7 Days Terraweek Challenge: Day 1 - Getting Started with Terraform

Hello connections, It's my day 1 of the 7 days Terraweek challenge, where we will dive into the **basic concepts of Terraform.**

- What is Terraform and how does it revolutionize infrastructure management?
- Why do we need it and how does it simplify provisioning?
- Learn crucial Terraform terminologies with examples.
- Understand Terraform Lifecycle.
- Easily install Terraform and configure environments for AWS.

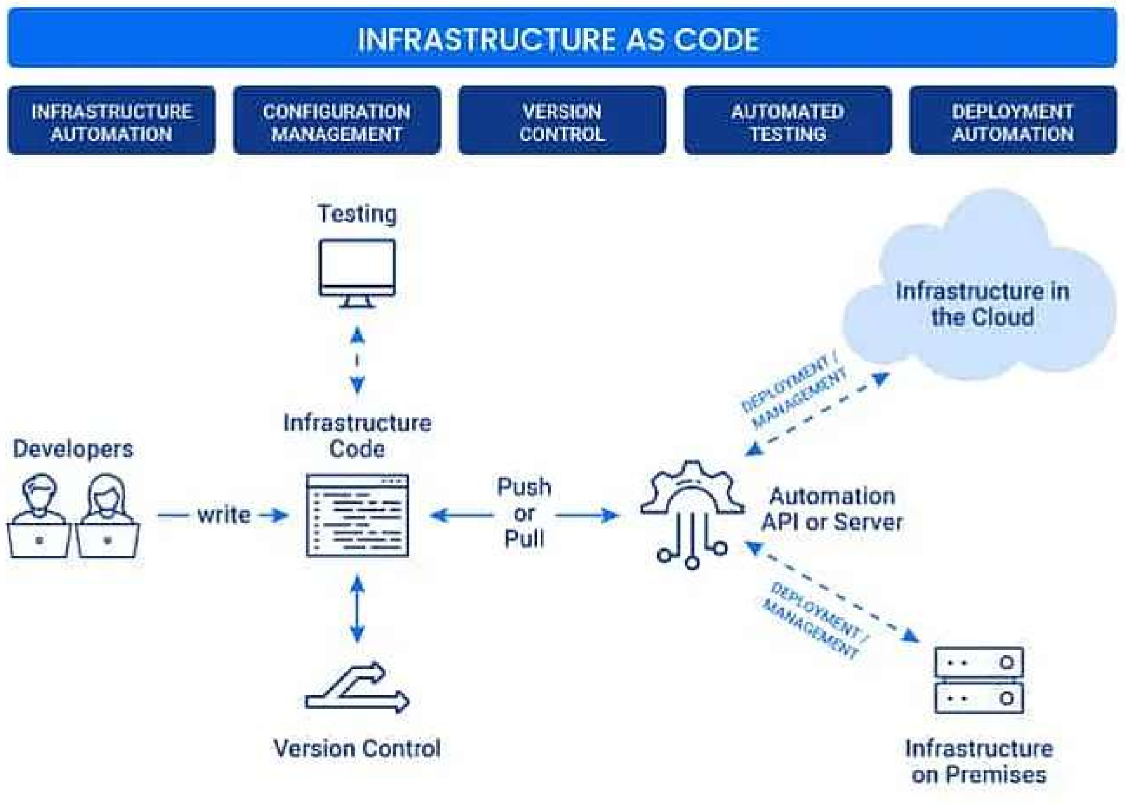
Before we deep dive into Terraform, let's first understand the **Infrastructure as Code (IaC).**

What Is Infrastructure as Code (IaC)?

Infrastructure as Code (IaC) is a fancy term that means **managing your infrastructure using code.** But what does that even mean? Think of it as a recipe for building things, but instead of cooking, we're talking about technical stuff like servers, networks, and databases.

Using IaC, you write instructions in a special language that tells your computer how to create and manage your infrastructure. It's like giving your computer a super detailed to-do list, and it follows those instructions to make your infrastructure dreams come true.

There are various IaC tools available like Terraform, Chef, Puppet, and Ansible. Read the blog to know why Terraform is preferred over other IaC tools.



Let’s now understand What Is Terraform?

What Is Terraform?

Terraform is like a magic wand that lets you **create and manage your infrastructure using code.** But what does that mean? Well, instead of clicking buttons or typing commands, you write simple instructions in a special language that Terraform understands. With Terraform, you can describe your desired infrastructure state in configuration files, similar to writing a recipe, and Terraform will work its magic to create and maintain your infrastructure.

Terraform is one of the most popular open-source Infrastructure-as-code (IaC) tools, used to automate infrastructure tasks. It is used to automate the provisioning of your cloud resources.

Why is Code Cool for Infrastructure?

You might wonder why we need to use code for infrastructure. Good question! Think of it this way: When you write code, you can **easily share it, keep track of changes, and work together with others.** With Terraform, you can do the same thing with your infrastructure. You have a set of instructions (your code) that you can use again and again to create awesome things (your infrastructure).

Why do we need Terraform and How Does it Simplify Infrastructure Provisioning?

Managing infrastructure manually is prone to human errors, is time-consuming, and lacks consistency. This is where Terraform comes to the rescue!

Here's why we need Terraform and how it simplifies infrastructure provisioning:

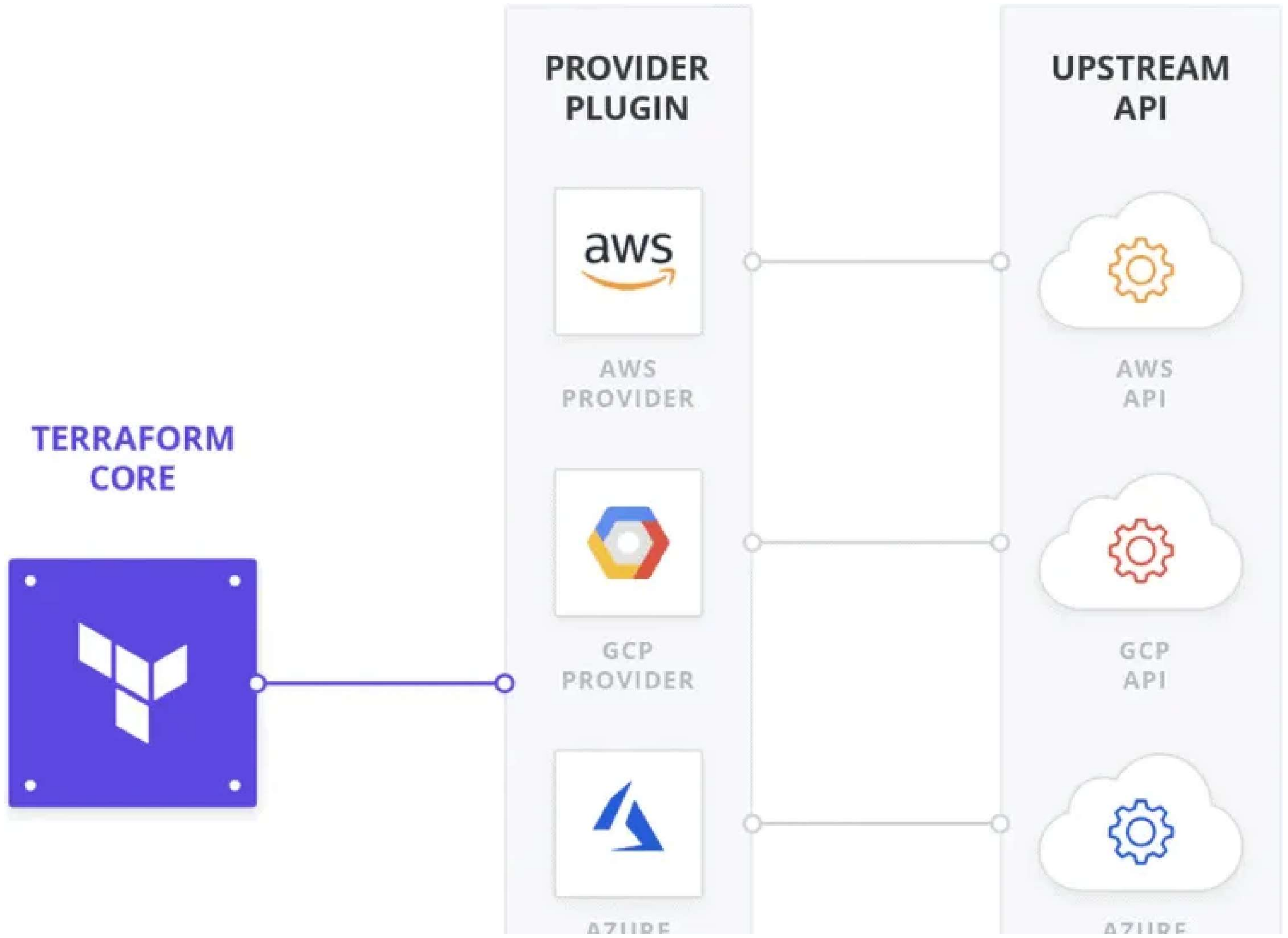
- **Declarative Configuration:** Terraform uses simple and readable language to describe your infrastructure requirements. Instead of worrying about the specific steps to create each resource, you can focus on what you want your infrastructure to look like. It's like telling Terraform your wish, and it grants it to you!
- **Automation Made Easy:** With Terraform, you can **automate the provisioning and management of your infrastructure.** That means you can sit back and relax while Terraform does the heavy lifting for you. It's like having a robot friend who follows your instructions flawlessly.
- **Multi-Cloud Magic:** Want to **use multiple cloud providers like AWS, Azure, or Google Cloud**? No problem! Terraform supports various cloud platforms, making it a breeze to manage resources across different providers. It's like having a universal remote control for your infrastructure.
- **Infrastructure as Code:** With Terraform, you can treat your infrastructure as code, just like you would with a cool video game. You write your infrastructure specifications in code, which brings a bunch of benefits. You can **version control your code, collaborate with others, and reuse code snippets.**
- **Plan and Apply:** Terraform helps you plan ahead before making any changes to your infrastructure. It **analyzes your code and shows you a preview of what will happen when you apply it.** This way, you can review and double-check everything, ensuring a smooth and error-free provisioning process. It's like having a virtual assistant who shows you a blueprint before building your dream house.
- **State Management:** Terraform **keeps track of the current state of your infrastructure in a file.** This state file is like a map that Terraform uses to understand what resources exist and their configurations. It helps Terraform make intelligent decisions when you modify or destroy resources. It's like having a memory bank for your infrastructure changes.

Common Terminologies in Terraform

- **Provider:** Imagine you want to build a house, but you need land to do so. In Terraform, a **provider** is like a landowner who gives you the resources to build your infrastructure. Providers are responsible for managing various cloud platforms or infrastructure providers, such as AWS, Azure, or Google Cloud. Terraform officially supports around 130 providers. Here's an example configuration block for the AWS provider:

```
provider "aws" {
  region = "us-west-2"
}
```

A provider is **responsible for understanding API interactions and exposing resources.** It interacts with the various APIs required to create, update, and delete various resources. Terraform configurations must declare which providers they require so that Terraform can install and use them.



- **Resource: Resources** are the building blocks of your infrastructure. They represent the individual components you want to create or manage, such as servers, databases, or networks. Resources are defined within a Terraform configuration file and are associated with a specific provider. Example:

```
resource "aws_instance" "web_server" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
}
```

- **Variable: Variables** are like placeholders that allow you to pass dynamic values to your Terraform code. They provide flexibility by allowing you to change values without modifying the code itself. Variables can be defined within your Terraform configuration files or in separate variable files. Example:

```
variable "region" {
  default = "us-west-2"
}
```

- **Module:** Think of a **module** as a ready-to-use blueprint for a specific set of infrastructure resources. It's like a pre-built package that you can reuse across different projects. Modules help keep your code organized and promote consistency in your infrastructure. Example:

```
module "vpc" {
  source = "terraform-aws-modules/vpc/aws"
  version = "2.0.0"

  # Additional configurations specific to your project
  vpc_name = "my-vpc"
  cidr_block = "10.0.0.0/16"
}
```

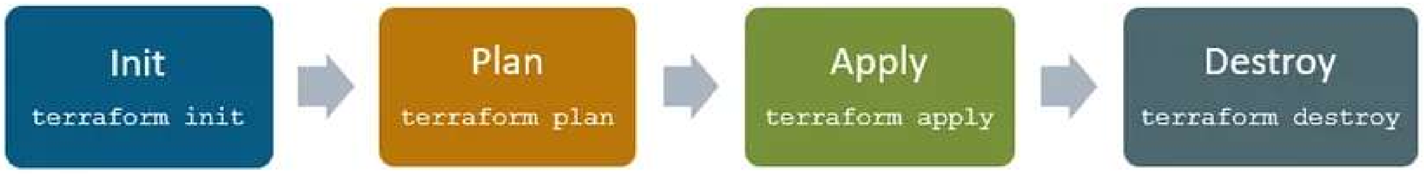
In this example, we're using a pre-built module from the Terraform Registry to create a Virtual Private Cloud (VPC). We specify the module source, and version, and provide additional configurations specific to our project.

- **State:** The **state** is like a memory bank for your infrastructure. It keeps track of the current state of your resources and their configurations. Terraform uses the state to understand what's already created, making it easier to manage and update your infrastructure. Example:

```
terraform {
  backend "s3" {
    bucket = "my-terraform-state"
  }
}
```

Terraform Lifecycle

Terraform lifecycle consists of — **init, plan, apply, and destroy.**



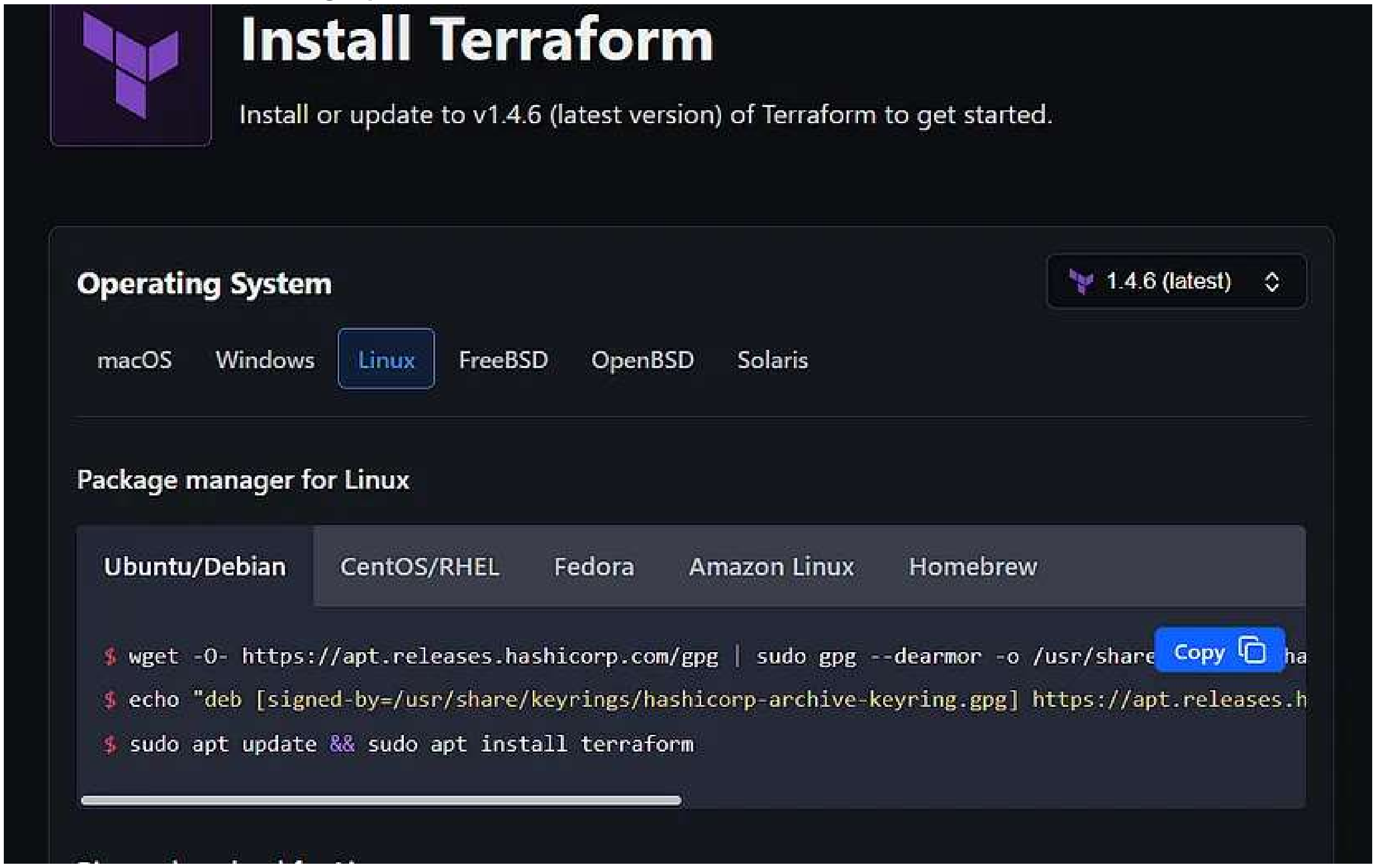
- **Terraform init:** initializes the (local) Terraform environment. Usually executed only once per session.
- **Terraform plan:** compares the Terraform state with the as-is state in the cloud, builds and displays an execution plan. This does not change the deployment (read-only).
- **Terraform apply:** executes the plan. This potentially changes the deployment.
- **Terraform destroy:** deletes all resources that are governed by this specific Terraform environment.

Steps to install Terraform and set up the environment for AWS

To install Terraform and set up the environment for AWS, you can follow these steps:

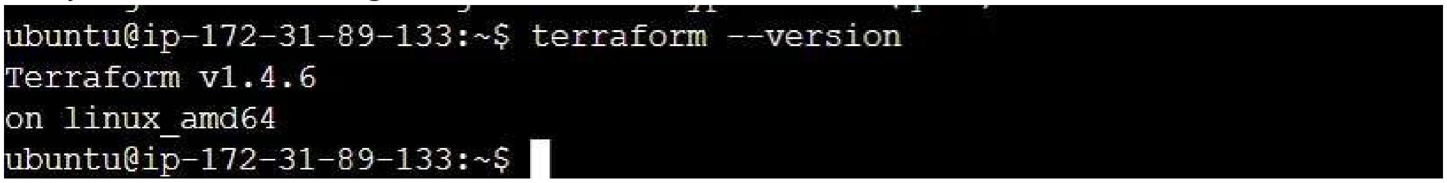
1. **Install Terraform:**
   - Download the Terraform binary suitable for your operating system from the official Terraform website (https://www.terraform.io/downloads.html).
   - Verify the installation using `terraform --version`.

In this demo, I'm setting up Terraform in an AWS EC2 instance(Ubuntu AMI):



```
$ wget -O- https://apt.releases.hashicorp.com/gpg | sudo gpg --dearmor -o /usr/share          ha
$ echo "deb [signed-by=/usr/share/keyrings/hashicorp-archive-keyring.gpg] https://apt.releases.h
$ sudo apt update && sudo apt install terraform
```

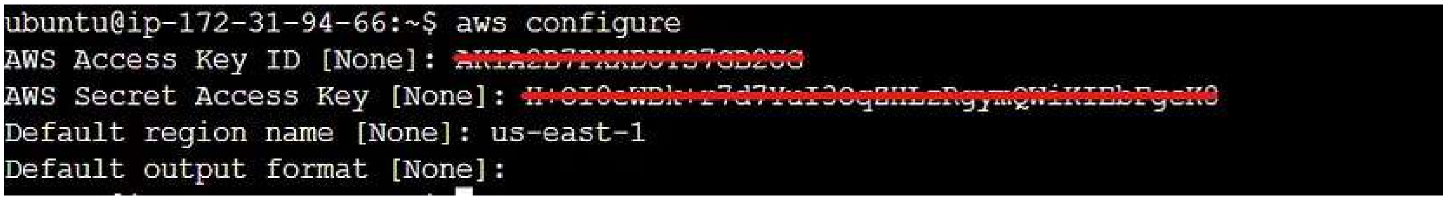
Verify the installation using **terraform — version:**



```
ubuntu@ip-172-31-89-133:~$ terraform --version
Terraform v1.4.6
on linux_amd64
ubuntu@ip-172-31-89-133:~$
```

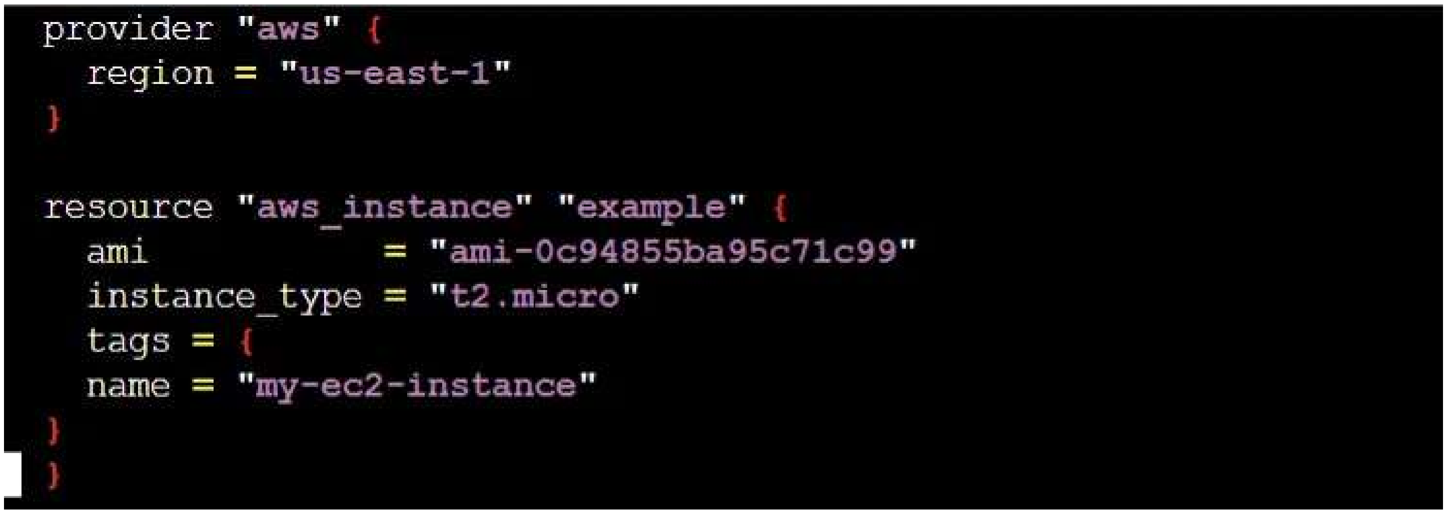
2. **Set Up AWS Credentials:**
   - Sign in to the AWS Management Console.
   - Create an IAM user or use an existing one with appropriate permissions for managing AWS resources. Ensure the user has permissions for EC2, VPC, S3, and other services you plan to work with.
   - Generate an access key and secret key for the IAM user. Make a note of these credentials as they will be required for Terraform to interact with AWS.
   - Configure the AWS CLI on your EC2 machine by running the following command in your terminal:

```
aws configure
```



```
ubuntu@ip-172-31-94-66:~$ aws configure
AWS Access Key ID [None]: [redacted]
AWS Secret Access Key [None]: [redacted]
Default region name [None]: us-east-1
Default output format [None]:
```

3. **Create Terraform configuration file:**
   - Create a new directory for your Terraform configuration files.
   - Open a terminal and navigate to the new directory.
   - Create a new Terraform configuration file with a .tf extension (e.g., main.tf). This file will contain your infrastructure code.
   - In the configuration file, specify the AWS provider and any resources you want to create or manage.

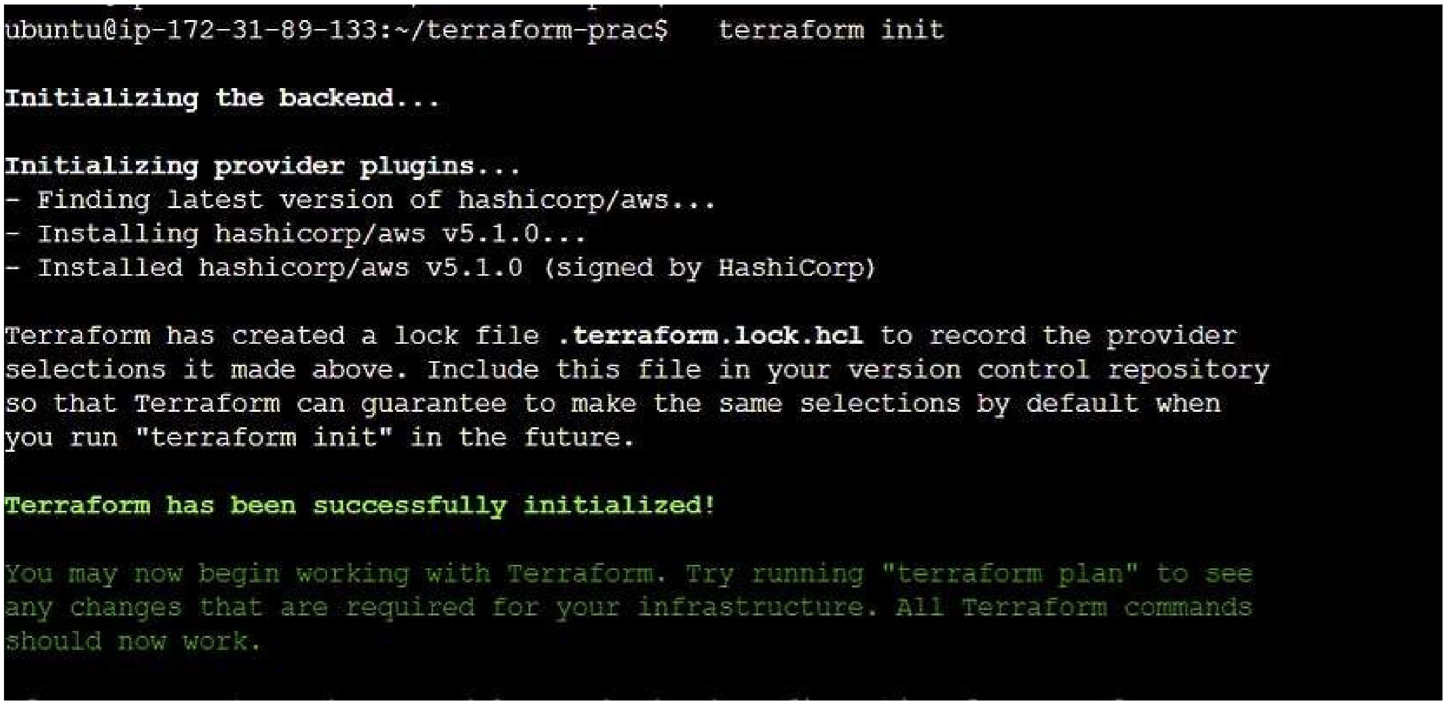


```
provider "aws" {
  region = "us-east-1"
}

resource "aws_instance" "example" {
  ami           = "ami-0c94855ba95c71c99"
  instance_type = "t2.micro"
  tags = {
  name = "my-ec2-instance"
}
}
```

4. **Initialize Terraform:**

- In the terminal, navigate to your Terraform directory (where your configuration file is located).
- Run the following command to initialize Terraform and download the necessary provider plugins:

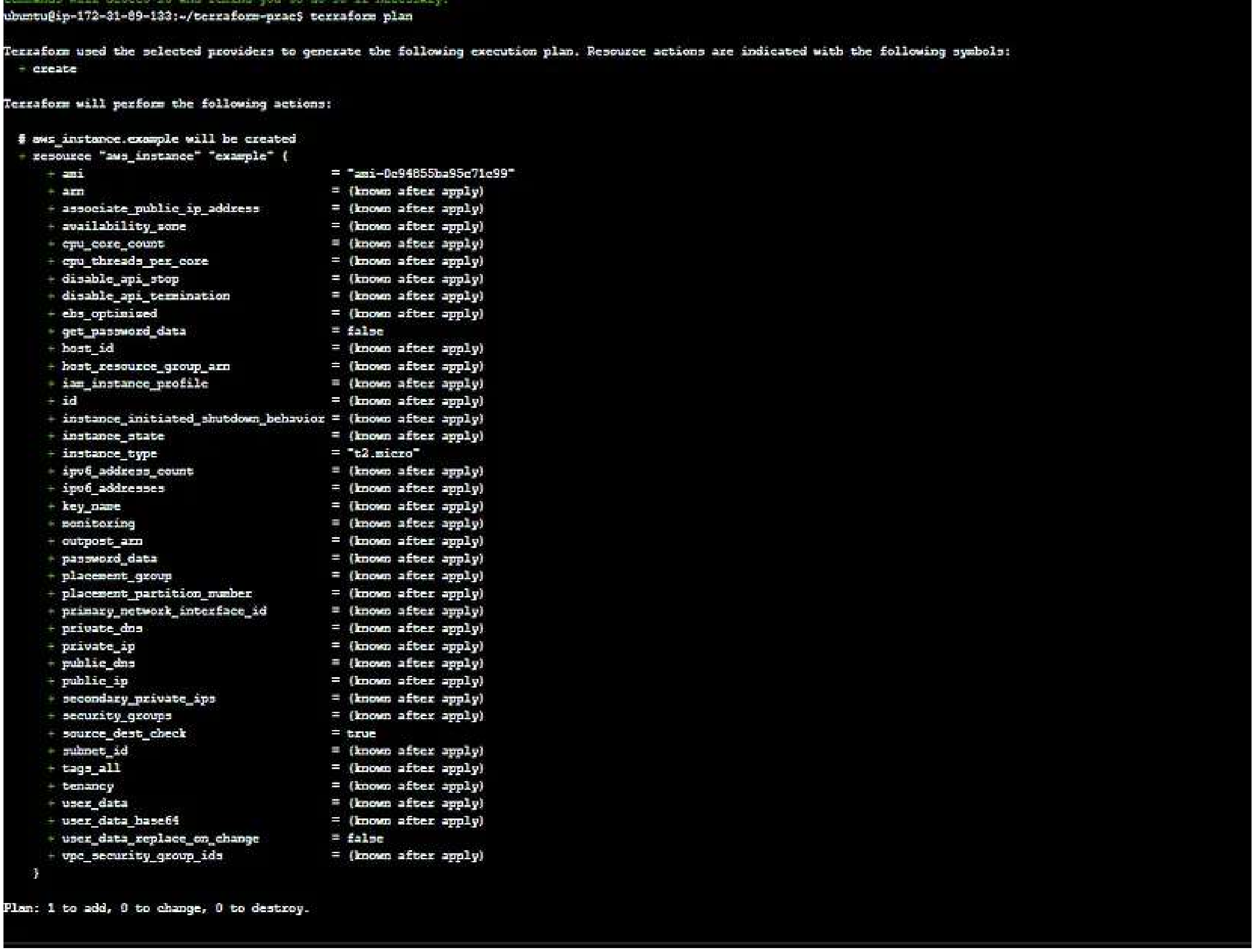
```
terraform init
```



```
ubuntu@ip-172-31-89-133:~/terraform-prac$   terraform init

Initializing the backend...

Initializing provider plugins...
- Finding latest version of hashicorp/aws...
- Installing hashicorp/aws v5.1.0...
- Installed hashicorp/aws v5.1.0 (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.
```

5. **Deploy Infrastructure:**
   - After initialization, run the following command to preview the changes Terraform will make to your AWS environment:

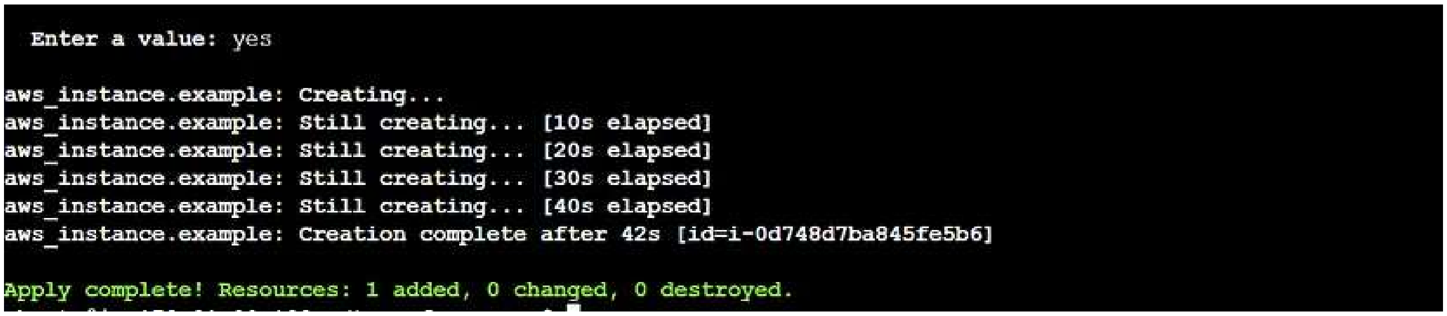
```
terraform plan
```



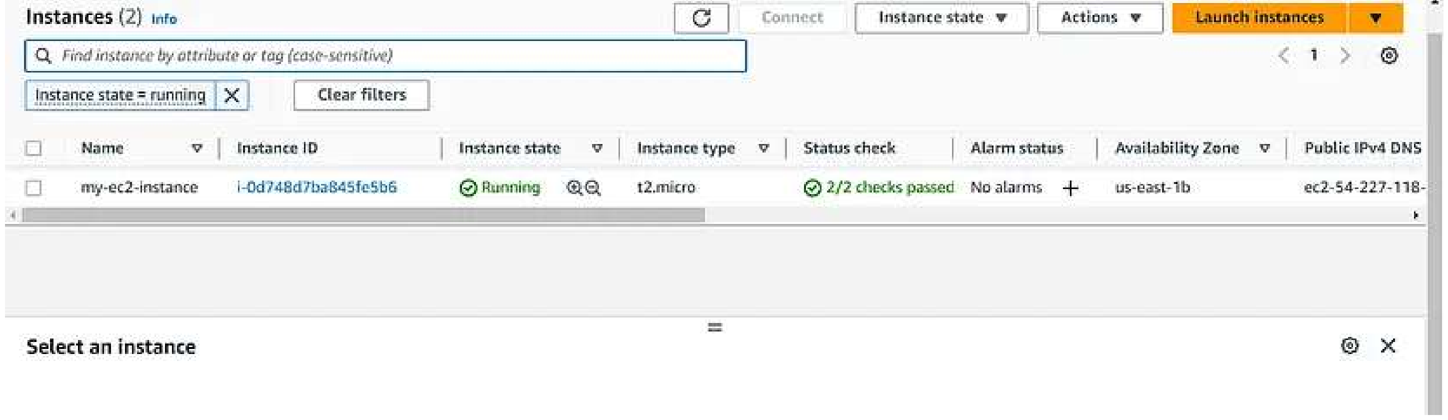
```
ubuntu@ip-172-31-89-133:~/terraform-prac$ terraform plan

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.example will be created
  + resource "aws_instance" "example" {
      + ami                                  = "ami-0c94855ba95c71c99"
      + 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_state                       = (known after apply)
      + instance_type                        = "t2.micro"
      + ipv6_address_count                   = (known after apply)
      + ipv6_addresses                       = (known after apply)
      + key_name                             = (known after apply)
      + monitoring                           = (known after apply)
      + outpost_arn                          = (known after apply)
      + password_data                        = (known after apply)
      + placement_group                      = (known after apply)
      + placement_partition_number           = (known after apply)
      + primary_network_interface_id         = (known after apply)
      + private_dns                          = (known after apply)
      + private_ip                           = (known after apply)
      + public_dns                           = (known after apply)
      + public_ip                            = (known after apply)
      + secondary_private_ips                = (known after apply)
      + security_groups                      = (known after apply)
      + source_dest_check                    = true
      + subnet_id                            = (known after apply)
      + tags_all                             = (known after apply)
      + tenancy                              = (known after apply)
      + user_data                            = (known after apply)
      + user_data_base64                     = (known after apply)
      + user_data_replace_on_change          = false
      + vpc_security_group_ids               = (known after apply)
    }

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

- Review the output and ensure it matches your intentions.
- If everything looks good, execute the following command to apply the changes and create the infrastructure:

```
terraform apply
```



```
  Enter a value: yes

aws_instance.example: Creating...
aws_instance.example: Still creating... [10s elapsed]
aws_instance.example: Still creating... [20s elapsed]
aws_instance.example: Still creating... [30s elapsed]
aws_instance.example: Still creating... [40s elapsed]
aws_instance.example: Creation complete after 42s [id=i-0d748d7ba845fe5b6]

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```



Name	Instance ID	Instance state	Instance type	Status check	Alarm status	Availability Zone	Public IPv4 DNS
my-ec2-instance	i-0d748d7ba845fe5b6	Running	t2.micro	2/2 checks passed	No alarms	us-east-1b	ec2-54-227-118-

Boom, our EC2 instance is up and running!

Conclusion

Terraform is the ultimate beginner's best friend for infrastructure provisioning. It **simplifies the process by allowing you to manage infrastructure with simple code, automate tasks, and work seamlessly across multiple cloud platforms.** With Terraform, you can embrace the power of declarative configuration, treat your infrastructure as code, and leverage state management for smooth operations. So, take a deep breath, embrace the magic of Terraform, and say goodbye to a tedious manual infrastructure setup. Happy provisioning!

The above information is up to my understanding. Suggestions are always welcome.

Thank you for reading! Connect me on LinkedIn: Rajlaxmi Rathore