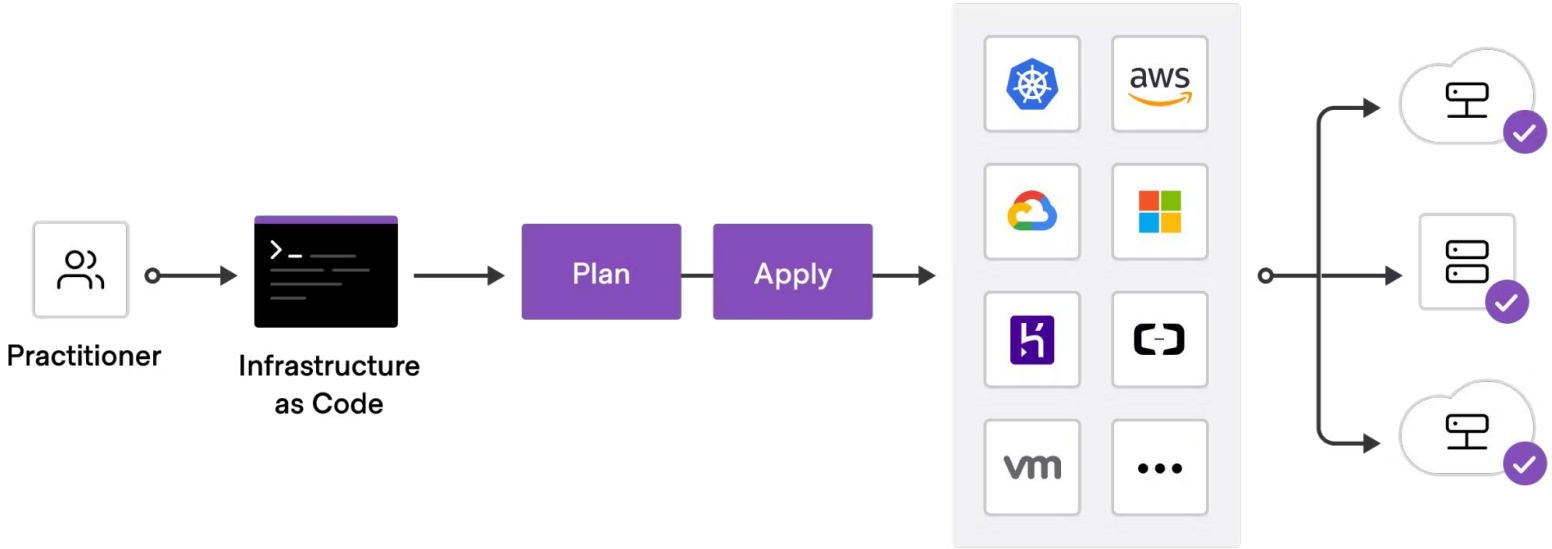
Terraform

Introduction & Modules

Terraform

- **Infrastructure as Code (IaC) tools allow you to manage infrastructure with configuration files rather than through a graphical user interface. IaC allows you to build, change, and manage your infrastructure in a safe, consistent, and repeatable way by defining resource configurations that you can version, reuse, and share.**
- **Terraform is HashiCorp's infrastructure as code tool. It lets you define resources and infrastructure in human-readable, declarative configuration files, and manages your infrastructure's lifecycle.**



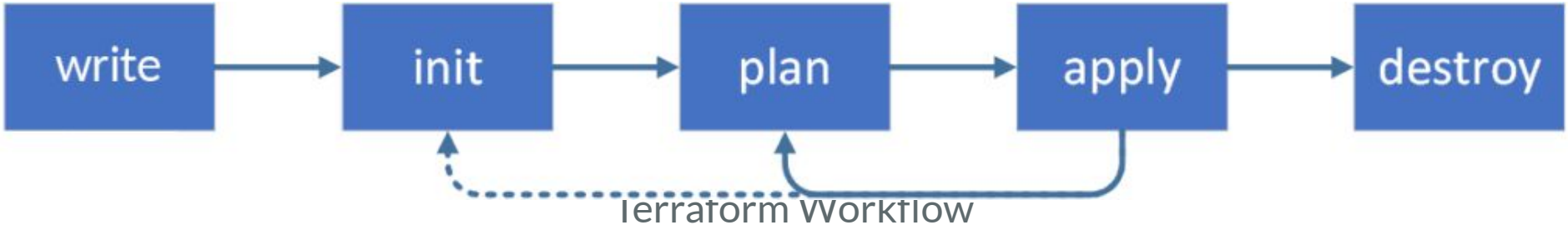
What Terraform Can Do ?

- **Terraform can manage infrastructure on multiple cloud platforms.**
- **The human-readable configuration language helps you write infrastructure code quickly.**
- **Terraform's state allows you to track resource changes throughout your deployments.**
- **You can commit your configurations to version control to safely collaborate on infrastructure.**

Installing Terraform

```
sudo apt-get update && sudo apt-get install -y gnupg software-properties-common

wget -O- https://apt.releases.hashicorp.com/gpg | \
        gpg --dearmor | \
        sudo tee /usr/share/keyrings/hashicorp-archive-keyring.gpg > /dev/null

gpg --no-default-keyring \
        --keyring /usr/share/keyrings/hashicorp-archive-keyring.gpg \
        --fingerprint

echo "deb [signed-by=/usr/share/keyrings/hashicorp-archive-keyring.gpg] \
        https://apt.releases.hashicorp.com $(lsb_release -cs) main" | \
        sudo tee /etc/apt/sources.list.d/hashicorp.list

sudo apt update
sudo apt-get install terraform
terraform -help
```

Terraform Workflow



Terraform Commands

https://cloud.google.com/docs/terraform/basic-commands

Command	Usage
terraform init	Initialize Terraform. You only need to do this once per directory. terraform ini
terraform plan	Review the configuration and verify that the resources that Terraform is going to create or update match your expectations:
terraform init	Apply the Terraform configuration by running the following command and entering yes at the prompt or using -auto-approve to bypass interactivity.
terraform destroy	Remove resources previously applied with your Terraform configuration by running the following command and entering yes at the prompt:
terraform refresh	The terraform refresh command reads the current settings from all managed remote objects and updates the Terraform state to match.

Terraform Files

Terraform configuration files are used for writing your Terraform code. They have a .tf extension and use a declarative language called HashiCorp Configuration Language (HCL) to describe the different components that are used to automate your infrastructure.

Terraform file types include:

1. main.tf – **containing the resource blocks that define the resources to be created in the target cloud platform.**
2. variables.tf – **containing the variable declarations used in the resource blocks.**
3. provider.tf – **containing the terraform block, s3 backend definition, provider configurations, and aliases.**
4. output.tf – **containing the output that needs to be generated on successful completion of "apply" operation.**
5. *.tfvars – **containing the environment-specific default values of variables.**

These File names cannot be changed

Terraform State File

- Terraform must store state about your managed infrastructure and configuration. This state is used by Terraform to map real world resources to your configuration, keep track of metadata, and to improve performance for large infrastructures.

- Terraform uses state to determine which changes to make to your infrastructure. Prior to any operation, Terraform does a refresh to update the state with the real infrastructure.

- Terraform state file is stored locally by default but it can also be stored in an encrypted S3 bucket which is the industry norm

Terraform Providers

https://registry.terraform.io/browse/providers

- **Terraform relies on plugins called providers to interact with cloud providers, SaaS providers, and other APIs.**
- **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.**
- **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.**

provider.tf

```
provider "aws" {

  #  access_key = "${var.access_key}"

  #  secret_key = "${var.secret_key}"

    region = "ap-southeast-2"

}
```

Assignment

- Install Terraform and create all the required files in a separate folder.
- Write the provider file for aws provider.
- Initialize the folder by running $ terraform init
- Wait for the process to complete

Terraform Output Files

- **Terraform output values let you export structured data about your resources. Such as the IP address, public DNS, Instance ID (the things which we had to grep|awk|sed)**
- **You can use this data to configure other parts of your infrastructure with automation tools, or as a data source for another Terraform workspace.**
- **The output file can be viewed by running the command $ terraform output.**
- **It does the job of output.txt that was used to grep|sed|awk.**

Output.tf

```
output "instance_private_ip" {
  value = aws_instance.web.private_ip
}
output "instance_public_ip" {
  description = "Public IP of EC2 instance"
  value     = aws_instance.web_server.public_ip
}
```

Terraform Variable Files

- **Input variables let you customize aspects of Terraform modules without altering the module's own source code. This functionality allows you to share modules across different Terraform configurations, making your module composable and reusable.**
- **Each input variable accepted by a module must be declared using a variable block:**
- **When you declare variables in the root module of your configuration, you can set their values using CLI options and environment variables.**

variables.tf

```
variable "aws_region" {
  description = "AWS region"
  type  = string
}
variable "ec2_instance_type" {
  description = "Instance type for EC2 instances"
  type  = string
  default  = "t2.small"
}
```

Terraform Main File

Resources

- *Resources* are the most important element in the Terraform language.
- Each resource block describes one or more infrastructure objects, such as virtual networks, compute instances, or higher-level components such as DNS records.

Example: type-aws instance

```
resource "aws_instance" "web" {
  ami           = "ami-a1b2c3d4"
  instance_type = "t2.micro"
}
```

Data

- *Data sources* allow Terraform to use information defined outside of Terraform, defined by another separate Terraform configuration, or modified by functions.
- Example Finds the 22.04 ubuntu ami

```
data "aws_ami" "ubuntu" {
  most_recent = true
  filter {
          name   = "name"
          values =
["ubuntu/images/hvm-ssd/ubuntu-jammy-22.04-amd64-se
rver-*"]
          }
  filter {
          name   = "virtualization-type"
          values = ["hvm"]
  }
  owners = ["099720109477"] # To download the official
ubuntu ami we use the ow>
}
```

Main.tf

1. **Install Git**
2. **Clone the directory or download zip file from the following repository.**

   https://github.com/suchintannit/Terraform_example

Assignment

- Write
  - main.tf,
  - variables.tf,
  - provider.tf,
  - output.tf

to create an EC2 instance and display the IP addresses of the instance. How can we automate the process using bash script.

Using S3 bucket as Terraform backend

Stores the state as a given key in a given bucket on Amazon S3. This backend also supports state locking and consistency checking via Dynamo DB, which can be enabled by setting the dynamodb_table field to an existing DynamoDB table name. A single DynamoDB table can be used to lock multiple remote state files. Terraform generates key names that include the values of the bucket and key variables.

```
terraform {
  backend "s3" {
    bucket = "mybucket"
    key    = "path/to/my/key"
    region = "us-east-1"
  }
}
```

Specify Cloud Networking using Terraform

```
resource "aws_vpc" "my_vpc" {
  cidr_block = "172.16.0.0/16"

  tags = {
    Name = "tf-example"
  }
}

resource "aws_subnet" "my_subnet" {
  vpc_id            = aws_vpc.my_vpc.id
  cidr_block        = "172.16.10.0/24"
  availability_zone = "us-west-2a"

  tags = {
    Name = "tf-example"
  }
}

resource "aws_network_interface" "foo" {
  subnet_id   = aws_subnet.my_subnet.id
  private_ips = ["172.16.10.100"]

  tags = {
    Name = "primary_network_interface"
  }
}
```

Assignment

Write a terraform file to create the state file in a s3 bucket. Create a VPC, attach it to a public subnet and to a network interface. Create 2 ec2 instances in the VPC.

Run the terraform file using the workflow.

Terraform init

Terraform plan

Terraform apply

After the resources are created now in the terminal execute

Terraform destroy.

Provisioners

You can use provisioners to model specific actions on the local machine or on a remote machine in order to prepare servers or other infrastructure objects for service.

local-exec

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

  provisioner "local-exec" {
    command = "echo first"
  }

  provisioner "local-exec" {
    command = "echo second"
  }
}
```

File provisioner

The `file` provisioner copies files or directories from the machine running Terraform to the newly created resource. The `file` provisioner supports both `ssh` and `winrm` type connections

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

  # Copies the myapp.conf file to /etc/myapp.conf
  provisioner "file" {
    source      = "conf/myapp.conf"
    destination = "/etc/myapp.conf"
  }

  # Copies the string in content into /tmp/file.log
  provisioner "file" {
    content     = "ami used: ${self.ami}"
    destination = "/tmp/file.log"
  }

  # Copies the configs.d folder to /etc/configs.d
  provisioner "file" {
    source      = "conf/configs.d"
    destination = "/etc"
  }

  # Copies all files and folders in apps/app1 to
D:/IIS/webapp1
  provisioner "file" {
    source      = "apps/app1/"
    destination = "D:/IIS/webapp1"
  }
}
```

Connection block in Terraform

You can create one or more `connection` blocks that describe how to access the remote resource. One use case for providing multiple connections is to have an initial provisioner connect as the `root` user to set up user accounts and then have subsequent provisioners connect as a user with more limited permissions.

Connection blocks don't take a block label and can be nested within either a `resource` or a `provisioner`.

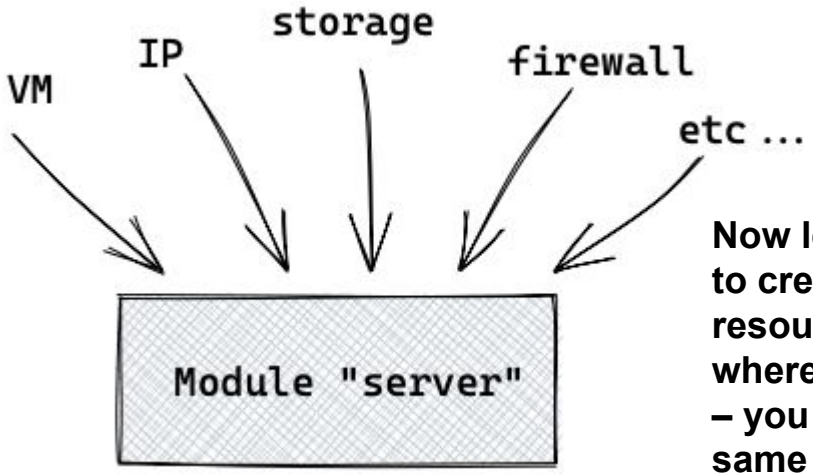
- **A `connection` block nested directly within a `resource` affects all of that resource's provisioners.**
- **A `connection` block nested in a `provisioner` block only affects that provisioner and overrides any resource-level connection settings.**

Connection block

```
resource "aws_instance" "master" {
  ami           = data.aws_ami.ubuntu.id
  instance_type = "t2.micro"
  key_name = "assignment-key-pair"
  tags = {
        Name = "master"
            }
  provisioner "file" {
   source      = "./master.sh"
   destination = "/home/ubuntu/master.sh"
   connection {
      type        = "ssh"
      user        = "ubuntu"
      private_key =
"${file("./assignment-key-pair.pem")}"
      host        = "${self.public_dns}"
    }
  }
```

Terraform Modules

- A *module* is a container for multiple resources that are used together. You can use modules to create lightweight abstractions, so that you can describe your infrastructure in terms of its architecture, rather than directly in terms of physical objects.
- The `.tf` files in your working directory when you run `terraform plan` or `terraform apply` together form the *root* module.



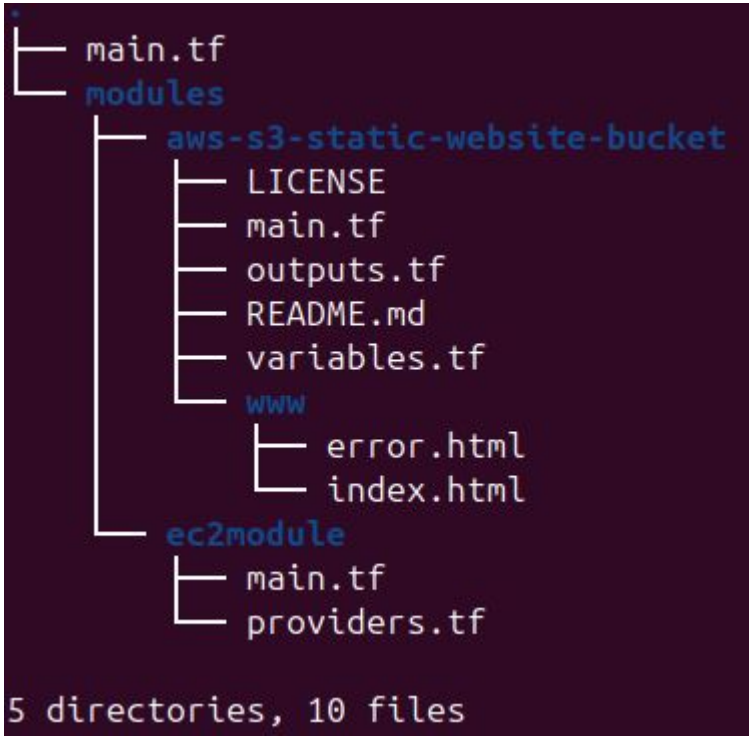
Now let's assume that you need to create this server with a set of resources many times. This is where modules are really helpful – you don't want to repeat the same configuration code over and over again, do you?

Module Sources

The module installer supports installation from a number of different source types.

- Local paths
- Terraform Registry
- GitHub
- Bitbucket
- Generic Git, Mercurial repositories
- HTTP URLs
- S3 buckets
- GCS buckets
- Modules in Package Sub-directories

Structure of a Local Module



Ec2 and VPC module

1. Create a module named “s3_module” that creates a s3 bucket, s3_bucket_policy.
2. Create a module named VPC that creates a VPC and a subnet. Store as output the VPCID and the subnet ID.

Assignment

- **Create a Terraform module for a security group tht allows SSH and HTTPS traffic. Attach the security group to multiple ec2 instances. Create a script to all of the above.**
- **Create a module called “cluster”. The module should create a VPC, a subnet in the VPC, a route table with route 0.0.0.0/0 and an Internet Gateway. Attach the route table to the IG and associate the route with the subnet. The module should also create an Ec2 instance and launch it in the above subnet.**

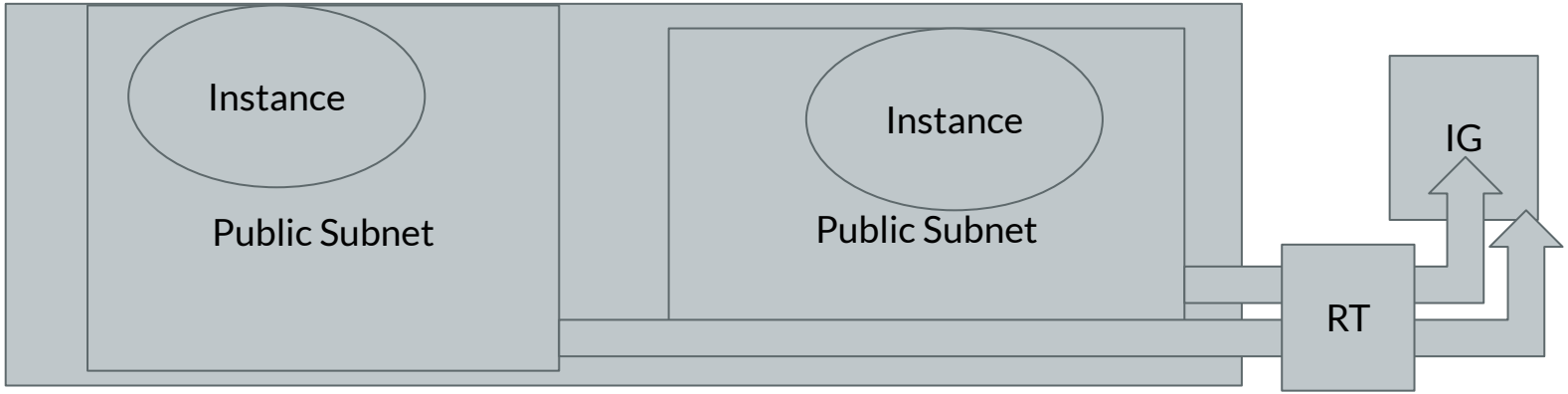
Terraform Workspaces

https://github.com/suchintannit/Terraform_example.git

Terraform ssh and file provisioner

Assignment

1. **Using terraform create the following architecture**



2. **Using terraform create an IAM user and provide the following permissions:**
   a. **EC2fullaccess**
   b. **Administratoraccess**
   c. **S3fullaccess**