



HashiCorp Terraform Associate Lecture Slides

These lecture slides are provided for personal and non-commercial use only

Please do not redistribute or upload these lecture slides elsewhere.

Good luck on your exam!

What is Infrastructure as Code?

The Problem with Manual Configuration

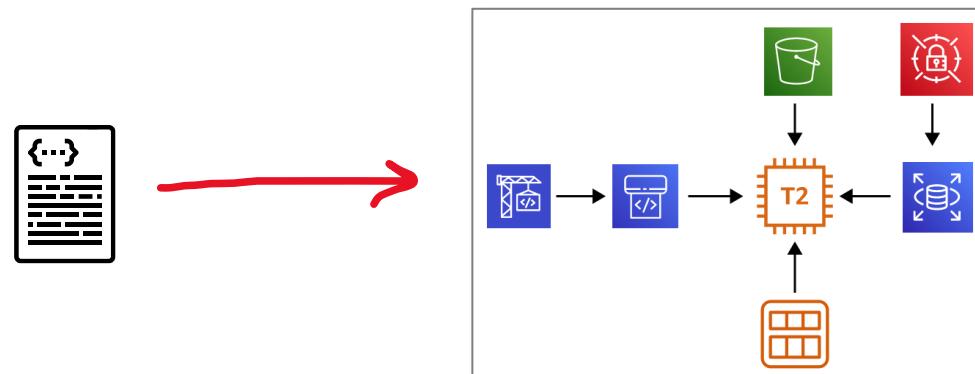
Manually configuring your cloud infrastructure allows you easily start using new service offerings to quickly prototype architectures however it comes with many downsides:

- 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. *Implicit*
- 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

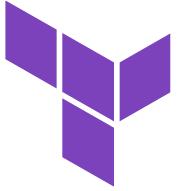
Many built-in templates for opinionated best practices



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.	Supports:	Imperative features is the utility of the entire feature set of the programming language.
In some cases you can add behaviour by extending the base language. E.g. CloudFormation Macros.	<ul style="list-style-type: none">• Loops (For Each)• Dynamic Blocks• Locals• Complex Data Structure<ul style="list-style-type: none">• Maps, Collections	

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

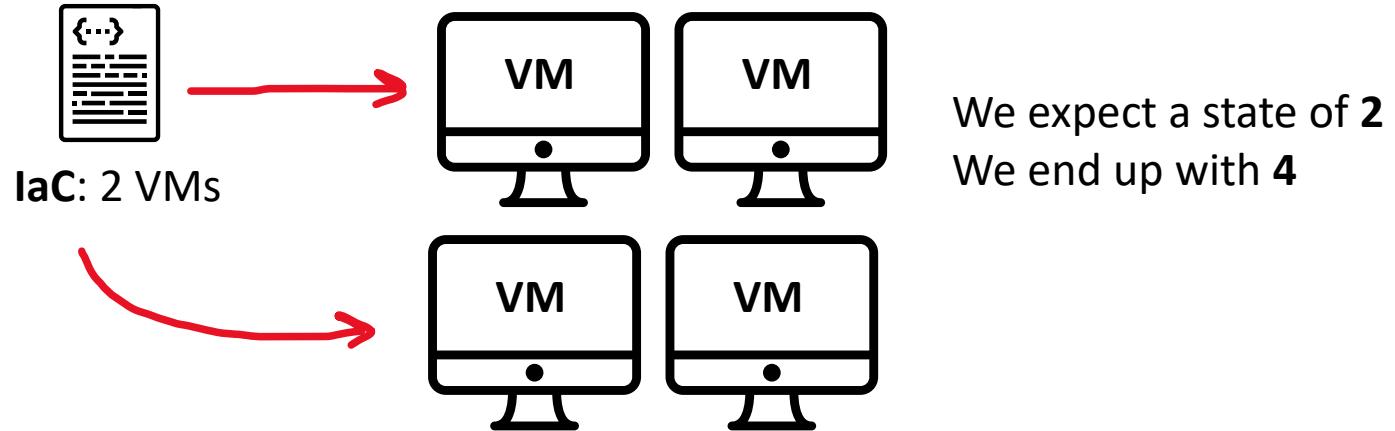
Sensibility

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

Non-Idempotent vs Idempotent

Non-Idempotent

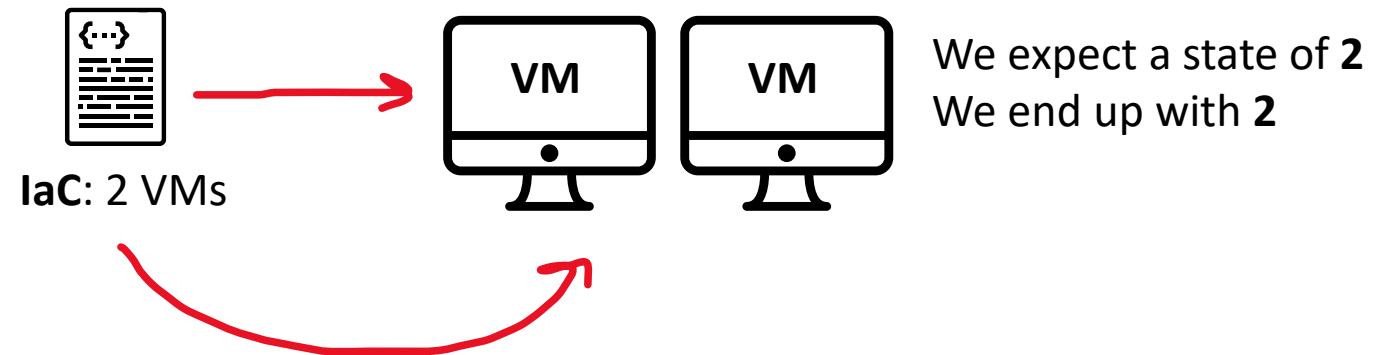
When I deploy my IaC config file it will **provision** and **launch** 2 virtual machines



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

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**

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 CodePipeline, Harness, Jenkins, Github Actions, CircleCI

Orchestration

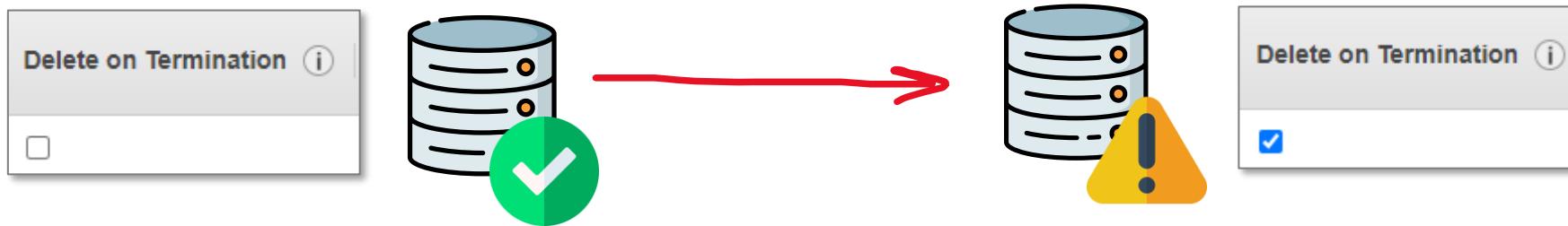
Orchestration is the act of coordinating multiple systems or services. Orchestration is a common term when working with microservices, Containers and Kubernetes. Orchestration could be Kubernetes, Salt, Fabric

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 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.



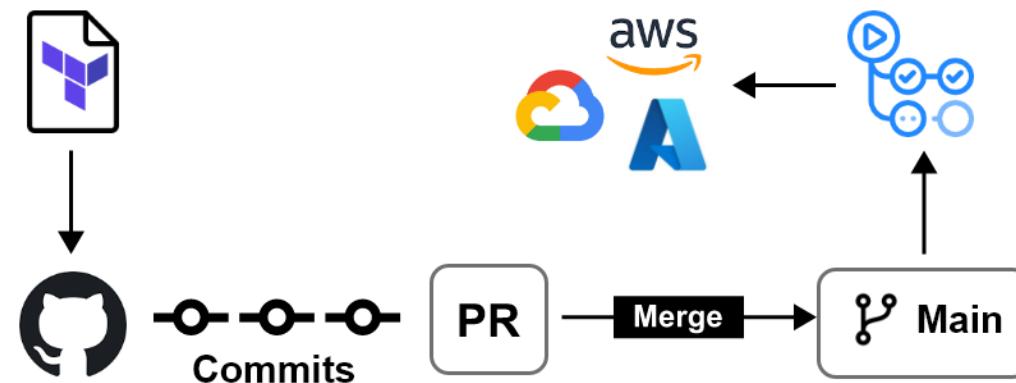
Packer

Golden Images

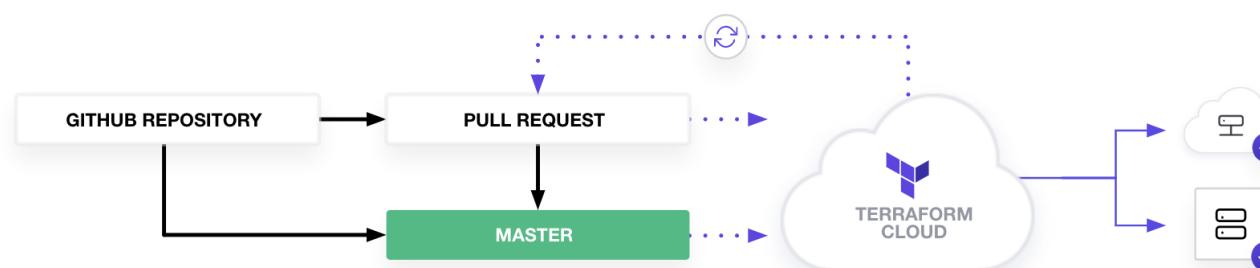
- Guarantee of 1-to-1 with your fleet
- Increased assurance of consistency, security
- Speeds up your deployments

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



TERRAFORM CLOUD AND GITHUB ACTIONS WORKFLOW



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**

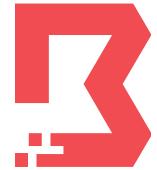
What is HashiCorp Cloud Platform (HCP)?

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

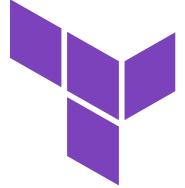
scheduling and deployment of tasks across worker nodes in a cluster



Packer

tool for building virtual-machine images for later deployment.

HashiCorp Products



Terraform

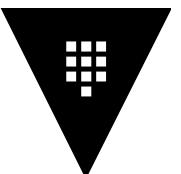
infrastructure as code software which enables provisioning and adapting virtual infrastructure across all major cloud provider

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

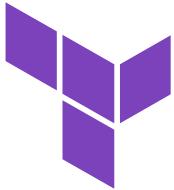
secrets management, identity-based access, encrypting application data and auditing of secrets for applications, systems, and users



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"  
    }  
}
```

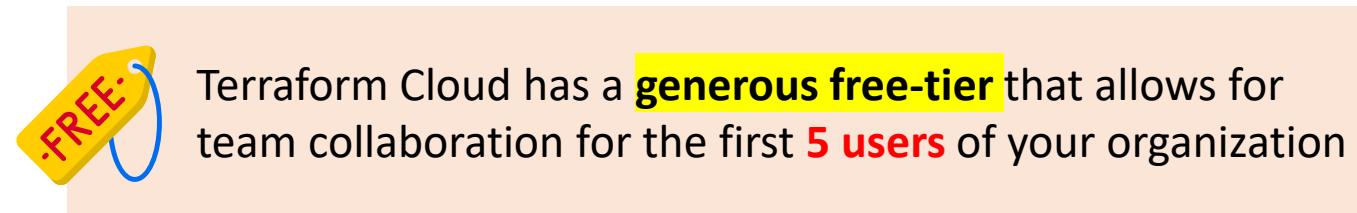
What is Terraform Cloud?

The screenshot shows the Terraform Cloud interface for a workspace named "example-workspace". The top navigation bar includes links for ExamPro, Workspaces, Registry, Settings, and HCP. The main content area displays the workspace's status (0 resources, Terraform version 1.0.4, updated 18 days ago), execution mode (Remote), and auto-apply settings (Off). A prominent yellow callout bubble with the word "FREE:" is overlaid on the metrics section. The "Latest Run" section shows a successful refresh-only run triggered by a user named andrewbrown via the UI, completed 18 days ago. The run summary indicates 0 policy checks, 0 estimated cost changes, and a plan duration of less than a minute. Below the run details, there are sections for Metrics (average plan duration < 1 min, average apply duration 2 mins, 18 total failed runs) and Tags (0). A note at the bottom states that workspaces configured with a Terraform version earlier than 0.12 will not display data here.

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

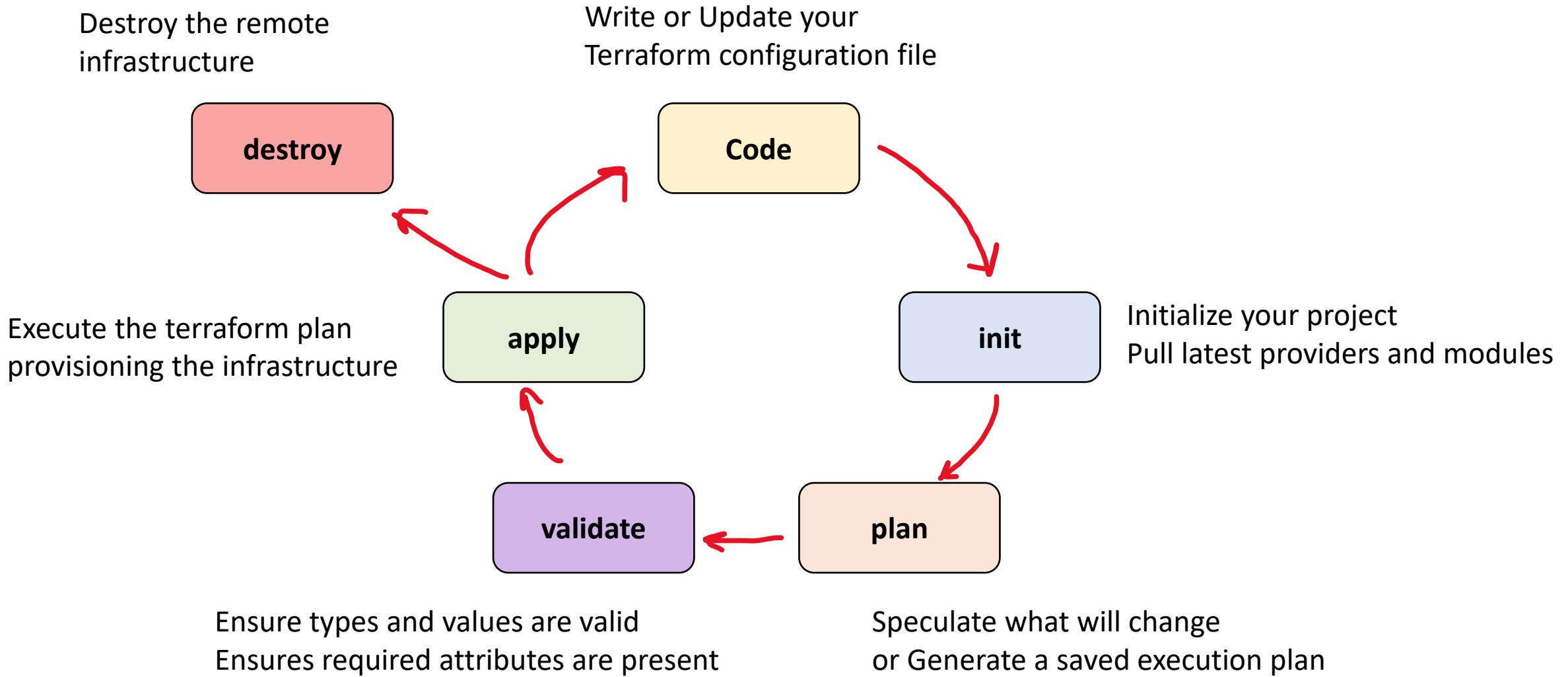


In majority of cases you should be using Terraform Cloud.

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:

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**

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

Terraform – Visualizing Execution Plans

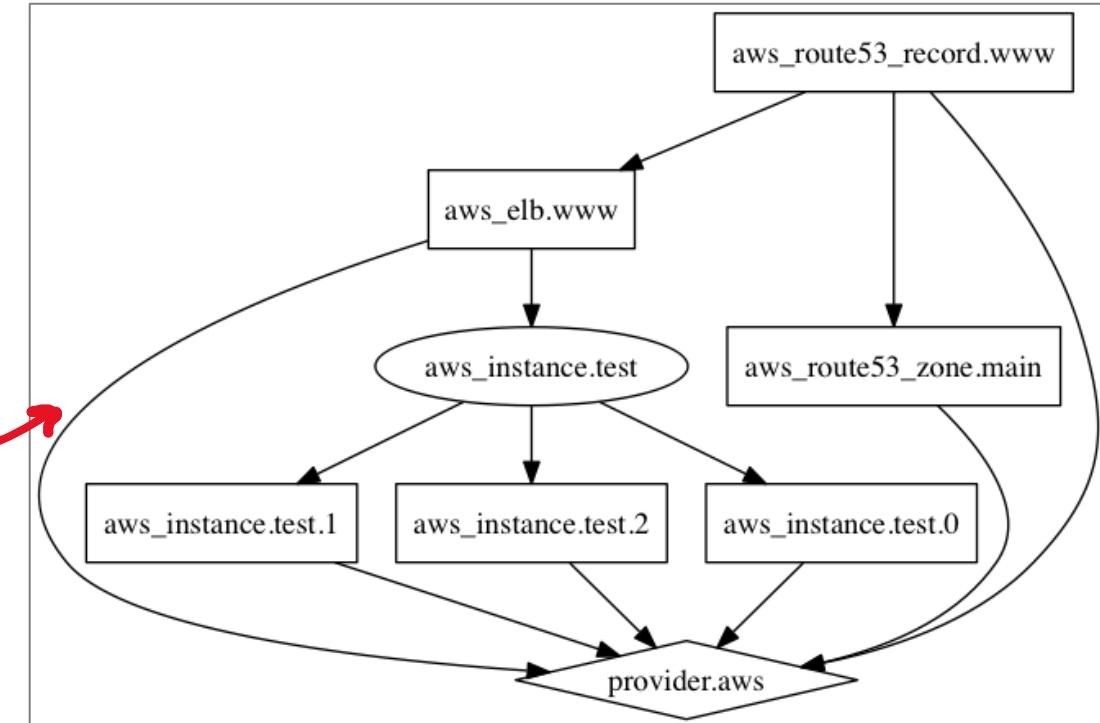
You can **visualize an execution plan** as a graph using the **terraform graph** command
Terraform will output a GraphViz file (you'll need GraphViz installed to view the file)



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
```



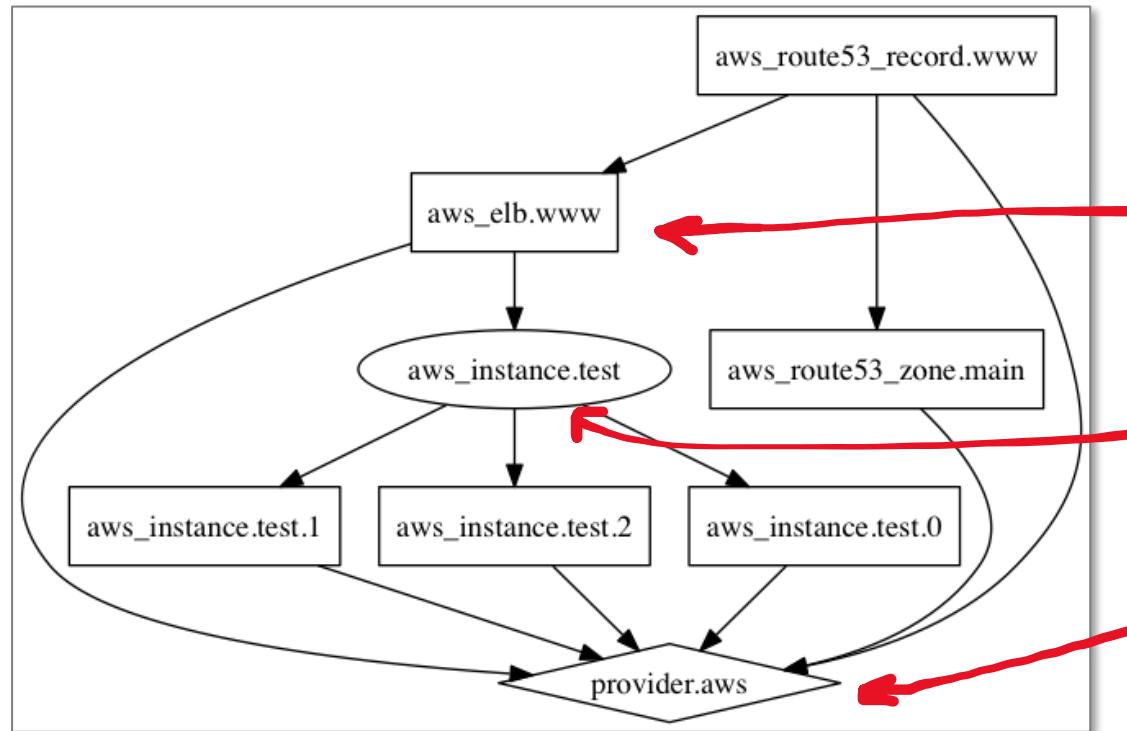
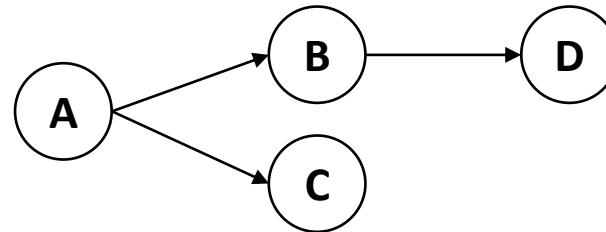
Terraform – Resource Graph

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

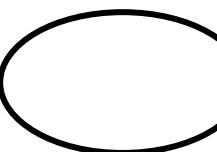
What is a dependency graph?

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



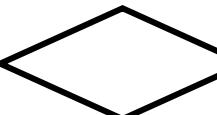
Resource Node

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 and 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:

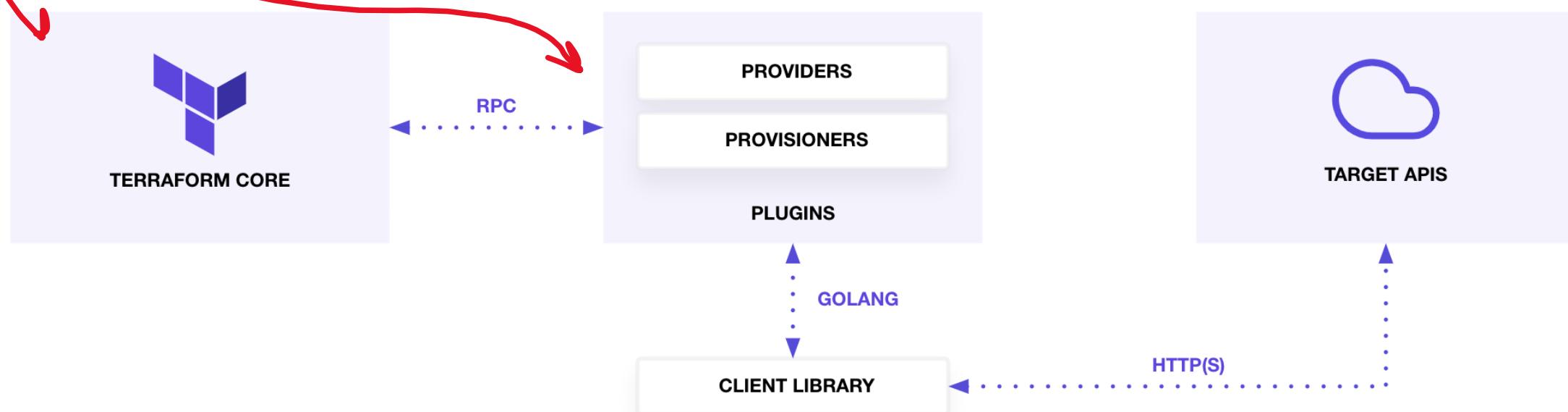
1. **Terraform Core**

- uses remote procedure calls (RPC) to communicate with Terraform Plugins

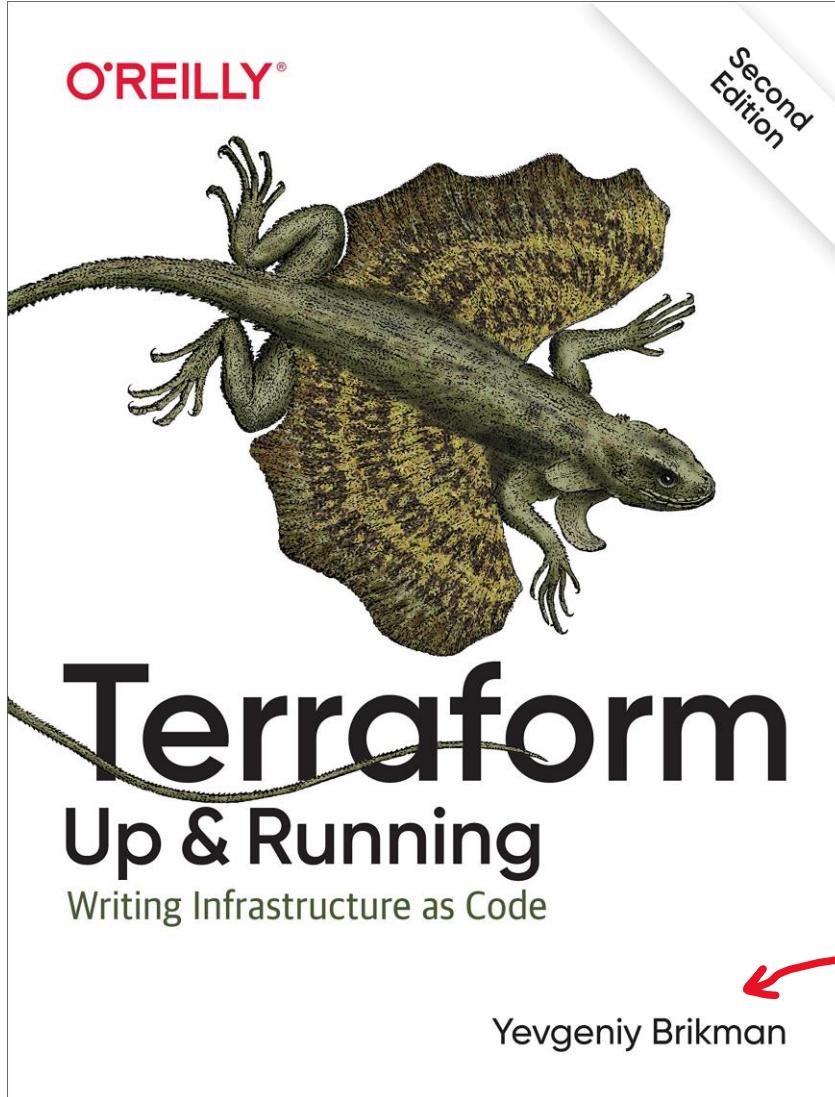
2. **Terraform Plugins**

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

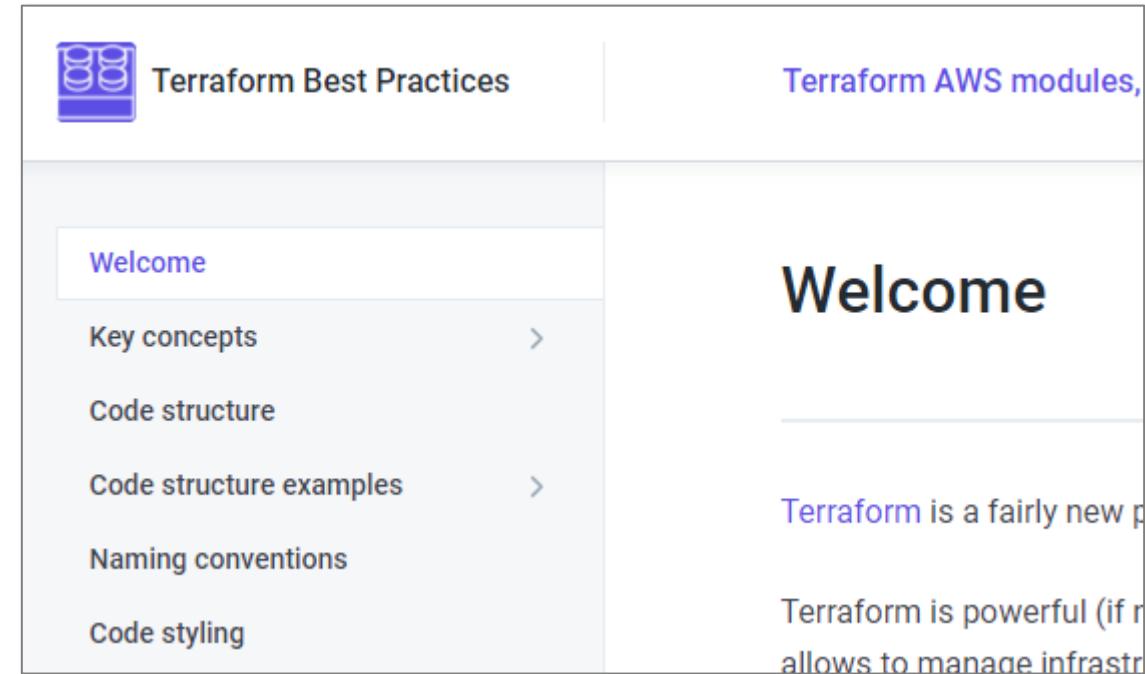
Yevgeniy ("Jim") Brikman is the co-founder of **Gruntwork**



Terraform Best Practices

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

www.terraform-best-practices.com



The screenshot shows the homepage of the Terraform Best Practices website. At the top, there is a header with the title "Terraform Best Practices" and a "Terraform AWS modules" link. Below the header is a navigation menu with the following items:

- Welcome
- Key concepts >
- Code structure
- Code structure examples >
- Naming conventions
- Code styling

The "Welcome" section contains text about Terraform being a fairly new tool and its power. The "Key concepts" section also has some descriptive text.

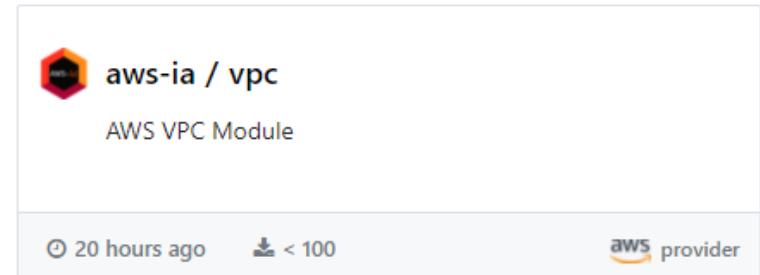
AWS-IA vs Terraform AWS Modules



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>

The screenshot shows a GitHub pull request interface. At the top, there's a comment from 'atlantisbot' with a profile picture of the Atlantis logo. The comment text is:

```
Ran Apply in dir: . workspace: default

null_resource.demo: Creating...
null_resource.demo: Creation complete after 0s (ID: 4542221565395344699)

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

Below the comment, there's a success message with a wrench icon:

Pull request successfully merged and closed
You're all set—the atlantis-workflow--- branch can be safely deleted.

A 'Delete branch' button is visible to the right of the message.

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

CDK for Terraform

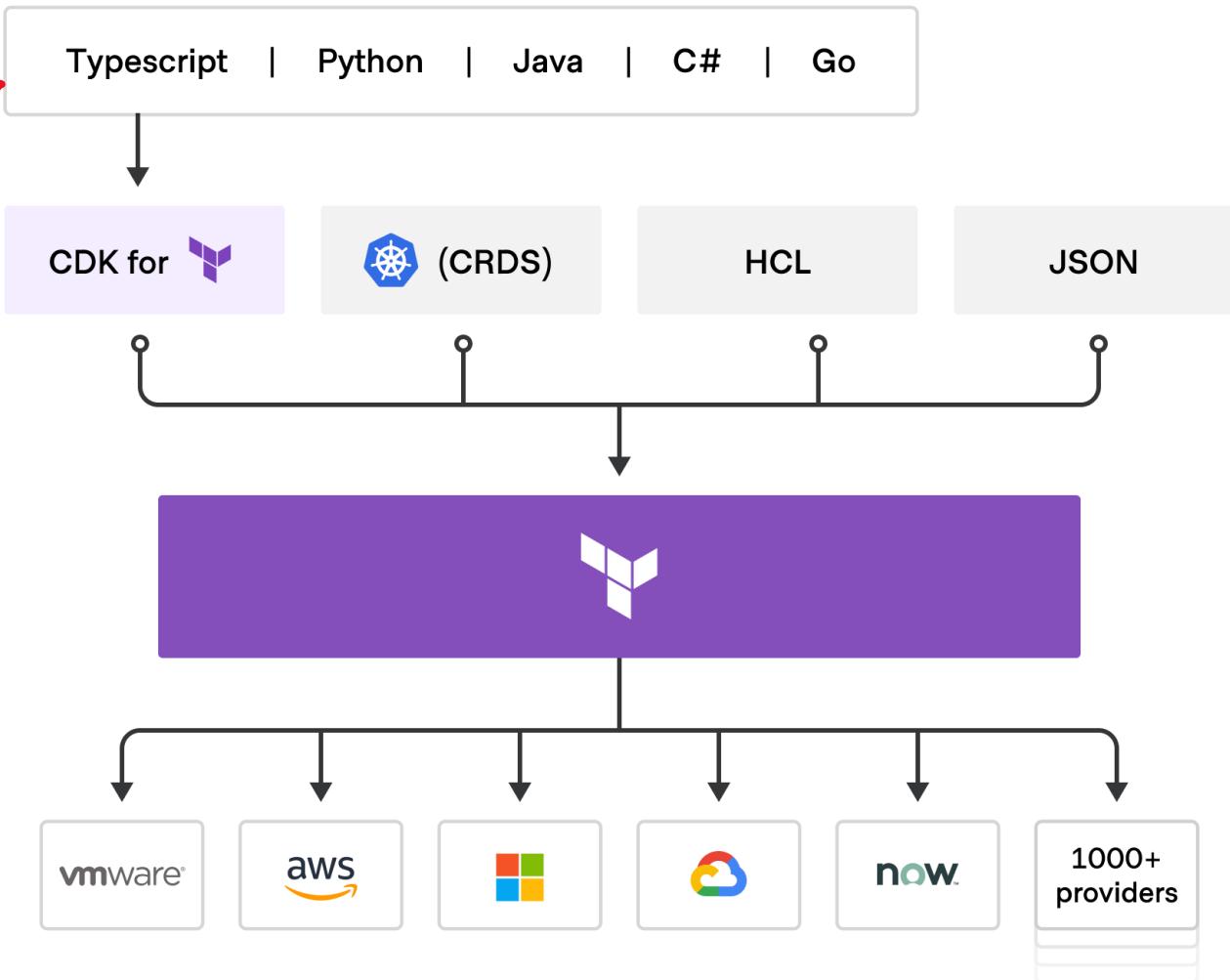


AWS Cloud Development Kit (CDK) is an **imperative** Infrastructure as Code (IaC) tool
With SDKs for your **favorite language**

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.



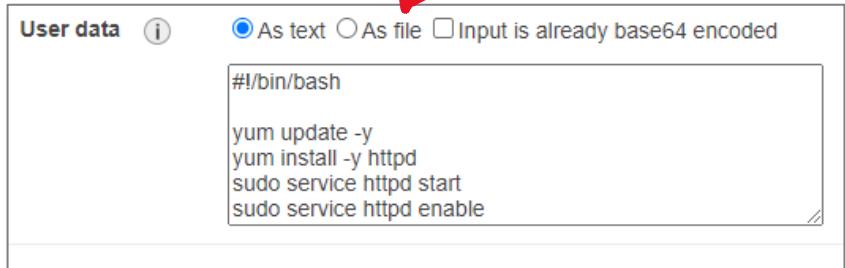
Terraform Provisioners

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

Terraform allows you to work with two different provisioners:



Cloud-Init 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.

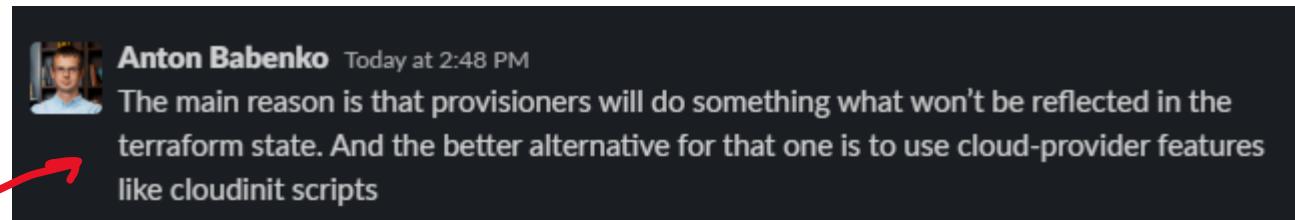


User data (i) As text As file Input is already base64 encoded

```
#!/bin/bash
yum update -y
yum install -y httpd
sudo service httpd start
sudo service httpd enable
```

A screenshot of the AWS CloudFormation 'User data' configuration section. It shows a text input field containing a Bash script to update yum and start the httpd service. The 'As text' radio button is selected.

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



Anton Babenko Today at 2:48 PM
The main reason is that provisioners will do something what won't be reflected in the terraform state. And the better alternative for that one is to use cloud-provider features like cloudinit scripts

A screenshot of a Slack message from Anton Babenko. The message discusses the limitations of provisioners and suggests using cloud-provider features like cloudinit instead.

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 used to directly 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"
```



ANSIBLE

It is uncertain if this advice extends to Ansible.
Terraform and Ansible have 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

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.



Build Server

eg. GCP Cloud Build, AWS CodeBuild, Jenkins

Outputs vs Local-Exec

Terraform outputs allows you to output results after running Terraform apply



Terraform Cloud Run Environment

single-use Linux virtual machine

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

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

working_dir

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

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

interpreter

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

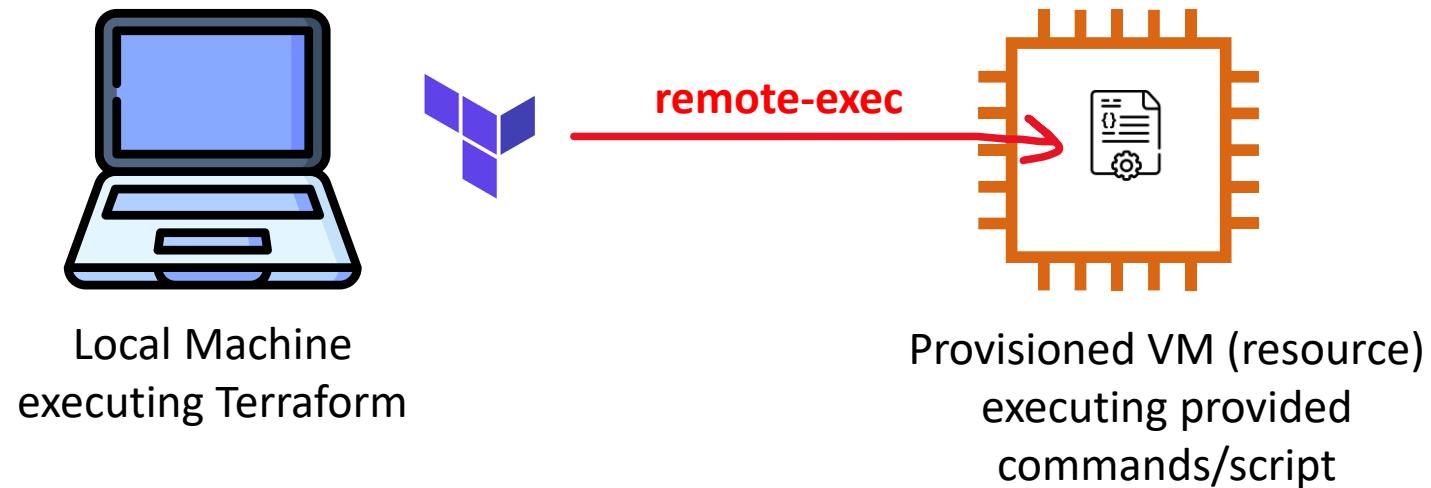
Environment

Key and value pair of environment variables

```
resource "aws_instance" "web" {  
    # ...  
  
    provisioner "local-exec" {  
        command = "echo $KEY $SECRET >> credentials.yml"  
  
        environment = {  
            KEY = "Jlmyn61Qmc8wiux9zprS9v4n"  
            SECRET = "U32KIy2T1fW3xGD1FLggXjuN"  
        }  
    }  
}
```

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}"  
    }  
}
```



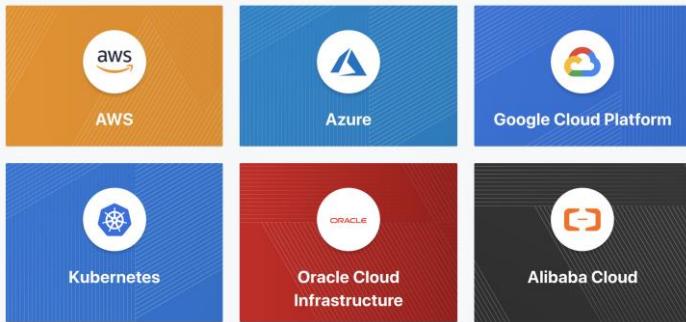
Terraform Data – replace_triggered_by

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

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

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

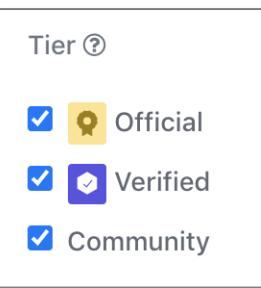
Terraform Providers



Providers are Terraform Plugins that allow you to interact 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>



A screenshot of the Terraform Registry website. The top navigation bar includes the HashiCorp logo, a search bar labeled "Search Providers and Modules", and links for "Browse", "Publish", and "Sign-in". Below the navigation is a main content area with tabs for "Providers" and "Modules", with "Providers" currently selected. A sidebar on the left contains "FILTERS" (with checkboxes for "Official", "Verified", and "Community") and a "Category" section listing various cloud and infrastructure categories like "HashiCorp Platform", "Public Cloud", "Asset Management", etc. The main content area displays a grid of provider icons: AWS (orange), Azure (blue), Google Cloud Platform (blue), Kubernetes (blue), Oracle Cloud Infrastructure (red), and Alibaba Cloud (dark grey). At the bottom of the grid are three smaller items: Active Directory (by hashicorp), Archive (by hashicorp), and Azure Active Directory (by hashicorp).

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

The screenshot shows the Terraform Registry provider page for the AWS provider. At the top, there's a navigation bar with 'Providers / hashicorp / aws / Version 3.53.0' and a 'Latest Version' button. Below the navigation, there's a header with the provider name 'aws' and its official status. A 'Public Cloud' badge is also present. The main content area contains a brief description of the provider's functionality, its version (3.53.0), and download statistics (2 days ago, 430.4M). A large callout box highlights the 'Documentation' tab, which is underlined in blue. Inside this box, there's a section titled 'How to use this provider' containing Terraform configuration code. Three red arrows point from the text 'You can easily find documentation and code sample for providers' to the 'Documentation' tab, the 'How to use this provider' section, and the code sample itself.

```
terraform {  
    required_providers {  
        aws = {  
            source = "hashicorp/aws"  
            version = "3.53.0"  
        }  
    }  
  
    provider "aws" {  
        # Configuration options  
    }  
}
```

The screenshot shows the documentation page for the AWS App Mesh gateway route provider. At the top, there are tabs for 'Overview' and 'Documentation'. The 'Documentation' tab is selected. The page title is 'aws' and it features a 'AWS DOCUMENTATION' sidebar with various links. The main content area starts with a 'Resource:' section for 'aws_appmesh_gateway_route'. It provides a brief description: 'Provides an AWS App Mesh gateway route resource.' Below this, there's an 'Example Usage' section with a code snippet:

```
resource "aws_appmesh_gateway_route" "example" {  
    name          = "example-gateway-route"  
    mesh_name     = "example-service-mesh"  
    virtual_gateway_name = aws_appmesh_virtual_gateway.example  
  
    spec {  
        http_route {  
            action {  
                target {  
                    virtual_service {  
                        virtual_service_name = aws_appmesh_virtual_node.example  
                    }  
                }  
            }  
        }  
    }  
}
```

Terraform Registry

A list of dependent modules

The screenshot shows a Terraform module page for the AWS VPC module. At the top left is the AWS logo. Next to it is the module name "vpc" followed by a blue hexagon icon with a white checkmark. Below the name is the text "AWS". A description follows: "Terraform module which creates VPC resources on AWS". To the right is a dropdown menu showing "Version 3.2.0 (latest)". On the left side of the main content area, there's a red curved arrow pointing from the "A list of dependent modules" text to the "Submodules" button. At the bottom of the main content area, there are two buttons: "Submodules" and "Examples". A red arrow points from the "Lots of examples for common use cases" text to the "Examples" button. On the right side, there's a callout box with the heading "Provision Instructions" containing sample Terraform code. A red arrow points from the "You can easily grab the module code:" text to the "Provision Instructions" box.

Lots of examples for common use cases

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

Version 3.2.0 (latest) ▾

aws vpc AWS

Terraform module which creates VPC resources on AWS

Published June 28, 2021 by [terraform-aws-modules](#)

Module managed by [antonbabenko](#)

Total provisions: 10.5M

Source Code: github.com/terraform-aws-modules/terraform-aws-vpc (report an issue)

Submodules Examples ▾

Readme Inputs (160) Outputs (108) Dependency (1) Resources (76)

Provision Instructions

Copy and paste into your Terraform configuration, insert the variables, and run `terraform init` :

```
module "vpc" {  
  source = "terraform-aws-modules/vpc/  
  version = "3.2.0"  
  # insert the 19 required variables !  
}
```

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

Add Module

This module will be created under the current organization, **ExamPro**. Modules can be added from all supported VCS providers. [🔗](#)

1 Connect to VCS

2 Choose a repository

3 Confirm selection

Connect to a version control provider

Choose the version control provider that hosts your module source code.



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

How to reference
an alias provider



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

Set an alternative 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**

```
provider "azurerm" { version = "0.11.0" }

resource "azurerm_resource_group" "myterraformgroup" {
  name     = "myterraformgroup"
  location = "West Europe"
}

resource "azurerm_virtual_network" "myterranetwork" {
  name                = "myterranetwork"
  address_space       = ["192.168.0.0/16"]
  resource_group_name = azurerm_resource_group.myterraformgroup.name
  subnet_cidr_blocks = ["192.168.1.0/24"]
  tags                = {}
  environment         = "Terraform Demo"
}

resource "azurerm_subnet" "myterranetworksubnet" {
  name                 = "myterranetworksubnet"
  resource_group_name = azurerm_resource_group.myterraformgroup.name
  virtual_network_name = azurerm_virtual_network.myterranetwork.name
  address_prefixes    = ["192.168.1.0/24"]
  environment         = "Terraform Demo"
}

resource "azurerm_public_ip" "myterrampublicip" {
  name                = "myterrampublicip"
  location            = "West Europe"
  resource_group_name = azurerm_resource_group.myterraformgroup.name
  allocation_method   = "Dynamic"
  tags                = {}
  environment         = "Terraform Demo"
}

resource "azurerm_security_group" "myterramsecuritygroup" {
  name                = "myterramsecuritygroup"
  location            = "West Europe"
  resource_group_name = azurerm_resource_group.myterraformgroup.name
  security_rule {
    name          = "allowSSH"
    priority      = 1000
    direction     = "Inbound"
    source_port_range = "22"
    destination_port_range = "22"
    source_address_prefix = "*"
    destination_address_prefix = "*"
  }
  tags                = {}
  environment         = "Terraform Demo"
}

resource "azurerm_network_interface" "myterramnic" {
  name                = "myterramnic"
  resource_group_name = azurerm_resource_group.myterraformgroup.name
  configuration {
    subnet_id        = azurerm_subnet.myterranetworksubnet.id
    private_ip_address = azurerm_ip_configuration.private_ip_address
    public_ip_address_id = azurerm_public_ip.myterrampublicip.id
  }
  tags                = {}
  environment         = "Terraform Demo"
}

resource "azurerm_network_interface_security_group_association" "myterramnic_sg" {
  network_interface_id = azurerm_network_interface.myterramnic.id
  security_group_id    = azurerm_security_group.myterramsecuritygroup.id
}

resource "azurerm_storage_account" "myterramstorage" {
  name                = "myterramstorage"
  account_tier        = "Standard"
  location            = "West Europe"
  access_type         = "blob"
  replication_type   = "LRS"
  tags                = {}
  environment         = "Terraform Demo"
}

resource "azurerm_key_vault" "myterramkeyvault" {
  name                = "myterramkeyvault"
  location            = "West Europe"
  sku                = "Standard"
  access_policies {
    key_vault_permissions = ["get", "list", "set", "delete"]
    object_id             = azurerm_keyVault.myterramkeyvault.id
    tenant_id             = azurerm_keyVault.myterramkeyvault.tenant_id
  }
  tags                = {}
  environment         = "Terraform Demo"
}

resource "azurerm_key_vault_certificate" "myterramcert" {
  name                = "myterramcert"
  key_vault_name       = azurerm_key_vault.myterramkeyvault.name
  certificate_attributes {
    alias           = "myterramcertalias"
    certificate_pem = file("myterramcert.pem")
    issuer_name     = "SelfSigned"
    subject_name    = "myterramcert.com"
  }
  tags                = {}
  environment         = "Terraform Demo"
}

resource "azurerm_virtual_machine" "myterramvm" {
  name                = "myterramvm"
  resource_group_name = azurerm_resource_group.myterraformgroup.name
  size                = "Standard_DS1_v2"
  ip_configuration {
    name            = "myterramnic"
    subnet_id       = azurerm_subnet.myterranetworksubnet.id
    public_ip_address_id = azurerm_ip_configuration.myterramnic.id
  }
  os_disk {
    name          = "myterramosdisk"
    storage_account_name = azurerm_storage_account.myterramstorage.name
    blob_type     = "page"
    caching       = "ReadWrite"
    create_option = "FromImage"
    disk_size_gb  = 10
  }
  network_interface_cards {
    name          = "myterramnic"
    primary       = true
  }
  tags                = {}
  environment         = "Terraform Demo"
}
```

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]
}

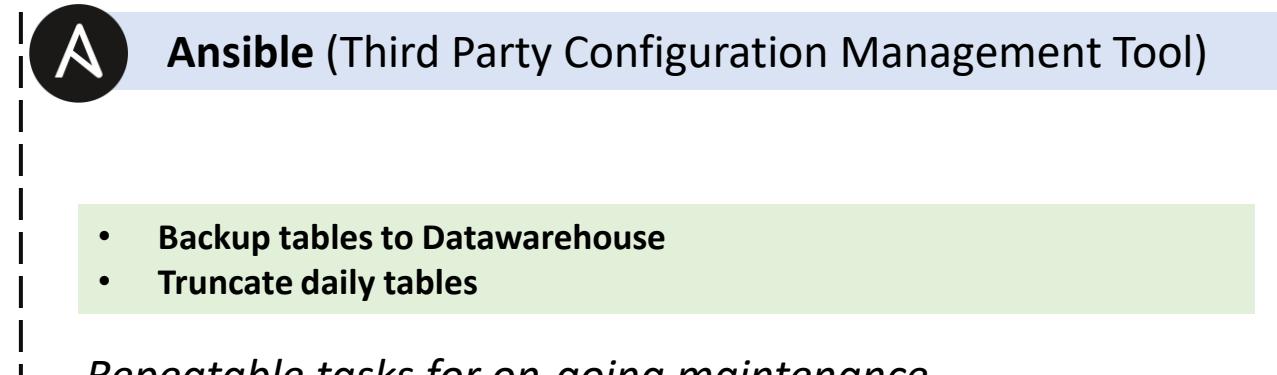
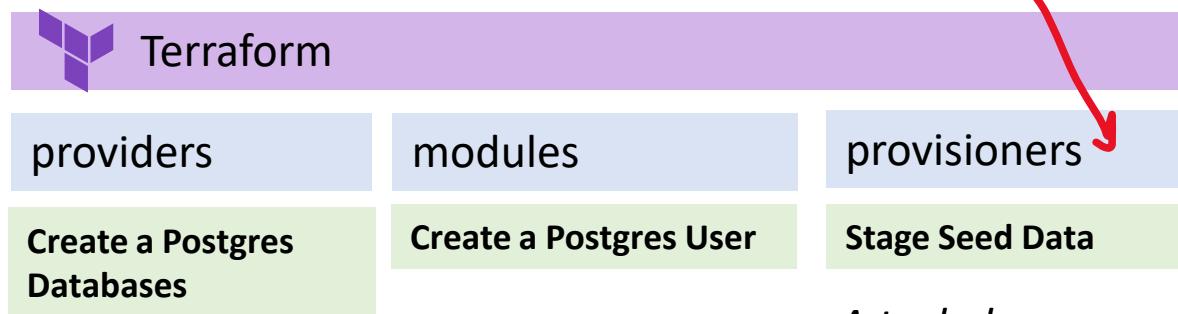
output "linux_vm_public_name" {
  value = module.linuxservers.public_ip_dns_name
}
```

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



Entities. If you want governance or asset resource management

A task done one time to setup the database

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

Block has a type “**resource**”
and it expects two Block Labels

different types have different number of
required labels.

```
resource "aws_instance" "example"  
{  
    ami = "abc123"  
  
    network_interface {  
        # ...  
    }  
}
```

A block can contain
nested blocks

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

Hashicorp Configuration Files – Syntax

```
image_id = "abc123"
```



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**

```
{  
  "resource": {  
    "aws_instance": {  
      "example": {  
        "instance_type": "t2.micro",  
        "ami": "ami-abc123"  
      }  
    }  
  }  
}
```

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)	eg. Dynamic Blocks, For Each...
	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)	
	Sentinel Policies X	Doesn't use HCL but its own ACL custom language

Input Variables

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

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

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"  
    }  
  ]  
}
```

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 specify 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 specify variables inline via the command line for individual overrides

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

You can specify 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.

Locals are set using
the **locals block**

You can define
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  
    }  
}
```

Static value
computed values

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

```
data "aws_ami" "web" {  
    filter {  
        name  = "state"  
        values = ["available"]  
    }  
  
    filter {  
        name  = "tag:Component"  
        values = ["web"]  
    }  
  
    most_recent = true  
}
```

You use filters to narrow down the selection

```
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.

```
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,
    ]
}
```

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



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.

```
Terraform used the selected providers
```

```
+ create
```

```
Terraform will perform the following actions:
```

```
# aws_instance.my_example_server
+ resource "aws_instance" "my_example_server"
  + ami
```

```
Terraform used the selected providers
```

```
- destroy
```

```
Terraform will perform the following actions:
```

```
# aws_instance.my_example_server
- resource "aws_instance" "my_example_server"
  - ami
```

```
Terraform used the selected providers to generate a
```

```
~ update in-place
```

```
Terraform will perform the following actions:
```

```
# aws_instance.my_example_server will be replaced
~ resource "aws_instance" "my_example_server"
  id
  ~ instance_type
```

```
Terraform used the selected providers to generate the
```

```
-/+ destroy and then create replacement
```

```
Terraform will perform the following actions:
```

```
# aws_instance.my_example_server will be replaced
-/+ resource "aws_instance" "my_example_server"
  ~ arn
```

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**

```
# 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

  # ...
}
```

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

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
```

complex/structural/collection types

list (tuple)

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

map (object)

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

null represents *absence* or *omission*

*when you want to use the **underlying default** of a provider's resource configuration option*

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
\UNNNNNNNNN	Unicode character from supplementary planes

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

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

special escape sequences:

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

Strings Templates

String **interpolation** allows you to evaluate an expression between the markers eg. `${....}` and converts 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 **strip whitespace** 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 iterate 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”, WORLD”]

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 like 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**

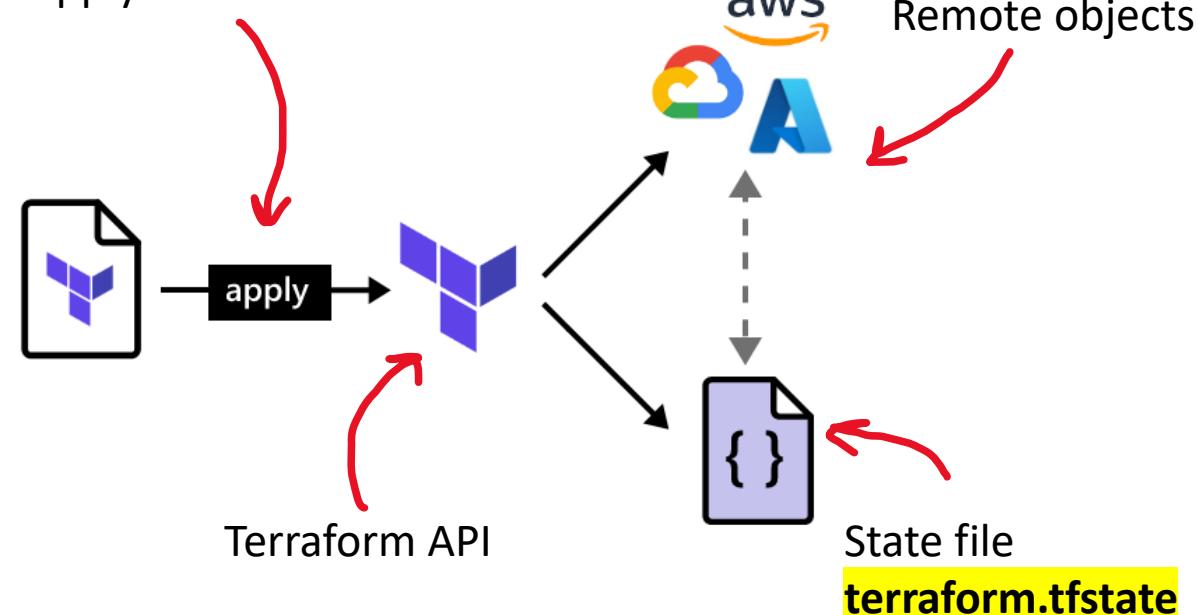
This **state file is a JSON data structure** with a one-to-one mapping from **resource instances** to **remote objects**

```
{  
  "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.terraform.io/providers/hashicorp/aws/v3\"]",  
      "instances": [  
        {  
          "schema_version": 1,  
          "attributes": {  
            "ami": "ami-0c2b8ca1dad447f8a",  
            "arn": "arn:aws:ec2:us-east-1:3184:instance/0c2b8ca1dad447f8a",  
            "associate_public_ip_address": true,  
            "availability_zone": "us-east-1a",  
            "dns_name": "ip-172-31-10-11.us-east-1.compute.amazonaws.com",  
            "id": "i-0c2b8ca1dad447f8a",  
            "private_ip": "172.31.10.11",  
            "public_ip": null,  
            "public_dns": null  
          }  
        }  
      ]  
    }  
  ]  
}
```

Configuration files:

- main.tf
- variables.tf
- terraform.tfvars

Terraform apply command



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
```

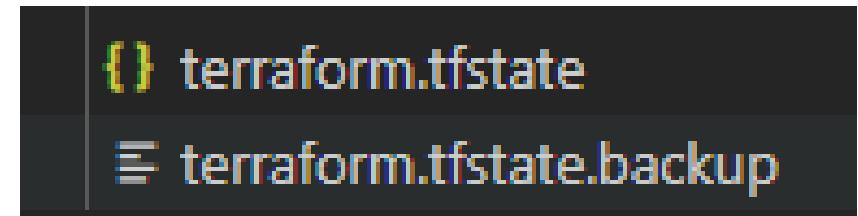
Move module into a module

```
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**



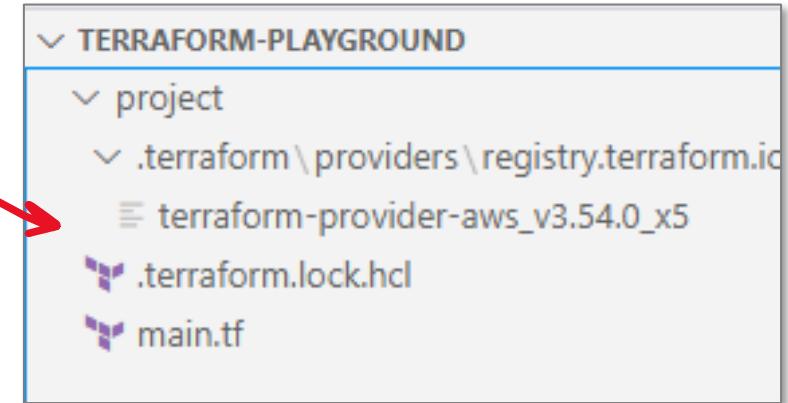
Backups cannot be disabled. This is by design to enforce best-practice for recovery

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
Skip plugin installation

Dependency lock file
.terraform.lock.hcl

terraform init -get-plugins=false

Force plugin installation to read plugins
only from target directory

State lock file
.terraform.tfstate.lock.hcl

terraform init -plugin-dir=PATH

Set a dependency lockfile mode

terraform init -lockfile=MODE

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"  
}
```

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"  
}
```

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 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

```
$ terraform validate
Error: Missing required argument

with aws_instance.app_server,
on main.tf line 16, in resource "aws_instance" "app_server":
16: resource "aws_instance" "app_server" {

  "instance_type": one of `instance_type,launch_template` must be specified
```

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

```
$ terraform console
> min([20,200,12])

Error: Invalid function argument

on <console-input> line 1:
(source code not available)

Invalid value for "numbers" parameter: number required.
```

We correct the
argument error

```
> min(20,200,12)
12
```

Terraform Plan

Terraform command creates an **execution plan** (aka Terraform Plan).

A plan consist of:

- Reading the current state of any already-existing remote objects to make sure that the Terraform state is up-to-date.
- Comparing the current configuration to the prior state and noting any differences.
- Proposing a set of change actions that should, if applied, make the remote objects match the configuration.

Terraform plan **does not carry out** proposed changes.

A Terraform Plan file is binary file. If you open it up you'll just see machine code

Speculative Plans

When you run **terraform plan**

Terraform will output the description of the effect of the plan but without any intent to actually apply it.

Saved Plans

When you run **terraform plan -out=FILENAME**

You can generate a saved plan file which you can then pass along to terraform apply. e.g. **terraform apply FILE**

When using save planed it will not prompt you to confirm and will act like auto-approve

```
/mnt/c/Users/Andrew/Sites/terraform-playground/terraform-example: terraform plan
Terraform used the selected providers to generate the following execution plan. Res
+ create
<= read (data resources)

Terraform will perform the following actions:

# aws_instance.my_example_server will be created
+ resource "aws_instance" "my_example_server" {
    + ami                                = "ami-0c2b8ca1dad447f8a"
    + 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
    + host_id                            = (known after apply)
    + id                                 = (known after apply)
    + instance_initiated_shutdown_behavior = (known after apply)
    + instance_state                     = (known after apply)
```

Terraform Apply

Terraform apply command executes the actions proposed in an execution plan

Automatic Plan Mode

When you run **terraform apply**

Executes plan, validate and the apply.

Requires users to **manually approve** the plan by writing “yes”

Terraform apply **–auto-approve** flag will automatically approve the plan.

Saved Plan Mode

When you provide a filename to terraform to saved plan file **terraform apply FILE**

performs exactly the steps specified by that plan file. It does not prompt for approval; if you want to inspect a plan file before applying it, you can use **terraform show**.



```
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
```

Manage Resource Drift

Drift (Configuration or Infrastructure) is when your expected resources are in different state then your expected state

We can resolve Drift in three ways in Terraform:

Replacing Resources

When a resource has become damaged or degraded that cannot be detected by Terraform we can use the **-replace** flag

Importing Resources

When an approved manual addition of resource needs to be added to our state file We use the **import** command.

Refresh State

When an approved manual configuration of a resource has changed or removed We use the **-refresh-only** flag to reflect the changes in our state file

Replacing Resources

Terraform Taint is used to **mark a resource** for *replacement*, the next time you run apply.

Why would want to mark a resource for replacement?

```
terraform taint aws_instance.my_web_app
```



A cloud resource can become **degraded or damaged** and you want to return the expected resource to a healthy state

Terraform taint was deprecated in 0.152

It is recommended in 0.152+ to use the **-replace** flag and providing a resource address

```
terraform apply -replace="aws_instance.example[0]"
```

The replace flag is available on **plan** and **apply**

The replace flag appears to only work for a single resource

Resource Addressing

A resource address is a string that **identifies zero or more resource instances** in your configuration.

An address is composed of two parts:

[module path][resource spec]

module.module_name[module index]

A module path addresses a module
within the tree of modules

module A namespace for modules

module_name User-defined name of the module

[module index] When multiple specific an index

resource_type.resource_name[instance index]

A resource spec addresses a specific
resource instance in the selected module

resource_type Type of the resource being addressed

resource_name User-defined name of the resource

[instance index] when multiple specific a index

```
resource "aws_instance" "web" {  
    # ...  
    count = 4  
}
```

aws_instance.web[3]

Select the 4th Virtual Machine

terraform import

The terraform import command is used to import ***existing resources*** into Terraform

Define a placeholder for your imported resource in configuration file.

You can leave the body blank and fill it in after importing. It will not be autofilled.

```
resource "aws_instance" "example" {  
    # ...instance configuration...  
}
```

terraform import <RESOURCE_ADDRESS> <ID>

```
terraform import aws_instance.example i-abcd1234
```

Proceed to importing your import file.

The command can only import one resource at a time.

Not all resources are importable, you need to check the bottom of resource documentation for support

Terraform refresh

The **terraform refresh** command **reads the current settings** from all managed remote objects **and updates the Terraform state to match.**

The `terraform refresh` command is an alias for the refresh only and auto approve

`terraform refresh`

`terraform apply -refresh-only -auto-approve`

Terraform refresh will not modify your real remote objects, but it will modify the Terraform state.

Terraform refresh has been *deprecated* and with the refresh-only flag because it was not safe since it did not give you an opportunity to review proposed changes before updating state file

Refresh-Only Mode

The **-refresh-only** flag for terraform plan or apply allows you to refresh and update your state file without making changes to your remote infrastructure.

Scenario

Imagine you create a terraform script that deploys a Virtual Machine on AWS. You ask an engineer to terminate the server, and instead of updating the terraform script they mistakenly terminate the server via the AWS Console.

terraform apply



You run....



terraform apply **-refresh-only**

- Terraform will notice that the VM is missing
- Terraform will propose to create a new VM

- Terraform will notice that the VM you provisioned is missing.
- With the refresh-only flag that the missing VM is intentional
- Terraform will propose to delete the VM from the state file

The State File is correct

Changes the infrastructure to match state file

The State File is wrong

Changes the state file to match infrastructure

Terraform Troubleshooting

There are **four types of errors** you can encounter with Terraform:

Language errors

Terraform encounters a syntax error in your configuration for the Terraform or HCL

Language



```
terraform fmt  
terraform validate  
terraform version
```

Easy to Solve

State errors

Your resources state has changed from the expected state in your configuration file



```
terraform refresh  
terraform apply  
terraform -replace flag
```

Core errors

A bug has occurred with the core library



```
TF_LOG  
Open Github Issue
```

Provider errors

The provider's API has changed or does not work as expected due to emerging edge cases



```
TF_LOG  
Open Github Issue
```

Harder to Solve

Debugging Terraform

Terraform has detailed logs which can be enabled by setting the **TF_LOG** environment variable to:

Logging can be enabled separately:

- TF_LOG_CORE
- TF_LOG_PROVIDER

takes the same options as TF_LOG

Choose where you want to log with **TF_LOG_PATH**

- TRACE
- DEBUG
- INFO
- WARN
- ERROR
- JSON — outputs logs at the TRACE level or higher, and uses a parseable JSON encoding as the formatting.

**Terraform
Log example**

```
2021-04-02T15:36:08.477-0400 [DEBUG] provider.terraform-provider-aws_v3.34.0_x5: plugin address:  
address=/var/folders/rf/mz_yf4bx0hv1r8m7yqw840jc0000gn/T/plugin730416856 network=unix timestamp=2021-04-02T15:36:08.476-0400  
2021-04-02T15:36:08.477-0400 [DEBUG] provider: using plugin: version=5  
2021-04-02T15:36:08.525-0400 [TRACE] provider.stdio: waiting for stdio data  
2021-04-02T15:36:08.525-0400 [TRACE] GRPCProvider: GetProviderSchema  
2021-04-02T15:36:08.593-0400 [TRACE] GRPCProvider: Close  
2021-04-02T15:36:08.596-0400 [DEBUG] provider.stdio: received EOF, stopping recv loop: err="rpc error: code = Unavailable desc = transport..."  
2021-04-02T15:36:08.598-0400 [DEBUG] provider: plugin process exited: path=.terraform/providers/registry.terraform.io/hashicorp/aws/3.34...  
2021-04-02T15:36:08.598-0400 [DEBUG] provider: plugin exited  
2021-04-02T15:36:08.598-0400 [TRACE] terraform.NewContext: complete  
2021-04-02T15:36:08.599-0400 [TRACE] backend/local: finished building terraform.Context  
2021-04-02T15:36:08.599-0400 [TRACE] backend/local: requesting interactive input, if necessary  
2021-04-02T15:36:08.599-0400 [TRACE] Context.Input: Prompting for provider arguments  
2021-04-02T15:36:08.599-0400 [TRACE] Context.Input: Provider provider.aws declared at main.tf:15,1-15  
2021-04-02T15:36:08.600-0400 [TRACE] Context.Input: Input for provider.aws: map[string]cty.Value{}  
2021-04-02T15:36:08.600-0400 [TRACE] backend/local: running validation operation  
2021-04-02T15:36:08.600-0400 [INFO] terraform: building graph: GraphTypeValidate  
2021-04-02T15:36:08.600-0400 [TRACE] Executing graph transform *terraform.ConfigTransformer
```

Debugging Terraform – Crash Log



If Terraform ever crashes (a "panic" in the **Go runtime**), it saves a log file with the debug logs from the session as well as the panic message and backtrace to crash.log

```
panic: runtime error: invalid memory address or nil pointer dereference

goroutine 123 [running]:
panic(0xabcc100, 0xd93000a0a0)
    /opt/go/src/runtime/panic.go:464 +0x3e6
github.com/hashicorp/terraform/builtin/providers/aws.resourceAwsSomeResourceCreate(...)
    /opt/gopath/src/github.com/hashicorp/terraform/builtin/providers/aws/resource_aws_some_resource.go:123 +0x123
github.com/hashicorp/terraform/helper/schema.(*Resource).Refresh(...)
    /opt/gopath/src/github.com/hashicorp/terraform/helper/schema/resource.go:209 +0x123
github.com/hashicorp/terraform/helper/schema.(*Provider).Refresh(...)
    /opt/gopath/src/github.com/hashicorp/terraform/helper/schema/provider.go:187 +0x123
github.com/hashicorp/terraform/rpc.(*ResourceProviderServer).Refresh(...)
    /opt/gopath/src/github.com/hashicorp/terraform/rpc/resource_provider.go:345 +0x6a
reflect.Value.call(...)
    /opt/go/src/reflect/value.go:435 +0x120d
reflect.Value.Call(...)
    /opt/go/src/reflect/value.go:303 +0xb1
net/rpc.(*service).call(...)
    /opt/go/src/net/rpc/server.go:383 +0x1c2
created by net/rpc.(*Server).ServeCodec
    /opt/go/src/net/rpc/server.go:477 +0x49d
```

This log file is meant to be passed along to the developers via a GitHub Issue.
As a user, you're not required to dig into this file

Finding Modules

Terraform Modules can be publicly found in the Terraform Registry

The screenshot shows the Terraform Registry interface. On the left, there's a sidebar with a search bar and two tabs: 'Providers' and 'Modules'. Below the tabs are 'FILTERS' and a 'Provider' dropdown menu containing 'aws', 'alicloud', 'aws' (selected), 'azurerm', 'google', and 'oci'. A red arrow points from the text 'You can filter based on popular providers' to the provider dropdown. On the right, a search bar at the top of a list of modules has 'azure compute' typed into it. A red arrow points from the text 'or search partial terms eg. azure compute' to the search bar. The search results show three items:

- Azure/compute**
Terraform Azure RM Compute Module
- Azure/computegroup**
Terraform Azure RM Compute Group Module
- adfinis-forks/compute**

Below the search results, two more modules are listed:

- terraform-aws-modules / vpc**
Terraform module which creates VPC resources on AWS
Last updated 4 days ago, 10.5M downloads
- terraform-aws-modules / security-group**
Terraform module which creates EC2-VPC security groups on AWS

You can filter based on popular providers

or search partial terms eg. azure compute

Only verified modules will be displayed in search terms

Using Modules

Public Modules

Terraform Registry is integrated **directly into** Terraform

The syntax for specifying a registry module is
`<NAMESPACE>/<NAME>/<PROVIDER>`



```
module "consul" {  
  source = "hashicorp/consul/aws"  
  version = "0.1.0"  
}
```

terraform init command will **download and cache any modules** referenced by a configuration

Private Modules

Private registry modules have source strings of the form
`<HOSTNAME>/<NAMESPACE>/<NAME>/<PROVIDER>`

```
module "vpc" {  
  source =  
  "app.terraform.io/example_corp/vpc/aws"  
  version = "0.9.3"  
}
```

To configure private module access, you need to authenticate against Terraform Cloud via **terraform login**

Alternatively you can create a user API Token and manually configure credentials in the CLI config file.

Publishing Modules

Anyone can publish and share modules on the Terraform Registry.

Published modules support:

- Versioning
- automatically generate documentation
- allow browsing version histories
- show examples
- READMEs

Repos names must be in the following format:

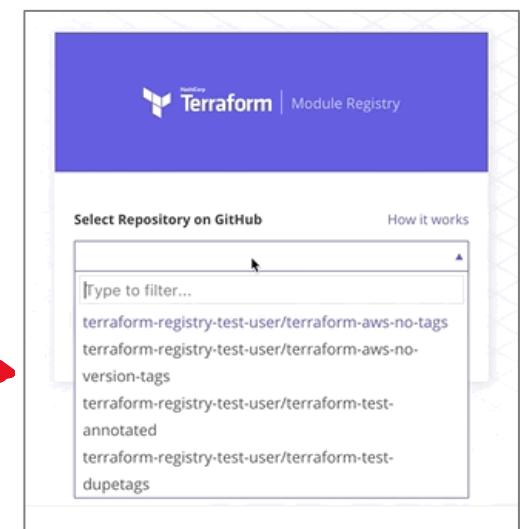
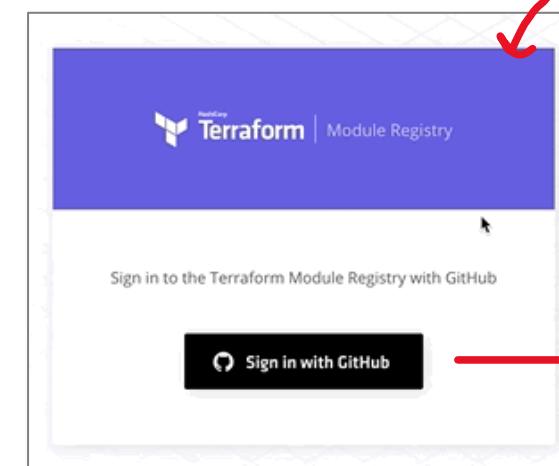
terraform-<PROVIDER>-<NAME>



Public modules are managed via a **public** Git repo on **GitHub**

Once a module is registered, to push updates you simply push new versions to properly formed Git Tags

You can **connect and publish** your module in seconds



Provider Tiers and Namespaces

Terraform modules and providers are categorized to help users understand their source and support level. The badges next to providers indicate their tier and the organization responsible for their maintenance.

- 1. Official Providers:** Maintained by HashiCorp.
- 2. Partner Providers:** Developed and maintained by third-party companies as part of the HashiCorp Technology Partner Program.
- 3. Community Providers:** Published by individual maintainers or community groups.
- 4. Archived Providers:** Previously official or partner providers no longer maintained due to deprecation or low interest.

Key Points:

- **Verified Modules:** Now integrated into the provider tier system. Verified modules are reviewed by HashiCorp or partners and maintained to stay up-to-date with Terraform and their respective providers.
- **Provider Badges:** The verified badge has been replaced with provider badges indicating the source and support level (Official, Partner, Community, Archived).
- **Quality Indication:** A badge indicates the source and maintenance level but not necessarily the feature set or flexibility. Unverified modules can still be high quality and actively maintained by community members.

Standard Module Structure

The Standard Module Structure is a file and directory layout recommend for module development

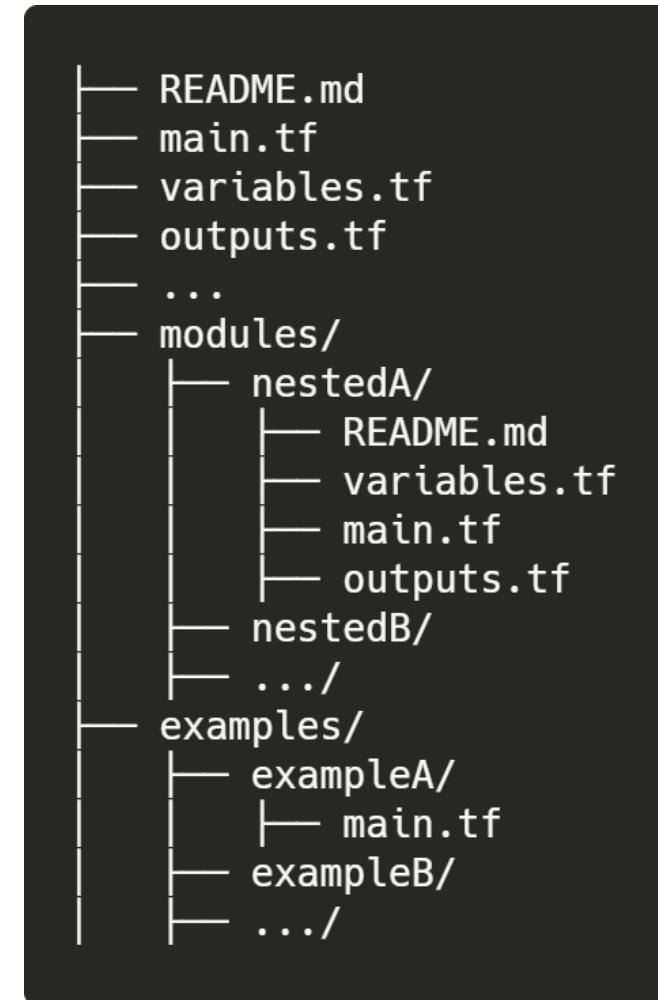
The primary entry point is the **Root Module**.

These are required files in the root directory:

- Main.tf - the entry point file of your module
- Variables.tf – variable that can be passed in
- Outputs.tf – Outputed values
- README – Describes how the module works
- LICENSE – The license under which this module is available

Nested modules are optional must be contained in the **modules/** directory:

- A submodule that contains a README is consider usable by external users
- A submodule that does not contain a README is consider internal use only
- Avoid using relative paths when sourcing module blocks.



Root Module

Nested
Module

Terraform Workflows

The core Terraform workflow has three steps:

1. **Write** - Author infrastructure as code.
2. **Plan** - Preview changes before applying.
3. **Apply** - Provision reproducible infrastructure.



As your team and requirements grow your workflow will evolve.

Let's look at what this workflow will look like for:

- Individual Practitioner (One person team)
- Teams using OSS
- Teams using Terraform Cloud

Team Workflows – Individual Practitioner

Write

- You write your Terraform configuration in your editor of choice
- You'll store your terraform code in a VCS e.g. Github
- You repeatedly run `terraform plan` and validate to find syntax errors.
- Tight feedback loop between editing code and running test commands

Plan

- When the developer is confident with their work in the write step they commit their code to their local repository.
- They may be only using a single branch e.g. `main`
- Once their commit is written they'll proceed to apply

Apply

- They will run `terraform apply` and be prompted to review their plan
- After their final review they will approve the changes and await provisioning
- After a successful provision they will push their local commits to their remote repository.

Team Workflows – Team

Write

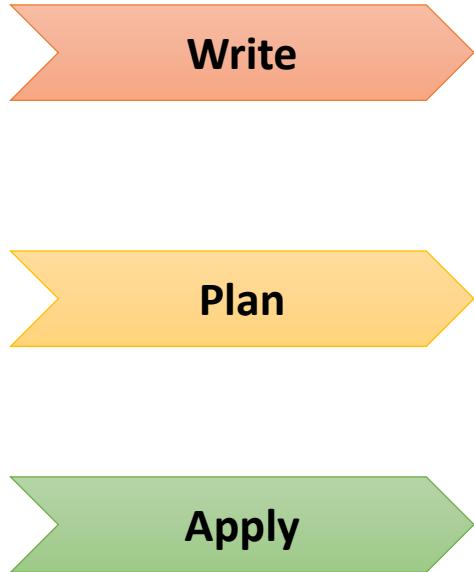
- Each team members writes code locally on their machine in their editor of choice
- A team member will store their code to a branch in their code repository
 - Branches help avoid conflicts while a member is working on their code
 - Branches will allow an opportunity to resolve conflict during a merge into main
- Terraform plan can be used as a quick feedback loop for small teams
- For larger teams a concern over sensitive credentials becomes a concern.
 - A CI/CD process may be implemented so the burden of credentials is abstracted away
- When a branch is ready to be incorporated on Pull Request an Execution Plan can be generated and displayed within the Pull Request for Review
- To apply the changes the merge needs to be approved and merged, which will kick off a code build server that will run terraform apply.

Plan

Apply

- The DevOps team has to setup and maintain their own CI/CD Pipeline
- They have to figure out how to store the state file e.g. Standard Backend remote state
- They are limited in their access controls (they can't be granular about what actions are allowed to be performed by certain members e.g. apply, destroy)
- They have to figure out a way to safely store and inject secrets into their build server's runtime environment
- Managing multiple environments can make the overhead of the infrastructure increase dramatically.

Team Workflows – Terraform Cloud



- A team will use Terraform Cloud as their remote backend.
 - Inputs Variables will be stored on Terraform Cloud instead of the local machines.
 - Terraform Cloud integrates with your VCS to quickly setup a CI/CD pipeline
 - A team member writes code to branch and commits per usual
-
- A pull request is created by a team member and Terraform Cloud will generate the speculative plan for review in the VCS. The member can also review and comment on the plan in Terraform Cloud.
 - After the Pull request is merge Terraform Cloud runtime environment will perform a Terraform apply. A team member can Confirm and apply the changes.

Terraform Cloud streamlines a lot of the CI/CD effort, storing and securing sensitive credentials and makes it easier to go back and audit the history of multiple runs.

Backends

Each Terraform configuration can specify a backend, which defines where and how operations are performed, where state snapshots are stored

Terraform's backends are divided into two types:

Standard backends

- only store state
- does not perform terraform operations eg. Terraform apply
 - To perform operations you use the CLI on your local machine
- **third-party backends** are Standard backends e.g. AWS S3

Enhanced backends

- can both store state
- can perform terraform operations

enhanced backends are subdivided further:

- **local** – files and data are stored on the local machine executing terraform commands
- **remote** – files and data are stored in the cloud eg. Terraform Cloud

Standard Backends



Simple Storage Service (S3) *locking via DynamoDB*

AWS Cloud Storage



AzureRM with locking

Blob Storage Account



Google Cloud Storage (GCS) with locking

Google Cloud's Object Storage



Alibaba Cloud Object Storage Service (OSS)

Cloud Storage

locking via TableStore



OpenStack Swift with locking

Rackspace's OpenStack Private Cloud Storage



Tencent Cloud Object Storage (COS) with locking



Manta (Triton Object Storage) with locking

Joynet's Cloud Storage



Artifactory no locking

a universal repository manager



HashiCorp Consul with locking

service networking platform (service mesh)



etcd no locking and **etcdv3** with locking

A distributed, reliable key-value store for the most critical data of a distributed system



Postgres database with locking

Relational database



Kubernetes secret with locking

Secrets storage in K8 cluster



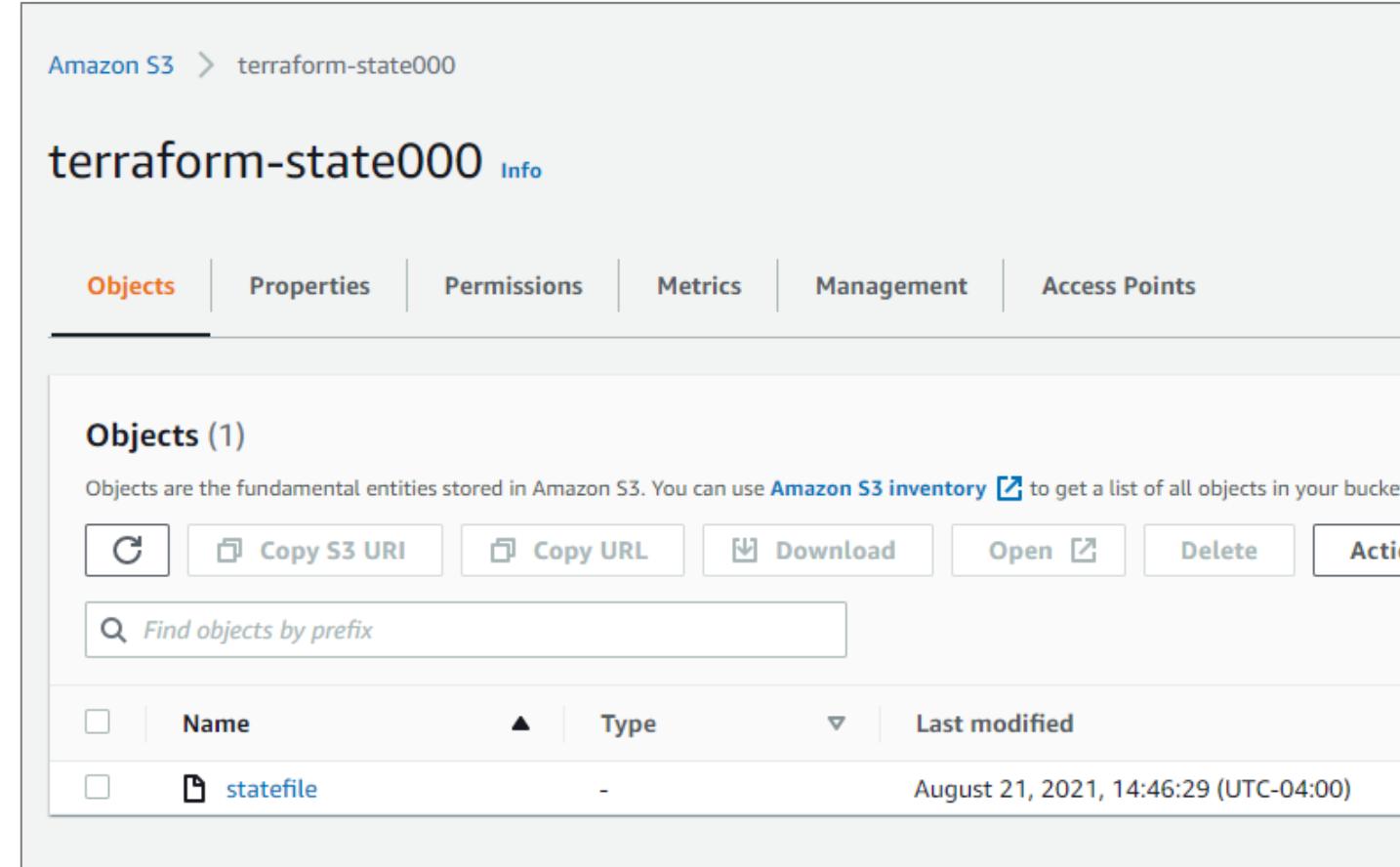
HTTP Protocol *optional* locking

Use a REST API to setup a custom remote backend

Standard Backends

```
backend "s3" {  
  bucket = "terraform-state000"  
  key    = "statefile"  
  region = "us-east-1"  
}
```

The backup of a state file will reside on your local machine.



Amazon S3 > terraform-state000

terraform-state000 [Info](#)

Objects Properties Permissions Metrics Management Access Points

Objects (1)

Objects are the fundamental entities stored in Amazon S3. You can use [Amazon S3 inventory](#) to get a list of all objects in your bucket.

[Find objects by prefix](#)

<input type="checkbox"/>	Name	Type	Last modified
<input type="checkbox"/>	statefile	-	August 21, 2021, 14:46:29 (UTC-04:00)

Configuring a standard backend *does not require* a Terraform Cloud account or workspace

Local Backends

The local backend:

- stores state on the local filesystem
- locks that state using system APIs
- performs operations locally

By default, you are using the backend state when you have no specified backend

```
terraform {  
}
```

You specific the backend with argument **local**, and you can change the path to the local file and **working_directory**

```
terraform {  
  backend "local" {  
    path = "relative/path/to/terraform.tfstate"  
  }  
}
```

You can set a backend to reference another state file so you can read its outputted values.

This is way of **cross-referencing stacks**

```
data "terraform_remote_state" "networking" {  
  backend = "local"  
  
  config = {  
    path = "${path.module}/networking/terraform.tfstate"  
  }  
}
```

Remote Backend

A Remote backend uses the Terraform platform which is either:

- Terraform Cloud
- Terraform Enterprise

With a remote backend when `terraform apply` is performed via the CLI

The Terraform Cloud Run Environment is responsible for executing the operation



Because the Terraform Cloud Run Environment is executing the command. Your provider credentials **need to be configured** in Environment Variables in Terraform Cloud



Environment Variables

These variables are set in Terraform's shell environment using `export`.

Key	Value
<code>AWS_ACCESS_KEY_ID</code> SENSITIVE	Sensitive - write only AWS Access Key ID
<code>AWS_SECRET_ACCESS_KEY</code> SENSITIVE	Sensitive - write only AWS Secret Access Key
+ Add variable	

Remote Backend

When using a remote backend you need to set a Terraform Cloud **Workspaces**

You can set a single workspace via **name**

```
terraform {  
  backend "remote" {  
    hostname = "app.terraform.io"  
    organization = "company"  
  
    workspaces {  
      name = "my-app-prod"  
    }  
  }  
}
```

You can set multiple workspaces via **prefix**

```
terraform {  
  backend "remote" {  
    hostname = "app.terraform.io"  
    organization = "company"  
  
    workspaces {  
      prefix = "my-app-"  
    }  
  }  
}
```

Setting both name and prefix will result in an error.

On terraform apply you will have to choose which workspace you want to apply the operation

```
Initializing the backend...  
  
Successfully configured the backend "remote"! Terraform will automatically  
use this backend unless the backend configuration changes.  
  
The currently selected workspace (default) does not exist.  
This is expected behavior when the selected workspace did not have an  
existing non-empty state. Please enter a number to select a workspace:  
  
1. dev  
2. prod  
  
Enter a value: 
```

Cloud Backend

When using Terraform Cloud as a remote backend state you should instead use the cloud block to configure its usage.

```
terraform {  
  backend "remote" {  
    hostname = "app.terraform.io"  
    organization = "company"  
  
    workspaces {  
      prefix = "my-app-"  
    }  
  }  
}
```

```
terraform {  
  cloud {  
    organization = "company"  
    hostname = "app.terraform.io"  
  
    workspaces {  
      tags = [ "app" ]  
    }  
  }  
}
```

You can still use backend remote for terraform cloud but its recommended to use the cloud backend block.

Backend Initialization

The **-backend-config** flag for **terraform init** can be used for partial backend configuration

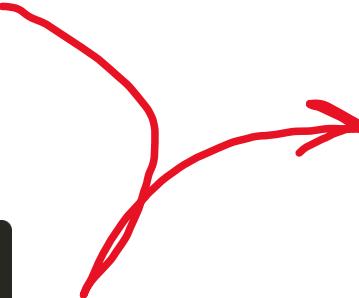
In situations where the backend settings are dynamic or sensitive
and so cannot be statically specified in the configuration file.

main.tf

```
terraform {  
    required_version = "~> 0.12.0"  
  
    backend "remote" {}  
}
```

backend.hcl

```
workspaces { name = "workspace" }  
hostname      = "app.terraform.io"  
organization = "company"
```



```
terraform init -backend-config=backend.hcl
```

terraform_remote_state

terraform_remote_state data source **retrieves the root module output values from another Terraform configuration**, using the latest state snapshot from the remote backend.

Remote Backend

```
data "terraform_remote_state" "vpc" {
  backend = "remote"

  config = {
    organization = "hashicorp"
    workspaces = {
      name = "vpc-prod"
    }
  }
}

resource "aws_instance" "foo" {
  subnet_id = data.terraform_remote_state.vpc.outputs.subnet_id
}
```

Local Backend

```
data "terraform_remote_state" "vpc" {
  backend = "local"

  config = {
    path = "..."
  }
}

resource "aws_instance" "my_server" {
  # ...
  subnet_id = data.terraform_remote_state.vpc.outputs.subnet_id
}
```

- Only the root-level output values from the remote state snapshot are exposed
- Resource data and output values from nested modules are not accessible
- To make a nested module output value accessible as a root module output value, you must **explicitly configure a passthrough in the root module**



```
module "app" {
  source = "..."
}

output "app_value" {
  value = module.app.example
}
```

Alternative to `terraform_remote_state`

`terraform_remote_state` only exposes output values, its user must have access to the entire state snapshot, which often includes some sensitive information.

It recommend explicitly publishing data for external consumption
to a separate location instead of accessing it via remote state

Use a **data source** and reference
resources whenever you can:

```
data "aws_s3_bucket" "selected" {
  bucket = "bucket.test.com"
}

data "aws_route53_zone" "test_zone" {
  name = "test.com."
}

resource "aws_route53_record" "example" {
  zone_id   = "${data.aws_route53_zone.test_zone.id}"
  name      = "bucket"
  type      = "A"

  alias {
    name      = "${data.aws_s3_bucket.selected.website_domain}"
    zone_id  = "${data.aws_s3_bucket.selected.hosted_zone_id}"
  }
}
```

State Locking

Terraform will lock your state for all operations that could write state.
This prevents others from acquiring the lock and potentially corrupting your state

State locking happens automatically on all operations that could write state
You won't see any message that it is happening. If state locking fails

Disabling Locking

You can disable state locking for most commands
with the **-lock** flag but it is not recommended



Terraform does not output when a lock is complete,
However, If acquiring the lock is taking longer than
expected, Terraform will output a status message

Force Unlock

Terraform has a **force-unlock** command to manually unlock the state if unlocking failed.

If you unlock the state when someone else is holding the lock **it could cause multiple writers**.

Force unlock should only be used to unlock your own lock in the situation where automatic unlocking failed.

To protect you, the force-unlock command **requires a unique lock ID**

- Terraform will output this lock ID if unlocking fails

```
terraform force-unlock 1941a539b-ff25-76ef-92d-547ab37b24d -force
```

← **-force** flag will skip use
confirmation

Protecting Sensitive Data

Terraform State file can contain sensitive data eg. long-lived AWS Credentials and is a possible **attack vector** for malicious actors.

Local State

When using local backend, state is stored in plain-text JSON files.

- You need to be careful you don't share this state file with anyone
- You need to be careful you don't commit this file to your git repository

Remote State with Terraform Cloud

When using the Terraform Cloud remote backend:

- That state file is held in memory and is not persisted to disk
- The state file is encrypted-at-rest
- The state file is encrypted-in-transit
- With Terraform Enterprise you have detailed audit logging for tamper evidence

Remote State with Third-Party Storage

You can store state with various third-party backends.

You need to carefully review your backends capabilities to determine if will meet your security and compliance requirements.

Some backends are not by default as secure as they could be:

- eg. With AWS S3 you have to ensure encryption and versioning is turned on, you need to create a custom trail for data events

Terraform Ignore File

When executing a **remote** plan or apply in a CLI-driven run, an archive of your configuration directory is uploaded to Terraform Cloud.

You can define paths to ignore from upload via a **.terraformignore** file at the root of your configuration directory.

If this file is not present, the archive will exclude the following **by default**:

- .git/ directories
- .terraform/ directories (exclusive of .terraform/modules)

.terraformignore works just like a .gitignore with the only difference is that you cannot have multiple .terraformignore files in subdirectories. Only the file in the root directory will be read

Resources

Resources in configuration files represent **infrastructure objects** e.g.
Virtual Machines, Databases, Virtual Network Components, Storage.....

A resource type determines the kind of infrastructure object:
e.g. **aws_instance** is a AWS EC2 instance

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



A resource belongs to a provider.

Some resource types provide a **special timeouts nested block** argument that allows you to customize how long certain operations are allowed to take before being considered to have failed

```
resource "aws_db_instance" "example" {  
    # ...  
  
    timeouts {  
        create = "60m"  
        delete = "2h"  
    }  
}
```



Complex Types

A *complex* type is a type that groups multiple values into a single value.

Complex types are represented by type constructors, but several of them also have shorthand keyword versions.

There are two categories of complex types:

- collection types (for grouping similar values)
 - List, Map, Set
- structural types (for grouping potentially dissimilar values)
 - Tuple , Object

Collection Types

A *collection* type allows multiple values of *one* other type to be grouped together as a single value.
The type of value *within* a collection is called its *element type*.

The three kinds of collection type **list, map, set**

List – Its like an array, you use an integer as the index to retrieve the value

```
variable "planet" {  
  type = list  
  default = ["mars", "earth", "moon"]  
}  
username = var.users[0]
```

Set - Is similar to a list but has no secondary index or preserved ordering, all values must of the same type and will be cast to match based on first element

```
> toset(["a", "b", 3])  
[  
  "a",  
  "b",  
  "3",  
]
```

```
variable "plans" {  
  type = map  
  default = {  
    "PlanA" = "10 USD"  
    "PlanB" = "50 USD"  
    "PlanC" = "100 USD"  
  }  
}  
plan = var.plans["PlanB"]
```

Structural Types

A *structural* type allows multiple values of *several distinct types* to be grouped together as a single value. Structural types require a **schema** as an argument, to specify which types are allowed for which elements.

```
variable "with_optional_attribute" {  
    type = object({  
        a = string          # a required attribute  
        b = optional(string) # an optional attribute  
    })
```

The two kinds of structural type, **object** and **tuple**

Object is a map with more explicit keying

```
object({ name=string, age=number })
```

```
{  
    name = "John"  
    age  = 52  
}
```

Tuple. Multiple return types with parameters

```
tuple([string, number, bool])
```

```
["a", 15, true]
```

Built-in Functions

Terraform language includes a number of built-in functions that you can call from within expressions to transform and combine values

- Numeric Functions
- String Functions
- Collection Functions
- Encoding Functions
- Filesystem Functions
- Date and Time Functions
- Hash and Crypto Functions
- IP Network Functions
- Type Conversion Functions

Numeric Functions

abs returns the absolute value of the given number

```
> abs(23)  
23  
> abs(0)  
0  
> abs(-12.4)  
12.4
```

ceil returns the closest whole number that is greater than or equal to the given value

```
> ceil(5)  
5  
> ceil(5.1)  
6
```

floor returns the closest whole number that is less than or equal to the given value, which may be a fraction

```
> floor(5)  
5  
> floor(4.9)  
4
```

Log returns the logarithm of a given number in a given base

```
> log(50, 10)  
1.6989700043360185  
> log(16, 2)  
4
```

min takes one or more numbers and returns the smallest number from the set

max takes one or more numbers and returns the greatest number from the set

Numeric Functions

parseint parses the given string as a representation of an integer in the specified base and returns the resulting number

pow calculates an exponent, by raising its first argument to the power of the second argument.

```
> pow(3, 2)  
9  
> pow(4, 0)  
1
```

signum determines the sign of a number, returning a number between -1 and 1 to represent the sign

```
> signum(-13)  
-1  
> signum(0)  
0  
> signum(344)  
1
```

```
> parseint("100", 10)  
100  
  
> parseint("FF", 16)  
255  
  
> parseint("-10", 16)  
-16  
  
> parseint("1011111011101111", 2)  
48879  
  
> parseint("aA", 62)  
656  
  
> parseint("12", 2)  
Error: Invalid function argument  
  
Invalid value for "number" parameter:  
cannot parse "12" as a base 2 integer.
```

String Functions

chomp removes newline characters at the end of a string.

```
> chomp("hello\n")
hello
> chomp("hello\r\n")
hello
> chomp("hello\n\n")
hello
```

formatlist produces a list of strings by formatting a number of other values according to a specification string

format produces a string by formatting a number of other values according to a specification string

```
> format("Hello, %s!", "Ander")
Hello, Ander!
> format("There are %d lights", 4)
There are 4 lights
```

```
> formatlist("Hello, %s!", ["Valentina", "Ander", "Olivia", "Sam"])
[
  "Hello, Valentina!",
  "Hello, Ander!",
  "Hello, Olivia!",
  "Hello, Sam!",
]
> formatlist("%s, %s!", "Salutations", ["Valentina", "Ander", "Olivia", "Sam"])
[
  "Salutations, Valentina!",
  "Salutations, Ander!",
  "Salutations, Olivia!",
  "Salutations, Sam!",
]
```

String Functions

indent adds a given number of spaces to the beginnings of all but the first line in a given multi-line string

join produces a string by concatenating together all elements of a given list of strings with the given delimiter

lower converts all cased letters in the given string to lowercase.

regex applies a regular expression to a string and returns the matching substrings

regexall applies a regular expression to a string and returns a list of all matches

```
> "  items: ${indent(2, ["\n  foo,\n  bar,\n"])\n}"  
items: [  
  foo,  
  bar,  
]
```

```
> join(", ", ["foo", "bar", "baz"])  
foo, bar, baz  
> join(", ", ["foo"])  
foo
```

```
> lower("HELLO")  
hello
```

```
> regex("^(?:(?P<scheme>[^:/?#]+):)?(?://(?P<authority>[^/?#]*)?)?", "https://terraform.io/docs/")  
{  
  "authority" = "terraform.io"  
  "scheme" = "https"  
}
```

String Functions

replace searches a given string for another given substring, and replaces each occurrence with a given replacement string

```
> replace("hello world", "/w.*d/", "everybody")
hello everybody
```

split produces a list by dividing a given string at all occurrences of a given separator.

```
> split(",", "foo,bar,baz")
[
  "foo",
  "bar",
  "baz",
]
```

strrev reverses the characters in a string

```
> strrev("hello")
olleh
```

substr extracts a substring from a given string by offset and length

```
> substr("hello world", 1, 4)
ello
```

title converts the first letter of each word in the given string to uppercase

```
> title("hello world")
Hello World
```

trim removes the specified characters from the start and end of the given string

```
> trim("?!hello?!", " !?")
hello
```

String Functions

trimprefix removes the specified prefix from the start of the given string. If the string does not start with the prefix, the string is returned unchanged

trimsuffix removes the specified suffix from the end of the given string

trimspace removes all types of whitespace from both the start and the end of a string

upper converts all cased letters in the given string to uppercase

```
> trimprefix("helloworld", "hello")  
world
```

```
> trimsuffix("helloworld", "world")  
hello
```

```
> trimspace(" hello\n\n")  
hello
```

```
> upper("hello")  
HELLO
```

Collection Functions

alltrue returns true if all elements in a given collection are true or "true". It also returns true if the collection is empty.

```
> alltrue(["true", true])
true
> alltrue([true, false])
false
```

chunklist splits a single list into fixed-size chunks, returning a list of lists

```
> chunklist(["a", "b", "c", "d", "e"], 2)
[
  [
    "a",
    "b",
  ],
  ...
]
```

```
> coalesce("a", "b")
a
> coalesce("", "b")
b
> coalesce(1,2)
1
```

coalesce takes any number of arguments and returns the first one that isn't null or an empty string

anytrue returns true if any element in a given collection is true or "true". It also returns false if the collection is empty

```
> anytrue(["true"])
true
> anytrue([true])
true
> anytrue([true, false])
true
> anytrue([])
false
```

```
> coalescelist(["a", "b"], ["c", "d"])
[ "a",
  "b",
]
```

coalescelist takes any number of list arguments and returns the first one that isn't empty

Collection Functions

compact takes a list of strings and returns a new list with any empty string elements removed →

```
> compact(["a", "", "b", "c"])
[  
  "a",  
  "b",  
  "c",  
]
```

concat takes two or more lists and combines them into a single list →

```
> concat(["a", ""], ["b", "c"])
[  
  "a",  
  "",  
  "b",  
  "c",  
]
```

contains determines whether a given list or set contains a given single value as one of its elements →

```
> contains(["a", "b", "c"], "a")
true
> contains(["a", "b", "c"], "d")
false
```

distinct takes a list and returns a new list with any duplicate elements removed →

```
> distinct(["a", "b", "a", "c", "d", "b"])
[  
  "a",  
  "b",  
  "c",  
  "d",  
]
```

Collection Functions

element retrieves a single element from a list

```
> element(["a", "b", "c"], 3)  
a
```

flatten takes a list and replaces any elements that are lists with a flattened sequence of the list contents

```
> flatten([[["a", "b"], [], ["c"]])  
["a", "b", "c"]
```

length determines the length of a given list, map, or string

Index finds the element index for a given value in a list

```
> index(["a", "b", "c"], "b")  
1
```

keys takes a map and returns a list containing the keys from that map

```
> keys({a=1, c=2, d=3})  
[  
  "a",  
  "c",  
  "d",  
]
```

```
> length([])  
0  
> length(["a", "b"])  
2  
> length({"a" = "b"})  
1  
> length("hello")  
5
```

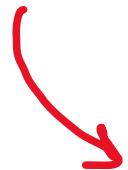
Collection Functions

Lookup retrieves the value of a single element from a map, given its key. If the given key does not exist, the given default value is returned instead.



```
> lookup({a="ay", b="bee"}, "a", "what?")
ay
> lookup({a="ay", b="bee"}, "c", "what?")
what?
```

Matchkeys constructs a new list by taking a subset of elements from one list whose indexes match the corresponding indexes of values in another list



```
> matchkeys(["i-123", "i-abc", "i-def"], ["us-west", "us-east", "us-east"], ["us-east"])
[
  "i-abc",
  "i-def",
]
```

Collection Functions

merge takes an arbitrary number of maps or objects, and returns a single map or object that contains a merged set of elements from all arguments

```
> merge({a="b", c="d"}, {e="f", c="z"})
{
  "a" = "b"
  "c" = "z"
  "e" = "f"
}
```

one takes a list, set, or tuple value with either zero or one elements. If the collection is empty, one returns null. Otherwise, one returns the first element. If there are two or more elements then one will return an error

```
> one([])
null
> one(["hello"])
"hello"
> one(["hello", "goodbye"])
Error: Invalid function argument
```

range generates a list of numbers using a start value, a limit value, and a step value



```
> range(3)
[
  0,
  1,
  2,
]
```

reserve takes a sequence and produces a new sequence of the same length with all of the same elements as the given sequence but in reverse order



```
> reverse([1, 2, 3])
[
  3,
  2,
  1,
]
```

Collection Functions

setintersection function takes multiple sets and produces a single set containing only the elements that all of the given sets have in common. In other words, it computes the intersection of the sets

setproduct function finds all of the possible combinations of elements from all of the given sets by computing the Cartesian product.

Setsubtract function returns a new set containing the elements from the first set that are not present in the second set. In other words, it computes the relative complement of the first set in the second set

```
> setintersection(["a", "b"], ["b", "c"], ["b", "d"])
[ "b", ]
```

```
> setproduct(["development", "staging", "production"], ["app1", "app2"])
[ [ "development", "app1", ],
  [ "development", "app2", ],
  ... ]
```

```
> setsubtract(["a", "b", "c"], ["a", "c"])
[ "b", ]
```

Collection Functions

setunion function takes multiple sets and produces a single set containing the elements from all of the given sets. In other words, it computes the union of the sets

```
> setunion(["a", "b"], ["b", "c"], ["d"])
[  
  "d",  
  "b",  
  "c",  
  "a",  
]
```

slice extracts some consecutive elements from within a list

```
> slice(["a", "b", "c", "d"], 1, 3)
[  
  "b",  
  "c",  
]
```

sort takes a list of strings and returns a new list with those strings sorted lexicographically

```
> sort(["e", "d", "a", "x"])
[  
  "a",  
  "d",  
  "e",  
  "x",  
]
```

Collection Functions

sum takes a list or set of numbers and returns the sum of those numbers

```
> sum([10, 13, 6, 4.5])  
33.5
```



transpose takes a map of lists of strings and swaps the keys and values to produce a new map of lists of strings

```
> transpose({"a" = ["1", "2"], "b" = ["2", "3"]})  
{  
  "1" = [  
    "a",  
  ],  
  "2" = [  
    "a",  
    "b",  
  ],  
  ...  
}
```

values takes a map and returns a list containing the values of the elements in that map

```
> values({a=3, c=2, d=1})  
[  
  3,  
  2,  
  1,  
]
```



zipmap constructs a map from a list of keys and a corresponding list of values

```
> zipmap(["a", "b"], [1, 2])  
{  
  "a" = 1,  
  "b" = 2,  
}
```



Encoding and Decoding Functions

Functions that will **encode** and **decode** for various formats



```
> base64encode("Hello World")
SGVsbG8gV29ybGQ=
```

```
> base64decode("SGVsbG8gV29ybGQ=" )
Hello World
```

- base64encode
- jsonencode
- textencodebase64
- Yamlencode
- base64gzip
- **urlencode**
- base64decode
- csvdecode
- jsondecode
- textdecodebase64
- yamldecode

```
> urlencode("Hello World")
Hello%20World
> urlencode("ø")
%E2%98%83
> "http://example.com/search?q=${urlencode("terraform urlencode")}"
http://example.com/search?q=terraform%20urlencode
```

Filesystem Functions

abspath takes a string containing a filesystem path and converts it to an absolute path. That is, if the path is not absolute, it will be joined with the current working directory

```
> abspath(path.root)  
/home/user/some/terraform/root
```

dirname takes a string containing a filesystem path and removes the last portion from it.

```
> dirname("foo/bar/baz.txt")  
foo/bar
```

pathexpand takes a filesystem path that might begin with a ~ segment, and if so it replaces that segment with the current user's home directory path

```
> pathexpand("~/ssh/id_rsa")  
/home/steve/.ssh/id_rsa  
> pathexpand("/etc/resolv.conf")  
/etc/resolv.conf
```

basename takes a string containing a filesystem path and removes all except the last portion from it

```
> basename("foo/bar/baz.txt")  
baz.txt
```

Filesystem Functions

file reads the contents of a file at the given path and returns them as a string

```
> file("${path.module}/hello.txt")
Hello World
```

fileset enumerates a set of regular file names given a path and pattern

```
> fileset(path.module, "files/*.txt")
[
  "files/hello.txt",
  "files/world.txt",
]
```

filebase64 reads the contents of a file at the given path and returns them as a base64-encoded string

```
> filebase64("${path.module}/hello.txt")
SGVsbG8gV29ybGQ=
```

templatefile reads the file at the given path and renders its content as a template using a supplied set of template variables

```
%{ for addr in ip_addrs ~}
backend ${addr}:${port}
%{ endfor ~}
```

```
> templatefile("${path.module}/backends.tpl", { port = 8080, ip_addrs = ["10.0.0.1", "10.0.0.2"] })
backend 10.0.0.1:8080
backend 10.0.0.2:8080
```

fileexists determines whether a file exists at a given path ↗

Date and Time Functions

formatdate converts a timestamp into a different time format

```
> formatdate("DD MMM YYYY hh:mm ZZZ", "2018-01-02T23:12:01Z")
02 Jan 2018 23:12 UTC
> formatdate("EEEE, DD-MMM-YY hh:mm:ss ZZZ", "2018-01-02T23:12:01Z")
Tuesday, 02-Jan-18 23:12:01 UTC
> formatdate("EEE, DD MMM YYYY hh:mm:ss ZZZ", "2018-01-02T23:12:01-08:00")
Tue, 02 Jan 2018 23:12:01 -0800
> formatdate("MMM DD, YYYY", "2018-01-02T23:12:01Z")
Jan 02, 2018
> formatdate("HH:mm:ss", "2018-01-02T23:12:01Z")
11:12pm
```

timeadd adds a duration to a timestamp, returning a new timestamp

```
> timeadd("2017-11-22T00:00:00Z", "10m")
2017-11-22T00:10:00Z
```

timestamp returns a UTC timestamp string in RFC 3339 format

```
> timestamp()
2018-05-13T07:44:12Z
```

Hash and Crypto Functions

Generate Hashes and cryptographic strings.

```
> bcrypt("hello world")
$2a$10$D5grTTzcsqyvAeIAnY/mY0IqliCoG7eAMX0/oFcuD.iErkksEbcA
```

- base64sha256
- base64sha512
- **bcrypt**
- filebase64sha256
- filebase64sha512
- filemd5
- filesha1
- filesha256
- filesha512
- **md5**
- rsadecrypt
- **sha1**
- sha256
- sha512
- **uuid**
- uuidv5

IP Network Functions

cidrhost calculates a full host IP address for a given host number within a given IP network address prefix

```
> cidrhost("10.12.112.0/20", 16)  
10.12.112.16  
> cidrhost("10.12.112.0/20", 268)  
10.12.113.12  
> cidrhost("fd00:fd12:3456:7890:00a2::/72", 34)  
fd00:fd12:3456:7890::22
```

cidrnetmask converts an IPv4 address prefix given in CIDR notation into a subnet mask address

```
> cidrnetmask("172.16.0.0/12")  
255.240.0.0
```

cidrsubnet calculates a subnet address within given IP network address prefix

```
> cidrsubnet("172.16.0.0/12", 4, 2)  
172.18.0.0/16  
> cidrsubnet("10.1.2.0/24", 4, 15)  
10.1.2.240/28  
> cidrsubnet("fd00:fd12:3456:7890::/56", 16, 162)  
fd00:fd12:3456:7800:a200::/72
```

cidrsubnets calculates a sequence of consecutive IP address ranges within a particular CIDR prefix.

```
> cidrsubnets("10.1.0.0/16", 4, 4, 8, 4)  
[  
  "10.1.0.0/20",  
  "10.1.16.0/20",  
  "10.1.32.0/24",  
  "10.1.48.0/20",  
]
```

Type Conversion Functions

can evaluates the given expression and returns a boolean value indicating whether the expression produced a result without any errors

```
> can(local.foo.bar)  
true
```

defaults a specialized function intended for use with input variables whose type constraints are object types or collections of object types that include optional attributes

nonsensitive takes a sensitive value and returns a copy of that value with the sensitive marking removed, thereby exposing the sensitive value

```
output "sensitive_example_hash" {  
  value = nonsensitive(sha256(var.sensitive_example))  
}
```

sensitive takes any value and returns a copy of it marked so that Terraform will treat it as sensitive, with the same meaning and behavior as for sensitive input variables.

```
locals {  
  sensitive_content = sensitive(file("${path.module}/sensitive.txt"))  
}
```

Type Conversion Functions

tobool converts its argument to a boolean value

```
tobool("true")  
true
```

tomap converts its argument to a map value

```
> tomap({"a" = 1, "b" = 2})  
{  
  "a" = 1  
  "b" = 2  
}
```

toset converts its argument to a set value.

```
> toset(["a", "b", "c"])[  
  "a",  
  "b",  
  "c",  
]
```

try evaluates all of its argument expressions in turn and returns the result of the first one that does not produce any errors

tolist converts its argument to a list value

```
> tolist(["a", "b", 3])[  
  "a",  
  "b",  
  "3",  
]
```

tonumber converts its argument to a number value

```
> tonumber("1")  
1
```

toString converts its argument to a set value

```
> toString(true)  
"true"
```

```
locals {  
  example = try(  
    [toString(var.example)],  
    tolist(var.example),  
  )  
}
```

Terraform Cloud



Terraform Cloud

Terraform Cloud features:

- Manages your state files
- History of previous runs
- History of previous states
- Easy and secure variable injection
- Tagging
- Run Triggers (chaining workspaces)
- Specify any version of terraform per workspace
- Global state sharing
- Commenting on Runs
- Notifications via Webhooks, Email, Slack
- Organization and Workspace Level Permissions
- Policy as Code (via Sentinel Policy Sets)
- MFA,
- Single Sign On (SSO) (at Business tier)
- Cost Estimation (at Teams and Governance Tier)
- Integrations with ServiceNow, Splunk, K8, and custom Run Tasks

Terraform Cloud is an application that helps teams use Terraform together.

The screenshot shows the Terraform Cloud interface. At the top, there's a navigation bar with tabs for 'ExamPro', 'Workspaces' (which is selected), 'Registry', 'Settings', and 'HCP'. Below the navigation is a breadcrumb trail: ExamPro / Workspaces / example-workspace / Overview / ...

The main content area displays the 'example-workspace'. It shows basic information: Resources 0, Terraform version 1.0.4, and Updated 18 days ago. There's also a status indicator 'Unlocked' and an 'Actions' dropdown menu.

A 'Latest Run' section is shown, titled 'Testing'. It details a run triggered by andrewbrown 18 days ago via UI. The run summary includes:

- Policy checks: Upgrade
- Estimated cost change: Upgrade
- Plan duration: Less than a minute
- Resources to be changed: +0 ~0 -0

A 'See details' button is present.

At the bottom of the workspace view, it says 'Current as of the most recent state version.' and 'Workspaces configured with a Terraform version earlier than 0.12 Learn about upgrading.'

On the right side, there are sections for 'Metrics (last 20 runs)', 'Tags (0)', and a note that 'Tags have not been added to this workspace.'

Terraform Cloud is available as a hosted service at app.terraform.io

Terraform Cloud - Terms

Organizations

An organization is a collection of workspaces

Workspaces

A workspace belongs to an organization

A workspace represents a unique environment or stack.

Teams

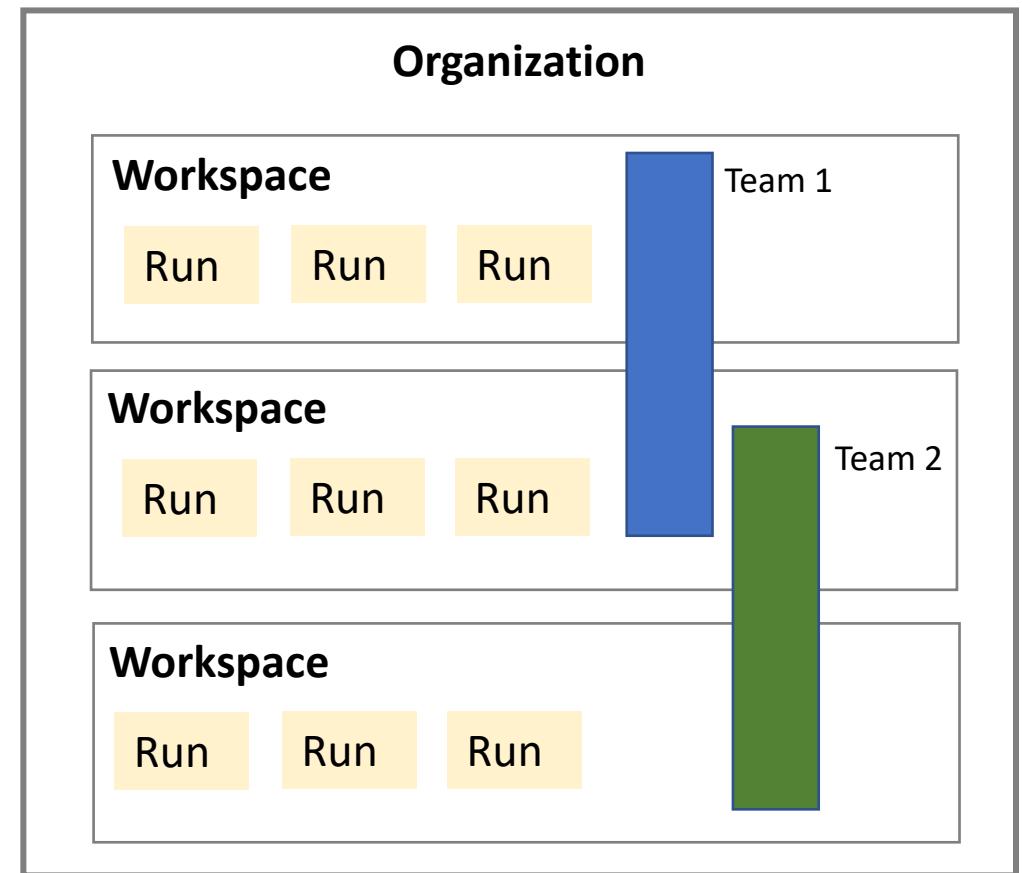
A team is composed of multiple members (users).

A team can be assigned to workspaces

Runs

A run represents a single-run of the terraform run environment that is operating on a execution plan.

Runs can be UI/VCS driven API driven or CLI driven



Terraform Cloud Run Workflows

Terraform Cloud offers **3 types** of Cloud Run Workflows

When you create a workspace you have **to choose a workflow**

Choose your workflow



Version control workflow Most common

Store your Terraform configuration in a git repository, and trigger runs based on pull requests and merges.

[Learn More](#)



CLI-driven workflow

Trigger remote Terraform runs from your local command line.

[Learn More](#)



API-driven workflow

A more advanced option. Integrate Terraform into a larger pipeline using the Terraform API.

[Learn More](#)



Terraform Cloud Run Workflows

Terraform Cloud offers **3 types** of Cloud Run Workflows

UI/VCS Driven (User Interface and Version Control System)

Terraform Cloud is integrated with a specific branch in your VCS eg. Github via webhooks.

Whenever pull requests are submitted for the branch speculative plans are generated

Whenever a merge occurs to that branch, then a run is triggered on Terraform Cloud

API-Driven (Application Programming Interface)

Workspaces are not directly associated with a VCS repo, and runs are not driven by webhooks on your VCS provider

A third-party tool or system will trigger runs via upload a configuration file via the Terraform Cloud API

The configuration file is a bash script that is packaged in an archive (.tar.gz). You are pushing a **Configuration Version**

CLI-Driven (Command Line Interface)

Runs are triggered by the user running terraform CLI commands e.g. `terraform apply` and `plan` locally on their own machine.

Terraform Cloud - Organization-Level Permissions

Organization-Level Permissions manage certain resources or settings **across an organization**

- **Manage Policies** - create, edit, and delete the organization's Sentinel policies
- **Manage Policy Overrides** - override soft-mandatory policy checks.
- **Manage Workspaces** - create and administrate all workspaces within the organization
- **Manage VCS Settings** - set of VCS providers and SSH keys available within the organization

Organization Owners

Every organization has organization owner(s).

This is a special role that has every available permission and some actions only available to owners:

- Publish private modules
- Invite users to organization
- Manage team membership
- View all secret teams
- Manage organization permissions
- Manage all organization settings
- Manage organization billing
- Delete organization
- Manage agent

Terraform Cloud - Workspace-Level Permissions

Workspace-Level Permissions manage resource and settings for a **specific workspace**

General Workspace Permissions

Granular permissions you can apply to a user via **custom workspace permissions**

- Read runs
- Queue plans
- Apply runs
- Lock and unlock workspaces
- Download sentinel mocks
- Read variable
- Read and write variables
- Read state outputs
- Read state versions
- Read and write state versions

Fixed Permission Sets

Premade permissions for quick assignment.

- **Read**
 - Read runs
 - Read variables
 - Read state versions
- **Plan**
 - Queue Plans
 - Read variables
 - Read state versions
- **Write**
 - Apply runs
 - Lock and unlock workspaces
 - Download Sentinel mocks
 - Read and write variables
 - Read and write state versions

Workspace Admins

A workspace admin is a special role that grants the all level of permissions and some workspace-admin-only permissions:

- Read and write workspace settings
- Set or remove workspace permissions of any team
- Delete workspace

Terraform Cloud – API Tokens

Terraform Cloud supports three types of API Tokens. User, Team and Organization Tokens.

Organization API Tokens

- Have permissions across the entire organization
- Each organization can have one valid API token at a time
- Only organization owners can generate or revoke an organization's token.
- Organization API tokens are designed for creating and configuring workspaces and teams.
 - Not recommended as an all-purpose interface to Terraform Cloud

Team API Tokens

- allow access to the workspaces that the team has access to, without being tied to any specific user
- Each team can have **one** valid API token at a time
- any member of a team can generate or revoke that team's token
- When a token is regenerated, the previous token immediately becomes invalid
- designed for performing API operations on workspaces.
- same access level to the workspaces the team has access to

User API Tokens

- Most flexible token type because they inherit permissions from the user they are associated with
- Could be for a real user or a machine user.

Terraform Cloud – API Tokens

Shoot video of documentation because Access levels is too hard to show

Terraform Cloud – Private Registry

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

Terraform Cloud's private module registry helps you share Terraform modules across your organization. It includes support for:

- module versioning
- a searchable and filterable list of available modules
- a configuration designer

All users in your organization can view your private module registry.

Authentication

You can use either a user token or a team token for authentication, but the type of token you choose may grant different permissions.

Using `terraform login` will obtain a user token

To use a team token you'll need to manually set it in your `terraform` CLI configuration file

Terraform Cloud – Cost Estimation

Cost Estimation is a feature to get a monthly cost of resources display along side your runs.

Cost Estimation is available with Teams and Governance plan and above

Cost Estimation is only for specific cloud resources for AWS, Azure and GCP.

The screenshot shows the Terraform Cloud interface. At the top, a green checkmark indicates a "Plan finished" status 3 minutes ago, with 1 resource added. Below it, a green checkmark indicates a "Cost estimation finished" status 3 minutes ago, with 1 of 1 estimated resource costing \$16.56/mo. A red underline highlights the cost values (\$0.023, \$16.56, and +\$16.56). The interface also shows a timeline from queued to finished, a resource table with columns for Resource, Name, Hourly Cost, Monthly Cost, and Monthly Delta, and a dropdown for the aws_instance resource.

RESOURCE	NAME	HOURLY COST	MONTHLY COST	MONTHLY DELTA
aws_instance	ubuntu	\$0.023	\$16.56	+\$16.56

You can could use a Sentinel Policy to assert the expectation that resource are under a particular cost.

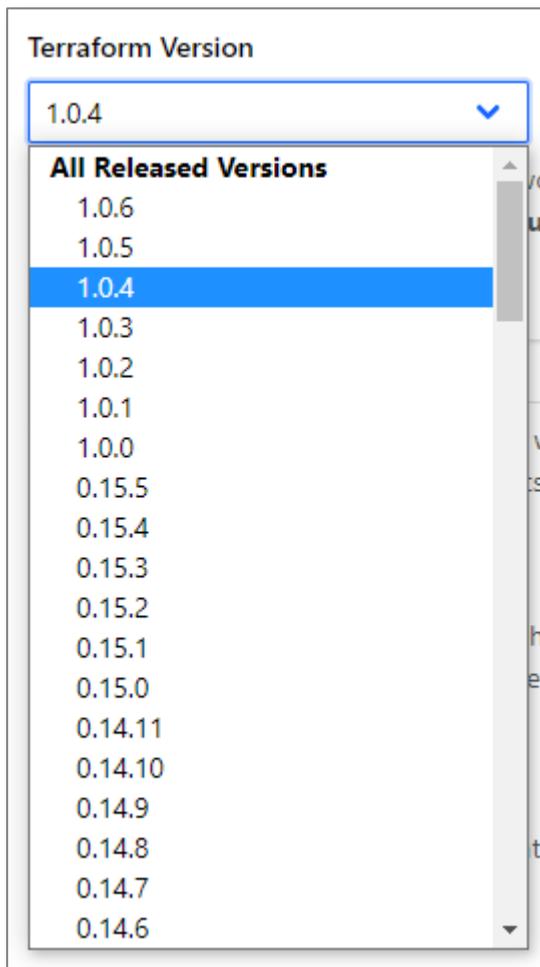
Terraform Cloud – Cost Estimation

Cost Estimation is only for specific cloud resources for AWS, Azure and GCP.

- aws_alb
- aws_autoscaling_group
- aws_cloudhsm_v2_hsm
- aws_cloudwatch_dashboard
- aws_cloudwatch_metric_alarm
- aws_db_instance
- aws_dynamodb_table
- aws_ebs_volume
- aws_elasticache_cluster
- aws_elasticsearch_domain
- aws_elb
- aws_instance
- aws_kms_key
- aws_lb
- aws_rds_cluster
- azurerm_app_service_custom_hostname_binding
- azurerm_app_service_environment
- azurerm_app_service_plan
- azurerm_app_service_virtual_network_swift_connection
- azurerm_cosmosdb_sql_database
- azurerm_databricks_workspace
- azurerm_firewall
- azurerm_hdinsight_hadoop_cluster
- azurerm_hdinsight_hbase_cluster
- azurerm_hdinsight_interactive_query_cluster
- azurerm_hdinsight_kafka_cluster
- azurerm_hdinsight_spark_cluster
- azurerm_integration_service_environment
- azurerm_linux_virtual_machine
- azurerm_linux_virtual_machine_scale_set
- azurerm_managed_disk
- azurerm_mariadb_server
- azurerm_mssql_elasticpool
- azurerm_mysql_server
- azurerm_postgresql_server
- azurerm_sql_database
- azurerm_virtual_machine
- azurerm_virtual_machine_scale_set
- azurerm_windows_virtual_machine
- azurerm_windows_virtual_machine_scale_set
- google_compute_disk
- google_compute_instance
- google_sql_database_instance

Terraform Cloud – Workflow Options

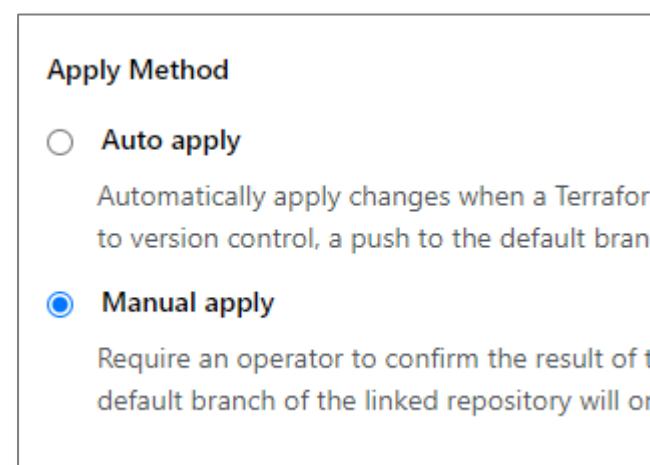
You can choose any version of Terraform for a Workspace



You can choose to share state globally across your organization



You can choose to auto approve runs (skip manual approval)



Migrating Default Local State

To migrate a local, Terraform project that only uses the **default** workspace to Terraform Cloud

- Create a workspace in Terraform Cloud
- Replace your Terraform Configuration with **a remote backend**.



Run `terraform init`, and copy the existing state by typing “**yes**”

Do you want to copy existing state to the new backend?

Pre-existing state was found while migrating the previous “local” backend to the newly configured “remote” backend. No existing state was found in the newly configured “remote” backend. Do you want to copy this state to the new “remote” backend? Enter “**yes**” to copy and “**no**” to start with an empty state.

Terraform Cloud VCS Integration

Terraform Cloud can integrate with the following Version Control Systems (VCS):

- Github
- Github (OAuth)
- Github Enterprise
- GitLab
- GitLab EE and CE
- Bitbucket Cloud
- Bitbucket Server and Data Center
- Azure DevOps Service
- Azure DevOps Server

Create a new Workspace

Workspaces allow you to organize infrastructure and collaborate on Terraform runs.

1 Connect to VCS

2 Choose a repository

3 Configure settings

Connect to a version control provider

Choose the version control provider that hosts the Terraform configuration for this workspace.



VERSION

GitHub.com

GitHub Enterprise

GitHub.com (Custom)



Runs require CLI or API



Runs require CLI or API



To perform runs in this workspace, you must connect to a version control provider. You can add a VCS connection later if you change your mind.

Terraform Cloud Run Environment

When Terraform Cloud executes your terraform plan it runs them in its own Run Environment.

What is a Run Environment?

A run environment is a Virtual Machine or container intended for the execution of code for a specific runtime environment. A run environment is essentially a code build server.



The Terraform Cloud Run Environment is a **single-use Linux machine** running on the `x86_64` architecture and the details of its internal implementation is not known.

Terraform Cloud will inject the following environment variables automatically on each run:

- **TFC_RUN_ID** - a unique identifier for this run (e.g. "run-CKuwsxMGgMd3W7Ui").
- **TFC_WORKSPACE_NAME** - name of the workspace used in this run
- **TFC_WORKSPACE_SLUG** - full slug of the configuration used in this run org/workspace eg. "acme-corp/prod-load-balancers".
- **TFC_CONFIGURATION_VERSION_GIT_BRANCH** - name of the branch used eg. "main"
- **TFC_CONFIGURATION_VERSION_GIT_COMMIT_SHA** - full commit hash of the commit used
- **TFC_CONFIGURATION_VERSION_GIT_TAG** - name of the git tag used eg. 1.1.0

These Env Vars can be accessed by defining a variable:

```
variable "TFC_RUN_ID" {}
```

Terraform Cloud Agents

Terraform Cloud Agents is a paid feature of the **Business** plan to allow Terraform Cloud to **communicate with isolated, private or on-premise infrastructure.**

This is useful for on-premise infrastructure types such as vSphere, Nutanix, or OpenStack

The agent architecture is pull-based, so no inbound connectivity is required

Any agent you provision will poll Terraform Cloud for work and carry out execution of that work locally.

- Agents currently only support x86_64 bit Linux operating systems.
- You can also run the agent within Docker using the official Terraform Agent Docker container.
- Agents support Terraform versions 0.12 and above.
- System requires at least 4GB of free disk space (for temporary local copies) and 2 GB of memory
- Needs access to make outbound requests on HTTPS (TCP Port 443) to
 - App.terraform.io
 - Registry.terraform.io
 - Releases.hashicorp.com
 - archivist.terraform.io

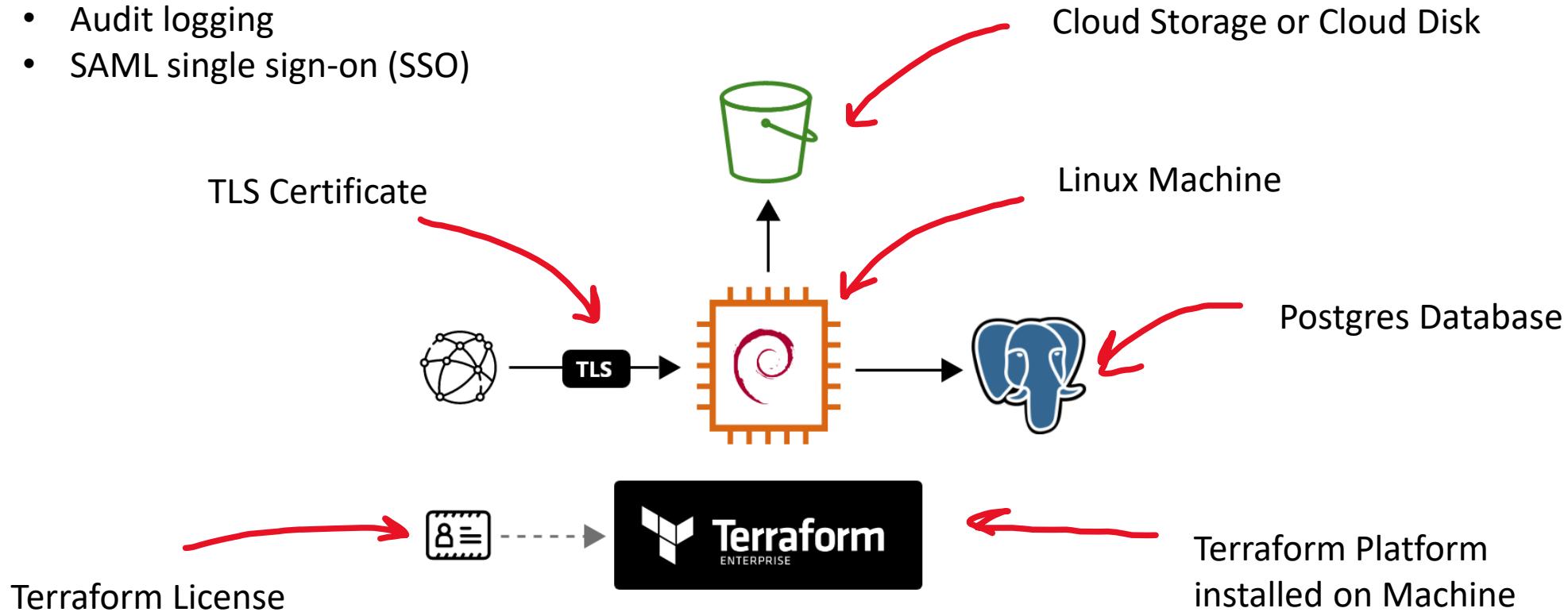
Terraform Enterprise



Terraform Enterprise is our **self-hosted distribution** of Terraform Platform

Terraform Enterprise offers a private instance of the Terraform Platform application with benefits such :

- no resource limits
- with additional enterprise-grade architectural features
 - Audit logging
 - SAML single sign-on (SSO)



Terraform Enterprise Requirements

Operational mode: - how data should be stored

- External Services
 - Postgres
 - and AWS S3 Bucket, GCP Cloud Storage Bucket or Azure Blob Storage or Minio Object Storage
- Mounted Disk – stores data in a separate directory on a host intended for an external disk e.g. EBS, iSCSI
- Demo – Stores all the data on the instance, data can be backed up with snapshots, *not recommend for production use*

Credentials – Ensure you have credentials to use Enterprise and have Secure connection

- Terraform Enterprise License – You obtain a license from HashiCorp
- TLS Certificate and Private Key – You need to prove you're your own TLS Certificate

Linux Instance – Terraform Enterprise is designed to run on Linux:

Supported OS

- Debian 7.7+
- Ubuntu 14.04.5 / 16.04 / 18.04 / 20.04
- Red Hat Enterprise Linux 7.4 - 7.9
- CentOS 7.4 - 7.9
- Amazon Linux 2014.03 / 2014.09 / 2015.03 / 2015.09 / 2016.03 / 2016.09 / 2017.03 / 2017.09 / 2018.03 / 2.0
- Oracle Linux 7.4 - 7.9

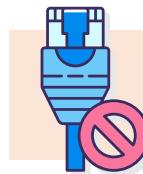
Hardware Requirements

- At least 10GB of disk space on the root volume
- At least 40GB of disk space for the Docker data directory (defaults to /var/lib/docker)
- At least 8GB of system memory
- At least 4 CPU cores

Air Gapped Environments

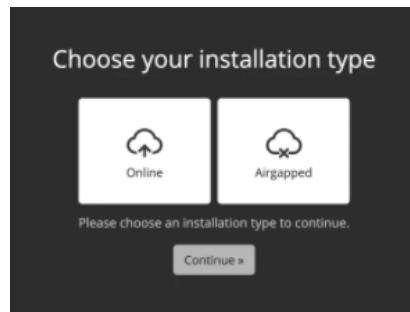
What is Air Gap?

Air Gap or disconnected network is a network security measure employed on one or more computers to ensure that a secure computer network is physically isolated from unsecure networks e.g. Public Internet



No internet. No outside connectivity

Industries in the **Public Sector** (e.g. government, military) or **large enterprise** e.g.
Finance and Energy often employ air gap networks.



HashiCorp Terraform Enterprise **supports** an installation type for Air Gapped Environments

To install or update Terraform Enterprise you will supply an “air gap bundle” which is an archive of a Terraform Enterprise release version

Terraform Cloud Features and Pricing

Open-Source Software (OSS)	Cloud	Self-Hosted	
Free	Teams and Governance	Business	Enterprise (self-hosted)
IaC, Workspaces, Variables, Runs, Resource Graph, Providers, Modules, Public Module Registry			
	Remote State, VCS Connection, Workspace Mgmt., Secure Variable Storage, Remote Runs, Private Module Registry		
		Team Management, Sentinel Policy as Code Management , Cost Estimation	
			Single Sign On (SSO), Audit Logging
			Self Hosted Agents
			Configuration Designer, ServiceNow Integration
1 Current Runs	2 Current Runs	Unlimited Current Runs	
Local CLI	Cloud		Private
Community	Bronze	Silver, Gold	
\$0 up to 5 users	Starting at \$20 user/month	Contact Sales	Contact Sales

Workspaces

Workplaces allows you to manage multiple environments or alternate state files.
eg. Development, Production

There are two variants of workspaces:

- CLI Workspaces – A way of managing alternate state files (locally or via remote backends)
- Terraform Cloud Workspaces – acts like completely separate working directories



*Think of workspaces as being similar to having different branches in a git repository
Workspaces are technically equivalent to renaming your state file*

In Terraform 0.9 used to call workspaces “environments”

By default, you already have a single workspace in your local backend called **default**

```
terraform workspace list  
* default
```

The default workspace can never be deleted

Workspace Internals

Depending if you a local or remote backend changes how the state file is stored

Local State

Terraform stores the workspace states in a folder called **terraform.tfstate.d**

In practice individuals or very small teams will have been known to commit these files to their repositories.

Remote State

The workspace files are stored directly in the configured backend.



using a remote backend instead is recommended when there are multiple collaborators

Current Workspace Interpolation

You can reference the current workspace name via **terraform.workspace**

```
resource "aws_instance" "example" {  
  count = "${terraform.workspace == "default" ? 5 : 1}"  
  
  # ... other arguments  
}
```

```
resource "aws_instance" "example" {  
  tags = {  
    Name = "web - ${terraform.workspace}"  
  }  
  # ... other arguments  
}
```

Multiple Workspaces

A Terraform configuration has a backend that:

- defines **how operations are executed**
- where **persistent data is stored** e.g. Terraform State

Multiple workspaces are currently supported by the following backends

- AzureRM
- Consul
- COS
- GCS
- Kubernetes
- Local
- Manta
- Postgres
- Remote
- S3

Certain backends support *multiple* named workspaces

- allowing multiple states to be associated with a single configuration.
- The configuration still has only one backend, but multiple distinct instances of that configuration to be deployed without configuring a new backend or changing authentication credentials.

Terraform Cloud Workspaces

You can create Workspaces on Terraform Cloud

The image shows the Terraform Cloud Workspaces interface. On the left, there is a list of workspaces under the heading "Workspaces 1 total". The single workspace listed is "example-workspace", which has a status of "Planned and finished" and was last updated "10 days ago". A red arrow points from this workspace row towards the right side of the screen. On the right, there is a detailed view of the "example-workspace" overview. This view includes sections for "Latest Run" (which shows a summary of the run, including policy checks, estimated cost change, plan duration, and resources to be changed), "Metrics (last 20 runs)", "Average plan duration", "Average apply duration", "Total failed runs", "Policy check failures", and "Tags (0)". The top navigation bar includes links for "ExamPro", "Workspaces", "Registry", "Settings", "HCP", "Upgrade Now", and a user profile icon.

Terraform Cloud Workspaces

You can see a history of previous runs.



Triggered via CLI

Applied

andrewbrown triggered a **destroy** run from CLI 10 days ago

Plan finished 10 days ago Resources: 0 to add, 0 to change, 21 to destroy

Apply finished 10 days ago Resources: 0 added, 0 changed, 21 destroyed

andrewbrown 10 days ago Run confirmed

Comment: Leave feedback or record a decision.

Add Comment

Comment on each run

ExamPro Workspaces Registry Settings HCP

ExamPro / Workspaces / example-workspace / Runs

example-workspace

No workspace description available. [Add workspace description](#).

Resources 0 Terraform version 1.0.4 Updated 2 days ago

Overview Runs States Variables Settings Actions Unlocked

Current Run

Testing CURRENT 2 days ago

#run-tsc5nFbEHkHcYwWD | refresh-only run | andrewbrown triggered via UI

Run List

Testing CURRENT 2 days ago

#run-tsc5nFbEHkHcYwWD | refresh-only run | andrewbrown triggered via UI

I wan to test 9 days ago

#run-DGRtmvTzs79e2sco | andrewbrown triggered via UI

Triggered via CLI 10 days ago

#run-y9wjhXsoNR1DMKaY | destroy run | andrewbrown triggered via CLI

Terraform Cloud Workspaces

Apply finished 10 days ago Resources: 0 added, 0 changed, 21 destroyed

Started 10 days ago > Finished 10 days ago

[View raw log](#) [Top](#) [Bottom](#) [Expand](#) [Full screen](#)

```
module.vpc.aws_vpc.this[0]: Destruction complete after 1s
aws_instance.my_example_server: Still destroying... [id=i-0d7b0ac2b95e7dcf3, 20s elapsed]
google_compute_instance.vm_instance: Still destroying... [id=projects/my-app-321517/zones/us-central1-a/instances/12345678901234567890]
google_compute_instance.vm_instance: Destruction complete after 21s
aws_instance.my_example_server: Still destroying... [id=i-0d7b0ac2b95e7dcf3, 30s elapsed]
aws_instance.my_example_server: Still destroying... [id=i-0d7b0ac2b95e7dcf3, 40s elapsed]
aws_instance.my_example_server: Still destroying... [id=i-0d7b0ac2b95e7dcf3, 50s elapsed]
aws_instance.my_example_server: Still destroying... [id=i-0d7b0ac2b95e7dcf3, 1m0s elapsed]
aws_instance.my_example_server: Still destroying... [id=i-0d7b0ac2b95e7dcf3, 1m10s elapsed]
aws_instance.my_example_server: Still destroying... [id=i-0d7b0ac2b95e7dcf3, 1m20s elapsed]
aws_instance.my_example_server: Still destroying... [id=i-0d7b0ac2b95e7dcf3, 1m30s elapsed]
aws_instance.my_example_server: Still destroying... [id=i-0d7b0ac2b95e7dcf3, 1m40s elapsed]
aws_instance.my_example_server: Still destroying... [id=i-0d7b0ac2b95e7dcf3, 1m50s elapsed]
aws_instance.my_example_server: Still destroying... [id=i-0d7b0ac2b95e7dcf3, 2m0s elapsed]
aws_instance.my_example_server: Still destroying... [id=i-0d7b0ac2b95e7dcf3, 2m10s elapsed]
aws_instance.my_example_server: Still destroying... [id=i-0d7b0ac2b95e7dcf3, 2m20s elapsed]
aws_instance.my_example_server: Still destroying... [id=i-0d7b0ac2b95e7dcf3, 2m30s elapsed]
aws_instance.my_example_server: Destruction complete after 2m31s

Apply complete! Resources: 0 added, 0 changed, 21 destroyed.
```

[State versions created:](#)

ExamPro/example-workspace#sv-U3ovoRxJhYd4tYGS (Aug 19, 2021 13:47:17 pm)

Triggered via CLI

Applied Triggered via CLI 10 days ago

andrewbrown triggered a **destroy** run from CLI 10 days ago

[Run Details](#)

Plan finished 10 days ago Resources: 0 to add, 0 to change, 21 to destroy

andrewbrown triggered a **destroy** run from CLI 10 days ago

[View raw log](#) [Top](#) [Bottom](#) [Expand](#) [Full screen](#)

Apply finished 10 days ago Resources: 0 added, 0 changed, 21 destroyed

andrewbrown 10 days ago

Run confirmed

Comment: Leave feedback or record a decision.

[Add Comment](#)

Plan finished 10 days ago Resources: 0 to add, 0 to change, 21 to destroy

Started 10 days ago > Finished 10 days ago

[View raw log](#) [Top](#) [Bottom](#) [Expand](#) [Full screen](#)

```
# module.vpc.aws_route.public_internet_gateway[0] will be destroyed
- resource "aws_route" "public_internet_gateway" {
  - destination_cidr_block = "0.0.0.0/0" -> null
  - gateway_id             = "igw-0ee6e95a7d6d8dd81" -> null
  - id                      = "rtb-0ba0139a4da3076701080289494" -> null
  - origin                 = "CreateRoute" -> null
  - route_table_id          = "rtb-0ba0139a4da307670" -> null
  - state                  = "active" -> null

  - timeouts {
    - create = "5m" -> null
  }
}

# module.vpc.aws_route_table.private[0] will be destroyed
- resource "aws_route_table" "private" {
  - arn           = "arn:aws:ec2:us-east-1:318412259206:route-table/rtb-0c8db62d83010"
  - id            = "rtb-0c8db62d83010a992" -> null
  - owner_id      = "318412259206" -> null
  - propagating_vgw_ids = [] -> null
  - route         = [] -> null
}
```

[Download Sentinel mocks](#) [Sentinel mocks can be used for testing your Sentinel policies](#)

Terraform Cloud Workspaces

You can see a history of previously held states (snapshots)

Download the state file

The screenshot illustrates the Terraform Cloud Workspaces interface. On the left, the 'States' tab is selected in the navigation bar. Below it, three previous states are listed:

- Triggered via CLI** (#sv-U3ovoRxJhYd4tYGS) triggered by andrewbrown from Terraform 10 days ago.
- Triggered via CLI** (#sv-KdsGMMYSh9nFHymf) triggered by andrewbrown from Terraform 11 days ago.
- New state #sv-vMbQ54cjrvzCvJt3** triggered by andrewbrown from Terraform 11 days ago.

A red arrow points from the second state entry to a detailed view of its state file on the right. This view shows the JSON structure of the state file, with a specific resource highlighted in red:

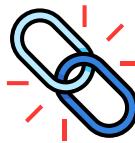
```
1 {  
2   "version": 4,  
3   "terraform_version": "1.0.4",  
4   "serial": 1,  
5   "lineage": "810d6321-1f58-6f83-b",  
6   "outputs": {},  
7   "resources": [  
8     {  
9       "mode": "managed",  
10      "type": "aws_instance",  
11      "name": "my_example_server",  
12      "provider": "provider[\"regi",  
13      "instances": [  
14        {  
15          "arn": "arn:aws:ec2:us-east-1:318412259206:internet-gateway/igw-0ee6d6d8d81",  
16          "attached_disk": [],  
17          "boot_disk": [  
18            {  
19              "auto_delete": true,  
20              "device_name": "persistent-disk-0",  
21            }  
22          ]  
23        }  
24      ]  
25    }  
26  ]  
27}
```

A red arrow also points from the 'Download' button at the top right of this view to a larger 'Changes in this version' panel on the far right, which displays the Terraform diff output:

```
Changes in this version  
1 resources: [  
2   1: {  
3     instances: [  
4       0: {  
5         attributes: {  
6           advanced_machine_features: [],  
7           allow_stopping_for_update: null,  
8           arn: "arn:aws:ec2:us-east-1:318412259206:internet-gateway/igw-0ee6d6d8d81",  
9           attached_disk: [],  
10          boot_disk: [  
11            {  
12              auto_delete: true,  
13              device_name: "persistent-disk-0",  
14            }  
15          ]  
16        }  
17      }  
18    ]  
19  ]  
20}
```

A red arrow points from the text 'A diff of what changed since last state' at the bottom left towards the changes panel.

Terraform Cloud Run Triggers



Terraform Cloud provides a way to **connect your workspace to one or more workspaces** via **Run Triggers** within your organization, known as "source workspaces".

Run Triggers

allow runs to queue automatically in your workspace on successful apply of runs in any of the source workspaces. You can connect each workspace to up to 20 source workspaces.

Run triggers are designed for workspaces that rely on information or infrastructure produced by other workspaces.

If a Terraform configuration uses data sources to read values that might be changed by another workspace, run triggers let you explicitly specify that external dependency.

The screenshot shows the Terraform Cloud interface. The top navigation bar includes the logo, workspace name 'run-triggers-demo', 'Workspaces', 'Modules', 'Settings', and user profile. The main area shows a workspace named 'terraform-random-pet'. A sidebar menu is open, with 'Run Triggers' highlighted in purple. The 'Runs' tab is selected. The 'Current Run' section shows a run queued by 'terraform-random-separator workspace'. The 'Run List' section shows another run queued by the same workspace. Both runs include details like commit hash '1f911a2'.

terraform workspace

terraform workspace **list**

list all existing workspaces
current workspace is indicated using an asterisk (*)

```
$ terraform workspace list
default
* development
production
```

terraform workspace **show**

Show the current

```
$ terraform workspace show
development
```

terraform workspace **select**

Switch to target workspace

```
$ terraform workspace select default
Switched to workspace "default"
```

terraform workspace **new**

Create and **switch** to workspace

```
$ terraform workspace new example
Created and switched to workspace "example"
```

terraform workspace **delete**

Delete target workspace

```
$ terraform workspace delete example
Deleted workspace "example"
```

Workspaces Differences

Terraform Cloud workspaces and local working directories serve the same purpose, but they store their data differently:

Component	Local Terraform	Terraform Cloud
Terraform configuration	On disk	In linked version control repository, or periodically uploaded via API/CLI
Variable values	As .tfvars files, as CLI arguments, or in shell environment	In workspace
State	On disk or in remote backend	In workspace
Credentials and secrets	In shell environment or entered at prompts	In workspace, stored as sensitive variables

Sentinel



Sentinel is an embedded **policy-as-code framework** integrated with the Terraform Platform

What is Policy as Code?

When you write code to automate regulatory or governance policies

Features of Sentinel:

Embedded - enable policy enforcement in the data path to actively reject violating behavior instead of passively detecting.

Fine-grained, condition-based policy - Make policy decisions based on the condition of other values

Multiple Enforcement Levels - Advisory, soft and hard-mandatory levels allow policy writers to warn on or reject behaviour

External Information - Source external information to make holistic policy decisions

Multi-Cloud Compatible - Ensure infrastructure changes are within business and regulatory policy across multiple providers



Sentinel is a paid service part of
Team & Governance upgrade package

Policy as Code

Sentinel is built around the idea and provides all the benefits of policy as code

Benefits of Policy as Code

- **Sandboxing** – The ability to create guardrails to avoid dangerous actions or remove the need of manual verification
- **Codification** – The policies are well documented and exactly represent what is enforced
- **Version Control** – Easy to modify or iterate on policies, with a chain of history of changes over time
- **Testing** - syntax and behavior can be easily validated with Sentinel, ensuring policies are configured as expected
- **Automation** – policies existing as code allows you direct integrate policies in various systems to auto-remediate, notify.

Sentinel and Policy as Code

- **Language** - All Sentinel policies are written using the Sentinel language
 - Designed to be non-programmer and programmer friendly, embeddable and safe.
- **Development** - Sentinel provides a CLI for development and testing.
- **Testing** - Sentinel provides a test framework designed specifically for automation.

Sentinel Policy Language Example

Examples of policies you can write and enforce for various providers

AWS

- Restrict owners of the aws_ami data source
- Enforce mandatory tags on taggable AWS resources
- Restrict availability zones used by EC2 instances
- Disallow 0.0.0.0/0 CIDR block in security groups
- Restrict instance types of EC2 instances
- Require S3 buckets to be private and encrypted by KMS keys
- Require VPCs to have DNS hostnames enabled

Google Cloud Platform

- Enforce mandatory labels on VMs
- Disallow 0.0.0.0/0 CIDR block in network firewalls
- Enforce limits on GKE clusters
- Restrict machine type of VMs

VMWare

- Require Storage DRS on datastore clusters
- Restrict size and type of virtual disks
- Restrict CPU count and memory of VMs
- Restrict size of VM disks
- Require NFS 4.1 and Kerberos on NAS datastores

Azure

- Enforce mandatory tags of VMs
- Restrict publishers of VMs
- Restrict VM images
- Restrict the size of Azure VMs
- Enforce limits on AKS clusters
- Restrict CIDR blocks of security groups

Cloud-Agnostic

- Allowed providers
- Prohibited providers
- Limit proposed monthly costs
- Prevent providers in non-root modules
- Require all modules have version constraints
- Require all resources be created in modules in a private module registry
- Use most recent versions of modules in a private module registry

Sentinel Policy Language Example

Example of Sentient Policy that restricts the Availability Zones for EC2 instances on AWS

Import policy language functions



```
import "tfplan-functions" as plan

# Allowed EC2 Zones
# Include "null" to allow missing or computed values
allowed_zones = [
    "us-east-1a", "us-east-1b", "us-east-1c", "us-east-1d", "us-east-1e", "us-east-1f",
]

# Get all EC2 instances
allEC2Instances = plan.find_resources("aws_instance")
```

Specify AZs



Get all VMs



Filter that restricts AZ use for VMs



```
# Filter to EC2 instances with violations
# Warnings will be printed for all violations since the last parameter is true
violatingEC2Instances = plan.filter_attribute_not_in_list(allEC2Instances,
    "availability_zone", allowed_zones, true)
```

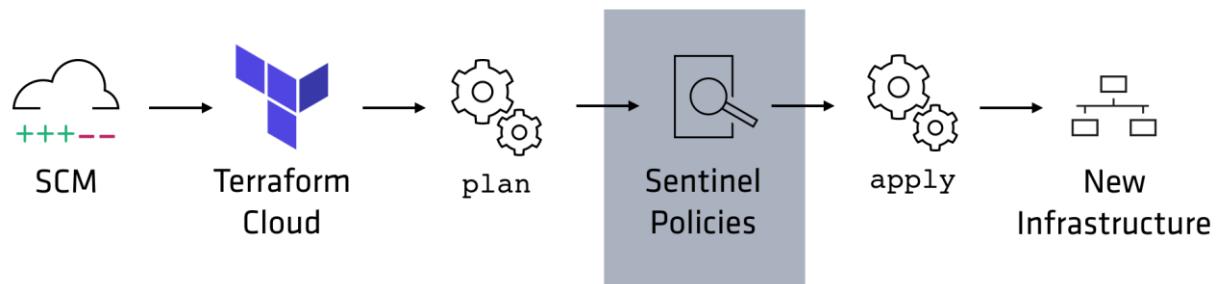
Define the rule



```
# Main rule
main = rule {
    length(violatingEC2Instances["messages"]) is 0
}
```

Sentinel with Terraform

Sentinel can be *integrated* with Terraform via Terraform Cloud as part of your IaC provisioning pipeline.



Within Terraform Cloud you can **create a Policy Set** and apply these to and the apply a Policy Set to a Terraform Cloud Workspace

Organization settings

- General
- Teams
- Users
- Integrations
- Cost Estimation
- Policies
- Policy Sets**
- Security
- Agents
- API Tokens
- Authentication
- SSH Keys
- SSO
- Version control

Connect a Policy Set

Policy sets are groups of Sentinel policies which may be enforced on workspaces.

1 Connect to VCS 2 Choose a repository 3 Configure settings

Choose a repository

Choose the repository that hosts your Terraform source code. We'll watch this for commits and pull requests.

1 repository of 23 total

Filter learn-sentinel-policy-upload

Can't see your repository? Enter its ID below, e.g. acme-corp/infrastructure :

acme-corp/infrastructure

Hashicorp Packer



Packer is a developer tool to **provision a build image** that will be stored in a repository.

Using a build image before you deploy provides:

- immutable infrastructure
- your VMs in your fleet are all one-to-one in configuration
- Faster deploys for multiple servers after each build
- Earlier detection and intervention of package changes or deprecation of old technology



Packer configures a machine via a Packer Template

- Packer Templates use the HashiCorp Configuration Language (HCL)

Packer Template File

Packer configures a machine or container via a **Packer Template**

- Packer Templates use the HashiCorp Configuration Language (HCL)

The **source** says *where* and what kind of image to build

In this case it will create an **EBS-backed AMI**

The image will be stored directly in AWS under EC2 Images

The **build** allows us to provide configuration scripts.

Packer supports a wide range of **provisioners**:

- Chef, Puppet, Ansible, PowerShell, Bash, Salt....

Post-provisioners run after the image is built.

They can be used to upload artifacts or re-package

```
variable "ami_id" {  
  type  = string  
  default = "ami-01e78c5619c5e68b4"  
}  
  
locals {  
  app_name = "httpd"  
}  
  
source "amazon-ebs" "httpd" {  
  ami_name      = "my-server-${local.app_name}"  
  instance_type = "t2.micro"  
  region        = "us-west-2"  
  source_ami    = "${var.ami_id}"  
  ssh_username  = "ec2-user"  
  tags = {  
    Env  = "DEMO"  
    Name = "Pmy-server--${local.app_name}"  
  }  
}  
  
build {  
  sources = ["source.amazon-ebs.httpd"]  
  
  provisioner "shell" {  
    script = "script/script.sh"  
  }  
  
  post-processor "shell-local" {  
    inline = ["echo foo"]  
  }  
}
```

Terraform and Packer Integration

To integrate packer there two steps:

1. Building the image:



Packer is not a service but a development tool so you need to manually run packer or automate the building of images with a build server running packer

2. Referencing the Image:

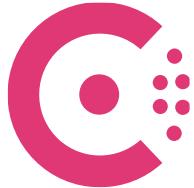
Once an image is built you can reference the image as a **Data Source**

eg. for AWS AMI we can ask it to match
on regex and to select the most recent



```
data "aws_ami" "example" {  
  executable_users = ["self"]  
  most_recent     = true  
  name_regex       = "^myami-\d{3}"  
  owners           = ["self"]  
}
```

Terraform and Consul



Consul is a **service networking platform** which provides:

- **service discovery** – central registry for services in the network
 - Allows for direct communication, no single-point of failure via load balancers
- **service mesh** – managing network traffic between services
 - A communication layer on top of your container application, *think middleware*
- **application configuration capabilities**

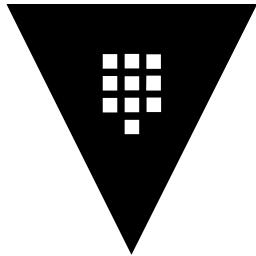


Consul is useful when you have a micro-service or **service-oriented architecture** with **hundred or thousands of services** (containerized apps or workloads)

Consul integrates with Terraform in the following ways:

- Remote backend
 - Consul has a Key Value (KV) Store to store configurations
- Consul Provider

HashiCorp Vault



What is HashiCorp Vault?

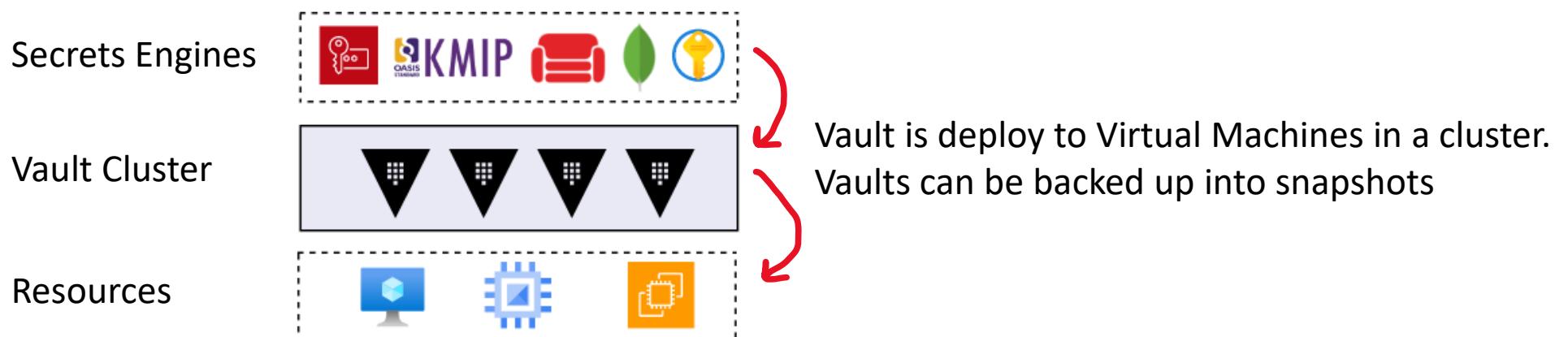
Vault is a tool for **securely accessing secrets** from multiple secrets data stores.

Vault is deployed to a server where:

- Vault Admins can directly manage secrets
- Operators (developers) can access secrets via an API

Vault provides a unified interface:

- to any secret
 - AWS Secrets, Consul Key Value, Google Cloud KMS, Azure Service principles....
- providing tight access control
 - Just-in-Time (JIT) - reducing surface attack based on range of time
 - Just Enough Privilege (JeP) - reducing service attack by providing least-permissive permissions
- recording a detailed audit log – tamper evidence



Terraform and Vault

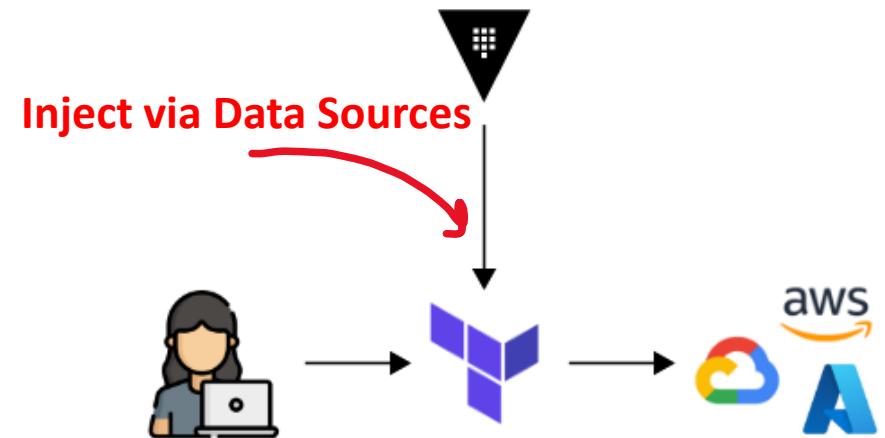
When a developer is working with Terraform and they need to deploy to a provider eg. AWS, they will need AWS credentials

AWS Credentials are long-lived, meaning a user generates a key and secret, and they are usable until they are deleted.

The AWS Credentials reside on the developer's local machine, and so that machine is at risk of being compromised by a malicious actors looking to steal the credentials.

If we could:

- provide credentials Just-In-Time (JIT)
 - expire the credentials after a short amount of time (short-lived)
- we can reduce the attack surface area of the local machine.



Vault can be used to **inject** short-lived secrets at the time of terraform apply

Vault Injection via Data Source

A vault server is provisioned

A vault engine is configured e.g. AWS Secret Engine

Vault will create a machine user for AWS

Vault will generate short-live AWS credentials from that machine user

Vault will manage and apply the AWS Policy

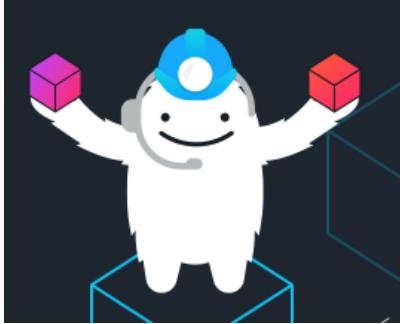
Within our Terraform we can provide a Data Source to Vault

```
data "vault_aws_access_credentials" "creds" {  
    backend = data.terraform_remote_state.admin.outputs.backend  
    role    = data.terraform_remote_state.admin.outputs.role  
}  
  
provider "aws" {  
    region      = var.region  
    access_key  = data.vault_aws_access_credentials.creds.access_key  
    secret_key  = data.vault_aws_access_credentials.creds.secret_key  
}
```

When Terraform Apply is run, it will pull short-lived credentials to be used scope for the duration of the current run.

Everytime you run apply you will get new short-lived credentials.

Gruntwork



Gruntwork is a software company that builds
DevOps tools **that extends or leverages Terraform**

- **Infrastructure as Code Library** - 300,000 lines of reusable, battle-tested, production-ready infrastructure code for AWS, GCP, and Azure, bash scripts, Go programs, and other tools
- **Terragrunt** - a thin wrapper that provides extra tools for keeping your configurations DRY, working with multiple Terraform modules, and managing remote state
- **TerraTest** – a testing framework for Infrastructure provisioned with Terraform.
- **Gruntwork Landing Zone for AWS** – a collection of baselines for multi-account security on AWS
- **Gruntwork Pipelines** - security-first approach to a CI/CD pipeline for infrastructure
- **Gruntwork Reference Architecture** - opinionated, battle-tested, best-practices way to assemble the code from the Infrastructure as Code Library

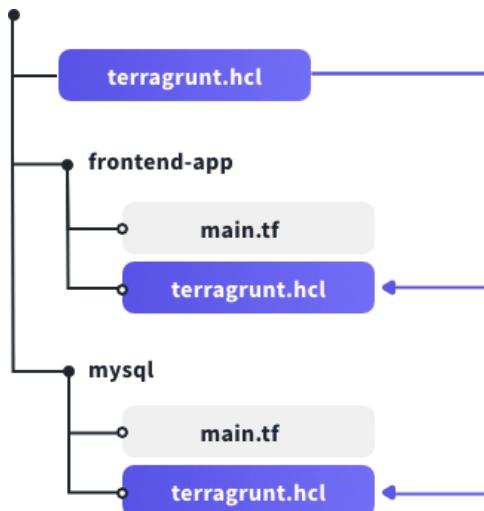
Terragrunt



Terragrunt is a thin wrapper for Terraform that provides extra tools for:

- keeping your configurations **DRY**
- working with multiple Terraform modules
- managing remote state.

<https://terragrunt.gruntwork.io>



Don't-Repeat-Yourself (DRY) is a **programming methodology** to abstract repeated code into function, modules or libraries and often in **isolate files** to reduce code complexity effort and errors.

Terragrunt

Terragrunt can work around the limitations of Terraform in variety of ways.

One example is **providing dynamic values to a provider** on definition which cannot be done with Terraform.

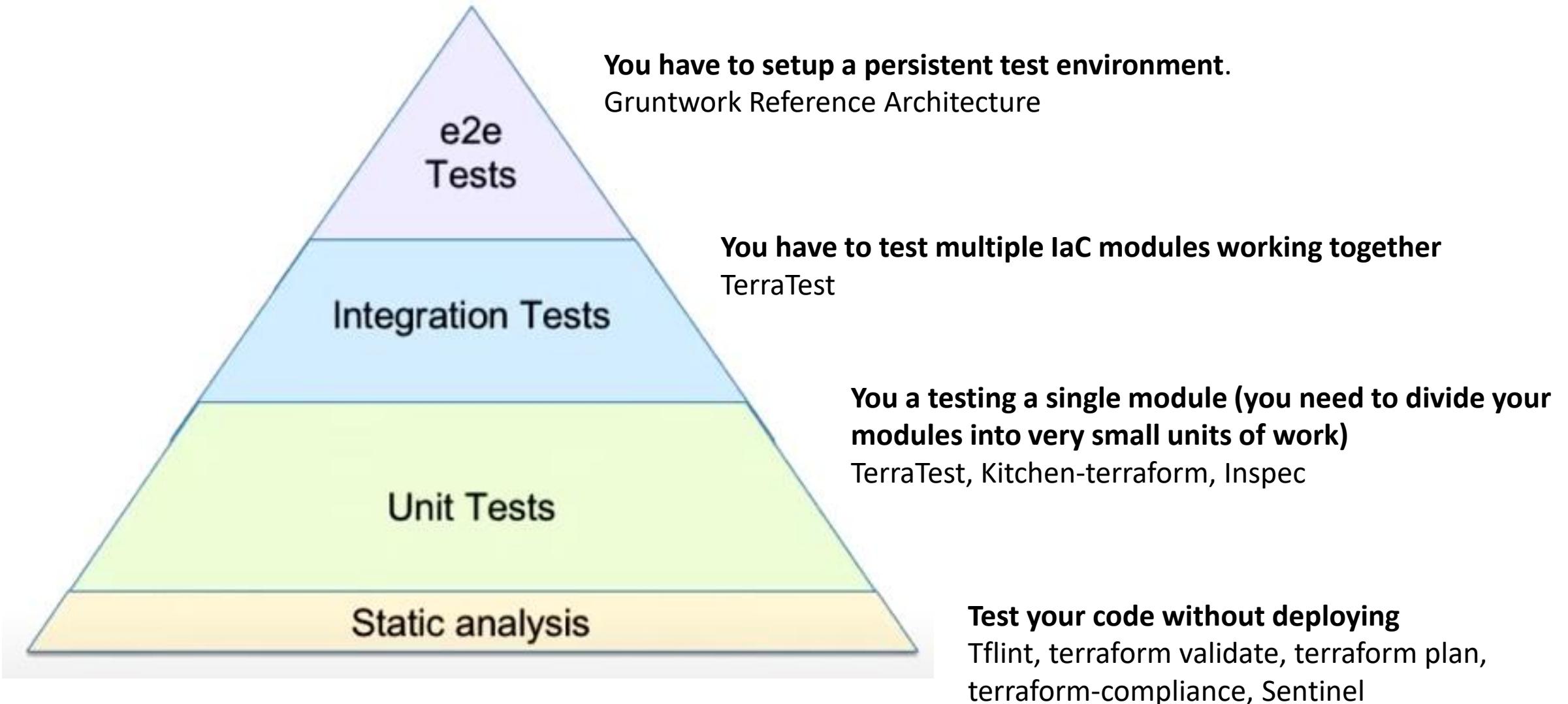
Terragrunt supports better granularity for modules by reducing lots of boilerplate.

This is important when you need to start writing UnitTests for your infrastructure



```
# Indicate where to source the terraform module from.  
# The URL used here is a shorthand for  
# "tfr://registry.terraform.io/terraform-aws-modules/vpc/aws?version=3.5.0".  
# Note the extra `/` after the protocol is required for the shorthand  
# notation.  
terraform {  
    source = "tfr:///terraform-aws-modules/vpc/aws?version=3.5.0"  
}  
  
# Indicate what region to deploy the resources into  
generate "provider" {  
    path = "provider.tf"  
    if_exists = "overwrite_terragrunt"  
    contents = <<EOF  
provider "aws" {  
    region = "us-east-1"  
}  
EOF  
}  
  
# Indicate the input values to use for the variables of the module.  
inputs = {  
    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 = false  
}
```

Testing in Terraform



Testing in Terraform

Jim from GruntWork has a complete talk on Automated Testing for IaC.



Filmed at
QCon San Francisco 2019

Brought to you by
InfoQ

Technique	Strengths	Weaknesses
Static analysis	<ul style="list-style-type: none">1. Fast2. Stable3. No need to deploy real resources4. Easy to use	<ul style="list-style-type: none">1. Very limited in errors you can catch2. You don't get much confidence in your code solely from static analysis
Unit tests	<ul style="list-style-type: none">1. Fast enough (1 – 10 min)2. Mostly stable (with retry logic)3. High level of confidence in individual units	<ul style="list-style-type: none">1. Need to deploy real resources2. Requires writing non-trivial code
Integration tests	<ul style="list-style-type: none">1. Mostly stable (with retry logic)2. High level of confidence in multiple units working together	<ul style="list-style-type: none">1. Need to deploy real resources2. Requires writing non-trivial code3. Slow (10 – 30 min)
End-to-end tests	<ul style="list-style-type: none">1. Build confidence in your entire architecture	<ul style="list-style-type: none">1. Need to deploy real resources2. Requires writing non-trivial code3. Very slow (60 min – 240+ min)*4. Can be brittle (even with retry logic)*

Terra Test

TerraTest allows you to **perform Unit Test and Integration Tests** on your Infrastructure.

It tests your infrastructure by:

- temporarily deploying it
- validating the results
- then tearing down the test environment.

Tests in TerraTest are written in Golang.
You don't need to know much about Go to write tests.

```
// Validate the "Hello, World" app is working
func validateHelloWorldApp(t *testing.T, terraformOptions *terraform.Options) {
    // Run `terraform output` to get the values of output variables
    url := terraform.Output(t, terraformOptions, "url")

    // Verify the app returns a 200 OK with the text "Hello, World!"
    expectedStatus := 200
    expectedBody := "Hello, World!"
    maxRetries := 10
    timeBetweenRetries := 3 * time.Second
    http_helper.HttpGetWithRetry(t, url, nil, expectedStatus, expectedBody, maxRetries, timeBetweenRetries)
}
```

Terragrunt

Terragrunt

gruntwork-io / **terraform-aws-service-catalog** Private

<> Code Issues 59 Pull requests 18 Actions

master terraform-aws-service-catalog / modules / landingzone

marinalimeira Merge pull request #794 from gruntwork-io/renovate/gruntv...

..

account-baseline-app	Remove duplicate
account-baseline-root	Terraform fmt
account-baseline-security	Merge pull request
gruntwork-access	Bump test tf ver
iam-users-and-groups	Merge branch 'r'

Terragrunt - Features

Keep your Terraform Code DRY

Instead of repeating code, or having to create and publish your own modules, reference a tf file in a github repo at a specific location and instantiate by passing inputs

Keep your remote state configuration DRY

Adds support for expressions, variables, or function for remote backends e.g. S3

Keep your CLI Flags DRY - Pass along repeated CLI Flags

Execute Terraform commands on multiple modules at once

Instead of having to execute manually apply command in each subfolder, perform all at once.

Work with multiple AWS accounts

Assume roles for a security account to access resources across multiple accounts

Before and After Hooks - define custom actions that will be called either before or after execution of the terraform command

Auto-Init - automatically run Terraform Init when it would be required before a plan or apply.

Auto-Retry - automatically rerun a terraform command, in cases where rerunning would solve the problem e.g. networking problem

Terraform vs Terragrunt

Infrastructure as Code Library

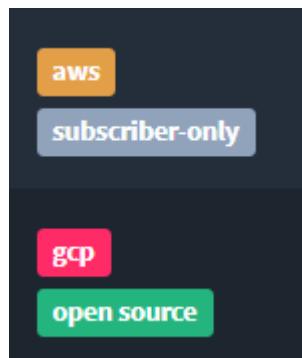
The Infrastructure as Code Library contains several open-source repositories hosted on Github.

These libraries are battle-tested and opinionated but are well maintained by Gruntwork for production-grade use cases, aligned to industry best practices.

The IaC Library covers:

- AWS, GCP, and Azure
- written in Terraform, Go, Bash, and Python

There is a mix of libraries for:
• paid (subscriber-only) →
• free (open-source) →



What's in the Library?

The Infrastructure as Code Library consists of 40+ GitHub repos, some open source, some private, each of which contains reusable, battle-tested infrastructure code for AWS, GCP, and Azure, written in Terraform, Go, Bash, and Python. Check out [How to use the Gruntwork Infrastructure as Code Library](#) to see how it all works.

Search: *E.g., Try searching for Kubernetes, Vault, Jenkins, Kafka, Lambda, AWS, GCP, Azure, etc...*

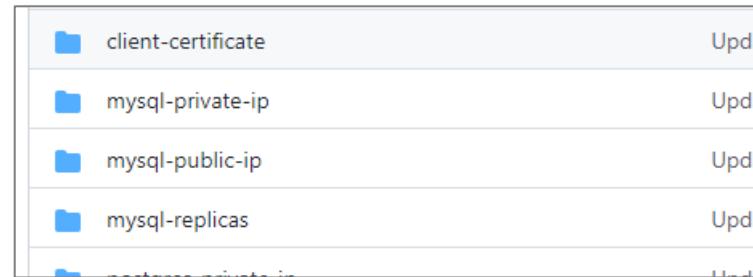
Name	Description	Tags
AWS VPC	Create a best-practices Virtual Private Cloud (VPC) on AWS. Includes multiple subnet tiers, netwo...	aws subscriber-only
Monitoring and Alerting	Configure monitoring, log aggregation, and alerting using CloudWatch, SNS, and S3. Includes Slack...	aws subscriber-only
EC2 Container Service (ECS)	Deploy a best-practices ECS Cluster and run Docker containers on it as ECS Services. Includes zer...	aws subscriber-only
EC2 Kubernetes Service (EKS)	Deploy a best-practices EKS cluster and run Docker containers on it as Kubernetes services. Suppo...	aws subscriber-only
Kubernetes Services	Package services into a best-practices deployment for Kubernetes. Supports zero-downtime, roll...	aws gcp azure open source
Auto Scaling Group	Run stateless and stateful services on top of an Auto Scaling Group. Supports zero-downtime, roll...	aws subscriber-only

IaC Library Core Concepts

All published IaC Libraries have the following:

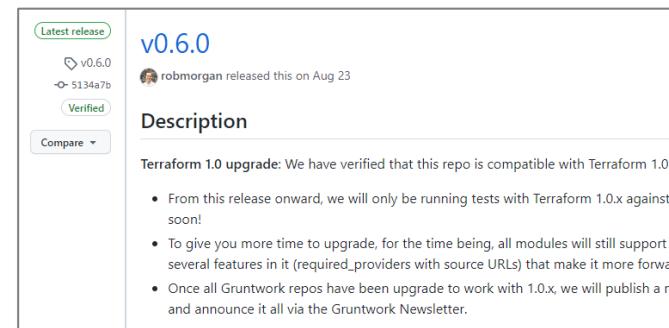
Examples

All IaC Libraries contain multiple examples for common production-grade use-cases



Versioning

All IaC Libraries following strict versioning guidelines will have well written descriptions.



Automated Tests



Gruntwork Reference Architecture

An **opinionated, battle-tested, best-practices way** to assemble infrastructure code from the **Infrastructure as Code Library**

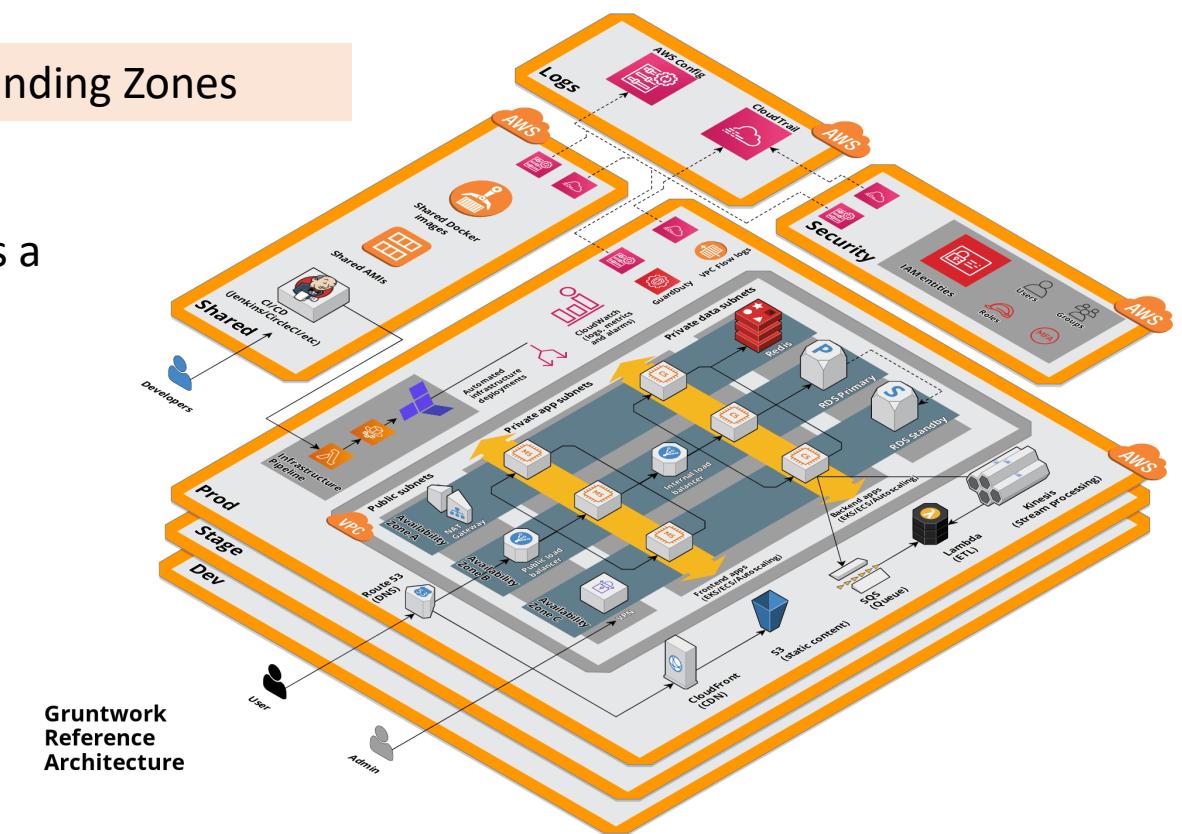
Reference Architecture *is the Terraform Equivalent* to AWS Landing Zones

What is a Landing Zone?

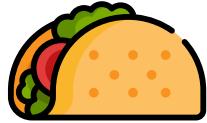
A Landing Zone is foundational infrastructure that establishes a security and operation **production-grade** baseline for your cloud workloads.

Gruntwork offers a compliant version of Reference Architecture based on the **CIS** AWS Foundations Benchmark
Center of Internet Security

The Reference Architect is only available to Gruntwork Subscribers (*paid customers of Gruntwork*)



Terraform Automation & Collaboration (TACOS)



Terraform Automation & Collaboration Software (TACOS) is a framework for solving IaC at scale for organizations.

TACOs can provide the following:

- GitOps experience for your IaC (e.g. CI/CD, Pull Requests)
- Role-Based-Access-Controls (RBACs)
- Hierarchical Structure for organizing projects
- Policy as Code (PoC)
- Security (They store your statefile securely)
- Governance (e.g. Product Catalog)

