**NAME: Bhushan Malpani** **DIV: D15C** **ROLL NO: 33**

## CASE STUDY

### Basic Infrastructure Management with Terraform

- **Concepts Used**: Terraform, AWS EC2, and S3.
- **Problem Statement**: "Use Terraform to create an EC2 instance and an S3 bucket. Store the EC2 instance's IP address in the S3 bucket."
- **Tasks**:
    - Write a simple Terraform script to provision an EC2 instance and an S3 bucket.
    - Use Terraform outputs to extract the EC2 instance's IP address.
    - Store the IP address in a text file in the S3 bucket.

# Introduction

**Case Study Overview:**

In this case study, we delve into the realm of Basic Infrastructure Management with Terraform by leveraging Terraform's Infrastructure-as-Code (IaC) capabilities to automate the provisioning and management of cloud resources on AWS. Specifically, the case study focuses on deploying an EC2 instance (virtual machine) and an S3 bucket (storage service) on AWS using Terraform, highlighting the power of automation in modern cloud infrastructure.

The problem statement presents a real-world scenario where cloud resources need to be set up automatically without manual intervention, ensuring that the infrastructure is consistent, scalable, and easy to manage. Terraform serves as the backbone for achieving this by enabling declarative configuration for infrastructure provisioning. This case demonstrates the process of creating an EC2 instance and S3 bucket, extracting the public IP address of the EC2 instance, and storing it as a text file within the S3 bucket, illustrating a common use case in cloud environments for automating cloud resource management.

The goal is to simplify the traditionally complex task of setting up infrastructure by writing a Terraform script that automates this entire workflow, ensuring efficiency, repeatability, and reduced risk of human error.

### EC2 (Elastic Compute Cloud)

Amazon EC2 provides scalable computing capacity in the cloud, allowing users to run virtual servers (instances) for a wide range of applications. EC2 instances offer flexibility in selecting operating systems, instance types, and storage options, enabling customized computing environments. It supports automatic scaling, load balancing, and integration with other AWS services, making it ideal for dynamic workloads. EC2 is widely used for hosting websites,

running applications, or processing large datasets. The pay-as-you-go pricing model ensures cost efficiency by billing only for the compute resources used.

## S3 (Simple Storage Service) Bucket

Amazon S3 is an object storage service designed to store and retrieve large amounts of data reliably and securely. S3 buckets are containers for objects, which can be any type of file, such as documents, images, or backups. It offers virtually unlimited storage with options for different storage classes based on access frequency and durability needs. S3 integrates with a wide range of AWS services, making it a key component in data pipelines, backups, and content distribution. S3's strong security and access control mechanisms ensure that stored data is well-protected against unauthorized access.

## Terraform

Terraform is an open-source Infrastructure as Code (IaC) tool that allows users to define, provision, and manage cloud resources through human-readable configuration files. It supports multiple cloud providers, including AWS, Azure, and Google Cloud, enabling users to automate the setup of infrastructure across different environments. Terraform uses a declarative language, meaning users specify the desired state of resources, and Terraform handles the provisioning process to achieve that state. With features like **state management** and **resource dependencies**, Terraform ensures predictable, consistent deployments, making it ideal for automating cloud infrastructure and reducing manual efforts.

---

Key Feature and Application:

The standout feature of this case study lies in Terraform’s ability to automate cloud resource management and manage outputs dynamically. This includes Terraform’s capability to create an EC2 instance and S3 bucket, extract the EC2 instance's public IP address, and use that data seamlessly in real-time to update the S3 bucket—all in one coherent, automated process.

1. **Automated Infrastructure Provisioning:**
    - Terraform enables the automation of creating cloud resources. In this case, the provisioning of an EC2 instance and S3 bucket happens through Terraform scripts, eliminating the need to manually set up these services via the AWS Management Console.
    - With a simple configuration file, users can describe the desired state of their infrastructure, and Terraform will handle the details, ensuring resources are created, updated, or destroyed accordingly.

2. **Dynamic Outputs with Terraform:**
    - One of Terraform's key strengths is its ability to dynamically handle and output information from the created resources. In this case, the public IP address of the EC2 instance is dynamically extracted after the instance is provisioned.
    - Terraform's output functionality captures this data and makes it available for further use, which in this case is to store it as a file in the S3 bucket.
3. **Integration Between AWS Services (EC2 and S3):**
    - This solution highlights the integration between AWS services, with Terraform acting as the intermediary. The EC2 instance and S3 bucket, although different in their purpose (compute vs storage), are seamlessly connected using Terraform.
    - The EC2 instance's IP address is stored in the S3 bucket as a text file. This is a practical example of how cloud services can interact to fulfill specific use cases, such as logging IP addresses for future access, audit trails, or further automated workflows.

---

**Practical Use Case:**

This automation approach is particularly useful in scenarios where developers or IT teams need to:

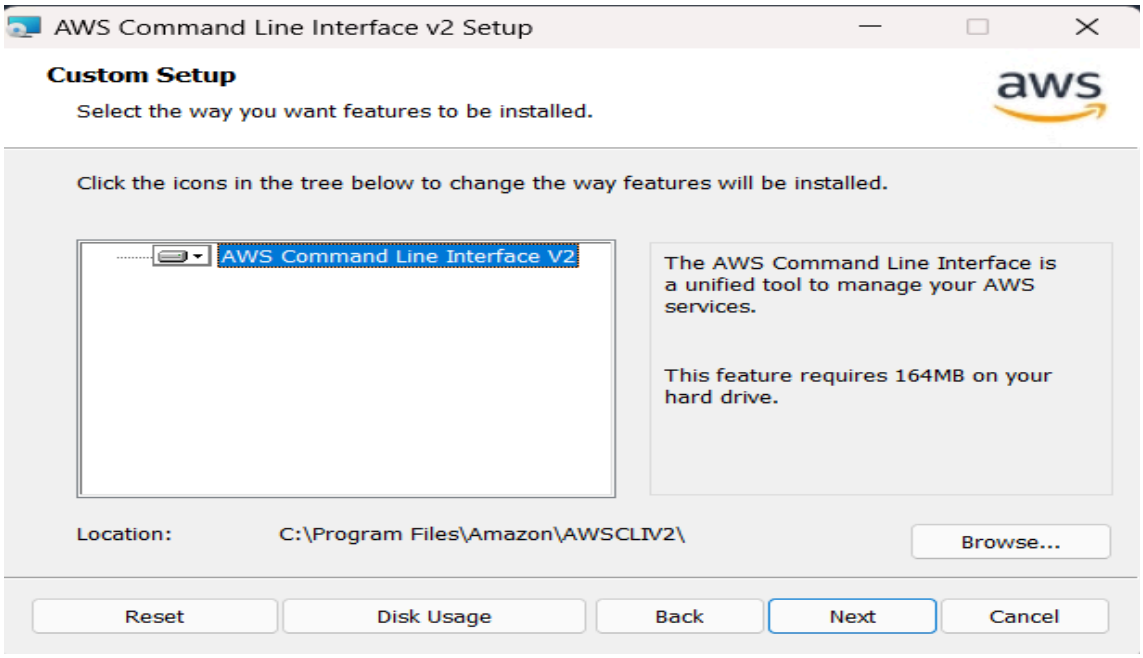
- Regularly spin up new infrastructure (EC2 instances) and track their details, such as IP addresses.
- Centralize and store important details (like IP addresses) in S3 for further processing, auditing, or access.
- Maintain a scalable and easily reproducible infrastructure setup, reducing the risk of human error and saving time on manual cloud configuration.

For instance, a company that frequently deploys new EC2 instances as part of its dynamic scaling strategy can use this approach to automate both provisioning and logging IP addresses in S3, ensuring the information is always up to date and accessible.

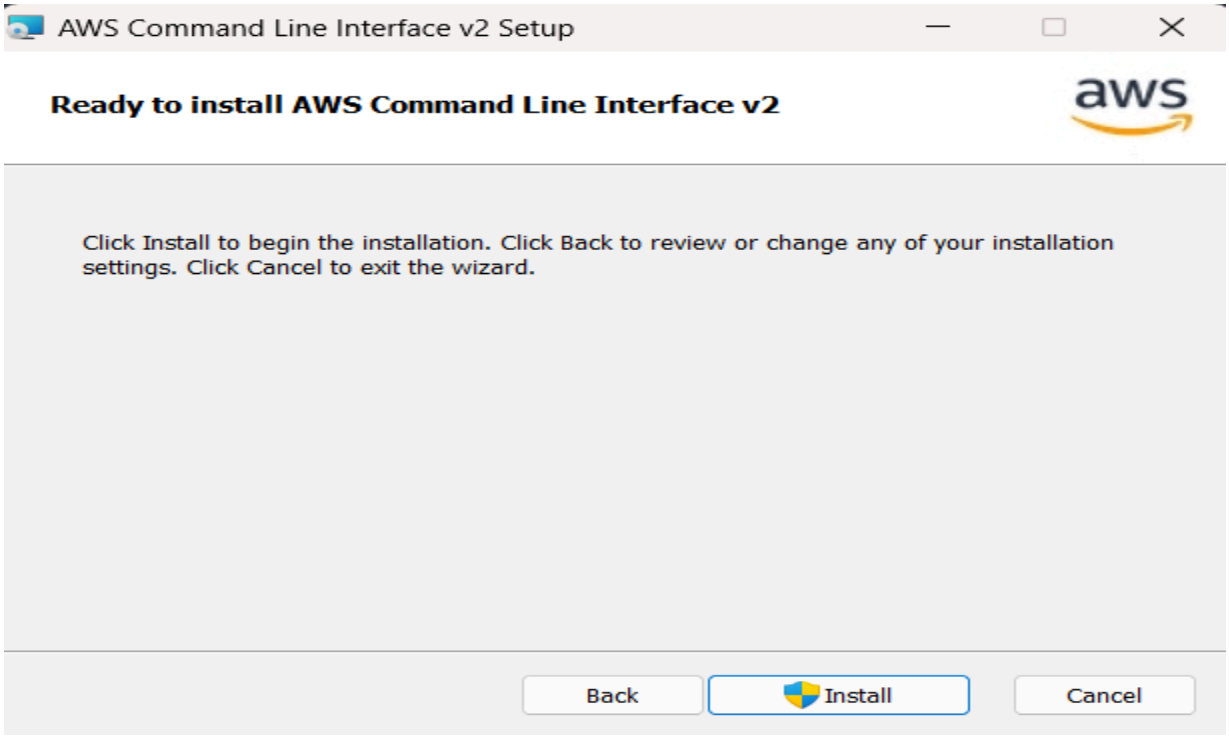
By using Terraform to automate these tasks, cloud operations become more streamlined, ensuring that infrastructure is managed efficiently, consistently, and with reduced manual oversight.

**Prerequisites**:
(If Downloaded then skip):
**Step 1:** Downloading AWS CLI to perform steps:
Download according to your OS as i have windows i downloaded it for windows by clicking the download link

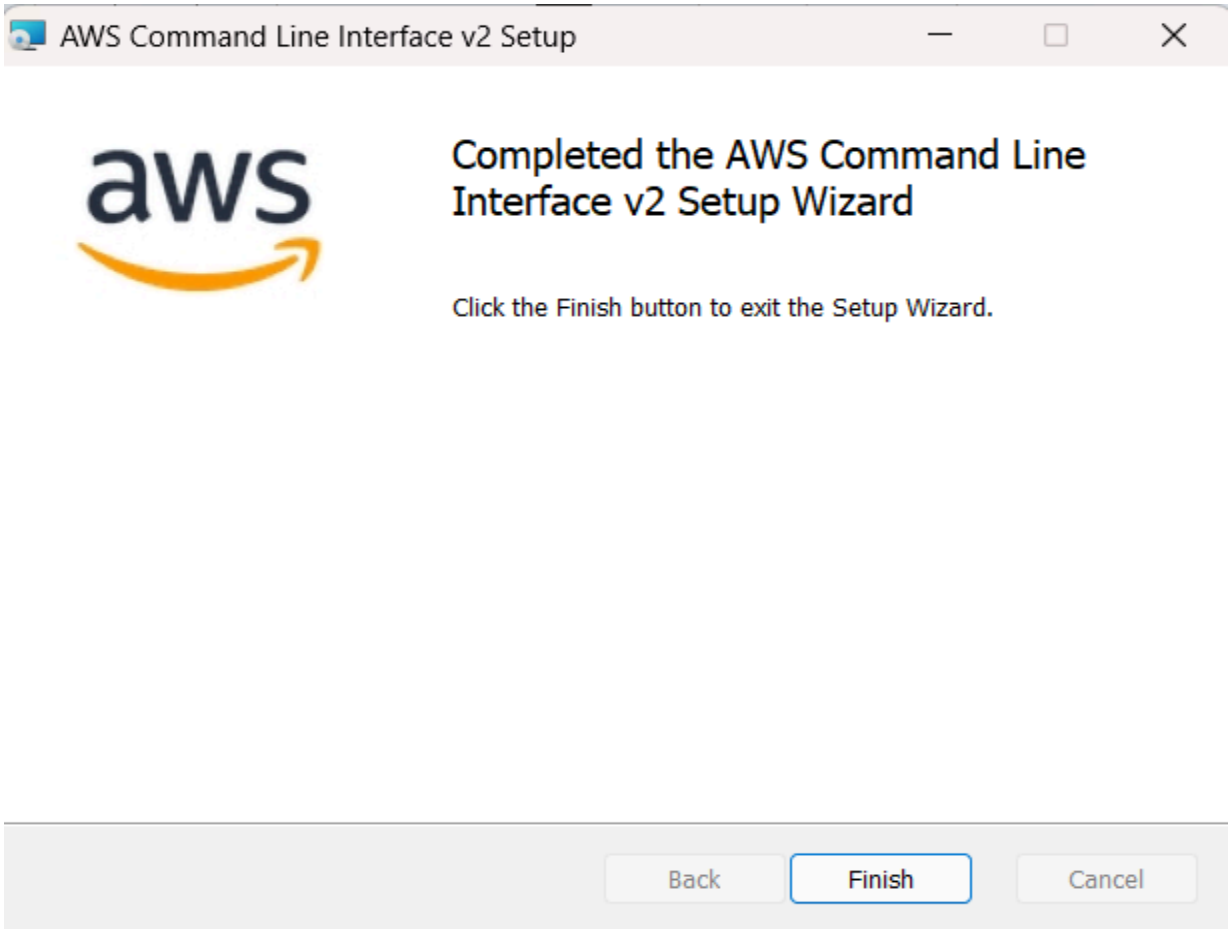
Open the downloaded file and click on next:




Click on install for installing it



Click on finish




**Step 2.** To check weather it is ready to use, open Command prompt and use command:
**aws --version**

```
C:\Users\HP>aws --version
aws-cli/2.18.10 Python/3.12.6 Windows/11 exe/AMD64
```

## 1. Log In to the AWS Management Console

- Go to the AWS Management Console and sign in with your AWS account.

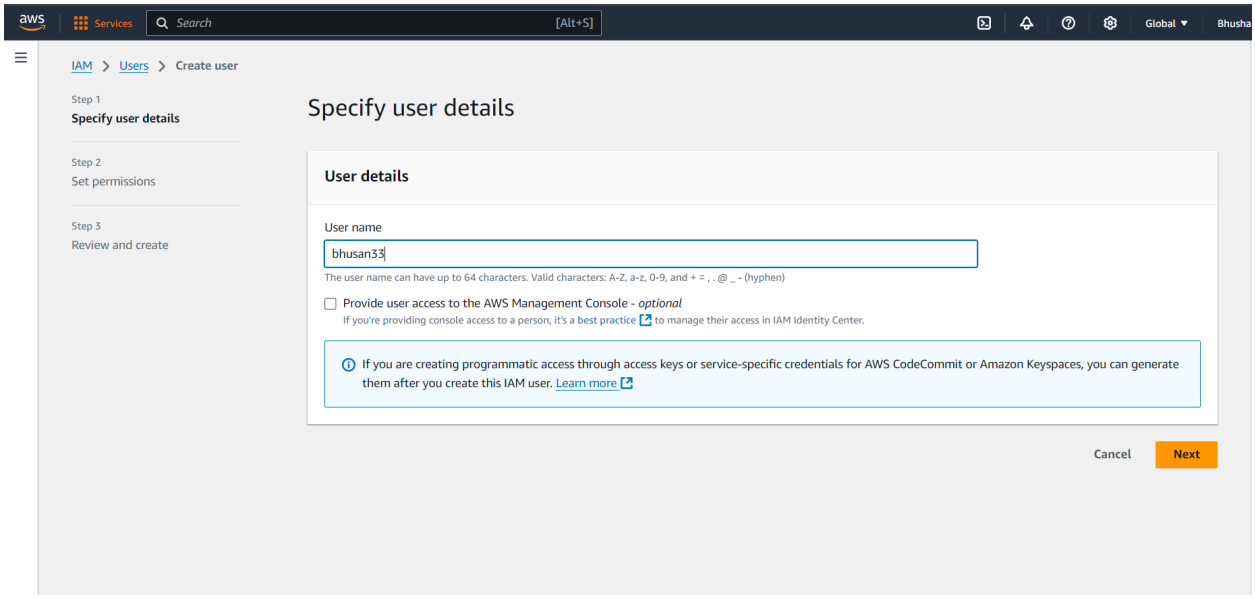
## 2. Navigate to IAM (Identity and Access Management)

- In the AWS Management Console, type **IAM** in the search bar and select **IAM**.
- IAM is used to manage user permissions and credentials.

## 3. Create a New User (if you don't have one)

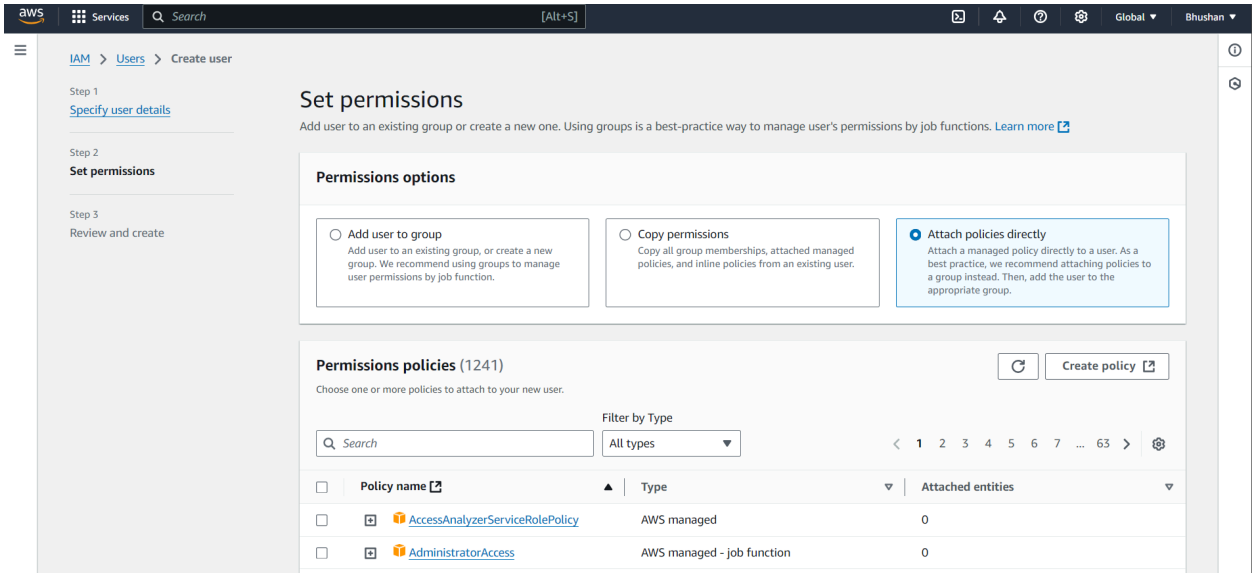
If you already have an IAM user with the necessary permissions, skip to **Step 4**. Otherwise, create a new IAM user:

1. In the left-hand sidebar, click **Users**, then **Add user**.
2. In the **User name** field, provide a name for the user



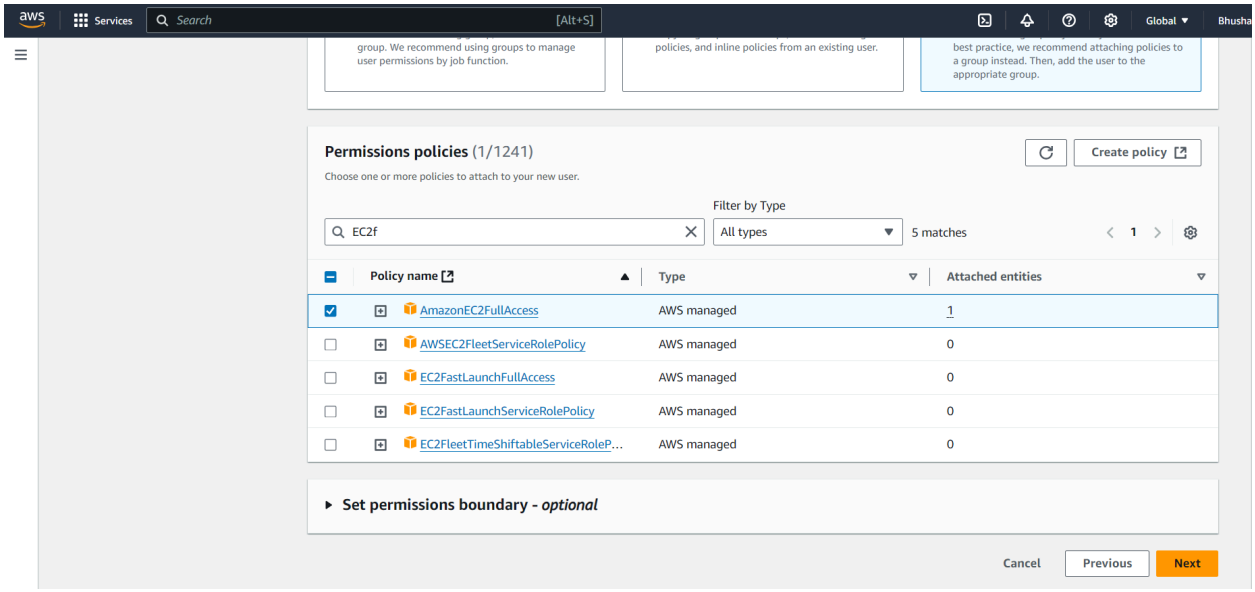

3. Click **Next: Permissions** to assign necessary permissions.

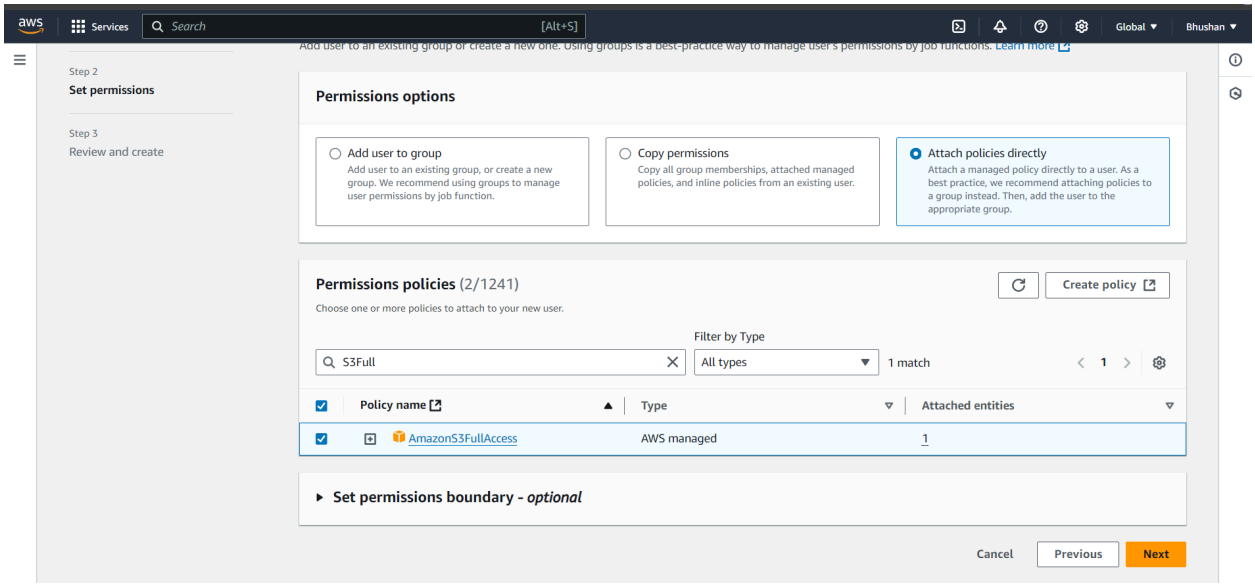
   Choose **Attach policies Directly**



Attach policies that grant permissions to S3 and EC2 resources. The following policies will provide the necessary permissions:

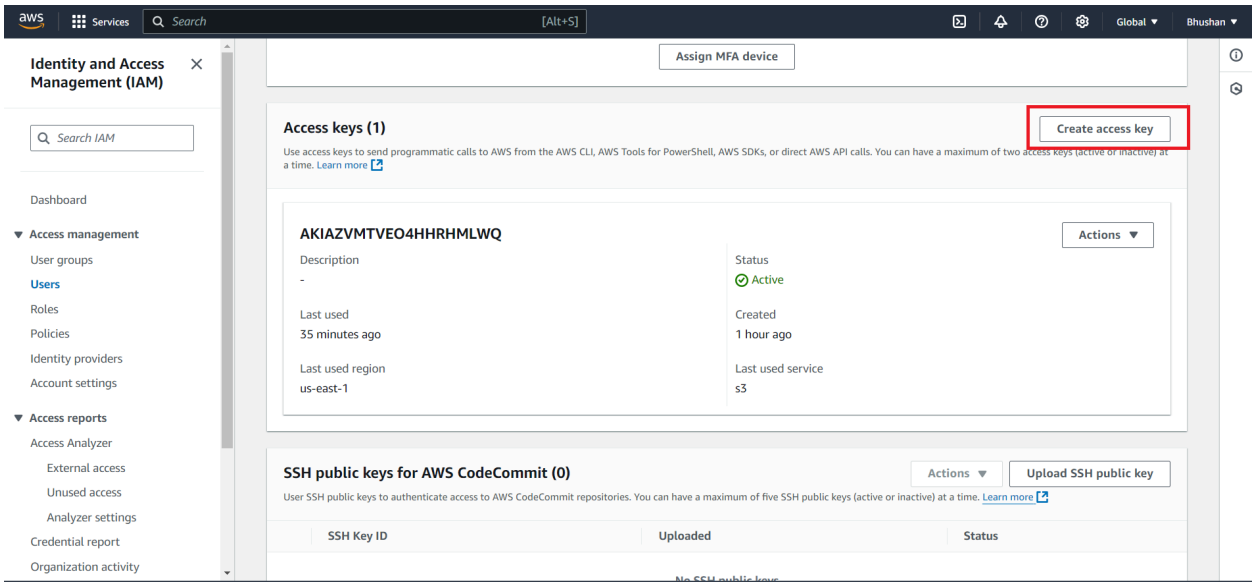
- **AmazonS3FullAccess**
- **AmazonEC2FullAccess** (or custom EC2 policy allowing



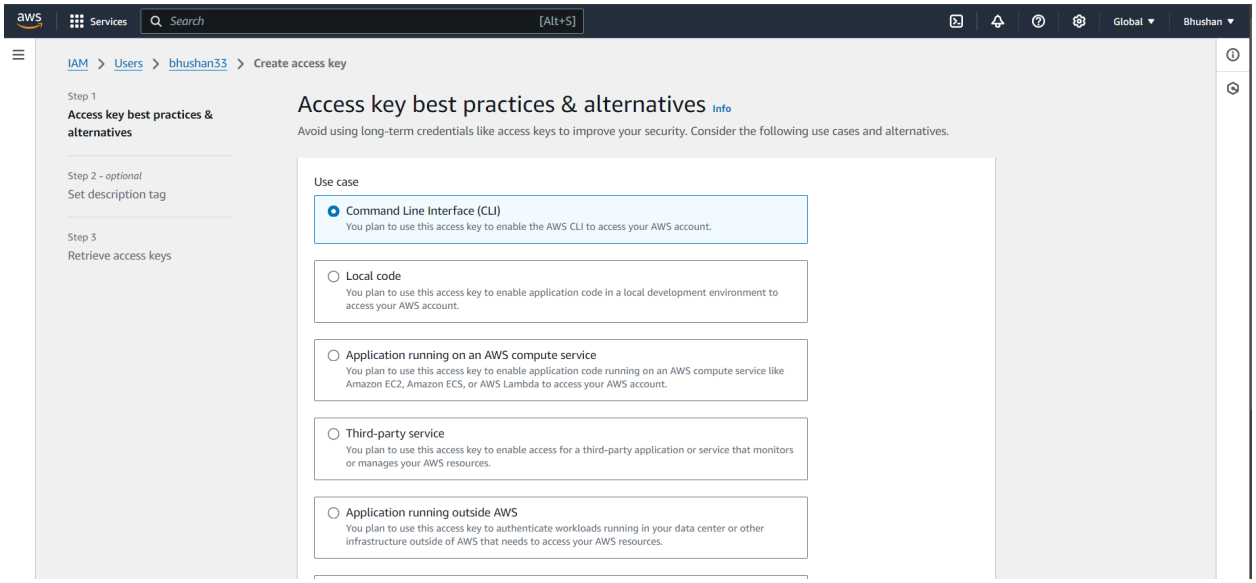




Then click on Next

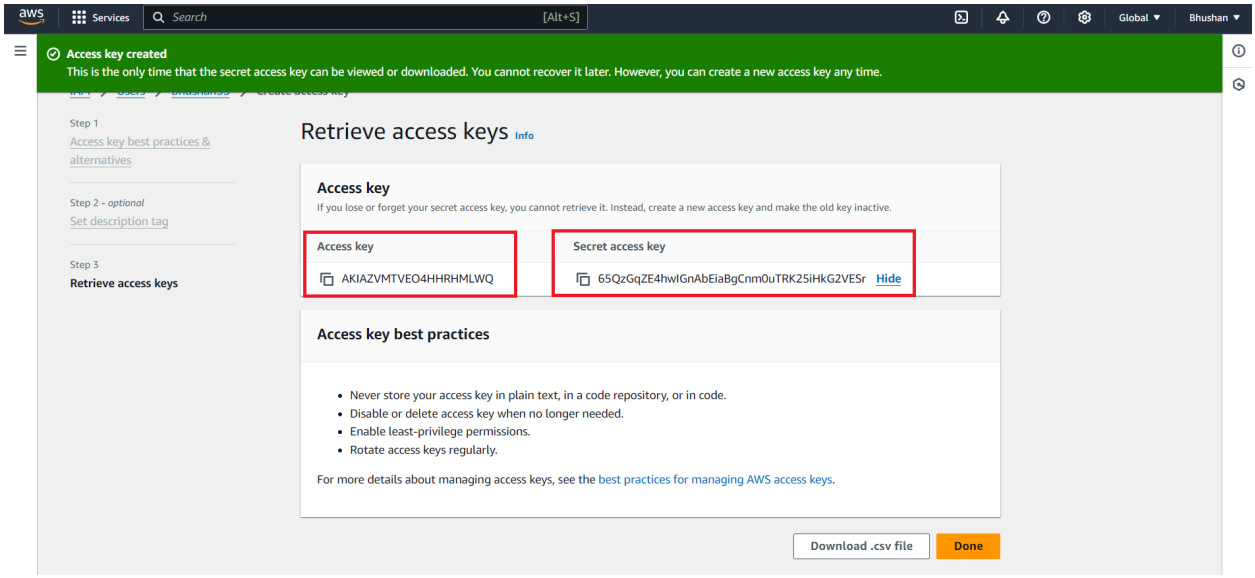
Then go inside the user to generate access key and secret access key




Select Command Line Interface(CLI):




Copy the access key or download the .csv file because you can see it only once. If you forget then you need to create a new access key.




**Step 3)** Then to configure your aws account you can use command:

**Option 1: aws configure**

```
C:\Users\HP>aws configure
AWS Access Key ID [None]: ASIASDKXT7V7Y4FU7UT5
AWS Secret Access Key [None]: lgBZ15rXGJ9IABH8DauUe4QSAYHm57SCJZOswmZc
Default region name [None]: us-east-1
Default output format [None]: json
```

**Option 2**: Manually create a file called **~/.aws/credentials** and add your keys:

**[default]**

**aws_access_key_id = <Your Access Key>**

**aws_secret_access_key = <Your Secret Key>**

**Step 4)** Create a new directory for your project:

**mkdir terraform-aws-ec2-s3**

**cd terraform-aws-ec2-s3**

```
C:\Users\HP>mkdir terraform-aws-ec2-s3

C:\Users\HP>cd terraform-aws-ec2-s3
```

Inside the directory, create a new file called **`main.tf`**:

```
C:\Users\HP\terraform-aws-ec2-s3>touch main.tf
```

**Step 4)** Open main.tf file in any text editor and add the following

```
# main.tf

# 1. Initialize the AWS provider

provider "aws" {

  region = "us-east-1"

}


# 2. Create an S3 Bucket

resource "aws_s3_bucket" "my_bucket" {

  bucket = "<Unique bucket name>"

}




# 3. Create an EC2 instance

resource "aws_instance" "my_ec2" {

  ami           = "ami-06b21ccaeff8cd686"  // AMI id for your region (by default us-east-1)

  instance_type = "t2.micro"


  tags = {

    Name = "MyEC2Instance"

  }
```

```
  vpc_security_group_ids = [aws_security_group.my_sg.id]
}

# 5. Create a security group that allows SSH and HTTP access
resource "aws_security_group" "my_sg" {
  name_prefix = "my-sg"

  ingress {
    from_port   = 22
    to_port     = 22
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  ingress {
    from_port   = 80
    to_port     = 80
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  egress {
    from_port   = 0
    to_port     = 0
```

```
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }
}

# 6. Output the public IP of the EC2 instance
output "ec2_ip" {
  value = aws_instance.my_ec2.public_ip
}

# 7. Store the EC2 IP address in a file and upload it to the S3 bucket using aws_s3_object
resource "local_file" "ec2_ip_file" {
  content  = aws_instance.my_ec2.public_ip
  filename = "ec2-ip.txt"
}

resource "aws_s3_object" "upload_ip" {
  bucket = aws_s3_bucket.my_bucket.bucket
  key    = "ec2-ip.txt"
  source = local_file.ec2_ip_file.filename
  acl    = "private"
}
```

Change the mentioned things

C: > Users > HP > Desktop > terraform-aws-ec2-s3 > main.tf

```hcl
# 1. Initialize the AWS provider
provider "aws" {
  region = "us-east-1"
}

# 2. Create an S3 Bucket
resource "aws_s3_bucket" "my_bucket" {
  bucket = "bhushan210104"
}


# 3. Create an EC2 instance
resource "aws_instance" "my_ec2" {
  ami           = "ami-06b21ccaeff8cd686"
  instance_type = "t2.micro"

  tags = {
    Name = "MyEC2Instance"
  }

  vpc_security_group_ids = [aws_security_group.my_sg.id]
}

# 5. Create a security group that allows SSH and HTTP access
resource "aws_security_group" "my_sg" {
  name_prefix = "my-sg"

  ingress {
    from_port   = 22
    to_port     = 22
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  ingress {
    from_port   = 80
    to_port     = 80
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }
}

# 6. Output the public IP of the EC2 instance
output "ec2_ip" {
  value = aws_instance.my_ec2.public_ip
}

# 7. Store the EC2 IP address in a file and upload it to the S3 bucket using aws_s3_object
resource "local_file" "ec2_ip_file" {
  content  = aws_instance.my_ec2.public_ip
  filename = "ec2-ip.txt"
}

resource "aws_s3_object" "upload_ip" {
  bucket = aws_s3_bucket.my_bucket.bucket
  key    = "ec2-ip.txt"
  source = local_file.ec2_ip_file.filename
  acl    = "private"
}
```

**Step 5) Initialize the Terraform Project**

1. Open your terminal and navigate to the directory where **main.tf** is located.
2. Run the following command to initialize the Terraform project:

   **terraform init**

```
PS C:\Users\HP\Desktop\terraform-aws-ec2-s3> terraform init
Initializing the backend...
Initializing provider plugins...
- Finding latest version of hashicorp/local...
- Finding latest version of hashicorp/aws...
- Installing hashicorp/local v2.5.2...
- Installed hashicorp/local v2.5.2 (signed by HashiCorp)
- Installing hashicorp/aws v5.72.1...
- Installed hashicorp/aws v5.72.1 (signed by HashiCorp)
Terraform has created a lock file .terraform.lock.hcl to record the provider
selections it made above. Include this file in your version control repository
so that Terraform can guarantee to make the same selections by default when
you run "terraform init" in the future.

Terraform has been successfully initialized!

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

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

Before applying any changes, you should review what Terraform will create:

1. Run the **plan** command to see the resources that will be created:

   **terraform plan**

```
C:\Users\HP\Desktop\terraform-aws-ec2-s3>terraform plan

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the
following symbols:
  + create

Terraform will perform the following actions:

  # aws_instance.my_ec2 will be created
  + resource "aws_instance" "my_ec2" {
      + ami                                  = "ami-0c55b159cbfafe1f0"
      + arn                                  = (known after apply)
      + associate_public_ip_address          = (known after apply)
      + availability_zone                    = (known after apply)
      + cpu_core_count                       = (known after apply)
      + cpu_threads_per_core                 = (known after apply)
      + disable_api_stop                     = (known after apply)
      + disable_api_termination              = (known after apply)
      + ebs_optimized                        = (known after apply)
      + get_password_data                    = false
      + host_id                              = (known after apply)
      + host_resource_group_arn              = (known after apply)
      + iam_instance_profile                 = (known after apply)
      + id                                   = (known after apply)
      + instance_initiated_shutdown_behavior = (known after apply)
      + instance_lifecycle                   = (known after apply)
      + instance_state                       = (known after apply)
      + instance_type                        = "t2.micro"
      + ipv6_address_count                   = (known after apply)
      + ipv6_addresses                       = (known after apply)
```

```
      + owner_id               = (known after apply)
      + revoke_rules_on_delete = false
      + tags_all               = (known after apply)
      + vpc_id                 = (known after apply)
    }

  # local_file.ec2_ip_file will be created
  + resource "local_file" "ec2_ip_file" {
      + content              = (known after apply)
      + content_base64sha256 = (known after apply)
      + content_base64sha512 = (known after apply)
      + content_md5          = (known after apply)
      + content_sha1         = (known after apply)
      + content_sha256       = (known after apply)
      + content_sha512       = (known after apply)
      + directory_permission = "0777"
      + file_permission      = "0777"
      + filename             = "ec2-ip.txt"
      + id                   = (known after apply)
    }

Plan: 6 to add, 0 to change, 0 to destroy.

Changes to Outputs:
  + ec2_ip = (known after apply)

---

Note: You didn't use the -out option to save this plan, so Terraform can't guarantee to take exactly these actions if
you run "terraform apply" now.
```

2. Apply the configuration to create the resources:

**terraform apply**

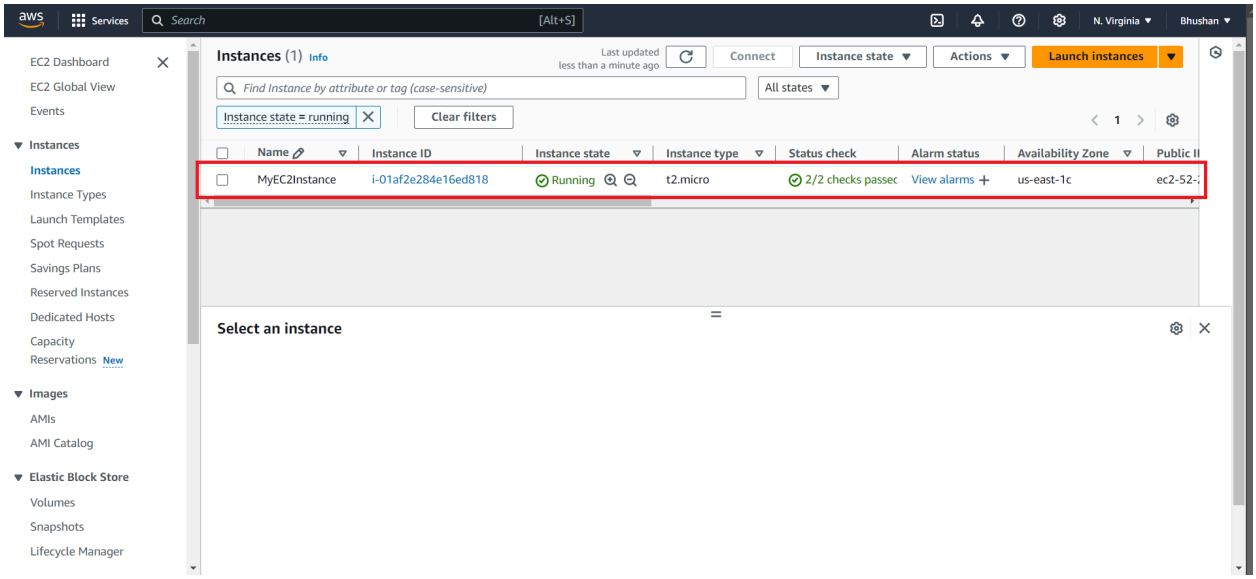
```
C:\Users\HP\Desktop\terraform-aws-ec2-s3>terraform apply
aws_security_group.my_sg: Refreshing state... [id=sg-0e8a598e871f76496]
aws_s3_bucket.my_bucket: Refreshing state... [id=bhushan210104]

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the
following symbols:
  + create

Terraform will perform the following actions:

  # aws_instance.my_ec2 will be created
  + resource "aws_instance" "my_ec2" {
      + ami                                  = "ami-06b21ccaeff8cd686"
      + arn                                  = (known after apply)
      + associate_public_ip_address          = (known after apply)
      + availability_zone                    = (known after apply)
      + cpu_core_count                       = (known after apply)
      + cpu_threads_per_core                 = (known after apply)
      + disable_api_stop                     = (known after apply)
      + disable_api_termination              = (known after apply)
      + ebs_optimized                        = (known after apply)
      + get_password_data                    = false
      + host_id                              = (known after apply)
      + host_resource_group_arn              = (known after apply)
      + iam_instance_profile                 = (known after apply)
      + id                                   = (known after apply)
      + instance_initiated_shutdown_behavior = (known after apply)
      + instance_lifecycle                   = (known after apply)
```

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

  Enter a value: yes

aws_instance.my_ec2: Creating...
aws_instance.my_ec2: Still creating... [10s elapsed]
aws_instance.my_ec2: Creation complete after 16s [id=i-01af2e284e16ed818]
local_file.ec2_ip_file: Creating...
local_file.ec2_ip_file: Creation complete after 0s [id=670a5c5c4000494eef48b5ac64f65141634e57c1]
aws_s3_object.upload_ip: Creating...
aws_s3_object.upload_ip: Still creating... [10s elapsed]
aws_s3_object.upload_ip: Still creating... [20s elapsed]
aws_s3_object.upload_ip: Creation complete after 21s [id=ec2-ip.txt]

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

Outputs:

ec2_ip = "52.22.10.198"
```

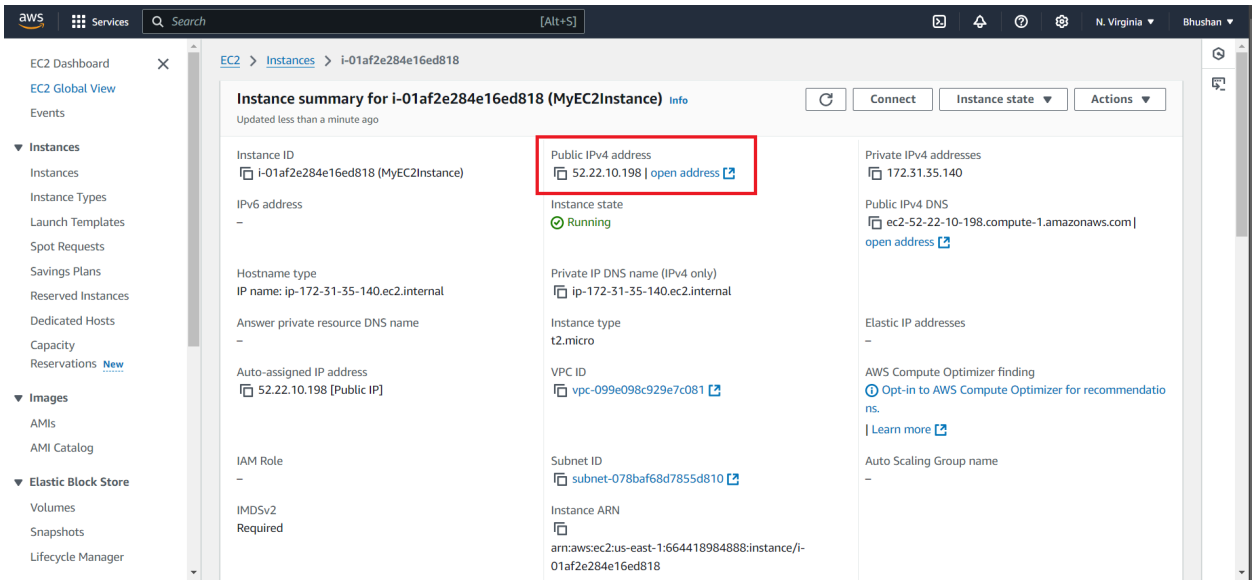
The **OUTPUT** can be seen : **ec2_ip: “52.22.10.198”**

**Step 6)** To verify that the IP of EC2 is stored inside S3 bucket follow the below steps:

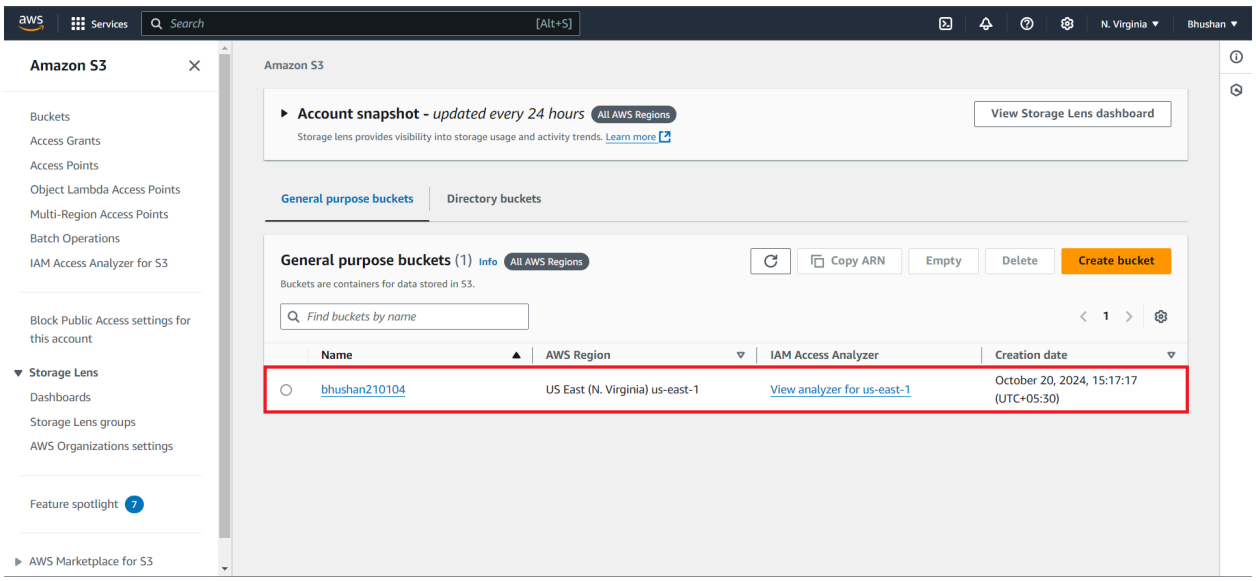
1. We can see the instance we defined is running below:



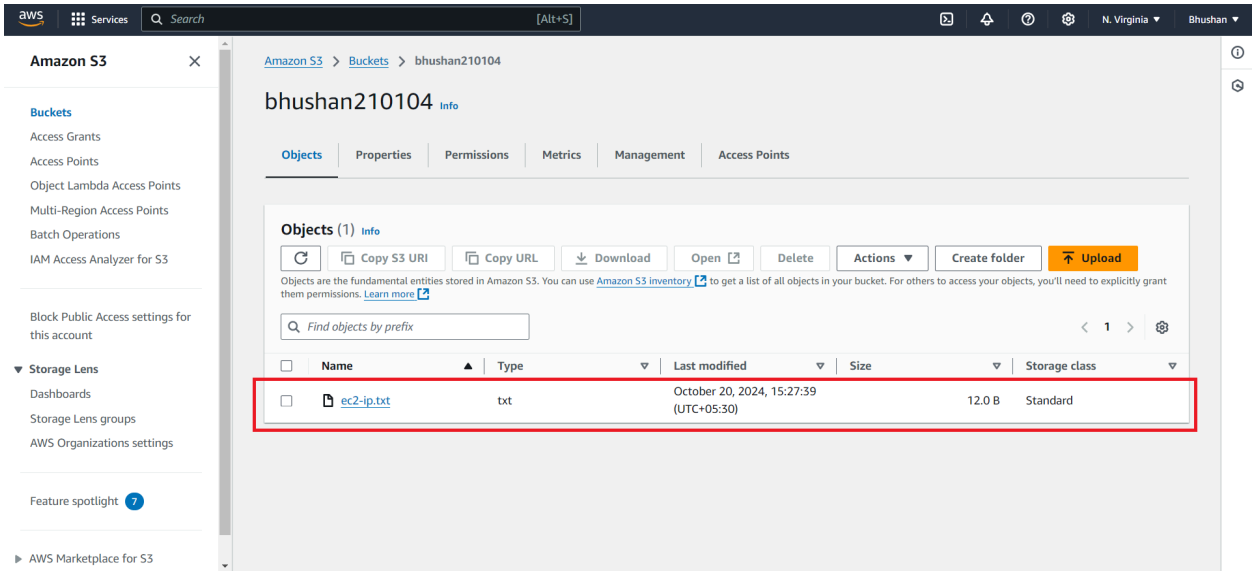

2. Inside the instance dashboard we can check the IPv4 address: **"52.22.10.198"**



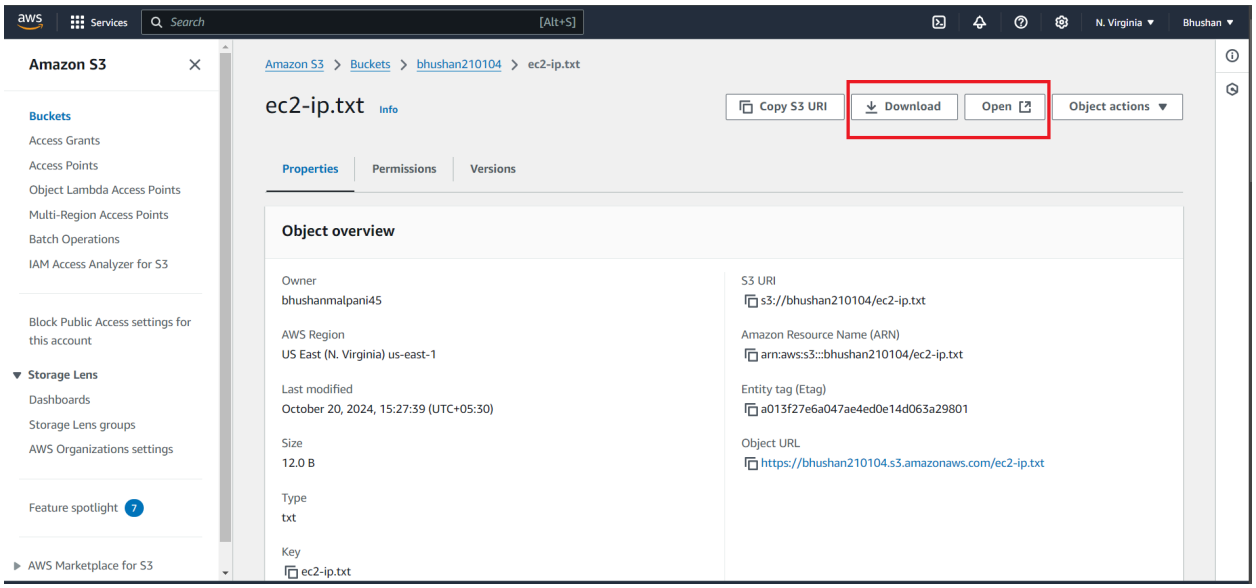
Now we can see S3 bucket in the Amazon S3 service where we have stored the IP of EC2 Instance:



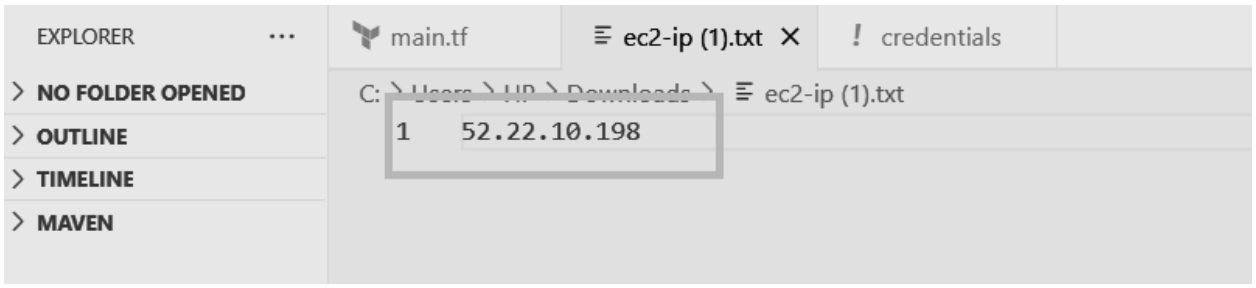

Inside the S3 bucket we can see the file which we have created to store ip address(ec-2-ip.txt):



Here we can click on Open button, the ec-2-ip.txt file will get downloaded:




Opening the txt file we can see the IPv4 Address of EC2 Instance we created:




**The IPv4 Address we saw of EC2 instance is same as the IPv4 address present inside the S3 bucket**

**EXTRA STEPS:**

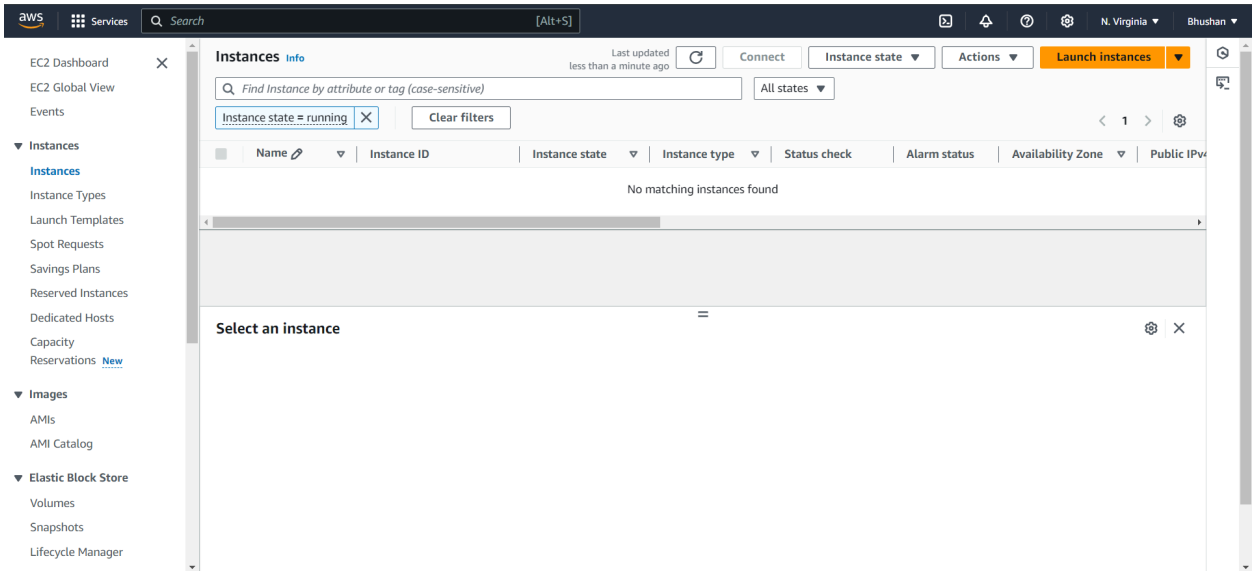
**These steps are to delete the EC2 Instance and S3 bucket we created to store IP address, so that we can avoid any charges from the charges from AWS services.**

### terraform destroy

```
C:\Users\HP\Desktop\terraform-aws-ec2-s3>terraform destroy
aws_s3_bucket.my_bucket: Refreshing state... [id=bhushan210104]
aws_security_group.my_sg: Refreshing state... [id=sg-0e8a598e871f76496]
aws_instance.my_ec2: Refreshing state... [id=i-01af2e284e16ed818]
local_file.ec2_ip_file: Refreshing state... [id=670a5c5c4000494eef48b5ac64f65141634e57c1]
aws_s3_object.upload_ip: Refreshing state... [id=ec2-ip.txt]

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the
following symbols:
  - destroy

Terraform will perform the following actions:

  # aws_instance.my_ec2 will be destroyed
  - resource "aws_instance" "my_ec2" {
      - ami                                  = "ami-06b21ccaeff8cd686" -> null
      - arn                                  = "arn:aws:ec2:us-east-1:664418984888:instance/i-01af2e284e16ed818" -> null
      - associate_public_ip_address          = true -> null
      - availability_zone                    = "us-east-1c" -> null
      - cpu_core_count                       = 1 -> null
      - cpu_threads_per_core                 = 1 -> null
      - disable_api_stop                     = false -> null
      - disable_api_termination              = false -> null
      - ebs_optimized                        = false -> null
      - get_password_data                    = false -> null
      - hibernation                          = false -> null
      - id                                   = "i-01af2e284e16ed818" -> null
      - instance_initiated_shutdown_behavior = "stop" -> null
      - instance_state                       = "running" -> null
```

```
    }

Plan: 0 to add, 0 to change, 5 to destroy.

Changes to Outputs:
  - ec2_ip = "52.22.10.198" -> null

Do you really want to destroy all resources?
  Terraform will destroy all your managed infrastructure, as shown above.
  There is no undo. Only 'yes' will be accepted to confirm.

  Enter a value: yes

aws_s3_object.upload_ip: Destroying... [id=ec2-ip.txt]
aws_s3_object.upload_ip: Destruction complete after 1s
aws_s3_bucket.my_bucket: Destroying... [id=bhushan210104]
local_file.ec2_ip_file: Destroying... [id=670a5c5c4000494eef48b5ac64f65141634e57c1]
local_file.ec2_ip_file: Destruction complete after 0s
aws_instance.my_ec2: Destroying... [id=i-01af2e284e16ed818]
aws_s3_bucket.my_bucket: Destruction complete after 1s
aws_instance.my_ec2: Still destroying... [id=i-01af2e284e16ed818, 10s elapsed]
aws_instance.my_ec2: Still destroying... [id=i-01af2e284e16ed818, 20s elapsed]
aws_instance.my_ec2: Still destroying... [id=i-01af2e284e16ed818, 30s elapsed]
aws_instance.my_ec2: Destruction complete after 33s
aws_security_group.my_sg: Destroying... [id=sg-0e8a598e871f76496]
aws_security_group.my_sg: Destruction complete after 1s

Destroy complete! Resources: 5 destroyed.
```

We can verify that the instances have been deleted. We can visit EC2 Instance on AWS service, we can see there are no instances running.




**Conclusion:**

In this experiment, we used Terraform to automate the deployment of AWS infrastructure by creating an EC2 instance and an S3 bucket, and storing the instance's public IP address in the S3 bucket. We began by configuring AWS CLI with temporary credentials provided by AWS Academy, allowing us to authenticate and interact with AWS services. Using Terraform, we defined the necessary resources in the main.tf file, specifying the AWS provider and creating an EC2 instance with Amazon Linux 2. We also set up an S3 bucket to store a file containing the EC2 instance's public IP address. The Terraform script utilized outputs to extract the public IP and the local_file resource to save it locally, followed by uploading it to the S3 bucket. This experiment demonstrated how Terraform simplifies infrastructure management by automating the provisioning, configuration, and management of cloud resources, making it highly useful for maintaining scalable and repeatable infrastructure setups across multiple environments.