**Terraform (IaC tool)**

Configuration files are written in **HCL** (Hashicorp Configuration Language)

Registry has 1000+ providers.

Language is **Declarative** (what you see is what you get so it is explicit). Leaves zero change of misconfiguration.

**Idempotent** – no matter how many times you run the IaC code you will end up with the same state that is expected. I.e., editing a VM will not result in two VM’s but one updated VM.

Common commands:

**INIT > FMT > VALIDATE > PLAN > APPLY > DESTROY**

**Execution plan** (produced from Apply or Plan commands) is a manual review of what will be added, changed, or destroyed. Type Yes to approve the changes (-auto-approve to skip this).

You can **visualise an Execution plan as a graph** using the **terraform graph** command. Terraform outputs to a GraphViz file. Extension file name is gv.

Diagram

Description automatically generated

Need a **Terraform block** in the script (includes source and version) and **Provider block** (i.e., AWS, Azure) to start with then configure options for provider.

**Terraform Files**

**.tfvars** file allows you to set default values for your variables.

**terraform.tfstate** file defines the current state of your infrastructure as declared in your .tf script files. There is also a backup (**terraform.tfstate.backup)** of this file in case it is ever lost or deleted.

**State files** allow you to detect configuration drift.

**.terraform.lock.hcl** file is automatically created and maintained after running *terraform init.* The file contains details of the provider and the version.

*provider "registry.terraform.io/hashicorp/aws" {*

*version = "4.49.0"*

It is a good idea to **divide files** in terraform script to make it more readable. These are the common files:

main.tf, providers.tf, variables.tf, outputs.tf a ReadMe.md file. And .tf files for each type of resource (i.e., S3.tf, IAM.tf)

**Locals** are like variables but hard coded. Use string interpolation to reference the local.

*Locals {*

*project\_name = “bishop”*

*}*

*tags = {*

*Name = “alex-${local.project\_name}”*

*}*

Results in a tag of Name: alex-bishop

**Outputs** display outputted values such as *private or public IP* and display to the console after running a terraform apply. Find information after resource provisioning. Can add a **description** and marking as **sensitive** hides from terminal – but still visible in state file.

**Modules** allow you to configure a bunch of resources using samples i.e., for a VPC module creates a VPC, subnets, selects the AZ’s and adds tags.

**Terraform Cloud**

***terraform workspace list***command shows the workspace you are working in. It will be a single workspace called *default* by default. Every project comes with a default workspace, and you cannot delete it. You can create multiple workspaces like development, production etc to have different variables for your infrastructure across different environments.

To move to the cloud, you must define a backend. The backend currently being used by default is the ***local*** backend. In the initial terraform settings block is where the backend is defined. You need to add the **remote backend**, the **hostname** (default name, can be found online). The **organization** (created once signed up to terraform cloud) and the **workspace** name (multiple workspaces can be added to an organization).

*terraform {*

***backend*** *= “remote” {*

***hostname*** *= “app.terraform.io”*

***organization*** *= “alex-org”*

***workspaces*** *{*

*name = “first-workspace”*

*}*

*}*

Once this is added to the script enter command ***terraform login*** which creates an API token to verify the login.

**Credential providers** also need to be added to the Terraform cloud console for authentication. For AWS this would be Access key ID, Secret access key and Region. These three are added as **Environment variables**.

**Terraform Provisioners**

Allow you to install software, edit files and provision machines created with terraform.

Terraform allows you to work with two provisioners:

**Cloud-init** – industry standard. When a VM is launched with a cloud service provider you provide a YAML or Bash script in the User Data section.

**Packer** – automated image-builder. Provide a configuration file to create and provision the machine image and the image is delivered to a repository for use. Similar to EC2 image builder. Packer is good because it is cloud agnostic (use same config for any provider).

Note that Packer provisioners will do something that won’t be reflected in the terraform state file. Cloud-init scripts and cloud provider features can be a better alternative to this.

No direct support for third-party provisioning tools: **Chef**, **Puppet** or **Salt** as Terraform considers them bad practice.

**Terraform Provisioners** use case is when a task is done **once** to set up a database. Third party provisioning/configuration management tools such an **Ansible** are better for **repeatable** tasks for ongoing maintenance.

**Local-exec** allows you to execute **local commands** after a resource is provisioned. The machine executing terraform is where the command executes (local pc, build server such as AWS CodeBuild or Jenkins, terraform cloud run environment).

**Example use case**: after provisioning a VM you need to supply the public IP to a third-party security service and add the IP. You can do this using locally installed third-party CLI on your build server.

Different to **Outputs** as Local-exec runs commands on your local machine. Commonly used to trigger configuration management e.g., Ansible, Chef, Puppet.

*resource “aws\_instance” “web” {*

*provisioner “local-exec” {*

*command = “echo ${self.private\_ip} >> private\_ips.txt”*

*}*

*}*

**Remote-exec** allows you to execute commands on a **target resource** (such as a provisioned VM) after a resource is provisioned. More complex tasks recommend using **cloud-init.** Also recommended to bake Golden Images via Packer or EC2 image builder.

Modes (one can be used at a time):

**Inline** = list of command strings

**Script** = local script copied to the remote resource and executed

**Scripts** = local scripts copied to the remote resource and executed in order.

Remote-exec has **no** ‘interpreter’ mode.

*resource “aws\_instance” “web” {*

*provisioner “remotel-exec” {*

*inline = [*

*“puppet apply”*

*“consul join ${aws\_instance.web.private.ip}”*

*]*

*}*

*}*

Or…

*resource “aws\_instance” “web” {*

*provisioner “remotel-exec” {*

*scripts = [*

*“./setup-users.sh”*

*“/home/alex/Desktop/bootstrap”*

*]*

*}*

*}*

**File provisioner** is used to copy files or directories to the newly created resource.

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

Destination – where you want the files on remote machine.

Content – can be used instead of Source to provide content directly

**Connection** block tells a provisioner or resource how to establish a connection e.g., SSH or WinRM.

**null\_resources** – a placeholder for resources that have no specific association to a provider.

**Terraform Providers**

API mapping to the service. Providers power all terraforms resources. AWS, Azure, GCP etc. The provider must be declared in every terraform script. In some cases, multiple providers can be used.

**Terraform Modules**

Reusable sample terraform scripts for deploying common infrastructure. Saves you typing everything out and remembering every aspect of a resource. i.e., VPC and subnets, AZ, CIDR ranges, NAT Gateways. Reduces time to develop scripts.

The reference of **alias** can be used to set alternative providers or deploy resources in different regions:

*provider “aws” {*

*alias = “west”*

*region = “us-west-1”*

*}*

**Hashicorp Configuration Files**

Hashicorp configuration files (aka Terraform files) contain information about providers and resources.

End in .tf or .tf.json.

Terraform files are written in the Terraform Language using HCL (similar to JSON). HCL is an open-source toolkit for creating structured configuration languages. All Hashicorp applications are written using HCL. Apart from Sentinel.

Basic elements:

**Blocks**: containers for content. Block Type has a label and a **Body**. Block **Label** is the name of the block.

**Arguments**: assign a value to a name. Appear within blocks.

**Expressions**: represent a **value** either literally or by referencing and combining other values.

Terraform also supports an alternative syntax that is JSON-compatible. JSON syntax files are names **tf.json**

**Terraform Settings**

The terraform block type terraform { } is used to configure behaviours of terraform itself.

We can specify:

* **required\_version** (expected version of terraform)
* **required\_providers** (providers pulled during terraform init)
* **experiments** (experimental language features the community can try and provide feedback)
* **provider\_meta** (module-specific information for providers)

**Input Variables**

Parameters for terraform modules. Can be declared in root or child modules.

Defined with ‘variables’ block.

Options:

* **default** – default value if not set.
* **type** – value types accepted for the variable e.g., string, list(string), number, bool.
* **description** – variables description (optional)
* **validation** – block to define validation rules
* **sensitive** – limits terraform UI output when the variable is used in the configuration

**Variable Definition Files**

Allows you to set values for multiple variables at once. Named. tfvarsor **tfvars.json**

A file names **terraform.tfvars** is by default autoloaded when included in the root of your project directory. You can use other.tfvars file names but they are not autoloaded. You will have to use the command line to load them. Useful for dev and prod environments.

If you want files to autoload, you can modify the file to have **auto.tfvars** e.g. **dev.auto.tfvars**

To use variable files that are not configured to be autoloaded use syntax: **-var-file** dev.tfvars

To override a variable in a.vfvars file use syntax: **-var ec2\_type=”t2.medium”**

No variable blocks needed. Use identifies (names) and give them values:

*image\_id = “ami-12354”*

*availability\_zone\_names = [*

*“us-east-1a”,*

*“us-east-1b”,*

*]*

**Variables via Environment Variables**

Useful when using your own CI/CD process, terraform cloud or a build server to get environment variable into the build servers. Not used much locally.

Terraform looks for a variable starting with **TF\_VAR\_<NAME>** which is read and loaded. E.g.:

*export TF\_VAR\_image\_id=ami-abc123*

**terraform.tfvars** file **overrides** environment variables declared in other files. **terraform.tfvars.json** overrides **terraform.tfvars. auto.tfvars** overrides all of the above. -var and -var-file overrides all.

Precedence order:

environment variables

* **terraform.tfvars**
* **terraform.tfvars.json**
* **auto.tfvars**
* **-var and -var-file**

**Local Values**

Assigns a name to an expression to be used multiple times within a module without repeating it. Set using **locals { }** block.

*locals {*

*Service name = “forum”*

*Owner = “community Team”*

*}*

Can also be nested within each other e.g., **Owner = local.owner** and multiple local blocks can be defined.

To reference a local use **local.<name>** when referencing use the singular local not locals.

**Data Sources**

Information defined outside of Terraform such as by another terraform configuration or modified by functions.

*data “aws\_ami” “web” {*

*filter {*

*name = “state”*

*values = [“available”]*

*}*

Reference with: **data.aws\_ami.web.id**

**Named Values**

Built in expressions to reference 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 module where expression is placed
  + **path.root** – path of root module of the configuration
  + **path.cwd** – path of current working directory
  + **terraform**.**workspace** – name of currently selected workspace
* **Block-local vales –** within a body of blocks (things within a resource, provisioners etc)
  + count**.index** – to find iterations (when you use the count meta argument)
  + each.**key** / each.**value** – access key and values during interations (when you use the for\_each meta-argument)
  + **self**.<attribute> - references information within provisioners or connections

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

**Resource Meta Arguments**

Terraform language defines several **meta-arguments**, which can be used with any resource type to **change to behaviour 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 non-default provider configuration.
* **lifecycle,** for lifecycle customisations.
* **provisioner and connector,** for taking extra actions after resource creation.

**Depends\_on:**

Order in which resources are provisioned. When resources depend on others before they are provisioned. As in some cases terraform cannot determine the correct order.

**Count:**

When managing a pool of objects e.g., a fleet of virtual machines you can use **count.**

*resource “aws\_instance” “server” {*

*count = 4*

*ami – “ami-123”*

*instance\_type = “t2.micro”*

*tags = {*

*Name = ”Server ${count.index}”*

*}*

Value starts at “0” so server indexes will be 0, 1, 2, 3.

Count also takes **numeric expressions**:

*resource “aws\_instance” “server” {*

*count = length(var.subnet\_ids)*

**For\_each:**

For iterating over resource meta-arguments like count… but allows you to map over dynamic values (more flexible).

**Resource Behaviour**

When you perform execution via terraform apply, one of the following will be performed on a resource:

**Create** – resources exist in the configuration but not associated with real infrastructure. +

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

**Update in-place** – resources whose arguments have changed. ~

**Destroy and re-create** – resources who’s arguments have changed but which cannot be updated in-place due to remote API limitations. -/+

**Lifecycle:**

Block nested within resources that allow you to change what happens to a resource e.g., **create**, **update**, **destroy**.

*resource “azure\_respirce\_group” “example” {*

*lifecycle {*

*create\_before\_destroy = true*

*}*

*}*

* **Create\_before\_destroy** – when replacing a resource create the new before deleting the old.
* **Prevent\_destroy** – ensures a resource is not destroyed.
* **Ignore\_changes** – don’t change the resource (create, destroy, update) if a change occurs for the listed attributes.

**Resource Providers and Alias:**

The default provider can be overridden at a per resource level. To do this create an alternative provider with “**alias**”:

*provider “google” {*

*region = “us-central1”*

*}*

*provider “google” {*

*alias = “europe”*

*region = “europe-west1”*

*}*

*resource “google\_compute\_instance” “example” {*

*provider = google.europe*

*}*

Reference the alias under the attribute provider for a resource.

**Terraform Expressions**

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

**Types and Values –** result of an expression is a value. All values have a type:

**Primitive types:**

**String**: ami = **“ami-12345”**

**Number**: size = **6.2323**

**Bool**: termination\_protection = **true**

**No type:**

**Null:** endpoint = **null**

Null uses whatever the providers default value is.

**Complex/structural/collection types:**

**List (tuple):** regions = **[“us-east-1a”, “us-east-1b”]**

**Map (object):** tags = **{env = “Production”, priority =3}**

**STRINGS**

Always double quoutes: “”.

Can use escape sequences:

**\n** newline

**\r** carriage return

**\t** tab

**\”** literal quote (without terminating the string)

**\\** literal backslash

**\uNNNN** Unicode character from the basic multilingual plane

**\UNNNNNNNN** Unicode character from the supplementary planes

**special escape sequences:**

**$${**

**%%{**

Terraform also supports UNIX “heredoc” style multi-line strings:

*<<EOT*

*Hello*

*World*

*EOT*

(EOT can be anything.)

**STRINGS TEMPATES**

**String interpolation** evaluates an expression between the markers: ${…} and converts to a string:

*“Hello, ${var.name}!”*

Good for adding strings together.

**String directive** allows you to evaluate conditional logic between markers %{…}:

*“Hello, %{ if var.name != “” }!${var.name}%{ else }unnamed%{ endif }!”*

Stripe whitespace left by directive blocks using a trailing tilde: **~**

**CONDITIONAL EXPRESSIONS**

Terraform supports ternary if else conditions:

condition ? true\_val : false\_val

If variable a does not equal blank, then use the variable or set to default-a:

*var.a != “” ? var.a : “default-a”*

Return type for if and else must be the same Type (string or number etc.).

**FOR EXPRESSIONS**

Allows to iterate over complex types. Accepts input as a list, set, tuple, map or object.

Example to uppercase each string in the provided list.

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

Square braces [] return a tuple:

[“HELLO”, “WORLD”]

Curly braces {} returns an object.

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

{hello = “HELLO”, world = “WORLD”}

**Splat operator** represented by an asterisk (\*) is used to roll up a bunch of iterations in a ***for*** expression:

*var.list[\*].id*

**Dynamic Blocks**

Dynamically construct repeatable nested blocks.

Example use when creating numerous ingress rules for a security group:

*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”]*

*}*

*}*

*}*

**Terraform State**

Condition of cloud resources at a specific time. When you provision infrastructure a state file names **terraform.tfstate** is created. The file is a **JSON** data structure with a one-to-one mapping from resource instances to remote objects.

**Terraform state commands:**

1. terraform state **list –** list resources in the state
2. terraform state **mv –** move an item in the state
3. terraform state **pull –** pull current remote state and output stdout
4. terraform state **push –** update remote state from a local state
5. terraform state **replace-provider –** replace provider in a state
6. terraform state **rm –** remove instances from the state
7. terraform state **show –** show a resource in the state

**Terraform state mv**

Allows you to:

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

To rename a resource or move it to another module with a terraform apply Terraform will destroy and create the resource then create it. **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***

All terraform state subcommands that modify state will write a **backup** file called **terraform.tfstate.backup**. Cannot be disabled

Read only commands do not modify state e.g., list, show.

**Terraform Init**

Terraform init initializes your terraform project by:

* Downloading plugin dependencies e.g., Providers and Modules
* Creating a **.terraform** directory (hidden)
* Create a dependency lock file to enforce expected versions for plugins and terraform itself **(.terraform.lock.hcl)**
* Dependencies end up in the **terraform-providers-aws.v4.49.0exe** file.

Dependency lock file: ***.terraform.lock.hcl***

State lock file: ***.terraform.tfstate.lock.hcl***

Run terraform init each time you need to modify or change dependencies to apply the changes.

Flags:

* terraform init **-upgrade** – upgrade plugins to latest version that complies with the configurations version constraint.
* terraform init **-get-plugins=false** – Skip plugin installation.
* terraform init **-plugin-dir=PATH** – Force plugin installation to read plugins from target directory.
* terraform init **-lockfile=MODE** – Change lockfile mode

**Terraform commands:**

**terraform get** is used to download and update modules in the root module. Alternative to terraform init for use with local module development.

**terraform fmt** looks in the current directory and applies formatting for readability to all .tf files.

**terraform validate** runs checks to verify if configurations have valid syntax. **terraform apply** also runs terraform validate and terraform plan.

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

**terraform plan -out=FILE** extracts the terraform plan to a saved file.

**terraform apply FILE** can then be passed to provision resources. This method does acts like an -auto-approve.

**Resource Drift:**

When your resources are in a different state than your expected state.

Resolved by:

* Replacing resources with **-replace** flag
* Importing approved manual resources to the state file with **import** command

*terraform import aws\_instance.example i-abcd241*

Not all resources can be imported.

* Refresh state with -**refresh-only**, when an approved manual resource has been changed or removed and needs to be reflected in state file.

**Terraform errors:**

* **Language errors –** syntax errors (resolved with fmt, validate, terraform version etc)
* **State errors –** resource state is changed from expected state in config file (resolve with terraform -refresh-only, apply and -replace flag)
* **Core errors –** bug in the core library (check terraform logs/ open github issue)
* **Provider errors –** providers API has changed or not working as expected (check terraform logs/ open github issue).

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

* TRACE
* DEBUG
* ERROR
* WARN
* INFO
* JSON – outputs logs at the TRACE level or higher and uses **JSON** encoding and formatting.

**PowerShell**

> $env:TF\_LOG="TRACE"

> $env:TF\_LOG\_PATH="terraform.txt"

**Bash**

$ export TF\_LOG="TRACE"

$ export TF\_LOG\_PATH="terraform.txt"

Logging can be enabled separately:

* TF\_LOG\_CORE
* TF\_LOG\_PROVIDER

Choose where to log with **TF\_LOG\_PATH**

**Crash Log:**

If terraform ever crashes it saves a log file with the debug logs from the session as well as the panic message and backtrace to **crash.log.**

**Terraform Modules**

Can be found publicly in the Terraform Registry. Browse based on provider. Only verified and official modules are displayed in the search.

**Public Modules:** in terraform registry (which is integrated directly into terraform). Takes module name, source, and version.

*module “consul” {*

*source = “hasicorp/consul/aws”*

*version = “0.1.0”*

*}*

**terraform init** downloads and caches any modules referenced by a configuration.

**Private Modules:** in terraform cloud or enterprise. Need to authenticate against terraform cloud via **terraform login**. Or create a user API token and manually configure credentials in the CLI config file.

*module “vpc” {*

*source = “app.terraform.io/example\_corp/vpc/aws”*

*version = “0.9.3”*

*}*

**Publishing modules** supports:

Versioning, automatically generating documentation, allowing users to browse version histories, show examples, and include READMEs.

All modules are hosted on **GitHub**. Naming convention must be terraform-<PROVIDER>-<NAME> i.e., **terraform-aws-vpc.** Connect and publish your module via your GitHub account.

Verified Modules are reviewed by HashiCorp and actively maintained by contributors. Verified badge appears next to module. Unverified modules should not be considered bad quality. Unverified means it hasn’t been created by a HashiCorp partner.

**Standard Module Structure**

A file and directory layout recommended for module development. Has a **root module** and **nested module.**

**Root module** is primary entry point. Requires main.tf, variables.tf, outputs.tf, README and LICENSE.

**Nested modules** are optional and must be contained in the “modules/” directory. A sub module that contains a README is considered usable by external users. Without a README it is considered external use only.

Whenever you create a terraform script you are technically creating modules even if you don’t intent to publish them.

**WORKFLOWS**

**Write > Plan > Apply.**

**Individual Practitioner:**

**Write** –

* person using editor of choice locally.
* Stores terraform code in a VCS e.g., GitHub.
* Run terraform plan and validate to find syntax errors.
* Benefit is tight feedback loop between editing code and running test commands as it is all local, no build servers etc.

**Plan** –

* when developer is confident with work, commit the code to their local repo.
* May be using a single branch i.e., main. Once commit is written proceed to apply.

**Apply** –

* run apply and be prompted to review the plan.
* After final review approve changes and await provisioning.
* After successfully provisioning, push local commits to remote repo.

**Team (not terraform cloud. Using old fashioned way):**

**Write** –

* each member using editor of choice locally.
* Stores terraform code in a BRANCH in their code repo (branches help avoid conflicts while another member is working on their code, also allows an opportunity to resolve any conflicts during a merge to main).
* Run terraform plan as quick feedback for SMALL teams.
* For LARGER teams, sensitive credentials become a concern:
  + A CI/CD process may be implemented so that the CI/CD process is in charge of the credentials. You don’t run plan just push to your branch and the CI/CD will run it.

**Plan** –

* when a branch is ready to be incorporated on a Pull Request, an Execution Plan (speculative) can be generated and displayed within the Pull Request for review.

**Apply** –

* to apply the changes, the merge must be approved and merged. Which kicks off a code build server that will run terraform apply.

**Downsides to team workflow:**

* DevOps team must set up and maintain their own CI/CD pipeline.
* They have to figure out how to store the statefile
* Limited on access controls (cannot be granular over who can run apply, destroy etc.)
* Have to figure out a way to safely store and inject secrets to the build server’s runtime environment.
* Managing multiple environments can make the overhead of the infrastructure increase dramatically.

**Terraform Cloud**

**Write** –

* Team uses Terraform Cloud as remote backend. Editor of choice.
* Input variables stored on Terraform Cloud instead of locally.
* Integrates with git your VCS to set up a CI/CD pipeline quickly
* Team member writes code and commits to a branch as usual.

**Plan** –

* A pull request is created by a team member and terraform cloud generates a speculative plan for review in the VCS. The member can also review and comment on the plan in terraform cloud.

**Apply** –

* After the pull request is merged terraform cloud runtime performs a terraform apply. A team member can confirm and apply the changes.

**Benefits to Terraform Cloud workflow:**

* Streamlines a lot of the CI/CD effort.
* Stores sensitive credentials.
* Makes it easier to go back and audit the history of multiple runs.

**Backends**

Each terraform config can specify a backend which defines where and how operations are performed and where state snapshots are stored.

Divided into two types:

**Standard backends** – only store state. Does not perform terraform operations such as terraform apply. To apply operations, use the CLI on your local machine. They are third party backends are standard backends e.g., AWS S3.

Standard backends: S3, Azure blob storage, GCP object storage, Alibaba cloud storage, HashiCorp Consul.

**Enhanced backends** – can store the state and perform terraform operations.

Enhanced backends are subdivided further:

* Local – files are stored on local machine executing terraform commands.
* Remote – files are stored in cloud e.g., terraform cloud.

**Local** backends:

State stored on local filesystem. Locks that state using system APIs. Performs operations locally.

You use local by default when you specify no backend in the terraform block.

**Remote** backends:

Uses terraform platform which is either terraform cloud or terraform enterprise.

When terraform apply is performed the terraform cloud run environment is responsible for executing the operation.

**Environment variables** must be configured in terraform cloud with provider credentials i.e., Access Key ID and Secret Access Key.

Also need to create a terraform cloud **Workspace** and define it in the terraform block. Organization must also be in terraform block and backend must be declared as remote.

If two workspaces are in the terraform block using **“prefix =”** then on terraform apply you will need to choose which workspace to apply the operation.

**BACKEND INITILAZATION**

You can use the **-backend-config** flag with **terraform init** for partial backend configuration. Useful in situations where the backend settings are dynamic or sensitive and so cannot be in the configuration file:

**backend.hcl:**

*workspaces {name = “workspace”}*

*hostname = “app.terraform.io”*

*organization = “company”*

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

**TERRAFORM\_REMOTE\_STATE**

Data source that retrieves the root module output values from another terraform configuration using the latest state snapshot from the remote backend. So, reference a statefile from somewhere else.

Reference the other backend using **data** block.Can be done for either remote or local backends.

Only exposes output values. User must have access to entire state snapshot, which often includes some sensitive information.

**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 wont see any messages that it is happening unless it is taking longer than usual.

**-lock** flag disables lock (not recommended).

**force-unlock** command manually unlocks the state if unlocking failed. Requires a unique lock ID:

**terraform force-unlock 12512523525235235 – force.**

**Local state file:**

* Stores state in plain text JSON
* Be careful not to share the file
* Be careful not to commit the file to git repo

**Remote state file:**

* Held in memory not persisted to disk.
* Encrypted at rest and transit.
* Audit logging for tamper evidence (terraform enterprise)
* Can store state with various third party backends such as S3.

**TERRAFORM IGNORE FILE**

When executing a remote plan or apply with CLI, an archive of the configuration directory is uploaded to terraform cloud. You can define paths to ignore form upload via a **.terraformignore** file at the root of the config directory.

Cannot have multiple of these files and must be in root directory. If file is not present the following is excluded by default:

**.git** and **.terraform**.

**Resources**

In terraform config files represent infrastructure objects e.g., VM’s, databases, Virtual networks etc…

Follows format of:

**resource > “resource type > “instance name”**

***resource > “aws\_instance” > “web\_instance”***

Some resources have timeout nested blocks that lets you choose how long certain operations are allowed to take place for before being considered to have failed:

*resource “aws\_db\_instance” “example” {*

*# …*

***timeouts {***

***create = “60m”***

***delete = “2h”***

*}*

*}*

**COMPLEX TYPES**

A type that groups multiple values into a single value. Represented by type constructors but several also have shorthand keyword versions.

Two types of complex types:

* **Collection types** (for grouping similar values):
  + List, Map and Set
* **Structural types** (potentially dissimilar values):
  + Tuple and Object

Values within a collection are called element types.

* **List** types are referenced using the index i.e., [0].
* **Map** types use a Key and Value. Select the value by referencing the key name.
  + **plan = var.plans[“Plan-A”]**
* **Set** types are like lists but have no secondary index or preserved ordering. All values must be of the same type.
  + **toset [(“a”, “b”, 3)]**
  + **[**

**“a”,**

**“b”,**

**“3”,**

**]**

Structural types require a schema as an argument to specify which types are allows for which elements:

*variable “with\_optional\_attribute” {*

*type = object({*

*a = string # a required resource*

*b = optional(string) # an optional resource*

*})*

**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 has a number of built in functions you can call from within expressions to transform and combine values:

1. Numeric Functions
2. String Functions
3. Collection Functions
4. Encoding Functions
5. Filesystem Functions
6. Date and Time functions
7. Hash and Crypto Functions
8. IP Network Functions
9. Type Conversion Functions

**NUMERIC FUNCTIONS**

1. **abs** returns the absolute value of the given number (negative flips to positive):

* abs(23)

23

* abs(0)

0

* abs(-12.4)

12.4

1. **Floor** returns the closes whole number LESS than or equal to the given value:

* floor(5)

5

* floor(4.9)

4

1. **Log** returns the logarithm of a given number in a given base:

* log(16, 2)

4

1. **Ceil** always rounds UP to the closest whole number:

* ceil(5. 2)

5

* ceil(5)

6

1. **Min** takes one or more numbers and returns the smallest number from the set.
2. **Max** takes one or more numbers and returns the largest number from the set.
3. **Parseint** takes a given string as representation of an integer in the specified base and returns the resulting integer ( “value”, “BASE-Encoding”):

* parseint(“100”, “10”)

10

* parseint(“FF”, “16”)

255

1. **Pow** calculates an exponent by raising the first argument to the power of the second argument:

* pow(3, 2)

9

* pow(4, 0)

1

1. **Signum** determines the sign of a number, returning a number of -1, 1, or 0 to represent the sign:

* signum(-13)

-1

* signum(0)

0

* signum(133)

1

**STRING FUNCTIONS**

1. **Chomp** – removes newline (\n) characters at the end of a string.
2. **Format** – produces a string by formatting a number of other values.
3. **Formatlist** – produces a list of strings by formatting a number of other values according to a specification string.
4. **Indent** – adds a given number of spaces to the beginnings of all but the first line in a given multi-line string.
5. **Join** – makes a string of all elements of a given list of strings with the given delimiter.
6. **Lower** – converts string to lowercase.
7. **Regex** – applies regular expression to a string and returns matching substrings.
8. **Regexall** – applies a regular expression to a string and returns a list of all matches.

**Terraform Cloud**

3 types of workflows:

**UI/VCS (user interface and version control system)**

* Terraform cloud integrated with for VCS i.e., GitHub via webhooks.
* When pull requests are submitted for the branch speculative plans are generated.
* When a merge occurs to the branch, then a run is triggered in terraform cloud.

**API-Driven (Application Programming Interface)**

* Workspaces not directly associated with a VCS repo.
* Third-party tool or system triggers runs via an upload configuration file via the Terraform Cloud API
* This config file is a bash script that is packaged in an archive (.tar.gz). And you push configuration versions each time.

**CLI-Driven**

* Runs triggered by the user running terraform CLI commands e.g., terraform plan and apply on their machine.

**TERRAFORM CLOUD ORGANIZATION-LEVEL PERMISSIONS**

Manage resources or settings across an organisation.

* **Manage policies** – create, edit, and delete the organisations sentinel policies.
* **Manage policy overrides** – override soft-mandatory policy checks.
* **Manage workspaces** – create and administrate all workspaces within the organisation.
* **Manage VCS settings** – set VCS providers and SSH keys available within the organisation.

**For the Organization Owners ONLY**

Every organisation has an owner or owners who have a special role that contains every permission and some actions only available to the owners:

* Publish private modules.
* Invite users to organisation.
* Manage team membership.
* View all secret teams.
* Manage organisation permissions.
* Manage all organisation settings.
* Manage organisation billing.
* Delete organisation.
* Manage agent.

**API TOKENS**

Terraform supports three types.

**Organisation API Tokens:**

* Have permissions across entire organisation.
* Each organisation can have one valid API token at a time.
* Only organisation owners can generate or revoke an organisations token.
* Organisation API tokens are designed for creating and configuring workspaces and teams.
  + not recommended for an all-purpose interface to terraform cloud (useful when you want to set up your organisation for the first time and do it programmatically).

**Team API Tokens:**

* Allow access to the workspaces that the team has access to, not tied to any specific user.
* Each time can have one valid API token at a time.
* Any member of a team can generate or revoke that teams token.
* When a token is regenerated, the previous token becomes invalid.
* Designed for performing API operations on workspaces.
* Same access level the team has access to.
* To use a team token you must manually set it in your terraform CLI config file.

**User API Tokens:**

* Most flexible token as they inherit permissions from the users they are associated with.
* Could be a real user or machine user.
  + When you do a ***terraform login*** you get a user API token

User and Team API tokens are used to authenticate to publish private modules for your organisation to the **Terraform Cloud Private Registry**.

**COST ESTIMATION** is a feature to get a monthly cost of resources displayed alongside your runs. Only available with the Teams and Governance plan and above.

Only for specific cloud resources within AWS, Azure and GCP. Shows hourly and monthly costs for each services and for services combined.

A Sentinel Policy can be used to keep resources under a particular cost.

**TERRAFORM CLOUD RUN ENVIRONMENT**

When terraform cloud executes your plan it runs them in its OWN run environment.

A run environment is a VM or container intended for the execution of code for specific runtime environment. Essentially a code build server. Terraform cloud run environment is a single-**use Linux machine**.

The following environment variables are injected on each run:

* **TFC\_RUN\_ID** – a unique identifier for the run (e.g., run-2241241)
* **TFC\_WORKSPACE\_NAME** – name of workspaces used for run
* **TFC\_CONFIGURATION\_VERSION\_GIT\_BRANCH** – full slug of the configuration used in the run in org/workspace format e.g., alexorganisation/firstworkspace.
* **TFC\_CONFIGURATION\_VERSION\_GIT\_BRANCH** – name of branch used i.e., main
* **TFC\_CONFIGURATION\_VERSION\_GIT\_COMMIT\_SHA** – full commit has of commit used

**TFC\_CONFIGURATION\_VERSION\_GIT\_TAG** – name of git tag used e.g., 1.1.0

To access these variables in the code define the variable and its name to access throughout the code:

***variable “TFC-RUN-ED” { }***

**TERRAFORM CLOUD AGENTS**

Feature of the Business plan to allow terraform cloud to communicate with isolated, private, or on-prem infrastructure.

Useful for on premise types such as vSphere, Nutanix or OpenStack.

No inbound connectivity required all pull-based. Any agent provisioned polls terraform cloud for work and carries out execution of that work locally.

* Agents only support x86 or 64bit Linux OS’s
* Can also run in Docker using official Terraform Agent Docker container.
* From version 0.12 and up
* System requires at least 4GB of free disk space and 2GB memory
* Needs access to make outbound requests on HTTPS (TCP port 443) to:
  + App.terraform.io
  + Registry.terraform.io
  + Releases.hasicorp.com
  + Achivist.terraform.io

**Terraform Enterprise**

Self-hosted distribution of the Terraform platform.

Offers a private instance of the terraform platform application with following benefits:

* No resource limits.
* Additional enterprise-grade features:
  + Audit logging
* SAML (SSO)

Diagram

Description automatically generated

**REQUIREMENTS**

**Operational mode** – how data should be stored:

* External services
  + Postgress
  + S3, GCP cloud storage bucket, Blob storage
* Mounted disk
  + EBS, iSCSI
* Demo
  + All data stored on the instance; data can be backed up with snapshots

**Credentials** – ensure you have credentials to use Enterprise and have a secure connection.

* Terraform Enterprise License – obtained by HashiCorp
* TLS Certificate and Private Key – to prove you own your own TLS certificate.

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

Supported OS:

* Ubuntu
* Debian
* RedHat
* CentOS
* Amazon Linux

**Hardware Requirements**

* 10GB disk space
* 40GB disk space for Docker data directory
* At least 8GB RAM
* 4 CPU cores

Terraform enterprise also supports Air Gapped environments (disconnected networks isolated from unsecure networks such as the public internet).

Table

Description automatically generated

**WORKSPACES**

Allow you to manage multiple environments or alternate state files e.g., dev, prod.

Two variants:

**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 like different branches in a git repo.

There is already a single workspace in your local backend called **default**. It cannot be deleted.

Depending on if you have a local or remote backend there are changes how the state file is stored:

* **Local state:**
  + Terraform stores states in a folder called **terraform.tfstate.d**
  + Small teams or individuals are known to commit to repos
* **Remote state:**
  + Workspace files are stored directly in the configured backend.
  + Recommended when there are multiple collaborators.

**RUN TRIGGERS** on terraform cloud allow you to schedule workspace runs that rely on information or infrastructure produced from other workspaces. Connect up to 20 workspaces.

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

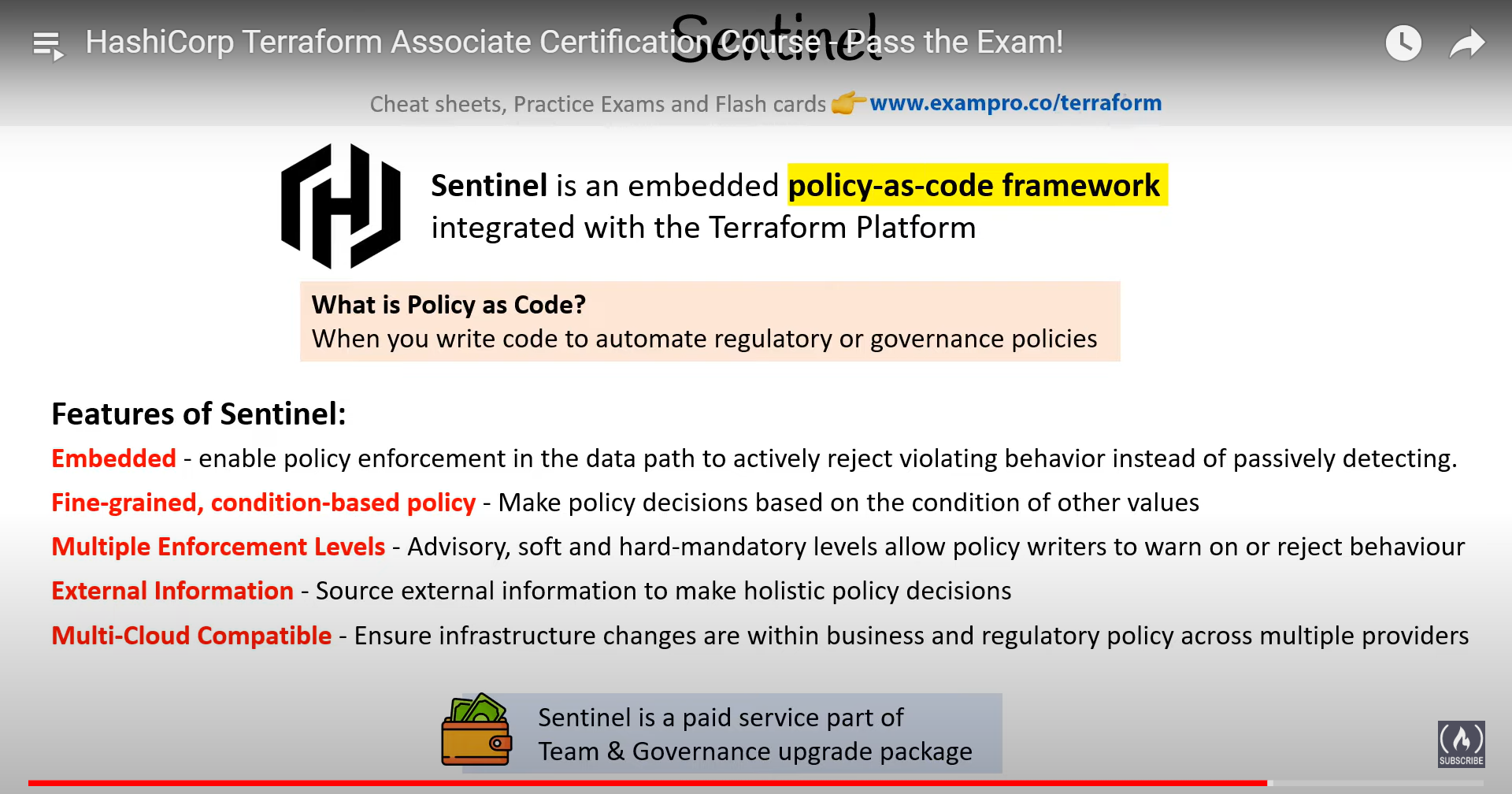
**Sentinel**

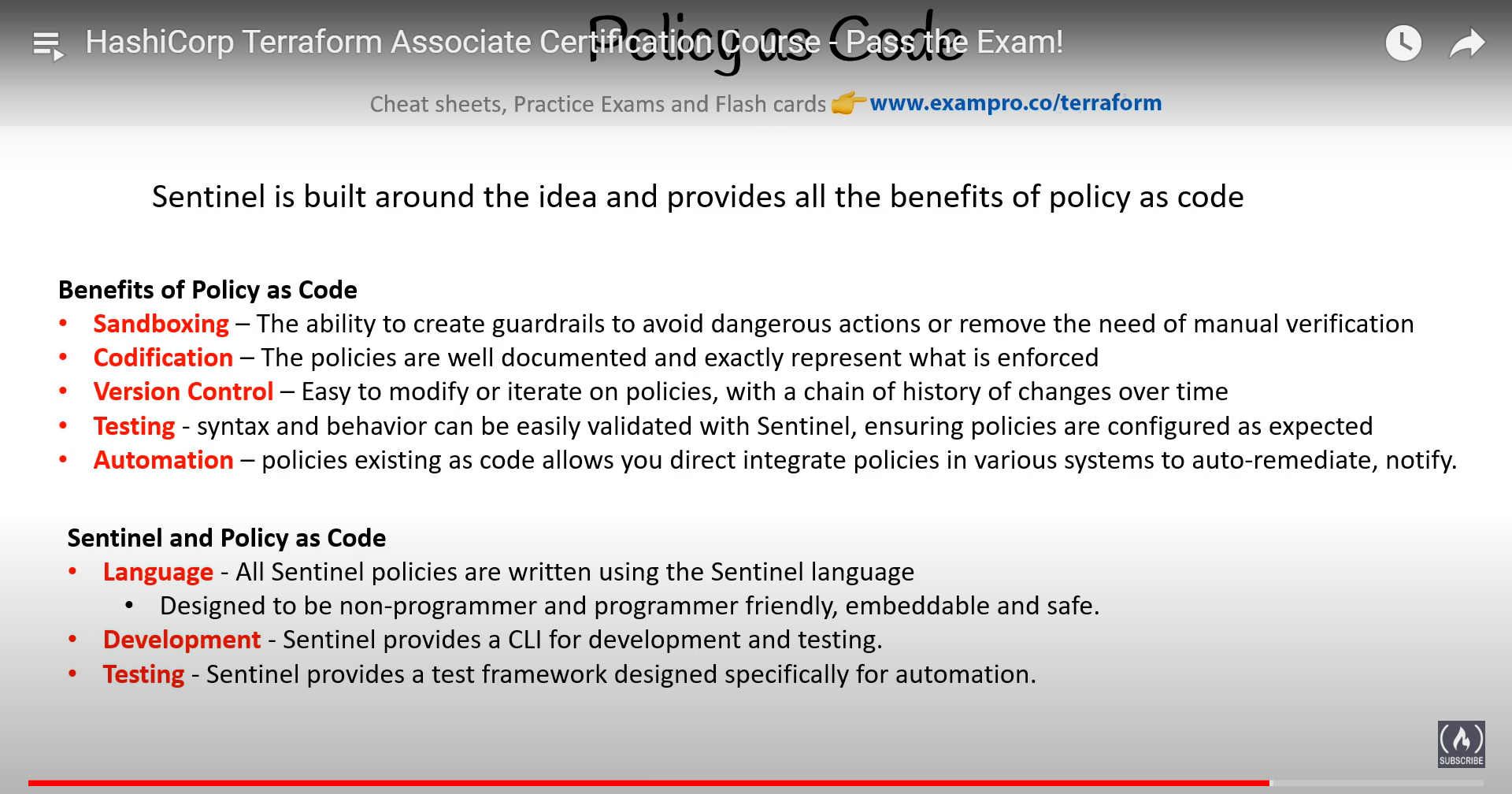
Embedded **policy-as-code** framework integrated with Terraform via Terraform Cloud as part of your IaC provisioning pipeline between plan and apply.

A picture containing text, clock, gauge

Description automatically generated

**Policy as code** – when you write code to automate regulatory or governance policies.





**Examples of policies you can write and enforce depending on provider:**

AWS:

Restrict instance types of EC2s.

Disallow 0.0.0.0/0 CIDR Block in security groups.

Restrict availability zones for instances.

Azure:

Enforce mandatory tags of VMs.

Restrict the size on VMs.

Enforce limits on AKS clusters.

GCP:

Restrict machine type VMs

Disallow 0.0.0.0/0 CIDR block in network firewalls.

VMWare:

Restrict size and type of virtual disks

Restrict memory and CPU count of VMs.

Cloud-Agnostic:

Allowed providers

Prohibited providers

Limit proposed monthly costs etc…

**HashiCorp 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.
* VM’s if fleet are all on-to-one config.
* Faster deploys for multiple servers after each build.
* Earlier detection and intervention of package changes or depreciation of old technology.

**Commit** code to CI/CD pipeline >

Start a **build server** running **Packer** which will generate a **build image** >

Use something to **provision** i.e., **Ansible** >

Packer then **stores** the image somewhere such as am **AMI** on AWS >

Reference image in terraform code >

**Provision** the infrastructure (**instances**).