Terraform

* Infrastructure as code
* Terraform is a cloud agnostic tool. Can deploy to any cloud and providers.
* - Can be stored in version control systems e.g. git

**Infrastructure orchestration (io) vs config management (cm)**

* IO are tools that provision and manage infrastructure resources. E.g. terraform, AWS CloudFormation
* It focuses on creating and wiring resources
* Used to spin up identical environments (dev, stag, prod) and reproducible infra deployments
* CM are tools that configure software and operations on servers after they exist. e. g ansible, chef
* Focuses on what happens in servers
* Used for consistent package installs, users, permissions. Enforcing security hardening and rolling out app config changes.

Terraform

* User deploys infrastructure as code using terraform
* User runs a plan – tells you exactly what your code is trying to do. Has resources you’re trying to create.
* User then uses apply – Runs exactly what you need to deploy your resources into the cloud.
* AWS, Microsoft azure, GCP, Google Cloud Platform – Cloud providers where you can create, update and version your resources.

**Terraform state file = blueprint**

* Up – to – date Record of your existing infrastructure.
* Helps ensure idempotency – Produces same result no matter how many times it is run as if it were only executed once.

**Desired vs current state**

* Current state (.tfstate) = Terraform state file
* Desired state (.tf) = Terraform configuration of changes you want
* Terraform state file compares desired state to current state and makes changes to match them.

**Terraform providers**

* Plug-in that allows you to interact with cloud platform services
* Enables terraform to manage resources in the cloud.

**Terraform provider code**

* Consists of terraform block and provider block
* Terraform block – required providers, source, version
* Provider block – Provider name

**Terraform init**

* First command ran in any terraform project
* Initialises backend – where terraform stores state of your infrastructure. Allows terraform to track your resources.
* Initialises provider plugins – Downloads providers from terraform registry. Providers are like plugins that let terraform communicate with the cloud.

**Terraform plan**

* Allows you to see what your changes will do
* Analyses configuration files, compare them to the current state of your infrastructure and then generates a plan.
* Output of running ‘terraform plan’ represents your desired state.
* **+** means resources will be created
* **-** means resource will be destroyed
* **~** means resources will be modified
* Once your plan looks good it’s time to **apply.**

**Terraform apply**

* Applies ‘plan’ changes to your infrastructure.
* First it generates an ‘execution plan’ then you can confirm changes.

**Terraform destroy**

* Command to destroy all remote objects managed by a terraform configuration.
* Used to safely remove all infrastructure after completing projects
* Reads current configuration and state file and then generates ‘destruction plan’. After confirming, it destroys all resources.
* Terraform updates state file to reflect that resources have been destroyed.

**Resource block**

* Used to define piece of infrastructure you want to manage e.g. EC2, database
* Each resource block is for a specific resource type, provided by the cloud provider

**Terraform registry**

* Contains documents including how to deploy resources for each cloud using code.
* Create file (provider.tf) then paste the code using docs and the terraform init

**Deploying EC2 instance via terraform**

1. Create file (ec2.tf)
2. Paste code from terraform docs (required parts only) to create resource block
3. Launch EC2 instance and obtain AMI
4. Create access keys from IAM user
5. Paste into resource block :

export AWS\_ACCESS\_KEY\_ID="your\_access\_key\_id"

export AWS\_SECRET\_ACCESS\_KEY="your\_secret\_access\_key"

export AWS\_DEFAULT\_REGION="your\_default\_region

1. Run terraform plan to see the desired state
2. Run terraform apply and confirm actions. Make sure IAM has EC2FullAccess permission

**Terraform importing intro**

* Used to bring existing resources into cloud under terraforms management
* Resource block needs to be created manually

**Benefits**

* Useful because you can create consistent reproducible environments.
* No “it works on my account but not yours” issues
* Allows automation and scaling
* Version control and teamwork – terraform files can be stored in git.
* Multi-cloud and provider flexibility

| **Aspect** | **Manual Setup** | **Terraform (IaC)** |
| --- | --- | --- |
| **Consistency** | Human error prone | Reproducible every time |
| **Speed** | Slow, manual | Automated and fast |
| **Scalability** | Hard beyond few resources | Easy to scale up |
| **Auditing** | No version control | Full Git history |
| **Change Safety** | Risky | Safe with plan/apply |
| **Multi-cloud** | Complex | Unified through providers |

**Importing code block**

* Terraform registry provides import i.d for each resource.
* Run terraform plan after importing resources
* Desired state should match current state when importing

**Local statefiles**

* Stored on your computer. Suitable for small projects and individual use.
* Easy to set up. No additional configuration needed

**Remote statefiles**

* Stores remotely such as AWS S3, Azure Blob storage or terraform cloud.
* Suitable for team environment and larger projects.
* State locking – Prevents users from making changes at same time.
* Can automatically backup statefile and apply encryption.

**Configure backend with statefile**

* Error statuscode: 403 relates to permission issues.

**Terraform workflow**

1. terraform init – downloads providers and configures backend which manages statefile
2. terraform validate – validates terraform config files. Ensure there aren’t errors
3. terraform plan – creates execution plan. Compares current state to desired state and tells you necessary actions needed.
4. terraform apply -Applies the changes from terraform plan
5. terraform destroy – Destroys all terraform managed infrastructure

**Variables:**

* Used to set parameters in config files. Makes them more dynamic and resuable.
* DRY = Avoids repetition by defining a value once

**Input variables:**

* Create variable.tf file
* Copy from terraform doc
* Create terraform.tfvars file

**Local variable**

Used to store named values that stay in the configuration and can be reused.

| * **Use case** | * **Locals help by** |
| --- | --- |
| * Repeated expressions | * Defining once, reusing everywhere |
| * Complex logic | * Giving readable names |
| * Derived values | * Computing intermediate results |
| * Maintainability | * Centralising changes |

**Output variable**

* Output block: name of output, description, and value you want to output
* Used to expose critical information
* Be cautious when outputting sensitive information. Use terraforms sensitive attribute to prevent the values being displayed.

**Variable hierarchy**

* Lowest priority = Default values
* .tfvars file
* TF\_VAR environmental variables. Used to set values for variables.
* Highest priority = Command line flags

**Types of variables**

Primitive type – Simple individual values

Complex types – Combination of values

**Primitive types**

* String (some text e.g. hello)
* Numbers ( can be integers or decimal)
* Bool (either true or false)

**Complex types**

* List (sequence with same type of value)
* Map (collection of key-pairs)
* Object (collection of attributes that can each have a different type)

**Modules**

* Collection of configuration files that are grouped together to serve a specific process.
* The blueprint for building a simple piece of infrastructure.

**Benefits of modules**

* Reusability. Makes code DRY. Can be packaged for multiple projects
* Organisation
* Consistency across different environments (from dev to prod)
* Collaboration (modules can be shared)

**What makes a good module?**

* Simplicity – Not mixed with other dependencies that make the module more complex
* Documentation – Include readme or comments to explain how to use the module.
* Reusability
* Output values – modules should expose useful outputs that other parts of terraform configuration can use.

**Module DEMO**

1. Create 1 folder named modules and within it another folder named EC2
2. Move ec2.tf and variable.tf into folder
3. Create main.tf file outside that folder to ‘call’ module
4. Use a module block. source = “./modules/ec2”
5. Run terraform init. Initialises module
6. Run terraform plan
7. Run terraform apply