Infrastructure as Code with Terraform

Key Takeaways

TECHWORLD WITH NANA

Introduction to Terraform - 1

- An **infrastructure as code (IaC) tool** by HashiCorp
- Let's you **automate and manage**:

  your infrastructure | your platform | services that run on that platform

- By defining the resources in human-readable **configuration files** that you can:

  ✓ version ✓ reuse ✓ share

- Definition of configuration files is **declarative**!

  Declarative = define **WHAT** end result or desired state you want

  Imperative = define exact steps - HOW

Introduction to Terraform - 2

Infrastructure Provisioning

Prepare Infrastructure:

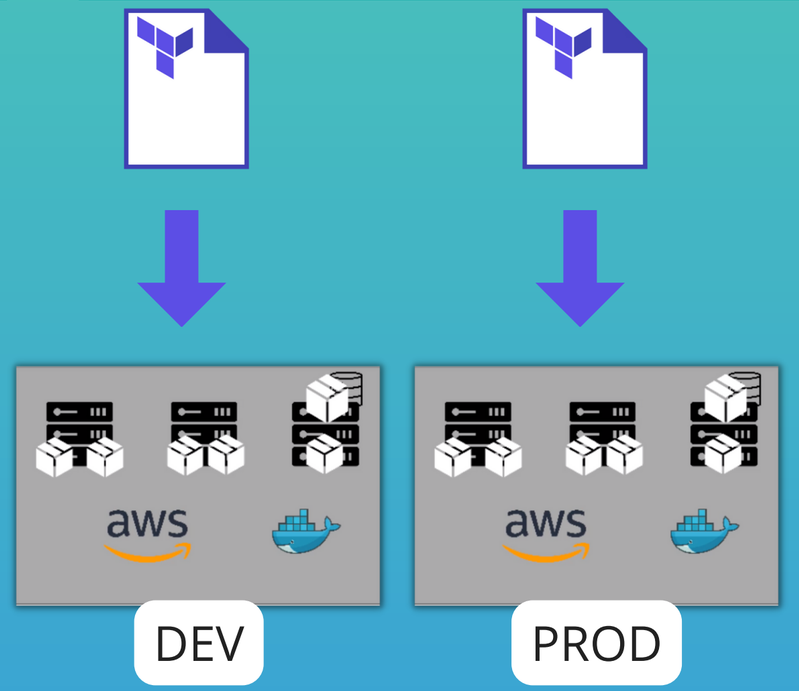
1. Private Network space
2. EC2 server instances
3. Install Docker and other tools
4. Configure Security

Manage existing infratructure

- **Automate** the **continuous changes** to your infrastructure

Replicating Infrastructure

- Easily **replicate** infrastructure **on different environments**, like DEV, STAGING, PROD



Differences between Ansible and Terraform

HashiCorp Terraform

- Mainly **infrastructure provisioning tool**
- More advanced in orchestration

Better for: Provisioning the infrastructure

Both:
Infrastructure as Code

- Mainly a **configuration management tool**

Better for: Configuring the provisioned infrastructure

How Terraform works - 1

An important part of Terraform (TF) is that TF:

- knows your **desired state (config file**) and
- keeps track of your existing real infrastructure (in a **state file**)

- TF **compares your desired with actual state** to know which changes it needs to make to your infrastructure
- **Without the state**, you would always have to check the current state yourself and see how to update your desired state!

How Terraform works - 2

This is the core workflow

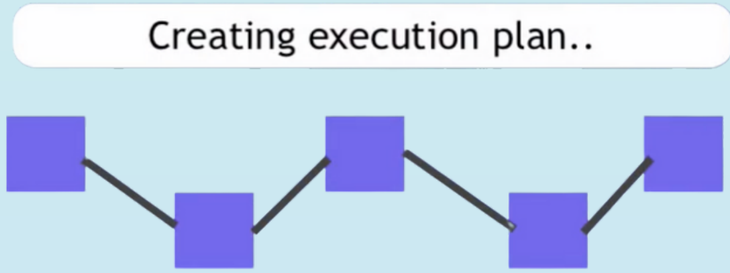
1) Write

- Define your infrastructure resources in the configuration file
- E.g. creating 2 EC2 instances on AWS

main.tf

2) Plan

- Based on the config file and state file TF creates an **execution plan** - the changes it will make to your infrastructure

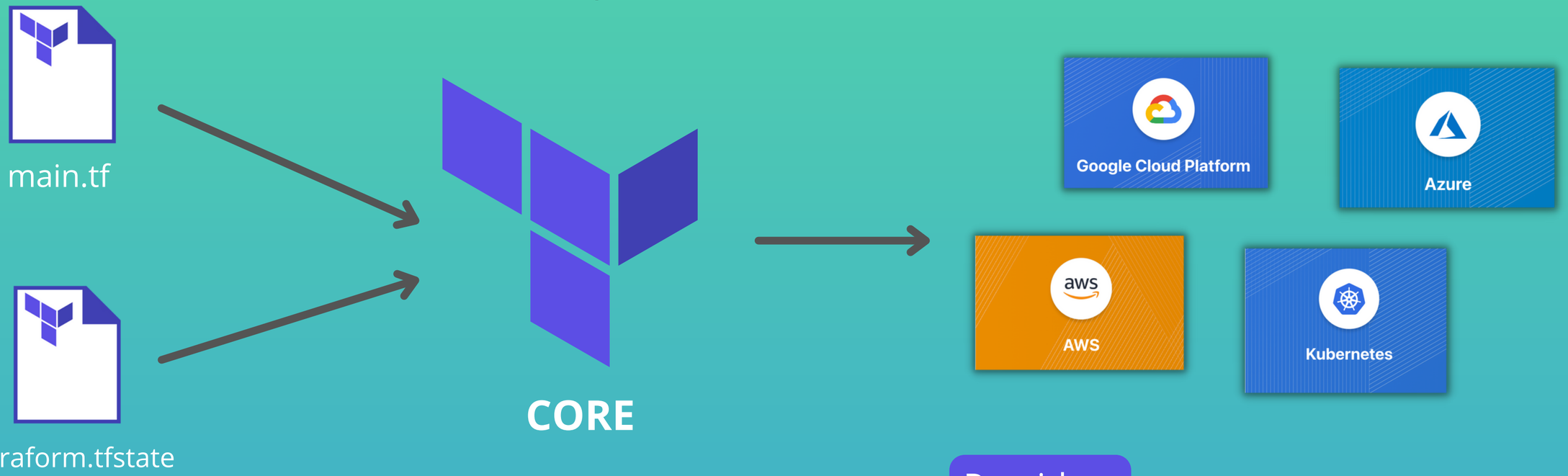


3) Apply

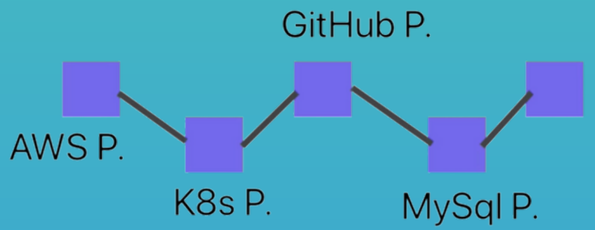
- TF **executes the changes** to your infrastructure
- Then **updates the state file**

Terraform Architecture

- TF executes the plan with Providers
- real infrastructure



- 2 input sources



Providers

- IaaS: AWS | Azure
- PaaS: Kubernetes
- SaaS: Fastly

- Providers enable TF to work with any platform or service

Core Terraform Commands

Example configuration file - the desired state

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

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

# Create a VPC
resource "aws_vpc" "example" {
  cidr_block = "10.0.0.0/16"
}
```

refresh

- query infrastructure provider to get current state

plan

- create an execution plan and review it

apply

- actually execute the plan

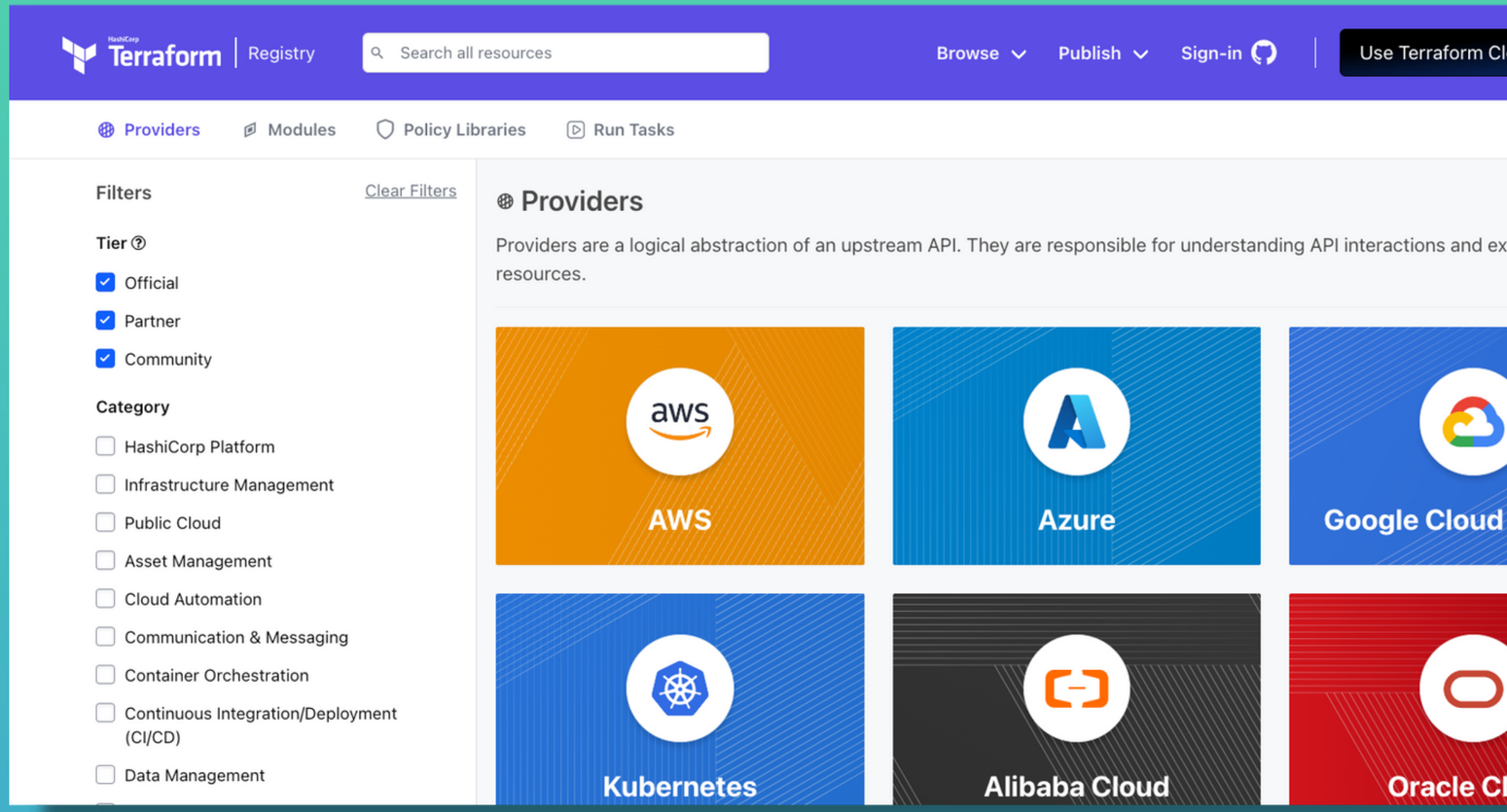
destroy

- destroy the resources/infrastructure

TECHWORLD WITH NANA

Terraform Providers

- Plugins Terraform uses to manage the resources
- Providers expose resources for specific infrastructure platform (e.g. AWS)
- **Responsible for understanding API** of that platform
- It is just code that knows how to talk to specific technology or platform



TECHWORLD WITH NANA

Resources and Data Sources

Resources

- To create a new resource

Data Sources

- To query an existing resource

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

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

# Create a VPC
resource "aws_vpc" "example" {
  cidr_block = "10.0.0.0/16"
}
```

Resource

Data Source

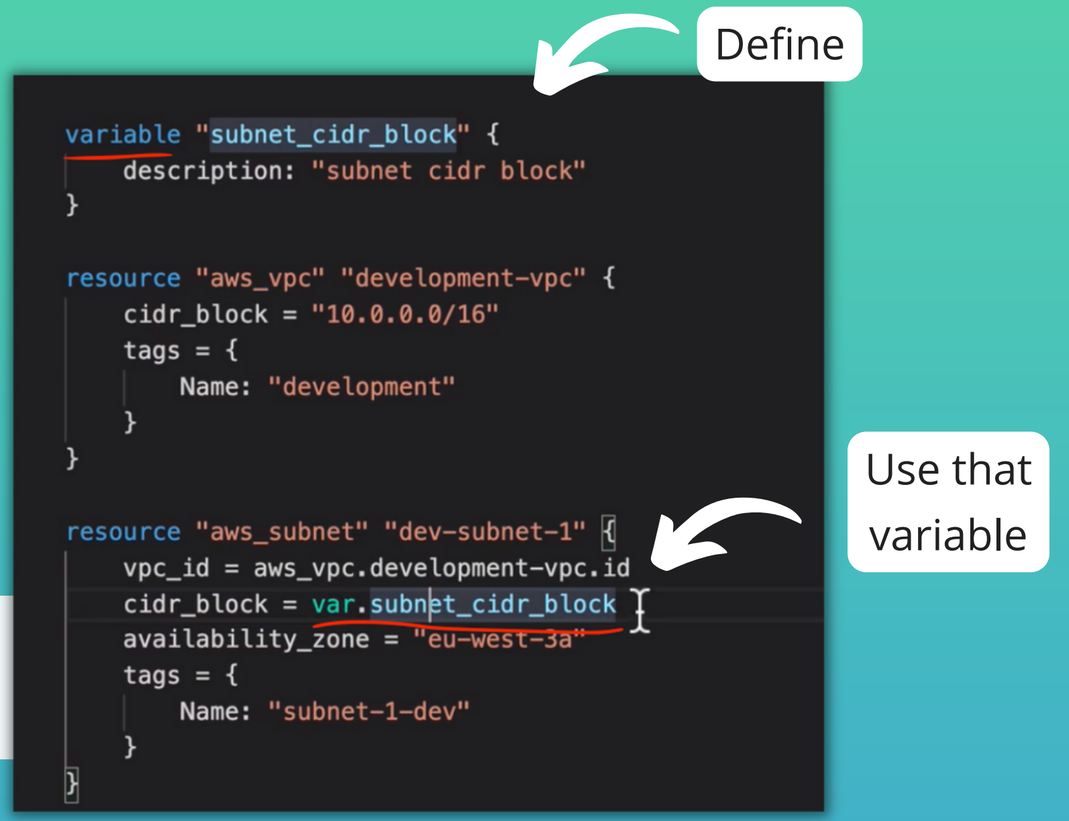
```
data "aws_ami" "example" {
  most_recent = true

  owners = ["self"]
  tags = {
    Name   = "app-server"
    Tested = "true"
  }
}
```

Variables in Terraform - 1

- Variables (input variables) let you **customize** behavior without editing the Terraform configuration file

2 steps to use variables in Terraform

1) **Define** variable and **use** it in your TF script:



2) **Set** the variable **values** when applying the script

TECHWORLD WITH NANA

Variables in Terraform - 2

- **3 ways** to set values for the defined variables

1) **Interactively** when applying the TF script

```
nana@macbook /Users/nana/terraform [master]
% terraform apply
var.subnet_cidr_block
  Enter a value: 10.0.20.0/24
```

2) Better: Set **as CLI option**

```
nana@macbook /Users/nana/terraform [feature/provisioners]
% terraform apply -var "subnet_cidr_block=10.0.30.0/24"
```

3) Even better: Set **in variable file**

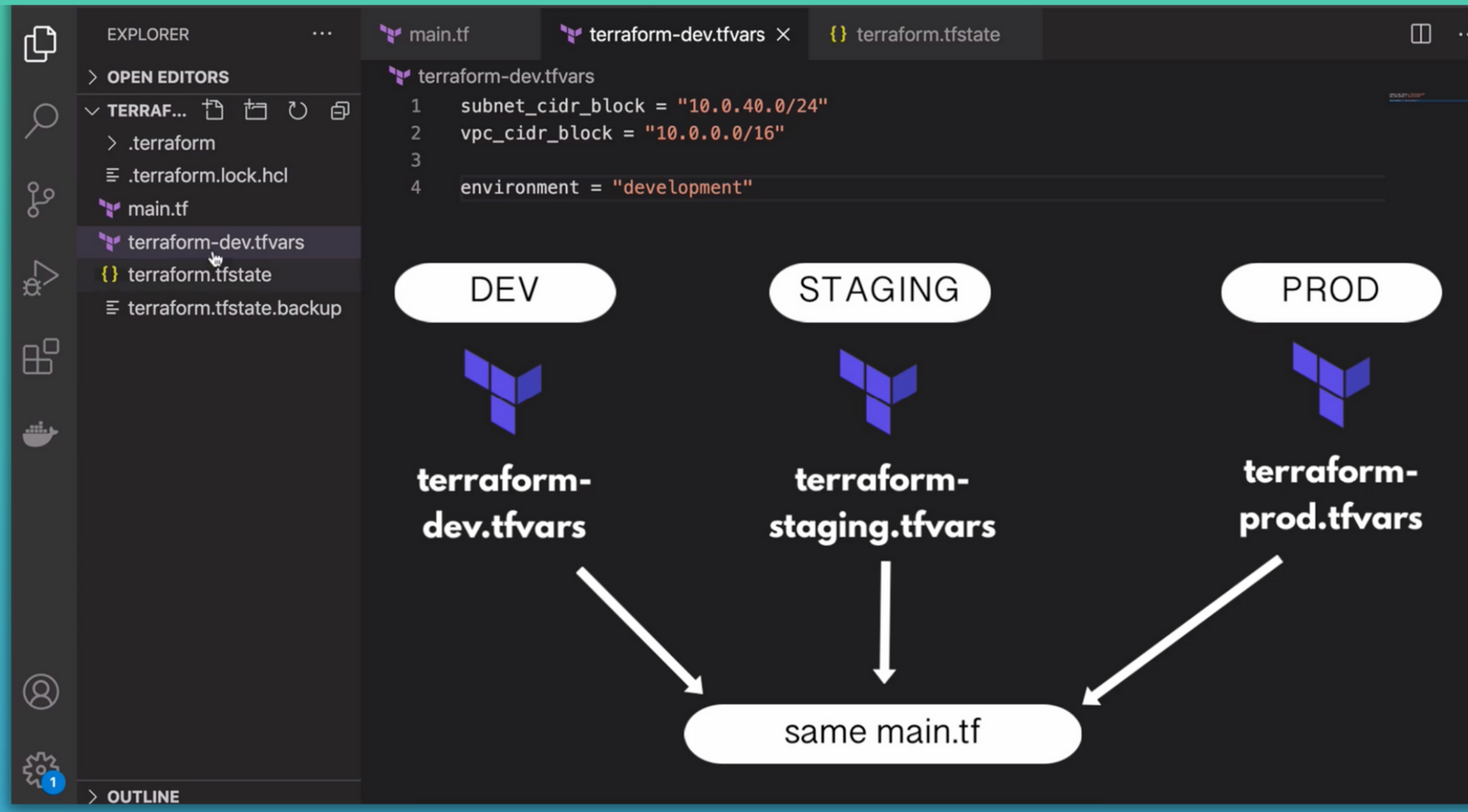
terraform.tfvars

```
1   subnet_cidr_block = "10.0.40.0/24"
```

Variables in Terraform - 3

Real use case with variables:

- Same script parameterized with variables
- And **own vars file for each environment**



TECHWORLD WITH NANA

Environment Variables in Terraform

Predefined Env Vars

- **TF-Env Vars**: TF has env vars, which you can use to change some of TF's default behavior, for example enabling detailed logs
- **AWS Env Vars**: Set AWS credentials for AWS provider as environment variable

```
export TF_LOG=off
```

```
provider "aws" {}
```

Usage:

```
$ export AWS_ACCESS_KEY_ID="anaccesskey"
$ export AWS_SECRET_ACCESS_KEY="asecretkey"
$ export AWS_DEFAULT_REGION="us-west-2"
$ terraform plan
```

Define your own custom Env Vars

- This is technically the **4th way of setting a variable value**, because we **define a variable** and **set its value** through TF environment variable

```
nana@macbook /Users/nana/terraform [feature/provisioners]
% export TF_VAR_avail_zone="eu-central-1a"
```

```
variable avail_zone {}

resource "aws_vpc" "development-vpc" {
    cidr_block = var.cidr_blocks[0].cidr_block
    tags = {
```

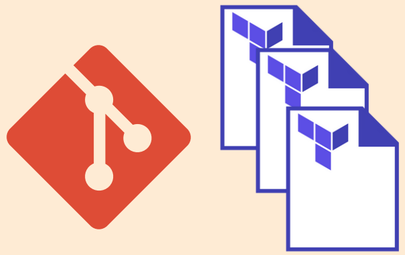
Git Repository for Terraform Project

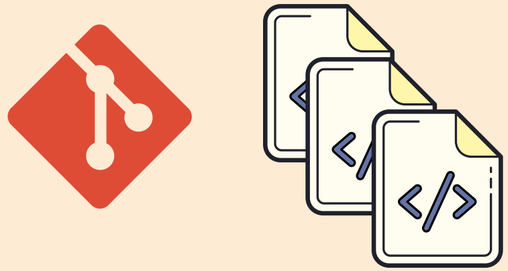
- Like your application code, Terraform scripts being Infrastructure as code, should be **managed by version control system Git and be hosted in a git repository**

Best Practice: Have a **separate git repository** for app code and terraform code

Why use Version Control?

- safekeeping
- history of changes
- team collaboration
- review infrastructure changes using merge requests





Executing commands on virtual servers

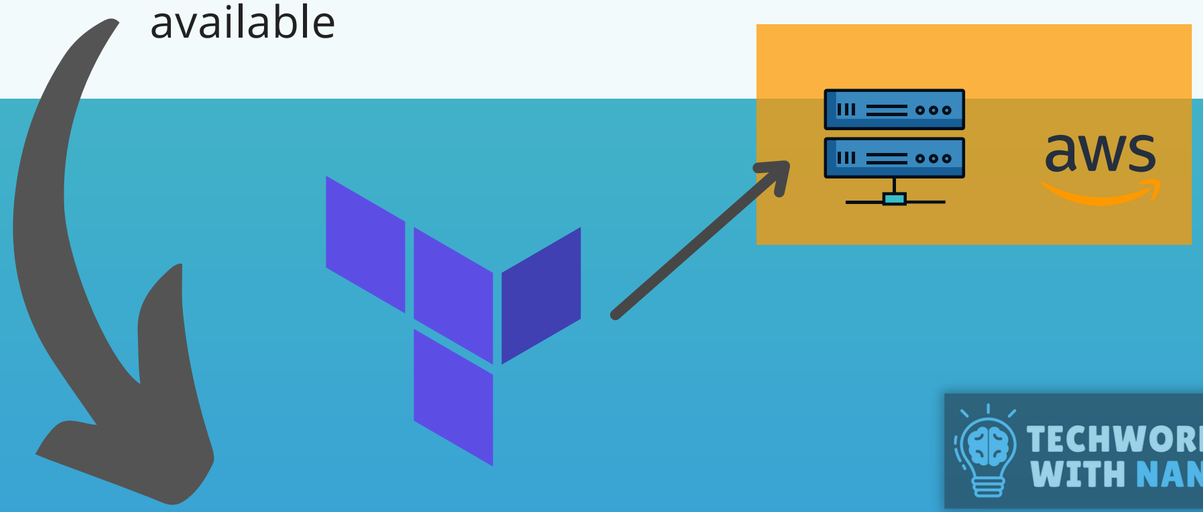
"user_data" attribute

- When deploying virtual machines, we often need to pass in initial data when launching the instance
- **Most cloud providers have a mechanism** to pass data to instances at the time of creation:

- Alibaba Cloud: `user_data` on `alicloud_instance` or `a`
- Amazon EC2: `user_data` or `user_data_base64` on `aws` `aws_launch_configuration` .
- Amazon Lightsail: `user_data` on `aws_lightsail_instan`
- Microsoft Azure: `custom_data` on `azurerm_virtual_mach`

Provisioners

- Another way are Provisioners
- Can be used to execute commands on the local machine or remote machine to prepare the infrastructure
- There are **different types of provisioners** available



Provisioners - 1

Different types of provisioners

"remote-exec" provisioner

- **Invokes script** on a remote resource after it's created
  - **inline**: list of commands
  - **script**: path

```
provisioner "remote-exec" {
    inline = [
        |
    ]
}
```

"file" provisioner

- **Copy files or directories** from local to newly created resource
  - **source:** source file or folder
  - **destination:** absolute path

```
provisioner "file" {
    source = "entry-script.sh"
}
```

"local-exec" provisioner

- Invokes a local executable after resource is created
- Locally, NOT on the created resource!

```
resource "aws_instance" "web" {
  # ...
  provisioner "local-exec" {
    command = "echo ${self.private_ip} >> priv
  }
```

Difference:

=> user_data: passing data to AWS

=> **remote-exec**: connect via SSH using TF

TECHWORLD WITH NANA

Provisioners - 2

Provisioners are NOT recommended:

- Use user_data if available
- ❌ Breaks idempotency concept
- ❌ TF doesn't know what you execute
- ❌ Breaks current-desired state comparison

Alternative to remote-exec:

- ✅ Use configuration management tools
- ✅ Once server provisioned, hand over to those tools, like Ansible

Provisioners are a Last Resort

> **Hands-on:** To learn about more declarative ways to handle provisioning actions, try the Provision Infrastructure Deployed with Terraform collection on HashiCorp Learn.

Terraform includes the concept of provisioners as a measure of pragmatism, knowing that there will always be certain behaviors that can't be directly represented in Terraform's declarative model.

However, they also add a considerable amount of complexity and uncertainty to Terraform usage. Firstly, Terraform cannot model the actions of provisioners as part of a plan because they can in principle take any action. Secondly, successful use of provisioners requires coordinating many more details than Terraform usage usually requires: direct network access to your servers, issuing Terraform credentials to log in, making sure that all of the necessary external software is installed, etc.

The following sections describe some situations which can be solved with provisioners in principle, but where better solutions are also available. We do not recommend using provisioners for any of the use-cases described in the following sections.

Even if your specific use-case is not described in the following sections, we still recommend attempting to solve it using other techniques first, and use provisioners only if there is no other option.

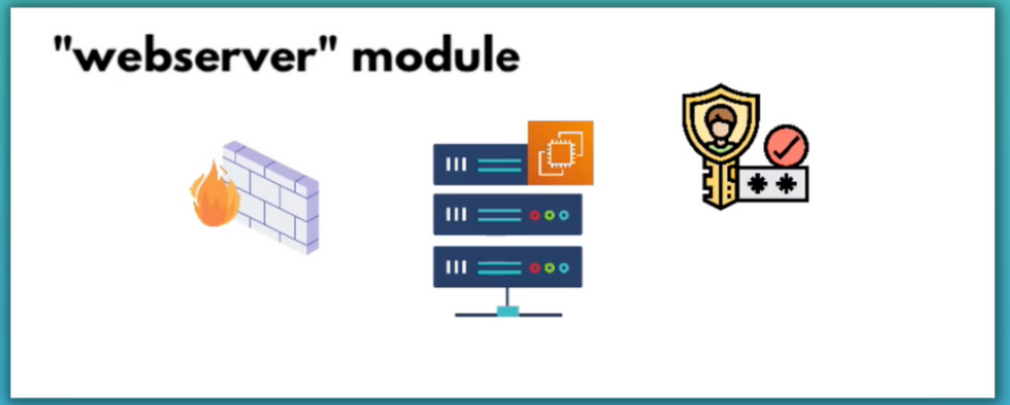
Modules - 1

A module is a container for multiple resources that are used together

Why modules?

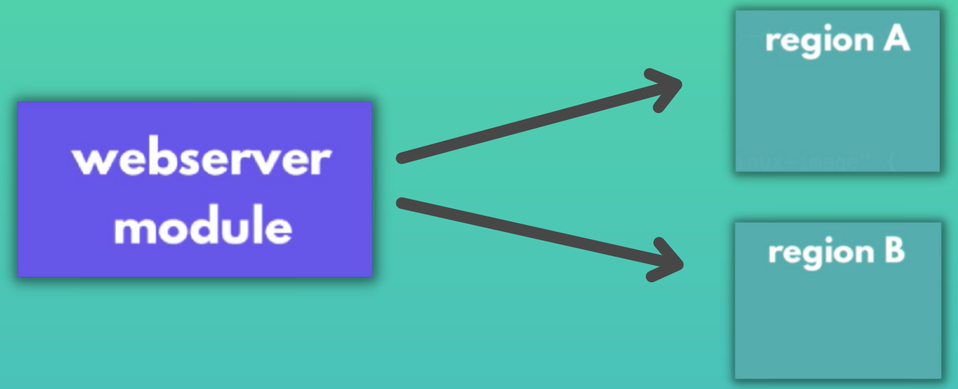
- Organize and group configurations
- Encapsulate into distinct logical components
- Reuse
- Without modules, complex configurations in a huge file with no overview

- An example could be a module for EC2 instance with configured networking and permissions:



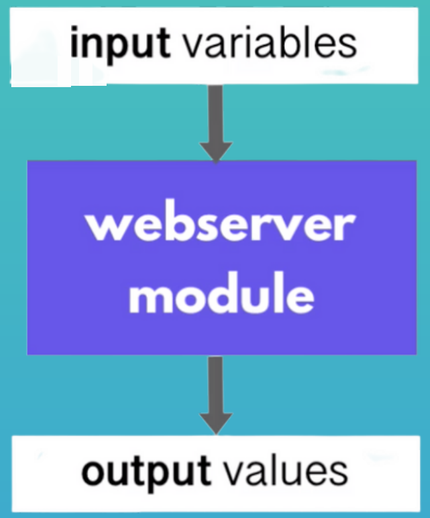
Modules - 2

- Then you can **easily reuse same configuration**, e.g. for different AWS regions:



- You can **customize** the configuration with variables

- And **expose** created resources or specific attributes with output values



Like Function Definition

input variables

= like function arguments

output values

= like function return values

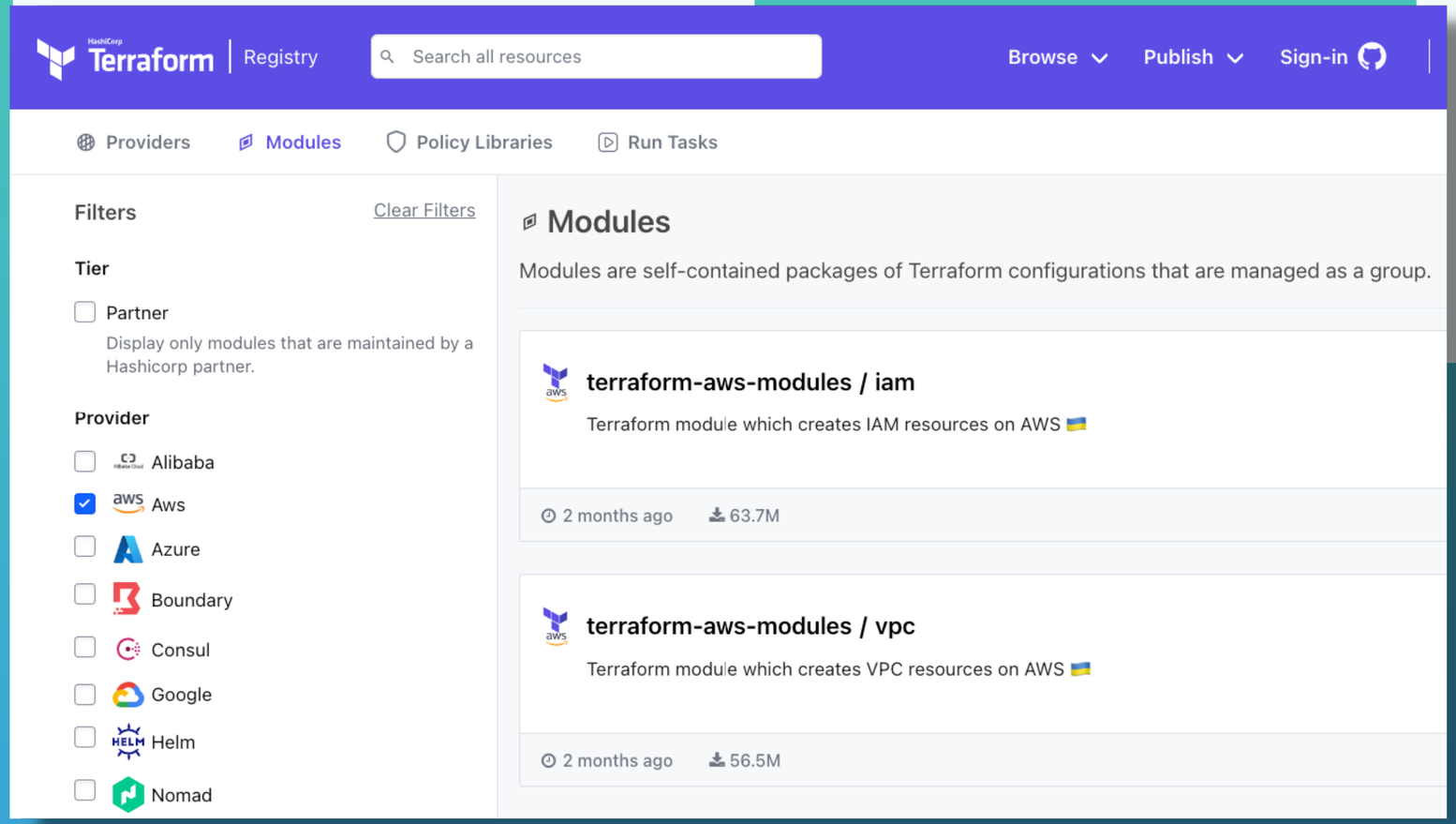
TECHWORLD WITH NANA

Modules - 3

Use existing modules

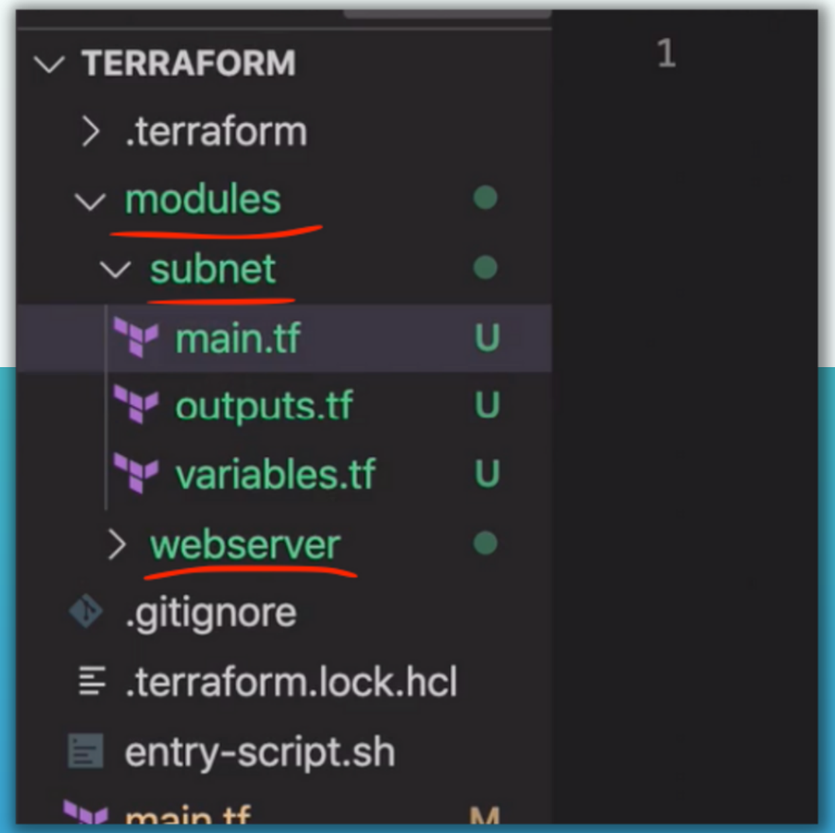
- There are many available on

  TF registry



Create your own modules

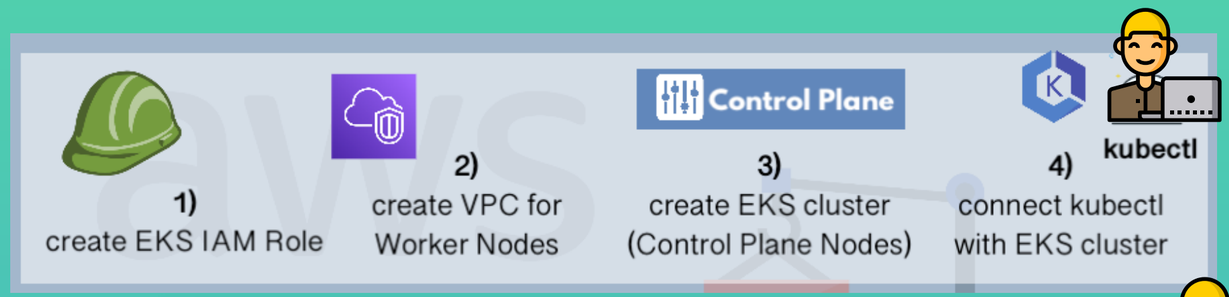
- To clean up your code
- Modules have a **pre-defined structure**:

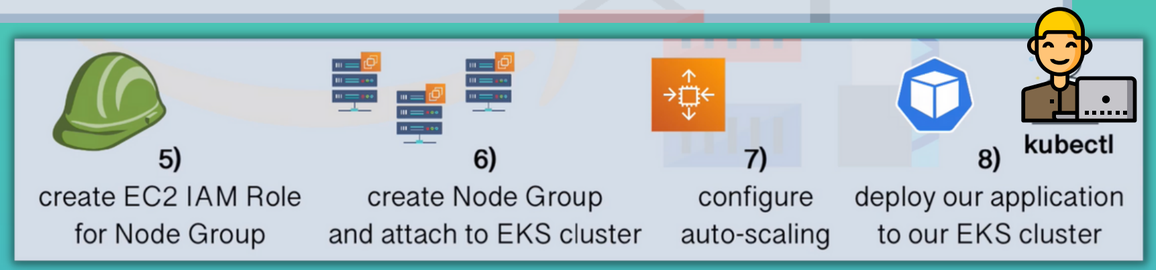


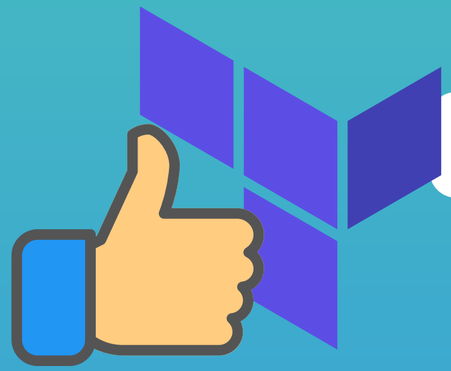
Automate provisioning EKS cluster with TF

Previously, we created EKS cluster **manually**

- Many components to create and configure
  - ❌ No version control (history)
  - ❌ No simple replication of infra possible
  - ❌ No simple clean up
  - ❌ Team collaboration difficult







Terraform is currently the **best way to provision an EKS cluster!**

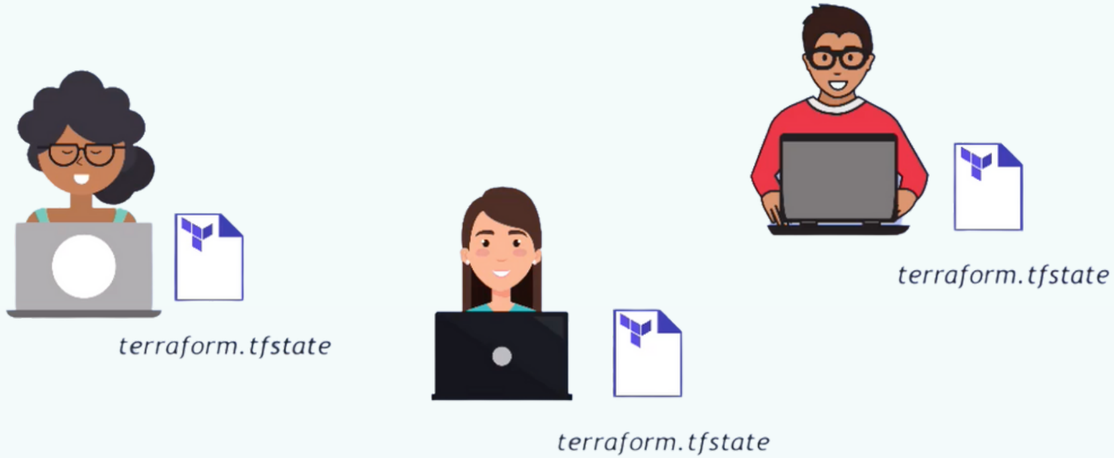
TECHWORLD WITH NANA

Remote State in Terraform - 1

With remote state, Terraform **writes the state data to a remote data store**

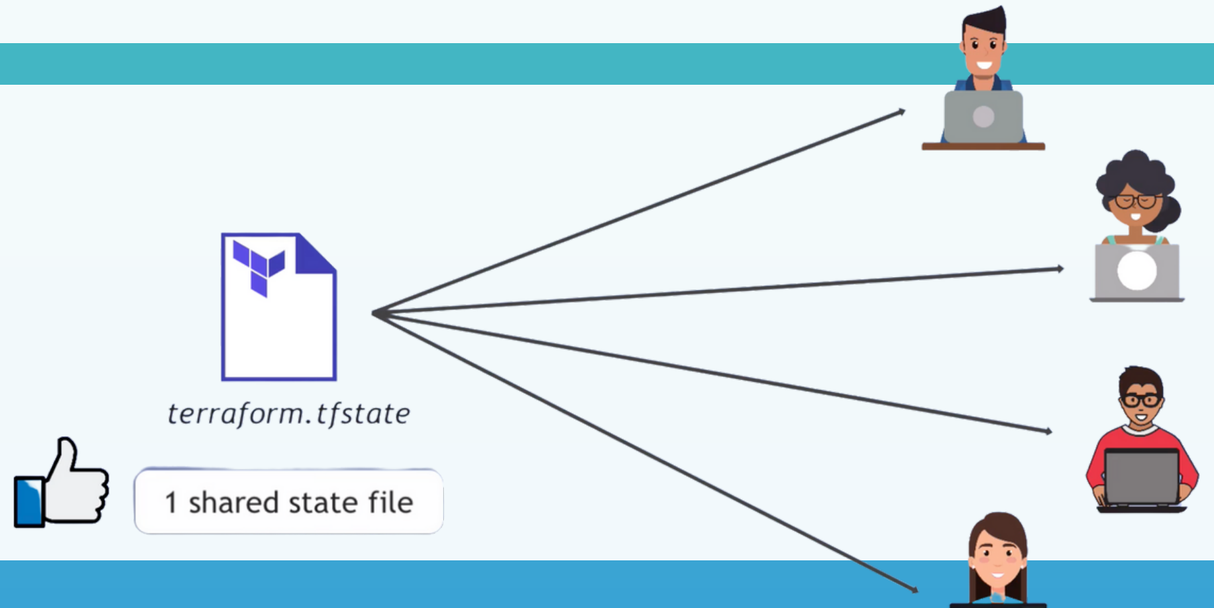
Own state file:

- ✗ Each user/CI server must **make sure** they always **have the latest state data** before running TF
- ✗ So team collaboration very difficult



With shared remote state:

- ✓ Data backup
- ✓ Can be shared
- ✓ Keep sensitive data off disk



Remote State in Terraform - 2

- Terraform supports storing state in...

Terraform Cloud

Azure Blob Storage

& more!

Amazon S3

Google Cloud Storage

Steps to configure remote state in AWS S3 bucket:

1) **Create bucket**

2) **Configure bucket as remote state location**

terraform > main.tf

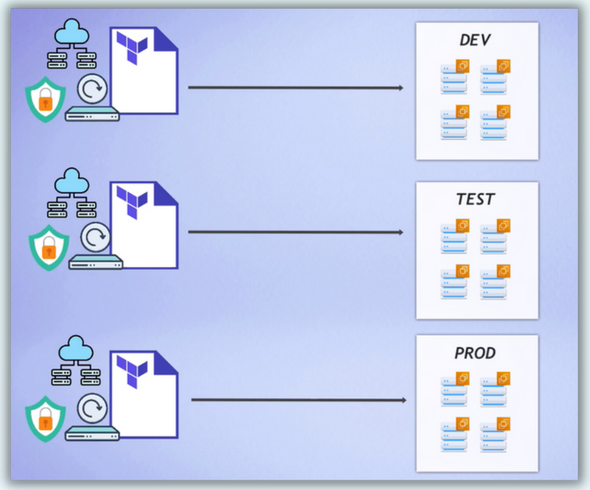
```
1  terraform {
2      required_version = ">= 0.12"
3      backend "s3" {
4          bucket = "myapp-bucket"
5          key = "myapp/state.tfstate"
6      }
7  }
8
9  provider "aws" {
10     region = var.region
11 }
```

Best Practices - 1

Best practice around Terraform state:

- Manipulate state only through TF commands
- Always set up a shared remote state instead of on your laptop or in Git
- Use state locking (locks state file until writing of state file is completed)
- Back up your state file and enable versioning (allows for state recovery)

- Use 1 state per environment



Best Practices - 2

Others:

- Host TF scripts in Git repository
- CI for TF code (review TF code, run automated tests)
- Apply TF ONLY through CD pipeline (instead of manually)
- Use _ (underscore) instead of - (dash) in all resource names, data source names, variable names, outputs etc.
- Only use lowercase letters and numbers
- Use a consistent structure and naming convention
- Don’t hardcode values as much as possible - pass as variables or use data sources to get a value