AWS Terraform Workshop
Build AWS Resources with Infrastructure as Code

The Slide Deck

Join the fun on your screen by scanning:

https://bit.ly/3G82pGJ

⚙️ If you haven't set up your environment yet, please refer to the **configuration.md** file in the repository.

Introduction

- Abdellah Boufous
- Data Engineer
- Used Terraform for learning and deploying cloud infrastructure
- VSCode

Brief Introduction on Infrastructure as Code

Terraform

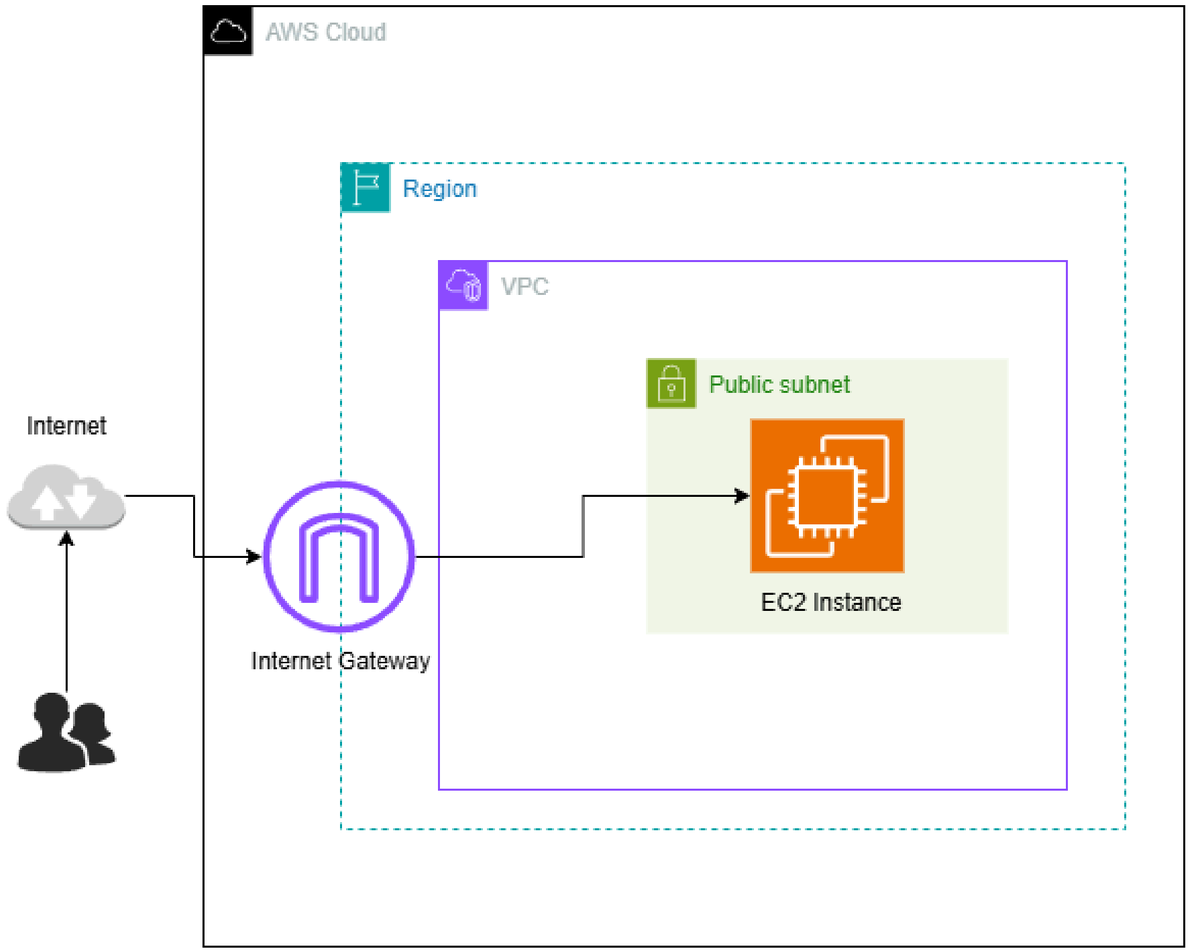


Table of Contents

Chapter 1
Introduction to Terraform

What is Infrastructure as Code?

Infrastructure as Code (IaC) is the process of managing and provisioning cloud infrastructure with machine-readable definition files.

Think of it as **executable documentation**.

Infrastructure as Code Allows Us To...

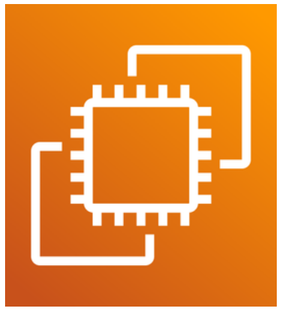
terraform

You describe your infrastructure by writing code

You get Infrastructure Deployed

```
resource "aws_instance" "wordpress_01" {
  ami                    = data.aws_ssm_parameter.latest_ami.value
  instance_type          = "t2.micro"
  subnet_id              = data.aws_subnet.default.id
  vpc_security_group_ids = [data.aws_security_group.default.id]

  tags = {
    Name = "wordpress-01"
  }
}
```

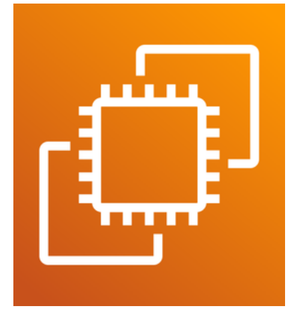


t2.micro

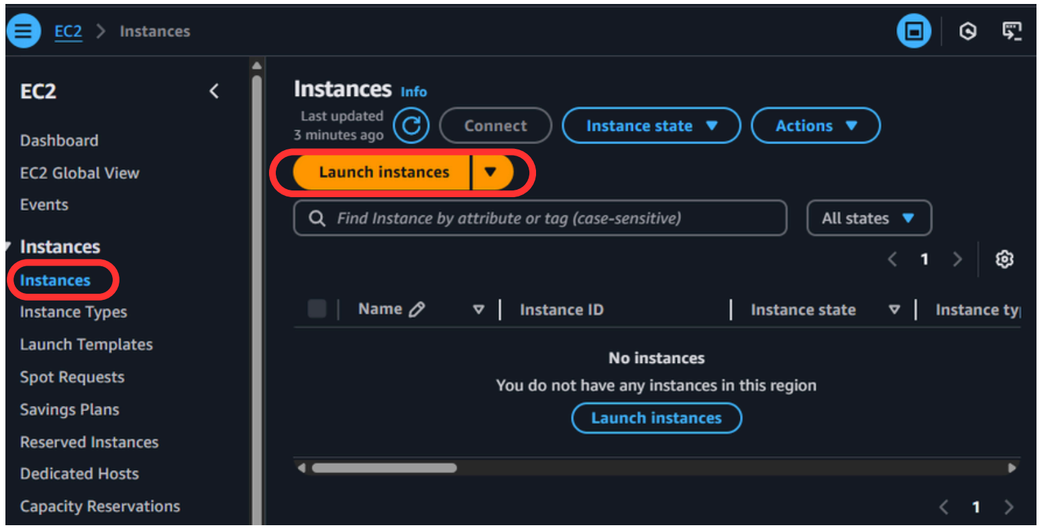
How to Provision an AWS EC2 Instance

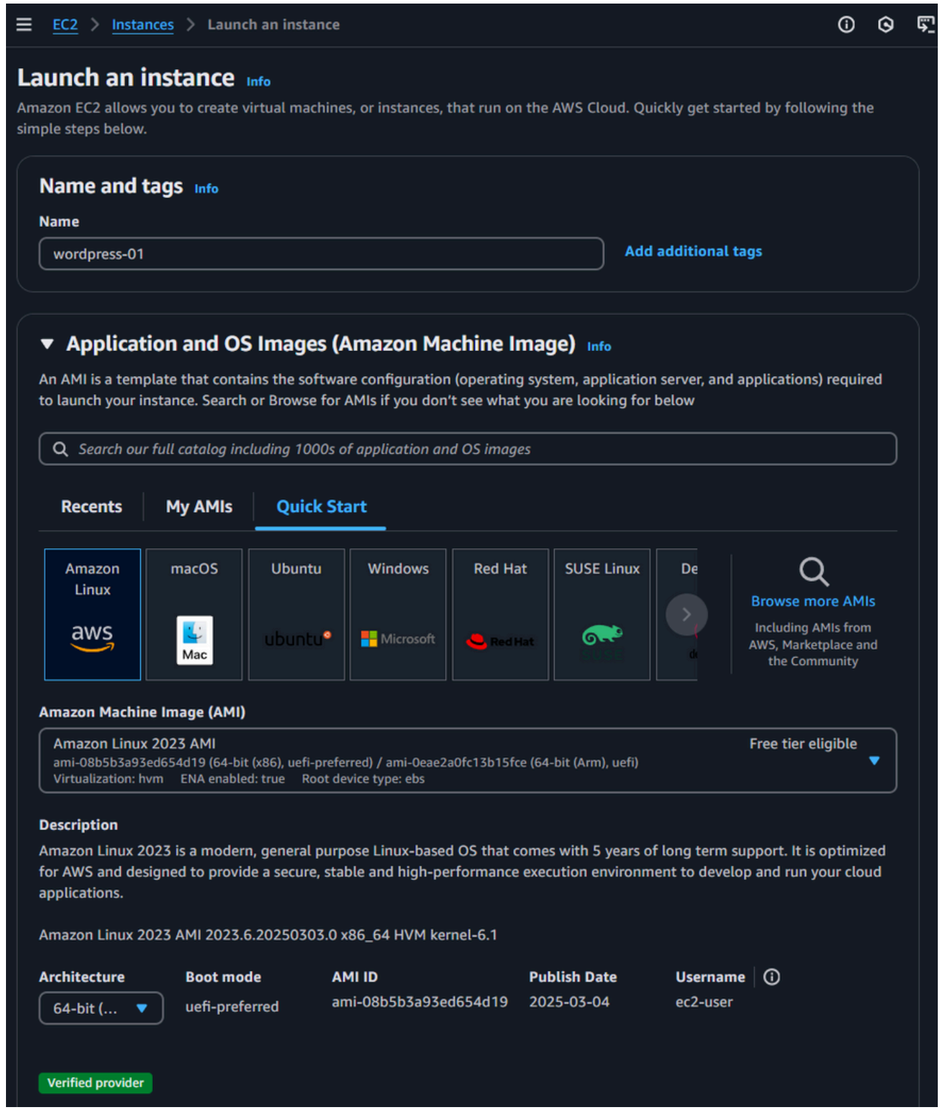
Let's look at a few different ways you could provision a new AWS EC2 Instance. Before we start we'll need to gather some basic information including (but not limited to):

- Instance Name : ec2-instance-01
- Operating System (Image) : Amazon Linux Image 2023
- VM Size : t2.micro
- Geographical Location (Region) : us-east-01
- Security Groups

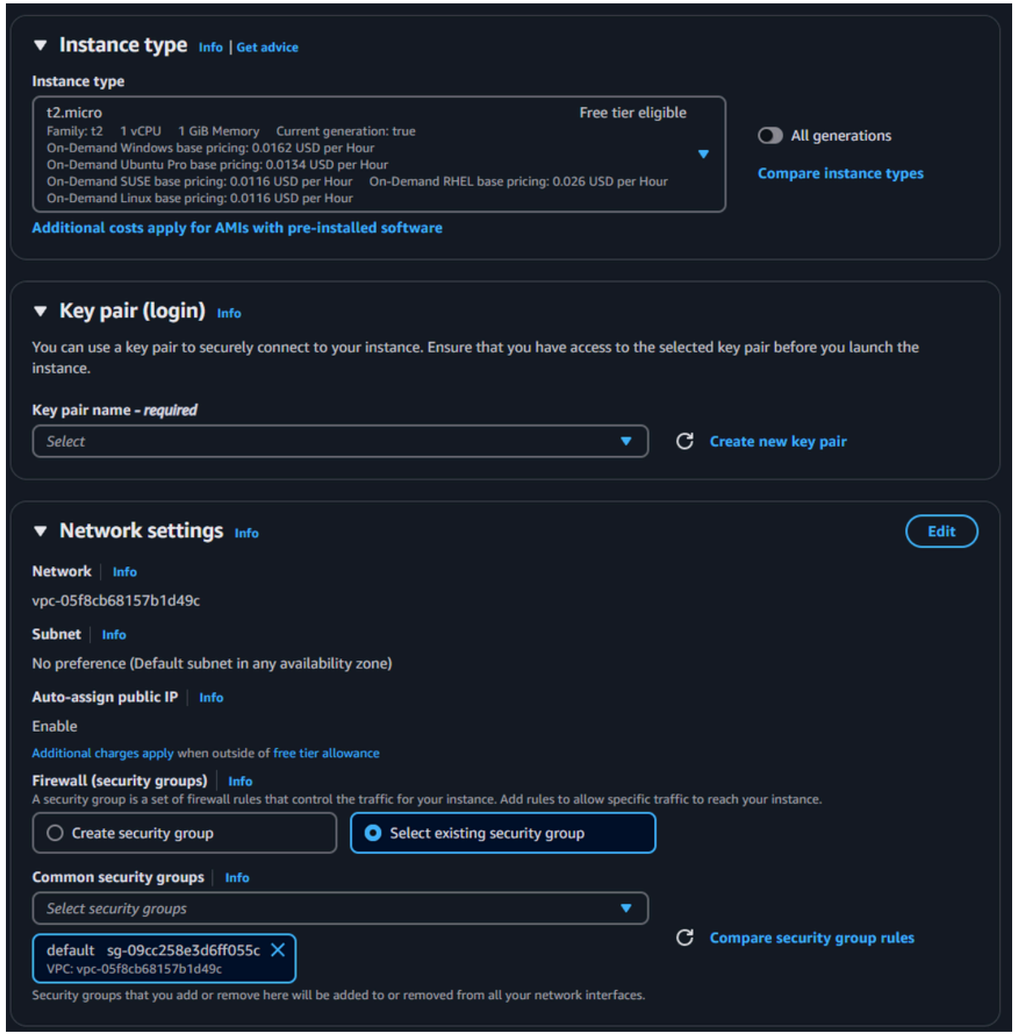


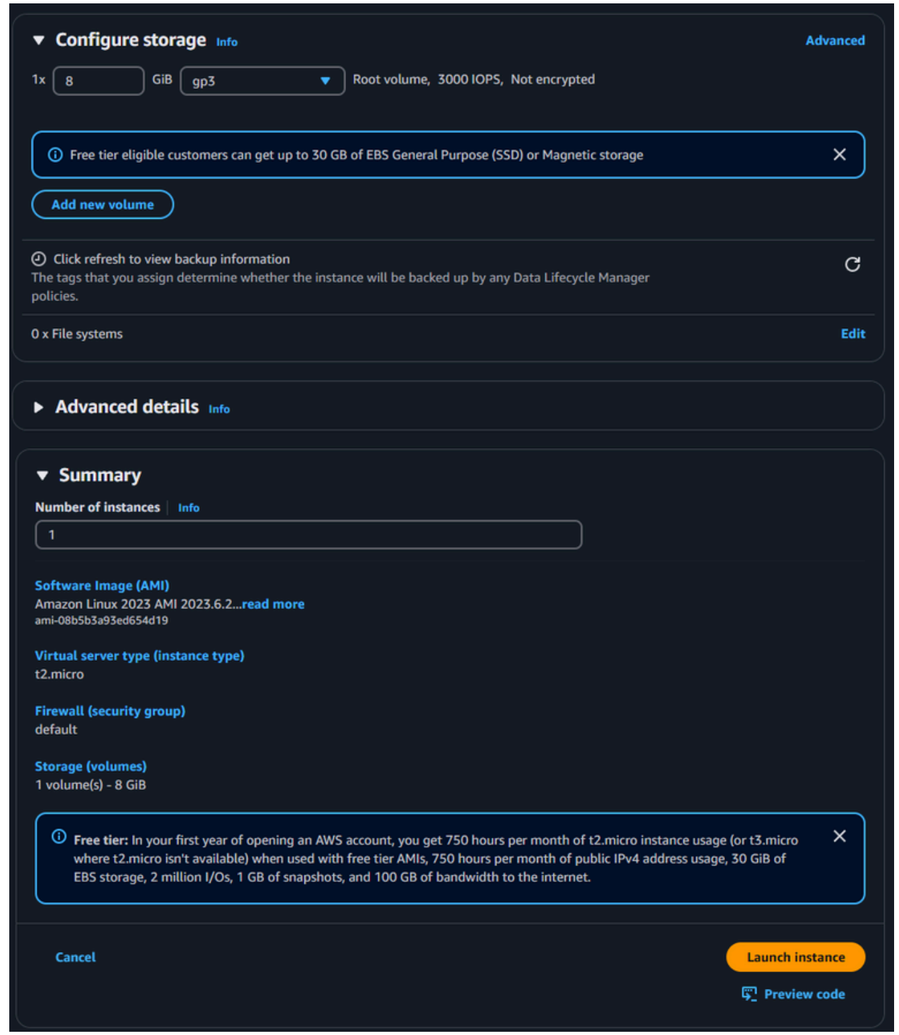
Method 1: AWS Console (GUI)





Method 1: AWS Console (GUI)





Method 2: Cloudformation (YAML)

CloudFormation Templates provide a consistent and reliable way to provision AWS resources.

```yaml
AWSTemplateFormatVersion: '2010-09-09'
Description: CloudFormation Template to provision an EC2 instance named wordpress-01 in us-east-1 using
 the default VPC and security group, with no SSH access and no key pair.

Resources:
  WordPressInstance:
    Type: AWS::EC2::Instance
    Properties:
      InstanceType: t2.micro  # Change as needed
      ImageId: !Ref LatestAmiId
      SecurityGroupIds:
        - !Ref DefaultSecurityGroup
      SubnetId: !Ref DefaultSubnet
      Tags:
        - Key: Name
          Value: wordpress-01
```

Method 3: Provision with Terraform

Example Terraform code for building an AWS instance.

```
resource "aws_instance" "wordpress_01" {
  ami                    = data.aws_ssm_parameter.latest_ami.value
  instance_type          = "t2.micro"
  subnet_id              = data.aws_subnet.default.id
  vpc_security_group_ids = [data.aws_security_group.default.id]

  tags = {
    Name = "wordpress-01"
  }
}
```

What is Terraform?

```
resource "aws_instance" "wordpress_01" {
  ami                    = data.aws_ssm_parameter.latest_ami.value
  instance_type          = "t2.micro"
  subnet_id              = data.aws_subnet.default.id
  vpc_security_group_ids = [data.aws_security_group.default.id]

  tags = {
    Name = "wordpress-01"
  }
}
```

- Executable Documentation (IaC)
- Human and machine readable
- Easy to learn and use
- Works on all major cloud providers

Infrastructure as Code Allows Us To...

- Automate the creation and management of cloud resources

Infrastructure as Code Allows Us To...

- Automate the creation and management of cloud resources
- Easily update and change existing infrastructure

Infrastructure as Code Allows Us To...

- Automate the creation and management of cloud resources
- Easily update and change existing infrastructure
- Safely test changes with a "dry run" mode

Infrastructure as Code Allows Us To...

- Automate the creation and management of cloud resources
- Easily update and change existing infrastructure
- Safely test changes with a "dry run" mode
- Integrate with Git and CI/CD tools

Infrastructure as Code Allows Us To...

- Automate the creation and management of cloud resources
- Easily update and change existing infrastructure
- Safely test changes with a "dry run" mode
- Integrate with Git and CI/CD tools
- Share reusable modules for teamwork

Infrastructure as Code Allows Us To...

- Automate the creation and management of cloud resources
- Easily update and change existing infrastructure
- Safely test changes with a "dry run" mode
- Integrate with Git and CI/CD tools
- Share reusable modules for teamwork
- Enforce security and company standards

Infrastructure as Code Allows Us To...

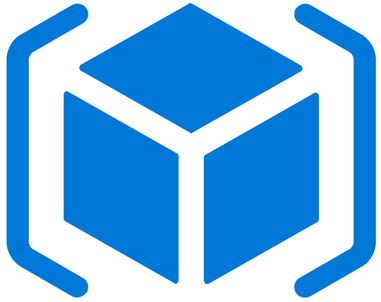
- Automate the creation and management of cloud resources
- Easily update and change existing infrastructure
- Safely test changes with a "dry run" mode
- Integrate with Git and CI/CD tools
- Share reusable modules for teamwork
- Enforce security and company standards
- Enable collaboration between teams

Other Infrastructure as Code Tools

These tools are great for configuring OS and apps, but not built for provisioning cloud infrastructure.

Native Cloud Provisioning Tools

AWS Cloudformation



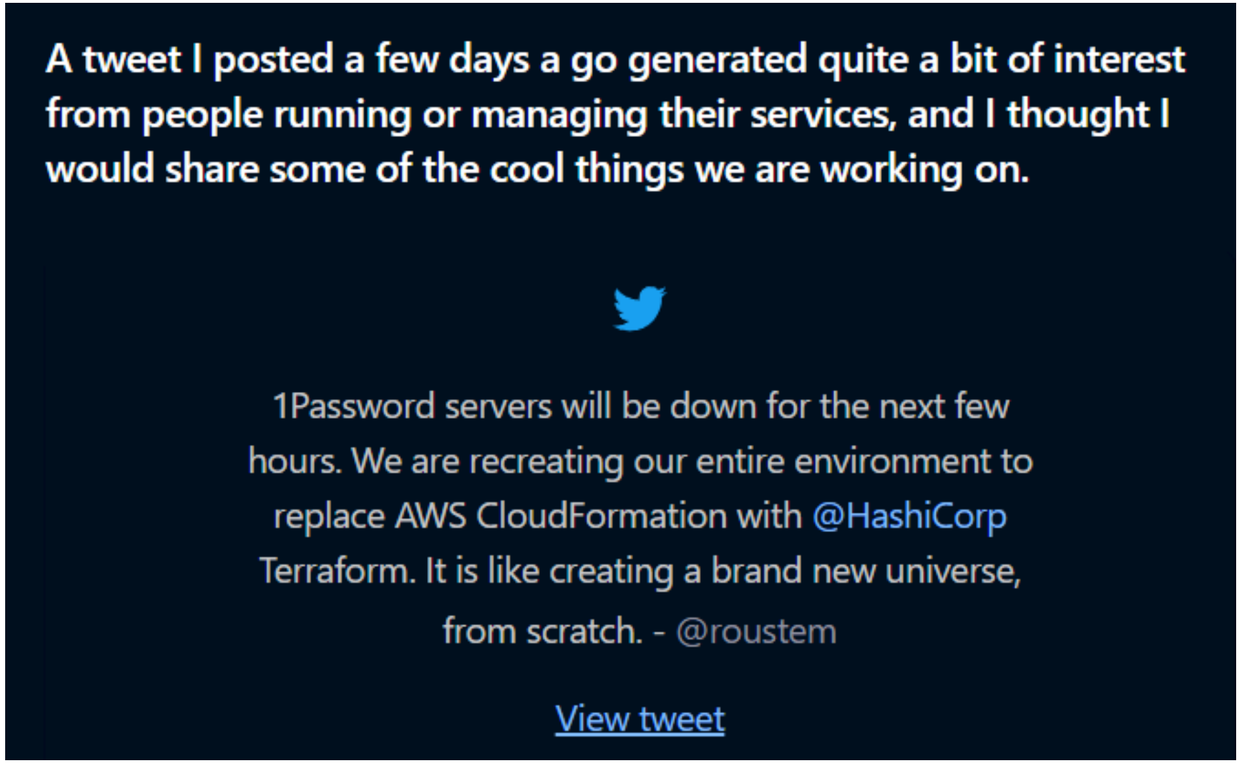
Azure Resource Manager

Google Cloud Deployment Manager

Each cloud has its own YAML or JSON based provisioning tool.

Terraform can be used across all major cloud providers and VM hypervisors.

Why Terraform?



https://blog.1password.com/terraforming-1password/

Why Terraform on AWS?

- Supports multi-cloud & hybrid infrastructure

Why Terraform on AWS?

- Supports multi-cloud & hybrid infrastructure
- Migrate from other cloud providers

Why Terraform on AWS?

- Supports multi-cloud & hybrid infrastructure
- Migrate from other cloud providers
- Increase provisioning speed

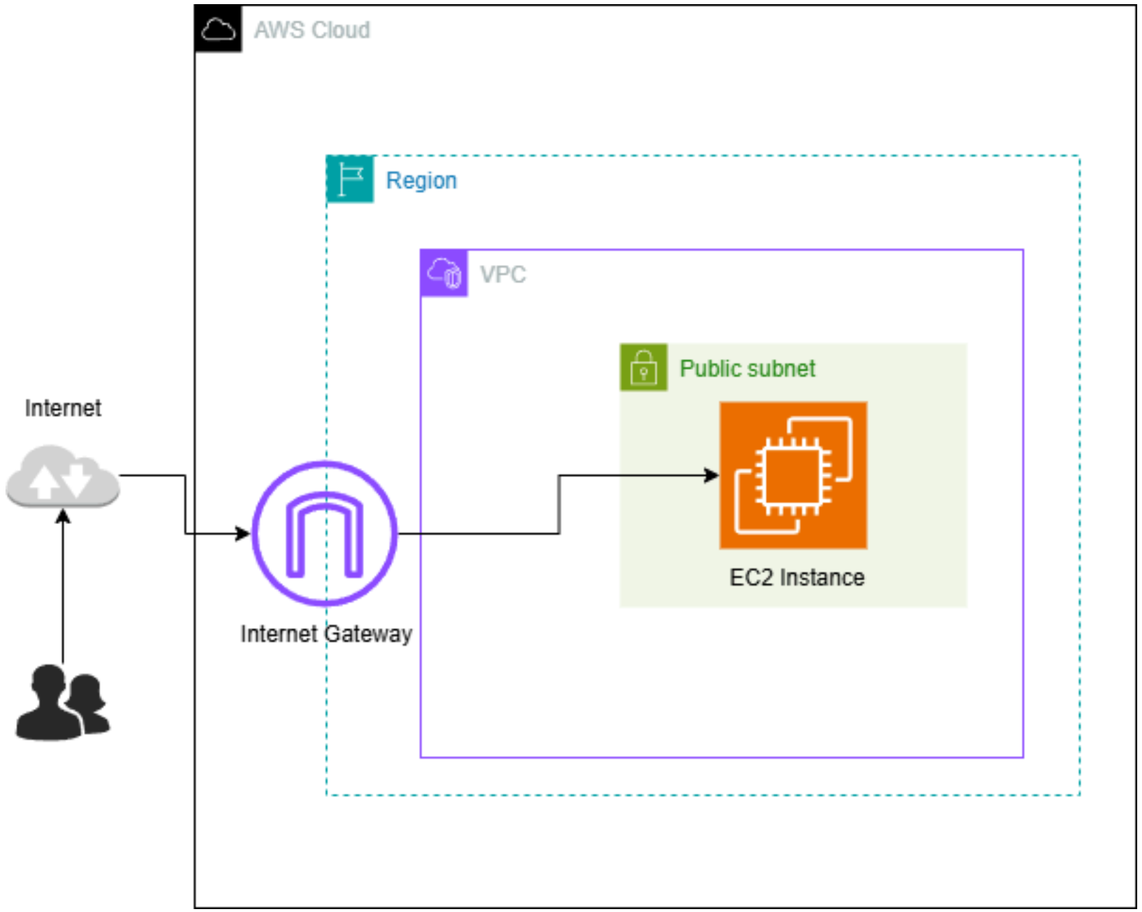
Why Terraform on AWS?

- Supports multi-cloud & hybrid infrastructure
- Migrate from other cloud providers
- Increase provisioning speed
- Improve efficiency

Why Terraform on AWS?

- Supports multi-cloud & hybrid infrastructure
- Migrate from other cloud providers
- Increase provisioning speed
- Improve efficiency
- Reduce risk (Disaster Recovery)

Live Demo
(Deploying EC2 Using Terraform)



👩‍🔬 Lab - Setup and Basic Usage

This lab introduces the basics of Terraform by creating a simple EC2 instance that you can access via SSH.

Basic EC2 Instance with SSH Access

Chapter 1 : Recap

- Terraform = Infrastructure as Code → Codify your infrastructure into version-controlled files
- Safe, Fast, and Scalable → Preview changes, automate deployments, reuse modules
- Built for Automation → Easily integrates with CI/CD pipelines for consistent, hands-free deployments

Chapter 2
Terraform Basics

What is Terraform?

HashiCorp Terraform

Mitchell Hashimoto

Armon Dadgar

- What it is: Free tool for provisioning cloud infrastructure.
- Cross-platform: Works on Linux, Windows, macOS.

Terraform Code

```
resource aws_vpc "main" {
  cidr_block       = "10.0.0.0/16"
  instance_tenancy = "dedicated"
}
```

Terraform code is based on the **HCL2 toolkit**. HCL stands for HashiCorp Configuration Language.

Terraform code, or simply terraform is a declarative language that is specifically designed for provisioning infrastructure on any cloud or platform.

Terraform Workspaces

A terraform workspace is simply a folder or directory that contains terraform code.

Terraform files always end in either a *.tf or *.tfvars extension.

Most terraform workspaces contain a minimum of three files:

main.tf - Most of your functional code will go here.
variables.tf - This file is for storing variables.
outputs.tf - Define what is shown at the end of a terraform run.

Terraform Workspaces

```
provider "aws" {
  region = "ca-central-1"
}

resource "aws_instance" "example" {
  ami           = var.ami_id
  instance_type = "t3.micro"
}
```

main.tf

Creates an EC2 instance using the AMI from the variable.

```
variable "ami_id" {
  description = "The AMI to use for the EC2 instance"
  type        = string
}
```

variables.tf

Which AMI id we wanna use for the EC2 Instance

```
output "instance_ip" {
  description = "Public IP of the EC2 instance"
  value       = aws_instance.example.public_ip
}
```

outputs.tf

Displays the public IP address of your new EC2 instance

Terraform Commands

Terraform is a command line tool.

Terraform commands are either typed in manually or run automatically from a script.

The commands are the same whether you are on Linux or Windows or MacOS.

Terraform has subcommands that perform different actions.

```
# Basic Terraform Commands
terraform version
terraform help
terraform init
terraform apply
terraform destroy
terraform graph
```

Terraform Workflow

init	Validate	Plan	Apply	Destroy
• Used to Initialize a working directory containing terraform config files • This is the first command that should be run after writing a new Terraform config • Downloads **Providers**	• Validates the terraform config files in that respective directory to ensure they are **Syntactically valid and internally consistent.**	• Creates an execution plan • Terraform performs a refresh and determines what actions are necessary to achieve the **desired state** specified in config files.	• Used to apply the changes required **to reach the desired state** of the configuration. • By default, apply scan the current directory for the config and applies the changes appropriately.	• Used to destroy the terraform managed infrastructure • This will ask for confirmation before destroying.

Terraform Provider Configuration

The terraform core program requires at least one provider to build anything.

You can manually configure which version(s) of a provider you would like to use. If you leave this option out, Terraform will default to the latest available version of the provider.

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 5.0"
    }
  }
}
```

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

Terraform Init

```
$ terraform init
Initializing provider plugins...
- Checking for available provider plugins...
- Downloading plugin for provider "aws" (hashicorp/aws) 5.0...
...
 provider.aws: version = "~> 5.0"
Terraform has been successfully initialized!
```

Terraform fetches any required providers and modules and stores them in the .terraform directory. If you add, change or update your modules or providers you will need to run init again.

Terraform Plan

```
$ terraform plan
An execution plan has been generated and is shown below.
Terraform will perform the following actions:
  # aws_vpc.main will be created
  + resource "aws_vpc" "main" {
      + arn                                  = (known after apply)
      + cidr_block                           = "10.0.0.0/16"
      ...
      + instance_tenancy                     = "dedicated"
    }
```

Preview your changes with **terraform plan** before you apply them.

Terraform Comments

Line Comments begin with an octothorpe*, or pound symbol: #

```
# This is a line comment.
```

Block comments are contained between /* and */ symbols.

```
/* This is a block comment.
Block comments can span multiple lines.
The comment ends with this symbol: */
```

Terraform Help

```
$ terraform help
Usage: terraform [-version] [-help] <command> [args]
...
Common commands:
    apply              Builds or changes infrastructure
    console            Interactive console for Terraform interpolations
    destroy            Destroy Terraform-managed infrastructure
    env                Workspace management
    fmt                Rewrites config files to canonical format
    graph              Create a visual graph of Terraform resources
```

Type **terraform subcommand** help to view help on a particular subcommand.

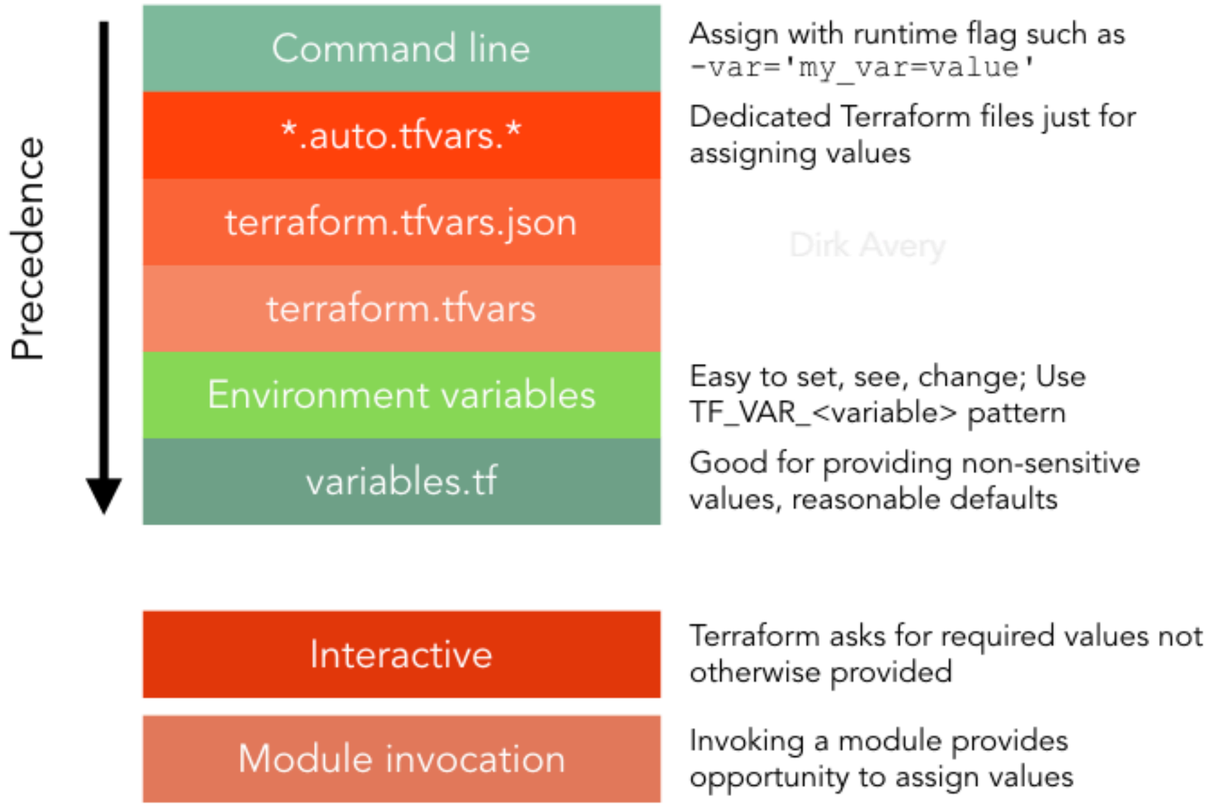
Where are Variables Defined?

Terraform variables are placed in a file called variables.tf. Variables can have default settings. If you omit the default, the user will be prompted to enter a value. Here we are declaring the variables that we intend to use.

```
variable "prefix" {
  description = "This prefix will be included in the name of most resources."
}
variable "instance_tenancy" {
  description = "A tenancy option for instances launched into the VPC."
  default     = "dedicated"
}
```

How are Variables Set?

Once you have some variables defined, you can set and override them in different ways. Here is the level of precedence for each method.



👩‍💻 Lab - Terraform Variables for AWS Provisioning

This lab demonstrates how to create an EC2 instance using Terraform with variables for better configuration management and reusability.

Lab - Terraform Variables for AWS Provisioning

📝 Chapter 2 Review

In this chapter we:

- Used the **terraform init** command
- Ran the **terraform plan** command
- Learned about variables
- Usecase : Set prefixes using variables

Chapter 3
Terraform in Action

Anatomy of a Resource

Every terraform resource is structured exactly the same way.

```
resource type "name" {
  parameter = "foo"
  parameter2 = "bar"
  list = ["one", "two", "three"]
}
```

resource = Top level keyword
type = Type of resource. Example: aws_instance.
name = Arbitrary name to refer to this resource. Used internally by terraform. This field cannot be a variable.

Terraform Apply

```
$ terraform apply
An execution plan has been generated and is shown below.
Terraform will perform the following actions:
  # aws_vpc.main will be created
  + resource "aws_vpc" "main" {
      + cidr_block                          = "10.0.0.0/16"
      + instance_tenancy                    = "dedicated"
        ...
      + tags                                = {
          + "Name" = "main"
        }
    }
Plan: 1 to add, 0 to change, 0 to destroy.
```

terraform apply runs a plan and then if you approve, it applies the changes.

Terraform Destroy

```
$ terraform destroy
An execution plan has been generated and is shown below.
Terraform will perform the following actions:
  # aws_vpc.main will be destroyed
  - resource "aws_vpc" "main" {
      - cidr_block                       = "10.0.0.0/16" -> null
      - instance_tenancy                 = "dedicated" -> null
        ...
      - tags                             = {
          - "Name" = "main"
        } -> null
    }
Plan: 0 to add, 0 to change, 1 to destroy.
```

terraform destroy does the opposite. If you approve, your infrastructure is destroyed.

Terraform Format

```
resource "aws_instance" "example" { instance_type="t2.micro"
ami = "ami-12345678"
tags={ Name="example-instance"}}
```

```
resource "aws_instance" "example" {
  instance_type = "t2.micro"
  ami           = "ami-12345678"

  tags = {
    Name = "example-instance"
  }
}
```

Terraform comes with a built in code formatter/cleaner. It can make all your margins and list indentation neat and tidy. Beauty works better.

Terraform Data Sources

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

# Get latest Amazon Linux 2 AMI
data "aws_ami" "amazon_linux" {
  most_recent = true

  filter {
    name   = "name"
    values = ["amzn2-ami-hvm-*-x86_64-gp2"]
  }

  filter {
    name   = "virtualization-type"
    values = ["hvm"]
  }

  owners = ["137112412989"] # Amazon official account
}
```



```hcl
# Create EC2 Instance using latest AMI
resource "aws_instance" "web" {
  ami           = data.aws_ami.amazon_linux.id
  instance_type = "t2.micro"

  tags = {
    Name = "AmazonLinuxLatest"
  }
}
```

Data sources are a way of querying a provider to return an existing resource, so that we can access its parameters for our own use.

Terraform Authentification

Terraform needs to authenticate to AWS to manage resources.
There are multiple ways to do this:

AWS Access Key & Secret Key (not recommended for production)

```
provider "aws" {
  region     = "ca-central-1"
  access_key = "YOUR_ACCESS_KEY"
  secret_key = "YOUR_SECRET_KEY"
}
```

AWS CLI Default Profile

```
aws configure
```

Terraform will automatically pick up ~/.aws/credentials

Environment Variables (recommended for local dev)

```
export AWS_ACCESS_KEY_ID="your_access_key"
export AWS_SECRET_ACCESS_KEY="your_secret_key"
```

Assume Role (for advanced setups, teams, or CI/CD)

Terraform Authentification

Terraform needs to authenticate to AWS to manage resources.
There are multiple ways to do this:

AWS Access Key & Secret Key (not recommended for production)

```
provider "aws" {
  region     = "ca-central-1"
  access_key = "YOUR_ACCESS_KEY"
  secret_key = "YOUR_SECRET_KEY"
}
```

AWS CLI Default Profile

`aws configure`

Terraform will automatically pick up ~/.aws/credentials

Environment Variables (recommended for local dev)

```
export AWS_ACCESS_KEY_ID="your_access_key"
export AWS_SECRET_ACCESS_KEY="your_secret_key"
```

Assume Role (for advanced setups, teams, or CI/CD)

👩‍💻 Lab Exercise: Terraform Modules

This lab demonstrates how to use Terraform modules to create reusable infrastructure components. The project creates a VPC with a public subnet and launches an EC2 instance within it.

Terraform Modules

🔧 Next Steps / Further Reading

Terraform : Get Started - AWS

Terraform Documentation

HashiCorp Terraform Associate Certification Course (003)

Spacelift's Tutorial : Getting Started with Terraform

Happy Terraforming on AWS