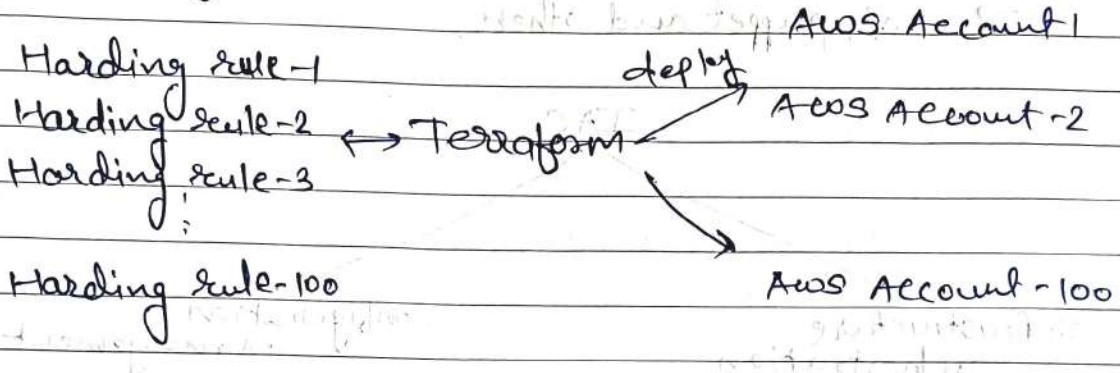
Terraform

Terraform

→ Terraform allows you to create reusable code that can deploy identical set of infrastructure in repeatable fashion.

HCL config



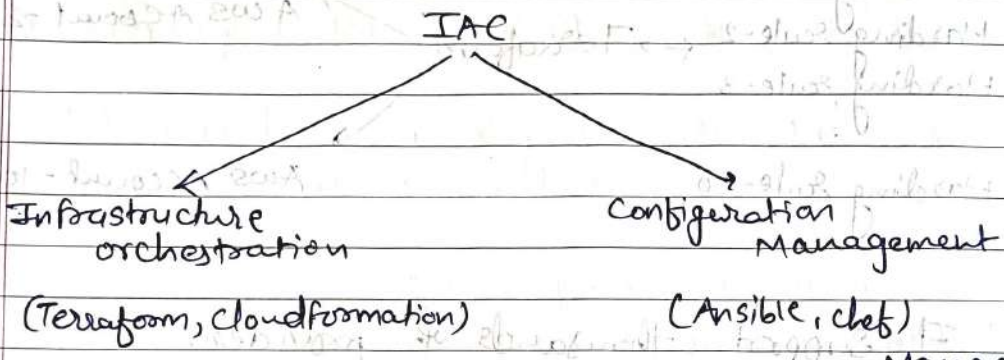
→ It support thousands of providers.

→ IAC (Infrastructure As Code)

IAC is the mangging and provisioning of infrastructure through code instead of through manual process.

→ Benefits
- → Speed of Infrastructure management
- → Low risk of Human Errors.
- → Version control
- → Easy collobration between teams.

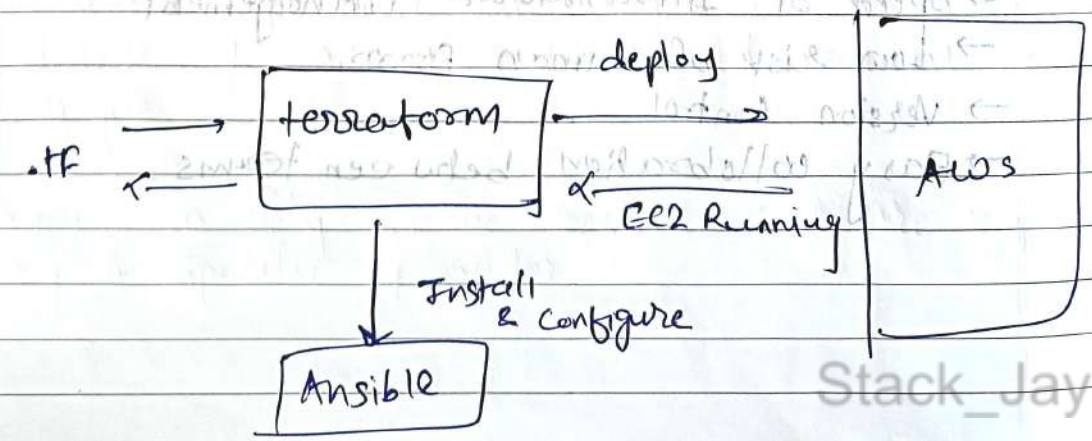
→ There are various type of tools for deploy infrastructure as code:
- → Terraform
- → Cloud Formation
- → Heat
- → Ansible
- → saltstack
- → chef, puppet and others.



→ primary aim to create and manage infrastructure environment.

→ Configuration Management tool tools are primarily used to maintain configuration of systems. (inside servers)
(eg:- install AV in servers)

→ Ex:- Create 3 servers with 4 GB RAM.



→ How to choose IAC tools.

(i) Is your infrastructure going to be vender specific in longer term? Ex:- AWS.

(ii) Are you planning to have multi-cloud / hybrid cloud based infrastructure?

(iii) How well does it integrate with configuration management?

(iv) price & support.

⇒ First tf. Launch virtual machine using terraform

```
Provider = "aws" {
    region = "us-east-1"
    access_key = " "
    secret_key = " "
}

resource "aws_instance" "myec2" {
    ami = "ami-_ _ _"
    instance_type = "t2.micro"
}
```

→ terraform init → download appropriate plugin of define provider

→ terraform plan → define what we are creating.

→ terraform apply → it create a resource

→

⇒ Learning-1 provider plugins

→ A provider is a pluging that lets Terraform manage an external API

→ when we run terraform init, plugins required for the provider are automatically downloaded and saved locally to a .terraform directory.

⇒ Learning -2 Resources

→ Resource block describes one or more infrastructure objects.

→ Resource type and Name together serve as an identifier for a given resource and so must be unique.

→ You can only use the resource that are supported by a specific provider.

→ The core concept, and standard syntax remain similar across all provider.

→ There are 3 primary types of provider tiers in Terraform
1. Official
2. partner
3. Community

→ Important learning

Terraform required explicit source information for any providers that are not Hashicorp-maintained, using a new syntax in the required_providers nested block inside the terraform configuration block.

⇒ Terraform destroy

→ Terraform destroy allows to destroy all the resources that are created within the folder.

→ terraform destroy with -target flag allows us to destroy specific resources
eg:- terraform destroy -target aws_instance.myec2

→ The -target option can be used to focus Terraform attension on only a subset of resources

Combination of: Resource Type + Local Resource Name.

Resoure type	Local Resource name
aws_instance	myec2
github_repository	example

→ Also by commeting the code (resource) we can destroy the resources.

* Terraform State files

→ Terraform stores the state of the Infrastructure that is being created from the TF files.

→ This state allows terraform to map real-world resource to your existing configuration.

→ State file store all the infrastructure related data.

* Desired state vs current state

→ Desired state:-

Terraform primary function is to create, modify, and destroy infrastructure resources to match the desire desired state described in a terraform configeration.

→ Current state:

→ current state is the actual state of the resource that is currently deployed.

→ not always current state and desired state are same.

→ Important

→ Terraform tries to ensure that the deployed infrastructure is based on the desired state.

→ If there is a difference between the two, terraform plan present describe description of the changes necessary to achieve the desired state.

* Terraform provider versioning

→ provider Architecture

do.droplet.tf → terraform → Digital ocean → New server

→ provider plugins are release seperately from terraform itself.

→ They have different set of versions numbe

⇒ Explicitly Setting provider version

During terraform init, if version argument is not specified, the most recent provider will be downloaded during initilization.

→ for production use, you should ~~contrain~~ constrain the acceptable provider versioning via configuration to ensure that new version with breaking changes will not be automatically installed.

```
[version = "~> 3.0"]
```

* Argument for Specifying provider

→ There are multiple ways

version number argument	description	means
>=1.0	Greater than equal to the version	means version should be greater or equal to 1.
<=1.0	lesser than equal to the version	means version shoulld be lesser or equal to 1.
~> 2.0	Any version in the 2.x range	means 2.0, 2.2, 2.22....
>=2.10, <=2.30	Any version between 2.10 to 2.30	

* Dependency lock file

→ Terraform dependency lock file allows us to lock to a specific version of the provider.

→ If a particular provider already had a selection recorded in the lock file, Terraform will always re-select that version for installation, even if a new version has become available.

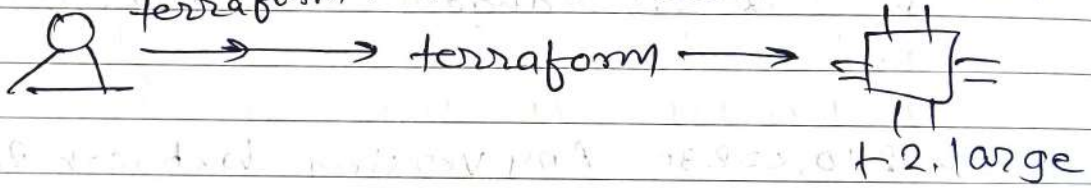
→ you can override that behaviour by adding the -upgrade option when you run ~~terr~~ terraform init.

Terraform Refresh

→ Terraform create any infrastructure based on configuration you specified.

→ It can happen that the infrastructure get modified manually.

→ The terraform refresh command will check the latest state of your infrastructure and update the state file accordingly



Chang in state file.

→ You should not typically ~~not~~ need to call this command all the times.

→ because it automatically perform as a part of creating a plan or apply command.

→ The terraform refresh command is deprecated in newer version of terraform.

→ The -refresh-only option for terraform plan and terraform apply was introduced in Terraform v0.15.4.

* AWS provider - Authentication Configuration

→ we are manually hardcoding the access / secret keys within the provider block.

→ Although a working solution, but it is not optimal from security point of view.

→ So we need to run code without hard coding the secret in the provider book.

→ Better Approach :-

→ AWS provider can source credentials and other settings from the shared configuration and credential files.

→ but the issue is that if 10 people are working and saved credential at different path. so it got error.

→ If the shared lines are not added to the provided block, by default, Terraform will locate this files at $HOME/.aws/config and $HOME/.aws/credentials on linux.

→ but in windows
%USERPROFILE%\.aws\config
%USERPROFILE%\.aws\credential.

→ AWS CLI allow customers to manage AWS resources directly from CLI.

→ when you configure Access/Secret key in AWS CLI, the location in which credential are stored are same as terraform search.

* AWS Firewall

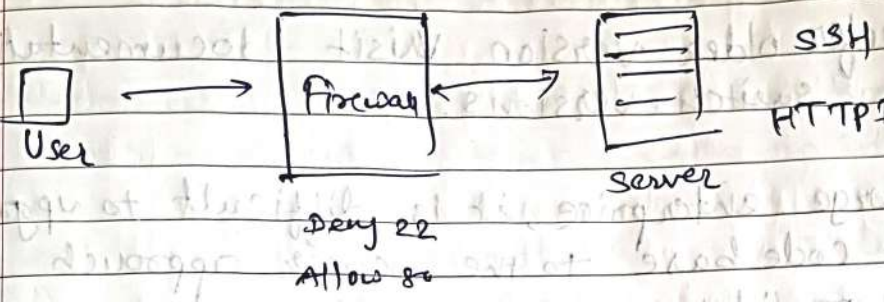
→ ports :-

→ A port act as an endpoint of communication to identify a given application or process on a linux operating system.

→ Firewall :-

→ Firewall is a network security system that monitors and controls incoming and outgoing network traffic based on predetermined

security rules.



→ In AWS security group work as a firewall.

→ In security groups, in terraform
inbound rule is known as ingress_rule
outbound rule is known as egress_rule

→ -auto-approve → skips interactive approval of plan before applying. This option is ignored when you pass previously-saved plan file. because Terraform consider you passing the plan file as the approval and so will never prompt in that case.

→ Imp :-

Just because a better approach is recommended, does not always mean that the older approach will stop working.

→ Organization can continue to use the approach that suits best in its environment.

→ to refer older version visit documentation and switch versions.

→ for large enterprice, it is difficult to upgrade their codebase to the newer approach that provider recommends.
→ So they can stick to older version of provider.

* Elastic IP in AWS

→ An Elastic Ip Address is a static IPv4 address in AWS
→ You can create it and associate it with EC2 instance.

* Attributes

→ Each resource has its associated set of attributes.

→ Attributes are the field in a resource that holds the value that ends up in state.

eg:- EC2 → Attribute — value
IP — ...
Public IP — ...

→ All the attributes are stored in state file.

→ It can happen that in a sigle terraform file, you can defining two different resources.

→ However Resource 2 might be dependent on some value of Resource 1.

→ we have to find a way in which attribute value of "public_IP" is reference to the cidr_IPV4 block of security group rule resources

⇒ Cross Referencing Resource Attribute

→ Terraform allows us to reference the attribute of one resource to be used in a different resource.

Overall

→ overall syntax:-

<ResourceType>.<Name>.<Attribute>

eg:- "aws_eip.lb.public_ip"

⇒ String Interpolation in terraform

→ ${} → this syntax indicates that Terraform will replace the expression inside the curly braces with its calculated value

eg:- "${aws_eip.lb.public_ip}/32"

⇒ Output Values

→ Output values make information about your infrastructure available on the command line, and can expose information for other Terraform configuration to use.

→ output = "public_ip" {
    value = aws_eip.lb.public_ip
}

→ output values defined in project A can be referenced from code in project B as well.

→ when we need to set multiple values in reference attribute, it become complicated so we make centerilized it and used it all other places.

* Variables

→ Terraform input variables are used to pass certain values from outside of the configuration.

→ Benefits :-
① Update implement important values in one central place instead of searching and replacing them throughout your code, saving time and potential mistakes.

② No need to touch the core Terraform configuration file. This can avoid human mistakes while editing.

→ To use we need to define in central file and use var.vpn_id

* TFVARS

→ Managing variables in production environment is one of the very imp aspect to keep code clean and reusable.

→ Hashicorp recommended creating a seperate file with name of *.tfvars to define all Variable Value in a project.

→ Structure look like in production

① Main Terraform configuration file.
② Variables.tf file that defines all the variables
③ terraform.tfvars file that defines value to all the variables.

→ in company there are multiple stages like dev, stag, prod. So we can create diff files of .tfvars for diff stages.

→ we can also manually call the tfvars file by -var-file="prod.tfvars"

→ you can also declare variable value in CLI.

→ we can define variables in various ways!-

1. Variable defaults
2. Variable Definition file (*.tfvars)
3. Environment variable
4. Setting variable in the command line.

→ Environment Variable :-
↳ Terraform searches the environment of its own process for environment variables named TF_VAR_ followed by the name of declared variable.

* Variable Definition Precedures

Terraform loads variable in the following order,

1. Environment variables
2. The terraform.tfvars file, if present
3. The terraform.tfvars.json file, if present
4. Any *.tfvars or *.auto.tfvars.json files, processed in lexical order of their filesname
5. Any -var and -var-file options on the command line.

eg :- Env variable < Value in terraform.tfvars < terraform plan -var = "instance_type = ms.1"

* Data types in Terraform

→ Data type refer to the type of value.

→ Depending on the requirement, you can use wide variety of values in Terraform configuration.

eg:- "Hello world" → string (data type)
7676 → Number

→ we can restrict value of a variable to a data type.

Ex:- only number should be allowed in AWS Usernames.

data types → string, number, bool, list, set, map, null.

⇒ List:-

→ Allow us to store collection of values for a single variable / arguments.

→ Represent by a pair in a square brackets.

→ useful when multiple values need to be added for a specific arguments.

→ Map :-

→ A map datatype represent a collection of key value pair elements.

* The count Meta Argument

→ The count argument accepts a whole number, and creates that many instances of the resources.

eg:- Count = 3

→ The instance created through count and identical copies, but you might want to customize certain properties for each one

⇒ Count Index

→ When using count, you can also make use of count.index which allow better flexibility.

→ tags = {
Name = "payments-system-${count.index}"

→ This attribute holds a distinct index number, starting from 0, that uniquely identifies each instance created by the count meta-argument.

→ we can also use count.index to iterate through the list to have more customization.

→ we can also give name in list.

* Conditional Expression

Conditional expression in terraform allow you to choose between two values based on a condition.

→ Syntax:-

```
condition ? true_val : false_val
```

If true then true value otherwise false value.

* Terraform Functions

→ A function is a block of code that perform specific task.
→ max(), file()

⇒ File function

→ File function can be reduced the overall terraform code size by loading contents from external source during terraform operations.

→ The terraform language does not support user-defined function, and so only the functions built in to the language are available for us.

* Local Values

→ Local values are similar to variable in a sense that it allow you to store data centrally and that can be referenced in multiple parts of configuration.

→ Benefits

→ you can add expression to locals, which allows you to compute values dynamically.

eg:- concat(aws_instance.blue.*id, aws_instance.green.*.id)

→ Local vs variables

→ Local are more private resources. you have to directly modify the code.

→ Variable value can be defined in a variety of places like terraform.tfvars, ENV variables, CLI and so on.

→ Locals are used when you want to avoid repeating the same expression multiple times.

→

→ Important point

→ local values are often referred to as just "locals"

→ local values are created by a "locals" block (plural), but you reference them as attribute on an object named local (singular)

* Data Sources

→ Data sources allow terraform to use/fetch information defined outside of Terraform

→ ${path.module} it return path of current folder.

→ A data source is accessed via a special kind of resources known as a data resources, declared using a data block!

→ Following data block request that Terraform read from a given data source ("aws_instance") and export the result under the given local name ("foo")

* Terraform Debug

→ Terraform has detailed logs which can be enabled by setting the TF_LOG environment variable to any value.

→ You can set TF_LOG to one of the log levels TRACE, DEBUG, INFO, WARN or ERROR to change the verbosity of the logs.

→ TRACE is the most verbose and it is the default if TF_LOG is set to something other than log level name.

→ To get log output in different file we can set TF_LOG_PATH.

* Terraform Format

→ Terraform fmt

→ make file in proper format.

* Load Order and Semantics

→ Terraform generally loads all the configuration files within the directory specified in alpha order.

→ file must either .tf or .tf.json.

* Dynamic Blocks

→ Dynamic blocks allow us to dynamically construct repeatable nested blocks which is supported inside resources, data provider, and provisioners blocks.

→ Iterators

→ iterator argument set the name of a temperory variable that represent the current element of a complex value.

* Terraform Validate

→ "Terraform validate" it primarly check whether a configuration is syntacially valid.

→ like unsupported argument, undeclared variables

* Tainting Resources

- replace :-

→ The replace option with terraform apply to force terraform to replace an object even though there are no configuration changes that would required.

eg:- terraform apply -replace="aws_instance.web"

→ In older version their is terraform taint

⇒ Spalat expression

→ Spalat expression allow us to get a list of all the attributes.

* Terraform Graph

→ Terraform graph refers to a visual representation of the dependency relationships between resources defined in your terrafom configuration.

→ It help to plan, debug, and manage complex infrastructure configuration.

* Saving Terraform plan to File

→ TO save the file of plan.

→ eg infra.plan (-out infra.plan)

* Terraform Settings

→ Terraform setting are used to configure project specific Terraform behaviors, such as requiring a minimum terraform version to apply to your configuration.

→ all the settings are gather in terraform blocks.

→ multiple option to write in terraform{}
  → Required terraform version
  → Required provider and version
  → Backend configuration
  → Experimental features.

→ we can prevent terraform from quering the current state during operation like terraform plan

→ we can achive with -refresh=false flag

→ Specifig the target

→ -target=resource flag used to target specific resources.

eg:- terraform plan -target=ec2

* Zipmap function

→ Construct a map from list of key and a corresponding list of value.

* Comments

#, //, /*-*/

* Resource Behavior and Meta Arguments

→ A resource block declare that you want to a particular infrastructure object to exist with a given settings.

→ meta argument

→ Terraform allow us to include meta-argument within the resource block which allow some details of the standard resources behavior to be customized on a per-resource basis.

meta argument → depends_on, count, for_each, lifecycle, provider

⇒ lifecycle block

→ create_before_destroy
→ prevent_destroy
→ ignore_changes
→ replace_triggered_by

⇒ Create_before_destroy :-

→ By default, when we change argument then first of all it will delete old resource and create new resource.

→ But using this first of all it create new resource and then delete old resource.

⇒ Prevent_Destroy

→ This meta argument, when set to true, will cause terraform to reject with an error any plan that could destroy the infrastructure, destroy object associated with the resource, as long as the argument remain present in the config.

* Ignore_changes

→ to ignore_changes some specific changes has been done but we do not need to change it.

→ all = to ignore all the attributes.

* Data type - SET

→ SET allows to store multiple item in a single variable

→ unordered and no duplicate allowed.

→ toset will convert list into set

* For_Each in terraform

→ each.key
each.value

* Terraform Modules

→ Terraform modules allow us to centralized the resources configuration and it make it easier for multiple projects to re-use the terraform code for projects.

→ we can use multiple module for a single projects.

→ For some infra resources not always the direct reference will work.

→ some module required specific input and values from the user side to be filled in before a resource get created.

→ some module page in github can contain multiple set of Modules together for different features.

→ so in that case we just need to extract exact sub-module required.

→ Choosing right terraform modules

1. View the downloads.
2. View the github repo document
3. Contributors
4. Reported issues
5. version history
6. use module maintain by hashicoop partner.

* Creating base Module structure for Custom Module

→ A base module folder
→ A sub folder contain name for each modules that are available.
→ Each sub folder contain achual Module code that other terraform project reference.
→ Module source code can be stored in varied location.

1. Github
2. HTTP URLs
3. S3 Bucket
4. Terraform Registry
5. Local paths.

→ to reperense we need to use module blocks.
→ source that contain location of Module.

→ Also module version is important.
→ So we need to specify the module version while reference.

→ To avoid hard coding we use required_provider block.

→ Also we can convert into variable to neglet hard-coding.

* Module Output

→ Output values make information about your infrastructure available on the command line and can ~~expoes~~ expose information for other terraform configuration to use.

* ROOT Module

→ Root module resides in the main working directory of your Terraform configuration. This is the entry point for your infra-definition.

eg:- module "~~ec~~ec2" {
source = "../../"
}

* Child Module

→ A module that has been called by another module is often refferred to as an child module.

* Standard Module Structure

→ The standard module structure is a file and directory layout Hashicorp recommends for reusable module.

→ A minimum recommended module is :-

eg:- $ tree minimal-module/

```
├ Readme.md
├ main.tf
├ variables.tf
└ outputs.tf
```

* Requirement for publishing Module

Requirement

→ Github → must be on github and public repo, only for public registry

→ Named → Module repositories must use this three-part name format (terraform-<provider>-<name>)

→ Repository description → github repository description is used to populate the short description of the module.

→ Standard module structure → The module must adhere to the Standard module structure.

→ X.Y.Z tag for release → The registory uses the tags to identify module version. eg :- V → V1.0.9

* Terraform workspace

→ Terraform workspaces enables us to manage multiple set of ~~in env~~ deployment from the same set of configuration file.

eg :- Terraform workspace list
new
select
delete

* gitignore :-

→ to not push secret file on git.

* Terraform Backend

→ Backend primarily determine where Terraform stores its states.

→ Basically stores in local disk.

→ ideal Architecture
  → code in Git repo
  → state file in central backend.

→ like backend eg :- S3, consul, HTTP

→ Accessing state in a remote service generally requires some kind of access credential.

* State Lock File Locking

→ State locking happen automatically on all operation that could write state you won't see any message that it is happening.

→ If it fail, it would continue.
→ Sometime not all backend support locking It is define in documentation.

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

→ It also cause multiple writers.
→ should only use when own lock is there.

* State locking S3

→ By default s3 not support state locking.

→ for that we need to use DynamoDB to achive it.

* Terraform state Modification

→ As your terraform usage become advance there are some cases where you may need to modify the terraform state.

→ It is imp to never modify the statefile directly. Instead make use of terraform state command.

→ Sub command
↳ list, mv, pull, push, rm, show, ~~remove~~

* Remote state data source

→ terraform_remote_state → allow to fetch output value.

* Terraform Import

→ Can automatically create the terraform configuration files of the resources you want to import.

→ available after 1.5

* Multiple provider configuration

→ Multi Alias Meta-Argument

→ Each provider can have one default configuration and any number of alternate configuration that includes an extra name segment

* Hashicorp Vault

→ Hashicorp vault allow organization to securly store secrets like tokens, password, certificates along with access management for protecting secrets.

→ Intracting with vault from terraform. Causes any secrets that you read and write to be persisted in both Terraform's state file.

→ when installing a particular provider for the first time, Terraform will pre-populate the hash with any checksum that are covered by the provider developers cryptographic signature, which usually covers all of the available package for that provider version across all supported Platforms.

→ At present, the dependency lock file tracks only provider dependencies.

* Sentinel

→ Sentinel is a policy-as-code framework integrated with the Hashicorp Enterprice products.

→ It enables fine grained, logic based policy decisions and can be extended to use information from the external sources.

plan → Sentinel → apply.

* Terraform cloud- Remote Backend

→ it stores Terraform state and may be used to run operation in Terraform cloud.

→ It also used with local operations.

*⇒ Air Gapped Environment

→ Air gapped env is a network security measure employed to ensure that a secure computer network is physically isolated from unsecured network, such as the public Internet.

→ used in military, financial stock exchange.