What is Terraform & Infrastructure as Code (IaC)?

If you're just getting started in software engineering, or you've been around a long time, you've probably at least heard the terms "Infrastructure as Code" and "Terraform" mentioned. But what are they, and why are they important?

Terraform is an Infrastructure as Code (IaC) tool that allows engineers to define their software infrastructure in code. While the idea of "code" may not be novel to engineers; the ability to provision infrastructure this way is a powerful abstraction that enables managing large distributed systems at scale.

In this article, we'll take a look at what Infrastructure as Code and Terraform are, how they can help you in your work as a developer, and how you can get started using them.

Grab the Terraform cheat sheet
Check out the top 10 Terraform commands and get a full rundown of all the basic commands you need to get the most out of Terraform in our Terraform cheat sheet.

What is Infrastructure as Code?

Infrastructure as Code is a way of defining and managing your infrastructure using code, rather than manual processes like clicking through a UI or using the command line. This means that you can manage your infrastructure in the same way that you manage your application code – with version control, automation, and collaboration. In other words, infrastructure as code is a way of making your infrastructure more like software.

Historically, Infrastructure as Code has seen many iterations, starting with configuration management tools like CFEngine, Chef, Puppet, Ansible, and Salt. Newer tooling like Cloudformation and Terraform take a declarative approach and focus on the actual provisioning of resources, as opposed to the configuration of existing ones. The newest generation of tools focuses on using the capabilities of existing imperative programming languages. AWS and Terraform both provide Cloud Development Kits(CDKs), and Pulumi is also a popular option for provisioning infrastructure with traditional software tools.

What is Terraform?

Terraform is a tool for provisioning, managing, and deploying infrastructure resources. It is an open-source tool written in Golang and created by the HashiCorp company. With Terraform, you can manage infrastructure for your applications across multiple cloud providers – AWS, Azure, GCP, etc. – using a single tool.

To get started with Terraform, developers simply need to download the Terraform binary, choose which provider/platform they'll be working with, create some boilerplate configuration for that provider, and they can get started creating infrastructure code.

One of the key features of Terraform is its declarative syntax. This means that you define what your desired end state is, and Terraform figures out the best way to achieve that. Compared to the imperative workflow of traditional programming languages, this can be a bit of a shift in mindset – but it enables managing infrastructure deployments at scale without the steep learning curve typical to software development.

The typical workflow for provisioning resources with Terraform is as follows:

1. Some Terraform configuration is written, including the provider definition.
2. The working directory is initialized(this is called the root module).
3. Provider plugins are downloaded.
4. The command `terraform plan` is run in the root module, generating a proposed plan to provision resources.
5. If the plan is acceptable, the `terraform apply` command is run and resources are provisioned.

Terraform keeps track of the state of the resources it manages in a state file. This file is essentially a large JSON data structure that tracks proposed changes to infrastructure, as well as out-of-band changes that may have occurred to live resources outside of the Terraform configuration. New Terraform users typically maintain a local state file on their workstation or laptop. At scale with multiple engineers managing infrastructure, the state is typically broken down into multiple files and stored remotely using services like AWS S3.

Terraform is also "idempotent", which means that repeated plan/apply cycles will not trigger a re-deploy of resources; only changes to the existing state will be reflected in a new plan and apply invocation.

Now that you've learned a little bit about what Terraform is and what it does, we can start to explore the "why" of Terraform and the benefits it provides.

What are the benefits of Terraform?

Utilizing Infrastructure as Code to manage and deploy infrastructure resources unlocks several benefits for developers and engineers. Making Terraform your tool of choice for IaC confers additional benefits that allow developers to leverage easy-to-use tools to manage complex software architecture.

1. Terraform configuration is written using a declarative paradigm.

The best way to understand declarative code is to compare it to the more familiar pattern of imperative logic that most modern programming languages use.

Imagine a basic Python program that takes a series of numbers as input from the user, prints each number out in ascending order, then returns the sum of all the inputs. A programmer needs to write the code in a specific, logical order, or the program will fail. If the program attempts to sum all the numbers before the input is received, it is likely to generate some kind of exception or error. If the programmer wants the numbers to be sorted into the correct order, that will need to occur before they are printed as output.

In each part of the program, the programmer is having to specifically define both the logic and order of logic when the program executes. In contrast, declarative program syntax means the programmer needs to only “declare” the desired end state of the program. The compiler, in this case, the Terraform binary, is programmed to determine the best order of operations path through which they achieve the desired end state described in the configuration. This is especially helpful in dealing with cloud provider APIs, as many cloud resources have dependencies on the creation of other foundational resources before creation can proceed.

2. Hashicorp Configuration Language (HCL) is a Domain Specific Language (DSL).

Domain-specific languages are designed with a specific use case in mind, specialized to handle the requirements and constraints of a specific application or program domain.

If the utility of a DSL doesn’t seem immediately obvious, you should consider one of the most famous and widely used DSLs: HTML. HTML is a markup language that focuses on the domain of hyper-text (read: the internet). HTML does away with complex program logic and syntax and focuses on the specific use-case of content presentation.

In the case of Terraform, developers only need to learn a minimal amount of HCL syntax before they can be productive. As a DSL, Terraform makes development easier and more efficient by abstracting away the complexity inherent in general-purpose languages. That’s not to say that Terraform doesn’t have more complex logic; advanced users can use newer, imperative constructs like for-loops and if/then logic.

HCl is considered a superset of the JSON language, which means it shares similar syntax, but also has additional features beyond the scope of JSON.

3. Terraform is widely adopted

Terraform is generally considered the industry standard when it comes to Infrastructure as Code tooling. It isn’t always good advice to go with the herd, but when it comes to technology implementation, choosing a tool with a large community, solid support base, and multi-year longevity is critically important. No one wants to have to explain to stakeholders and customers that the application is down because the code or platform is no longer supported!

The other benefit of wide adoption is the collective, shared knowledge of the community. Best practices are developed and shared, and an ecosystem of supporting tools and documentation can be built. A great example of the power of community support is the awesome-terraform repository on Github, a curated list of tools, libraries, documentation, blogs, and more.

4. Terraform enables immutability

Immutable infrastructure makes managing complex distributed systems easier and safer and allows them to scale much more reliably. What exactly is meant by “immutable infrastructure”, and how does Terraform enable it?

Consider a hypothetical scenario: a developer needs to deploy some changes to fix a production application. They create their changes locally, run some tests and linting to validate the changes and check syntax, and now they’re ready to deploy. However, to get their code to run in the staging environment, they have to change some configuration to point to the staging database. Then they need to give the QA team access to the staging servers. Everything manages to check out in the staging environment, but when deploying to production disaster strikes, the application stops serving traffic and an outage occurs.

Was the original code bad? Or was it the changes made in staging? Because the lines between environments and stages were blurred, it's nearly impossible to tell. Mutable infrastructure means changes can occur at any point in the lifecycle of an application or its infrastructure.

With immutable infrastructure, build, release, and deploy stages are kept separate. Once code changes are built, they are stamped with an immutable release tag. Further changes or fixes result in a new tag being generated. Developers and engineers know that a change that was made in a local development environment is the same change across different environments and deployment stages. The 12-factor app framework highlights this pattern in factor V.

5. Terraform is modular

Modularity is an important feature in a variety of languages and systems. Abstracting logic and resources behind simple interfaces is one of the best ways to manage complexity at scale. As Terraform deployments grow more complex, developers can consider employing modules to encapsulate various resources in a reusable package.

A common use case is providing other development teams with Kubernetes clusters for testing and development. Normally, provisioning a Kubernetes cluster in a cloud provider like AWS requires a lot of boilerplate configuration and resources, even when using managed services. With modules, that configuration can be hidden behind a basic configuration interface. Developers can import the module, specify whatever inputs the module author has provided, and they can provision a complete stack without having to duplicate effort needlessly.

How can I use Terraform?

Although it's simple to get started, Terraform is a very powerful tool for provisioning infrastructure. The key for new users is to start small with basic configuration, and work towards fully automating their infrastructure as code.

New developers should start with something simple; a couple of basic resources without complex dependency chains. You don't need to start by trying to manage your entire production application environment on day 1: try using Terraform to manage the S3 bucket where deployment artifacts are stored, or Google DNS records for a frontend website.

Once you're comfortable, you can start to iterate and grow your Terraform usage. One of the first steps is to move from local state to a remote state file with locking. It's virtually impossible to safely manage larger Terraform deployments with multiple users without the state having a locking mechanism. When a Terraform plan or apply is running, locks prevent other users from making changes that could result in an inconsistent state or corruption.

With multiple users now contributing Terraform configuration, you can start to expand usage to cover more and more of your infrastructure resources, including networking, security, CDN, and more. As your infrastructure increases in complexity, consider encapsulating certain segments of your architecture in modules.

Finally, your team can start to use Terraform to provision resources as part of your CI/CD strategy. CI/CD is a complex topic that we won't cover here, but GitLab and GitHub both provide great batteries-included solutions for deployment automation that can be used with Terraform. Hashicorp provides solid documentation specifically targeted at users who are considering automating their Terraform deployments.

Conclusion

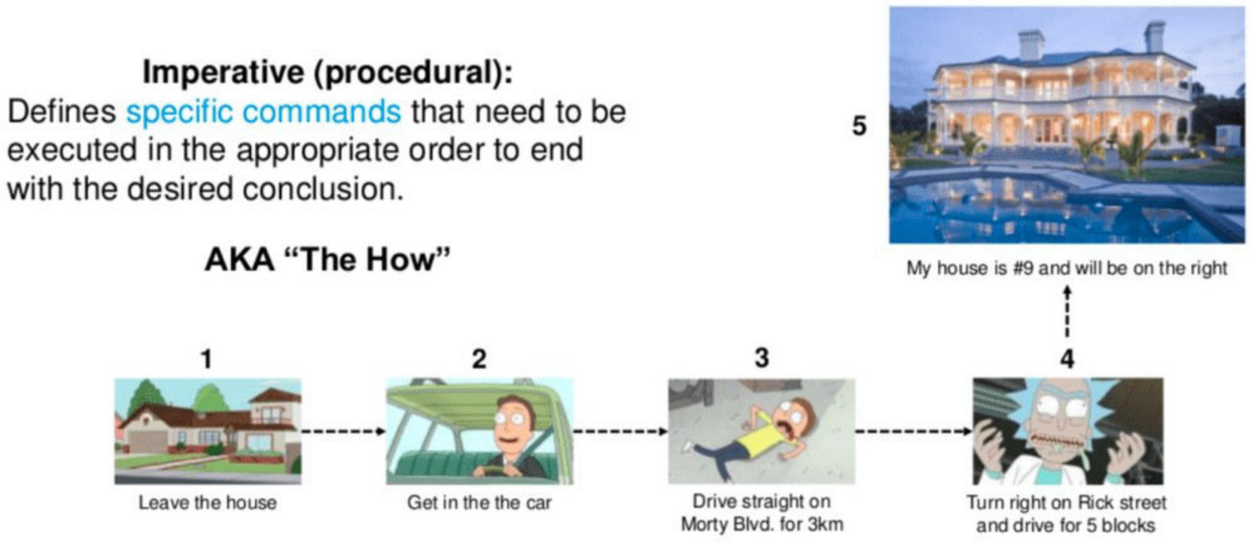
Why provision infrastructure with clicks and manual processes while writing application code? Infrastructure as Code tools like Terraform means that infrastructure configuration can be brought into the same development processes, allowing for testing, standardization, and scalability. The modern Infrastructure as Code ecosystem has a broad variety of resources for learning and getting started.

Want to learn more about Terraform?

Why not try studying for the Hashicorp Certified: Terraform associate certification? Pluralsight offers an excellent cert prep course which teaches you everything you need to know about Terraform, including how to use it effectively in your own projects. It's a great way to learn about Terraform, even if you don't end up taking the certification exam.

Infrastructure as Code (IaC): Declarative vs Imperative

IaC: Imperative


Imperative (procedural):
Defines specific commands that need to be executed in the appropriate order to end with the desired conclusion.
AKA "The How"
5
My house is #9 and will be on the right
1
Leave the house
2
Get in the the car
3
Drive straight on Morty Blvd. for 3km
4
Turn right on Rick street and drive for 5 blocks

Imperative – focus on the **actual provisioning process** and may reference a file containing a list of settings and configuration values.

Declarative – focused on the **desired end state** of deployment and relies on an interpretation engine to create and configure the actual resources.

When to use IaC

- You use a large amount of IaaS resources.
- Your infrastructure is rented from many different providers or platforms.

- You need to make regular adjustments to your infrastructure.
- You need proper documentation of changes made to your infrastructure.
- You want to optimize collaboration between administrators and developers.

Advantages of IaC

- Minimizing shadow IT inside of businesses and enabling quick, effective infrastructure upgrades made concurrently with application development
- Creating trackable and auditable configurations by enabling version-controlled infrastructure and configuration modifications.
- Maintaining infrastructure and settings in the appropriate condition while effectively managing configuration drift.
- Connecting directly to platforms for CI/CD.

Declarative (functional):
Defines the desired state and the system executes what needs to happen to achieve that desired state.

AKA "The What"

IaC Tools & Platforms – Overview

Terraform

The top infrastructure as code (IaC) solution for managing infrastructure across a variety of clouds, including AWS, Azure, GCP, Oracle Cloud, Alibaba Cloud, and even platforms like Kubernetes and Heroku, is Terraform by HashiCorp.

While assuring the intended state throughout the settings, Terraform may be utilized to facilitate any infrastructure provisioning and management use cases across many platforms and providers.

Ansible

Ansible is a free, open-source configuration management solution with IaC capabilities rather than a specialized infrastructure management tool. It is an agentless solution that works with both cloud and on-premises systems and may be used by SSH or WinRM.

It has limitations when it comes to maintaining that infrastructure but shines at provisioning infrastructure and configuration management.

Chef/Puppet

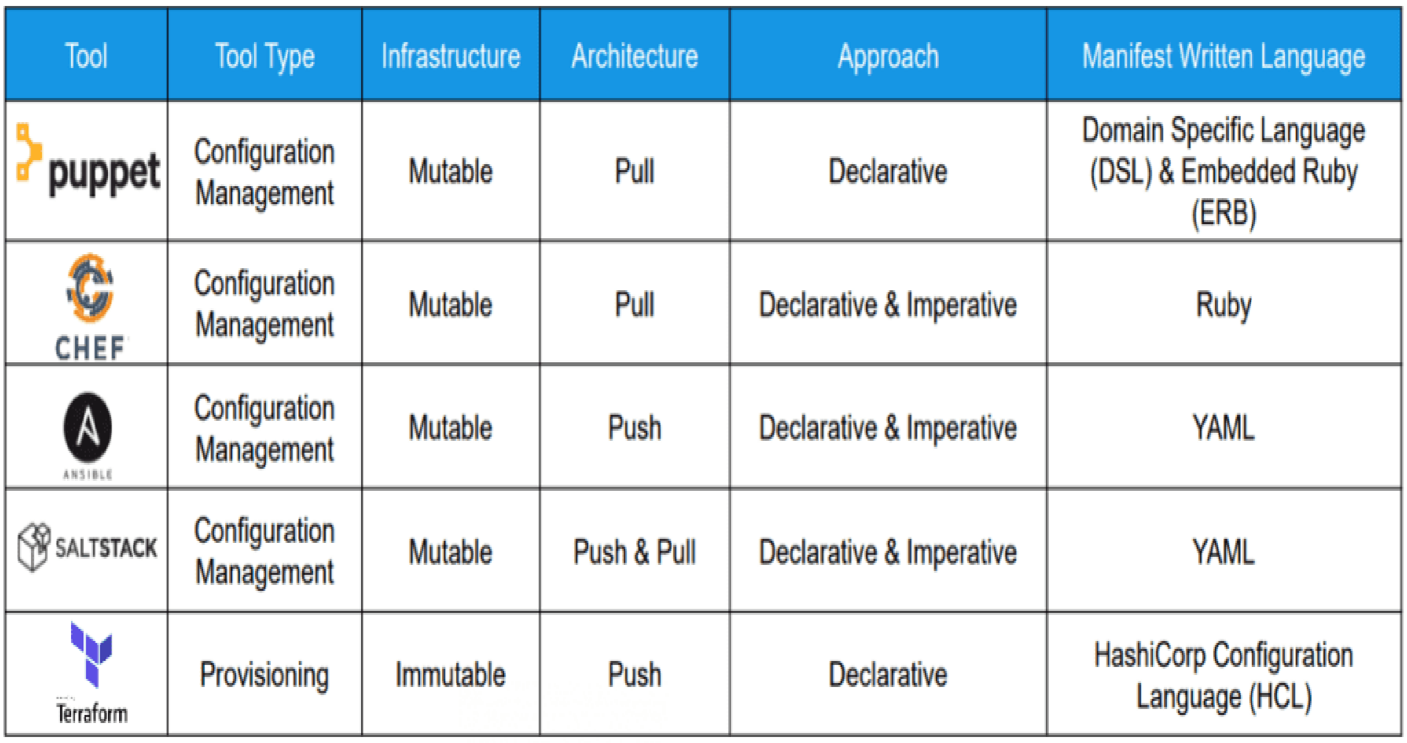
There are two effective configuration management tools: Chef and Puppet. Both seek to offer infrastructure management skills with configuration management and automation across the development process.

1. Chef was created with stronger collaboration features to be readily integrated into DevOps methods. It is used for configuration management and is Best used for Deploying and

configuring applications using a pull-based approach.

2. Puppet developed as a result of focusing on pure process automation. Currently, Puppet has automatic observers built in to detect configuration drift. It is a popular tool for configuration management and needs agents to be deployed on the target machines before puppet can start managing them.

Tools Overview Table:



Tool	Tool Type	Infrastructure	Architecture	Approach	Manifest Written Language
puppet	Configuration Management	Mutable	Pull	Declarative	Domain Specific Language (DSL) & Embedded Ruby (ERB)
CHEF	Configuration Management	Mutable	Pull	Declarative & Imperative	Ruby
ANSIBLE	Configuration Management	Mutable	Push	Declarative & Imperative	YAML
SALTSTACK	Configuration Management	Mutable	Push & Pull	Declarative & Imperative	YAML
Terraform	Provisioning	Immutable	Push	Declarative	HashiCorp Configuration Language (HCL)

IaC DevOps Best Practices

Avoid automating everything right away

- Try not to automate everything right away if you are a company, an application, or a platform that is still in the early stages of development. This is due to the potential for rapid change. You may start automating your platform's provisioning and maintenance after it has more or less reached a stable state.

Check and keep an eye on your setups

- Infrastructure as Code should and can still be tested since it is still code. Before deploying your servers, you should install testing and monitoring tools for IaC to look for flaws and inconsistencies.

The more rigid the better

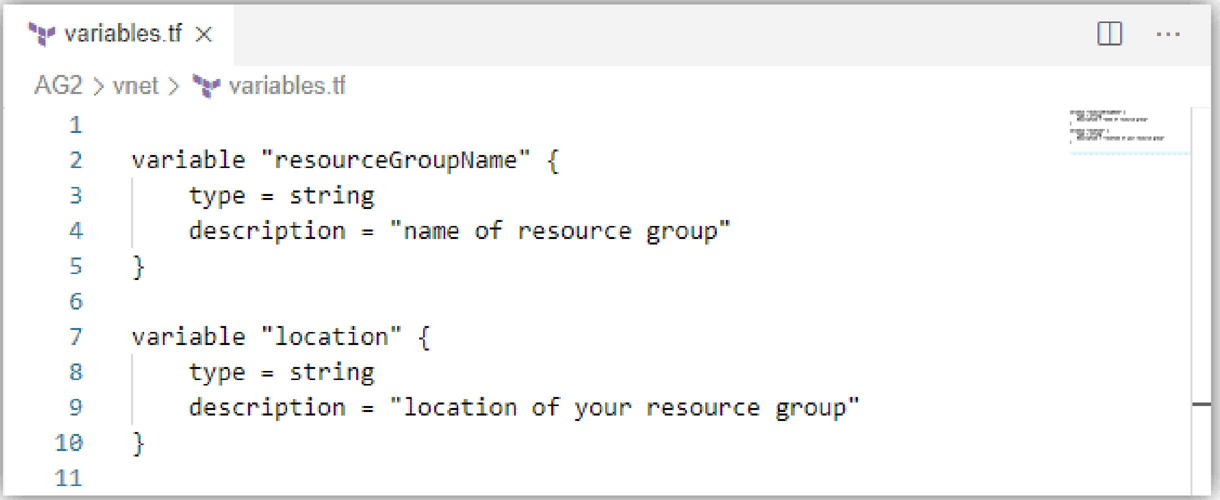
- Give as much detail as you can about the atmosphere you wish to create. Include the developers in the creation of the IaC standards for the runtime environments and infrastructure components. Making sure there are no flaws in the code can make it operate more efficiently.

Terraform Variables: Terraform Variable Types and Uses for Beginners

Terraform Input Variables

Terraform input variables are used as parameters to input values at run time to customize our deployments. Input terraform variables can be defined in the main.tf configuration file but it is a best practice to define them in a separate variable**.tf** file to provide better readability and organization.

A variable is defined by using a **variable** block with a label. The label is the name of the variable and must be unique among all the variables in the same configuration.



```

variable "resourceGroupName" {
    type = string
    description = "name of resource group"
}

variable "location" {
    type = string
    description = "location of your resource group"
}

```

The variable declaration can optionally include three arguments:

- **description:** briefly explain the purpose of the variable and what kind of value is expected.
- **type:** specifies the type of value such as string, number, bool, map, list, etc.
- **default:** If present, the variable is considered to be optional and if no value is set, the default value is used.

Check out: How to Install Terraform in Linux, Mac, Windows

Terraform Input Variables

The type argument in a variable block allows you to enforce *type constraints* on the variables a user passes in. Terraform supports a number of types, including **string**, **number**, **bool**, **list**, **map**, **set**, **object**, **tuple,** and **any**.

If a type isn't specified, then Terraform assumes the type is any. Now, let's have a look at some of these terraform variables types and their classification.

Primitive Types

A *primitive* type is a simple type that isn't made from any other type. The available primitive types are:

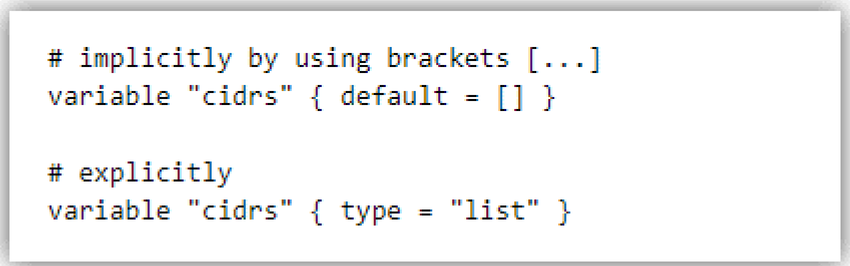
- **string:** a sequence of characters representing some text, such as "hello".
- **number:** a numeric value. The number type can represent both whole numbers like 15 and fractional values such as 6.28318.
- **bool:** either true or false.

Also Read: Terraform Workflow.

Complex Types

A *complex* type is a type that groups multiple values into a single value. These values could be of a similar type or different types.

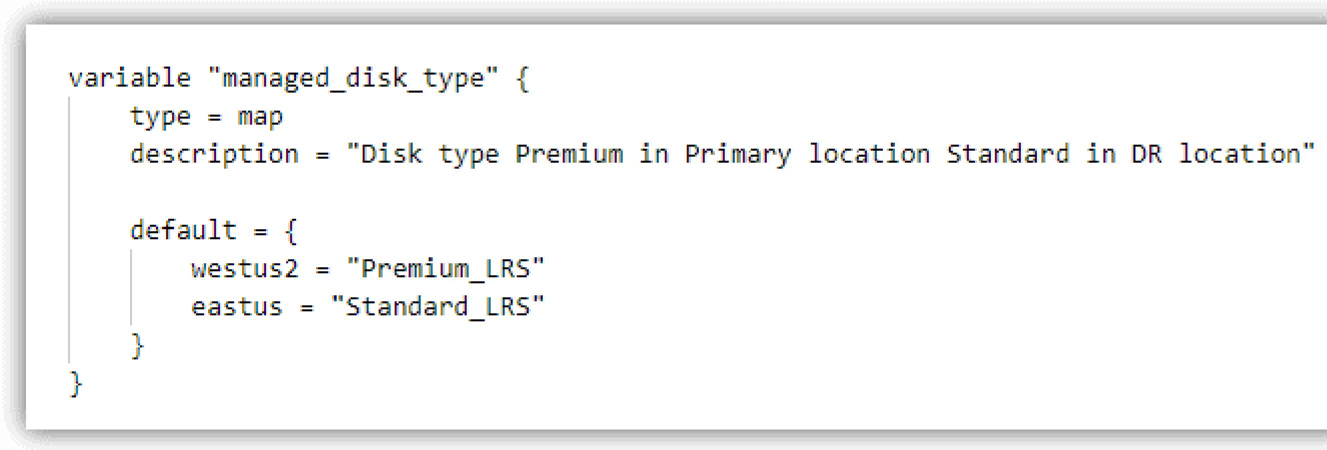
1. **List:** A Terraform list variable is a sequence of similar values indexed by numbers (starting with 0). It accepts any type of value as long as they are all of the same types. Lists can be defined either implicitly or explicitly.



```
# implicitly by using brackets [...]
variable "cidrs" { default = [] }

# explicitly
variable "cidrs" { type = "list" }
```

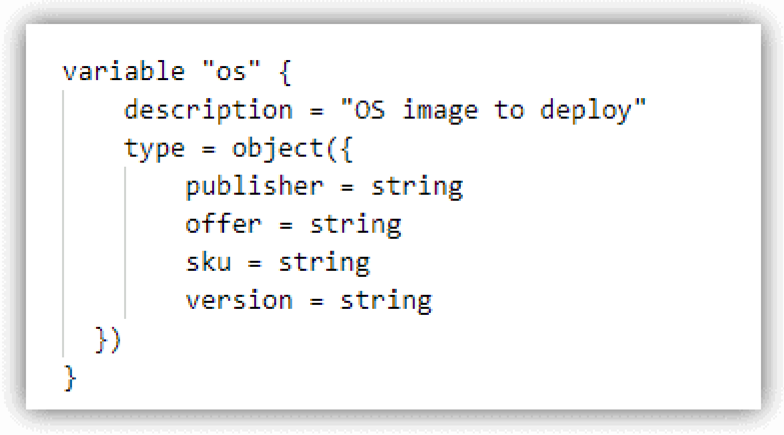
2. **Map:** A map is a collection of values where each value is identified by a string label.
In the example below is a map variable named **managed_disk_type** to define the type of storage we want to use based on the region in Azure. In the default block, there are two string values, "Premium_LRS" and "Standard_LRS" with a named label to identify each one

westus2 and eastus.



```
variable "managed_disk_type" {
    type = map
    description = "Disk type Premium in Primary location Standard in DR location"

    default = {
        westus2 = "Premium_LRS"
        eastus = "Standard_LRS"
    }
}
```

3. **Object:** An object is a structural type that can contain different types of values, unlike map, list. It is a collection of named attributes that each have their own type.
   In the below example, we have declared an object type variable **os** for the os image that can be used to deploy a VM.



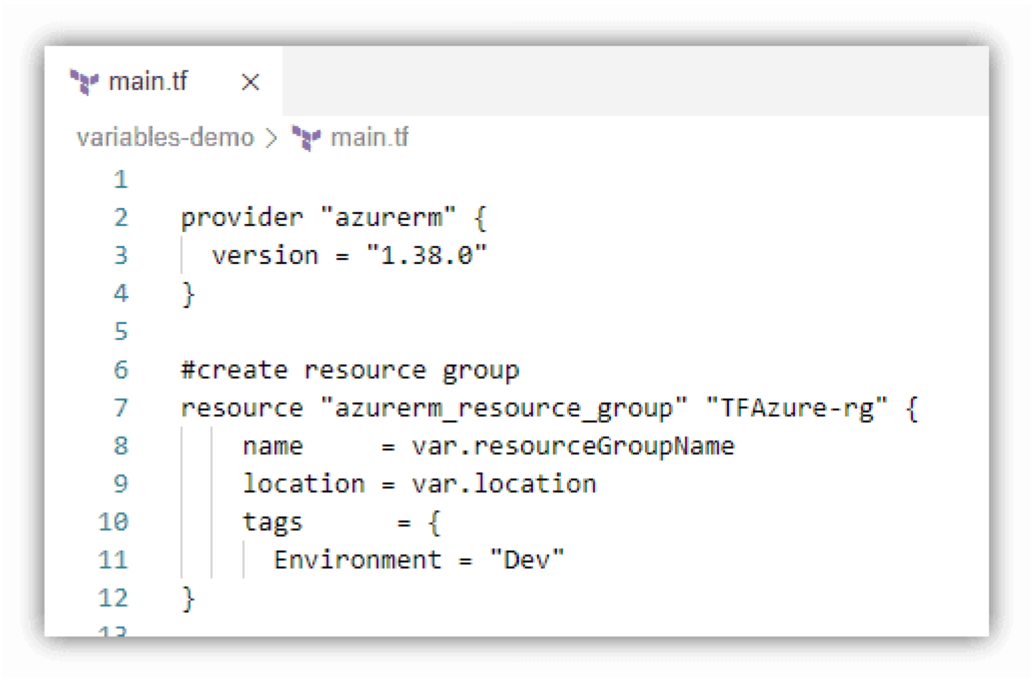
```
variable "os" {
    description = "OS image to deploy"
    type = object({
        publisher = string
        offer = string
        sku = string
        version = string
  })
}
```

Lists, maps, and objects are the three most common complex variable types. They all can be used for their specific use cases.

Also read: Step-by-step guide on **Terraform Certification**

Use Input Variables

After declaration, we can use the variables in our main.tf file by calling that variable in the **var.<name>** format. The value to the variables is assigned during **terraform apply**.


main.tf
variables-demo > main.tf

```
provider "azurerm" {
  version = "1.38.0"
}

#create resource group
resource "azurerm_resource_group" "TFAzure-rg" {
    name     = var.resourceGroupName
    location = var.location
    tags      = {
      Environment = "Dev"
}
```

Also Check Our blog post on **[Terraform Cheat Sheet](#)**. Click here

Assign Values To Input Variables

There are multiple ways to assign values to variables. The following is the descending order of precedence in which variables are considered.

1. Command-line flags

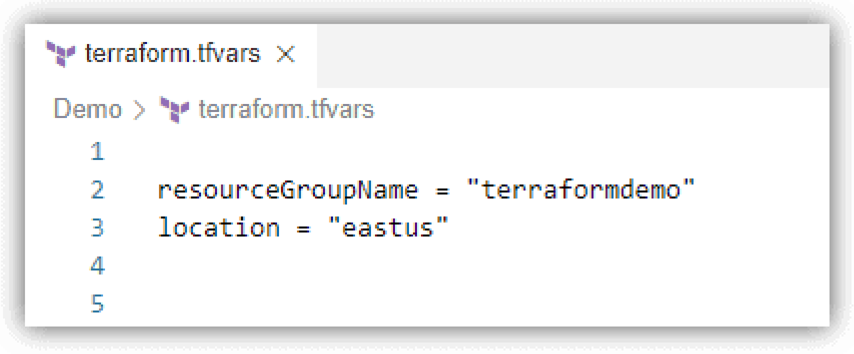
The most simple way to assign value to a variable is using the -var option in the command line when running the terraform plan and terraform apply commands.

```
$ terraform apply -var="resourceGroupName=terraformdemo-rg" -
var="location=eastus"
```

2. Variable Definition (.tfvars) Files

If there are many variable values to input, we can define them in a variable definition file. Terraform also automatically loads a number of variable definitions files if they are present:

- Files named exactly **terraform.tfvars** or **terraform.tfvars.json**
- Any files with names ending in **.auto.tfvars** or **.auto.tfvars.json**


terraform.tfvars
Demo > terraform.tfvars

```

resourceGroupName = "terraformdemo"
location = "eastus"


```

If the file is named something else, then use the **-var-file** flag directly to specify a file.

```
$ terraform apply -var-file="testing.tfvars"
```

3. Terraform Environment Variables

Terraform searches the environment of its own process for environment variables named **TF_VAR_<var-name>** followed by the name of a declared variable. Terraform scans all variables starting with TF_VAR and uses those as variable values for Terraform.

```
$ export TF_VAR_location=eastus
```

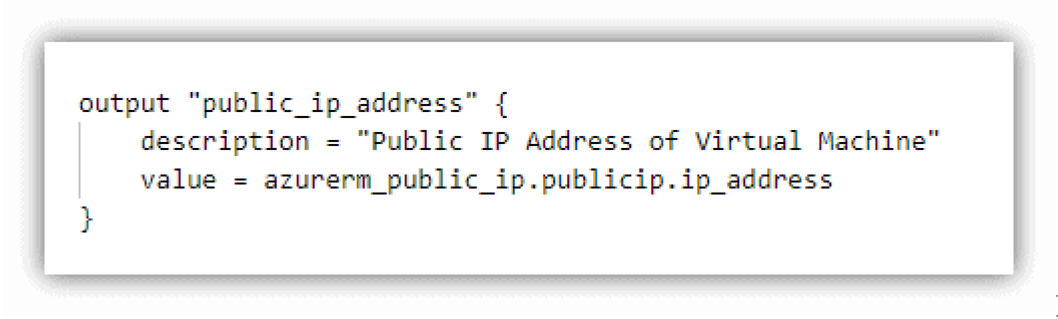
This can be useful when running Terraform in automation, or when running a sequence of Terraform commands in succession with the same terraform variables.

Read More: About Hashicorp Terraform. Click here

Define Output Variables

Outputs allow us to define values in the configuration that we want to share with other resources or modules. For example, we could pass on the output information for the public IP address of a server to another process.

An output variable is defined by using an **output** block with a label. The label must be unique as it can be used to reference the output's value. Let's define an output to show us the public IP address of the server. Add this to any of the *.tf files.



```
output "public_ip_address" {
    description = "Public IP Address of Virtual Machine"
    value = azurerm_public_ip.publicip.ip_address
}
```

Multiple output blocks can be defined to specify multiple output variables. Outputs are only shown when Terraform applies your plan, running a terraform plan will not render any outputs.

How to Provision AWS Infrastructure with Terraform?

Cloud itself is a big domain with various services running simultaneously. It will require a lot of effort for a person/organization to manage the cloud without automation. Thus, cloud automation is on its pace and various tools are proposed for faster and efficient development. One such automation tool is Terraform.

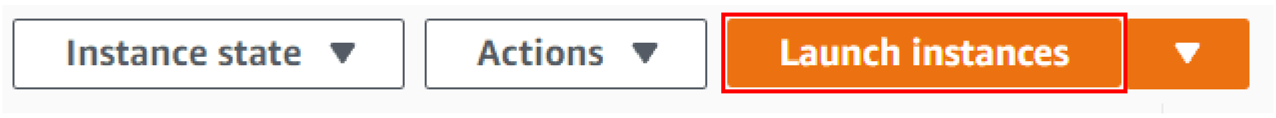
Prepare Your System

In this tutorial, we will use the Amazon EC2 instance with the Ubuntu system to run Terraform. But, it's your choice to run Terraform on any cloud platform or machine. For a better understanding, I will recommend you to follow the steps with us.

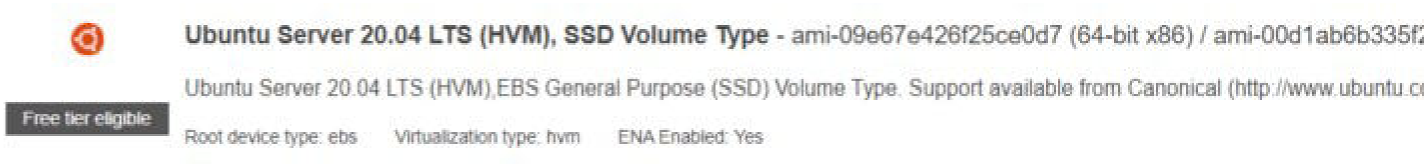
Step 1) Open up your AWS console or directly visit '*console.aws.amazon.com*'. If you don't have access to AWS, create one free tier account.

Step 2) Search for EC2 in your AWS console and open it.

Step 3) Click on Launch Instances to create a new EC2 instance.



Step 4) Select an AMI (Amazon Machine Image). In our case, we will use Ubuntu.



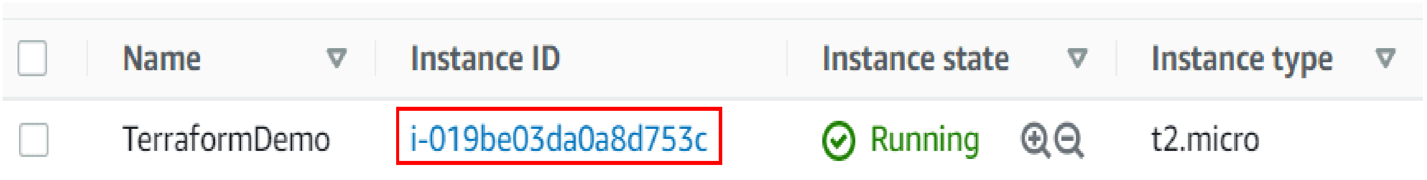
Free tier eligible

Ubuntu Server 20.04 LTS (HVM), SSD Volume Type - ami-09e67e426f25ce0d7 (64-bit x86) / ami-00d1ab6b335f

Ubuntu Server 20.04 LTS (HVM),EBS General Purpose (SSD) Volume Type. Support available from Canonical (http://www.ubuntu.c

Root device type: ebs Virtualization type: hvm ENA Enabled: Yes

Step 5) You can fill up all the details for your instance. To quickly create an EC2 instance, leave the settings to default, launch your instance and save your new RSA key pair safe.



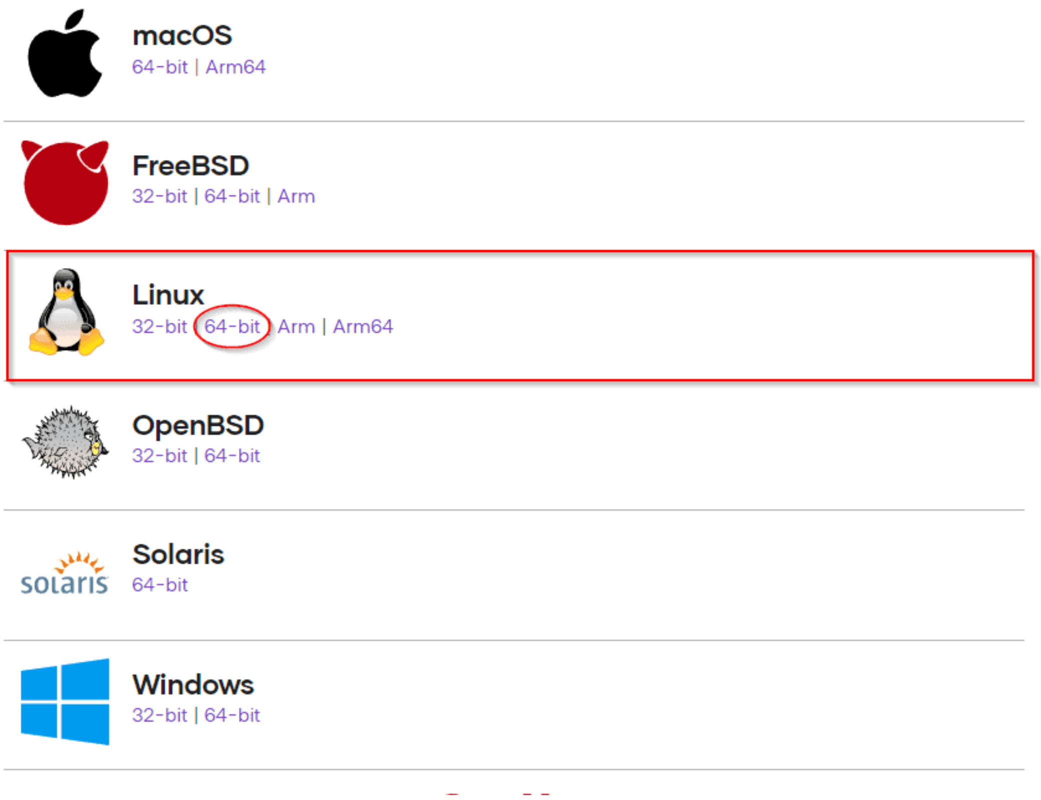
☐	Name	Instance ID	Instance state	Instance type
☐	TerraformDemo	i-019be03da0a8d753c	✓ Running	t2.micro

Step 6) After a few minutes, your instance will be live and running, as shown in the image above. Select it and click on connect to launch your EC2 instance.

Download, Install and Start Terraform

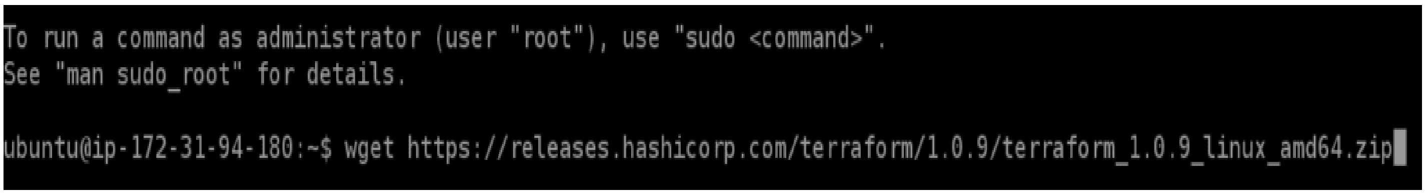
Based on your system requirement, you can download and install Terraform from the official download page.

Step 1) From the official Terraform download page, copy the link to the Linux file as shown in the image below.



Step 2) To download Terraform to your AWS instance, use ‘*wget’* command followed by the copied download URL as shown in the image below.

```
wget
https://releases.hashicorp.com/terraform/1.0.9/terraform_1.0.9_linux_amd64.zip
```



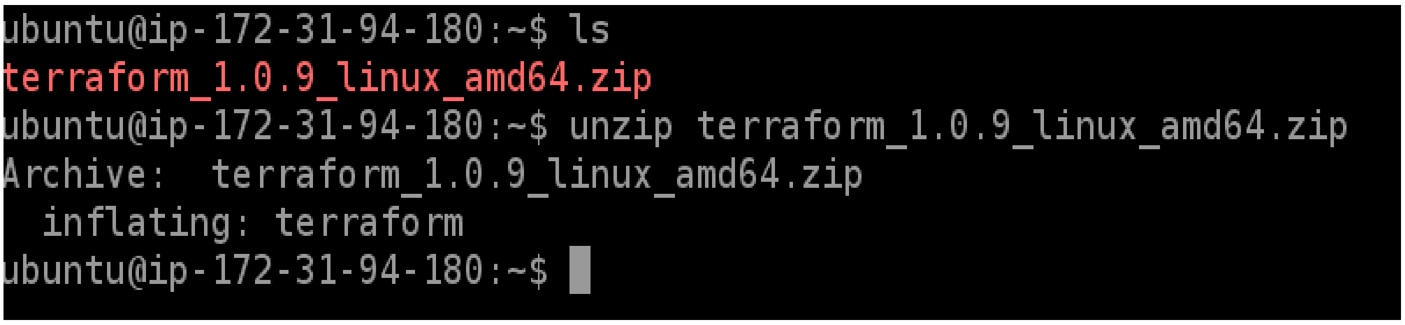
```
To run a command as administrator (user "root"), use "sudo <command>".
See "man sudo_root" for details.

ubuntu@ip-172-31-94-180:~$ wget https://releases.hashicorp.com/terraform/1.0.9/terraform_1.0.9_linux_amd64.zip
```

Step 3) To unzip the downloaded package, you need to install unzip tool in your system. Use the below command to download the tool.

```
sudo apt-get install unzip
```

Step 4) Check the downloaded package name using the ‘*ls*‘ command. Copy the package name and use the unzip command with the package name, as shown below.

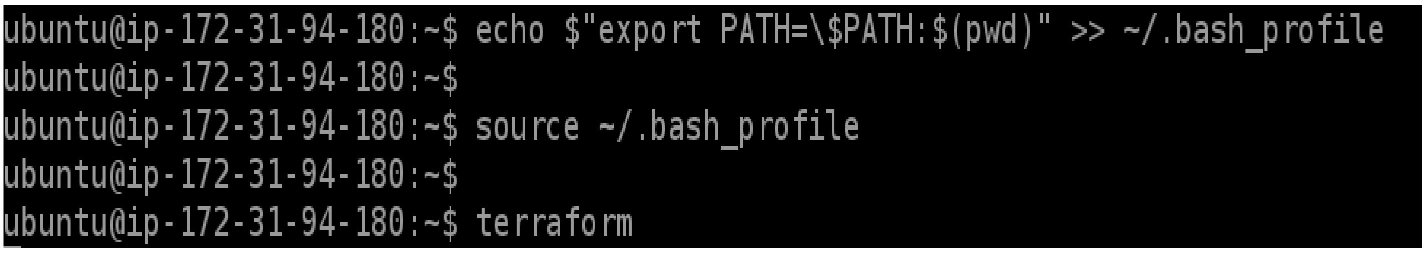
```
ls
unzip terraform_1.0.9_linux_amd64.zip
```



```
ubuntu@ip-172-31-94-180:~$ ls
terraform_1.0.9_linux_amd64.zip
ubuntu@ip-172-31-94-180:~$ unzip terraform_1.0.9_linux_amd64.zip
Archive:  terraform_1.0.9_linux_amd64.zip
  inflating: terraform
ubuntu@ip-172-31-94-180:~$
```

Step 5) Now, the Terraform is extracted successfully. Run the below commands one by one to run the terraform commands hassle-free by ignoring the directories. Finally, use the command ‘*terraform*‘ to activate terraform, as shown in the image below.

```
echo $"export PATH=\$PATH:$(pwd)" >> ~/.bash_profile
source ~/.bash_profile

terraform
```

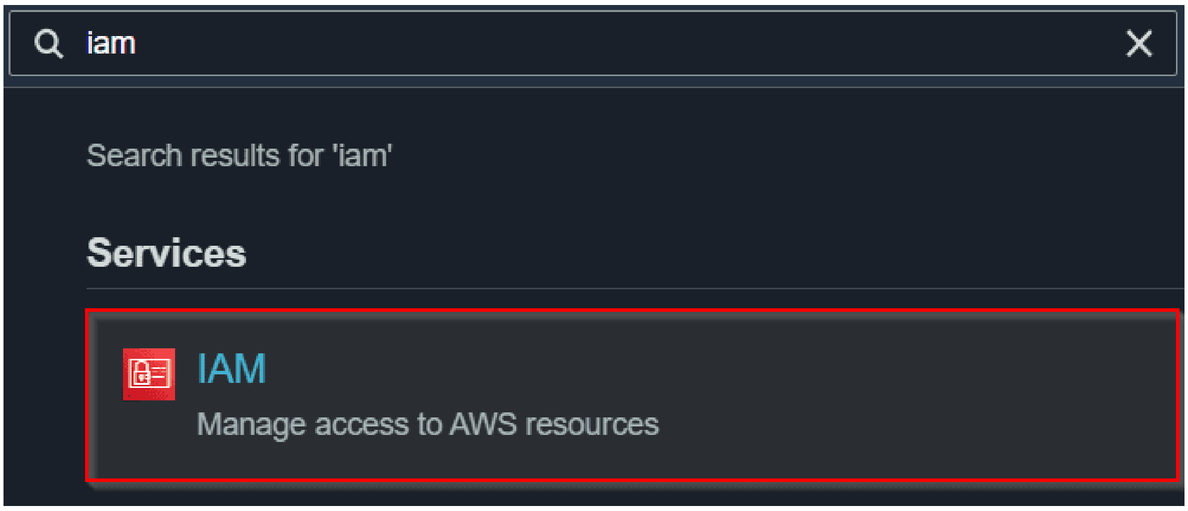


```
ubuntu@ip-172-31-94-180:~$ echo $"export PATH=\$PATH:$(pwd)" >> ~/.bash_profile
ubuntu@ip-172-31-94-180:~$
ubuntu@ip-172-31-94-180:~$ source ~/.bash_profile
ubuntu@ip-172-31-94-180:~$
ubuntu@ip-172-31-94-180:~$ terraform
```

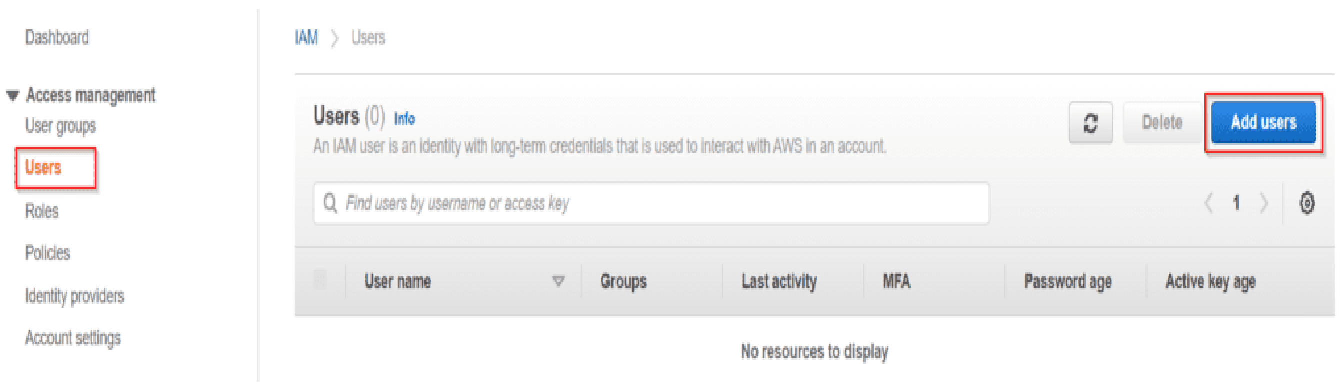
Generate AWS IAM Access Key

We installed terraform in our system, but now the question arises: How Terraform will provision anything without AWS permission. For this, we create an access key that will be used for resource provisioning by Terraform.

Step 1) Search and visit IAM in your AWS console.



Step 2) Select *Users* and *Add Users* here if you don’t have one, as shown in the image below.



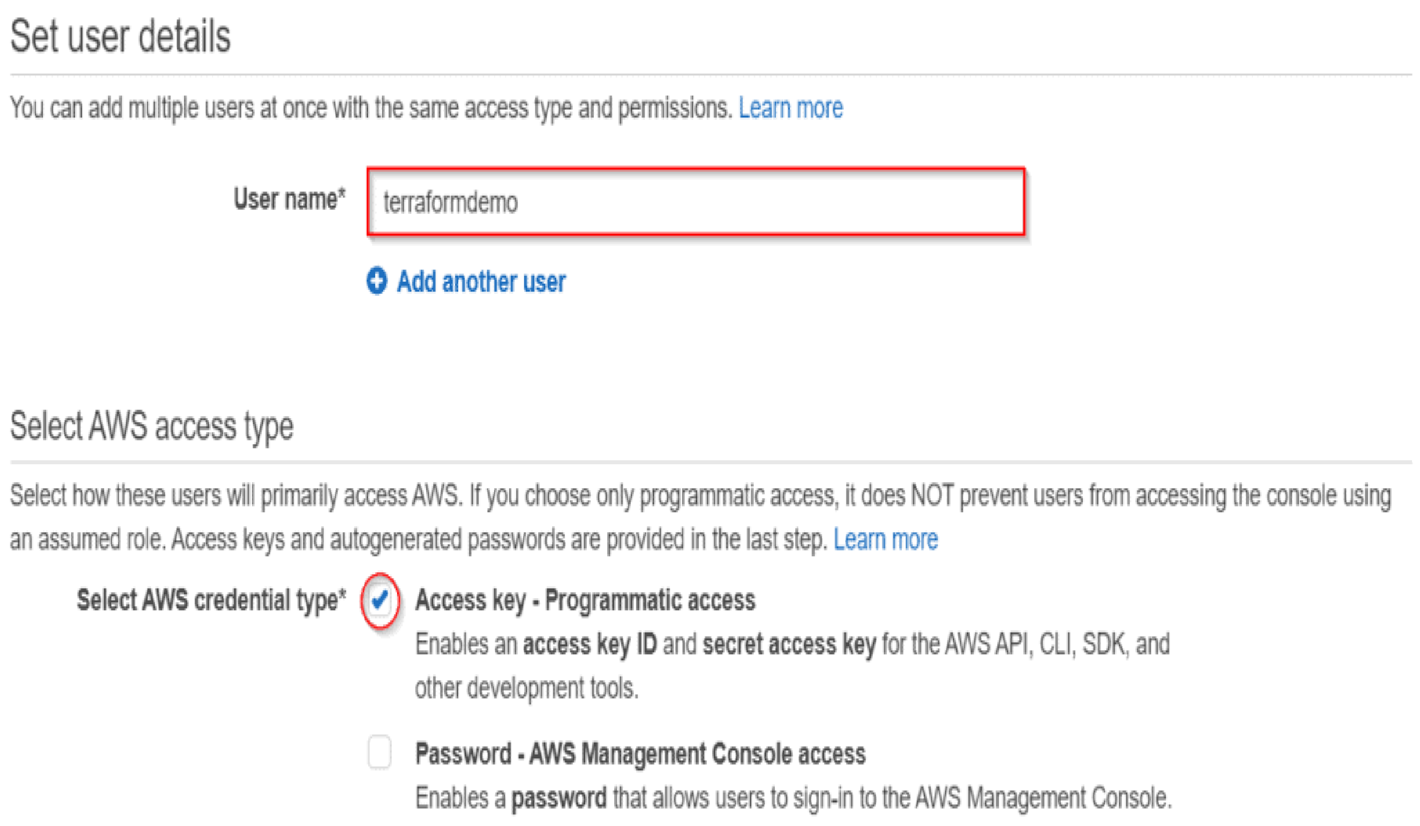
Users (0) Info

An IAM user is an identity with long-term credentials that is used to interact with AWS in an account.

	User name	Groups	Last activity	MFA	Password age	Active key age

No resources to display

Step 3) Name the user and select Access Key in the checkbox below. Now go to the Next window that is permission settings.



Set user details

You can add multiple users at once with the same access type and permissions. Learn more

User name* terraformdemo

Select AWS access type

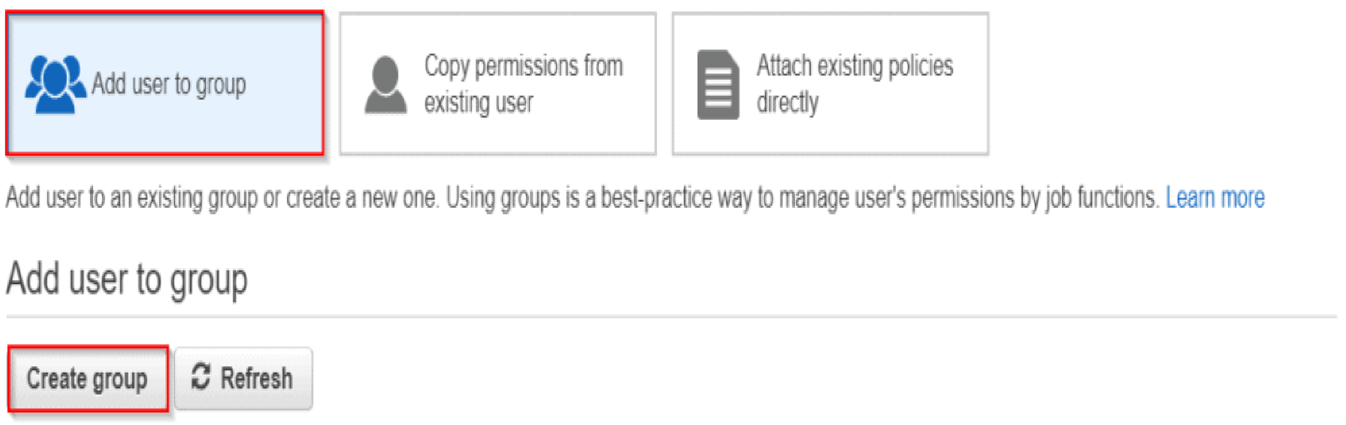
Select how these users will primarily access AWS. If you choose only programmatic access, it does NOT prevent users from accessing the console using an assumed role. Access keys and autogenerated passwords are provided in the last step. Learn more

Select AWS credential type*

[✓] **Access key - Programmatic access**
Enables an **access key ID** and **secret access key** for the AWS API, CLI, SDK, and other development tools.

[ ] **Password - AWS Management Console access**
Enables a **password** that allows users to sign-in to the AWS Management Console.

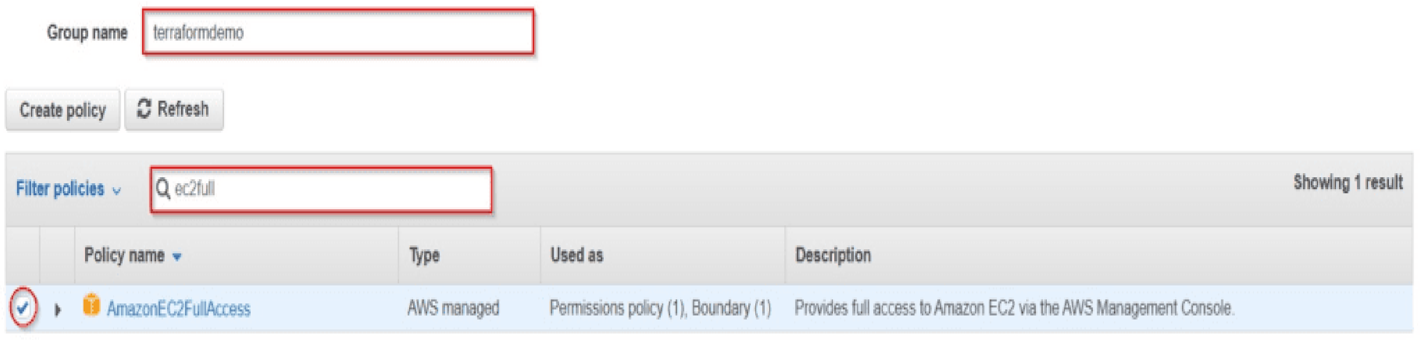
Step 4) Here, create one new user group.


Add user to group
Copy permissions from existing user
Attach existing policies directly

Add user to an existing group or create a new one. Using groups is a best-practice way to manage user's permissions by job functions. Learn more

Add user to group

Step 5) Name the group and search for required permission for allowing Terraform to perform its task. In our case, we will be creating one EC2 instance, so we selected *AmazonEC2FullAccess* permission, as shown in the image below. Now to generate this user group, click on the button Create.



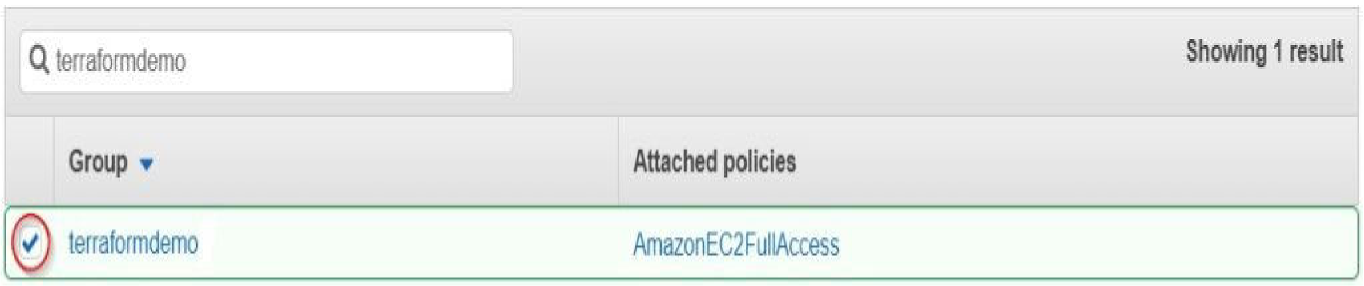
Group name: terraformdemo

ec2full

Showing 1 result

		Policy name	Type	Used as	Description
✓	▸	AmazonEC2FullAccess	AWS managed	Permissions policy (1), Boundary (1)	Provides full access to Amazon EC2 via the AWS Management Console.

Step 6) Select the created group for the IAM user and proceed with the final steps to make the new user active.

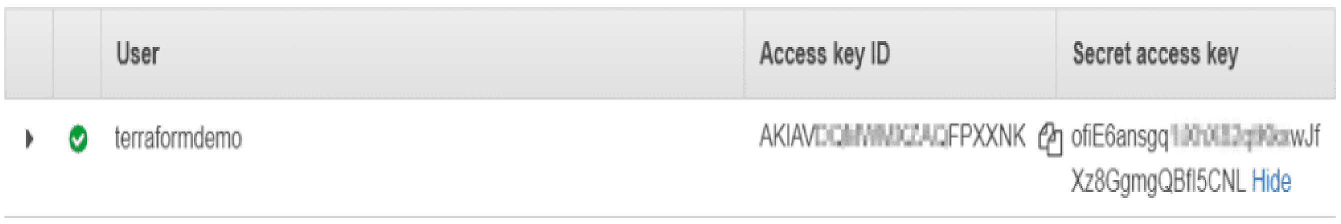


terraformdemo

Showing 1 result

	Group	Attached policies
✓	terraformdemo	AmazonEC2FullAccess

Step 7) After creating the user, you will see the keys on your screen or visit the user to find them. Copy your Access Key ID and Secret Access Key as they will be used while provisioning AWS resources using Terraform.



		User	Access key ID	Secret access key
▸	✓	terraformdemo	AKIAV[redacted]FPXXNK	ofiE6ansgq[redacted]wJf Xz8GgmgQBfI5CNL Hide

Create EC2 Instance with Terraform

For an easy understanding, we will create one EC2 instance using the terraform file with all the instructions to create the EC2 instance.

Step 1) Create a new directory using the ‘mkdir’ command and name it whatever you want. Then, visit the directory using the below commands.

```
mkdir terraform-lab
```

```
cd terraform-lab/
```

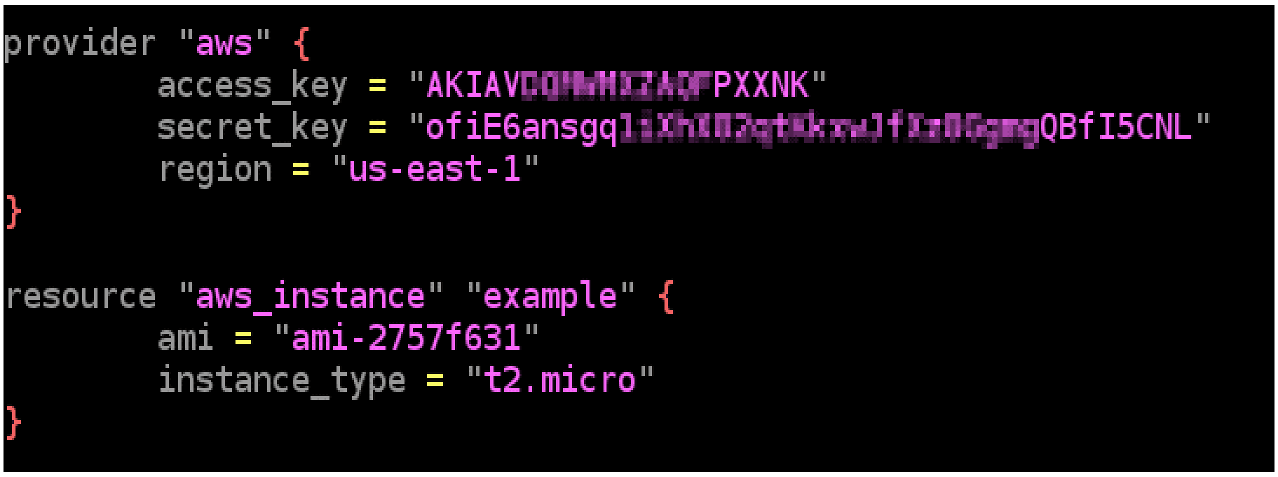
Step 2) Create a new file here that will have the instruction to provision the AWS resource. Use the below commands to create the file.

```
vim ec2.tf
```

Step 3) After running the above command, the file is created but to enable the edit mode, you need to press the INSERT or any other button on your keyboard. Copy the below code, replace it with your IAM keys and paste it on your file. Also, don't forget to maintain proper spacing and lines in your code, as shown in the image below.

> The provider section in code uses the credentials but if you have an AWS CLI setup you may not require this. The resource section in code will create the resource *aws-instance* with the name *example* with mentioned *AMI (Amazon Machine Image)*.

```
provider "aws" {
  access_key = "ACCESS_KEY_HERE"
  secret_key = "SECRET_KEY_HERE"
  region     = "us-east-1"
}

resource "aws_instance" "example" {
  ami           = "ami-2757f631"
  instance_type = "t2.micro"
}
```

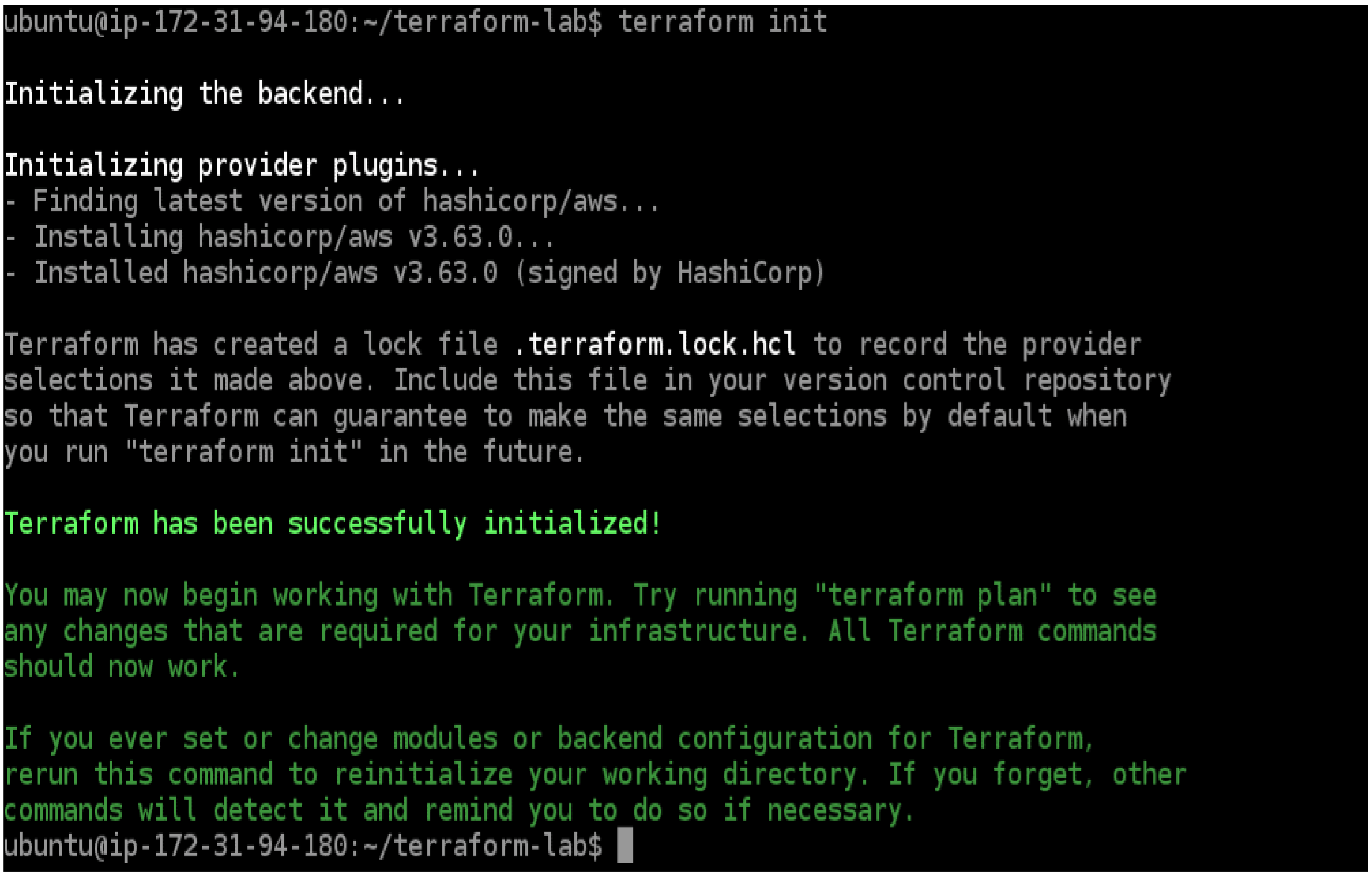


```
provider "aws" {
        access_key = "AKIAV[redacted]PXXNK"
        secret_key = "ofiE6ansgq[redacted]QBfI5CNL"
        region = "us-east-1"
}

resource "aws_instance" "example" {
        ami = "ami-2757f631"
        instance_type = "t2.micro"
}
```

Step 4) For saving the above file in the Ubuntu system, press the '*Esc*' button on your keyboard, type ':wq' and press '*Enter*'.

Step 5) Now everything is ready, use the below command to initialize terraform to compile the file. You will receive an error here if there is any syntax error in the file.

```
terraform init
```

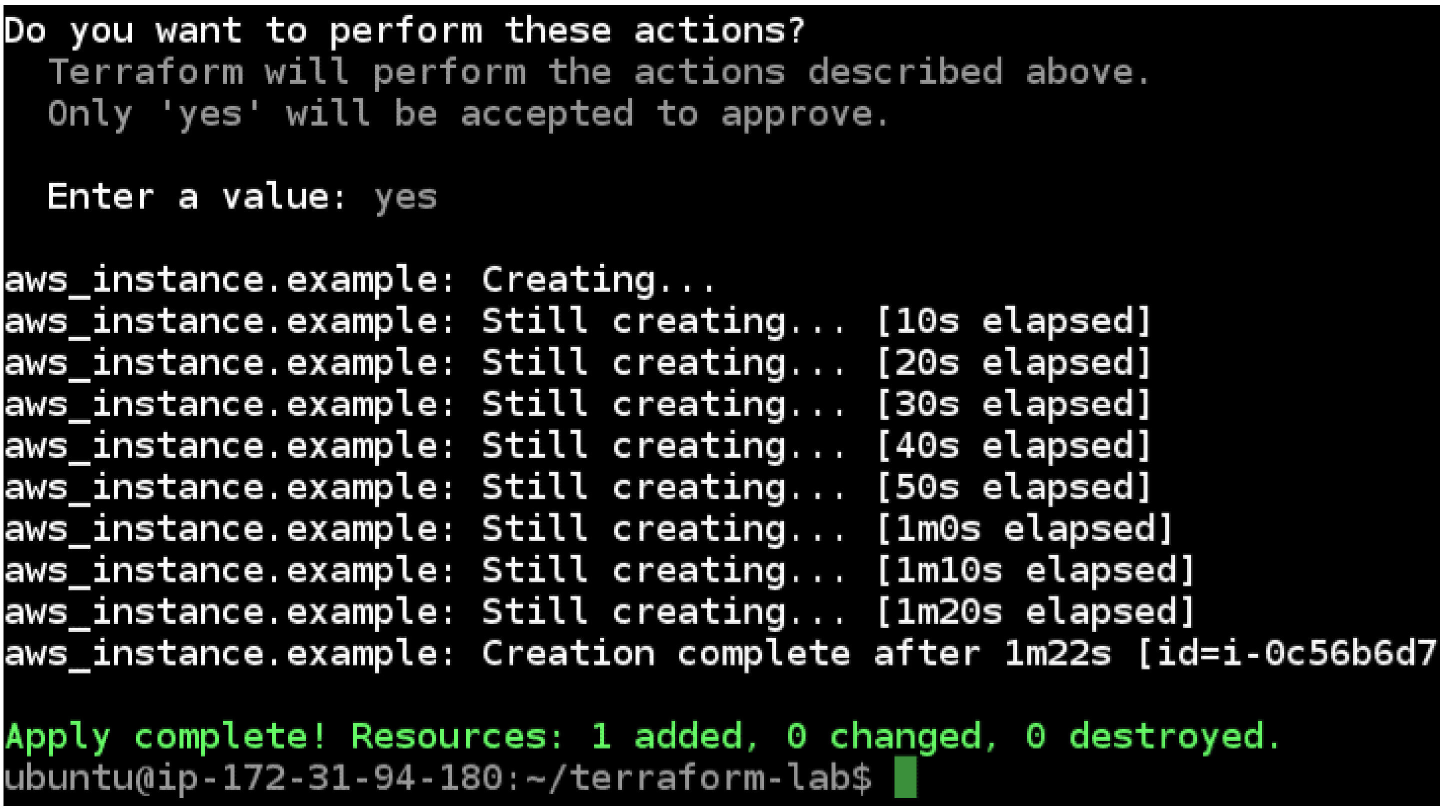


```
ubuntu@ip-172-31-94-180:~/terraform-lab$ terraform init

Initializing the backend...

Initializing provider plugins...
- Finding latest version of hashicorp/aws...
- Installing hashicorp/aws v3.63.0...
- Installed hashicorp/aws v3.63.0 (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.

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.
ubuntu@ip-172-31-94-180:~/terraform-lab$
```

Step 6) Finally, proceed with resource provisioning with the below command. You will receive an error here if the access keys are wrong, AMI (Amazon Machine Image) is wrong, or permission to provision EC2 Instance is not provided in the IAM (Identity Access Management).

If everything goes well, you will see a confirmation on your screen to provision. Type yes to proceed, and your resource will be live in a few minutes.

```
terraform 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.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: Still creating... [50s elapsed]
aws_instance.example: Still creating... [1m0s elapsed]
aws_instance.example: Still creating... [1m10s elapsed]
aws_instance.example: Still creating... [1m20s elapsed]
aws_instance.example: Creation complete after 1m22s [id=i-0c56b6d7

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
ubuntu@ip-172-31-94-180:~/terraform-lab$
```

Step 7) You can check your new AWS EC2 instance created in your AWS console. To delete all the resources created by Terraform, use the below command to clear all the things.

```
terraform destroy
```

Conclusion

Terraform is one of the most used Infrastructures as a Code tool by IT companies. In this blog, we created an AWS EC2 instance with Terraform using declarative syntax on our system. Using similar steps, we can create, manage and destroy any resource in AWS Infrastructure with the help of Terraform.

Terraform vs. Ansible: Key Differences and Comparison of Tools

Ansible vs Terraform battle continues to escalate as the DevOps environment focuses more on automation and orchestration. These two tools help in automating configurations and deploying infrastructure. Terraform offers to deploy Infrastructure as a Code, helps in readability and lift and shift deployments. Ansible is a configuration management tool for automating system configuration and management.

Terraform vs. Ansible

Terraform is a tool designed to help with the provisioning and deprovisioning of cloud infrastructure using an infrastructure as code approach. It is highly specialized for this purpose. On the other hand, Ansible is a more general tool that can be used for automation across various domains. Both Terraform and ansible have strong open-source communities and commercial products that are well supported.

To explain in short, **Terraform is a tool used for creating and managing IT infrastructure. Ansible automates provisioning, deployment, and other IT processes.**

Similarities between Terraform and Ansible

Given the features of both technologies, Terraform and Ansible appear to be extremely similar tools at a high level.

- They are both capable of setting up the new cloud infrastructure and equipping it with the necessary application components.
- On the freshly formed virtual machine, remote commands can be carried out by both Terraform and Ansible. This indicates that neither tool requires an agent. Agent deployment on the computers is not necessary for operational reasons.
- Terraform builds infrastructure utilizing the APIs of cloud providers, and SSH is used for simple configuration operations. The same is true of Ansible; all necessary configuration activities are carried out over SSH. Both tools are masterless since the "state" information for neither requires a separate piece of infrastructure to manage.

Difference between Terraform and Ansible Provisioning (Terraform vs. Ansible)

Let's see how Terraform vs. Ansible battle differentiates from each other:

Terraform	Ansible
Terraform is a provisioning tool.	Ansible is a configuration management too
It follows a declarative Infrastructure as a Code approach.	It follows both declarative & procedural appro
It is the best fit for orchestrating cloud services and setup cloud infrastructure from scratch.	It is mainly used for configuring servers with th software and updating already configured reso
Terraform does not support bare metal provisioning by default.	Ansible supports the provisioning of bare metal
It does not provide better support in terms of packaging and templating.	It provides full support for packaging and temp
It highly depends on lifecycle or state management.	It does not have lifecycle management at a

1. Orchestration vs. Configuration Management

Terraform and Ansible have so many similarities and differences at the same time. The difference comes when we look at two significant concepts of DevOps: Orchestration and configuration management.

Configuration management tools solve the issues locally rather than replacing the system entirely. Ansible helps to configure each action and instrument and ensures smooth functioning without any damage or error. In addition, Ansible comes up with hybrid capabilities to perform both orchestration and replace infrastructure.

Orchestration tools ensure that an environment is in its desired state continuously. Terraform is explicitly designed to store the state of the domain. Whenever there is any glitch in the system, terraform automatically restores and computes the entire process in the system after reloading. It is the best fit in situations where a constant and invariable state is needed. **Terraform Apply** helps to resolve all anomalies effectively.

Let’s have a look at the Procedural and Declarative nature of Terraform and Ansible.

2. Declarative vs. Procedural

There are two main categories of DevOps tools: Procedural vs. Declarative. These two categories tell the action of tools.

Terraform follows the **declarative approach,** ensuring that if your defined environment suffers changes, it rectifies those changes. This tool attempts to reach the desired end state described by the *sysadmin.* Puppet also follows the declarative approach. With terraform, we can automatically describe the desired state and figure out how to move from one state to the next.

Ansible is of hybrid nature. It follows both declarative and **procedural style** configuration. It performs ad-hoc commands to implement procedural-style configurations. Please read the documentation of Ansible very carefully to get in-depth knowledge of its behavior. It’s important to know whether you need to add or subtract resources to get the desired result or need to indicate the resources required explicitly.

3. Mutable vs. Immutable

A workflow for application deployment involves providing the infrastructure, installing the correct version of the source code, and installing any dependencies.

The infrastructure that serves as the foundation for later versions of apps and services has a property known as mutability. Either existing infrastructure is used for deployment, or we can create an entirely new set of infrastructure for it.

It depends on the deployment procedures whether the infrastructure is mutable or immutable. It is said to as malleable when subsequent versions of apps are released on the same infrastructure. However, it is said to as immutable if the deployment takes place during releases on entirely new infrastructure.

Although mutability appears convenient, there is a higher chance of failure. The previous version must first be uninstalled before the desired version can be installed when application configurations are applied again on the same infrastructure. Additional steps increase the likelihood of failure. This can lead to inconsistent setups and unpredictable behavior across a fleet of servers.

Instead, if we concentrate on minimizing these steps by skipping the uninstalling process and carrying out the installation on fresh infrastructure resources, we will have the opportunity to test the new deployment and roll it back in case it doesn’t work. This approach to treating infrastructure as immutable gives administrators more control over making changes.

4. State Management

The full lifecycle of the resources under Terraform's administration is managed. It keeps up the state files' mapping of infrastructure resources to the most recent configuration. State management is crucial to Terraform's operation.

States are used to monitor configuration changes and provide the same. Additionally, it is possible to import preexisting resources managed by Terraform by bringing in state files from the infrastructure of the real world.

It is possible to query the Terraform state files at any moment to learn about the infrastructure components and their available characteristics.

Ansible, in contrast, does not provide any form of lifecycle management. Any changes made to the configuration are automatically implemented on the target resource because Ansible focuses on configuration management and assumes immutable infrastructure by default.

Terraform vs Ansible Provisioning

Terraform deals with **infrastructure automation**. Its current declarative model lacks some features which arise complexity. Using Terraform, the elements of required environments are separately described, including their relationships. It assesses the model, creates a plan based on dependencies, and gives optimized commands to Infrastructure as a Service. If there is no change in the environment or strategy, repeated runs will do nothing. If there is any **update** in the plan or environment, it will **synchronize** the cloud infrastructure.

Ansible follows a **procedural approach**. Various users create playbooks that are evaluated through top to bottom approach and executed in sequence. **Playbooks** are responsible for the configuration of network devices that contributes towards a procedural approach. Of course, Ansible provisions the cloud infrastructure as well. But its procedural approach limits it to large infrastructure deployments.

How does Terraform work?

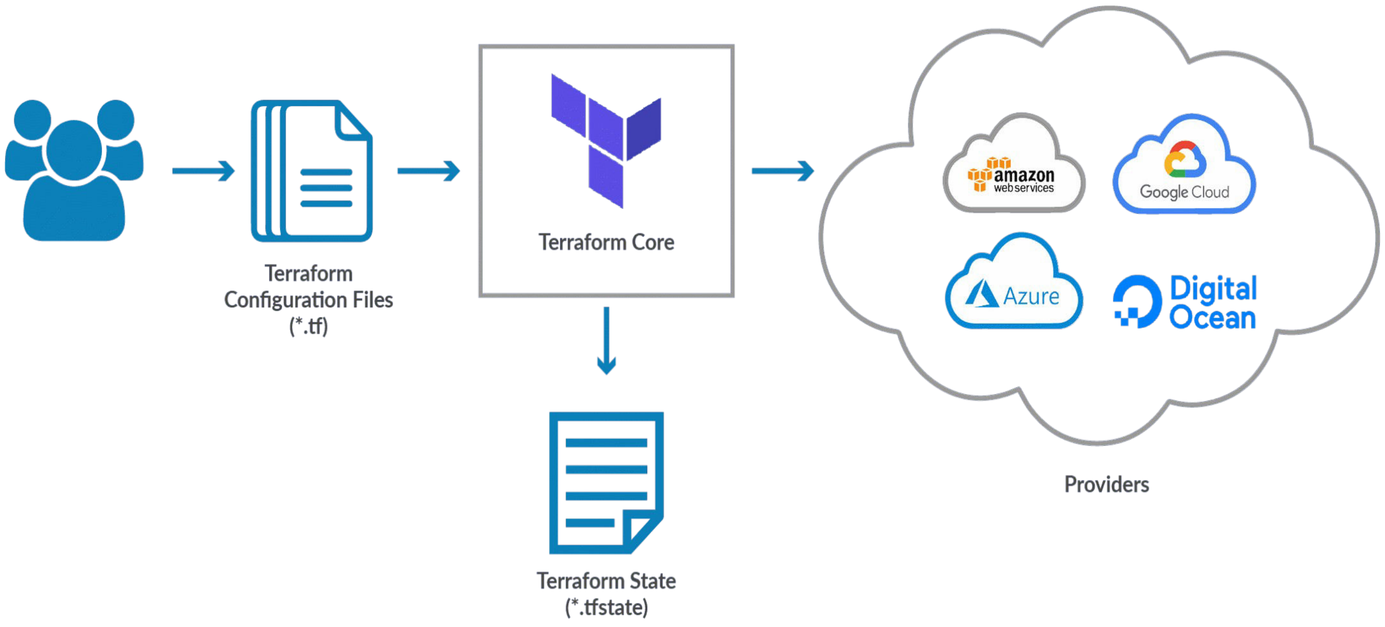
There are two main working components of terraform.

- **Terraform Core**
- **Providers**

Terraform is of **declarative nature**. It directly describes the end state of the system without defining the steps to reach there. It works at a high level of abstraction to describe what services and resources should be created and defined.

Terraform core takes two input sources to do its job. The first input source is a **terraform configuration** that is configured by its users. Users define what needs to be provisioned and created. The second input source is a state that holds information about the infrastructure.

So terraform core takes the input and figures out various plans for what steps to follow to get the desired output.



The second principal component is **providers**, such as cloud providers like AWS, GCP, Azure, or other Infrastructure as service platforms. It helps to create infrastructure on different levels. Let's take an example where users create an AWS infrastructure, deploy Kubernetes on top of it, and then create services inside the cluster of Kubernetes. Terraform has multiple providers for various technologies; users can access resources from these providers through terraform. This is the basic working terminology of terraform that helps to provision and cover the complete application set up from infrastructure to fully developed application.

Check out: Terraform cheat sheet.

Features of Terraform

As we have discussed the working of Terraform, now we will look at the features of Terraform.

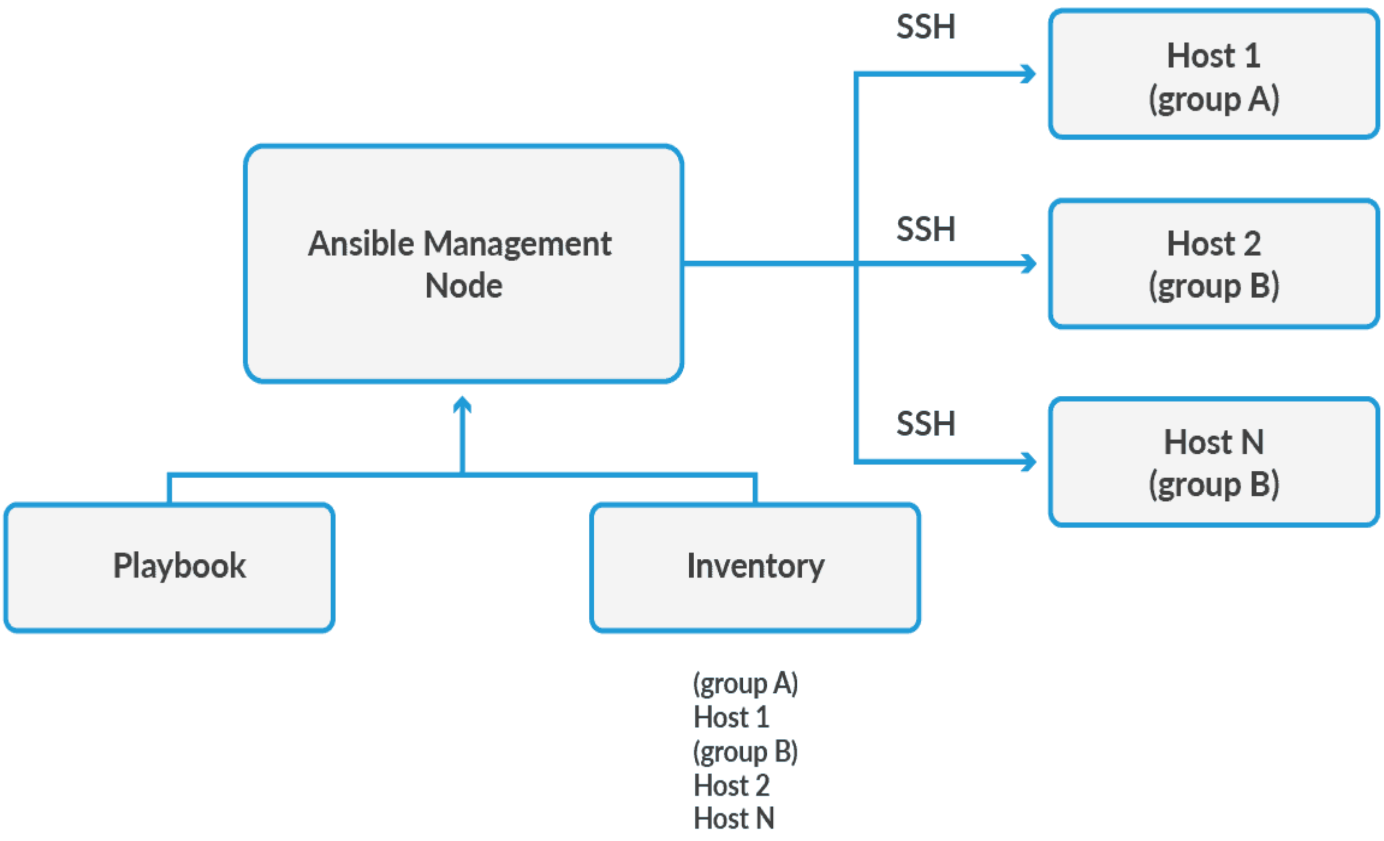
- Terraform follows a **declarative approach** which makes deployments fast and easy.
- It is a convenient tool to display the resulting model in a **graphical form**.
- Terraform also manages **external service providers** such as cloud networks and in-house solutions.
- It is one of the rare tools to offer **building infrastructure** from scratch, whether public, private or multi-cloud.
- It helps **manage parallel environments**, making it a good choice for testing, validating bug fixes, and formal acceptance.

- Modular code helps in achieving **consistency, reusability,** and **collaboration**.
- Terraform can **manage multiple clouds** to increase fault tolerance.

Also Check: Our blog post on **Terraform Workflow**. Click here

How does Ansible work?

Ansible is agentless and doesn't run on target nodes. It makes **connections using SSH** or other authentication methods. It installs various **Python modules** on the target using JSON. These modules are simple instructions that run on the target. These modules are executed and removed once their job is done. This strategy ensures that there is no misuse of resources on target. Python is mandatory to be installed on both the controlling and the target nodes.



Ansible **management node** acts as a controlling node that controls the entire execution of the playbook. This node is the place to run the installations. There is an **inventory file** that provides the host list where the modules need to be run. The management node makes SSH connections to execute the modules on the host machine and installs the product. Modules are removed once they are installed in the system. This is the simple working process of Ansible.

Let's have a look at the features of ansible.

Features of Ansible

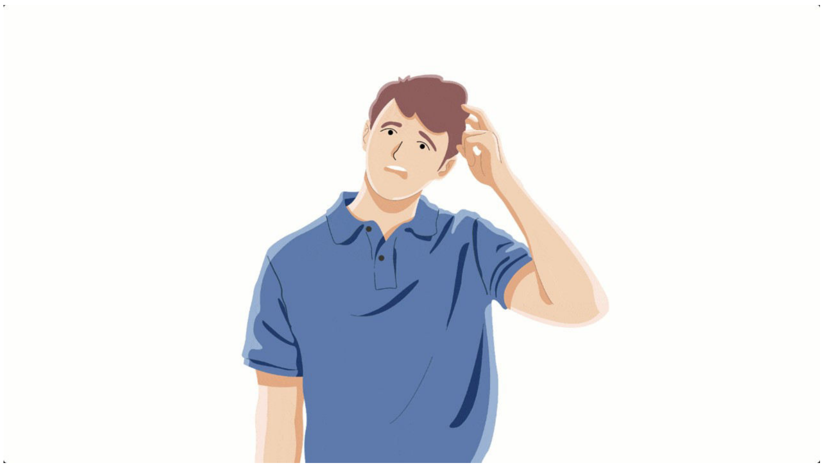
Now we will discuss various features Ansible provides to benefit its users.

- Ansible is used for **configuration management** and follows a procedural approach.
- Ansible deals with **infrastructure platforms** such as bare metal, cloud networks, and virtualized devices like hypervisors.

- Ansible follows **idempotent behavior** that makes it to place node in the same state every time.
- It uses **Infrastructure as a Code system configuration** across the infrastructure.
- It offers **rapid and easy deployment** of multi-tier apps with being agentless.
- If the code is **interrupted**, it allows entering the **code again** without any conflicts with other invocations.

Check out: ***Ansible Configuration Management Tools***

Which one to choose: Terraform or Ansible?



Terraform vs. Ansible: Every tool has its unique characteristics and limitations. Let's check out which one to go with.

Terraform comes with good **scheduling capabilities** and is **very user-friendly**. It integrates with docker well, as docker handles the configuration management slightly better than Terraform. But there is no clear evidence of how the target devices are brought to their final state, and sometimes, the final configuration is unnecessary.

Ansible comes with **better security** and **ACL functionality**. It is considered a mature tool because it adjusts comfortably with **traditional automation frameworks**. It offers simple operations and helps to code quickly. But, on the other hand, it is not good at services like logical dependencies, orchestration services, and interconnected applications.

You can now choose between these two, according to the requirement of the situation and the job. For example, if the containerized solution is used to provision software within the cloud, then Terraform is preferable. On the other hand, if you want to gain reasonable control of your devices and find other ways to deploy underlying services, Ansible is more suitable. These tools will provide more comprehensive solutions in the future.

Conclusion

It is essential to know which tool is used for which job among Terraform vs. Ansible. Terraform is mainly known for provisioning infrastructure across various clouds. It supports more than 200 providers and a great tool to manage cloud services below the server. In comparison, Ansible is optimized to perform both provisioning and configuration management. Therefore, we can say that both Terraform and Ansible can work hand in hand as **standalone tools** or **work together** but always pick up the right tool as per the job requirement.

Terraform Providers Overview

Terraform Providers: Terraform is one of the most popular tools used by DevOps teams to automate infrastructure tasks. It is used to provision and manages any cloud, infrastructure, or service.

Terraform officially supports around 130 providers. Its community-supported providers' page lists another 160. Some of those providers expose just a few resources, but others, such as AWS, OCI, or Azure, have hundreds of them.

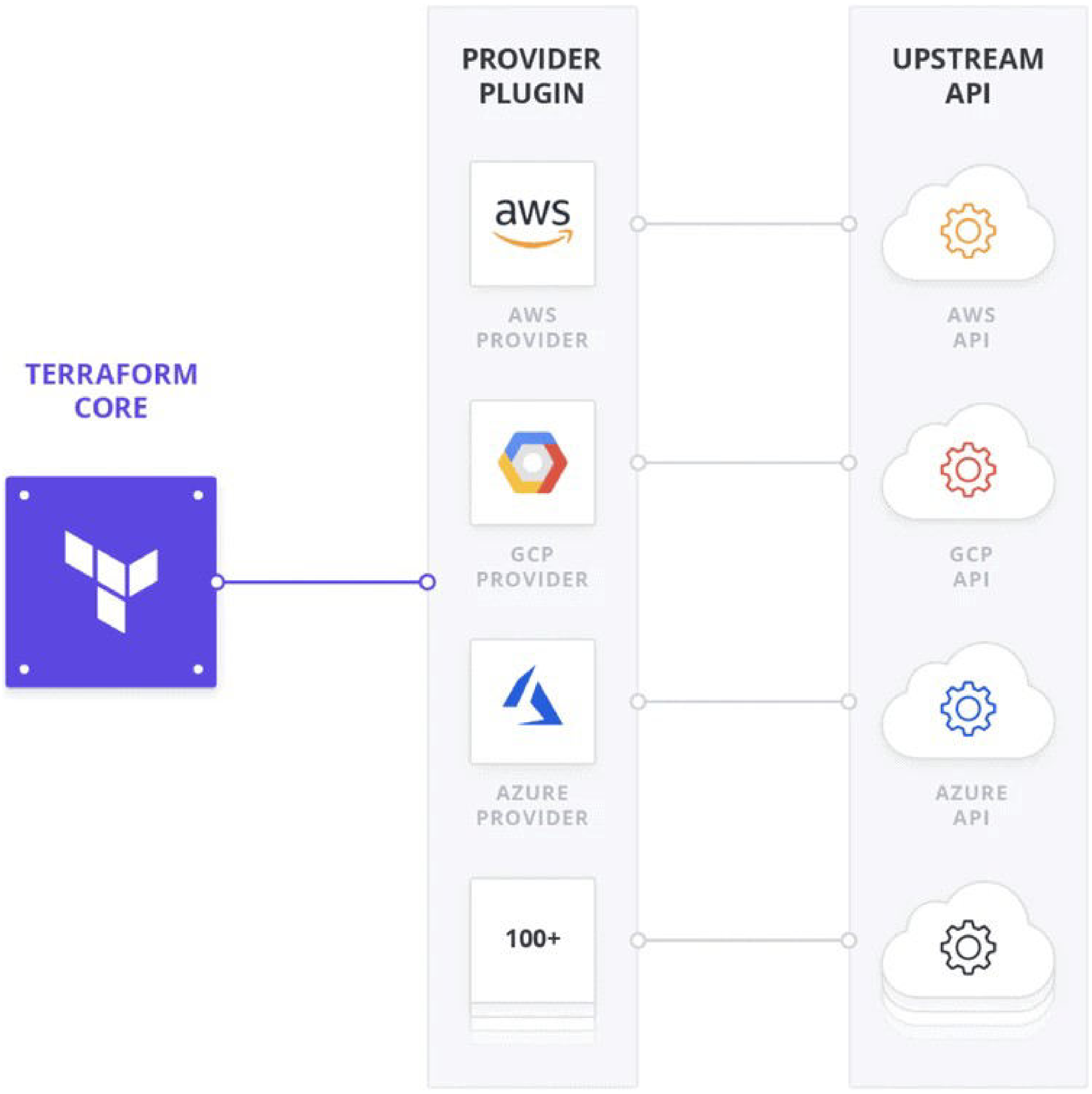
In this blog post, we cover a basic introduction of terraform providers and some major terraform cloud providers such as AWS, Azure, Google, and OCI.

A large percentage of Terraform users provision their infrastructure on the major cloud providers such as AWS, Azure, OCI, and others. To know more about various other terraform providers check **here.**

Let's understand the basic terminologies often used in Terraform:

- **Terraform** provisions, updates, and destroys infrastructure resources such as physical machines, VMs, network switches, containers, and more.
- **Configurations** are code written for Terraform, using the human-readable HashiCorp Configuration Language (HCL) to describe the desired state of infrastructure resources.
- **Providers** are the plugins that Terraform uses to manage those resources. Every supported service or infrastructure platform has a provider that defines which resources are available and performs API calls to manage those resources.
- **Modules** are reusable Terraform configurations that can be called and configured by other configurations. Most modules manage a few closely related resources from a single provider.
- **The Terraform Registry** makes it easy to use any provider or module. To use a provider or module from this registry, just add it to your configuration; when you run `terraform init`, Terraform will automatically download everything it needs.

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.



Also check: Types of [Terraform Variables]()

Terraform AWS Provider

Terraform can provision infrastructure across public cloud providers such as Amazon Web Services (AWS), Azure, Google Cloud, and DigitalOcean, as well as private cloud and virtualization platforms such as OpenStack and VMWare.

AWS is a good choice for learning Terraform because of the following:

- AWS is the most popular cloud infrastructure provider, by far. It has a 45% share in the cloud infrastructure market, which is more than the next [three biggest competitors]() (Microsoft, Google, and IBM) combined.
- AWS provides a huge range of reliable and scalable cloud hosting services, including Amazon Elastic Compute Cloud (Amazon EC2), which you can use to deploy virtual servers; Auto Scaling Groups (ASGs), which make it easier to manage a cluster of virtual servers; and Elastic Load Balancers (ELBs), which you can use to distribute traffic across the cluster of virtual servers.

- AWS offers a generous Free Tier for the first year that should allow you to run all of these examples for free. If you already used up your free tier credits, the examples in this book should still cost you no more than a few dollars.

Learn more about how to AWS Free Tier Account to avail the free tier services.

The two most popular options for deploying infrastructure to AWS are CloudFormation, a service native to AWS, and Terraform, an open-source offering from HashiCorp. Most of the AWS resources can be provisioned with Terraform as well and is often faster than CloudFormation when it comes to supporting new AWS features. On top of that, Terraform supports other cloud providers as well as 3rd party services.

Also read: Step by step guide on Terraform Certification

Azure Terraform Resource Provider

Azure Resource Providers for HashiCorp Terraform enables Azure customers using Azure Resource Manager (ARM) to provision and manage their resources with Terraform Providers as if they were native Azure Resource Providers.

Below are some of the core infrastructure services supported by Azure Resource Provider in Terraform:

- Virtual machines
- Storage accounts
- Networking interfaces
- Azure Container Instances
- Azure Container Service
- Managed Disks
- Virtual Machine Scale Sets

The ARM Resource Provider leverages HashiCorp Terraform to provide third-party services to ARM users directly via ARM. Some of these third-party services supported are listed below:

- Kubernetes
- Datadog
- Cloudflare

Terraform is built into Azure Cloud Shell and cloud shell automatically authenticates your default Azure CLI subscription to deploy resources through the Terraform Azure modules.

To know more about Azure provider for Terraform, click here.

Also Check: Our blog post on **Terraform Commands Cheat Sheet**. Click here

Oracle Cloud Infrastructure Terraform Provider

Oracle Cloud Infrastructure is an official provider of Hashicorp Terraform supporting infrastructure-as-code for oracle cloud customers.

The provider is compatible with Terraform 0.10.1 and later. Following are some of the main resources supported by the Terraform provider:

- Block Volumes
- Compute
- Container Engine
- Database
- File Storage
- Identity and Access Management (IAM)
- Load Balancing
- Networking
- Object Storage

A detailed list of supported resources and more information about how to get started is available on the **HashiCorp website.**

Oracle also provides **Resource Manager**, a fully managed service to operate Terraform. Resource Manager integrates with Oracle Cloud Infrastructure Identity and Access Management (IAM), so you can define granular permissions for Terraform operations. It also provides state locking, giving users the ability to share state, and lets teams collaborate effectively on their Terraform deployments. Most of all, it makes operating Terraform easier and more reliable.

To know more about Resource Manager, check **here.**

Oracle had announced two features to help you bring your existing infrastructure to Terraform and Resource Manager:

- **Terraform Resource Discovery** helps you move from a console- or SDK-managed infrastructure to an infrastructure managed by Terraform.
- **State File Import** helps customers who have a Terraform-managed infrastructure move to a Resource Manager–managed infrastructure.

To know more about Terraform Resource Discovery, check here.

Now that we got an overview of what a provider is and services provided by some major providers, let's see how we can use one in our terraform configuration files.

Also Read: Our blog post on **Hashicorp Terraform**. Click here

Google Cloud (GCP) Terraform Provider

The Google Cloud Terraform Provider is used to configure your Google Cloud Platform infrastructure. It is collaboratively maintained by the Google Terraform Team at Google and the Terraform team at HashiCorp.

It looks over the lifecycle management of GCP resources, including Compute Engine, Cloud Storage, Cloud SDK, Cloud SQL, GKE, BigQuery, Cloud Functions, and more.

Below is a general provider configuration snippet:

```
provider "google" {
  project     = "my-project-id"
  region      = "us-central1"
}
```

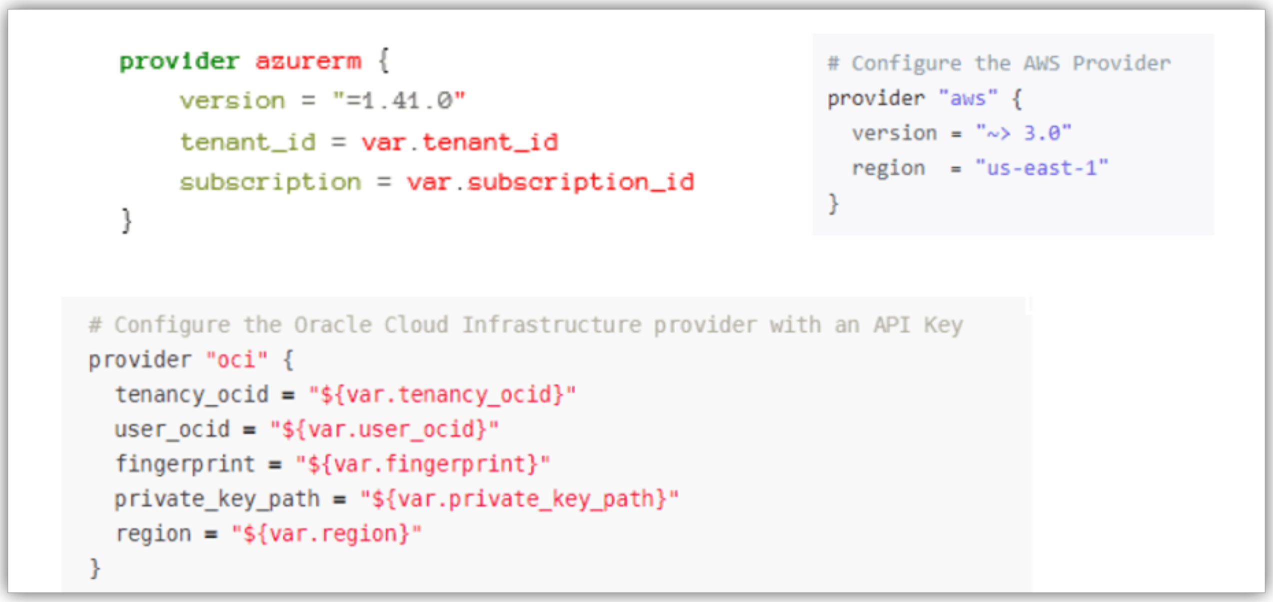
In order to get started with Google Terraform Provider, you need to have a Google Cloud Free Trial Account with billing enabled and terraform installed in your project. Follow the steps from this guide to create your Google Cloud Free Trial Account and create a project.

Provider Block

Terraform configurations must declare which providers they require so that Terraform can install and use them. Providers are executable plugins that contain the code necessary to interact with the API of the service it was written for.

A provider is defined by a **provider** block, the actual arguments in a provider block vary depending on the provider, but all providers support the meta-arguments of version and alias.

The below image shows the provider block format across different providers.



```
provider azurerm {
    version = "=1.41.0"
    tenant_id = var.tenant_id
    subscription = var.subscription_id
}
```

```
# Configure the AWS Provider
provider "aws" {
  version = "~> 3.0"
  region  = "us-east-1"
}
```

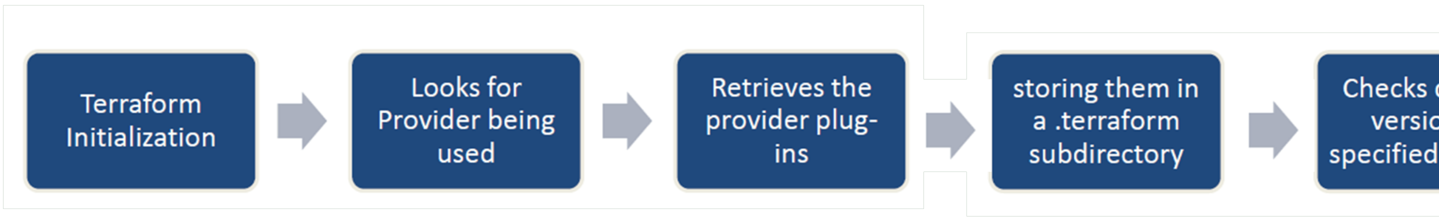
```
# Configure the Oracle Cloud Infrastructure provider with an API Key
provider "oci" {
  tenancy_ocid = "${var.tenancy_ocid}"
  user_ocid = "${var.user_ocid}"
  fingerprint = "${var.fingerprint}"
  private_key_path = "${var.private_key_path}"
  region = "${var.region}"
}
```

Check Out: How to **Install Terraform** on Mac, Windows & Ubuntu. Click here

Provider Workflow

Terraform finds and installs providers when initializing a working directory. It can automatically download providers from a Terraform registry, or load them from a local mirror or cache.

Hashicorp distributed providers are available for download automatically during Terraform initialization, while third-party providers must be placed in a local plug-ins directory located at either %APPDATA%\terraform.d\plugins for Windows or ~/.terraform.d/plugins for other operating systems.


Terraform Initialization
Looks for Provider being used
Retrieves the provider plug-ins
storing them in a .terraform subdirectory
Checks
versio
specified

The flow of steps performed is explained below:

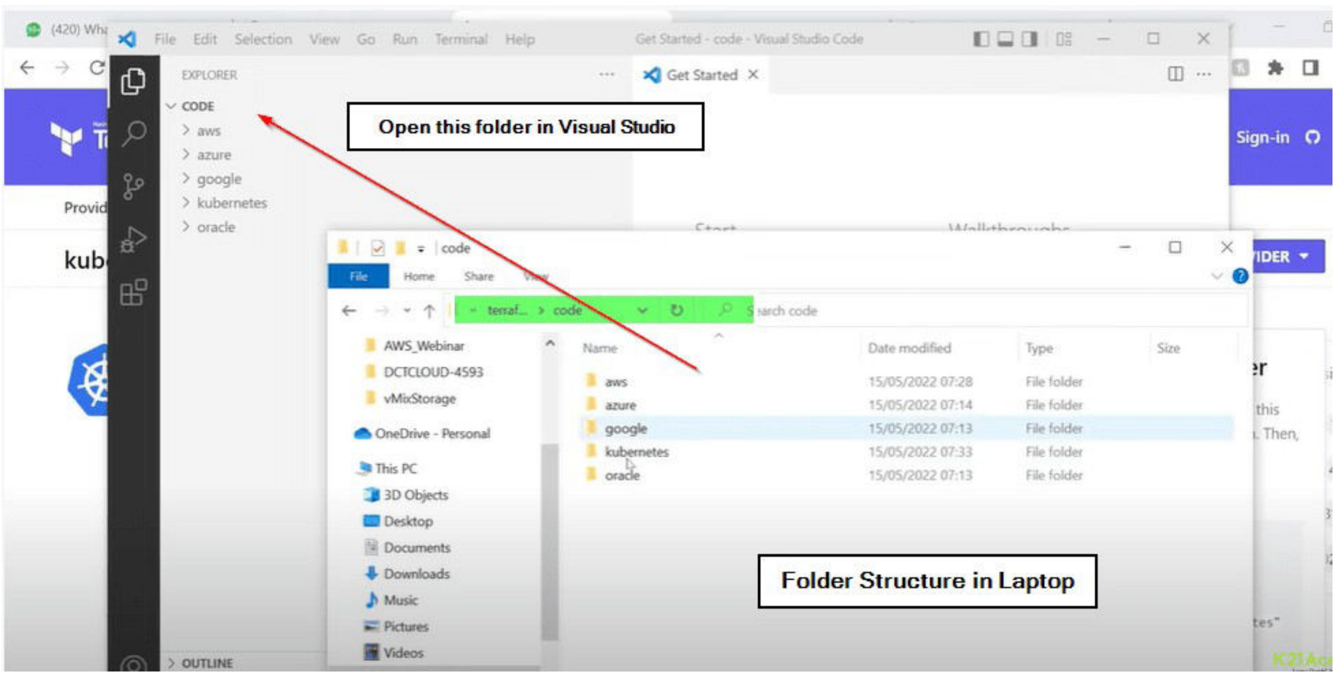
1. Initialize the Terraform configuration, i.e. run terraform init command.
2. It looks for provider being used, and download the provider plug-ins, if not found.
3. Retrieves the provider plug-ins
4. Stores them in a .terraform subdirectory
5. Check the default version or specified version

How To Use Providers

You can follow some simple steps to use your choice of provider be it AWS, Azure, Google Cloud, or Oracle in any IDE of your choice (here we are using VSCode- Visual Studio Code).

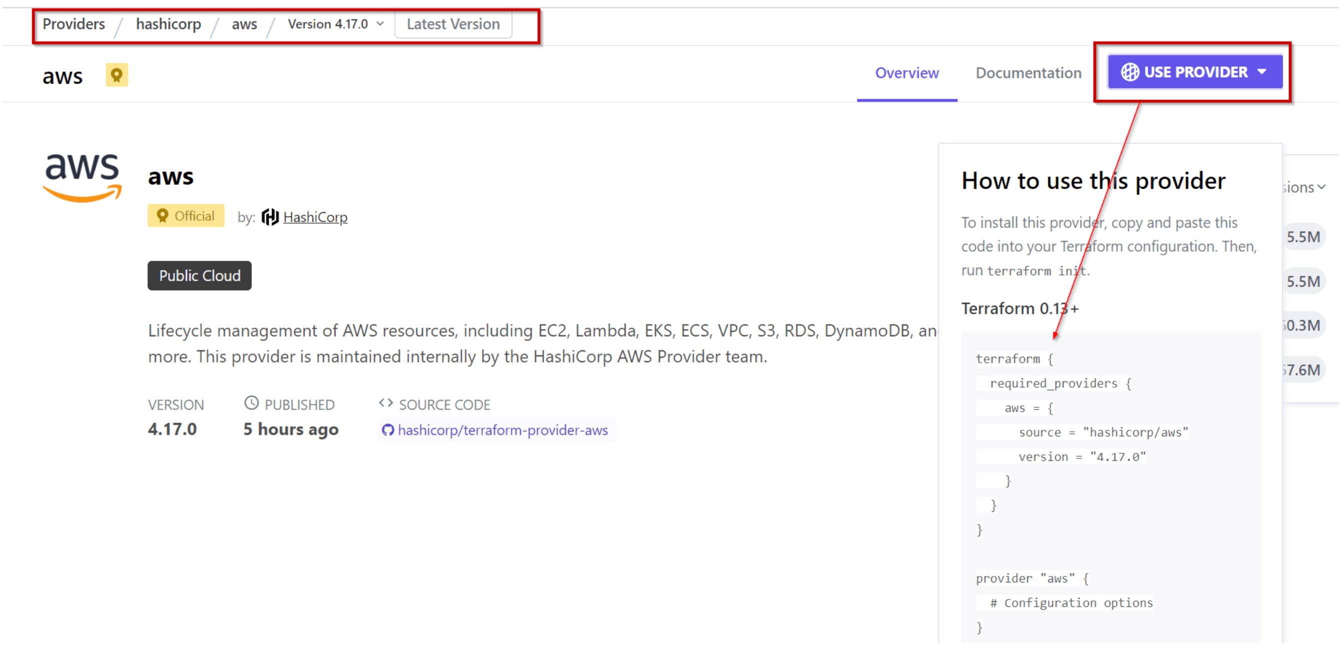
On your laptop, you can create a folder names Terraform and inside this folder create sub-folders like AWS, Azure, Google, etc.

Then open this complete folder in VSCode and also install the HashiCorp Terraform extension.



Name	Date modified	Type	Size
aws	15/05/2022 07:28	File folder	
azure	15/05/2022 07:14	File folder	
google	15/05/2022 07:13	File folder	
kubernetes	15/05/2022 07:33	File folder	
oracle	15/05/2022 07:13	File folder	

Now suppose you have to use the AWS provider, then in VSCode inside the AWS folder create a new file as the provider.tf and copy the code from the official AWS provider site in that file. And once done you need to initialize the terraform by running ***terraform init*** command in the terminal.



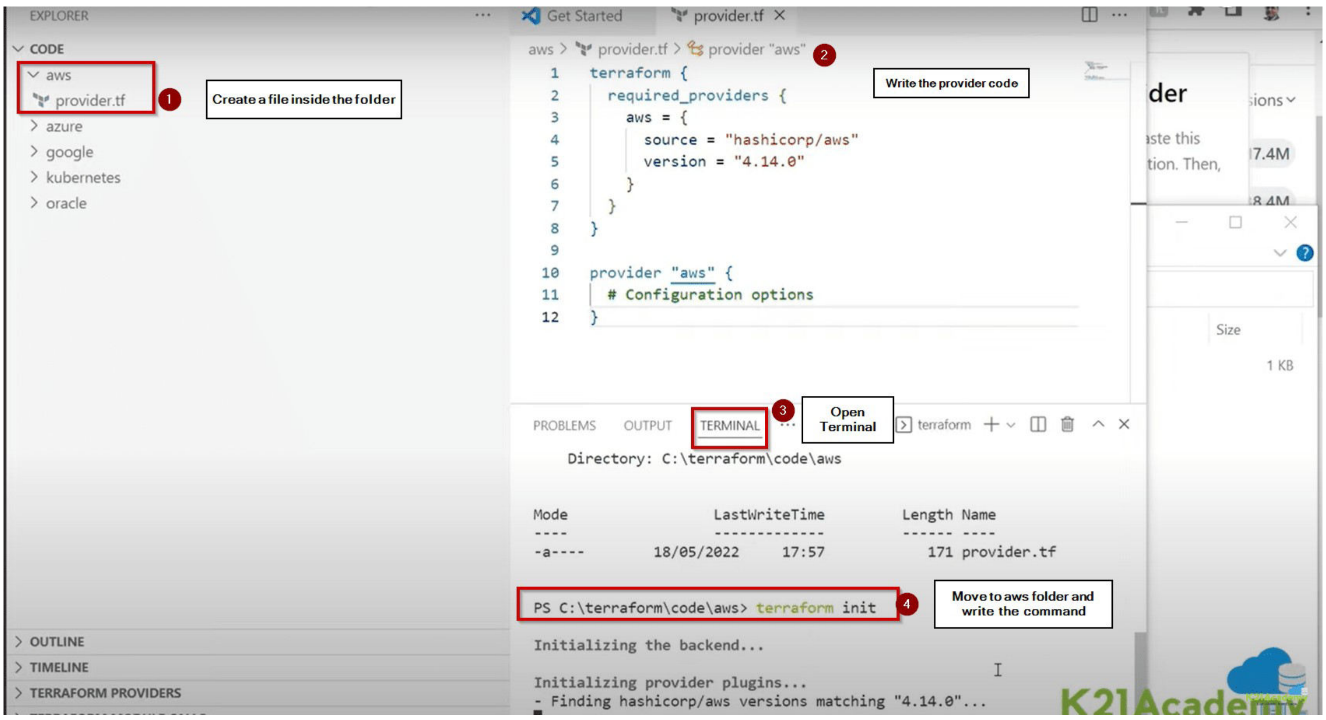
Lifecycle management of AWS resources, including EC2, Lambda, EKS, ECS, VPC, S3, RDS, DynamoDB, an more. This provider is maintained internally by the HashiCorp AWS Provider team.

How to use this provider

To install this provider, copy and paste this code into your Terraform configuration. Then, run `terraform init`.

Terraform 0.13+

```
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "4.17.0"
    }
  }
}

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



```
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "4.14.0"
    }
  }
}

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

```
    Directory: C:\terraform\code\aws


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----        18/05/2022     17:57            171 provider.tf

PS C:\terraform\code\aws> terraform init

Initializing the backend...

Initializing provider plugins...
- Finding hashicorp/aws versions matching "4.14.0"...
```

Download Terraform
https://www.terraform.io/downloads.html

Terraform CLI Documentation
https://www.terraform.io/docs/cli/index.html

Course: Using Terraform to Manage Applications and Infrastructure
https://bit.ly/3Bkr9nT