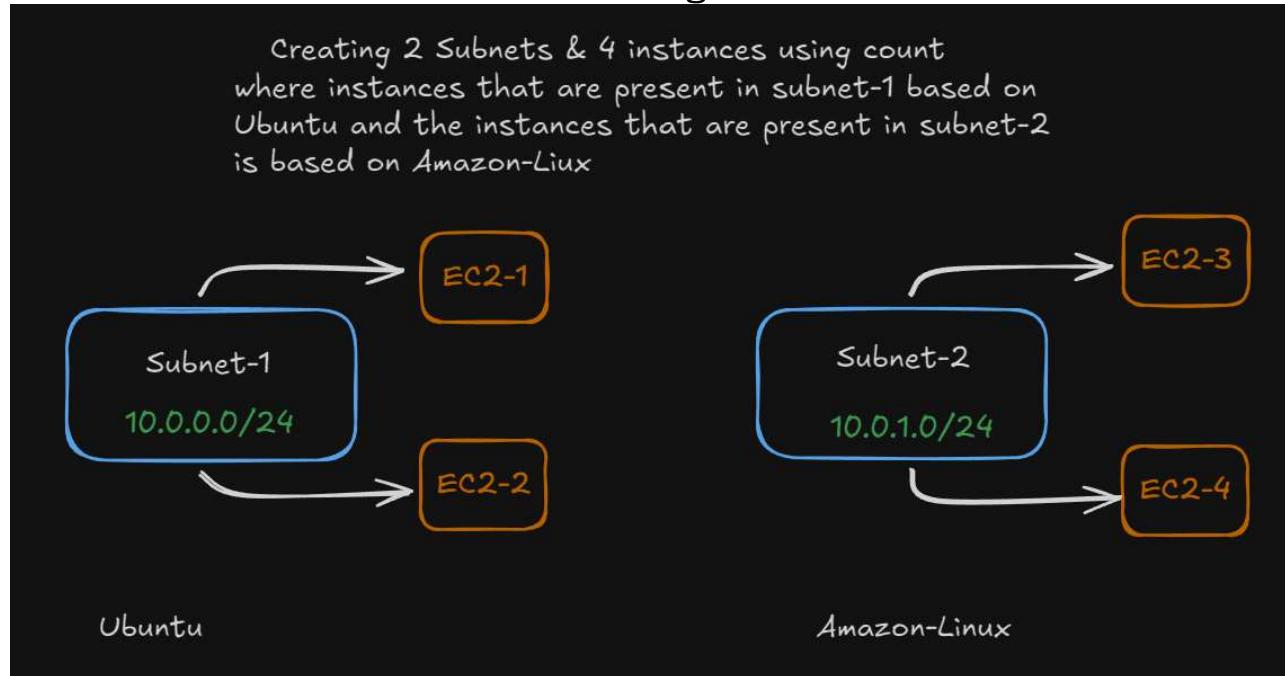
**Outputs:**

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

Now go to aws console to see the result

<input type="checkbox"/>	Name 	Subnet IDs
<input type="checkbox"/>	tf_vpc-instance-0	subnet-01152e2f4be905bfc
<input type="checkbox"/>	tf_vpc-instance-1	subnet-0acb21115d40a4b6a
<input type="checkbox"/>	tf_vpc-instance-2	subnet-01152e2f4be905bfc
<input type="checkbox"/>	tf_vpc-instance-3	subnet-0acb21115d40a4b6a

For the third demo we will be creating



First create **variables.tf**

```
1 variable "ec2_instance_ami" [
2   type = list(object({
3     ami = string
4   }))
5
6 ]
```

Second create **terraform.tfvars**

```
1 ec2_instance_ami = []
2   ami = "ami-0453ec754f44f9a4a" #amazon-linux
3   ,{
4     ami = "ami-0e2c8caa4b6378d8c" #ubuntu
5   }
6 ]
```

Third continuations of above infrastructure of the demo 2 project but slight in **resource "aws\_instance"**

```

39  #creating resources of instances for subnets
40  resource "aws_instance" "ec2_instance" {
41    count = 4
42    ami= var.ec2_instance_ami[count.index % length(aws_subnet.tf_vpc_subnet)].ami
43    instance_type = "t2.micro"
44    subnet_id = element(aws_subnet.tf_vpc_subnet[*].id, count.index % length(aws_subnet.tf_vpc_subnet))
45    # 0 % 2 = 0
46    # 1 % 2 = 1
47    # 2 % 2 = 0
48    # 3 % 2 = 1
49    tags = {
50      Name = "${local.project_name}-instance-${count.index}"
51    }
52  }

```

## 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

```

8   variable "ec2_instance_ami_map" {
9     # key-value pair
10    type = map(object({
11      ami = string
12    }))
13  }

```

#### terraform.tfvars

```

8 ec2_instance_ami_map = {
9   "amazon-1" = {
10     ami = "ami-0453ec754f44f9a4a" #amazon-linux
11   },
12
13   "ubuntu-1" = {
14     ami = "ami-0e2c8caa4b6378d8c" #ubuntu
15   }
16
17   "amazon-2" = {
18     ami = "ami-0453ec754f44f9a4a" #amazon-linux
19   },
20
21   "ubuntu-2" = {
22     ami = "ami-0e2c8caa4b6378d8c" #ubuntu
23   }
24 }

```

## main.tf

```
39 #creating resources of instances for subnets
40 ✓ resource "aws_instance" "ec2_instance" {
41   for_each = var.ec2_instance_ami_map
42   # we will get each.key and each.value from the map
43   ami= each.value.ami
44   instance_type = "t2.micro"
45   subnet_id = element(aws_subnet.tf_vpc_subnet[*].id,index(keys(var.ec2_instance_ami_map),each.key) % length(aws_subnet.tf_vpc_subnet))
46
47 ✓ tags = {
48   | Name = "${local.project_name}-instance-${each.key}"
49 }
50 }
```

```
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

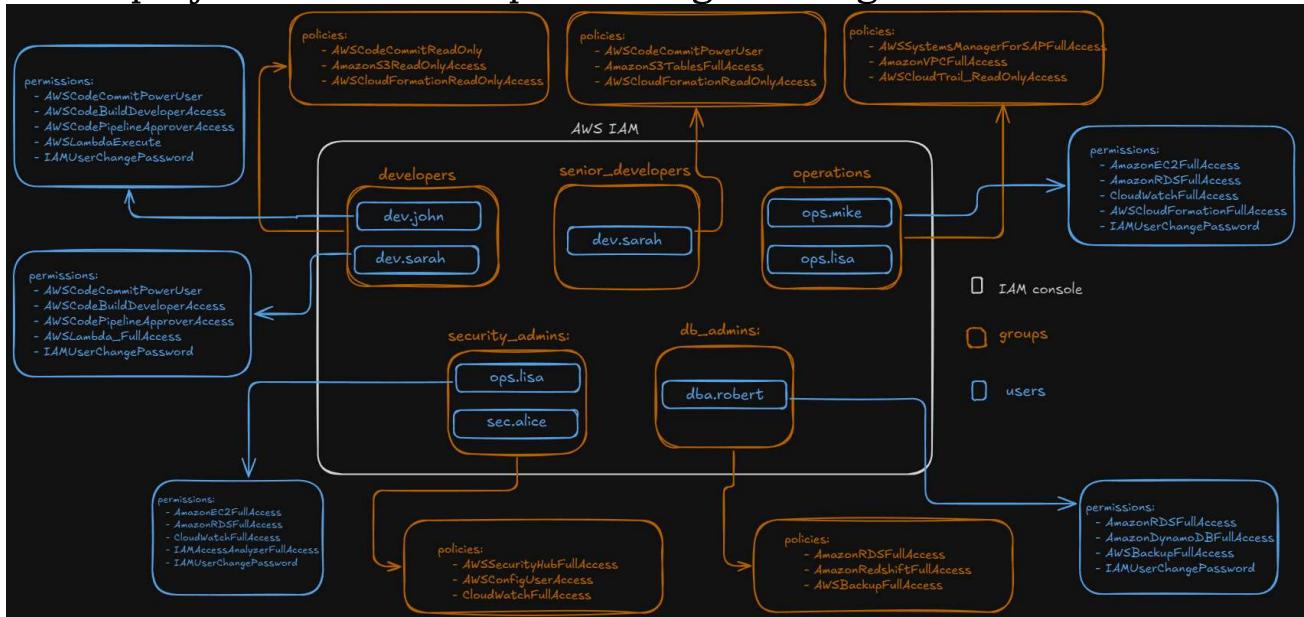
In this project you must have the knowledge of

- 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
- Attach 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
 1  # IAM Users and Roles Configuration
 2
 3  # Developer Team
 4  developers:
 5    - username: dev.john
 6      full_name: "John Smith"
 7      groups:
 8        - developers
 9      permissions:
10        - AWSCodeCommitPowerUser
11        - AWSCodeBuildDeveloperAccess
12        - AWSCodePipelineApproverAccess
13        - AWSLambdaExecute
14        - IAMUserChangePassword
15
16    - username: dev.sarah
17      full_name: "Sarah Johnson"
18      groups:
19        - developers
20        - senior_developers
21      permissions:
22        - AWSCodeCommitPowerUser
23        - AWSCodeBuildDeveloperAccess
24        - AWSCodePipelineApproverAccess
25        - AWSLambda_FullAccess
26        - IAMUserChangePassword
27
28  # Operations Team
29  operations:
30    - username: ops.mike
31      full_name: "Mike Anderson"
32      groups:
33        - operations
34      permissions:
35        - AmazonEC2FullAccess
36        - AmazonRDSFullAccess
37        - CloudWatchFullAccess
38        - AWSCloudFormationFullAccess
39        - IAMUserChangePassword
40
41    - username: ops.lisa
42      full_name: "Lisa Chen"
43      groups:
44        - operations
45        - security_admins
46      permissions:
47        - AmazonEC2FullAccess
48        - AmazonRDSFullAccess
49        - CloudWatchFullAccess
50        - IAMAccessAnalyzerFullAccess
51        - IAMUserChangePassword
52

```

### STEP-2

```

53  # Database Administrators
54  db_admins:
55    - username: dba.robert
56      full_name: "Robert Wilson"
57      groups:
58        - db_admins
59      permissions:
60        - AmazonRDSFullAccess
61        - AmazonDynamoDBFullAccess
62        - AWSBackupFullAccess
63        - IAMUserChangePassword
64
65  # Security Team
66  security:
67    - username: sec.alice
68      full_name: "Alice Brown"
69      groups:
70        - security_admins
71      permissions:
72        - IAMFullAccess
73        - SecurityAudit
74        - AWSSecurityHubFullAccess
75        - AWSConfigUserAccess
76        - IAMUserChangePassword
77
78  # Groups Configuration
79  groups:
80    developers:
81      policies:
82        - AWSCodeCommitReadOnly
83        - AmazonS3ReadOnlyAccess
84        - AWSCloudFormationReadOnlyAccess
85
86    senior_developers:
87      policies:
88        - AWSCodeCommitPowerUser
89        - AmazonS3TablesFullAccess
90        - AWSCloudFormationReadOnlyAccess
91
92    operations:
93      policies:
94        - AWSSystemsManagerForSAPFullAccess
95        - AmazonVPCFullAccess
96        - AWSCloudTrail_ReadOnlyAccess
97
98    security_admins:
99      policies:
100        - AWSSecurityHubFullAccess
101        - AWSConfigUserAccess
102        - CloudWatchFullAccess
103
104    db_admins:
105      policies:
106        - AmazonRDSFullAccess
107        - AmazonRedshiftFullAccess
108        - AWSBackupFullAccess
109

```

### 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

```
{
```

```
21 | #users = yamldecode(file("${path.module}/users.yaml"))
22 | 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
```

```
...
```

```
23 | Ctrl+L to chat, Ctrl+K to generate
24 | 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 = [
+ ...
+ ]
```

```
...
```

```
26 | # Get all user names across different teams
27 | all_users_names= toset(concat(
28 |   local.iml_data_decoded.developers[*].username,
29 |   local.iml_data_decoded.operations[*].username,
30 |   local.iml_data_decoded.db_admins[*].username,
31 |   local.iml_data_decoded.security[*].username
32 | ))
```

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

```
34 | # To get all the group names
35 | all_groups_name=toset(flatten([
36 |   yamldecode(local.iml_data).developers[*].groups,
37 |   yamldecode(local.iml_data).operations[*].groups,
38 |   yamldecode(local.iml_data).db_admins[*].groups,
39 |   yamldecode(local.iml_data).security[*].groups
40 | ]))
```

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

```

42 | # List of all teams from YAML
43 | teams = [
44 |     local.impl_data_decoded.developers,
45 |     local.impl_data_decoded.operations,
46 |     local.impl_data_decoded.db_admins,
47 |     local.impl_data_decoded.security
48 | ]

```

```

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

```

```

49 | # Extract users and their groups as pairs
50 | pair_users_groups = {
51 |     for pair in flatten([
52 |         for team in local.teams : [
53 |             for user in team : [
54 |                 for group in user.groups : {
55 |                     username = user.username,
56 |                     group = group
57 |                 }
58 |             ]
59 |         ]
60 |     ]) : "${pair.username}-${pair.group}" => pair
61 | }

```

```

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

```

```

64 | # Create pairs of users and their directly assigned permissions
65 | pair_user_permission= {
66 |     for pair in flatten([
67 |         for team in local.teams:[
68 |             for user in team:[
69 |                 for permission in user.permissions :{
70 |                     username = user.username,
71 |                     permission = permission
72 |                 }
73 |             ]
74 |         ]
75 |     ]) : "${pair.username}-${pair.permission}" => pair
76 | }

```

```

+ 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"
}
...

```

```

78 | # Get groups and their policies from YAML
79 | groups_policies = local.impl_data_decoded.groups

```

```

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

```

```

81 # Create pairs of groups and their assigned policies
82 pair_group_policy = {
83     for pair in flatten([
84         for group_name, group_obj in local.groups_policies : [
85             for policy in lookup(group_obj, "policies", []) : {
86                 group_name = group_name
87                 policy     = policy
88             }
89         ]
90     ]) : "${pair.group_name}-${pair.policy}" => pair
91 }

```

```

+ pairing_policies_with_group = {
+   db_admins-AWSBackupFullAccess      = {
+     group_name = "db_admins"
+     policy    = "AWSBackupFullAccess"
+   }
+   db_admins-AmazonRDSFullAccess     = {
+     group_name = "db_admins"
+     policy    = "AmazonRDSFullAccess"
+   }
}
...
```

```

93 # Get password policy configuration from YAML
94 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**

```

90 resource "aws_iam_user" "users" {
91     for_each = local.all_users_names
92     name    = each.value
93 }

```

### **# Generate AWS access keys for all users**

```

96 resource "aws_iam_access_key" "user_keys" {
97     for_each = local.all_users_names
98     user    = each.key
99 }

```

### **# Set up console access for users with initial password requirements**

```

102 resource "aws_iam_user_login_profile" "user_login_profile" {
103     for_each = local.all_users_names
104     user    = each.key
105     password_length = 20
106     password_reset_required = true
107 }

```

## # Configure organization-wide password policy settings

```
110 resource "aws_iam_account_password_policy" "password_policy" {
111   minimum_password_length      = local.password_policy.minimum_length
112   require_uppercase_characters = local.password_policy.require_uppercase
113   require_lowercase_characters = local.password_policy.require_lowercase
114   require_numbers              = local.password_policy.require_numbers
115   require_symbols              = local.password_policy.require_symbols
116   max_password_age             = local.password_policy.max_age_days
117   password_reuse_prevention    = local.password_policy.prevent_reuse
118
119   # Optional: Allow users to change their own password
120   allow_users_to_change_password = true
121 }
```

## # Create IAM groups for different team roles

```
132 ✓ resource "aws_iam_group" "groups" {
133   for_each = local.all_groups_name
134   name     = each.value
135 }
```

## # Assign users to their respective IAM groups

```
135 resource "aws_iam_user_group_membership" "pairing_users_groups" {
136   for_each = local.pair_users_groups
137
138   user    = each.value.username
139   groups  = [each.value.group]
140 }
```

## # Attach AWS managed policies to IAM groups

```
143 ✓ resource "aws_iam_group_policy_attachment" "group_policy_attachment" {
144   for_each = local.pair_group_policy
145   group    = each.value.group_name
146   policy_arn = "arn:aws:iam::aws:policy/${each.value.policy}"
147 }
```

## # Attach AWS managed policies directly to users

```
150 ✓ resource "aws_iam_user_policy_attachment" "user_policy_attachment" {
151   for_each = local.pair_user_permission
152   user     = each.value.username
153   policy_arn = "arn:aws:iam::aws:policy/${each.value.permission}"
154 }
```

## In “*output.tf*”

```
Pieces: Comment | Pieces: Explain
1 output "iml-data-without-decoding" {
2   value = local.iml_data
3 }
4
5 Pieces: Comment | Pieces: Explain
6 output "iml-data-with-decoding" {
7   value = local.iml_data_decoded
8 }
9
10 Pieces: Comment | Pieces: Explain
11 output "users-names" {
12   value = local.all_users_names
13 }
14
15 Pieces: Comment | Pieces: Explain
16 output "groups-names" {
17   value = local.all_groups_name
18 }
19
20 Pieces: Comment | Pieces: Explain
21 output "teams-names" {
22   value = local.teams
23 }
24
25 Pieces: Comment | Pieces: Explain
26 output "policy-names" {
27   value = toset(flatten([
28     yamldecode(local.iml_data).developers[*].permissions,
29     yamldecode(local.iml_data).operations[*].permissions,
30     yamldecode(local.iml_data).db_admins[*].permissions,
31     yamldecode(local.iml_data).security[*].permissions
32   ]))
33 }
34
35 Pieces: Comment | Pieces: Explain
36 output "pairing-users-with-groups" {
37   value = local.pair_users_groups
38 }
```

```

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 name	Path	Group	Last activity	MFA	Password age	Console last sign-in	Account
dba.robert	/	1	-	-	3 minutes	-	Act
dev.john	/	1	-	-	3 minutes	-	Act
dev.sarah	/	2	-	-	3 minutes	-	Act
[REDACTED]	/	0	2023-07-10 14:11:22	-	-	-	Act
ops.lisa	/	2	-	-	3 minutes	-	Act
ops.mike	/	1	-	-	3 minutes	-	Act
[REDACTED]	/	0	2023-07-10 14:11:22	-	-	-	Act
sec.alice	/	1	-	-	3 minutes	-	Act
[REDACTED]	/	0	2023-07-10 14:11:22	-	-	-	Act

## User group console

User groups (6) <small>Info</small>			
A user group is a collection of IAM users. Use groups to specify permissions for a collection of users.			
<input type="text"/> Search			
Group name	Users	Permissions	Creation time
<a href="#">admins</a>	1	Defined	1 month ago
<a href="#">db_admins</a>	1	Defined	8 minutes ago
<a href="#">developers</a>	2	Defined	8 minutes ago
<a href="#">operations</a>	2	Defined	8 minutes ago
<a href="#">security_admins</a>	2	Defined	8 minutes ago
<a href="#">senior_developers</a>	1	Defined	8 minutes ago

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

```
$ tree minimal-module/
.
├── README.md
└── main.tf
    ├── variables.tf
    └── outputs.tf
```

#### 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.

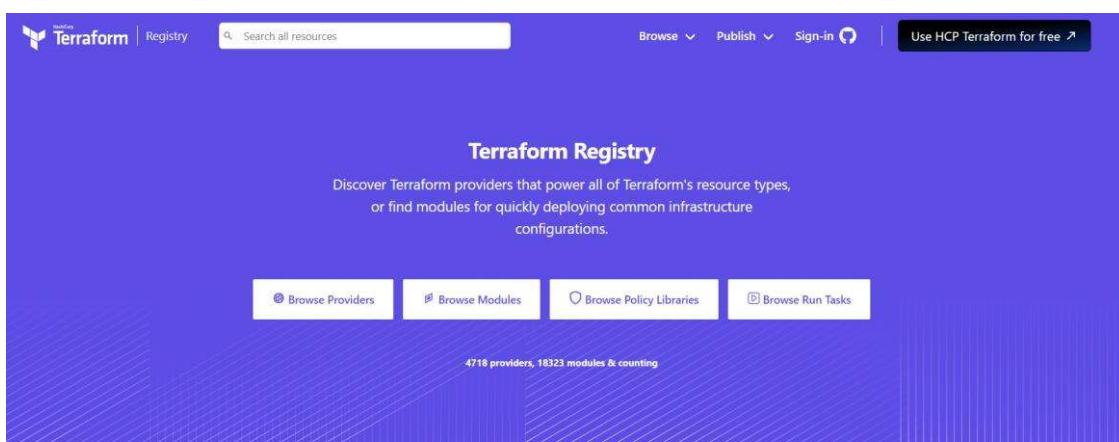
## **outputs.tf:**

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:

**Terraform Registry**

Search all resources

Browse ▾ Publish ▾ Sign-in

Use HCP Terraform for free ↗

Providers Modules Policy Libraries Run Tasks

Filters Clear Filters

**Tier**

Partner Display only modules that are maintained by a Hashicorp partner.

**Provider**

- Alibaba
-  AWS
-  Azure
-  Boundary
-  Consul
-  Google
-  Helm
-  Nomad
-  Oracle
-  Vault

## Modules

Modules are self-contained packages of Terraform configurations that are managed as a group.

 **terraform-aws-modules / iam**  
Terraform module to create AWS IAM resources 

 **cloudposse / label**  
Terraform Module to define a consistent naming convention by [namespace, stage, name, [attributes]] 

 **terraform-aws-modules / vpc**  
Terraform module to create AWS VPC resources 

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.

Providers Modules Policy Libraries Run Tasks

Filters Clear Filters

**Tier**

Partner Display only modules that are maintained by a Hashicorp partner.

**Provider**

- Alibaba
-  AWS
- Azure
- Boundary
- Consul
- Google
- Helm
- Nomad
- Oracle
- Vault

## Modules

Modules are self-contained packages of Terraform configurations that are managed as a group.

 **terraform-aws-modules / iam**  
Terraform module to create AWS IAM resources 

 **terraform-aws-modules / vpc**  
Terraform module to create AWS VPC resources 

 **terraform-aws-modules / s3-bucket**  
Terraform module to create AWS S3 resources 

## 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:

vpc  
aws provider

Terraform module to create AWS VPC resources

Published December 19, 2024 by [terraform-aws-modules](#)

Module managed by [antonbabenko](#)

Source Code: [github.com/terraform-aws-modules/terraform-aws-vpc](https://github.com/terraform-aws-modules/terraform-aws-vpc) (report an issue)

Version 5.17.0 (latest) All versions ▾

Module Downloads

- Downloads this week 998,164
- Downloads this month 2.9M
- Downloads this year 41.5M
- Downloads over all time 103.0M

Submodules Examples

**Provision Instructions**

Copy and paste into your Terraform configuration, insert the variables, and run `terraform init`:

```
module "vpc" {
  source  = "terraform-aws-modules/vpc"
  version = "5.17.0"
}
```

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

```
tf-module-vpc
└── compute.tf
```

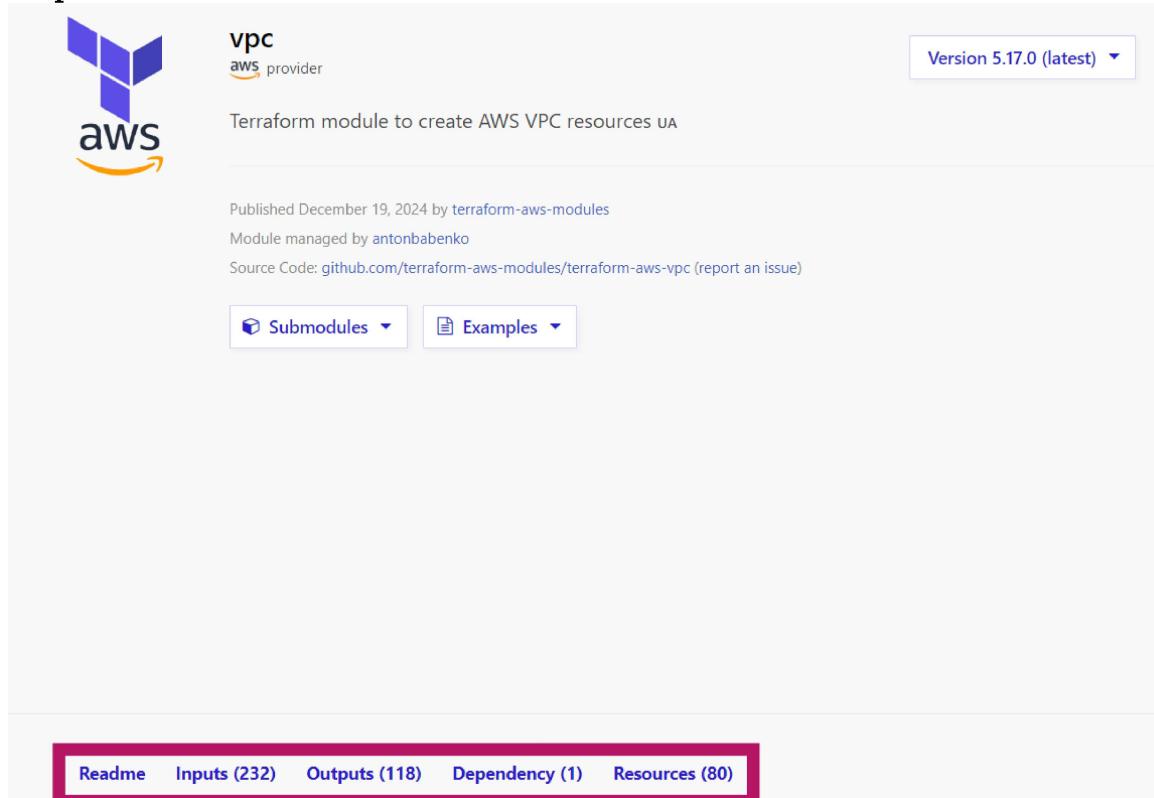
```
AWS > tf-module-vpc > compute.tf > module "vpc"
  Pieces: Comment | Pieces: Explain
1  module "vpc" {
2    source  = "terraform-aws-modules/vpc/aws"
3    version = "5.17.0"
4 }
```

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

```
tf-module-vpc
├── .terraform
├── modules
│   └── vpc
│       ├── .github
│       ├── examples
│       ├── modules
│       │   └── .editorconfig
│       │   └── .gitignore
│       └── .pre-commit-config.yaml
└── CHANGELOG.md
    └── LICENSE
    └── main.tf
    └── outputs.tf
    └── README.md
    └── UPGRADE-3.0.md
    └── UPGRADE-4.0.md
    └── variables.tf
    └── versions.tf
    └── vpc-flow-logs.tf
    └── modules.json
└── providers\registry.terraform.io\has...
    └── LICENSE.txt
    └── terraform-provider-aws_v5.82.2_x5...
    └── .terraform.lock.hcl
```

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.



In 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.

```

1 provider "aws" {
2   region = "us-east-1"
3 }
4 Pieces: Comment | Pieces: Explain
5 module "vpc" {
6   source  = "terraform-aws-modules/vpc/aws"
7   version = "5.17.0"

```

For further configuration do follow readme file of the module

### Final configuration of “compute.tf”

```

1 Pieces: Comment | Pieces: Explain
2 provider "aws" {
3   region = "us-east-1"
4 }
5 Pieces: Comment | Pieces: Explain
6 module "vpc" {
7   source  = "terraform-aws-modules/vpc/aws"
8   version = "5.17.0"
9
10  name = "my-vpc"
11  cidr = "10.0.0.0/16"
12
13  azs      = ["us-east-1a"]
14  private_subnets = ["10.0.1.0/24"]
15  public_subnets  = ["10.0.101.0/24"]
16
17  tags = {
18    Terraform = "true"
19    Environment = "dev"
20  }
21 }
22

```

### 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**”

```
1 module "ec2-instance" {  
2     source  = "terraform-aws-modules/ec2-instance/aws"  
3     version = "5.7.1"  
4  
5     name = "single-instance"  
6  
7     ami = "ami-0453ec754f44f9a4a"  
8  
9     instance_type          = "t2.micro"  
10    vpc_security_group_ids = [module.vpc.default_security_group_id]  
11    subnet_id              = module.vpc.public_subnets[0]  
12  
13    tags = {  
14        Name   = "trial-instance"  
15        Environment = "dev"  
16    }  
17  
18}
```

### *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

```
VPC ID
└─ vpc-04587757c585bbcae (my-vpc) ↗

Subnet ID
└─
    subnet-00b2649b5038a1c3b (my-vpc-public-us-east-1a) ↗
```

*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:

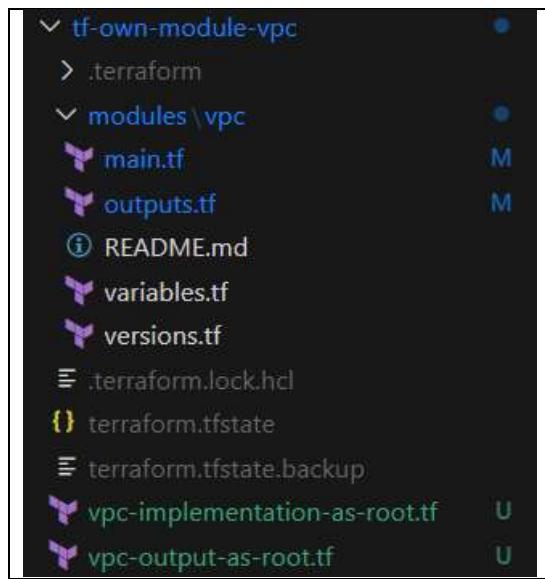
- Accept cidr\_block from user to create VPC
- User can create multiple subnets
  - Get CIDR block for subnets from user
  - Get AZS (availability zone)
  - User can mark a subnet as public (default is private)
    - If public, create IGW
    - Associate public subnet with Routing table

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/vhc/version.tf

```
1  terraform {
2      required_version = ">= 1.0.0"
3
4      provider {
5          object = {
6              source  = "hashicorp/aws"
7              version = ">= 4.0.0"
8          }
9      }
10 }
```

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

```
1  variable "vpc_config" {
2      description = "To get the CIDR and the name of the vpc from user"
3      type = object({
4          cidr_block = string
5          vpc_name = string
6      })
7
8      validation {
9          condition= can(cidrnetmask(var.vpc_config.cidr_block))
10         error_message = "Each cidr_block must be a valid CIDR format."
11     }
12 }
13 }
```

```

14 Piecs: Comment | Piecs: Explain
15 variable "aws_subnets" {
16     # sub1={cidr_block, availability_zone}, sub2={}, sub3={}
17     description = "To get the CIDR and availability zone from user"
18     type = map(object({
19         cidr_block = string
20         availability_zone = string |
21         public = optional(bool, false)
22     }))
23
24 Piecs: Comment | Piecs: Explain
25 validation {
26     #sub1={cidr=...}, sub2={cidr=...}, sub3={cidr=...} , [true, true, true]
27     condition= alltrue([for config in var.aws_subnets : can(cidrnetmask(config.cidr_block))])
28     error_message = "Each cidr_block must be a valid CIDR format."
}

```

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

```

1  ✓ resource "aws_vpc" "vpc-main" {
2      cidr_block = var.vpc_config.cidr_block
3
4  ✓   tags = {
5      | Name = var.vpc_config.vpc_name
6  }
7
8
9  Piecs: Comment | Piecs: Explain
10 resource "aws_subnet" "subnets_main" {
11     vpc_id = aws_vpc.vpc-main.id
12     for_each = var.aws_subnets
13
14     cidr_block = each.value.cidr_block
15     availability_zone = each.value.availability_zone
16
17     tags = {
18         | Name = each.key
19     }
20
21 Piecs: Comment | Piecs: Explain
22 ✓ locals {
23     public_subnets = {
24         #public_sub1 = {cidr_block = "10.0.1.0/24", availability_zone = "us-east-1a", public = true}
25         for key, config  in var.aws_subnets : key => config if config.public
26     }
27
28     private_subnets = {
29         for key, config  in var.aws_subnets : key => config if !config.public
30     }
}

```

```

31
32  #internet gateway, if there is atleast one public subnet
33  #internet gateway will be created only once
34  Pieces: Comment | Pieces: Explain
35  resource "aws_internet_gateway" "igw_main" {
36      vpc_id = aws_vpc.vpc-main.id
37      count = length(local.public_subnets) > 0 ? 1 : 0
38  }
39  Pieces: Comment | Pieces: Explain
40  resource "aws_route_table" "route_main" {
41      vpc_id = aws_vpc.vpc-main.id
42      count = length(local.public_subnets) > 0 ? 1 : 0
43
44      route {
45          cidr_block = "0.0.0.0/0"
46          gateway_id = aws_internet_gateway.igw_main[0].id
47      }
48
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 = "display 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.private_subnets : key => {
17              subnet_id = aws_subnet.subnets_main[key].id
18              availability_zone = aws_subnet.subnets_main[key].availability_zone
19          }
20      }
21  }

```

```

23  output "public-subnet-ids" {
24    description = "list of public subnet ids"
25    value = local.public_subnet_output
26  }
27
28  output "private-subnet-ids" {
29    description = "list of private subnet ids"
30    value = local.private_subnet_output
31  }

```

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

```

1  provider "aws" {
2    region = "us-east-1"
3  }
4
5  module "vpc" {
6    source = "./modules/vpc"
7
8    vpc_config = {
9      cidr_block = "10.0.0.0/16"
10     vpc_name = "implementing-vpc-using-own-module"
11   }
12
13   aws_subnets = {
14     public_sub1 = {
15       cidr_block = "10.0.1.0/24"
16       availability_zone = "us-east-1a"
17       public = true
18     }
19     public_sub2 = {
20       cidr_block = "10.0.2.0/24"
21       availability_zone = "us-east-1b"
22       public = true
23     }
24     private_sub3 = {
25       cidr_block = "10.0.3.0/24"
26       availability_zone = "us-east-1c"
27       public = false
28     }
29   }
30 }

```

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

```

1  output "vpc_id" {
2    value = module.vpc.vpc_id
3  }
4

```

```

5   output "public-subnet-ids" {
6     value = module.vpc.public-subnet-ids
7   }
8
9   output "private-subnet-ids" {
10    value = module.vpc.private-subnet-ids
11  }

```

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

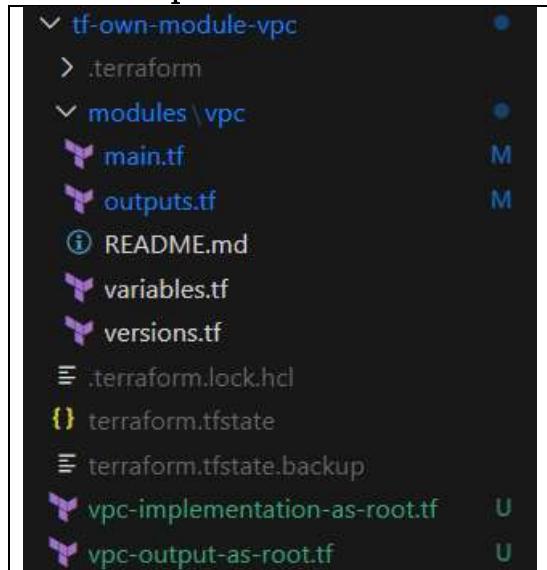
Creating your own Terraform module simplifies infrastructure management by enabling reusability, consistency, and scalability.

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*

```
1  #terraform-aws-vpc
2  | Ctrl+L to chat, Ctrl+K to generate
3  ## Overview
4
5  This Terraform module creates an AWS VPC with a given CIDR block.
6  It also creates multiple subnets (public and private), and for
7  public subnets, it sets up an Internet Gateway (IGW)
8  and appropriate route tables.
9
10 ## Features
11
12 - Creates a VPC with a specified CIDR block
13 - Creates public and private subnets
14 - Creates an Internet Gateway (IGW) for public subnets
15 - Sets up route tables for public subnets
16
17 ## Usage
18
19 ``
20 module "vpc" {
21   source = "./modules/vpc"
22
23   vpc_config = {
24     cidr_block = "10.0.0.0/16"
25     vpc_name = "test_vpc"
26   }
27
28   aws_subnets = {
29     public_sub1 = {
30       cidr_block = "10.0.1.0/24"
31       availability_zone = "us-east-1a"
32       # To set the subnet as public, default is private
33       public = true
34     }
35     public_sub2 = {
36       cidr_block = "10.0.2.0/24"
37       availability_zone = "us-east-1b"
38       public = true
39     }
40     private_sub3 = {
41       cidr_block = "10.0.3.0/24"
42       availability_zone = "us-east-1c"
43       public = false
44     }
45   }
46 }
47 ``
```

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:

```
└─ tf-own-module-vpc
    ├ .terraform
    └─ examples\complete
        └─ main.tf
        └─ output.tf
        └─ README.md
    └─ modules\vpc
        └─ main.tf
        └─ outputs.tf
        └─ README.md
        └─ variables.tf
        └─ versions.tf
    └─ .terraform.lock.hcl
    └─ terraform.tfstate
    └─ terraform.tfstate.backup
    └─ vpc-implementation-as-root.tf
    └─ vpc-output-as-root.tf
```

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.

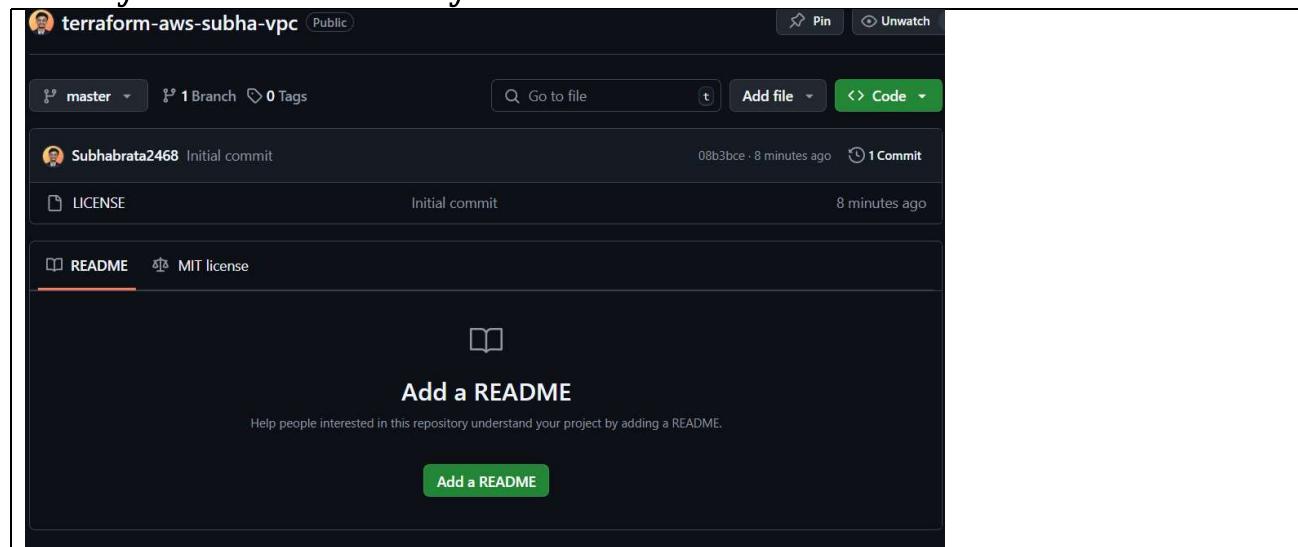


Then you will see this layout

The screenshot shows the 'Create a new repository' page. It includes fields for 'Owner' (Subhabrata2468), 'Repository name' (with a placeholder '/'), a note about repository names being short and memorable, a 'Description (optional)' field, and options for 'Public' or 'Private' visibility (Public is selected). Under 'Initialize this repository with:', there's a checkbox for 'Add a README file' which is unchecked. Below it, a note says 'This is where you can write a long description for your project.' A 'Add .gitignore' section with a dropdown menu set to 'None' is also visible.

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

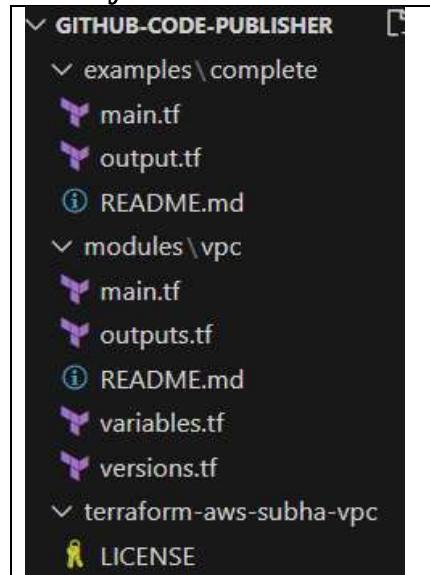
Then you will see this layout



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

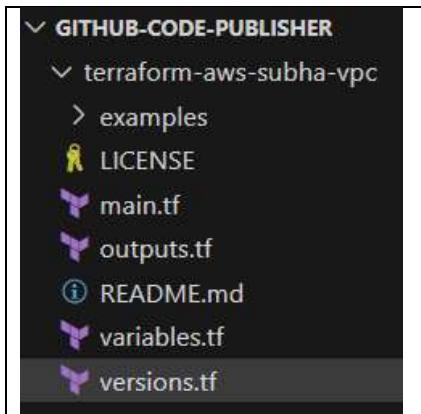


Move all files from the modules/vpc folder (e.g., main.tf, outputs.tf, variables.tf, versions.tf) to the root level of the repository, just outside the modules/vpc directory.

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



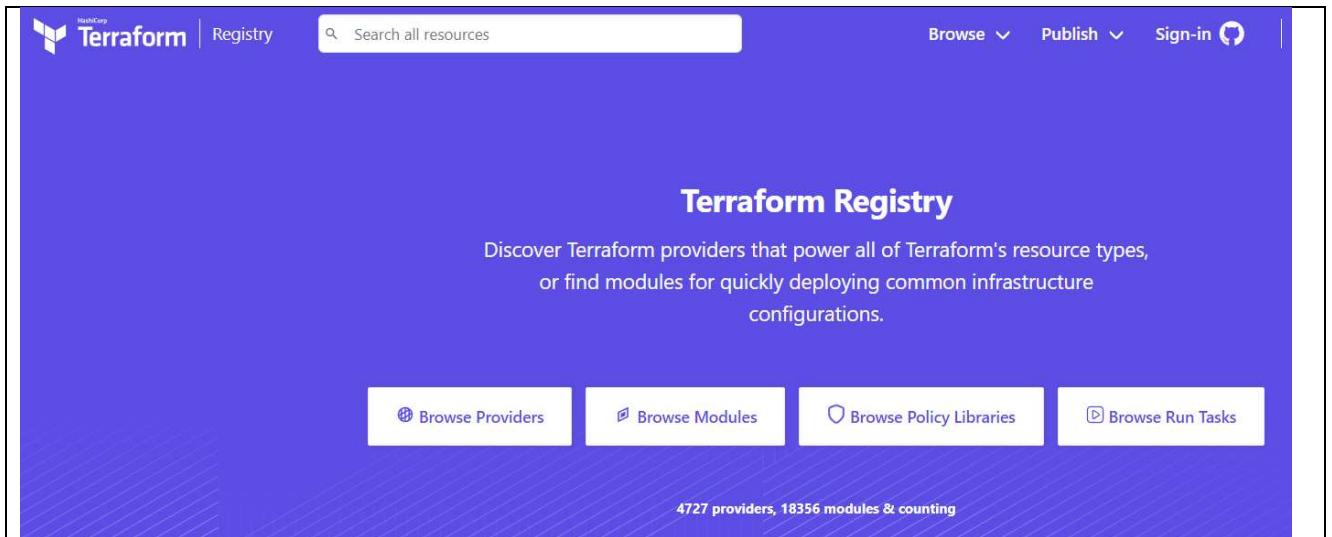
Now add the files and folder to GitHub. After you have added the files and folder you will see this type of layout in GitHub.

The screenshot shows a GitHub repository named 'modules' owned by 'Subhabrata2468'. The repository has 2 commits and was last updated 3 minutes ago. It contains several files: 'examples/complete', 'LICENSE', 'README.md', 'main.tf', 'outputs.tf', 'variables.tf', and 'versions.tf'. The 'README' file is open, displaying the text '#terraform-aws-vpc'.

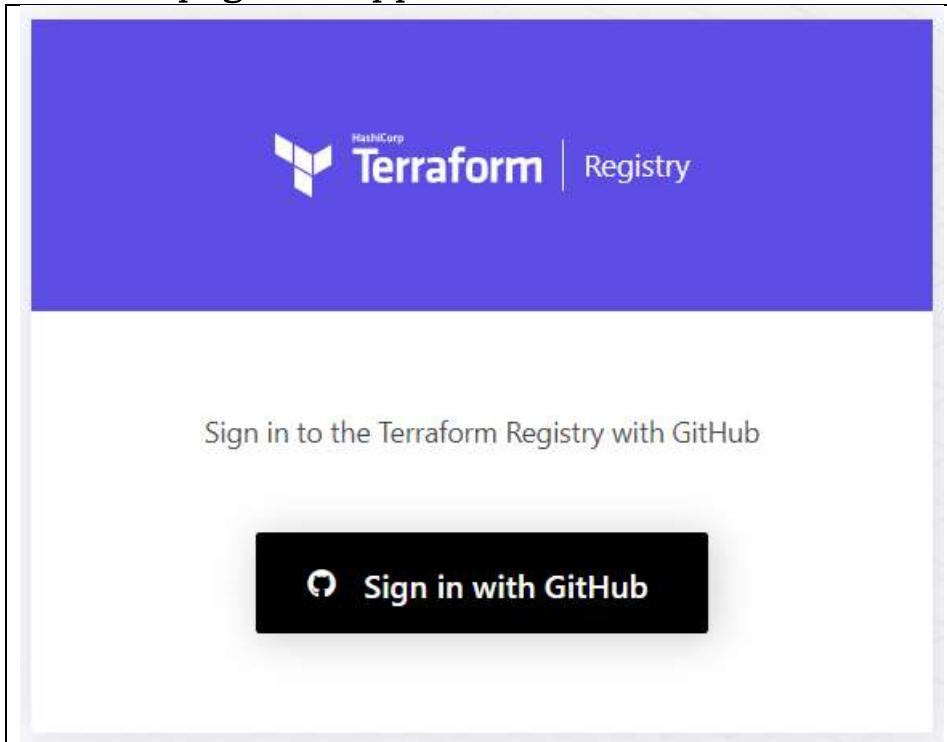
Now add the tag

```
PS D:\Downloads\GitHub-Code-Publisher\terraform-aws-subha-vpc> git tag "v1.0"
fatal: tag 'v1.0' already exists
PS D:\Downloads\GitHub-Code-Publisher\terraform-aws-subha-vpc> git push --tags
Total 0 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
To https://github.com/Subhabrata2468/terraform-aws-subha-vpc.git
 * [new tag]      complete -> complete
 * [new tag]      v1.0 -> v1.0
PS D:\Downloads\GitHub-Code-Publisher\terraform-aws-subha-vpc>
```

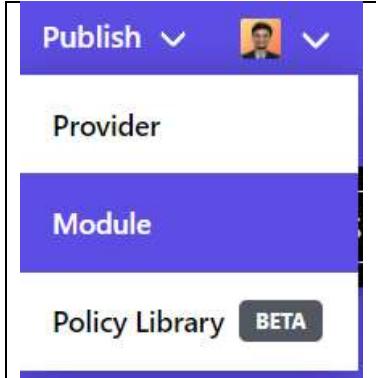
Now go to terraform registry click on sign in option.



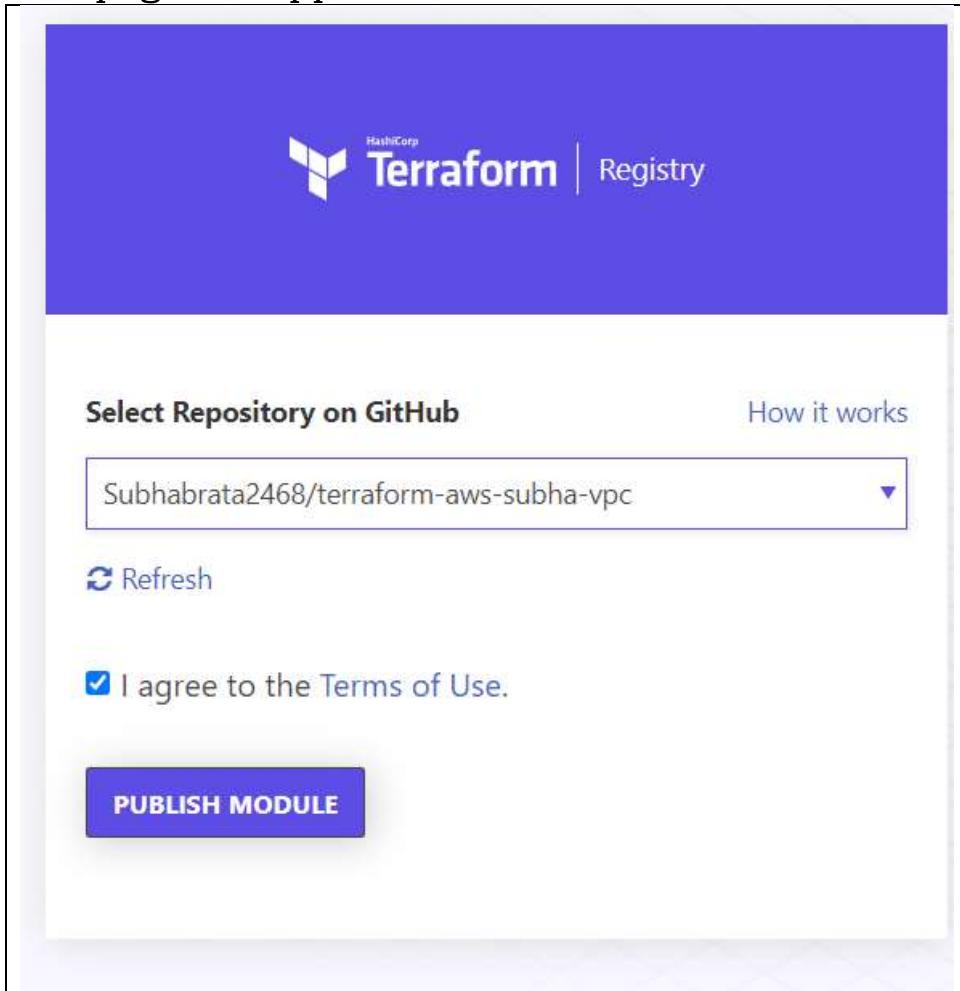
Now this page will appear



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

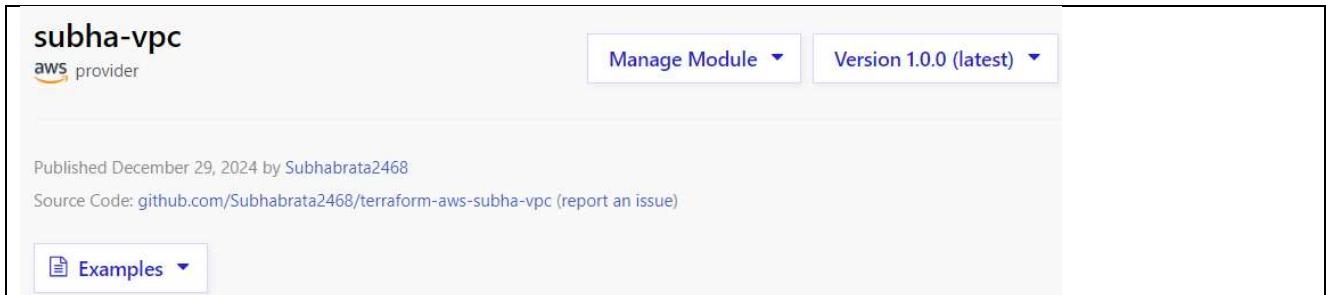


This page will appear



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
<pre>terraform {   required_providers {     aws = {       source = "hashicorp/aws"       version = "5.82.2"     }   }    provider "aws" {     # Configuration options     region = "us-east-1"   }    resource "aws_security_group" "main" {     name = "my-sg"      ingress {       from_port   = 80       to_port     = 80       protocol    = "tcp"       cidr_blocks = ["0.0.0.0/0"]     }   }    resource "aws_instance" "main" {     ami           = "ami-01816d07b1128cd2d"     instance_type = "t3.micro"   } }</pre>	<pre>terraform {   required_providers {     aws = {       source = "hashicorp/aws"       version = "5.82.2"     }   }    provider "aws" {     # Configuration options     region = "us-east-1"   }    resource "aws_security_group" "main" {     name = "my-sg"      ingress {       from_port   = 80       to_port     = 80       protocol    = "tcp"       cidr_blocks = ["0.0.0.0/0"]     }   }    resource "aws_instance" "main" {     ami           = "ami-01816d07b1128cd2d"     instance_type = "t3.micro"     depends_on = [aws_security_group.main]   } }</pre>
Output: → after Applying terraform init → terraform plan → terraform apply	Output: → after Applying terraform init → terraform plan → terraform apply
<b>Plan:</b> 2 to add, 0 to change, 0 to destroy. <code>aws_security_group.main: Creating...</code> <code>aws_instance.main: Creating...</code>	<b>Plan:</b> 2 to add, 0 to change, 0 to destroy. <code>aws_security_group.main: Creating...</code> <code>aws_security_group.main: Creation complete after 5s</code> <code>aws_instance.main: Creating...</code> <code>aws_instance.main: Still creating... [10s elapsed]</code>
In this both “aws_security_group.main” and “aws_instance.main” are being created simultaneously	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	<i>create_before_destroy = true</i>	<i>create_before_destroy = false</i>
<i>Order of Actions</i>	Create first, destroy later	Destroy first, create later
<i>Downtime</i>	No downtime	Downtime possible
<i>Resource Dependency</i>	May require sufficient resources (e.g., IPs, capacity) for two resources to coexist temporarily.	Fewer resource requirements.
<i>Usage</i>	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.

<i><b>Aspect</b></i>	<i><b>prevent_destroy = true</b></i>	<i><b>prevent_destroy = false</b></i>
<i><b>Behavior on Destroy</b></i>	Prevents resource from being destroyed	Allows resource to be destroyed
<i><b>Error on Destroy</b></i>	Terraform fails with an error	Terraform proceeds without error
<i><b>Use Case</b></i>	Critical resources (databases, VPCs)	Non-critical or temporary resources
<i><b>Override Requirement</b></i>	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).

```
resource "aws_iam_user_login_profile" "profile" {
  for_each          = aws_iam_user.users
  user              = each.value.name
  password_length = 12

  lifecycle {
    ignore_changes = [
      password_length,
      password_reset_required,
      pgp_key,
    ]
  }
}
```

***ignore\_changes***

<i>Aspect</i>	<i>With ignore_changes</i>	<i>Without ignore_changes</i>
<b>Manual Changes</b>	Ignored for specified attributes	Overwritten by Terraform
<b>Configuration Updates</b>	Ignored for specified attributes	Applied as part of the next plan/apply
<b>Use Case</b>	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 “postconditions” 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 my AWS console

The screenshot shows the AWS S3 console interface. At the top, there is a navigation bar with tabs for Objects, Metadata - Preview, Properties, Permissions, Metrics, Management, and Access Points. The 'Objects' tab is selected. Below the navigation bar, there is a toolbar with actions like Copy S3 URI, Copy URL, Download, Open, Delete, Actions (with a dropdown menu), Create folder, and Upload. A search bar labeled 'Find objects by prefix' and a 'Show versions' button are also present. The main area displays a message: 'No objects' and 'You don't have any objects in this bucket.' There is a prominent 'Upload' button at the bottom of this section.

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 {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "5.82.2"
    }
  }

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

  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
  bucketRegionalDomainName = "bucket-for-storing-git-files.s3.us-east-1.amazonaws.com"
  hostedZoneId              = "Z3AQBSTGFYJSTF"
  id                       = "bucket-for-storing-git-files"
  objectLockEnabled         = 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>
```

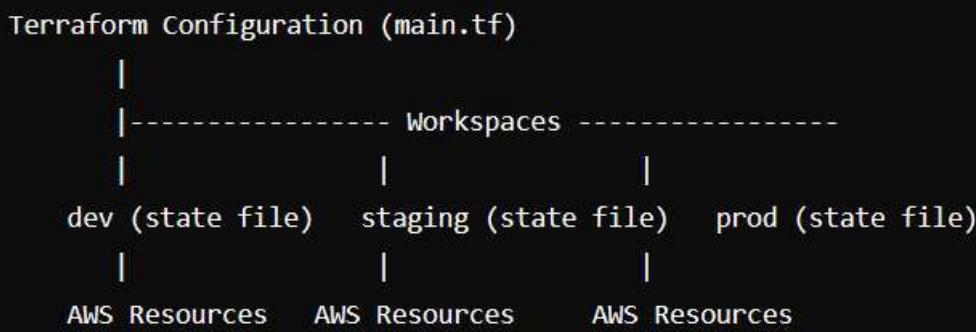
Delete a workspace (if empty):

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

## Diagram: Terraform Workspaces Example

Here's a simple diagram explaining workspaces:

### Terraform Workspaces for Environment Isolation:



- **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

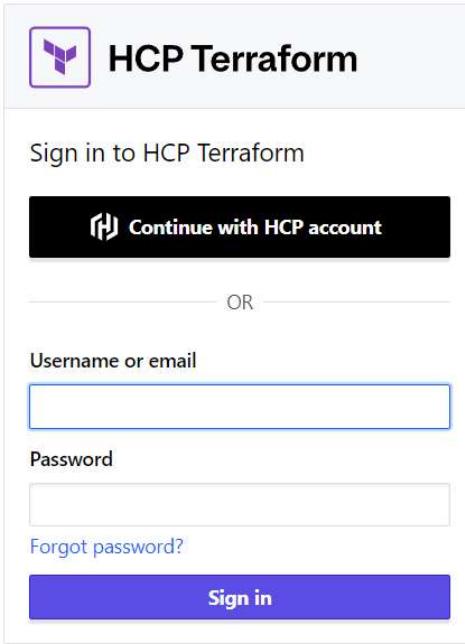
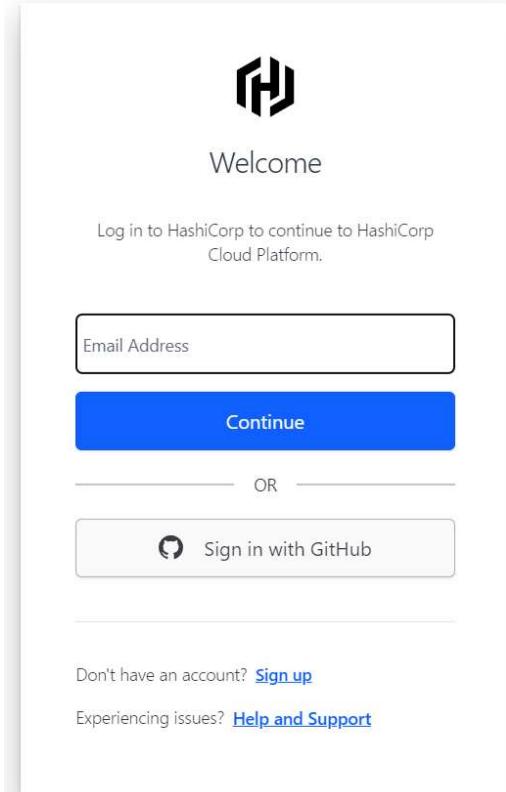


## 38. TERRAFORM CLOUD WITH GITHUB

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

Providing features like

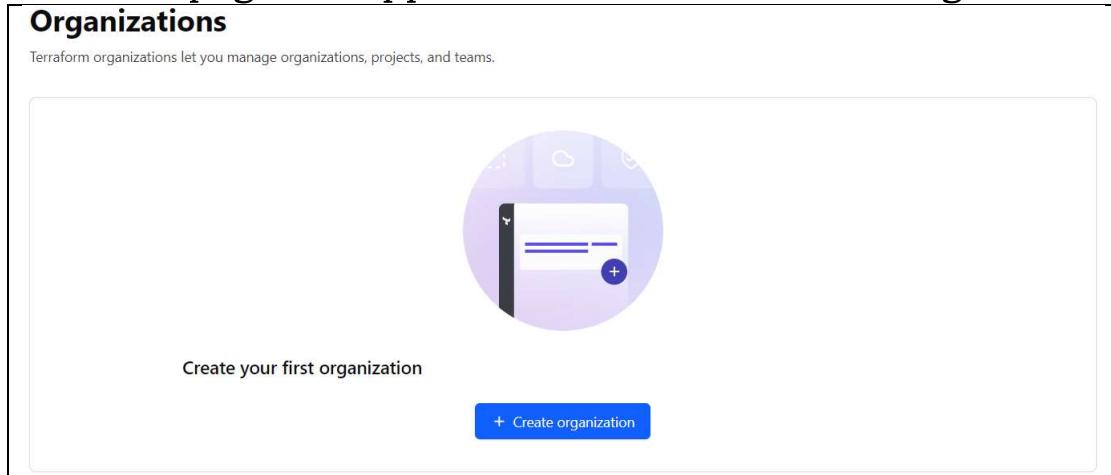
- remote state management,
- version control system (VCS) integration,
- automated runs, and
- secure variable management.

STEP-1	STEP-2
<p>Now go to any browser and type “app.terraform.io” then this page will appear</p>  <p>Sign in to HCP Terraform</p> <p>HCP Terraform</p> <p>Continue with HCP account</p> <p>OR</p> <p>Username or email</p> <p>Password</p> <p>Forgot password?</p> <p>Sign in</p> <p>Sign in with Terraform SSO.</p> <p>Need to sign up? Create your <a href="#">free account</a>.</p> <p>View <a href="#">Terraform Offerings</a> to find out which one is right for you.</p>	<p>Since you have an HCP account and are logged in, click on "<b>Continue with HCP account</b>"</p>  <p>Welcome</p> <p>Log in to HashiCorp to continue to HashiCorp Cloud Platform.</p> <p>Email Address</p> <p>Continue</p> <p>OR</p> <p>Sign in with GitHub</p> <p>Don't have an account? <a href="#">Sign up</a></p> <p>Experiencing issues? <a href="#">Help and Support</a></p>

STEP-3	STEP-4
<p>Now click on “Sign in with GitHub”. Then tick on all the terms and conditions then click on “Continue”</p> <p><b>Service agreements</b></p> <p><a href="#">Terms of Service</a> <small>Required</small>  <input checked="" type="checkbox"/> I accept the Terms of Service</p> <p><a href="#">Privacy Policy</a> <small>Required</small>  <input checked="" type="checkbox"/> I accept the Privacy Policy</p> <p><b>Marketing and newsletters</b></p> <p><input checked="" type="checkbox"/> I would like to be updated via email about HashiCorp products, features, events, announcements, and education materials.</p> <p><a href="#">Continue</a></p>	<p>Now again click on “Continue”.</p> <p><b>Create a new HCP-linked account</b></p> <p>Your HCP Account is your HashiCorp Cloud Platform login and the only credentials you need to access every HashiCorp product. From the HashiCorp Learn and community platforms, to HCP Terraform and the Terraform Registry. Log in once and stay logged in.</p> <p><small>(i) Some features, like two-factor authentication, are managed from the HashiCorp Cloud Platform.</small></p> <p><a href="#">Continue</a> <a href="#">Cancel</a></p>

## 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

## Create a new Workspace

HCP Terraform organizes your infrastructure resources by workspaces. A workspace contains infrastructure resources, variables, state data, and run history. [Learn more](#) about workspaces in HCP Terraform.

### Choose your workflow

#### Version Control Workflow

Trigger runs based on changes to configuration in repositories.

 Best for those who need traceability and transparency

#### CLI-Driven Workflow

Trigger runs in a workspace using the Terraform CLI.

 Best for those comfortable with Terraform CLI

#### API-Driven Workflow

Trigger runs using the HCP Terraform API.

 Best for those with custom integrations and pipelines

## STEP-7

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

## Create a new Workspace

HCP Terraform organizes your infrastructure resources by workspaces. A workspace contains infrastructure resources, variables, state data, and run history. [Learn more](#) about workspaces in HCP Terraform.

 1 Connect to VCS

 2 Choose a repository

 3 Configure settings

### Connect to a version control provider

Choose the version control provider that hosts the Terraform configuration for this workspace.

Project:  Default Project 

 GitHub

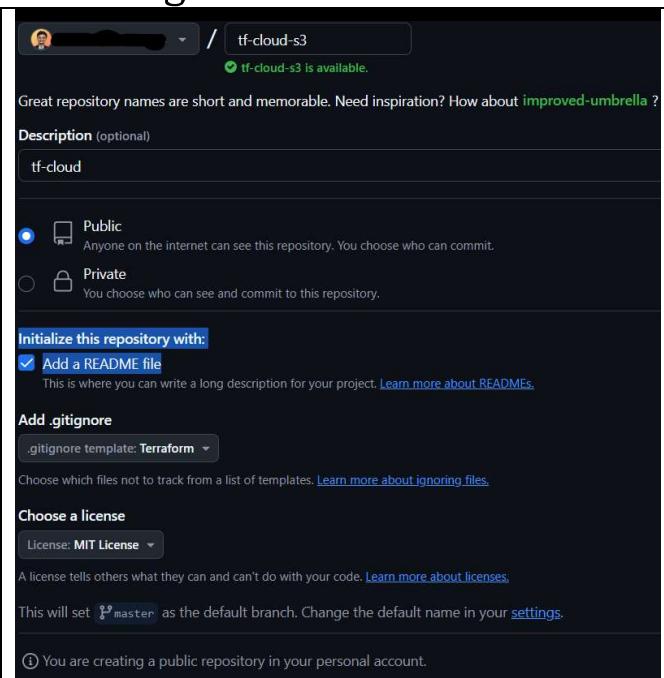
 GitLab

 Bitbucket

 Azure DevOps

## 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.

Terraform
terraform-aws-subha-vpc
tf-cloud-s3
Tools-Installition-Guide
ud282

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 "committing s3 through tf cloud"
[master 6eecac2] committing 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

.gitignore	Initial commit	23 minutes ago
LICENSE	Initial commit	23 minutes ago
README.md	Initial commit	23 minutes ago
main.tf	committing s3 through tf cloud	1 minute ago

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

### Latest Run

[View all runs](#)

Triggered via UI

 [REDACTED] triggered a run a few seconds ago via UI ~ 6eecac2

× Errored

Policy checks	Estimated cost change	Plan duration	Resources to be changed
Add	Enable	Less than a minute	+0 ~0 -0

[See details](#)

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

```
! Error: No valid credential sources found
with provider["registry.terraform.io/hashicorp/aws"]
on main.tf line 14, in provider "aws":
provider "aws" {

Please see https://registry.terraform.io/providers/hashicorp/aws
for more information about providing credentials.

Error: failed to refresh cached credentials, no EC2 IMDS role found, operation error ec2imds: GetMetadata, request canceled, context deadline exceeded
```

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

< Workspaces

tf-cloud-s3

Overview

Runs

States

Variables

Settings >

Then this layout will appear

**tf-cloud-s3**

ID: ws-B9qzCDYxWhwX88pc

workspace

Unlocked Resources 0 Terraform v1.10.3 Updated a few seconds ago

**Variables**

Terraform uses all [Terraform](#) and [Environment](#) variables for all plans and applies in this workspace. Workspaces using Terraform 0.10.0 or later can also load default values from any `*.auto.tfvars` files in the configuration. You may want to use the HCP Terraform Provider or the variables API to add multiple variables at once.

**Sensitive variables**

[Sensitive](#) variables are never shown in the UI or API, and can't be edited. They may appear in Terraform logs if your configuration is designed to output them. To change a sensitive variable, delete and replace it.

---

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”

Select variable category

**Terraform variable**  
These variables should match the declarations in your configuration. Click the HCL box to use interpolation or set a non-string value.

**Environment variable**  
These variables are available in the Terraform runtime environment.

Key	Value
key	value <input type="checkbox"/> Sensitive

Description (Optional)  
description (optional)

**Add variable** **Cancel**

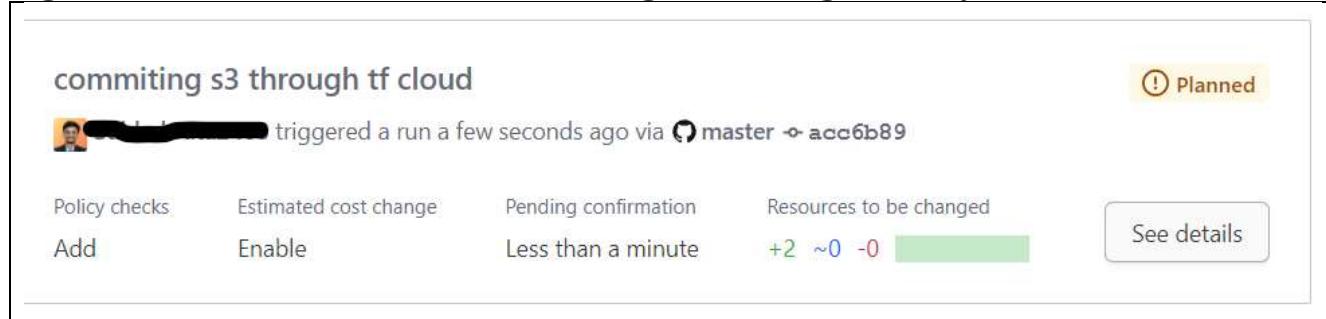
Subhabrata Panda

| 147

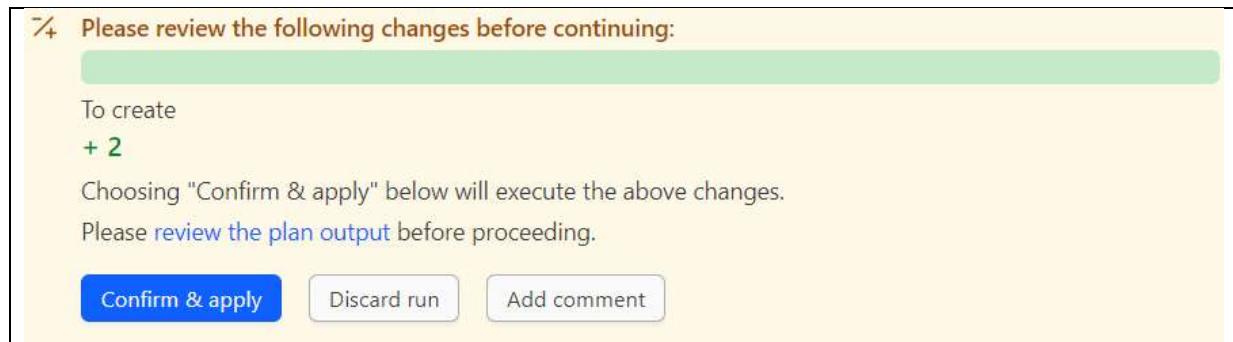
Give the key and value and also tick the “Sensitive option”  
Then click on “Add variables”

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

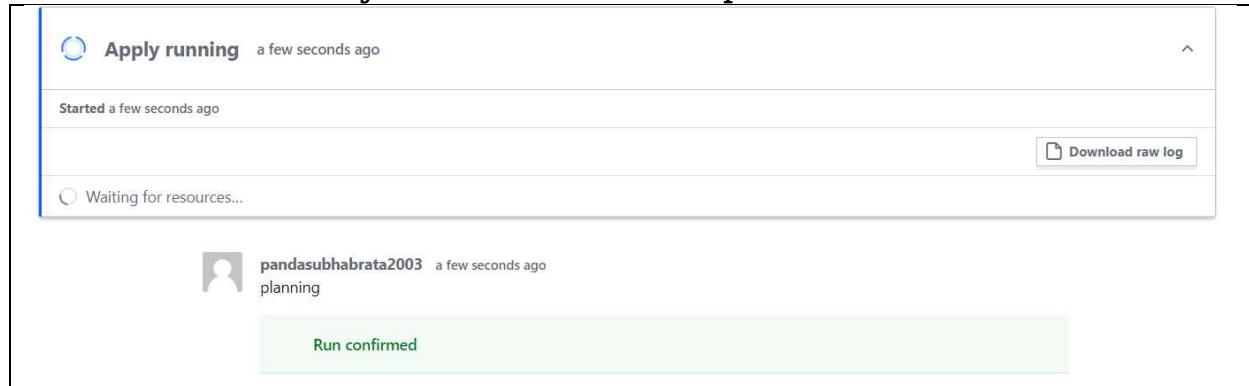
Again, commit it now see the change in the given layout



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”



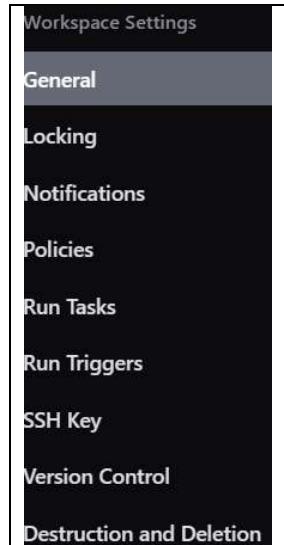
Go to S3 bucket in AWS console for verification



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”

#### Manually destroy

Queuing a destroy plan will redirect to a new plan that will destroy all of the infrastructure managed by Terraform. It is equivalent to running `terraform plan -destroy -out=destroy.tfplan` followed by `terraform apply destroy.tfplan` locally.

[Queue destroy plan](#)

Then read the instruction carefully and fill in the blank

This will destroy all infrastructure managed by this workspace.

Please proceed with caution. Selecting “Queue destroy plan” below will immediately create a new plan that will destroy all of the infrastructure managed by **tf-cloud-s3**. If you’re certain you wish to proceed, please enter the workspace name below to confirm.

Enter the workspace name to confirm:

[Queue destroy plan](#)

[Cancel](#)

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

Resources will be deleted.