HashiCorp Terraform Associate

Lecture Slides

Good luck on your exam!

What is Infrastructure as Code?

The Problem with Manual Configuration

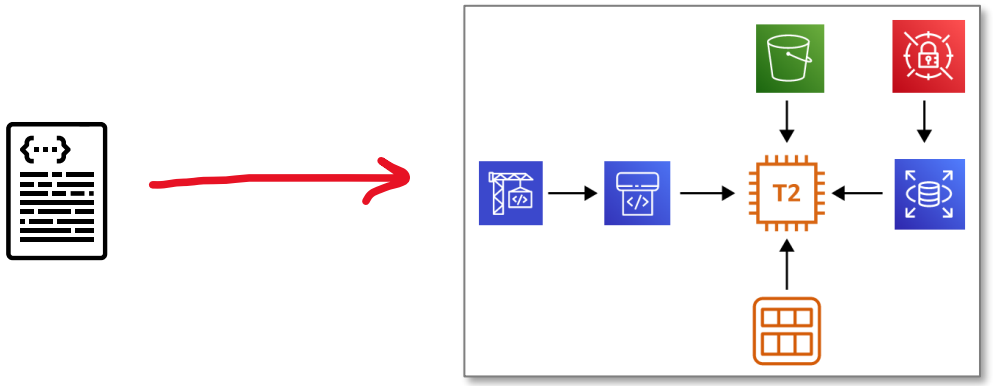
Manually configuring your cloud infrastructure allows you easily start using new service offerings to quickly prototype architectures however it comes with many downsides:

- Its easy to mis-configure a service though human error
- Its hard to manage the expected state of configuration for compliance
- Its hard to transfer configuration knowledge to other team members

Infrastructure as Code (IaC)

You write a configuration script to **automate** **creating, updating or destroying** cloud infrastructure.

- IaC is a **blueprint** of your infrastructure.
- IaC allows you to easily **share, version or inventory** your cloud infrastructure.



Popular Infrastructure as Code tools (IaC)

Declarative

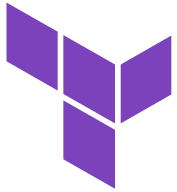
- What you see is what you get. ***Explicit***
- More verbose, but zero chance of mis-configuration
- Uses scripting languages eg. JSON, YAML, XML

ARM Templates
Supports only Azure

Azure Blueprints
Supports only Azure
Manages relationship between services

CloudFormation
Only for AWS

Cloud Deployment Manager
Supports on Google Cloud



Terraform
Supports many cloud service providers (CSPs) and cloud services.

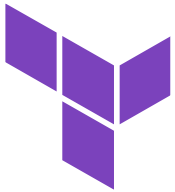
Imperative

- You say what you want, and the rest is filled in. ***Implict***
- Less verbose, you could end up with misconfiguration
- Does more than Declarative
- Uses programming languages eg. Python, Ruby, JavaScript

AWS Cloud Development Kit (CDK)
Supports only AWS
Many built-in templates for opinionated best practices

Pulumi
Supports AWS, Azure, GCP, K8

Declarative+



Terraform is declarative but the Terraform Language features **imperative-like functionality.**

Declarative		Imperative
YAML, JSON, XML	HCL-ish (Terraform Language)	Ruby, Python, JavaScript ….
Limited or no support for imperative-like features. In some cases you can add behaviour by extending the base language. E.g. CloudFormation Macros.	Supports: • Loops (For Each) • Dynamic Blocks • Locals • Complex Data Structure • Maps, Collections	Imperative features is the utility of the entire feature set of the programming language.

Infrastructure Lifecycle

What is Infrastructure Lifecycle?

a number **of clearly defined and distinct work phases** which are used by DevOps Engineers to **plan, design, build, test, deliver, maintain and retire** cloud infrastructure.

What is Day 0, Day 1 and Day 2?

Day 0-2 is a simplified way to describe phases of an infrastructure lifecycle

- Day 0 — Plan and Design
- Day 1 — Develop and Iterate
- Day 2 — Go live and maintain

Days do not literally mean **a 24 hour days** and is just a broad way of defining where a Infrastructure project would be.

Infrastructure Lifecycle

How does IaC enhance the Infrastructure Lifecycle?

Reliability

IaC makes changes **idempotent**, consistent, repeatable, and predictable.

Idempotent

No matter how many times you run IaC, you will always end up with the same state that is expected

Manageability

- enable mutation via code
- revised, with minimal changes

Sensibility

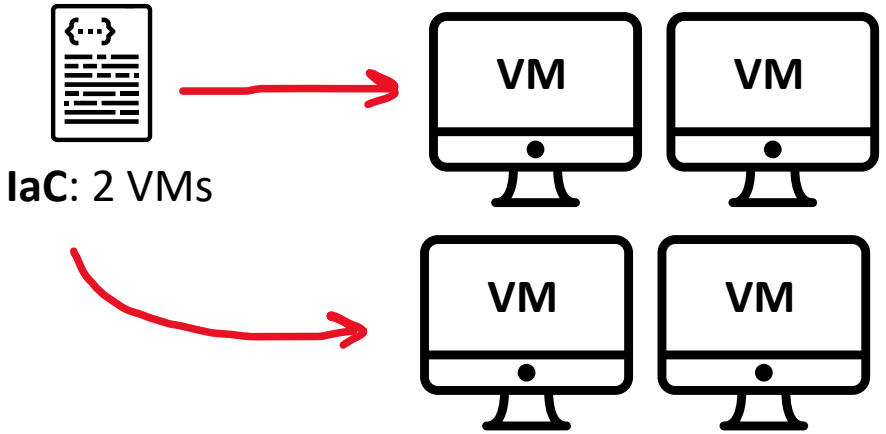
avoid financial and reputational losses to even loss of life when considering government and military dependencies on infrastructure.

Non-Idempotent vs Idempotent

Non-Idempotent

When I deploy my IaC config file it will **provision** and **launch** 2 virtual machines

When I update my IaC and deploy again, I will end up with 2 new VMs with a total of 4 VMs

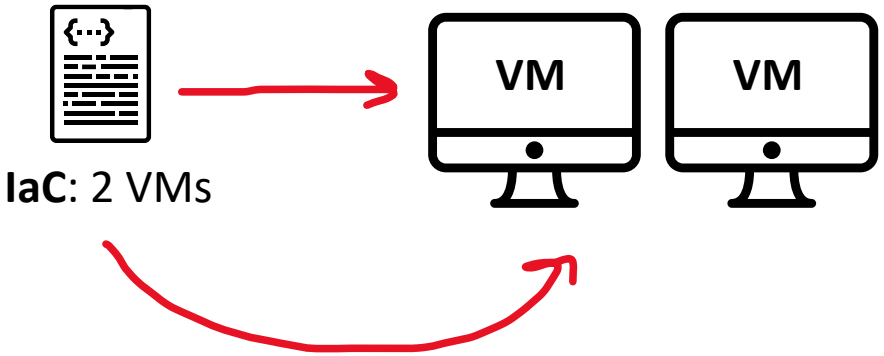


We expect a state of **2**
We end up with **4**

Idempotent

When I deploy my IaC config file it will **provision** and **launch** 2 virtual machines

When I update my IaC and deploy again, it will update the VMs if changed by **modifying or deleting and creating new VMs**



We expect a state of **2**
We end up with **2**

Provisioning vs Deployment vs Orchestration

Provisioning
To prepare a server with systems, data and software, and make it ready for network operation.
Using Configuration Management tools like Puppet, Ansible, Chef, Bash scripts, PowerShell or Cloud-Init you can provision a server.
When you launch a cloud service and configure it you are “provisioning”

Deployment
Deployment is the act of delivering a version of your application to run a provisioned server.
Deployment could be performed via AWS CodePipline, Harness, Jenkins, Github Actions, CircleCI

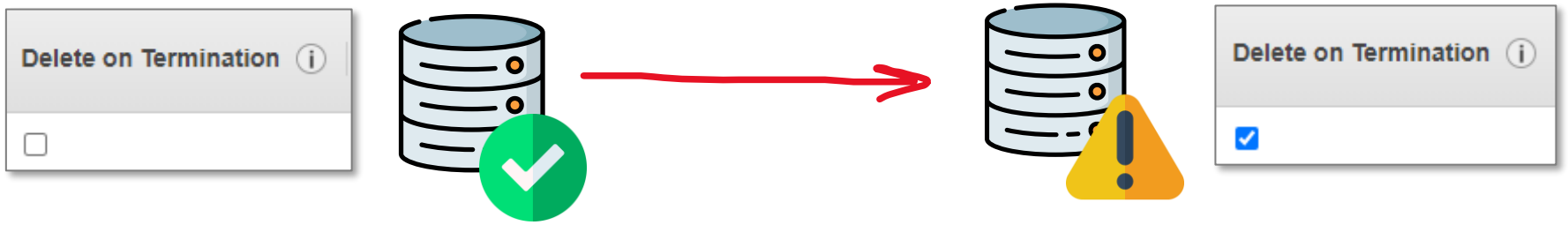
Orchestration
Orchestration is the act of coordinating multiple systems or services.
Orchestration is a common term when working with microservices, Containers and Kubernetes.
Orchestration could be Kubernetes, Salt, Fabric

Configuration Drift

Configuration Drift is when provisioned infrastructure has **an unexpected configuration change** due to:

- team members manually adjusting configuration options
- malicious actors
- side affects from APIs, SDK or CLIs.

eg, a junior developer turns on Delete on Termination for the production database.



Configuration Drift going unnoticed could be loss or breach of cloud services and residing data or result in interruption of services or unexpected downtime.

Configuration Drift

How to detect configuration drift?

- A compliance tool that can detect misconfiguration eg. AWS Config, Azure Policies, *GCP Security Health Analytics
- Built-in support for drift detection eg. AWS CloudFormation Drift Detection
- Storing the expected state eg. Terraform state files

How to correct configuration drift?

- A compliance tool that can remediate (correct) misconfiguration e.g. AWS Config
- Terraform refresh and plan commands
- Manually correcting the configuration (not recommended)
- Tearing down and setting up the infrastructure again

How to prevent configuration drift?

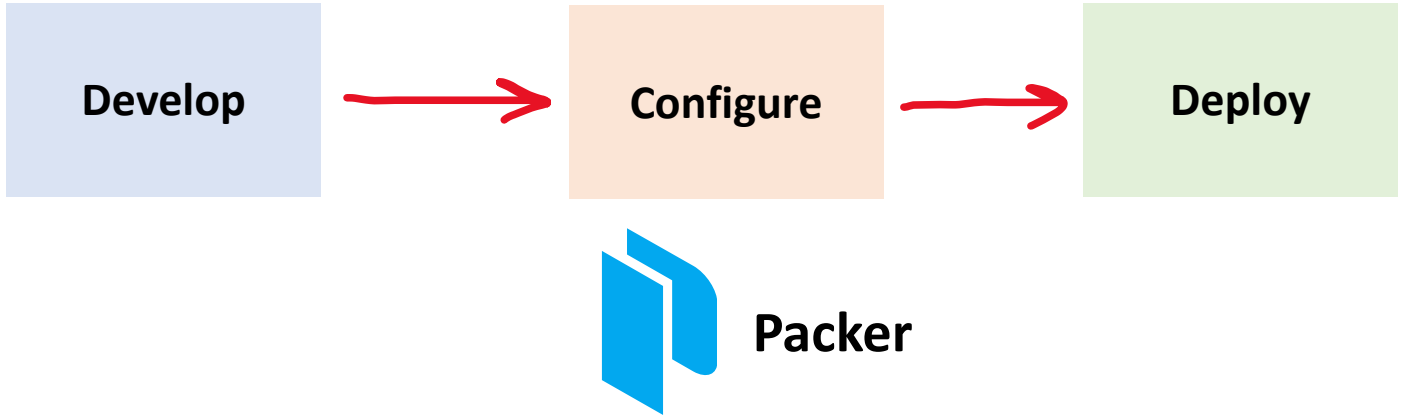
- **Immutable infrastructure**, always create and destroy, never reuse, Blue, Green deployment strategy.
    - Servers are never modified after they are deployed
        - Baking AMI images or containers via AWS Image Builder or HashiCorp Packer, or a build server eg. GCP Cloud Run
- Using GitOps to version control our IaC, and peer review every single via Pull Requests change to infrastructure

Mutable vs Immutable Infrastructure

Mutable Infrastructure

A Virtual Machine (VM) is deployed and then a Configuration Management tool like Ansible, Puppet, Chef, Salt or Cloud-Init is used to configure the state of the server

Immutable Infrastructure



A VM is launched and provisioned, and then and it is turned into a Virtual Image, stored in image repository, that image is used to deployed VM instances

Immutable Infrastructure Guarantee

Terraform encourages you towards an Immutable Infrastructure architect so you get the following guarantees.

Cloud Resource Failure – What if an EC2 instance fails a status check?

Application Failure – What if your post installation script fails due to change in package?

Time to Deploy - What if I need to deploy in a hurry?

Worst Case Scenario

- Accidental Deletion
- Compromised by malicious actor
- Need to Change Regions (region outage)

No Guarantee of 1-to-1

Every time Cloud-Init runs post deploy there is no guarantee its one-to-one with your other VMs.

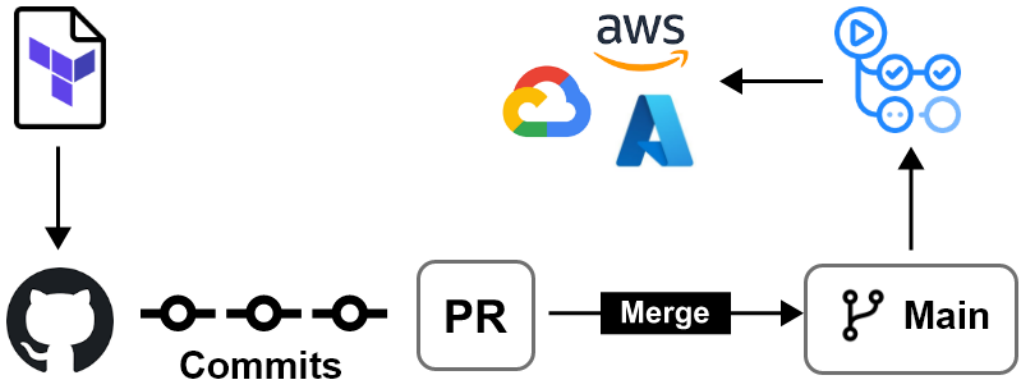
Packer

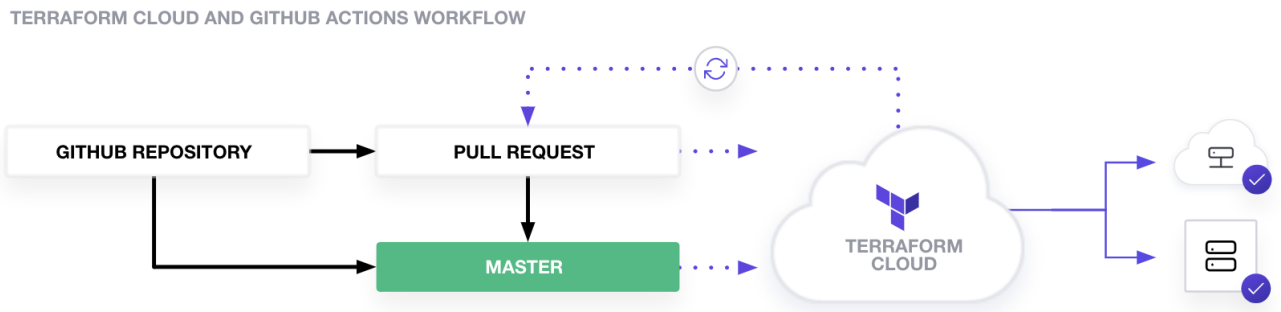
Golden Images

- Guarantee of 1-to-1 with your fleet
- Increased assurance of consistency, security
- Speeds up your deployments

What is GitOps?

GitOps is when you take Infrastructure as Code (IaC) and you use a git repository to **introduce a formal process to review and accept changes to infrastructure code**, once that code is accepted, it automatically triggers a deploy





HashiCorp

HashiCorp is a company specializing in managed **open-source tools** used to support the **development and deployment of large-scale service-oriented software installations**

What is HashiCorp Cloud Platform (HCP)?

HCP is a unified platform to access Hashicorp various products.

HCP services are **cloud agnostic**

- support for the main cloud service providers (CSPs)
  - eg. AWS, GCP and Azure
- highly suited for **multi-cloud** workloads

HashiCorp Products

Boundary
secure remote access to systems based on trusted identity.

Consul
service discovery platform. provides a full-featured service mesh for secure service segmentation across any cloud or runtime environment, and distributed key-value storage for application configuration

Nomad
scheduling and deployment of tasks across worker nodes in a cluster

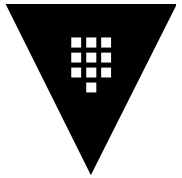
Packer
tool for building virtual-machine images for later deployment.

HashiCorp Products

Terraform
infrastructure as code software which enables provisioning and adapting virtual infrastructure across all major cloud provider

Terraform Cloud – a place to storage and manage IaC in the cloud or with teams

Vagrant
building and maintenance of reproducible software-development environments via virtualization technology



Vault
secrets management, identity-based access, encrypting application data and auditing of secrets for applications, systems, and users

Waypoint
modern workflow to build, deploy, and release across platforms

What is Terraform?

Terraform is an **open-source** and **cloud-agnostic** Infrastructure as Code (IaC) tool.

Terraform uses **declarative** configuration files.

The configuration files are written in **HashiCorp Configuration Language (HCL)**.

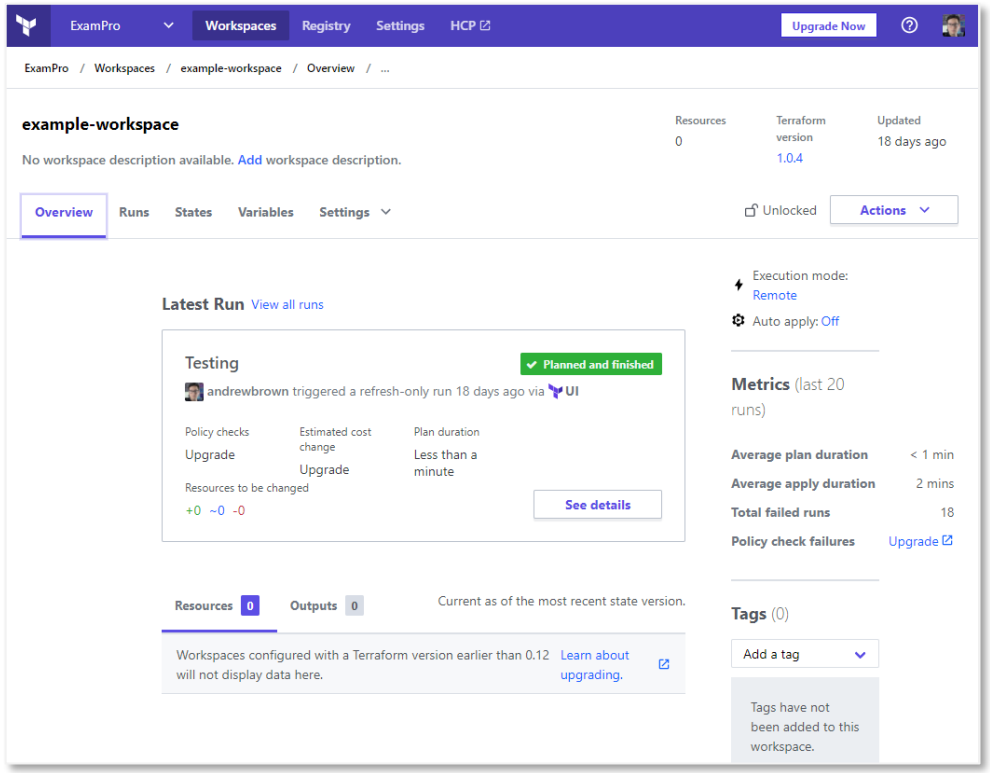
Notable features of Terraform:

- Installable modules
- Plan and predict changes
- Dependency graphing
- State management
- Provision infrastructure in familiar languages
  - via AWS CDK
- Terraform Registry with 1000+ providers

```
resource "aws_instance" "iac_in_action" {
  ami               = var.ami_id
  instance_type     = var.instance_type
  availability_zone = var.availability_zone

  // dynamically retrieve SSH Key Name
  key_name = aws_key_pair.iac_in_action.key_name

  // dynamically set Security Group ID (firewall)
  vpc_security_group_ids = [aws_security_group.iac_in_action.id]

  tags = {
    Name = "Terraform-managed EC2 Instance for IaC in Action"
  }
}
```

What is Terraform Cloud?

Terraform Cloud



Terraform Cloud is a Software as Service (SaaS) offering for:

- Remote state storage
- Version Control integrations
- Flexible workflows
- Collaborate on Infrastructure changes

in a single **unified web portal**.

www.terraform.io/cloud

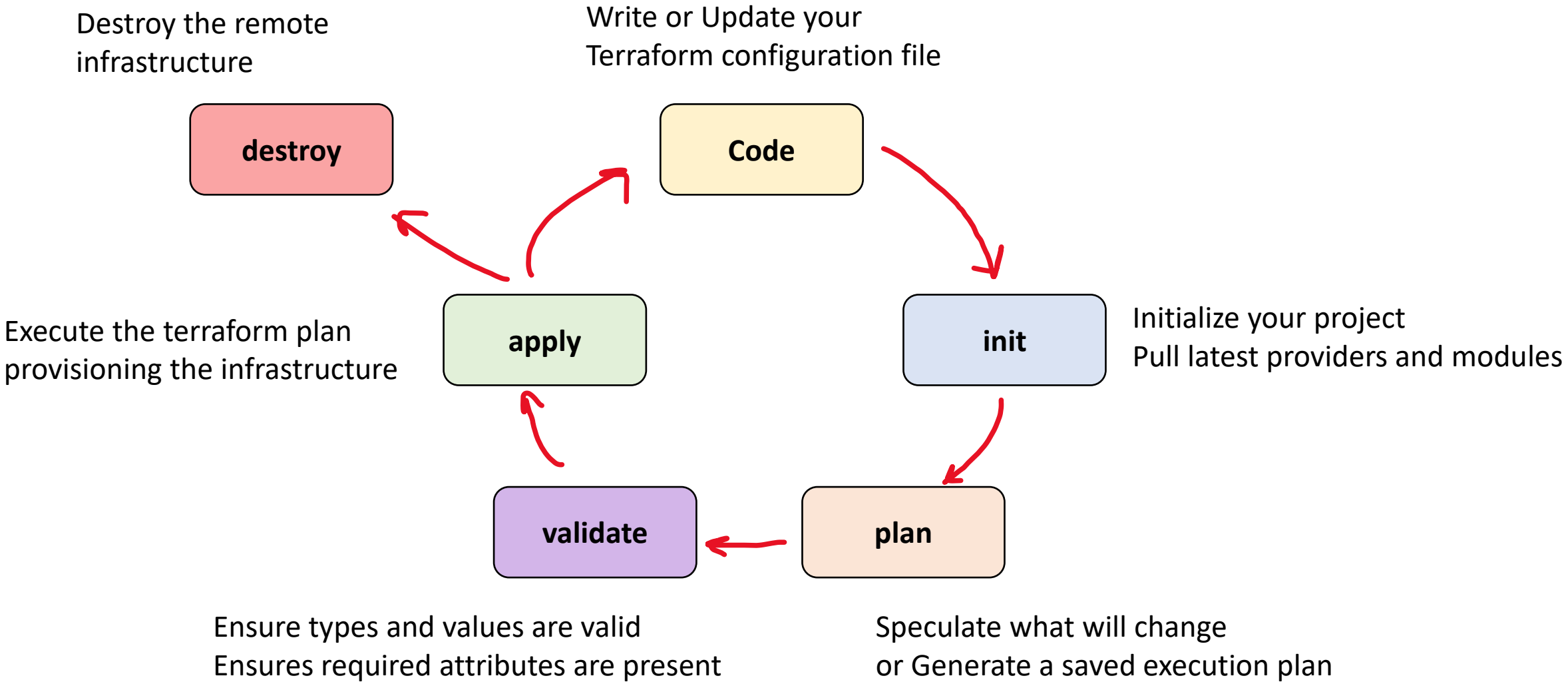
Terraform Cloud has a **generous free-tier** that allows for team collaboration for the first **5 users** of your organization

In majority of cases you should be using Terraform Cloud.

The only case where you may not want to use it to manage your state file is your company has many regulatory requirements along with a long procurement process so you could use Standard remote backend, Atlantis, or investigate Terraform Enterprise

The underlying software for Terraform Cloud and Terraform Enterprise is known as the “Terraform Platform”

Terraform Lifecycle



Terraform – Change Automation

What is Change Management?
A standard approach to apply change, and resolving conflicts brought about by change.
In the context of Infrastructure as Code (IaC), Change management is the procedure that will be followed when resources are modified and applied via configuration script.

What is Change Automation?
a way of **automatically** creating a consistent, systematic, and predictable way of managing change request via controls and policies

Terraform uses Change Automation in the form of **Execution Plans** and **Resources graphs** to apply and review complex **changesets**

What is a ChangeSet?
A collection of commits that represent changes made to a versioning repository. IaC uses ChangeSets so you can see what has changed by who over time.

Change Automation allows you to know exactly what Terraform will change and in what order, avoiding many possible human errors.

Terraform - Execution Plans

An **Execution Plan** is a **manual review** of what will **add, change or destroy** before you apply changes eg. terraform apply

resources and configuration settings will be listed.

It will indicate will be added, changed or destroyed if this plan is approved:

A user must approve changes by typing: **yes**

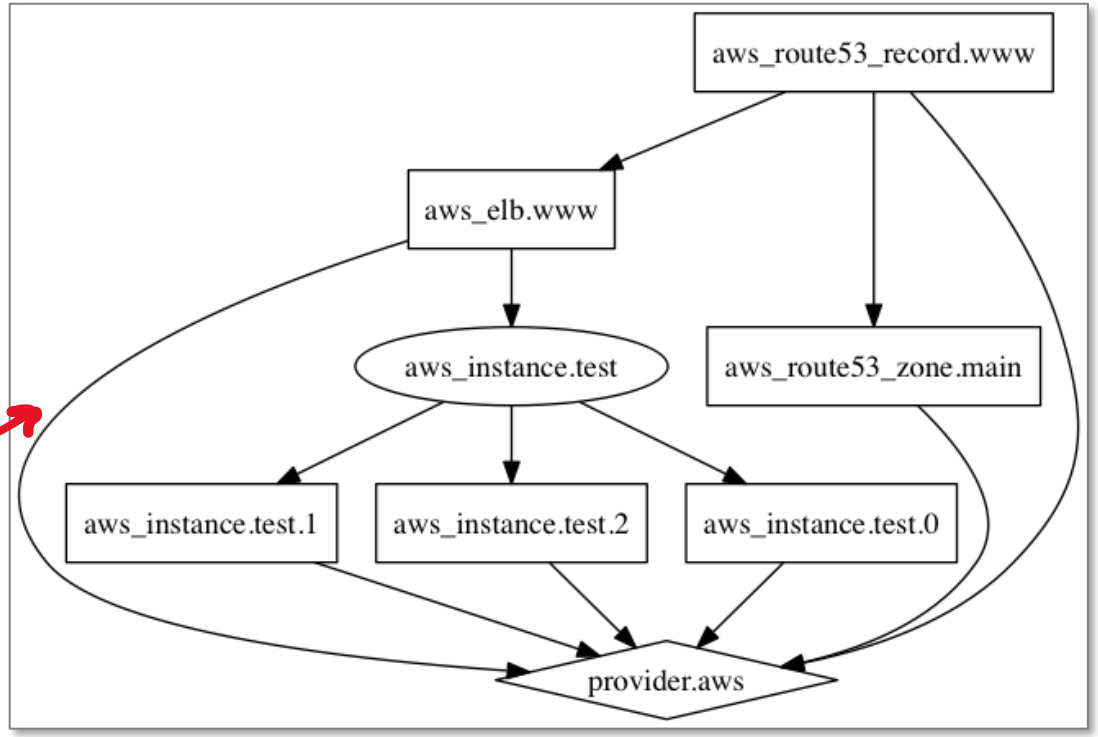
```
Terraform used the selected providers to generate the following execution
plan. Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # aws_instance.app_server will be created
  + resource "aws_instance" "app_server" {
      + ami                                  = "ami-830c94e3"
      + arn                                  = (known after apply)
      + associate_public_ip_address          = (known after apply)
      + availability_zone                    = (known after apply)
      + cpu_core_count                       = (known after apply)
      + cpu_threads_per_core                 = (known after apply)
      + disable_api_termination              = (known after apply)
      + ebs_optimized                        = (known after apply)
      + get_password_data                    = false
  ...
Plan: 1 to add, 0 to change, 0 to destroy.

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
```

Terraform – Visualizing Execution Plans

You can **visualize an execution plan** as a graph using the **terraform graph** command
Terraform will output a GraphViz file (you'll need GraphViz installed to view the file)

What is GraphViz?
open-source tools for drawing graphs specified in DOT language scripts having the file name extension "gv"

```
terraform graph | dot -Tsvg > graph.svg
```

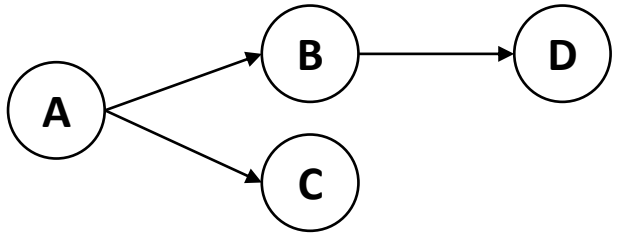


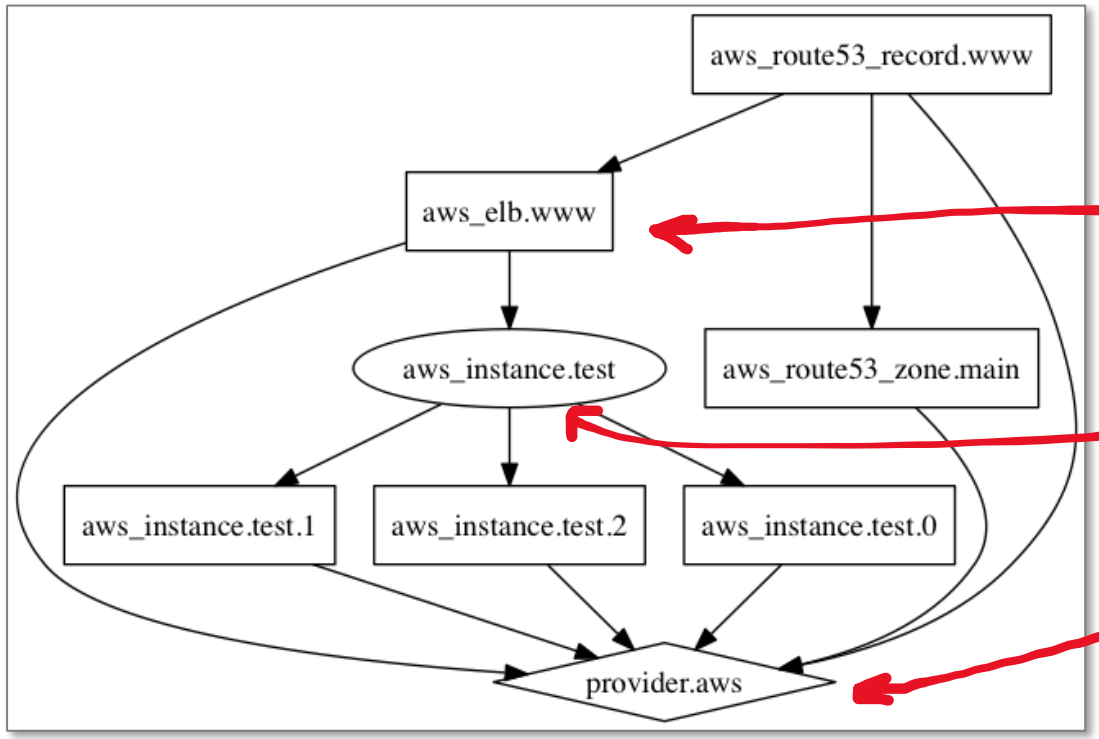
Terraform – Resource Graph

Terraform builds a **dependency graph** from the Terraform configurations, and walks this graph to generate plans, refresh state, and more.

When you use **terraform graph**, this is a **visual presentation** of the dependency graph

What is a dependency graph?
In mathematics is a directed graph representing dependencies of several objects towards each other





Resource Node
Represents a single resource

Resource Meta-Node
Represents a group of resources, but does not represent any action on its own

Provider Configuration Node
Represents the time to fully configure a provider

Terraform – Use Cases

IaC for Exotic Providers
Terraform supports a variety of providers outside of GCP, AWS, Azure and sometimes is the only provider. Terraform is open-source an extendable so any API could be used to create IaC tooling for any kind of cloud platform or technology. E.g. Heroku, Spotify Playlists.

Multi-Tier Applications – Terraform by default makes it easy to divide large and complex applications into isolate configuration scripts (module). It has a complexity advantage over cloud-native IaC tools for its flexibility while retaining simplicity over Imperative tools.

Disposable Environments – Easily stand up an environment for a software demo or a temporary development environment

Resource Schedulers – Terraform is not just defined to infrastructure of cloud resource but can be used to dynamic schedule Docker containers, Hadoop, Spark and other software tools. You can provision your own scheduling grid.

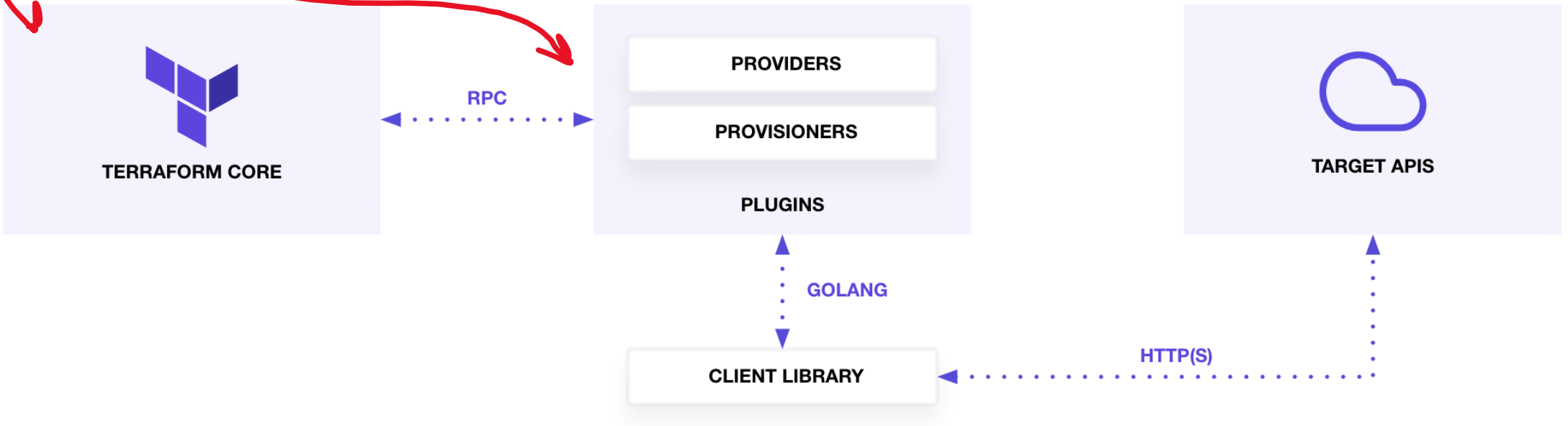
Multi-Cloud Deployment – Terraform is cloud-agnostic and allows a single configuration to be used to manage multiple providers, and to even handle cross-cloud dependencies.

Terraform Core and Terraform Plugins

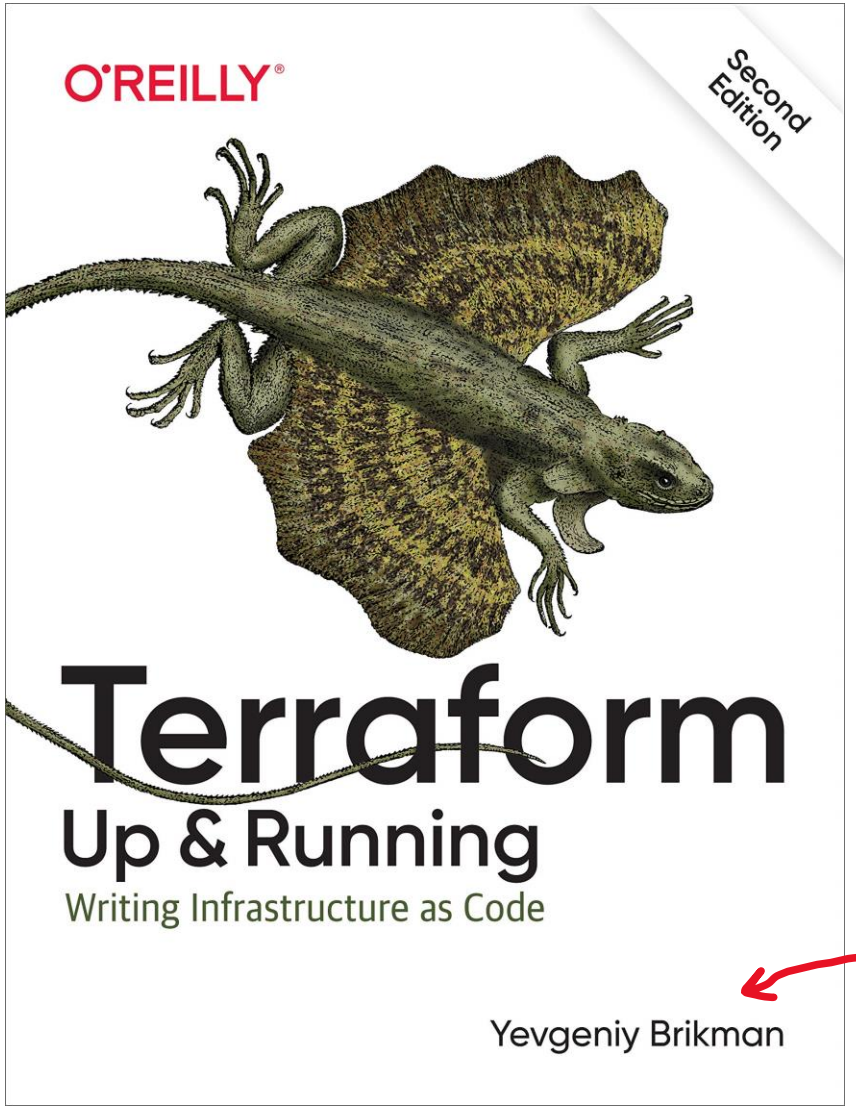
Terraform is logically split into two main parts:

1. **Terraform Core**
   - uses remote procedure calls (RPC) to communicate with Terraform Plugins
2. **Terraform Plugins**
   - expose an implementation for a specific service, or provisioner

Terraform Core is a statically-compiled **binary written in the Go programming language**.



Terraform Up and Running



Terraform Up & Running takes a deep dive into the internal workings of Terraform.

If you want to go beyond this course for things like:

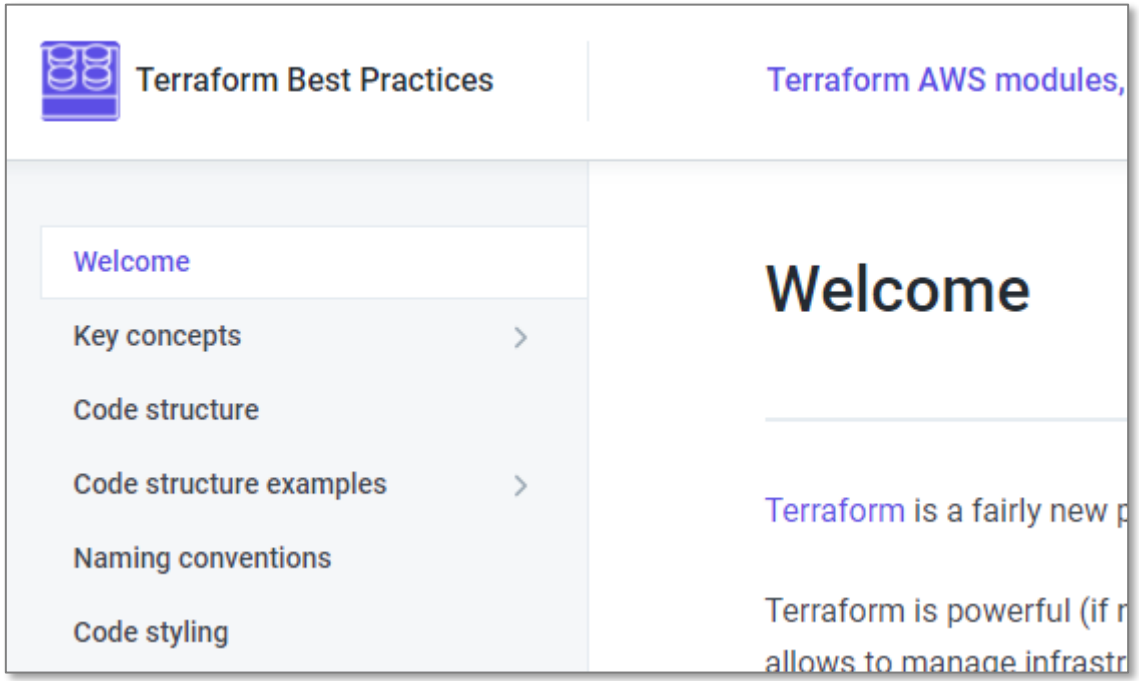
- Testing you Terraform Code
- Zero-Downtime Deployment
- Common Terraform Gotchas
- Composition of Production Grade Terraform Code

Yevgeniy ("Jim") Brikman is the co-founder of **Gruntwork**

Terraform Best Practices

Terraform is an **online open-book** about **Terraform Best Practices**

www.terraform-best-practices.com

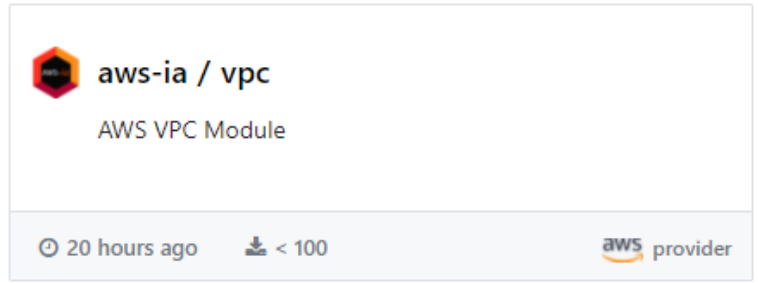


AWS-IA vs Terraform AWS Modules

AWS IA **Not Recommended**

Authored by AWS Integration and Automation team

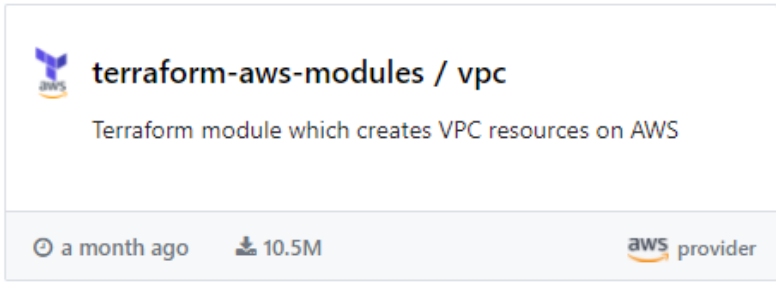
- They don't follow best practices for Terraform module development
- Very limited in functionality
- Very few libraries
- Are new, do not have much use



Terraform AWS Modules **Recommended**

Authored by Anton Babenko

- They follow best practices for Terraform Modules development
- Very flexible, covering many different use cases
- Many different libraries
- Have been around for **5 years**, heavily used (even previously by AWS)

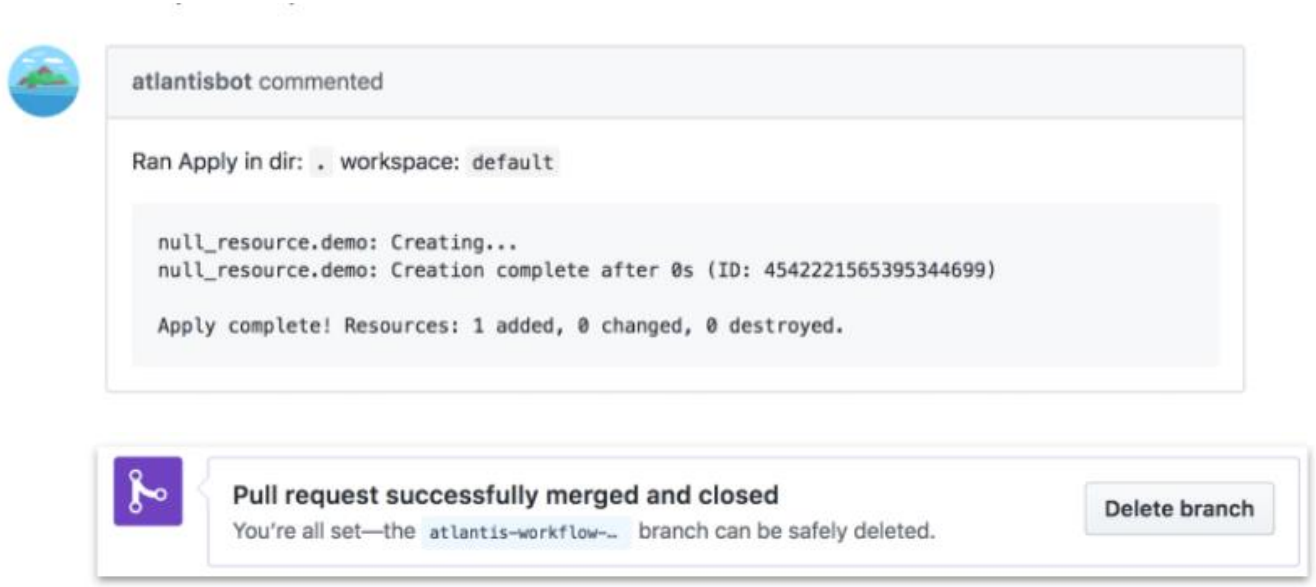


10M!

Atlantis

Atlantis is an open-source developer tool to
automate Terraform Pull Requests

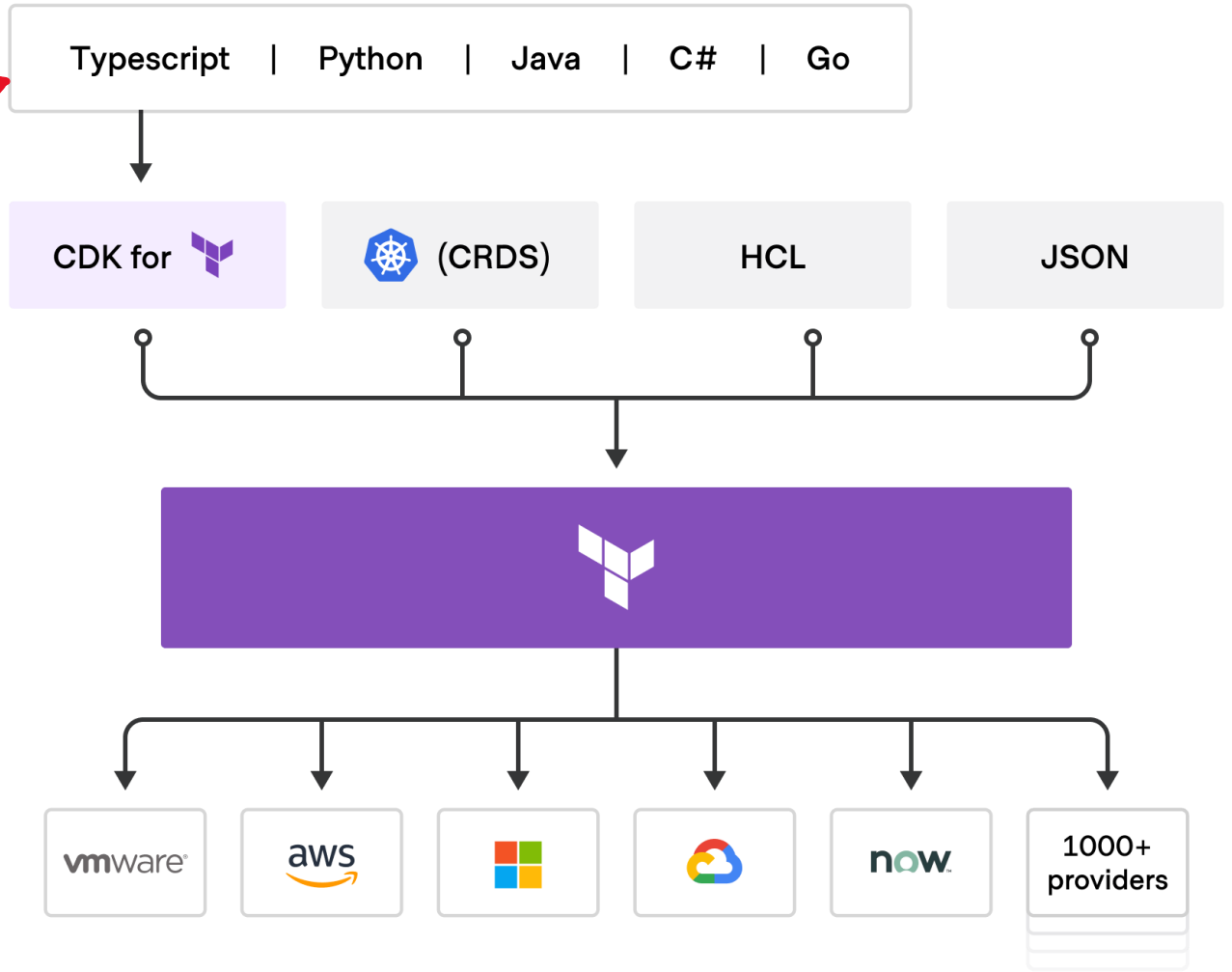
https://www.runatlantis.io



Atlantis was built as an alternative to Terraform Cloud automation.
The creators of Atantis now work at HashiCorp and is maintained by HashiCorp.

CDK for Terraform

AWS Cloud Development Kit (CDK) is an **imperative** Infrastructure as Code (IaC) tool With SDKs for your **favorite language**



AWS CDK is intended only for AWS cloud resources. CDK generates out CloudFormation (CFN) templates (known as synthesizing) and uses that for IaC.

CDK for Terraform is a standalone project by HashiCorp that allows you to use CDK, but instead of CFN templates it generates out Terraform templates.

This allows you **to use the CDK tooling to define IaC resources for any provider** available on Terraform via CDK.

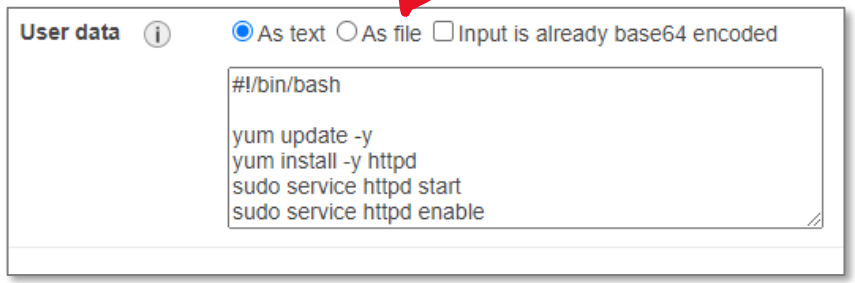
Terraform Provisioners

Terraform Provisioners install software, edit files, and provision machines created with Terraform

Terraform allows you to work with two different provisioners:

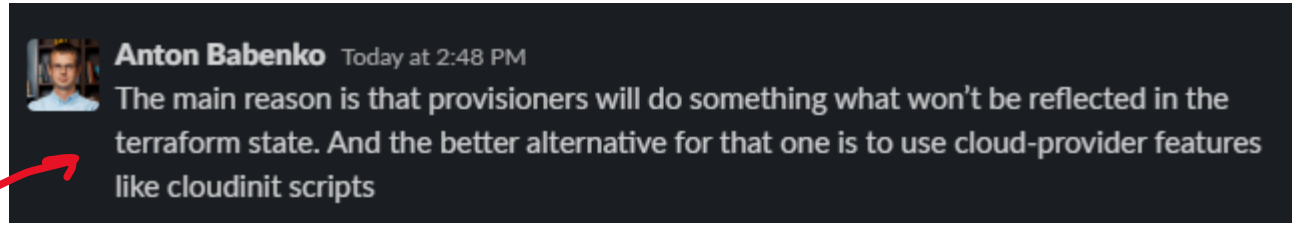
cloud-init

Cloud-Init is an industry standard for cross-platform cloud instance initializations. When you launch a VM on a Cloud Service Provider (CSP) you'll provide a YAML or Bash script.



Packer

Packer is an automated image-builder service. You provide a configuration file to create and provision the machine image and the image is the delivered to a repository for use.



Provisioners should only be used as a **last resort**. For most common situations there are better alternatives.

Terraform Provisioners

Create your own Cloud-Init script

```
users:
  - default
  - name: terraform
    gecos: terraform
    primary_group: hashicorp
    sudo: ALL=(ALL) NOPASSWD:ALL
    groups: users, admin
    ssh_import_id:
    lock_passwd: false
    ssh_authorized_keys:
      -  # Paste your created SSH key here
package_upgrade: yes
package_update: yes
packages:
  - httpd
runcmd:
  - sudo service httpd start
  - sudo service httpd enable
```

Define the template file

```
data "template_file" "user_data" {
  template = file("../scripts/add-ssh-web-app.yaml")
}

resource "aws_instance" "web" {
  ami                         = data.aws_ami.ubuntu.id
  instance_type               = "t2.micro"
  subnet_id                   = aws_subnet.subnet_public.id
  vpc_security_group_ids      = [aws_security_group.sg_22_80.id]
  associate_public_ip_address = true
  user_data                   = data.template_file.user_data.rendered

  tags = {
    Name = "Learn-CloudInit"
  }
}
```

Reference it in the userdata for the Virtual Machine

Terraform Provisioners

Terraform use to directly to support third-party Provisioning tools in the Terraform language Support was **deprecated** because Terraform considered using Provisioners to be poor practice suggesting better alternatives.

Cloud-Init supports Chef and Puppet, so you can just use Cloud-Init

```
puppet:
  install: true
  version: "7.7.0"
  install_type: "packages"
  collection: "puppet7"
  cleanup: true
  aio_install_url:
"https://raw.githubusercontent.com/puppetlabs/install-
puppet/main/install.sh"
  conf_file: "/etc/puppet/puppet.conf"
  ssl_dir: "/var/lib/puppet/ssl"
  csr_attributes_path: "/etc/puppet/csr_attributes.yaml"
  package_name: "puppet"
  exec: false
  exec_args: ['--test']
  conf:
    agent:
      server: "puppetserver.example.org"
```

It is uncertain if this advice extends to Ansible. Terraform and Ansible has lots of learning materials on how their technologies complement each other.

Local-exec

Local-exec allows you to execute **local commands** *after* a resource is provisioned.

The machine that is executing Terraform *eg. terraform apply* is **where** the command will execute.

A local environment could be…..

Local Machine
Your laptop / workstation

Build Server
eg. GCP Cloud Build, AWS CodeBuild, Jenkins

Terraform Cloud Run Environment
single-use Linux virtual machine

Example Use Case
After you provision a VM you need to supply the Public IP to a third-party security service to add the VM IP address and you accomplish this by using locally installed third-party CLI on your build server.

Outputs vs Local-Exec
Terraform outputs allows you to output results after running Terraform apply

local exec allows you to run any arbitrary commands on your local machine. Commonly used to trigger Configuration Management eg. Ansible, Chef, Puppet

Local-exec

command (required)
The command you want to execute

working_dir
Where the command will be executed
eg. /user/andrew/home/project

interpreter
The entry point for the command.
What local program will run the command eg.
Bash, Ruby, AWS CLI, PowerShell

Environment
Key and value pair of environment variables

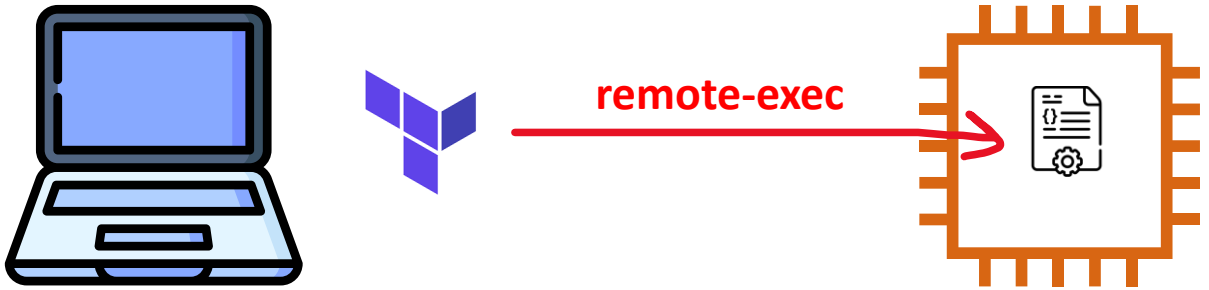
```
resource "aws_instance" "web" {
  # ...

  provisioner "local-exec" {
    command = "echo ${self.private_ip} >> private_ips.txt"
  }
}
```

```
resource "null_resource" "example2" {
  provisioner "local-exec" {
    command = "Get-Date > completed.txt"
    interpreter = ["PowerShell", "-Command"]
  }
}
```

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

  provisioner "local-exec" {
    command = "echo $KEY $SECRET >> credentials.yml"

    environment = {
      KEY = "Jlmyn61Qmc8wiux9zprS9v4n"
      SECRET = "U32KIy2T1fW3xGDlFLggXjuN"
    }
  }
}
```

Remote-exec

Remote-exec allows you to execute **commands on a target resource** *after* a resource is provisioned.



Remote-Exec is useful for provisioning a Virtual Machine with a simple set of commands.

For more complex tasks its recommended to use Cloud-Init, and strongly recommended in all cases to bake Golden Images via Packer or EC2 Image Builder

Remote-exec

Remote Command has three different modes:

Inline - list of command strings

Script - relative or absolute local script that will be copied to the remote resource and then executed

Scripts - relative or absolute local scripts that will be copied to the remote resource and then executed and executed in order.

You can only choose to use one mode at a time

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

  provisioner "remote-exec" {
    inline = [
      "puppet apply",
      "consul join ${aws_instance.web.private_ip}",
    ]
  }
}
```

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

  provisioner "remote-exec" {
    scripts = [
      "./setup-users.sh",
      "/home/andrew/Desktop/bootstrap"
    ]
  }
}
```

File

file provisioner is used to copy files or directories from our local machine to the newly created resource

Source – the local file we want to upload to the remote machine

Content – a file or a folder

Destination – where you want to upload the file on the remote machine

You may require a connection block within the provisioner for authentication

```
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

A connection block tells a **provisioner** or **resource** how to establish a connection

You can connect via **SSH**

With SSH you can connect through a Bastion Host eg:

- bastion_host
- bastion_host_key
- bastion_port
- bastion_user
- bastion_password
- bastion_private_key
- bastion_certificate

You can connect via **Windows Remote Management (winrm)**

```
# Copies the file as the root user using SSH
provisioner "file" {
  source      = "conf/myapp.conf"
  destination = "/etc/myapp.conf"

  connection {
    type     = "ssh"
    user     = "root"
    password = "${var.root_password}"
    host     = "${var.host}"
  }
}

# Copies the file as the Administrator user using
WinRM
provisioner "file" {
  source      = "conf/myapp.conf"
  destination = "C:/App/myapp.conf"

  connection {
    type     = "winrm"
    user     = "Administrator"
    password = "${var.admin_password}"
    host     = "${var.host}"
  }
}
```

Null Resources

null_resource is a placeholder for resources that have no specific association to a provider resources.

You can provide a **connection** an **triggers** to a resource

Triggers is a map of values which **should cause this set of provisioners to re-run**.

Values are meant to be interpolated references to variables or attributes of other resources

```
resource "aws_instance" "cluster" {
  count = 3

  # ...
}

resource "null_resource" "cluster" {
  # Changes to any instance of the cluster requires re-provisioning
  triggers = {
    cluster_instance_ids = "${join(",", aws_instance.cluster.*.id)}"
  }

  # Bootstrap script can run on any instance of the cluster
  # So we just choose the first in this case
  connection {
    host = "${element(aws_instance.cluster.*.public_ip, 0)}"
  }

  provisioner "remote-exec" {
    # Bootstrap script called with private_ip of each node in the cluster
    inline = [
      "bootstrap-cluster.sh ${join(" ", aws_instance.cluster.*.private_ip)}",
    ]
  }
}
```

Terraform Data

Similar to null_resources but **does not require or the configuration of a provider**.

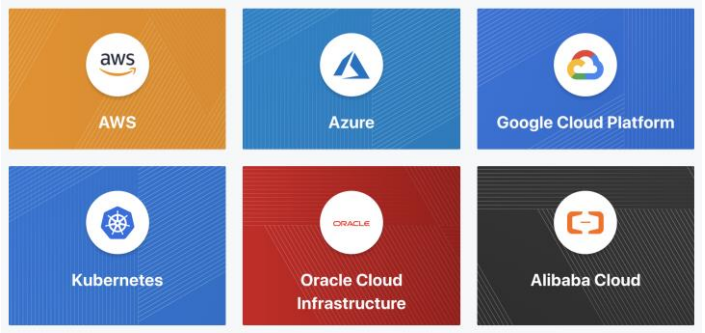
```
resource "null_resource" "main" {
  triggers = {
    version = var.version
  }

  provisioners "local-exec" {
    command = "echo ${self.triggers.version}"
  }
}
```

```
resource "terraform_data" "main" {
  triggers_replace = [
    version
  ]

  provisioners "local-exec" {
    command = "echo ${self.triggers_replace}"
  }
}
```

Terraform Data – replace_triggered_by

```
variable "revision" {
  default = 1
}

resource "terraform_data" "replacement" {
  input = var.revision
}


resource "example_database" "prod" {
  lifecycle {
    replace_triggered_by = [terraform_data.replacement]
  }
}
```

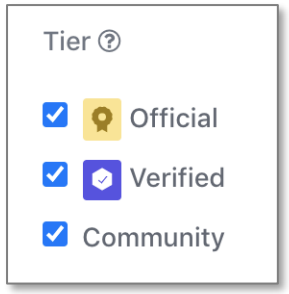
Terraform Providers



Providers are Terraform Plugins that allow you to interactive with:

- Cloud Service Providers (CSPs) eg. **AWS, Azure, GCP**
- Software as a Service (SaaS) Providers eg. **Github, Angolia, Stripe**
- Other APIs eg. **Kubernetes, Postgres**

Providers are required for your Terraform Configuration file to work.



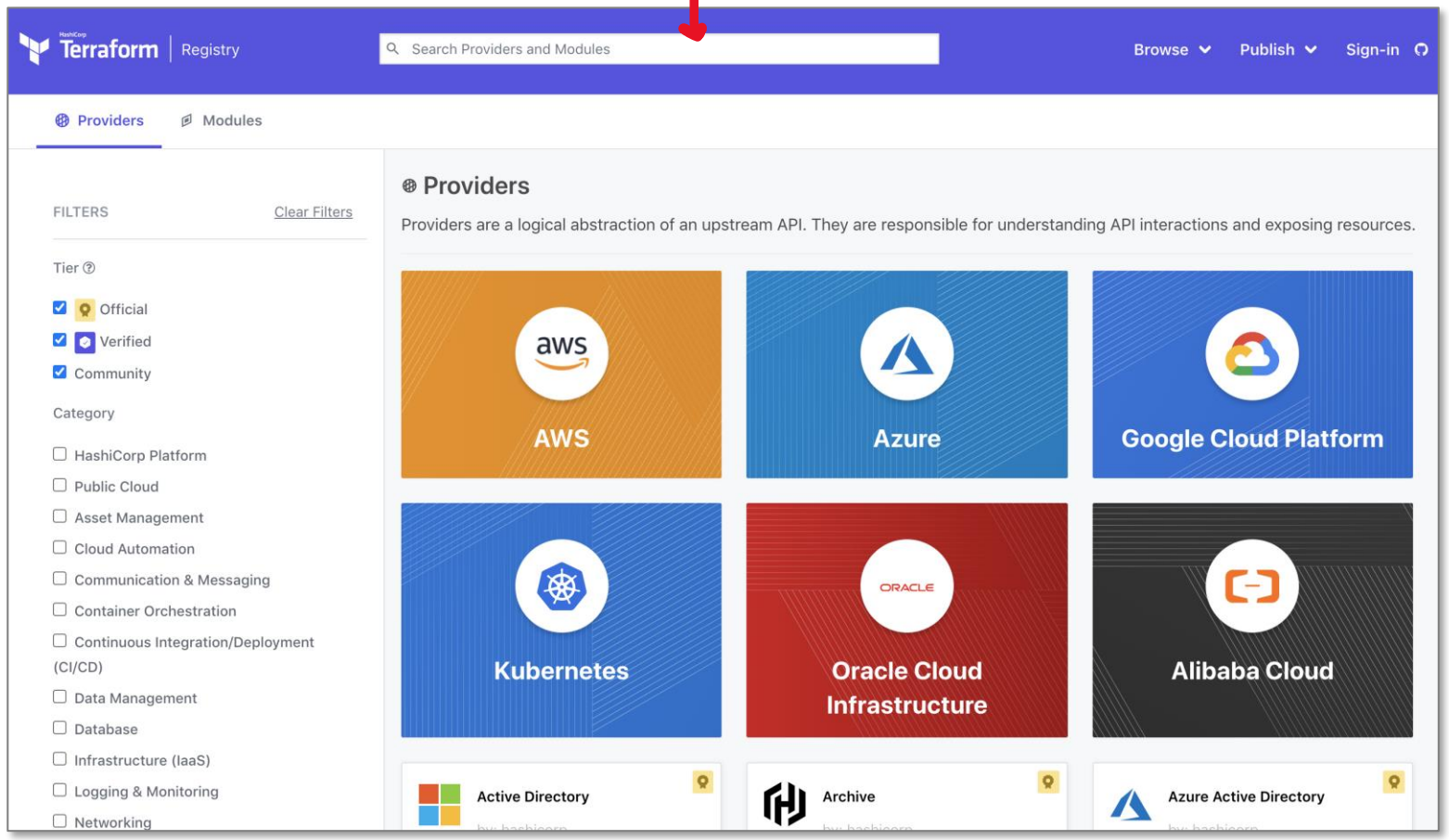
Providers come in three tiers:
Official — Published by the company that owns the provider technology or service
Verified — actively maintained, up-to-date and compatible with both Terraform and Provider
Community — published by a community member but no guarantee of maintenance, up-to-date or compatibility

Providers are distributed separately from Terraform and the plugins must be downloaded before use.
terraform init will download the necessary provider plugins listed in a Terraform configuration file.

Terraform Registry

Terraform Registry is a **website portal** to **browse, download or publish** available **Providers** or **Modules**

https://registry.terraform.io



Provider

A provider is a plugin that is mapping to a Cloud Service Provider (CSPs) API.

Module

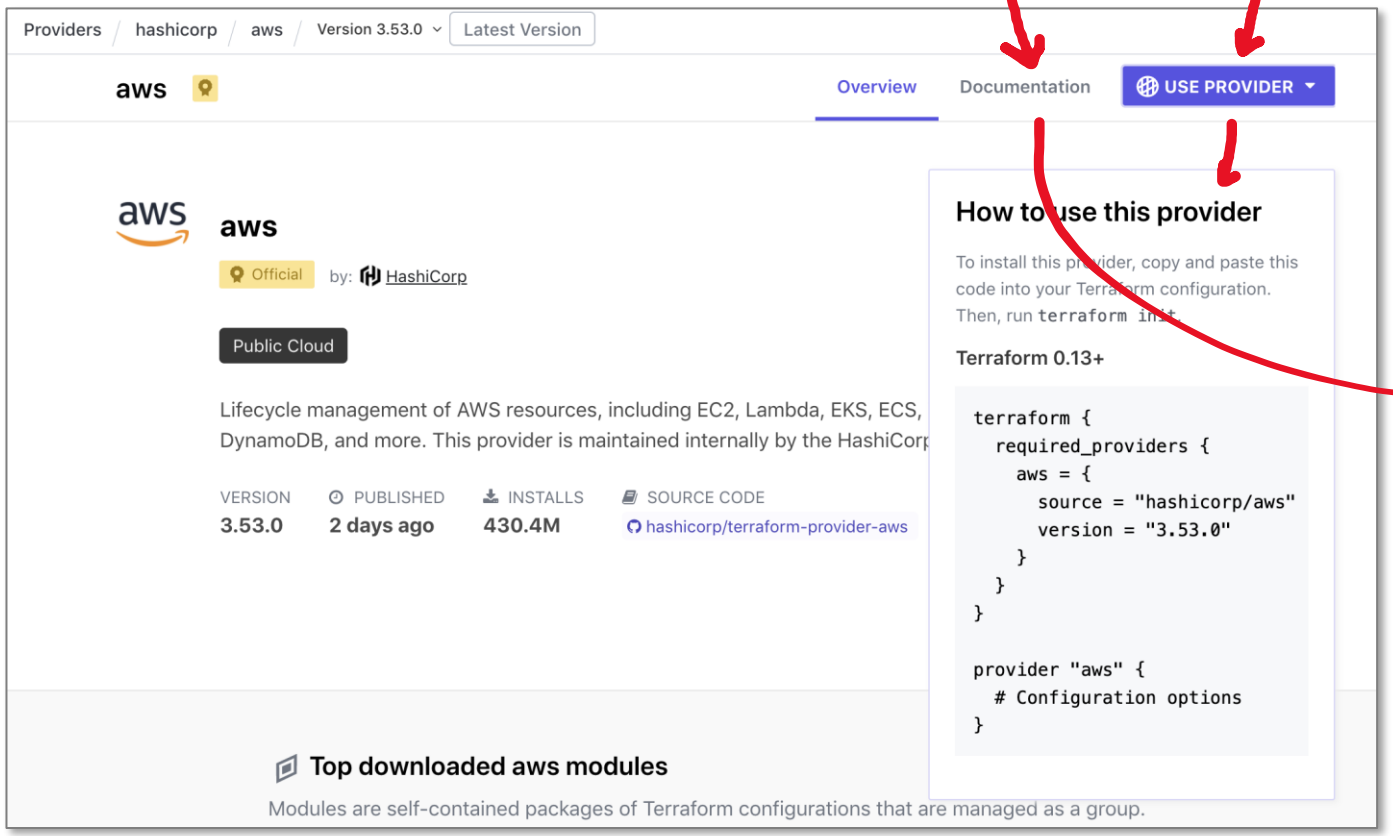
A module is a group of configuration files that provide common configuration functionality.

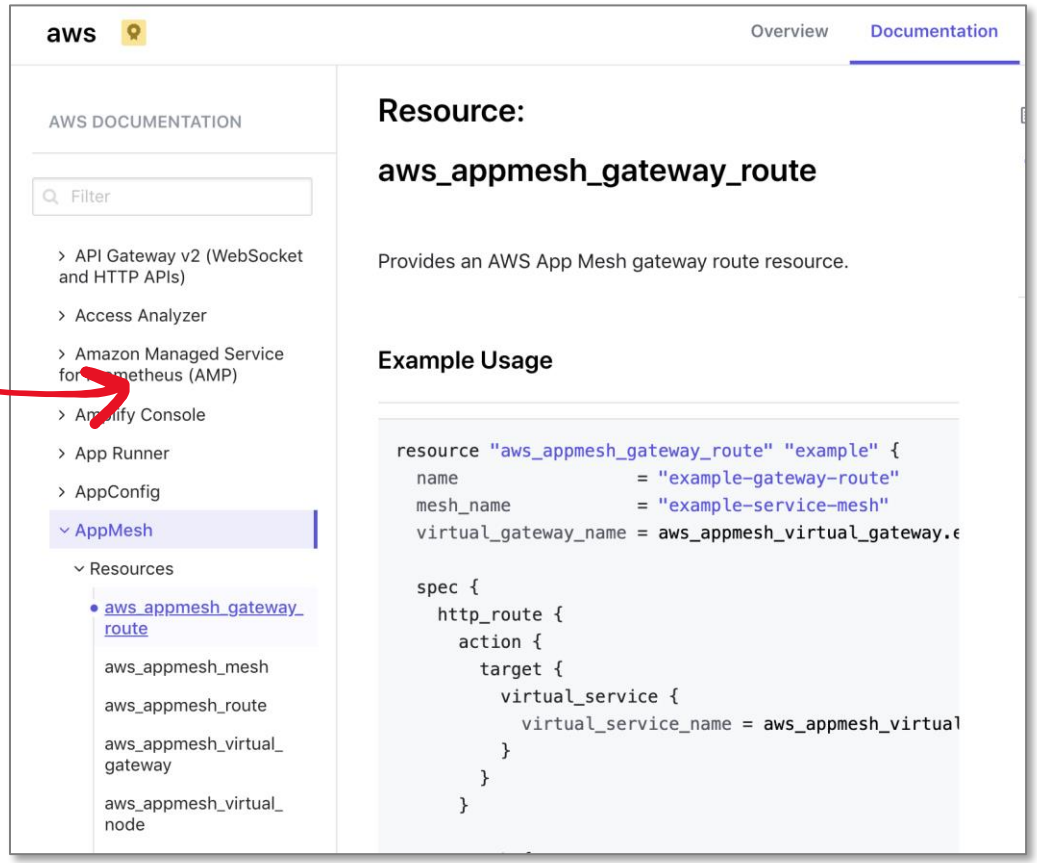
- Enforces best practices
- reduce the amount of code
- Reduce time to develop scripts

Everything published to Terraform Registry is **public-facing**

Terraform Registry – Providers

You can easily find **documentation** and **code sample** for providers



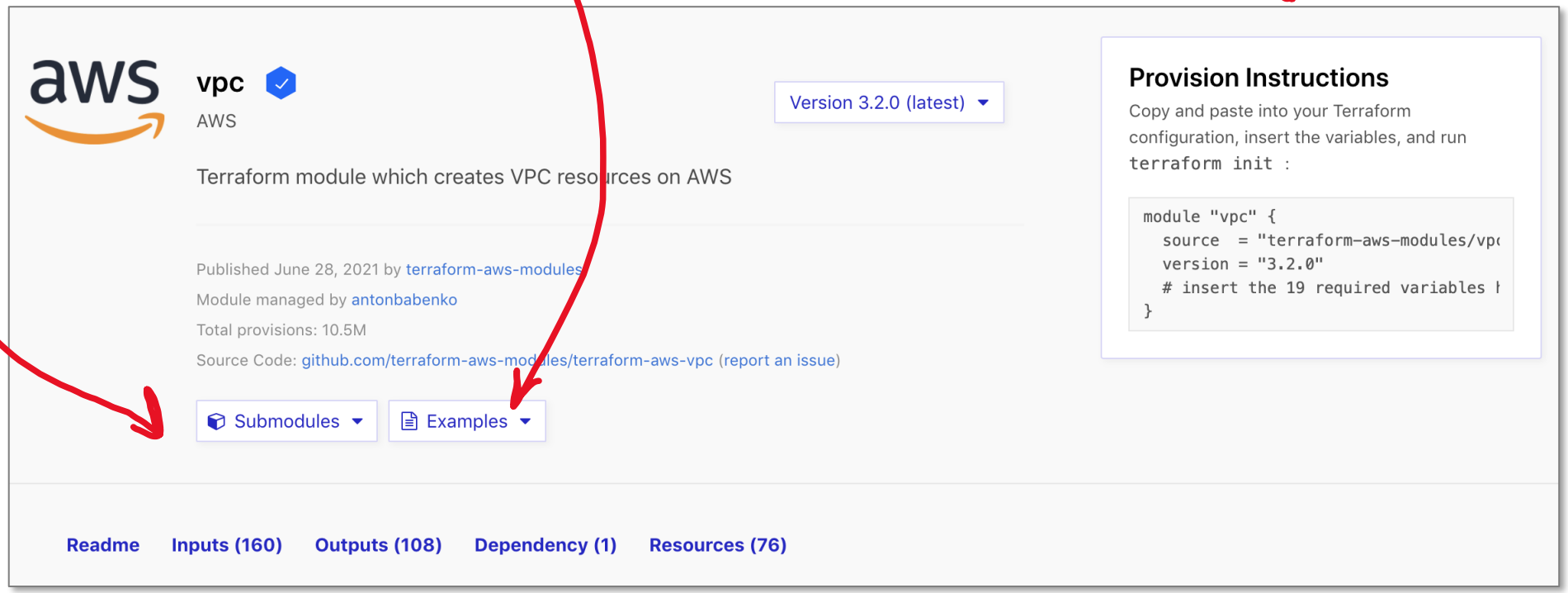


Terraform Registry

Lots of examples for common use cases

You can easily grab the **module code**:

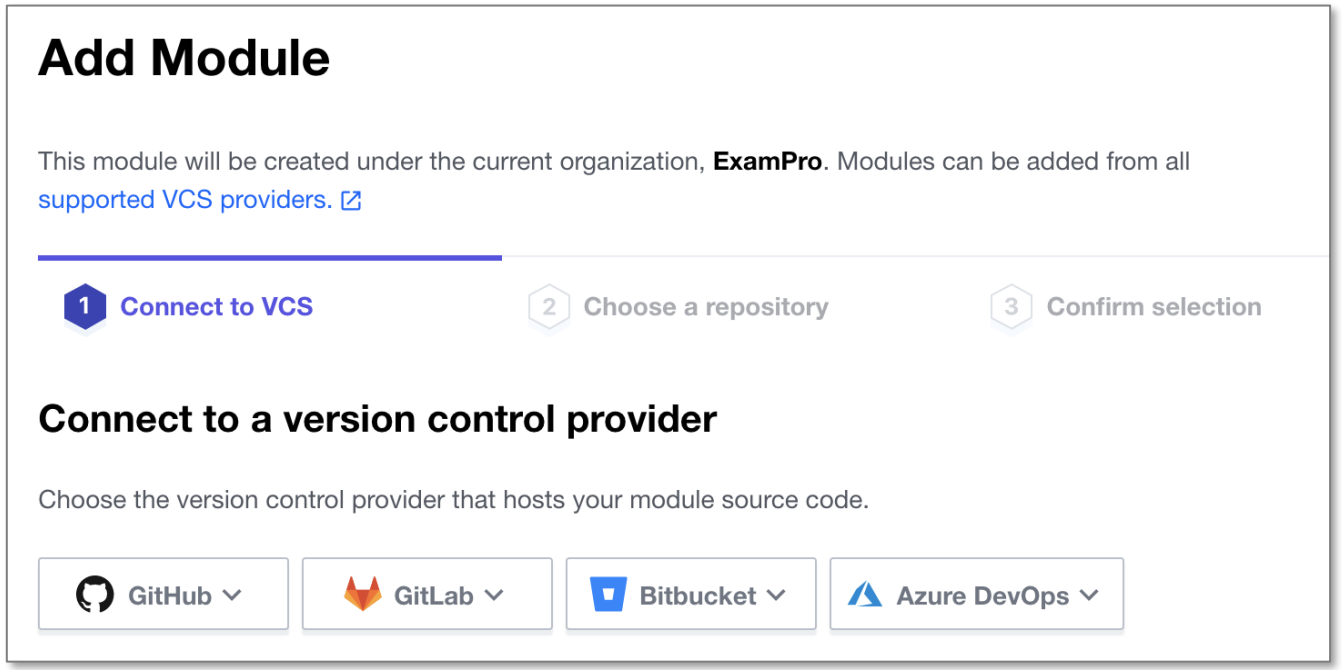
A list of dependent modules



Terraform Cloud – Private Registry

Terraform Cloud allows you to **publish private modules** for your Organization within the **Terraform Cloud Private Registry**

When creating a module you need to connect to a Version Control System (VCS) and choose a repository



Terraform Providers Command

Get a list of the current providers you are using

```
terraform providers
Providers required by configuration:
.
├── provider[registry.terraform.io/hashicorp/azurerm] 2.72.0
├── provider[registry.terraform.io/hashicorp/google] 3.80.0
├── provider[registry.terraform.io/hashicorp/aws] 3.54.0
├── module.network
│   └── provider[registry.terraform.io/hashicorp/azurerm]
└── module.linuxservers
    ├── provider[registry.terraform.io/hashicorp/azurerm]
    ├── provider[registry.terraform.io/hashicorp/random]
    └── module.os
```

Terraform Provider Configuration

```
provider "aws" {
  alias  = "west"
  region = "us-west-2"
}
```

Set an alternative provider

How to reference an alias provider

```
resource "aws_instance" "foo" {
  provider = aws.west
  # ...
}
```

How to set alias provider for a parent module

```
terraform {
  required_providers {
    mycloud = {
      source  = "mycorp/mycloud"
      version = "~> 1.0"
      configuration_aliases = [ mycloud.alternate ]
    }
  }}
```

How to set alias provider for a child module

```
module "aws_vpc" {
  source = "./aws_vpc"
  providers = {
    aws = aws.west
  }
}
```

Terraform Modules

A Terraform module is a group of configuration files that provide common configuration functionality.

- Enforces best practices
- reduce the amount of code
- Reduce time to develop scripts

AWS Provider (not a module)

If you had to create a VPC you would have specific many networking resources.

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

  tags = {
    Name = "main"
  }
}

resource "aws_subnet" "main" {
  vpc_id     = aws_vpc.main.id
  cidr_block = "10.0.1.0/24"

  tags = {
    Name = "Main"
  }
}

# ....
```

AWS VPC Module

Using a module you can use a a shorthand Domain Specific Language (DSL) That will reduce the amount of work.

```
module "vpc" {
  source = "terraform-aws-modules/vpc/aws"

  name = "my-vpc"
  cidr = "10.0.0.0/16"

  azs             = ["eu-west-1a", "eu-west-1b", "eu-west-1c"]
  private_subnets = ["10.0.1.0/24", "10.0.2.0/24", "10.0.3.0/24"]
  public_subnets  = ["10.0.101.0/24", "10.0.102.0/24", "10.0.103.0/24"]

  enable_nat_gateway = true
  enable_vpn_gateway = true

  tags = {
    Terraform = "true"
    Environment = "dev"
  }
```

Modules: Imagine clicking a wizard that creates many cloud resources e.g. VPC Wizard

Terraform Modules

Azure VM via the **Azure Provider**

Azure VM via a **Compute and Network Module**

```
resource "azurerm_resource_group" "example" {
  name     = "example-resources"
  location = "West Europe"
}

module "linuxservers" {
  source              = "Azure/compute/azurerm"
  resource_group_name = azurerm_resource_group.example.name
  vm_os_simple        = "UbuntuServer"
  public_ip_dns       = ["linsimplevmips"]
  vnet_subnet_id      = module.network.vnet_subnets[0]

  depends_on = [azurerm_resource_group.example]
}

module "network" {
  source              = "Azure/network/azurerm"
  resource_group_name = azurerm_resource_group.example.name
  subnet_prefixes     = ["10.0.1.0/24"]
  subnet_names        = ["subnet1"]

  depends_on = [azurerm_resource_group.example]
}

output "linux_vm_public_name" {
  value = module.linuxservers.public_ip_dns_name
}
```

Terraform Providers – The Fine Line

The Fine Line is understanding **the granularities of responsibility** between Terraform Infrastructure as Code and Third-Party Configuration Management.

When you have a Postgres Database which is a typical resource compared to a cloud service like an Amazon S3
Who should automate what?

A Terraform Provisioner could be using Ansible

Terraform

providers	modules	provisioners
Create a Postgres Databases	**Create a Postgres User**	**Stage Seed Data**

Entities. If you want governance or asset resource management

A task done one time to setup the database

Ansible (Third Party Configuration Management Tool)

- **Backup tables to Datawarehouse**
- **Truncate daily tables**

Repeatable tasks for on-going maintenance

Terraform Language

Terraform files contain the configuration information about **providers** and **resources**.

Terraform files end in the extension of **.tf** or either **.tf.json**
Terraform files are written in the **Terraform Language** and is the extension of HCL

Terraform language consists of only a few basic elements:

- **Blocks —** containers for other content, represent an object
  - block type — can have zero or more labels and a body
  - block label — name of a block
- **Arguments —** assign a value to a name
  - They appear within blocks
- **Expressions —** represent a value, either literally or by referencing and combining other values
  - They appear as values for arguments, or within other expressions.

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

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

You might come across HashiCorp Configuration Language (HCL) this is the low-level language for both the Terraform Language and alternative JSON syntax

Hashicorp Configuration Files – Syntax

Arguments

An *argument* assigns a value to a particular name

- arguments appear within blocks

```
image_id = "abc123"
```

Blocks

A block is a container for other content

```
resource "aws_instance" "example"
{
  ami = "abc123"

  network_interface {
    # ...
  }
}
```

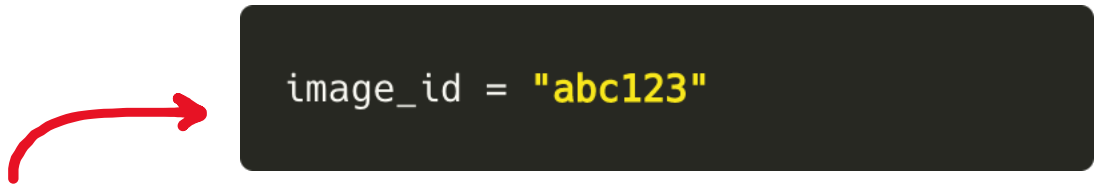
Block has a type **"resource"** and it expects two Block Labels

A block can contain **nested** blocks

different types have different number of required labels.

The brackets { } indicate the block **body**

Hashicorp Configuration Files – Syntax



Identifiers, uniquely identify objects of constructs in a Terraform Language:

- Argument names
- Block type names
- names Terraform-specific constructs eg. resources, input variables

Identifiers can contain:

- letters eg. abc
- digits eg. 012
- underscores eg. _
- hyphens eg. -

The first letter of an Identifiers cannot be a digit

Hashicorp Configuration Files – Syntax

Terraform language supports **3 different syntaxes** for comments

```
#  this is also a single line comment
```

```
// this is a single line comment
```

```
/*
This
is a
multiple comment
*/
```

This is the default comment style.
Formatting tools may transform // to #
eg. terraform fmt

Terraform configuration files must always be **UTF-8 encoded**
Terraform accepts configuration files with either Unix-style line endings (LF) or Windows-style line endings (CR + LF)

Hashicorp Configuration Files – Alternate JSON Syntax

Terraform also supports an *alternative syntax* that is JSON-compatible
Terraform expects JSON syntax files to be named with **.tf.json**

This syntax is useful when generating portions of a configuration programmatically, since existing JSON libraries can be used to prepare the generated configuration files.

Example of **JSON syntax**

Terraform Settings

The special **terraform configuration block type** eg. **terraform { … }**
is used to configure some behaviors of Terraform itself

In Terraform settings we can specify:

- **required_version**
  - The expected version of terraform
- **required_providers**
  - The providers that will be pull during an terraform init
- **experiments**
  - Experimental language features, that the community can try and provide feedback
- **provider_meta**
  - module-specific information for providers

```
terraform {
  required_providers {
    aws = {
      version = ">= 2.7.0"
      source = "hashicorp/aws"
    }
  }
}
```

HashiCorp Configuration Language

HCL is an open-source toolkit for creating **structured configuration languages** that are both human and machine friendly, for use with command-line tools

github.com/hashicorp/hcl

HashiCorp Configuration Language (HCL)

- Terraform Language (.tf)
- Packer Template (.pkr.hcl)
- Vault Policies (no extension)
- Boundary Controllers and Workers (.hcl)
- Consul Configuration (.hcl)
- Waypoint Application Configuration(.hcl)
- Nomad Job Specifications (.nomad)
- Shipyard Blueprint (.hcl)

eg. Dynamic Blocks, For Each…

- Sentinel Policies ✗

Doesn't use HCL but its own ACL custom language

Input Variables

Input variables (aka variables or Terraform Variables)
are **parameters** for Terraform modules

Variables are defined
via **variable blocks.**

```
variable "image_id" {
  type = string
}

variable "availability_zone_names" {
  type    = list(string)
  default = ["us-west-1a"]
}

variable "docker_ports" {
  type = list(object({
    internal = number
    external = number
    protocol = string
  }))
  default = [
    {
      internal = 8300
      external = 8300
      protocol = "tcp"
    }
  ]
}
```

You can declare variables in either:

- The root module
- The child modules

Default	A default value which then makes the variable optional
type	This argument specifies what value types are accepted for the variable
Description	This specifies the input variable's documentation
Validation	A block to define validation rules, usually in addition to type constraints
Sensitive	Limits Terraform UI output when the variable is used in configuration

Variable Definitions Files

A variable definitions file allows you to set the values for multiple variables at once.
Variable definition files are named .tfvars or tfvars.json

By default **terraform.tfvars** will be autoloaded when included in the root of your project directory

```
image_id = "ami-abc123"
availability_zone_names = [
  "us-east-1a",
  "us-west-1c",
]
```

Variable Definition Files use the Terraform Language.

Variables via Environment Variables

A variable value can be defined by Environment Variables

Variable starting with **TF_VAR_**<name> will be read and loaded

```
export TF_VAR_image_id=ami-abc123
```

Loading Input Variables

Default Autoloaded Variables file
terraform.tfvars

When you create a named **terraform.tfvars** file it will be automatically loaded when running terraform apply

Additional Variables Files (not autoloaded)
my_variables.tfvars

You can create additional variables files eg. dev.tfvars, prod.tfvars
They will not be autoloaded (you'll need to specific them in via command line)

Additional Variables Files (autoloaded)
my_variables.**auto.**tfvars

If you name your file with auto.tfvars it will always be loaded

Specify a Variables file via Command Line
-var-file dev.tfvars

You can specific variables inline via the command line for individual overrides

Inline Variables via Command Line
-var ec2_type="t2.medium"

You can specific variables inline via the command line for individual overrides

Environment Variables
TF_VAR_my_variable_name

Terraform will watch for environment variables that begin with TF_VAR_ and apply those as variables

Variable Definition Precedence

You can override variables via many files and commands
The definition precedence is the order in which Terraform will read variables and as it goes down the list it will override variable.

- Environment Variables
- terraform.tfvars
- terraform.tfvars.json
- *.auto.tfvars or *.auto.tfvars.json
- -var and -var-file

Output Values

Output Values are computed values after a Terraform apply is performed.
Outputs allows you:

- to obtain information after resource provisioning e.g. public IP address
- output a file of values for programmatic integration
- Cross-reference stacks via outputs in a state file via terraform_remote_state

You can optionally provide a description

You can mark the output as sensitive so it does not show in output of your Terminal

```
output "db_password" {
  value       = aws_db_instance.db.password
  description = "The password for logging in to the database."
  sensitive   = true
}
```

Sensitive outputs will still be visible within the statefile.

Output Values

To print all the outputs for a statefile use the **terraform output**

```
$ terraform output
lb_url = "http://lb-5YI-project-alpha-dev-2144336064.us-east-1.elb.amazonaws.com/"
vpc_id = "vpc-004c2d1ba7394b3d6"
web_server_count = 4
```

Print a specific output with **terraform output <name>**

```
$ terraform output lb_url
"http://lb-5YI-project-alpha-dev-2144336064.us-east-1.elb.amazonaws.com/"
```

Use the **–json** flag to get output as json data.

```
$ terraform output -json
{
  "db_password": {
    "sensitive": true,
    "type": "string",
    "value": "notasecurepassword"
  },
```

Use the **–raw** flag to preserve quotes for strings

```
$ curl $(terraform output -raw lb_url)
<html><body><div>Hello, world!</div></body></html>
```

Local Values

A local value (locals) **assigns a name to an expression**, so you can **use it multiple times within a module** without repeating it.

Static value

computed values

Locals are set using the **locals block**

You can defined **multiple** locals blocks

You can **reference locals within locals**

```
locals {
  service_name = "forum"
  owner        = "Community Team"
}

locals {
  # Ids for multiple sets of EC2 instances, merged together
  instance_ids = concat(aws_instance.blue.*.id, aws_instance.green.*.id)
}

locals {
  # Common tags to be assigned to all resources
  common_tags = {
    Service = local.service_name
    Owner   = local.owner
  }
}
```

Local Values

Once a local value is declared, you can reference it in expressions as **local.<NAME>.**

```
resource "aws_instance" "example" {
  # ...

  tags = local.common_tags
}
```

When you referencing **you use the singular "local"**

Locals help can help DRY up your code.
It is best practice to **use locals sparingly** since it Terraform is intended to be declarative and overuse of locals can make it difficult to determine what the code is doing.

Data Sources

Data sources allow Terraform **use information defined outside of Terraform**, defined by another separate Terraform configuration, or modified by functions.

You specific what kind of external resource you want to select

You use filters to narrow down the selection

```
data "aws_ami" "web" {
  filter {
    name   = "state"
    values = ["available"]
  }

  filter {
    name   = "tag:Component"
    values = ["web"]
  }

  most_recent = true
}
```

```
resource "aws_instance" "web" {
  ami           = data.aws_ami.web.id
  instance_type = "t1.micro"
}
```

You use **data.** to reference data sources

References to Named Values

Named Values are **built-in expressions** to **reference various values** such as:

Resources	<Resource Type>.<Name> e.g. aws_instance.my_server
Input variables	**var**.<Name>
Local values	**local**.<Name>
Child module outputs	**module**.<Name>
Data sources	**data**.<Data Type>.<Name>

Filesystem and workspace info

- path.module - path of the module where the expression is placed
- path.root - path of the root module of the configuration
- path.cwd - path of the current working directory
- terraform.workspace – name of the currently selected workspace

Block-local values (within the body of blocks)

- count.**index** (when you use the count meta argument)
- each.**key** / each.**value** (when you use the for_each meta argument)
- **self**.<attribute> - self reference information within the block (provisioners and connections)

Named values resemble the attribute notation for map (object) values but are not objects and do not act as objects. You cannot use square brackets to access attribute of Named Values like an object.

Resource Meta Arguments

Terraform language defines several meta-arguments, which can be used with any **resource type** **to change the behavior of resources**.

- **depends_on**, for specifying explicit dependencies
- **count**, for creating multiple resource instances according to a count
- **for_each**, to create multiple instances according to a map, or set of strings
- **provider**, for selecting a non-default provider configuration
- **lifecycle**, for lifecycle customizations
- **provisioner** and connection, for taking extra actions after resource creation

depends_on

The order of which resources are provisioned is important when resources depend on others before they are provisioned.

Terraform implicitly can determine the order of provision for resources but there may be some cases where it cannot determine the correct order.

depends_on allows you to explicitly specific a dependency of a resource.

```
resource "aws_iam_role_policy" "example" {
  name   = "example"
  role   = aws_iam_role.example.name
  # ...
}

resource "aws_instance" "example" {
  ami           = "ami-a1b2c3d4"
  instance_type = "t2.micro"
  iam_instance_profile = aws_iam_instance_profile.example
  depends_on = [
    aws_iam_role_policy.example,
  ]
}
```

count

When you are managing a pool of objects eg. a fleet of Virtual Machines you can use **count**.

Specify the **amount** of instances you want

```
resource "aws_instance" "server" {
  count = 4 # create four similar EC2 instances

  ami           = "ami-a1b2c3d4"
  instance_type = "t2.micro"

  tags = {
    Name = "Server ${count.index}"
  }
}
```

Get the current count **value** (index) via **count.index**

This value starts at 0

Count can accept **numeric expressions**:

- Must be whole number
- Number must be known before configuration

```
resource "aws_instance" "server" {
  # Create one instance for each subnet
  count = length(var.subnet_ids)
  # ....
```

for_each

for_each is similar to count for managing multiple related objects.
But you can iterate over a map for more dynamic values.

```
resource "azurerm_resource_group" "rg" {
  for_each = {
    a_group = "eastus"
    another_group = "westus2"
  }
  name     = each.key
  location = each.value
}
```

With a map:
each.key – print out the current key
each.value – print out the current value

```
resource "aws_iam_user" "the-accounts" {
  for_each = toset( ["Todd", "James", "Alice", "Dottie"] )
  name     = each.key
}
```

With a list:
each.key – print out the current key

Resource Behaviour

When you execute an execution order via Terraform Apply it will perform one of the following to a resource:

Create

- resources that exist in the configuration but are not associated with a real infrastructure object in the state.

Destroy

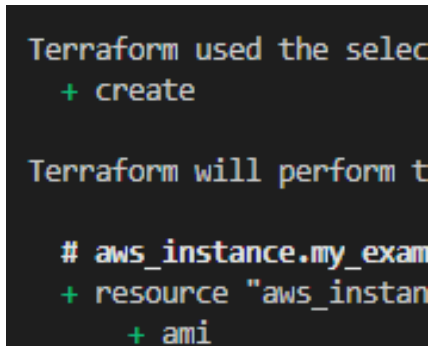
- resources that exist in the state but no longer exist in the configuration.

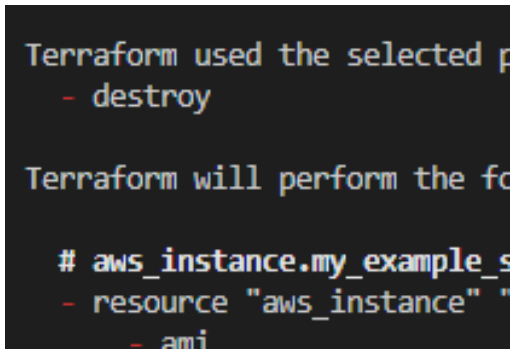
Update in-place

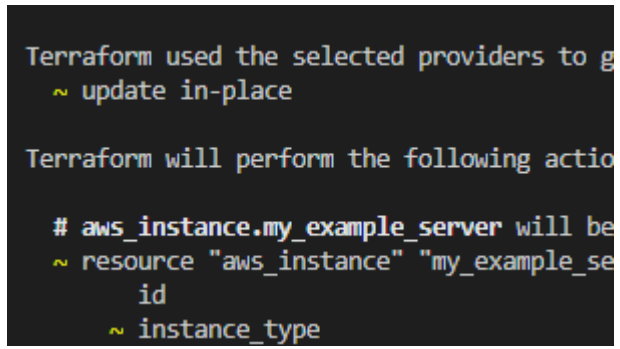
- resources whose arguments have changed.

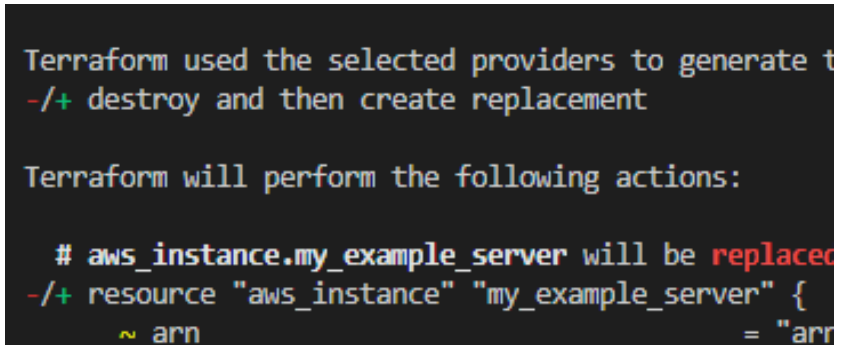
Destroy and re-create

- resources whose arguments have changed but which cannot be updated in-place due to remote API limitations.









lifecycle

Lifecycle block allows you to change what happens to resource e.g. create, update, destroy.
Lifecycle blocks are nested within resources

create_before_destroy (bool)
When replacing a resource first create the new resource before deleting it (the default is destroy old first)

prevent_destroy (bool)
Ensures a resource is not destroyed

ignore_changes (list of attributes)
Don't change the resource (create, update, destroy) the resource if a change occurs for the listed attributes.

```
resource "azurerm_resource_group" "example" {
  # ...

  lifecycle {
    create_before_destroy = true
  }
}
```

Resource Providers and Alias

If you need to override the default provider for a resource you can create alternative provider with **alias**

You reference the alias under in the attribute **provider** for a resource.

```
# default configuration
provider "google" {
  region = "us-central1"
}

# alternate configuration, whose alias is "europe"
provider "google" {
  alias  = "europe"
  region = "europe-west1"
}

resource "google_compute_instance" "example" {
  # This "provider" meta-argument selects the google provider
  # configuration whose alias is "europe", rather than the
  # default configuration.
  provider = google.europe

  # ...
}
```

Introduction to Terraform Expressions

Expressions are used to **refer to** or **compute values** within a configuration.

Terraform Expressions is a large topic, and we'll be covering:

- Types and Values
- Strings and Templates
- References to Values
- Operators
- Function Calls
- Conditional Expressions
- For Expressions
- Splat Expressions
- Dynamic Blocks
- Type Constraints
- Version Constraints

Types and Values

The result of an expression is a **value**. All values have a **type**

primitive types

string

```
ami = "ami-830c94e3"
```

number

```
size = 6.283185
```

bool

```
termination_protection = true
```

no type

null

```
endpoint = null
```

null represents *absence* or *omission*
*when you want to use the **underlying default** of a provider's resource configuration option*

complex/structural/collection types

list (tuple)

```
regions = ["us-east-1a", "us-east-1b"]
```

map (object)

```
tags = {env = "Production", priority = 3}
```

Strings

When quoting strings you use **double quotes** eg.

```
"hello"
```

Double quoted strings can interpret **escape sequences**.

\n	Newline
\r	Carriage Return
\t	Tab
\"	Literal quote (without terminating the string)
****	Literal backslash
\uNNNN	Unicode character from the basic multilingual plane
\UNNNNNNNN	Unicode character from supplementary planes

special escape sequences:

$${	Literal ${, without beginning an interpolation sequence.
%%{	Literal %{, without beginning a template directive sequence.

Terraform also supports a "heredoc" style.
Heredoc is a UNIX style multi-line string:

```
<<EOT
hello
world
EOT
```

Strings Templates

String interpolation allows you to evaluate an expression between the markers eg. **${ .... }** and coverts it to a string.

```
"Hello, ${var.name}!"
```

String directive allows you to evaluate an conditional logic between the markers eg. **%{ .... }**

```
"Hello, %{ if var.name != "" }${var.name}%{ else }unnamed%{ endif }!
```

You can use interpolation or directives within a **HEREDOC**

```
<<EOT
%{ for ip in aws_instance.example.*.private_ip }
server ${ip}
%{ endfor }
EOT
```

```
<<EOT
%{ for ip in aws_instance.example.*.private_ip ~}
server ${ip}
%{ endfor ~}
EOT
```

You can **stripe whitespacing** that would normally be left by directive blocks by providing a trailing tilde eg. **~**

Expressions – Operators

Operators are **mathematical operations** you can preform to numbers within expressions

- Multiplication a * b
- Division a / b
- Modulus a % b
- Addition a + b
- Subtraction a – b
- Flip to Negative (* -1) -a
- Equals a == b
- Does not Equal a != b
- Less Than a < b
- Less Then or Equal a <= b
- Greater Then a > b
- Greater Then or Equal a >= b
- Or a || b
- And a && b
- Flip Boolean !a

Conditional Expressions

Terraform support ternary if else conditions.

`condition ? true_val : false_val`

```
var.a != "" ? var.a : "default-a"
```

The return type for it the if and else must be the same type:

```
var.example ? tostring(12) : "hello"
```

Expressions – For Expressions

For expressions allows you to iterative over a complex type and apply transformations

A for expression can accept as input **a list, a set, a tuple, a map, or an object**.

Uppercase each string in the provided list

```
[for s in var.list : upper(s)]
```

For map you can get: **Key** and **value**

```
[for k, v in var.map : length(k) + length(v)]
```

For a list you can get the **index**

```
[for i, v in var.list : "${i} is ${v}"]
```

Square braces **[ ]** returns a tuple

```
[for s in var.list : upper(s)]
```

→ [“HELLO”, WOLRD”]

Curly braces **{ }** returns an object

```
{for s in var.list : s => upper(s)}
```

→ { hello = “HELLO”, world = “WORLD”}

Expressions – For Expressions

An if statement can be used in a for expression to filter / reduce the amount of elements returned.

```
[for s in var.list : upper(s) if s != ""]
```

Implicit Element Ordering on Conversion

Since Terraform can convert a unordered type (maps objects and sets) to a ordered type (list and tuples) it will need to choose an implied ordering.

- **Maps and Objects** – stored by key A-Z
- **Sets of Strings** – stored by string A-Z
- **Everything else** – arbitrary ordering

Expressions – Splat Expressions

A **splat** expression provides a **shorter expression** for **for expressions**

What is a splat operator?
A splat operator is represented by an asterisk (*), it originates from the ruby language
Splats in Terraform are used to rollup or soak up a bunch of iterations in a *for expression*

For lists, sets and tuples

```
[for o in var.list : o.id]
[for o in var.list : o.interfaces[0].name]
```

Can be written list this

```
var.list[*].id
var.list[*].interfaces[0].name
```

Expressions – Splat Expressions

Splat expressions have a special behavior when you apply them to **lists**

- If the value is anything other than a null value then the splat expression will transform it into a single-element list
- If the value is *null* then the splat expression will return an empty tuple.

This behavior is useful for modules that accept optional input variables whose default value is null to represent the absence of any value to adapt the variable value to work with other Terraform language features that are designed to work with collections.

```
variable "website" {
  type = object({
    index_document = string
    error_document = string
  })
  default = null
}

resource "aws_s3_bucket" "example" {
  # ...

  dynamic "website" {
    for_each = var.website[*]
    content {
      index_document = website.value.index_document
      error_document = website.value.error_document
    }
  }
}
```

Dynamic Blocks

Dynamic blocks allows you **dynamically construct repeatable nested blocks**

Lets say you need to create a bunch of ingress rules for an EC2 Security Group.

Define **objects** in locals:

Set **dynamic** block and utilized for_each block

```
locals {
  ingress_rules = [{
    port        = 443
    description = "Port 443"
  },
  {
    port        = 80
    description = "Port 80"
  }]
}

resource "aws_security_group" "main" {
  name   = "sg"
  vpc_id = data.aws_vpc.main.id

  dynamic "ingress" {
    for_each = local.ingress_rules

    content {
      description = ingress.value.description
      from_port   = ingress.value.port
      to_port     = ingress.value.port
      protocol    = "tcp"
      cidr_blocks = ["0.0.0.0/0"]
    }
  }
}
```

Version Constraints

Terraform utilizes Semantic Versioning for specifying Terraform, Providers and Modules versions

Semantic Versioning is **open-standard** on how to define versioning for software management e.g. **MAJOR.MINOR.PATCH**

2.0.0 2.0.0-rc.2 2.0.0-rc.1 1.0.0 1.0.0-beta

semver.org

1.MAJOR version when you make incompatible API changes,
2.MINOR version when you add functionality in a backwards compatible manner, and
3.PATCH version when you make backwards compatible bug fixes.
Additional labels for pre-release and build metadata are available as extensions to the MAJOR.MINOR.PATCH format.

A **version constraint** is a string containing one or more conditions, separated by commas.

- = or no operator. Match exact version number e.g. “1.0.0”, “=1.0.0”
- != Excludes an exact version number e.g. “!=1.0.0”
- > >= < <= Compare against a specific version e.g. “>= 1.0.0”
- ~> Allow only the rightmost version (last number) to increment e.g. ~> 1.0.0”

Progressive Versioning

Progressive Versioning is the practice of using the latest version to keep a proactive stance of security, modernity and development agility

Practicing Good Hygiene

By being up to date you are pushing left things you will need to fix to stay compatible.
You will have to deal with smaller problems instead of dealing with a big problem later on

Run Nightly Builds of your golden images or terraform plan as a warning signal to budget the time to improve for outage.

A nightly build is an automated workflow that occurs at night when developers are asleep. If the build breaks because a change is required for the code, the developers will see this upon arrival in the morning and be able to budget accordingly.

Terraform State

What is State?

A particular condition of cloud resources at a specific time.

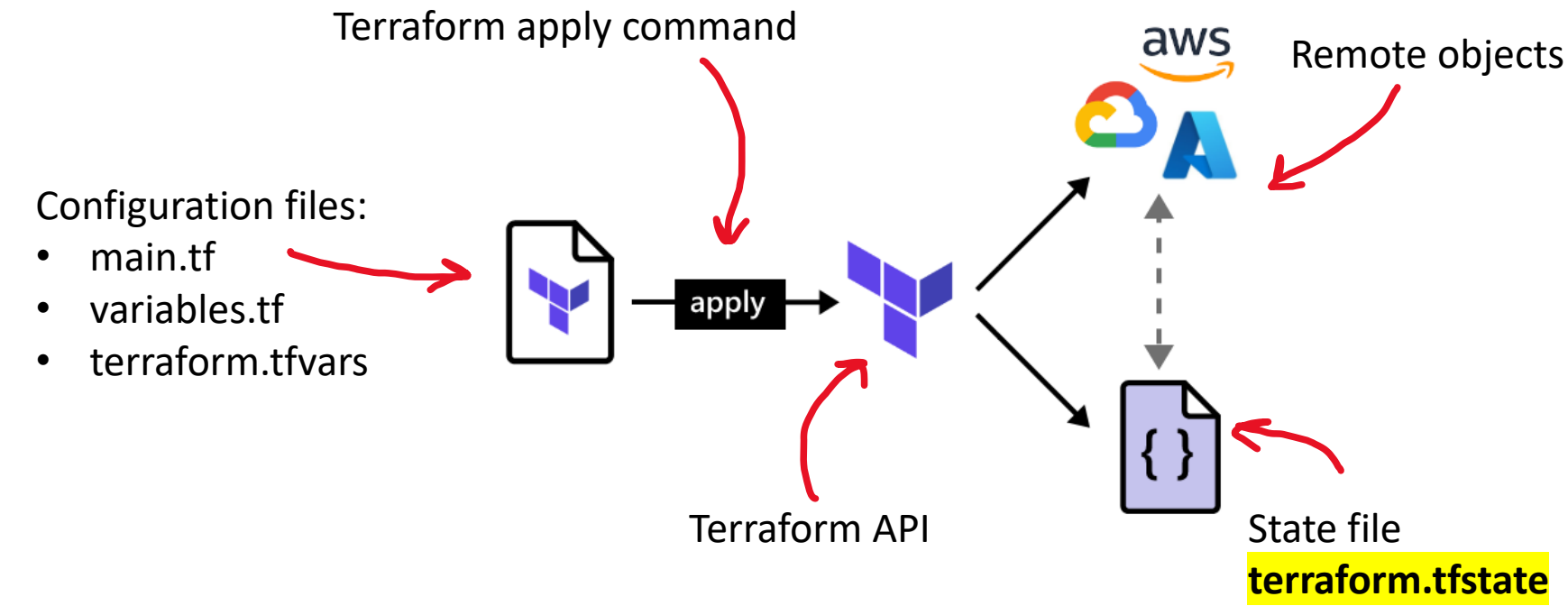
eg. We expect there to be a Virtual Machine running CentOS on AWS with a compute type of t2.micro.

How does Terraform preserve state?

When you provision infrastructure via Terraform it will create a state file named **terraform.tfstate**

This **state file is a JSON data structure** with a one-to-one mapping from **resource instances** to **remote objects**


```
{
  "version": 4,
  "terraform_version": "1.0.4",
  "serial": 1,
  "lineage": "7b78e5ae-eae4-de2e-95a2-2a947f5c
  "outputs": {},
  "resources": [
    {
      "mode": "managed",
      "type": "aws_instance",
      "name": "my_example_server",
      "provider": "provider[\"registry.terrafo
      "instances": [
        {
          "schema_version": 1,
          "attributes": {
            "ami": "ami-0c2b8ca1dad447f8a",
            "arn": "arn:aws:ec2:us-east-1:3184
            "associate_public_ip_address": tru
            "availability_zone": "us-east-1a"
```

Terraform State

terraform state **list**	List resources in the state
terraform state **mv**	Move an item in the state
terraform state **pull**	Pull current remote state and output to stdout
terraform state **push**	Update remote state from a local state
terraform state **replace-provider**	Replace provider in the state
terraform state **rm**	Remove instances from the state
terraform state **show**	Show a resource in the state

Terraform State Mv

terraform state mv allows you to:

- rename existing resources
- move a resource into a module
- move a module into a module

If you were to just rename a resource or move it to another module and run terraform apply Terraform will destroy and create the resource. State mv allows you to just change the reference so you can avoid a create and destroy action

Rename resource

```
terraform state mv packet_device.worker packet_device.helper
```

Move resource into module

```
terraform state mv packet_device.worker module.worker.packet_device.worker
```

Move module into a module

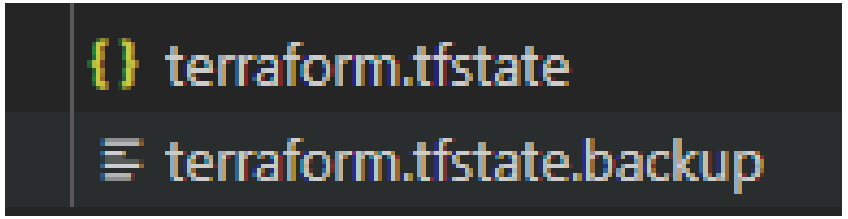
```
terraform state mv module.app module.parent.module.app
```

Terraform State Backups

All terraform state subcommands that *modify state* will write a backup file.

Read only commands will not modify state eg. list, show

Terraform will take the current state and store it in a file called **terraform.tfstate.backup**



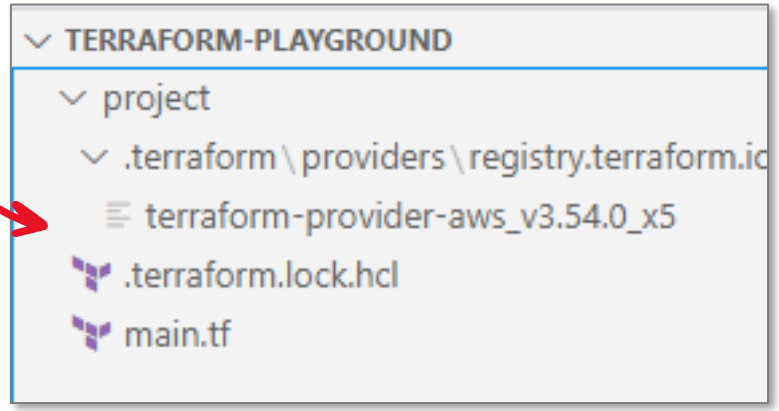
Backups cannot be disabled. This is by design to enforce best-practice for recovery

To get rid of the backup file you need to manually delete the files

terraform Init

terraform init initializes your terraform project by:

- Downloading plugin dependencies e.g. Providers and Modules
- Create a .terraform directory
- Create a dependency lock file to enforce expected versions for plugins and terraform itself.



Terraform init is generally the first command you will run for a new terraform project
If you modify or change dependencies run **terraform init** again to have it apply the changes

terraform init **-upgrade**	Upgrade all plugins to the latest version that complies with the configuration's version constraint
terraform init **-get-plugins-false**	Skip plugin installation
terraform init **-plugin-dir=PATH**	Force plugin installation to read plugins only from target directory
terraform init **–lockfile=MODE**	Set a dependency lockfile mode

Dependency lock file
.terraform.lock.hcl

State lock file
.terraform.tfstate.lock.hcl

terraform get

terraform get command is used to download and **update modules** in the root module.

When you're developer your own **Terraform Modules**
You may need to frequently pull updated modules but you do no want to initialize your state or pull new provider binaries.

Terraform Get is *lightweight* in this case because it only updates modules.

In most cases you want to use terraform init, with the exception of local module development

Writing and Modifying Terraform Code

Terraform has three CLI commands that **improve debugging configuration scripts**:

terraform **fmt**	rewrites Terraform configuration files to a standard format and style
terraform **validate**	validates the syntax and arguments of the Terraform configuration files in a directory
terraform **console**	an interactive shell for evaluating Terraform expressions

terraform fmt

This command applies a subset of the **Terraform language style conventions** along with other minor adjustments for readability

terraform fmt will by default look in the current directory and apply formatting to all .tf files.

adjusting spacing two spaces indent

```
provider "aws" {
    profile = "exampro"
    region  = "us-west-2"
}
```

```
provider "aws" {
  profile = "exampro"
  region  = "us-west-2"
}
```

syntax error

```
provider "aws"
{
  profile = "exampro"
  region  = "us-west-2"
}
```

```
Error: Invalid block definition

  on main.tf line 11:
  11: provider "aws"
  12: {

A block definition must have block
content delimited by "{" and "}",
starting on the same line as the block header
```

terraform fmt **--diff**

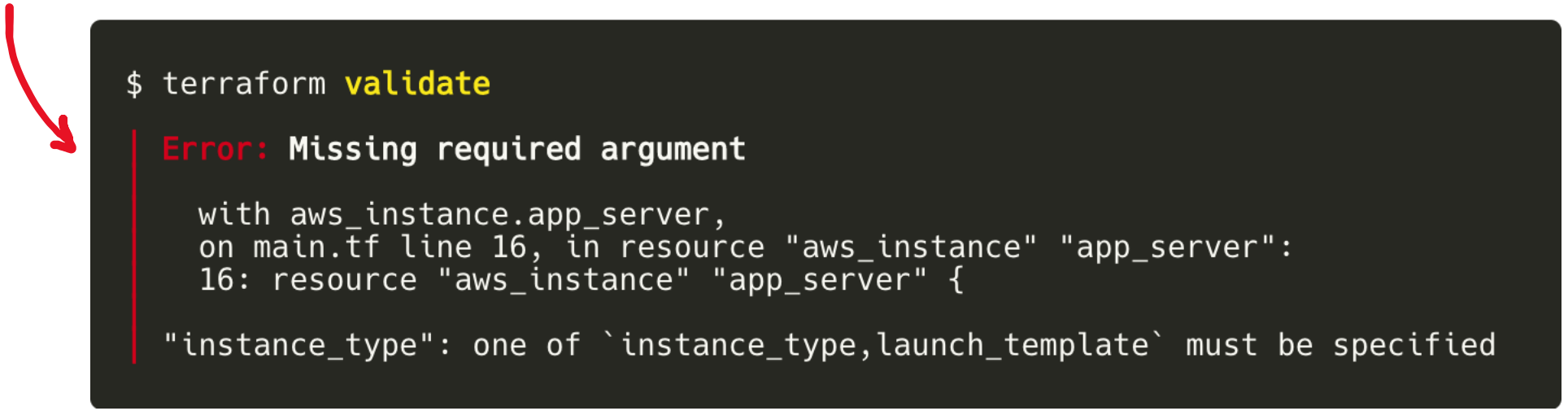
```
main.tf
--- old/main.tf
+++ new/main.tf
@@ -9,8 +9,8 @@
 }

 provider "aws" {
-    profile = "exampro"
-    region  = "us-west-2"
+  profile = "exampro"
+  region  = "us-west-2"
 }
```

terraform validate

Terraform Validate runs checks that verify whether a configuration is syntactically valid and internally consistent, regardless of any provided variables or existing state

Validate is useful for general verification of reusable modules, including correctness of attribute names and value types

We are missing an instance_type which is a required when lauching an AWS Virtual Machine



When you run terraform **plan** or terraform **apply**, validate will automatically be performed.

terraform console

Terraform console is an **interactive shell** where you can evaluate expressions.

We run a min command with the wrong arguments

We correct the argument error