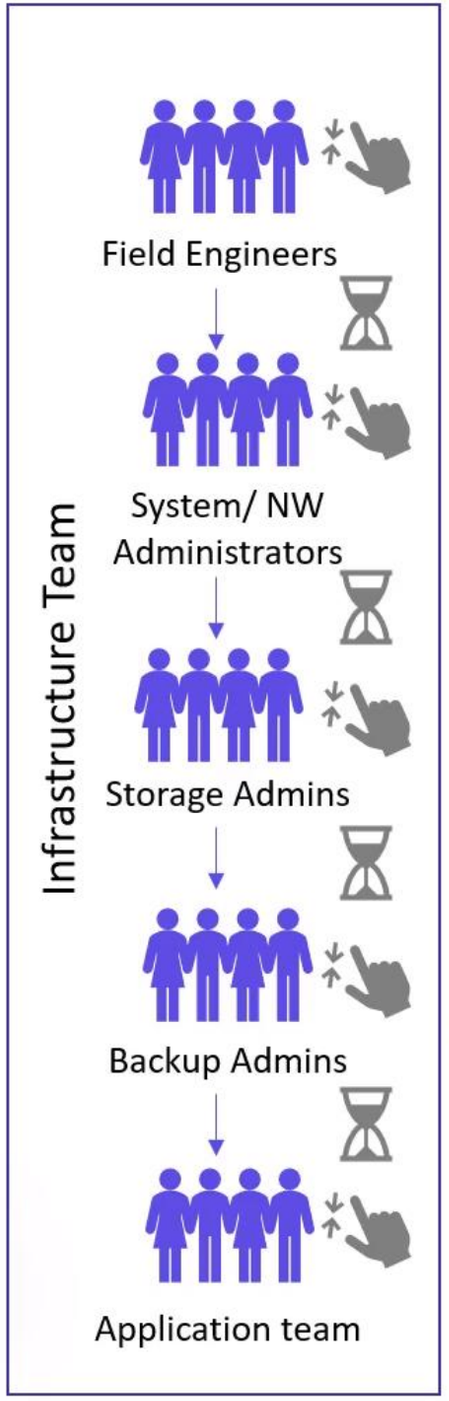
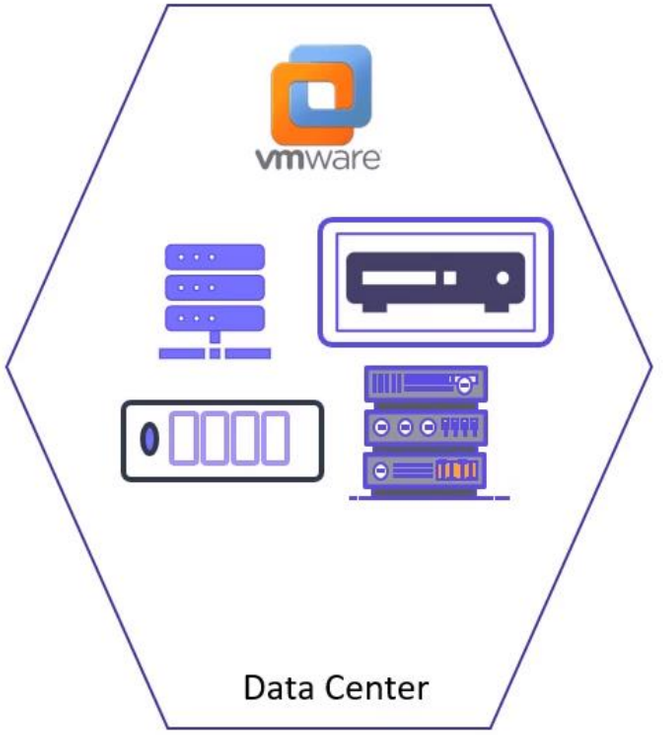
Infrastructure as Code with Terraform

Challenges…

- Slow Deployment
- Expensive
- Limited Automation
- Human Error
- Inconsistency
- Wasted Resources





IaaC

Imperative

ec2.sh

```
#!/bin/bash

IP_ADDRESS="10.2.2.1"

EC2_INSTANCE=$(ec2-run-instances --instance-type
t2.micro ami-0edab43b6fa892279)

INSTANCE=$(echo ${EC2_INSTANCE} | sed 's/*INSTANCE //'
| sed 's/ .*//')

# Wait for instance to be ready
while ! ec2-describe-instances $INSTANCE | grep -q
"running"
do
  echo Waiting for $INSTANCE is to be ready...
done

# Check if instance is not provisioned and exit
if [ ! $(ec2-describe-instances $INSTANCE | grep -q
"running") ]; then
  echo Instance $INSTANCE is stopped.
  exit
fi

ec2-associate-address $IP_ADDRESS -i $INSTANCE

echo Instance $INSTANCE was created successfully!!!
```

Declarative

main.tf

```
resource "aws_instance" "webserver" {
  ami           = "ami-0edab43b6fa892279"
  instance_type = "t2.micro"
}
```

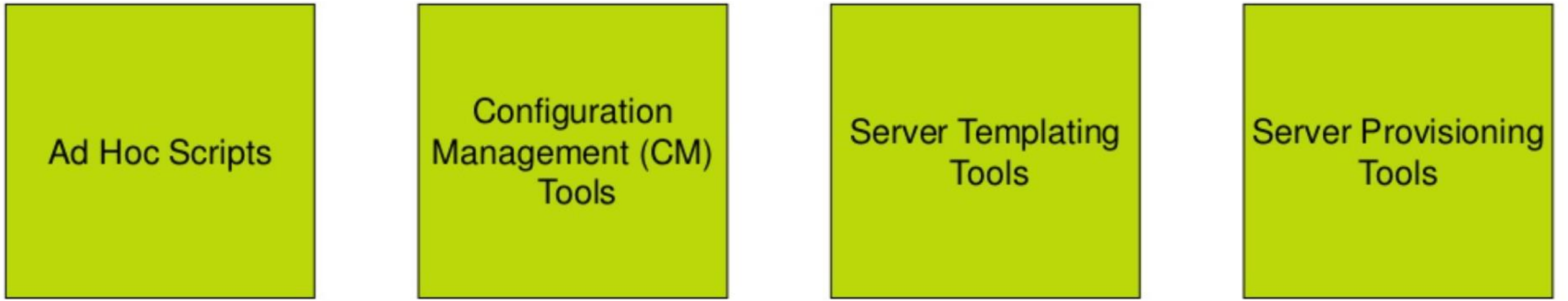
ec2.yaml

```
- amazon.aws.ec2:
    key_name: mykey
    instance_type: t2.micro
    image: ami-123456
    wait: yes
    group: webserver
    count: 3
    vpc_subnet_id: subnet-29e63245
    assign_public_ip: yes
```

What Is Infrastructure as (from) Code?

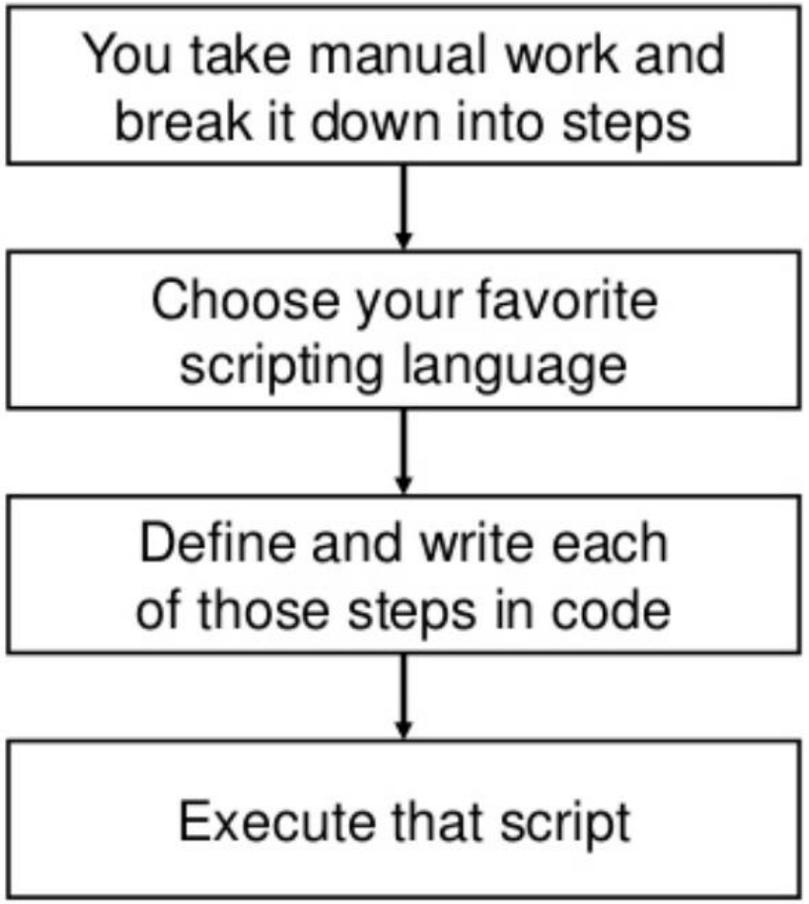
- Infrastructure as code (IaC) is an **approach** to infrastructure automation based on practices from software development.
- It emphasizes **consistent, repeatable** routines for **provisioning and changing** systems and their configuration.
- **Changes are made to definitions** and then rolled out to systems through **unattended processes** that include thorough **validation**.

Tooling Categories...



Ad-Hoc Scripts..

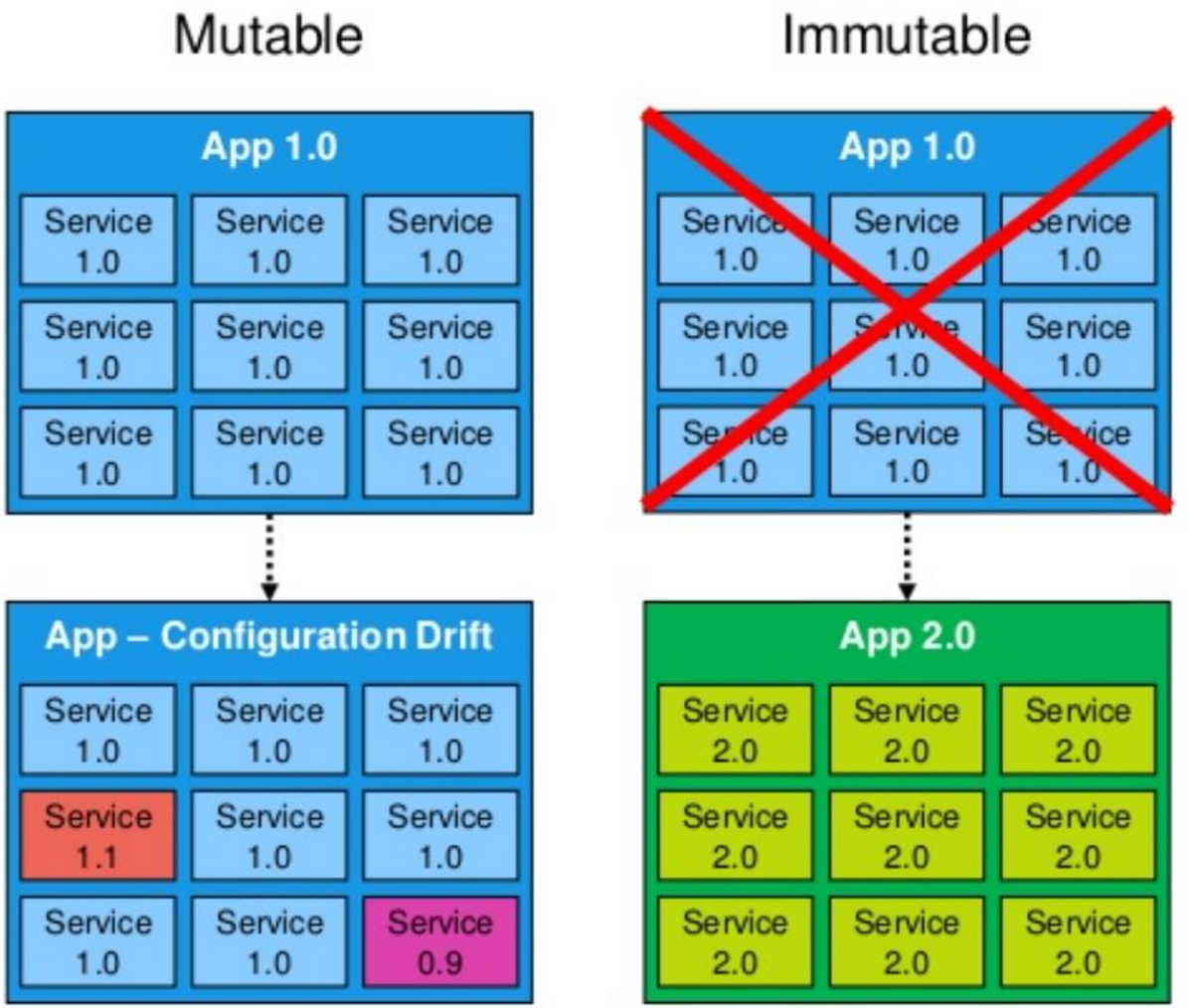
- The most straightforward approach to automating anything is to write an ad hoc script.




```
# Update the apt-get cache
sudo apt-get update

# Install PHP
sudo apt-get install -y php

# Install Apache
sudo apt-get install -y apache2

# Start Apache
sudo service apache2 start
```

Mutable & Immutable Infrastructure

- The Pets and Cattle debate.
- One approach is not necessary better then the other, it depends on your use-case.
- With the mutable approach, the team needs to be aware of the infrastructure “history”.
- Generally speaking, the immutable approach is better for stateless applications.
- Immutable drives no deviation and no changes. It is what it is.



Infrastructure as Code

docker

ANSIBLE

HashiCorp Terraform

CloudFormation

HashiCorp Vagrant

HashiCorp Packer

HashiCorp Packer

SALTSTACK

puppet

Types of IAC Tools

Configuration Management

Install and Manage SW
Version control
Idempotent / Mutable

Server Templating

Pre installed SW with Dependencies
VM or docker images
Immutable architecture

Provisioning Tools

Deploy Immutable infr resrce
Servers, DBs, NW components
Multi providers

HashiCorp
Terraform

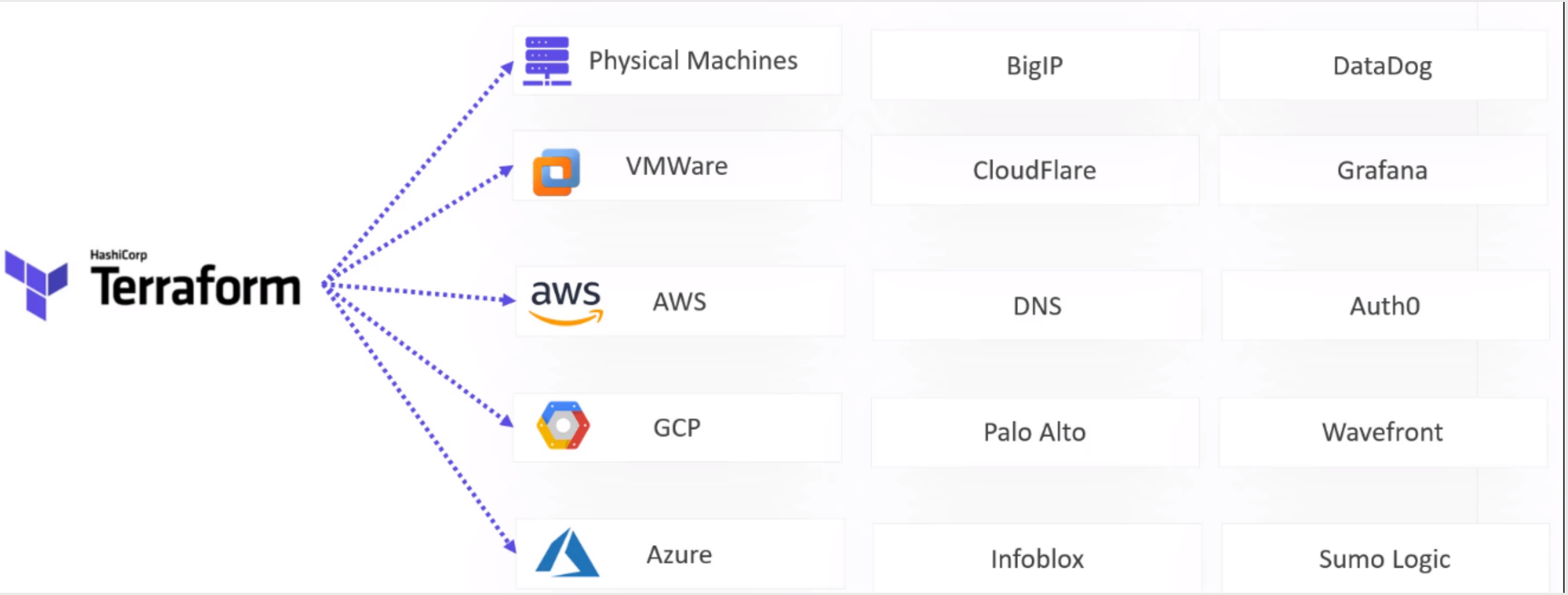
Terraform is an open-source infrastructure as code software tool that enables you to safely and predictably create, change, and improve infrastructure.

Terraform is an open-source infrastructure as code software tool that provides a consistent CLI workflow to manage hundreds of cloud services. Terraform codifies cloud APIs into declarative configuration files

https://www.terraform.io/

HashiCorp was founded by Mitchell Hashimoto and Armon Dadgar in 2012 with the goal of revolutionizing datacenter management: application development, delivery, and maintenance.

Why terraform



Installing Terraform

Download - Download Terraform - Terraform by HashiCorp

https://learn.hashicorp.com/tutorials/terraform/install-cli

 # Install yum-config-manager to manage your repositories

sudo yum install -y yum-utils

#Use yum-config-manager to add the official HashiCorp Linux repository

sudo yum-config-manager --add-repo https://rpm.releases.hashicorp.com/AmazonLinux/hashicorp.repo

 # Install terraform

sudo yum -y install terraform

terraform -install-autocomplete

HCL

main.tf

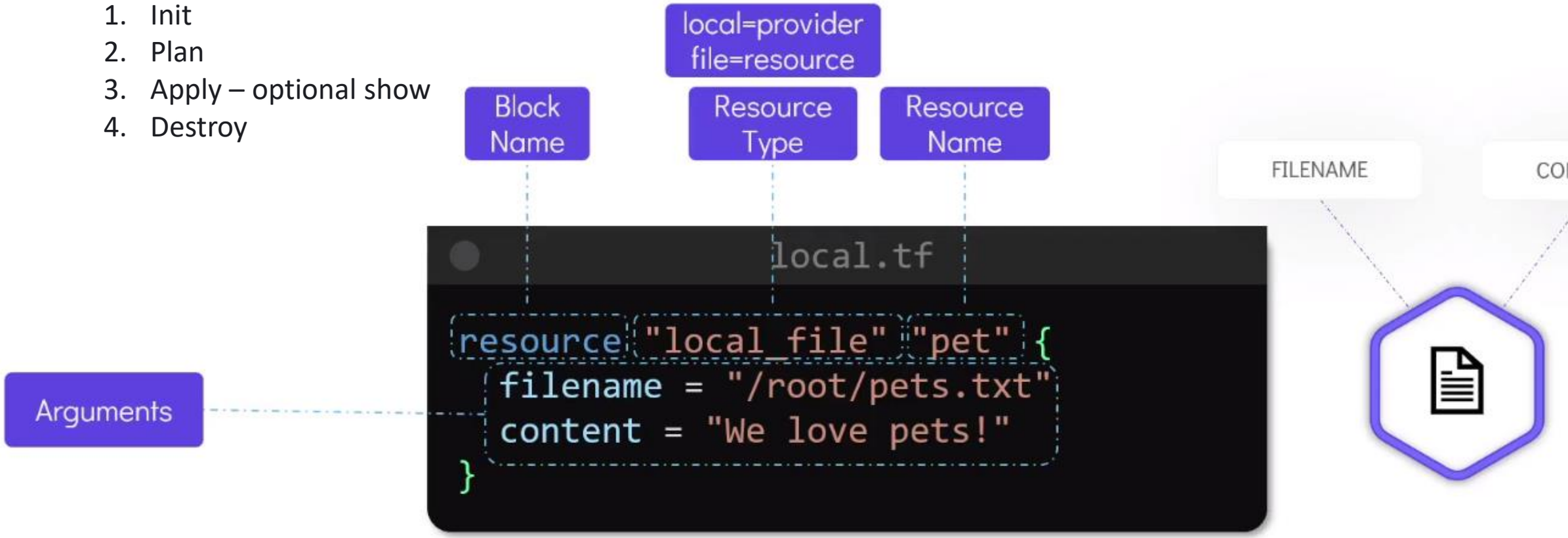
```
resource "aws_instance" "webserver" {
  ami           = "ami-0edab43b6fa892279"
  instance_type = "t2.micro"
}

resource "aws_s3_bucket" "finance" {
    bucket = "finanace-21092020"
    tags   = {
        Description = "Finance and Payroll"
    }
}
 resource "aws_iam_user" "admin-user" {
     name = "lucy"
     tags = {
       Description = "Team Leader"
     }
 }
```

```
# Configure the Microsoft Azure Provider
provider "azurerm" {
  features {}
}

# Create a resource group
resource "azurerm_resource_group" "example" {
  name     = "example-resources"
  location = "West Europe"
}

# Create a virtual network within the resource group
resource "azurerm_virtual_network" "example" {
  name                = "example-network"
  resource_group_name = azurerm_resource_group.example.name
  location            = azurerm_resource_group.example.location
  address_space       = ["10.0.0.0/16"]
}
```

.tf

Lets get in to action

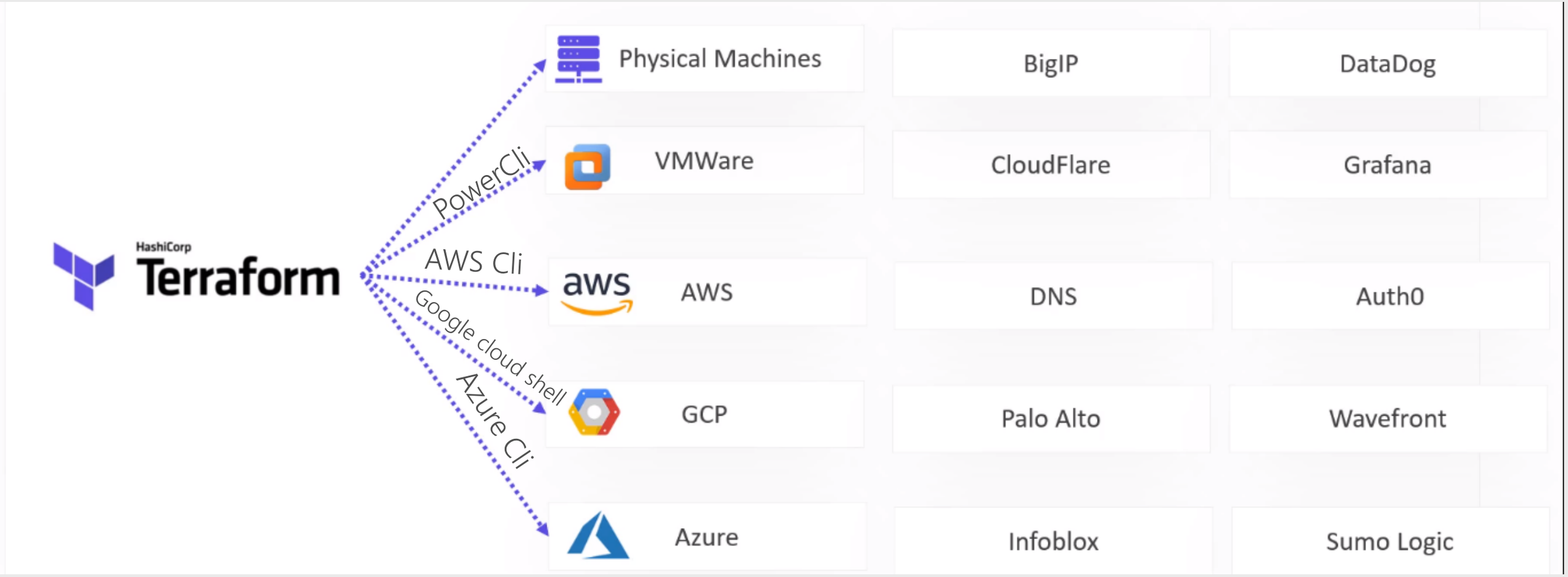
1. Create a folder
2. Create a new file ending with .tf extension
3. Terraform commands
    1. Init
    2. Plan
    3. Apply – optional show
    4. Destroy



[local_file | Resources | hashicorp/local | Terraform Registry]


Init
Plan
Apply
Destroy

Why terraform



HCL Resource…

HCL – Declarative Language

aws.tf

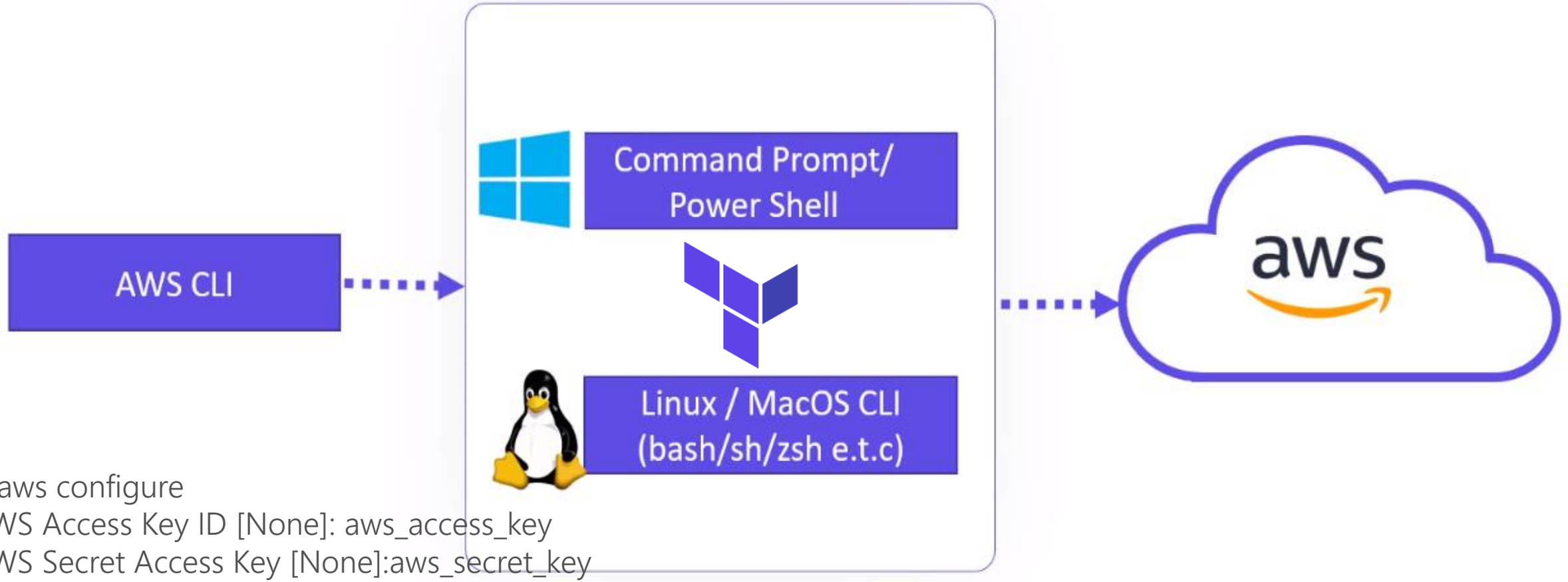
```
resource "aws_instance" "webserver" {
    ami = "ami-0c2f25c1f66a1ff4d"
    instance_type = "t2.micro"
}
```

The Terraform language or (HCL) is Terraform's primary user interface. In every edition of Terraform, a configuration written in the Terraform language is always at the heart of the workflow.


Terraform
TERRAFORM
PROVIDER
Target API

To manage AWS - Pre-Req

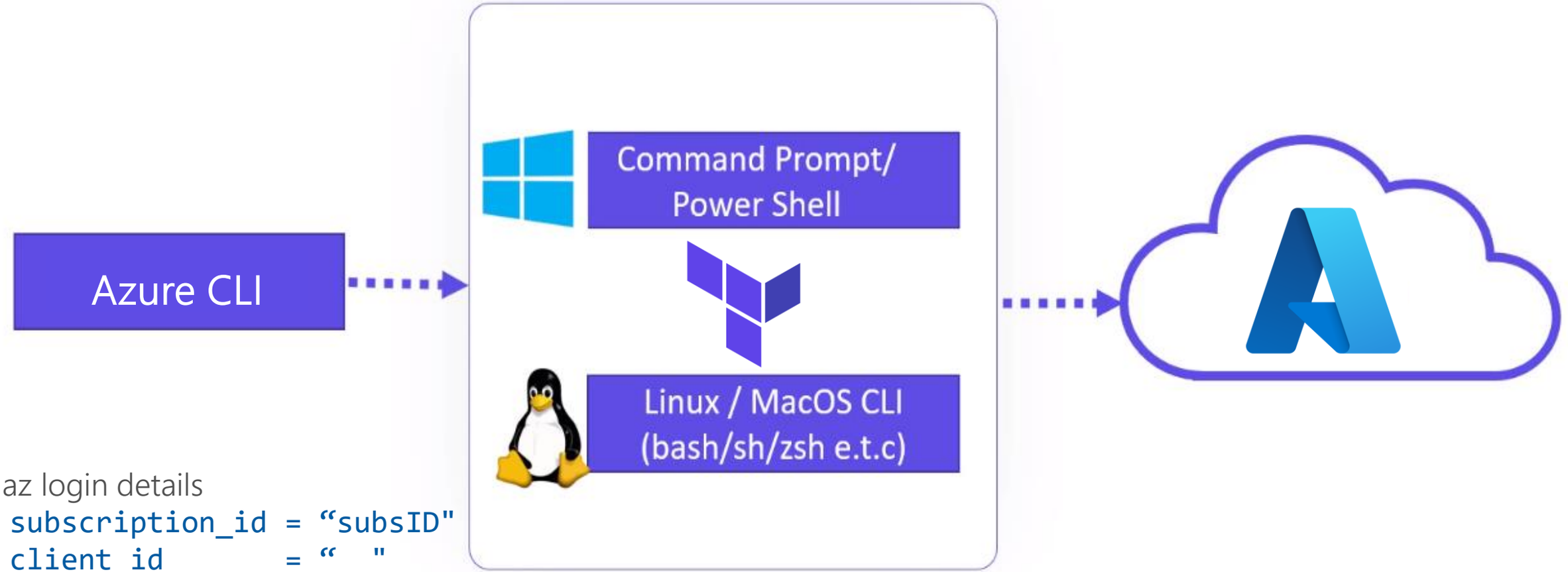
https://docs.aws.amazon.com/cli/latest/userguide/install-cliv2-linux.html




```
# aws configure
AWS Access Key ID [None]: aws_access_key
AWS Secret Access Key [None]:aws_secret_key
Default region name [None]: us-west-2
Default output format [None]: ENTER
```

To manage Azure - Pre-Req

https://learn.microsoft.com/en-us/cli/azure/install-azure-cli-linux?pivots=apt



az login details

```
subscription_id = “subsID"
client_id       = “  "
client_secret   = “  "
tenant_id       = “ "
```

How...

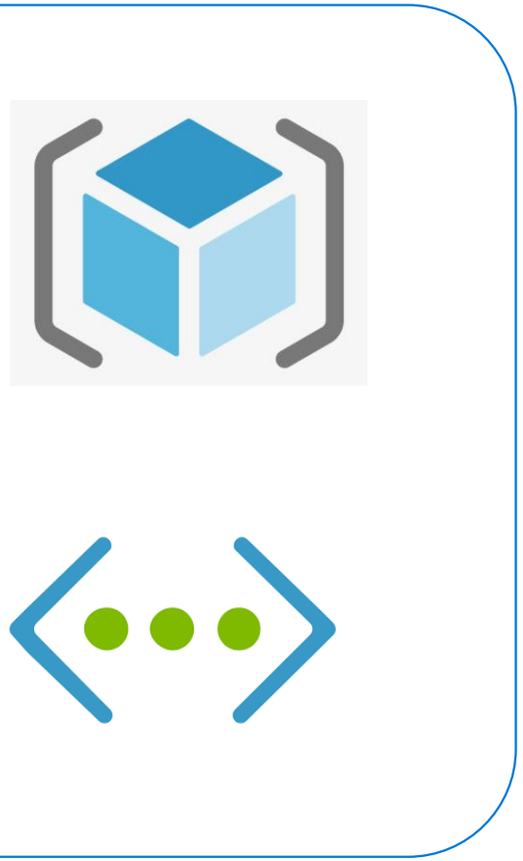
```
# Configure the Microsoft Azure Provider
provider "azurerm" {
  features {}
}

# Create a resource group
resource "azurerm_resource_group" "example" {
  name     = "example-resources"
  location = "West Europe"
}

# Create a virtual network within the resource group
resource "azurerm_virtual_network" "example" {
  name                = "example-network"
  resource_group_name =
azurerm_resource_group.example.name
  location            =
azurerm_resource_group.example.location
  address_space       = ["10.0.0.0/16"]
}
```

terraform

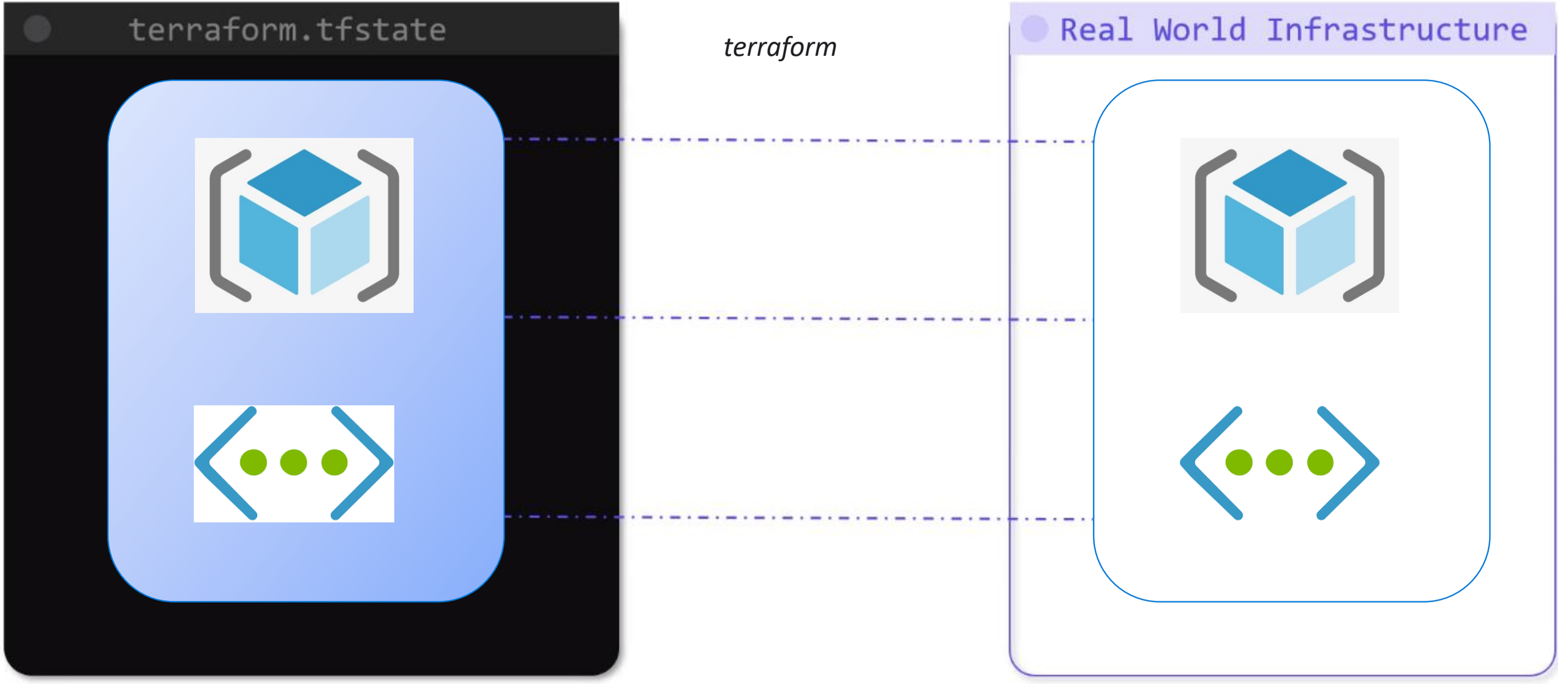
Init

Plan

Apply



Terraform State...



How...

main.tf

```
resource "aws_instance" "webserver" {
  ami           = "ami-0edab43b6fa892279"
  instance_type = "t2.micro"
}

resource "aws_s3_bucket" "finance" {
    bucket = "finanace-21092020"
    tags   = {
        Description = "Finance and Payroll"
    }
}
 resource "aws_iam_user" "admin-user" {
     name = "lucy"
     tags = {
       Description = "Team Leader"
     }
 }
```

terraform

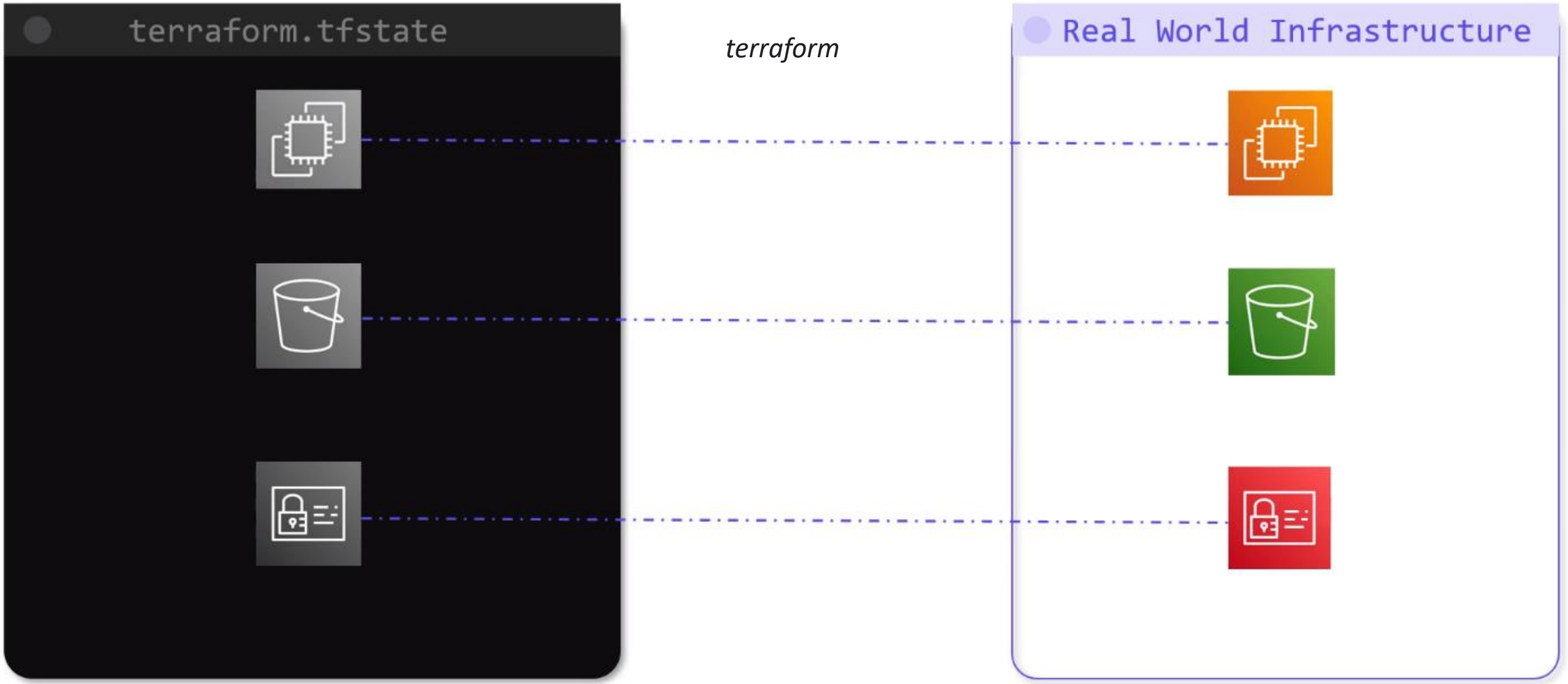
Init

Plan

Apply

Real World Infrastructure

Terraform State...



Terraform Language / HCL Syntax

https://www.terraform.io/docs/configuration/index.html
https://www.terraform.io/docs/configuration/syntax.html

```
<BLOCK TYPE> "<BLOCK LABEL>" "<BLOCK LABEL>" {
  # Block body
  <IDENTIFIER> = <EXPRESSION> # Argument
}
```

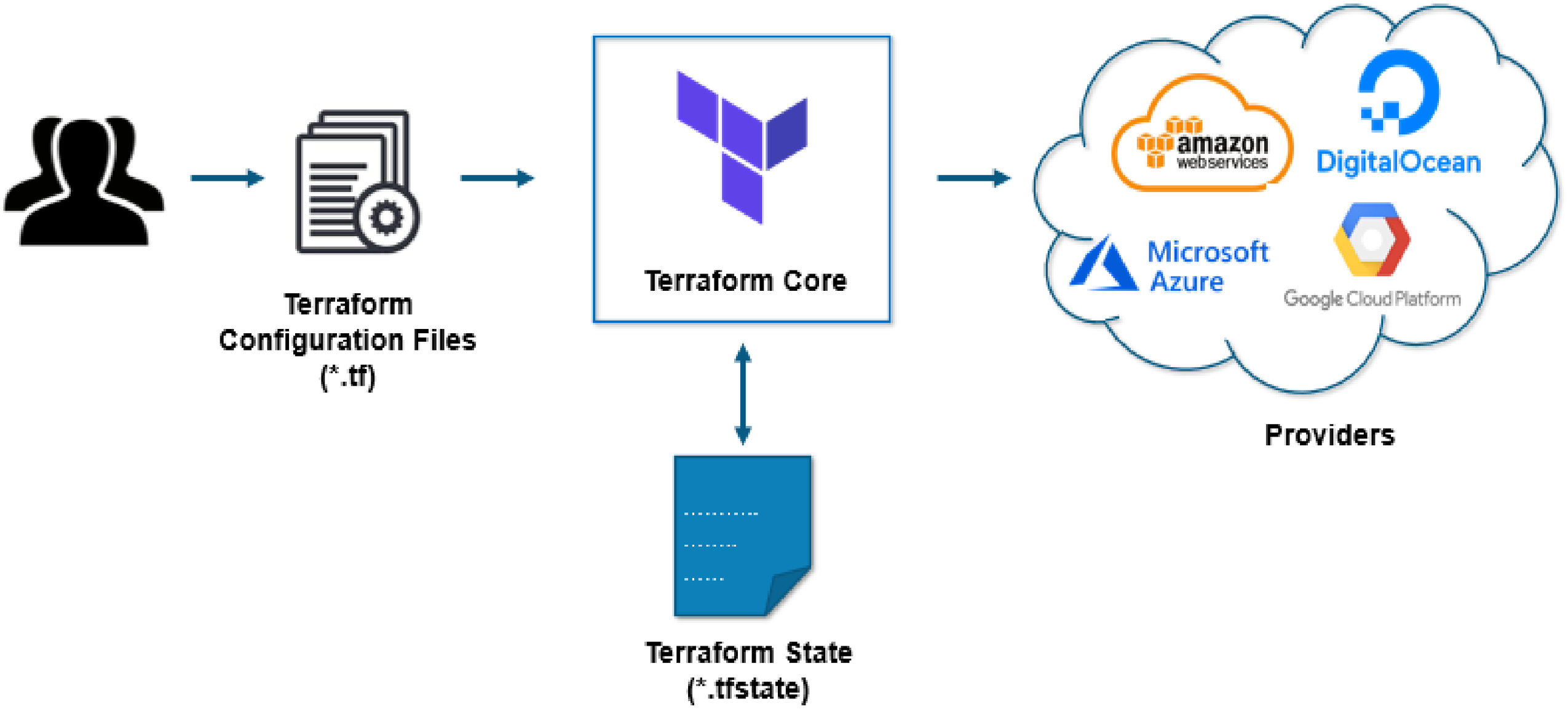
For example:

```
resource "aws_vpc" "main" {
  cidr_block = var.base_cidr_block
}
```

```
resource "local_file" "pet"  {
filename = "filename_path"
content = "we love pets"
}
```

Blocks are containers for other content and usually represent the configuration of some kind of object, like a resource.

Within the block body (between { and }) are the configuration arguments for the resource itself. Most arguments in this section depend on the resource type


Terraform
Configuration Files
(*.tf)
Terraform Core
Terraform State
(*.tfstate)
amazon
webservices
DigitalOcean
Microsoft
Azure
Google Cloud Platform
Providers

Providers…

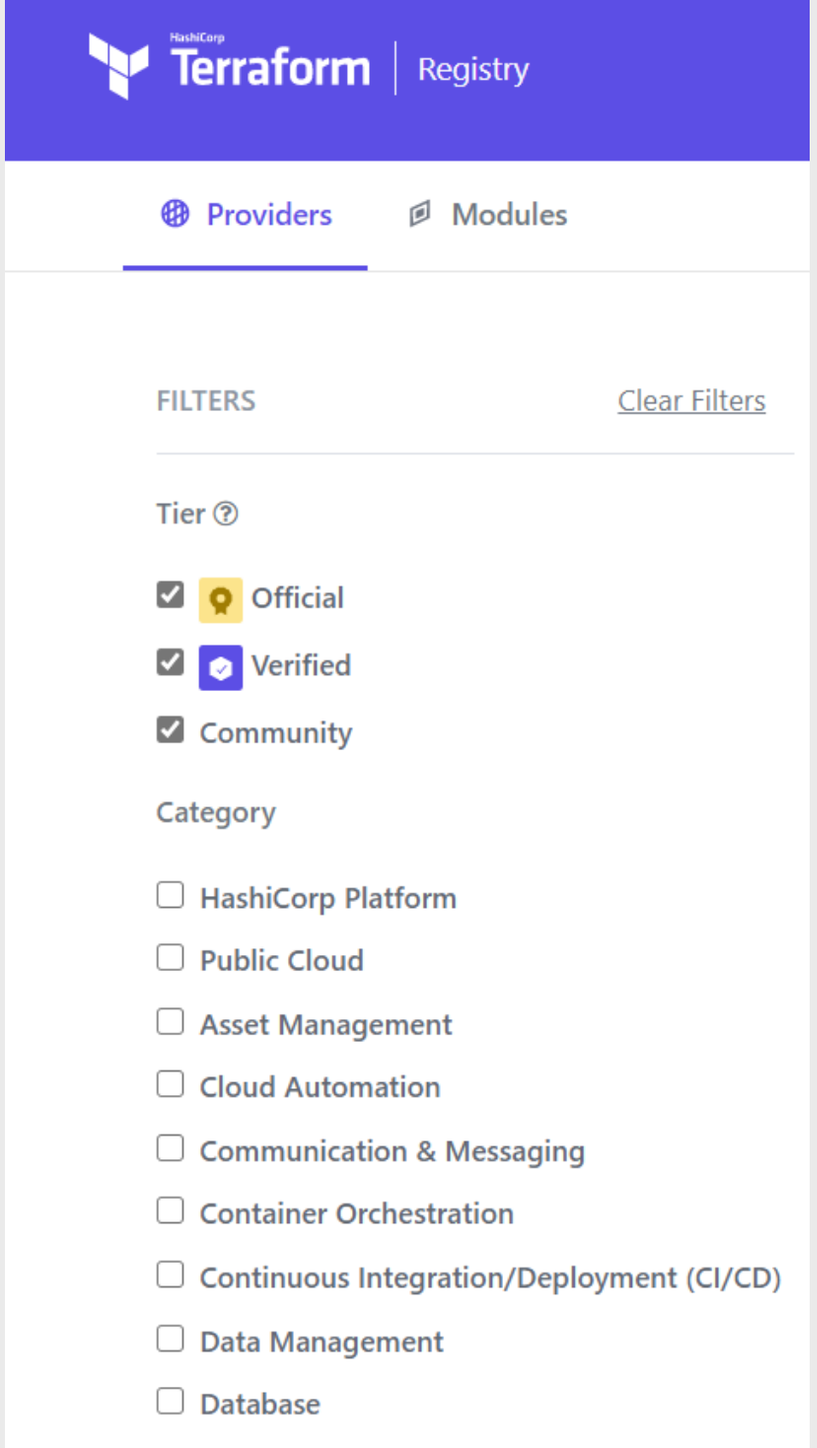
- plugins called "providers" to interact with remote systems.
  - Terraform relies on plugins called "providers" to interact with remote systems. Terraform configurations must declare which providers they require so that Terraform can install and use them. Additionally, some providers require configuration (like endpoint URLs or cloud regions) before they can be used.
- What providers do?
  - Each provider adds a set of resource types and/or data sources that Terraform can manage.
  - Every resource type is implemented by a provider; without providers, Terraform can't manage any kind of infrastructure.
  - Most providers configure a specific infrastructure platform (either cloud or self-hosted). Providers can also offer local utilities for tasks like generating random numbers for unique resource names.
- **Where Providers Come From**
  - Providers are distributed separately from Terraform itself, and each provider has its own release cadence and version numbers.
  - The Terraform Registry is the main directory of publicly available Terraform providers, and hosts providers for most major infrastructure platforms.

Providers…

Official

Verified

Community



[Browse Providers | Terraform Registry]

HOL

Create resource in aws

```
provider "aws" {
region = "ap-southeast-1"
access_key = ""
secret_key = "“

resource "aws_instance" "ec2" { # I called name "ec2", you can change your own name
ami = "ami-0dad20bd1b9c8c004" # Image: Ubuntu Server 18.04 LTS
instance_type = "t2.micro" # VM Spec
security_groups = ["${aws_security_group.allow_ssh.name}"]
key_name = "aws-existingkey"
}

resource "aws_security_group" "allow_ssh" {
name = "allow ssh"
....[truncated]
```

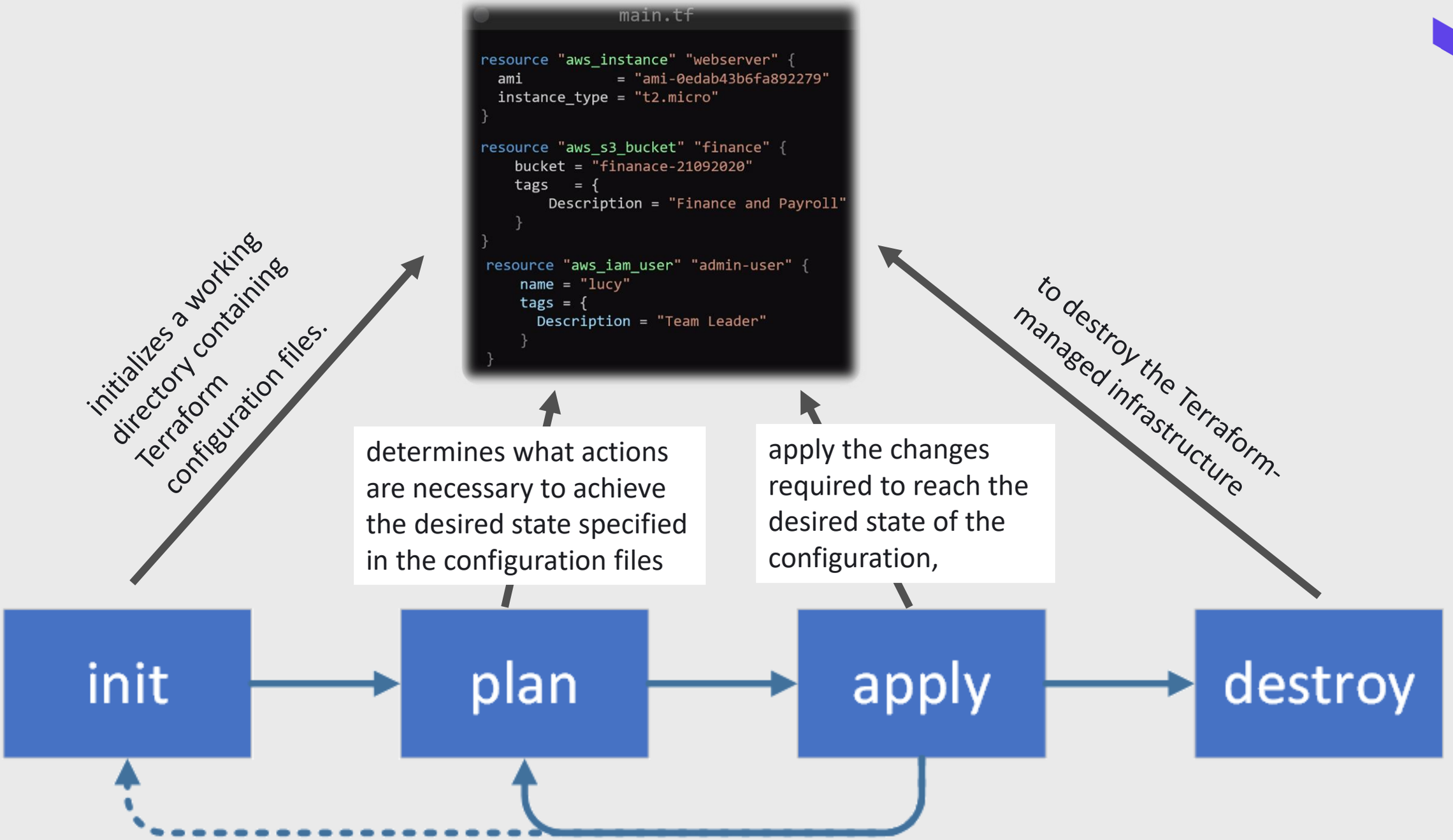
Step Guide – AWS EC2 instance creation…

1. AWS cli installed system
2. Create AWS credential – programmatic access - portal
3. Add terraform block in main.tf file
4. Add provider block in main.tf file with authentication
5. Add resource block with required attributes

https://registry.terraform.io/providers/hashicorp/aws/latest/docs

. init

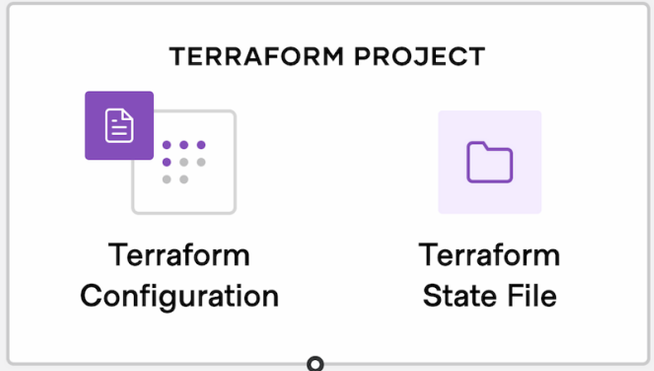
```
$ terraform init
Initializing the backend...

Initializing provider plugins...
- Finding latest version of hashicorp/local...
- Installing hashicorp/local v2.0.0...
- Installed hashicorp/local v2.0.0 (signed by HashiCorp)

The following providers do not have any version constraints in
configuration,
so the latest version was installed.

To prevent automatic upgrades to new major versions that may
contain breaking
changes, we recommend adding version constraints in a
required_providers block
in your configuration, with the constraint strings suggested
below.

* hashicorp/local: version = "~> 2.0.0"

Terraform has been successfully initialized!
```



[Terraform CLI Documentation - Terraform by HashiCorp]

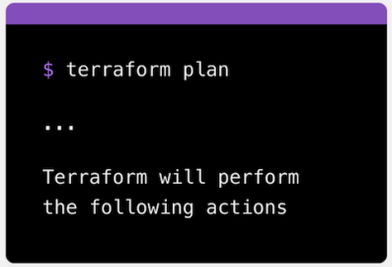
Write

Define infrastructure in configuration files



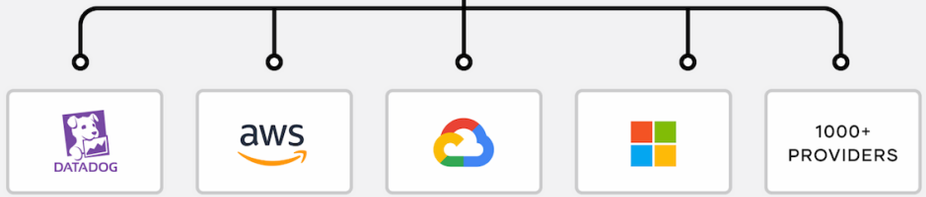
Plan

Review the changes Terraform will make to your infrastructure



Apply

Terraform provisions your infrastructure and updates the state file.



Variables..

- Input variables
  - used to define values that configure your infrastructure. These values can be used again and again without having to remember their every occurrence in the event it needs to be updated.
- Output variables
  - used to get information about the infrastructure after deployment. These can be useful for passing on information such as IP addresses for connecting to the server.

Input variables

Input variables are usually defined by stating a name, type and a default value.

The type and default values are not strictly necessary.

Terraform can deduct the type of the variable from the default or input value.

variable "variable_name" { }

```
variable “region" { }

# Configure the AWS Provider
provider "aws" {
  region = "${var.region}"
[...] turncated
```

terraform apply #will prompt region during execution

```
variable “region" {
   default = “us-east-1”
}
# Configure the AWS Provider
provider "aws" {
  region = "${var.region}"
[...] turncated
```

terraform apply #without supplying variable

terraform apply -var region=“us-east-2”
#with variable supplied…

[Input Variables - Configuration Language - Terraform by HashiCorp]

Using variables in main.tf

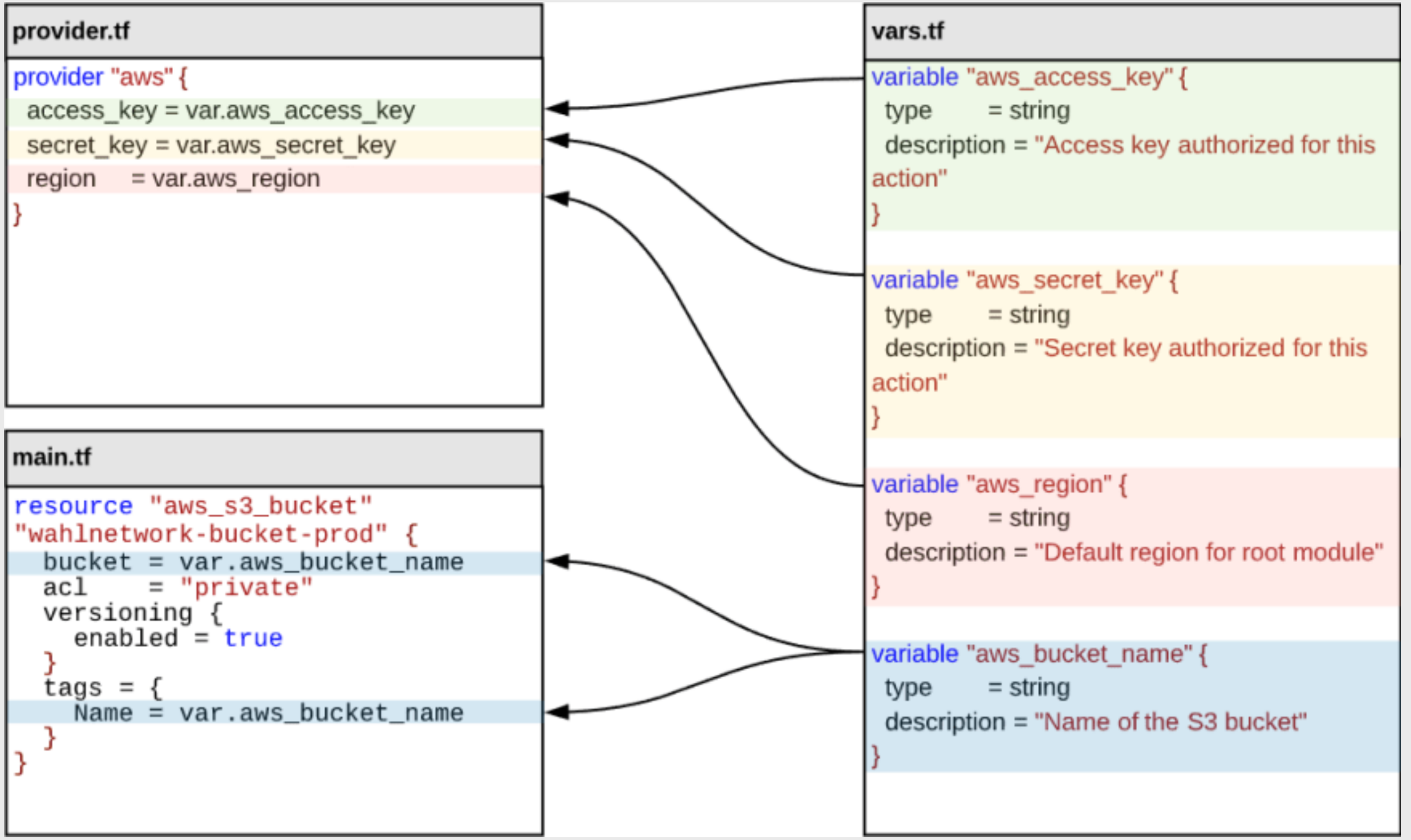
main.tf

```
resource "local_file" "pet" {
    filename = var.filename
    content = var.content

}

resource "random_pet" "my-pet" {
    prefix = var.prefix
    separator = var.separator
    length = var.length
}
```

variables.tf

```
variable "filename" {
        default = "/root/pets.txt"
}
variable "content" {
        default = "We love pets!"
}
variable "prefix" {
        default = "Mrs"
}
variable "separator" {
        default = "."
}
variable "length" {
        default = "1"
}
```

In action..aws


provider.tf
provider "aws" {
access_key = var.aws_access_key
secret_key = var.aws_secret_key
region = var.aws_region
}
main.tf
resource "aws_s3_bucket"
"wahlnetwork-bucket-prod" {
bucket = var.aws_bucket_name
acl = "private"
versioning {
enabled = true
}
tags = {
Name = var.aws_bucket_name
}
}
vars.tf
variable "aws_access_key" {
type = string
description = "Access key authorized for this action"
}
variable "aws_secret_key" {
type = string
description = "Secret key authorized for this action"
}
variable "aws_region" {
type = string
description = "Default region for root module"
}
variable "aws_bucket_name" {
type = string
description = "Name of the S3 bucket"
}

Output variables

provide a convenient way to get useful information about your infrastructure.

As you might have noticed, much of the resource details are calculated at deployment and only become available afterwards.

Using output variables you can extract any server-specific values including the calculated details.

```
terraform {…}
provider "aws" {…}
resource "aws_instance" "webserver" {…}
resource "local_file" “webserver_ip_details" {
  filename = "webserver_details.txt"
  content = <<-EOF
    public ip of webserver is ${aws_instance.webserver.public_ip}
    private ${aws_instance.webserver.private_ip}
    EOF
```

Config files…

File Name	Purpose
Main.tf	Contains resource definition. call modules, locals and data-sources to create all resources
Variables.tf	Contains variables declarations
Outputs.tf	Contains outputs from resources
Provider.tf	Contains provider definition

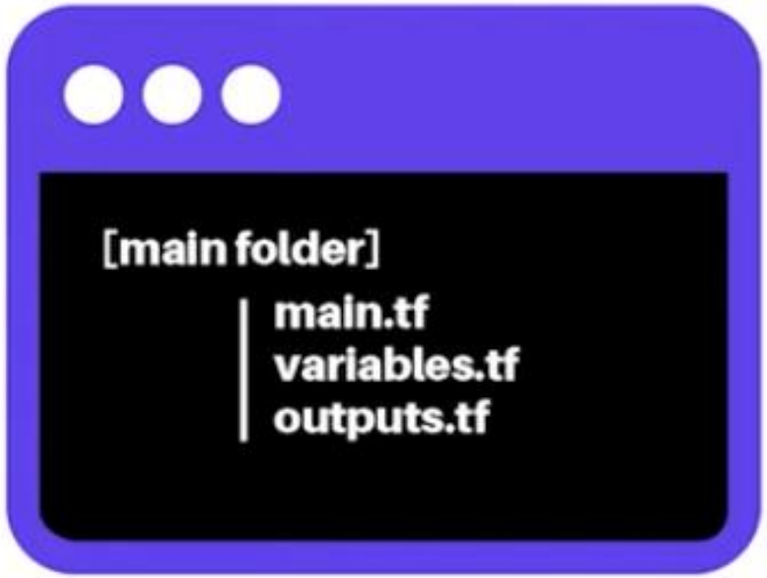
```
resource "aws_instance" "web" {
  # ...

  provisioner "local-exec" {
    command = "echo The server's IP address is ${self.private_ip}"
  }
}
```

```
resource   “aws_instance” “webserver” {
     ami = “ami-.....
     instance_type = “t2.micro”
provisioner "remote-exec" {
    inline = [
      “yum -y install httpd”,
      “systemctl enable httpd”,
      “systemctl start httpd”
    ]
```

Modules….

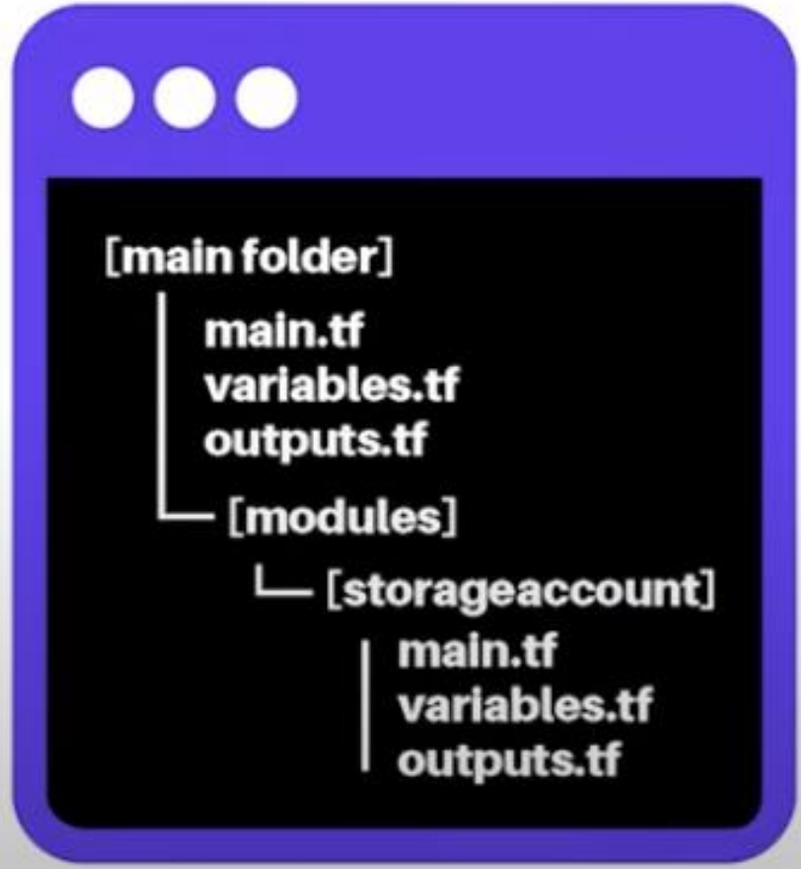
[Standard Structure]

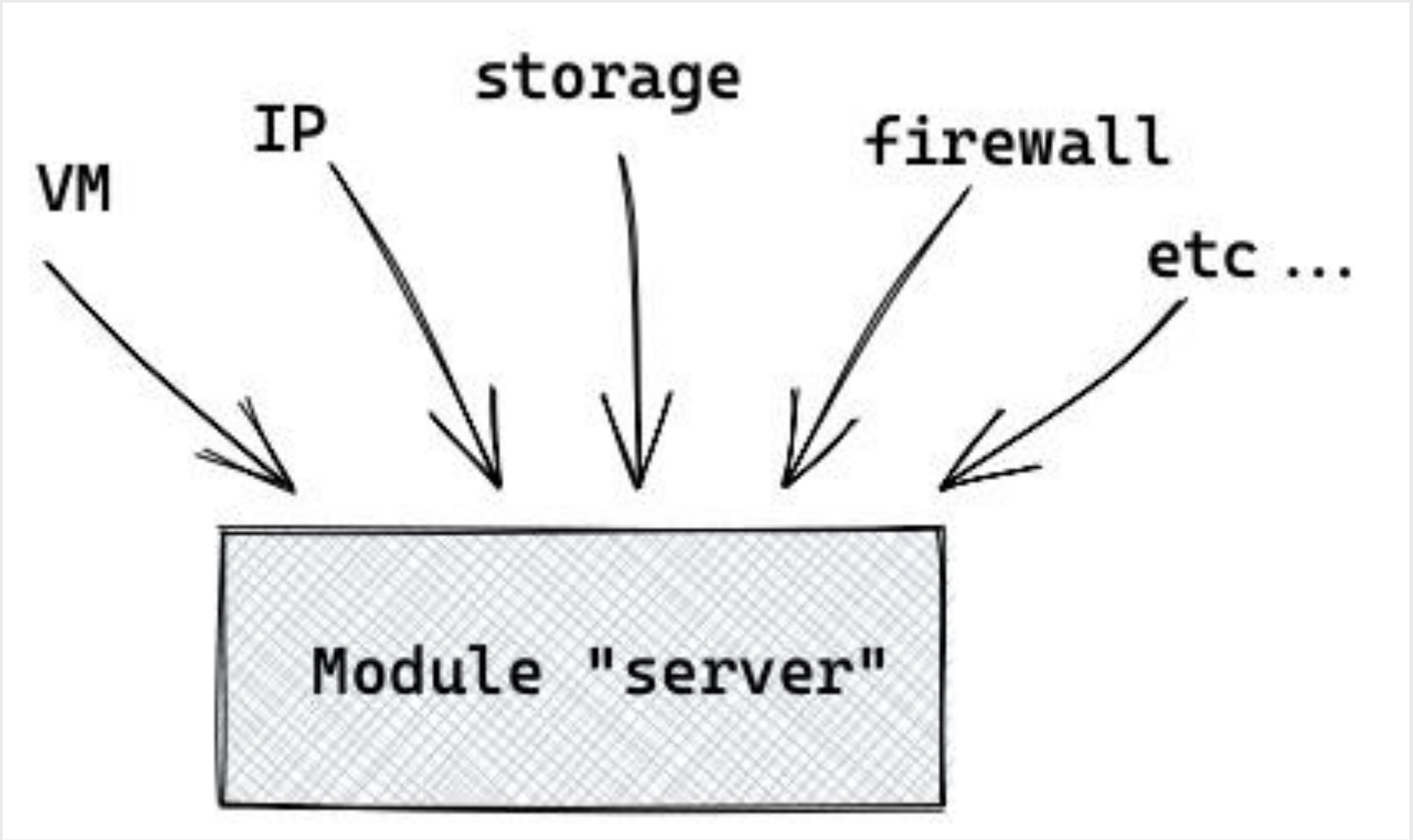
This is considered the root module and this is how you normally deploy infrastructure.

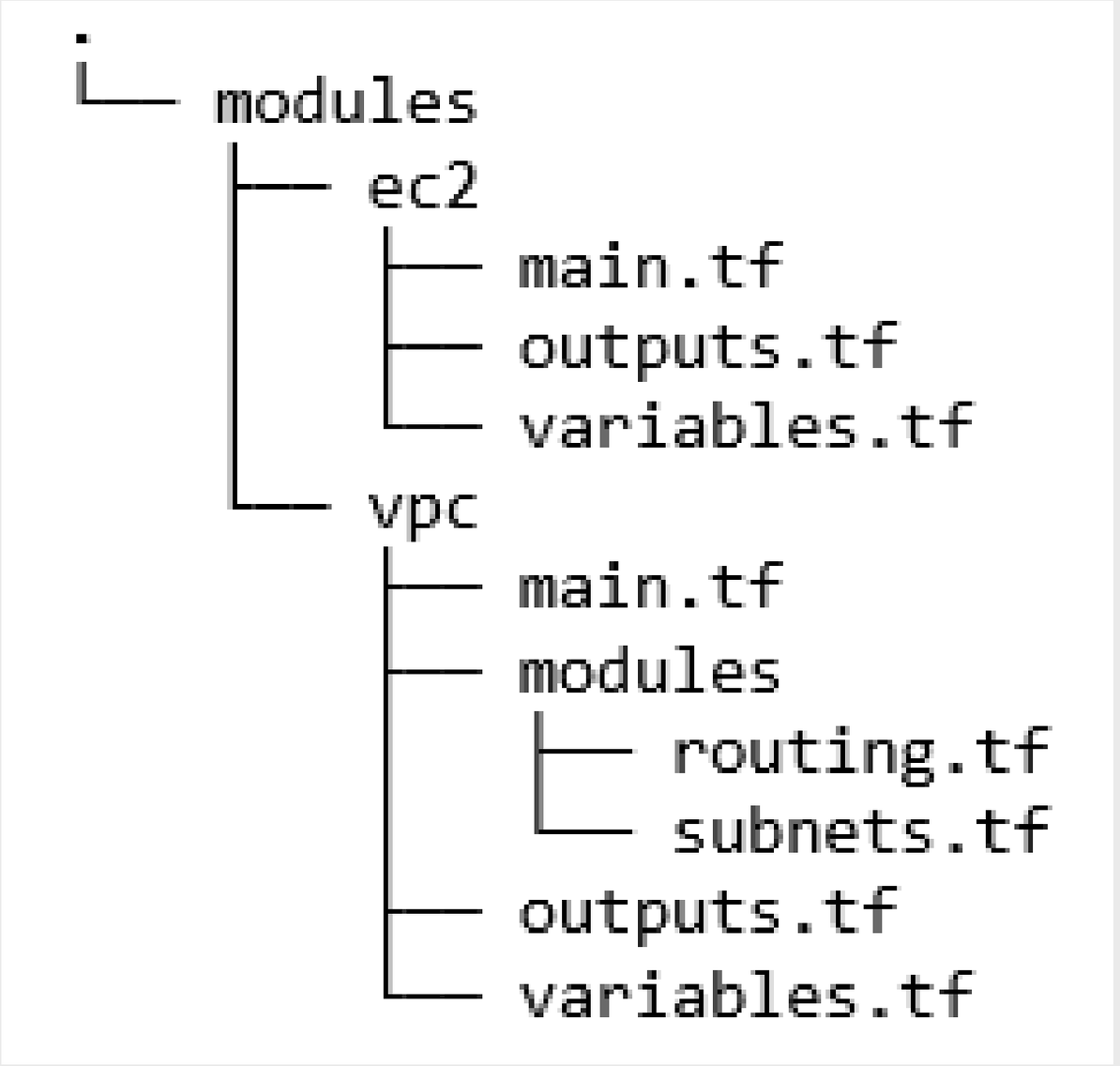


[Module Structure]

This is referred to as the child module as it is being called from the standard root module



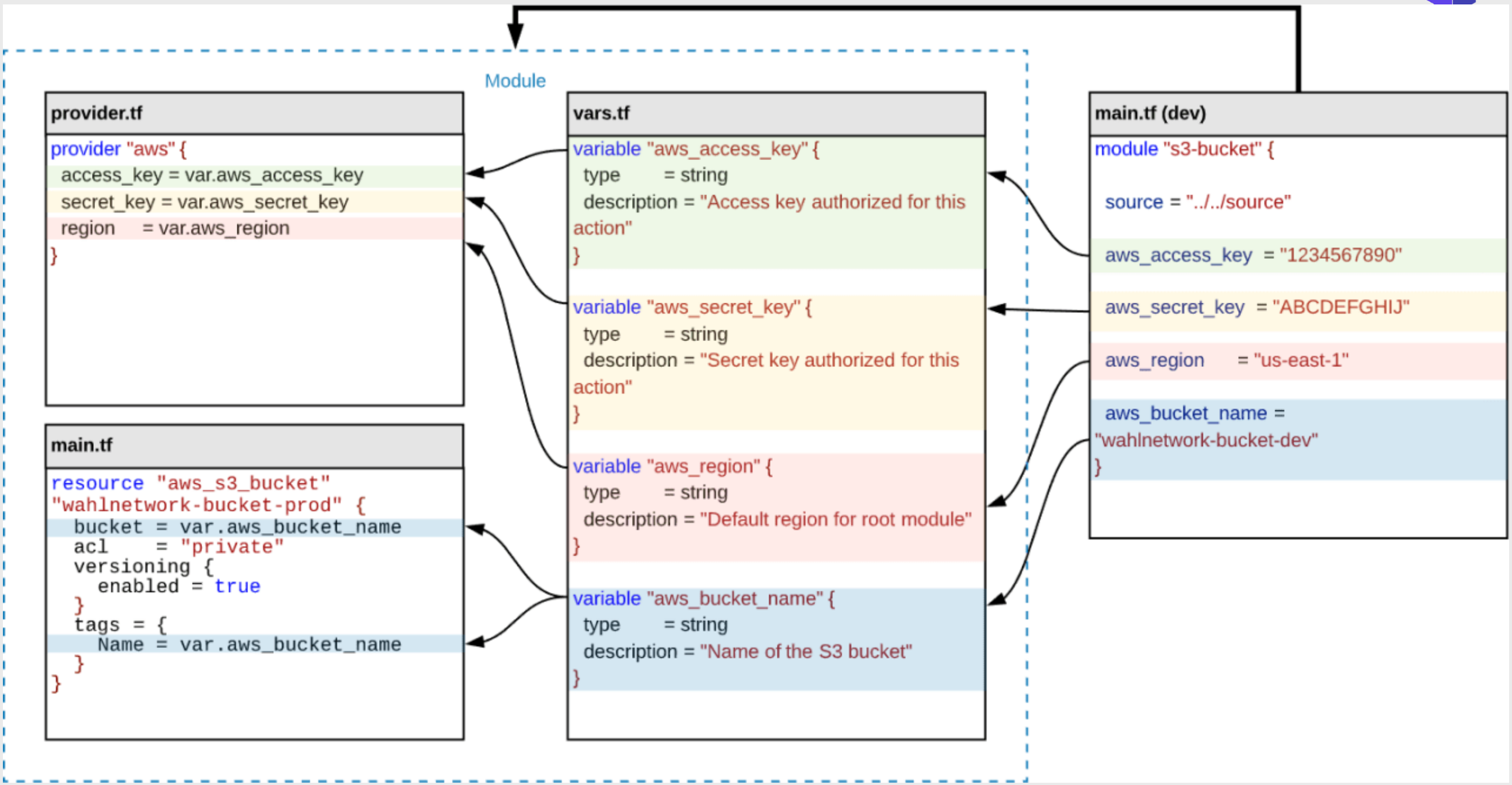

storage
IP
VM
firewall
etc ...
Module "server"


.
modules
ec2
main.tf
outputs.tf
variables.tf
vpc
main.tf
modules
routing.tf
subnets.tf
outputs.tf
variables.tf

Tf modules

What are modules for?

- **Organize configuration** - Modules make it easier to navigate, understand, and update your configuration by keeping related parts of your configuration together. Even moderately complex infrastructure can require hundreds or thousands of lines of configuration to implement.
- **Encapsulate configuration** - Another benefit of using modules is to encapsulate configuration into distinct logical components. Encapsulation can help prevent unintended consequences, such as a change to one part of your configuration accidentally causing changes to other infrastructure, and reduce the chances of simple errors like using the same name for two different resources.
- **Re-use configuration** - Writing all of your configuration from scratch can be time consuming and error prone. Using modules can save time and reduce costly errors by re-using configuration written either by yourself, other members of your team, or other Terraform practitioners who have published modules for you to use. You can also share modules that you have written with your team or the general public, giving them the benefit of your hard work.
- **Provide consistency and ensure best practices** - Modules also help to provide consistency in your configurations. Not only does consistency make complex configurations easier to understand, it also helps to ensure that best practices are applied across all of your configuration. For instance, cloud providers give many options for configuring object storage services, such as Amazon S3 or Google Cloud Storage buckets. There have been many high-profile security incidents involving incorrectly secured object storage, and given the number of complex configuration options involved, it's easy to accidentally misconfigure these services.


Module
provider.tf
provider "aws" {
access_key = var.aws_access_key
secret_key = var.aws_secret_key
region = var.aws_region
}
main.tf
resource "aws_s3_bucket" "wahlnetwork-bucket-prod" {
bucket = var.aws_bucket_name
acl = "private"
versioning {
enabled = true
}
tags = {
Name = var.aws_bucket_name
}
}
vars.tf
variable "aws_access_key" {
type = string
description = "Access key authorized for this action"
}
variable "aws_secret_key" {
type = string
description = "Secret key authorized for this action"
}
variable "aws_region" {
type = string
description = "Default region for root module"
}
variable "aws_bucket_name" {
type = string
description = "Name of the S3 bucket"
}
main.tf (dev)
module "s3-bucket" {
source = "../../source"
aws_access_key = "1234567890"
aws_secret_key = "ABCDEFGHIJ"
aws_region = "us-east-1"
aws_bucket_name = "wahlnetwork-bucket-dev"
}

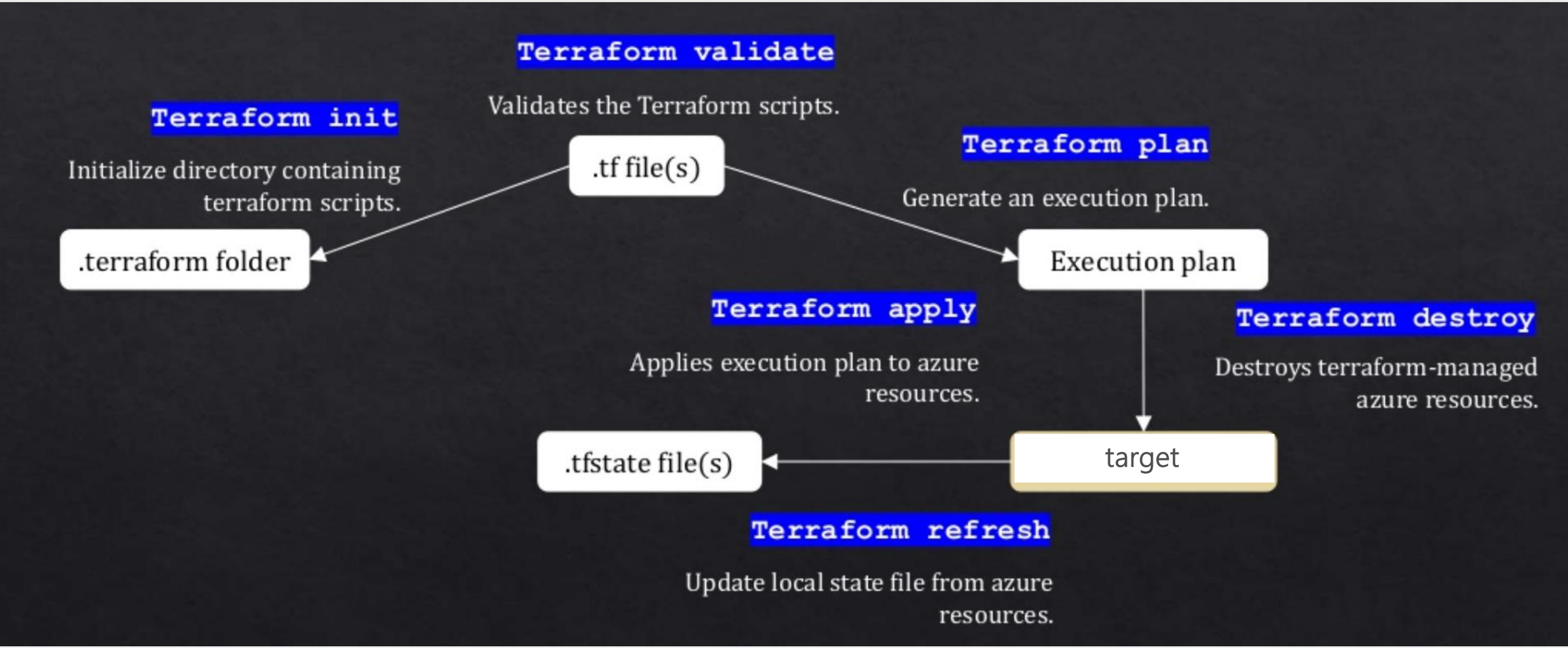
Local – with path

```
module "vpc_example_complete-vpc" {
  source  = “../modules/ec2instance"
  version = "2.70.0"
}
```

Tfregistry - https://registry.terraform.io/browse/modules

```
module "vpc_example_complete-vpc" {
  source  = "terraform-aws-modules/vpc/aws//examples/complete-vpc"
  version = "2.70.0"
}
```

tfState…


Terraform validate
Validates the Terraform scripts.
Terraform init
Initialize directory containing terraform scripts.
.tf file(s)
Terraform plan
Generate an execution plan.
.terraform folder
Execution plan
Terraform apply
Applies execution plan to azure resources.
Terraform destroy
Destroys terraform-managed azure resources.
.tfstate file(s)
target
Terraform refresh
Update local state file from azure resources.

Tf Remote state

Version Control

Remote State Backends

main.tf

```
resource "local_file" "pet" {
  filename = "/root/pet.txt"
  content  = "My favorite pet is Mr.Whiskers!"
}
resource "random_pet" "my-pet" {
  length = 1
}
resource "local_file" "cat" {
  filename = "/root/cat.txt"
  content  = "I like cats too!"
}
```

terraform.tfstate

```
{
      "mode": "managed",
      "type": "aws_instance",
      "name": "dev-ec2",
      "provider": "provider[\"registry.terraform.io/hashicorp/aws\"]",
      "instances": [
        {
          "schema_version": 1,
          "attributes": {
            "ami": "ami-0a634ae95e11c6f91",
.
.
.
            "primary_network_interface_id": "eni-0ccd57b1597e633e0",
            "private_dns": "ip-172-31-7-21.us-west-2.compute.internal",
            "private_ip": "172.31.7.21",
            "public_dns": "ec2-54-71-34-19.us-west-2.compute.amazonaws.com",
            "public_ip": "54.71.34.19",
            "root_block_device": [
              {
                "delete_on_termination": true,
                "device_name": "/dev/sda1",
                "encrypted": false,
                "iops": 100,
                "kms_key_id": "",
                "volume_id": "vol-070720a3636979c22",
```

Terraform design flow