Report By Mugle Sruthi

Infrastructure as Code: Automating EC2 Instance Setup on AWS with Terraform

Terraform: Infrastructure as Code (IaC) Tool

1. Introduction

Terraform is an open-source Infrastructure as Code (IaC) tool developed by HashiCorp. It enables users to define, provision, and manage cloud and on-premises infrastructure using a declarative configuration language. Terraform supports multiple cloud providers, including AWS, Azure, Google Cloud, and others, making it a versatile tool for infrastructure automation.

2. Why Do We Need Terraform?

Traditional infrastructure management involves manual processes, which are error-prone, slow, and difficult to scale. Terraform addresses these challenges by:

- Automating Infrastructure Deployment – Reduces manual errors and speeds up provisioning.
- Ensuring Consistency – The same configuration can be deployed across multiple environments.
- Supporting Multi-Cloud & Hybrid Cloud – Manages resources across different providers.
- Version Control & Collaboration – Infrastructure configurations can be versioned using Git.
- Cost Efficiency – Reduces over-provisioning and optimizes resource usage.

3. Use Cases of Terraform

Terraform is widely used in various scenarios:

- Cloud Resource Provisioning – Automating the creation of VMs, networks, and storage.
- Multi-Tier Applications – Deploying web servers, databases, and load balancers.
- Disaster Recovery – Quickly recreating infrastructure in case of failures.
- Kubernetes & Container Orchestration – Managing Kubernetes clusters and Docker environments.
- Compliance & Security – Enforcing security policies through code.

4. How to Use Terraform

Step 1: Install Terraform

Download Terraform from terraform.io and add it to your system PATH.

Step 2: Write a Terraform Configuration (HCL)

Create a **`.tf`** file (e.g., **`main.tf`**) to define infrastructure:

```
provider "aws" {

  region = "us-east-1"

}



resource "aws_instance" "web_server" {
```

```
  ami           = "ami-0c55b159cbfafe1f0"

  instance_type = "t2.micro"

  tags = {

    Name = "Terraform-WebServer"

  }

}
```

Step 3: Initialize & Apply

```
terraform init   # Downloads provider plugins

terraform plan   # Shows execution plan

terraform apply  # Creates infrastructure
```

Step 4: Destroy (Optional)

To remove resources:

```
terraform destroy
```

5. Terraform Architecture

Terraform follows a client-only architecture:

1. Terraform CLI – The command-line tool that parses configurations.
2. Providers (AWS, Azure, GCP, etc.) – Plugins that interact with cloud APIs.
3. State File (**terraform.tfstate**) – Tracks resource metadata and dependencies.

4.Backend (Remote Storage) – Stores state files remotely (e.g., S3, Terraform Cloud).

6. Advantages of Terraform

Declarative Syntax – Define "what" infrastructure should look like, not "how."
 Immutable Infrastructure – Changes create new resources instead of modifying existing ones.
Modularity – Reusable modules for standardized deployments.
Community & Ecosystem – Large collection of modules and integrations.

7. Challenges & Considerations

- State Management – Corruption or loss of state files can cause issues.
- Learning Curve – Requires understanding of HCL and cloud providers.
- Cost Management – Uncontrolled deployments may lead to unexpected costs

---

Project explanation :

code by code explanation

A detailed explanation of your Terraform script, which provisions **3 Ubuntu EC2 instances** in AWS and configures them with a **security group** allowing various ports.

Main.tf

1. AWS Provider Configuration

```
provider "aws" {

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

```
}
```

explanation

Specifies you're using AWS as the cloud provider.

- Region is set to **us-east-1** (Northern Virginia).

2. Fetch the Default VPC

```
data "aws_vpc" "default" {
  default = true
}
```

explanation :

Fetches information about the default VPC in your AWS account.

- This is used to attach resources like security groups to the correct network.

3. Security Group Creation

```
# Security Group
resource "aws_security_group" "ubuntu_sg" {
  name        = "ubuntu-sg"
  description = "Allow common ports"
  vpc_id      = data.aws_vpc.default.id
```

```
ingress {

  from_port   = 22

  to_port     = 22

  protocol    = "tcp"

  cidr_blocks = ["0.0.0.0/0"]  # SSH

}



ingress {

  from_port   = 80

  to_port     = 80

  protocol    = "tcp"

  cidr_blocks = ["0.0.0.0/0"]  # HTTP

}



ingress {

  from_port   = 443
```

```
  to_port     = 443

  protocol    = "tcp"

  cidr_blocks = ["0.0.0.0/0"]  # HTTPS

}



ingress {

  from_port   = 8080

  to_port     = 8080

  protocol    = "tcp"

  cidr_blocks = ["0.0.0.0/0"]

}



ingress {

  from_port   = 5000

  to_port     = 5000

  protocol    = "tcp"

  cidr_blocks = ["0.0.0.0/0"]
```

```
}

ingress {
  from_port   = 6664
  to_port     = 6664
  protocol    = "tcp"
  cidr_blocks = ["0.0.0.0/0"]
}

ingress {
  from_port   = 0
  to_port     = 65535
  protocol    = "tcp"
  cidr_blocks = ["0.0.0.0/0"]  # All TCP (optional if you want full open)
}

egress {
```

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

  tags = {
    Name = "Ubuntu Security Group"
  }
}
```

explanation :

This block creates a security group named `ubuntu-sg`, which is like a firewall that controls inbound/outbound traffic.

Ingress (Inbound Rules):

Port	Protocol	Purpose
22	TCP	SSH access
80	TCP	HTTP (Web)
443	TCP	HTTPS (Secure Web)
8080	TCP	Web apps (like Tomcat, etc.)
5000	TCP	Flask or custom apps
6664	TCP	Possibly custom/internal usage
0-65535	TCP	**All TCP traffic allowed** (dangerous for production!)

Note: Allowing all TCP ports (`0-65535`) is useful for testing but insecure for production

Egress (Outbound Rule):

```
 egress {

    from_port   = 0

    to_port     = 0

    protocol    = "-1"

    cidr_blocks = ["0.0.0.0/0"]

  }
```

Allows all outbound traffic, which is the default in AWS.

4. EC2 Instances Creation

```
# 3 EC2 Instances with Ubuntu

resource "aws_instance" "ubuntu_instance" {

  count         = 3

  ami           = "ami-084568db4383264d4"  # Your AMI ID for Ubuntu 22.04 in us-
east-1

  instance_type = "t2.micro"

  key_name      = "hello"  # Your key pair name without the .pem extension
```

```
  vpc_security_group_ids = [aws_security_group.ubuntu_sg.id]

  root_block_device {
    volume_size = 20   # 20 GB EBS volume
    volume_type = "gp2"
  }

  tags = {
    Name = "Ubuntu-TF-${count.index + 1}"
  }
}
```

explanation :

Creates 3 EC2 instances using count = 3.

Attribute	Description
ami	Specifies the Amazon Machine Image (Ubuntu 22.04 in your case)
instance_type	t2.micro (Free Tier eligible, 1 vCPU, 1 GB RAM)
key_name	"hello" — refers to your existing AWS key pair
vpc_security_group_ids	Associates the instance with the ubuntu_sg security group
root_block_device	Attaches a 20 GB EBS volume to each instance
tags.Name	Tags instances as Ubuntu-TF-1, Ubuntu-TF-2, and

Attribute	Description
	Ubuntu-TF-3

Output.tf

```
output "instance_ips" {
  value = [for instance in aws_instance.ubuntu_instance : instance.public_ip]
}
```

It is an output block in terraform

Purpose of this Output Block

After Terraform finishes provisioning your EC2 instances, this block:

- **Extracts the public IP addresses** of all 3 EC2 instances.
- **Displays them in the terminal output** once terraform apply is completed.

Breakdown of the Code

Part	Meaning
output "instance_ips"	Names the output variable instance_ips
for instance in aws_instance.ubuntu_instance	Loops over all 3 EC2 instances
instance.public_ip	Grabs the public IP of each instance
[...]	Collects the results into a list

Why It's Useful

- You **don’t need to go to AWS Console** to check the public IPs.
- Makes it easy to SSH into your instances or connect via browser.
- You can even use this output in other Terraform modules or scripts.

Output After terraform apply

```
Outputs:

instance_ips = [
  "3.88.222.100",
  "54.172.44.198",
  "52.90.176.23",
]
```

Now you can SSH like:

```
ssh -i hello.pem ubuntu@3.88.222.100
```

The file terraform.tfstate is **Terraform's state file**. It **tracks the actual infrastructure** Terraform has created in your cloud environment (like AWS).

Why It's Important

Terraform needs to **know what it has already created** so that it can:

- **Avoid duplicating resources**.
- **Compare changes** in your code (.tf files) with what is deployed.
- **Apply only the necessary updates**.

This file acts like Terraform's **memor**

Example:

Let’s say your Terraform code defines 3 EC2 instances.

- After terraform apply, those instances are created.
- Terraform saves their details (like IDs, IPs, and tags) in terraform.tfstate.
- Later, if you change the instance type or add another one, Terraform will **compare your updated code with the state file** and only apply the changes.

Important Notes

- **Do not manually edit** terraform.tfstate — it can break your setup.
- If you're working in a team or using automation (like GitHub Actions), consider using **remote state** (e.g., AWS S3 + DynamoDB) to store it safely and avoid conflicts.

Where is it stored?

By default, it's saved locally in your project folder unless you configure a remote backend.

terraform.tfstate.backup is an **automatic backup** of the previous state file — terraform.tfstate — created by Terraform **before it updates the current state**.

When is it created?

- Every time you run terraform apply or terraform refresh, Terraform makes changes to your infrastructure.
- Before it **writes the new state to terraform.tfstate**, it **saves the previous version** as terraform.tfstate.backup.

✅ Why it's useful:

Purpose	Explanation
Recovery	If something goes wrong (like corrupted state or wrong changes), you can **manually restore** the previous state.
Debugging	Helps you **compare old vs new** state for troubleshooting.
Safety	Prevents **accidental loss** of the last good infrastructure state.

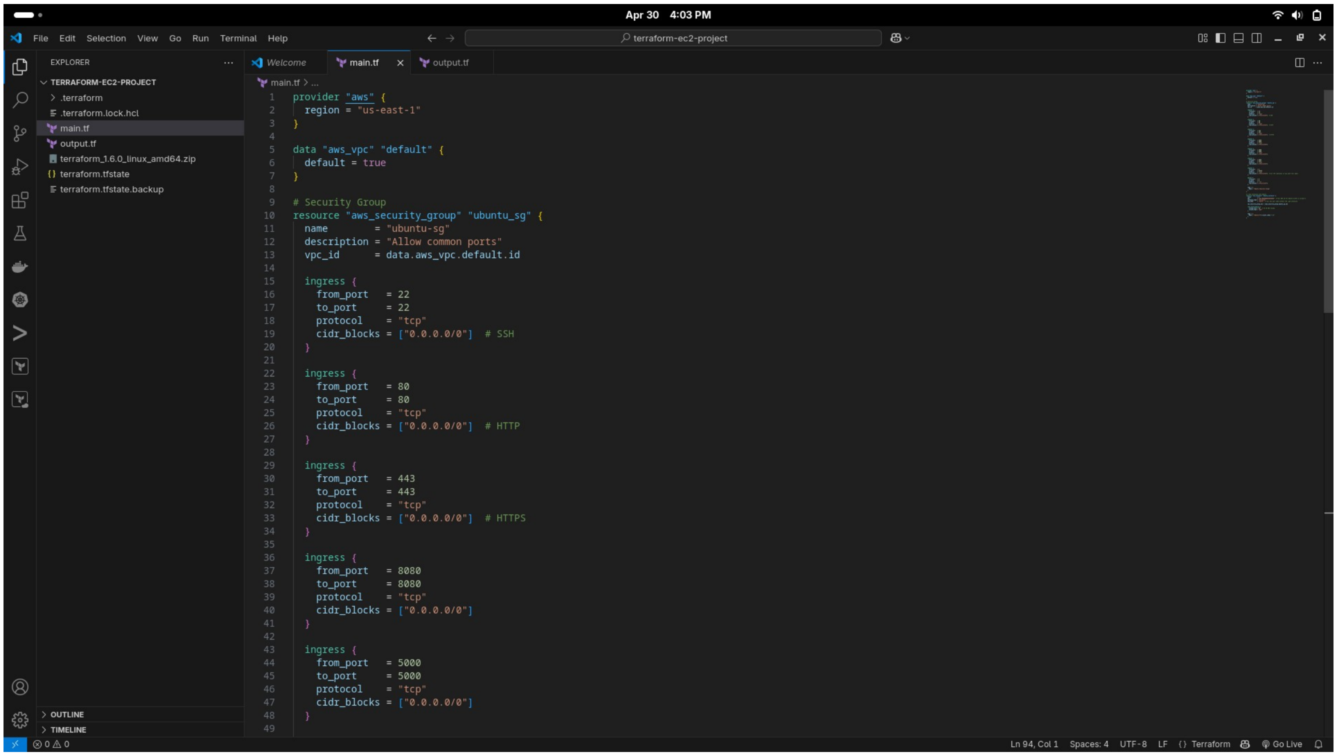
Example Scenario:
Let’s say:

1. You run terraform apply and it updates your EC2 instances.
2. The new state is saved in terraform.tfstate.
3. The old state is automatically saved as terraform.tfstate.backup.

If needed, you can **restore it manually** like this:

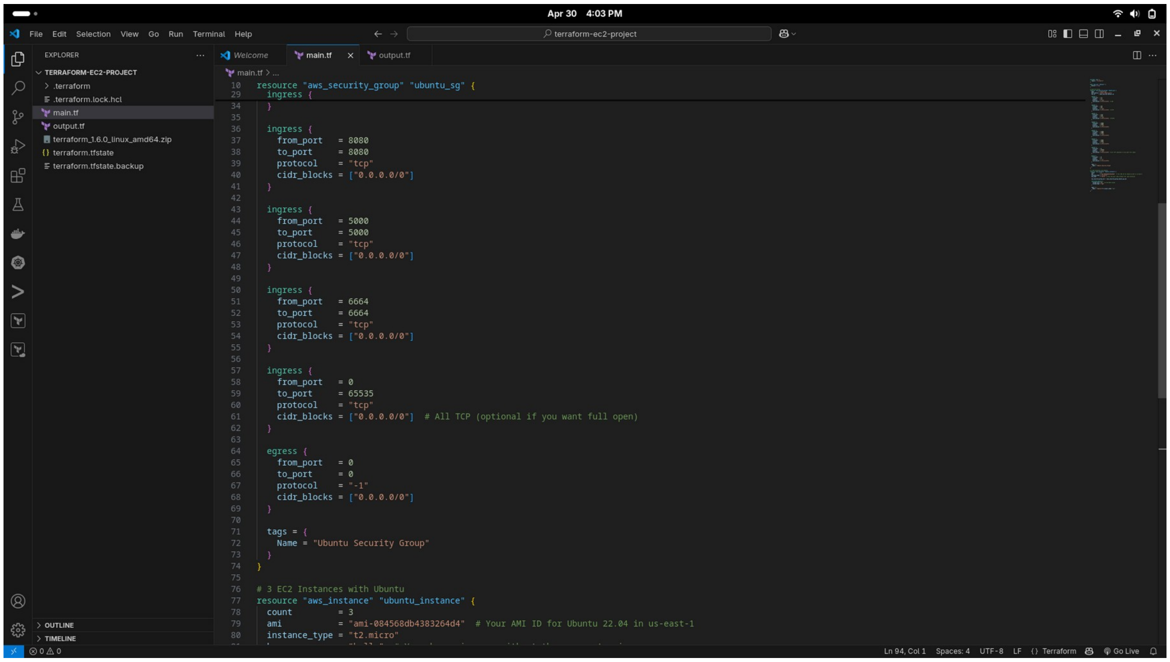
```
cp terraform.tfstate.backup terraform.tfstate
```

Images of project

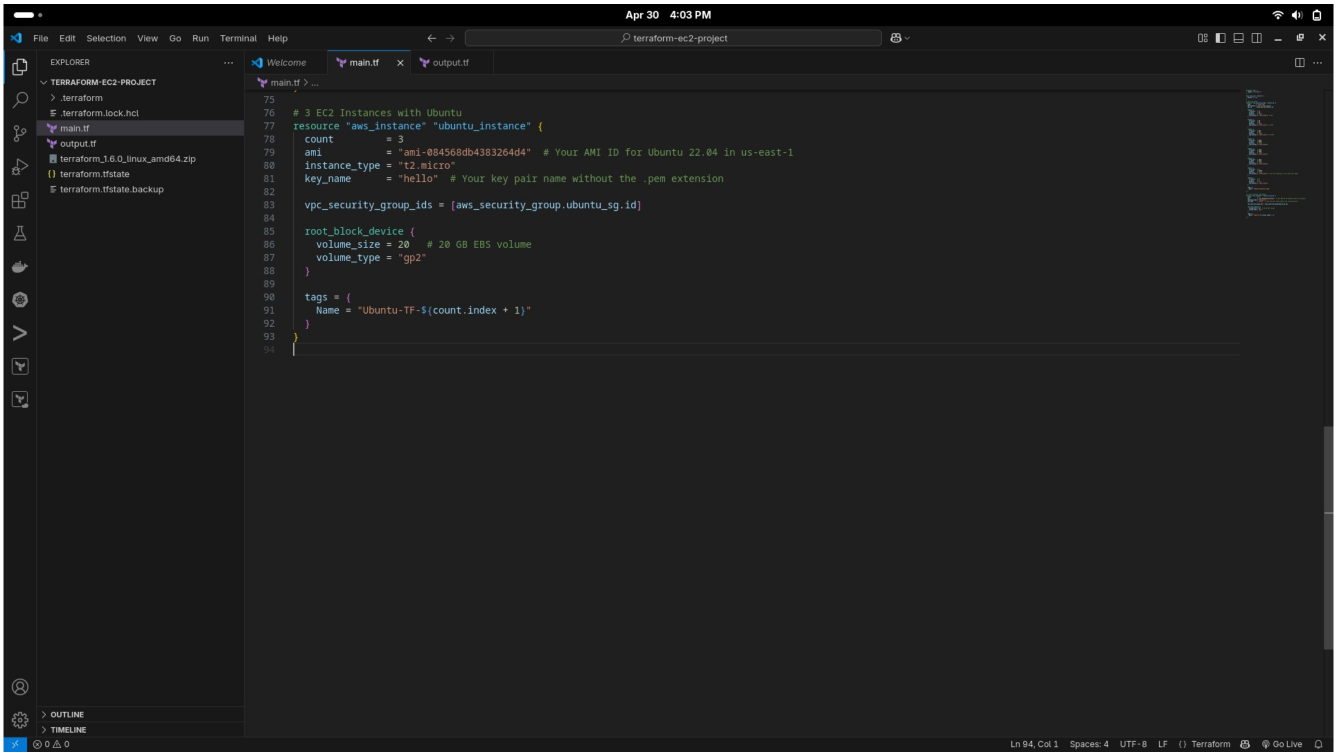


```
provider "aws" {
  region = "us-east-1"
}

data "aws_vpc" "default" {
  default = true
}

# Security Group
resource "aws_security_group" "ubuntu_sg" {
  name        = "ubuntu-sg"
  description = "Allow common ports"
  vpc_id      = data.aws_vpc.default.id

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

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

  ingress {
    from_port   = 443
    to_port     = 443
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]  # HTTPS
  }

  ingress {
    from_port   = 8080
    to_port     = 8080
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  ingress {
    from_port   = 5000
    to_port     = 5000
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

```

Fig : 1

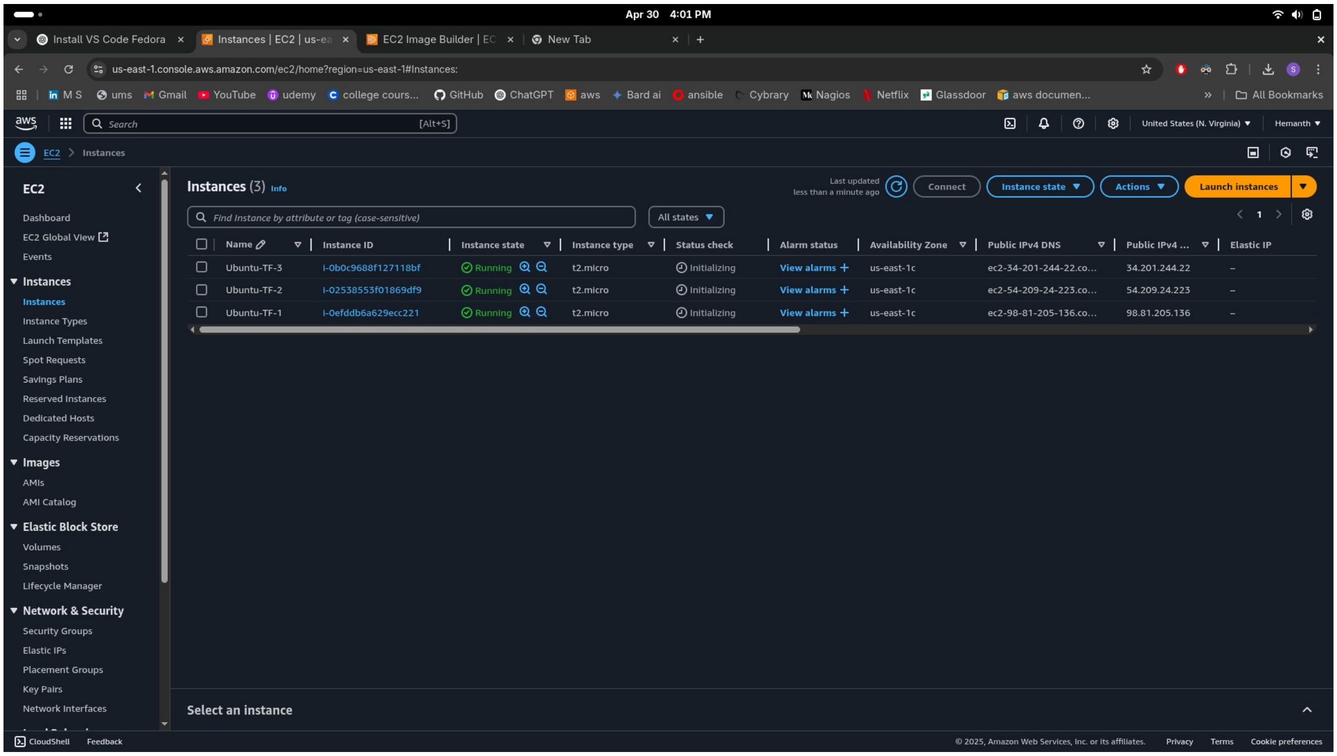


```
resource "aws_security_group" "ubuntu_sg" {
  ingress {
  }

  ingress {
    from_port   = 8080
    to_port     = 8080
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  ingress {
    from_port   = 5000
    to_port     = 5000
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  ingress {
    from_port   = 6664
    to_port     = 6664
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  ingress {
    from_port   = 0
    to_port     = 65535
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]  # All TCP (optional if you want full open)
  }

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

  tags = {
    Name = "Ubuntu Security Group"
  }
}

# 3 EC2 Instances with Ubuntu
resource "aws_instance" "ubuntu_instance" {
  count         = 3
  ami           = "ami-084568db4383264d4"  # Your AMI ID for Ubuntu 22.04 in us-east-1
  instance_type = "t2.micro"
```



```
# 3 EC2 Instances with Ubuntu
resource "aws_instance" "ubuntu_instance" {
  count         = 3
  ami           = "ami-084568db4383264d4"  # Your AMI ID for Ubuntu 22.04 in us-east-1
  instance_type = "t2.micro"
  key_name      = "hello"  # Your key pair name without the .pem extension

  vpc_security_group_ids = [aws_security_group.ubuntu_sg.id]

  root_block_device {
    volume_size = 20   # 20 GB EBS volume
    volume_type = "gp2"
  }

  tags = {
    Name = "Ubuntu-TF-${count.index + 1}"
  }
}
```

Fig : 3



Instances (3)

Name	Instance ID	Instance state	Instance type	Status check	Alarm status	Availability Zone	Public IPv4 DNS	Public IPv4 ...	Elastic IP
Ubuntu-TF-3	i-0b0c9688f127118bf	Running	t2.micro	Initializing	View alarms +	us-east-1c	ec2-34-201-244-22.co...	34.201.244.22	–
Ubuntu-TF-2	i-02538553f01869df9	Running	t2.micro	Initializing	View alarms +	us-east-1c	ec2-54-209-24-223.co...	54.209.24.223	–
Ubuntu-TF-1	i-0efddb6a629ecc221	Running	t2.micro	Initializing	View alarms +	us-east-1c	ec2-98-81-205-136.co...	98.81.205.136	–

fig : 4 aws ec2 instances created

commands

```
terraform init
terraform plan
terraform apply
```

Conclusion :

Terraform revolutionizes infrastructure management by enabling IaC, reducing manual efforts, and improving scalability. Its multi-cloud support, modularity, and automation capabilities make it a preferred choice for DevOps teams.

In this project, we successfully used **Terraform** to automate the provisioning of **three EC2 Ubuntu instances** within AWS. A custom **security group** was also created to allow access on essential ports like SSH (22), HTTP (80), HTTPS (443), and custom ports such as 5000 and 8080, ensuring flexibility for various use cases like web servers or Flask apps.

Key achievements include:

- Setting up and configuring Terraform on Fedora.
- Creating a reusable Terraform script with clean infrastructure as code (IaC).
- Managing AWS credentials securely using the AWS CLI.
- Defining and applying AWS resources like VPC data source, security groups, and EC2 instances.
- Verifying the setup by outputting the public IPs of all instances.

This project demonstrates the power of **IaC** to streamline cloud resource management, making deployments repeatable, consistent, and scalable. It lays the foundation for more advanced DevOps workflows such as CI/CD pipelines, monitoring, and infrastructure scaling.