

Terraform - Infrastructure-as-Code (IaC)

Eloi Perdereau, Hélène Coullon

<https://helene-coullon.fr/pages/ue-nuage-25-26/>

IMT Atlantique

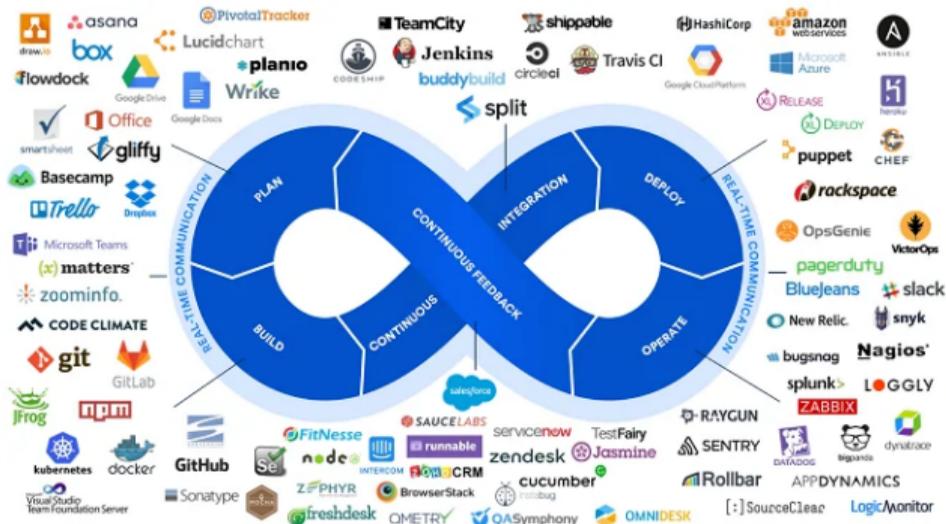
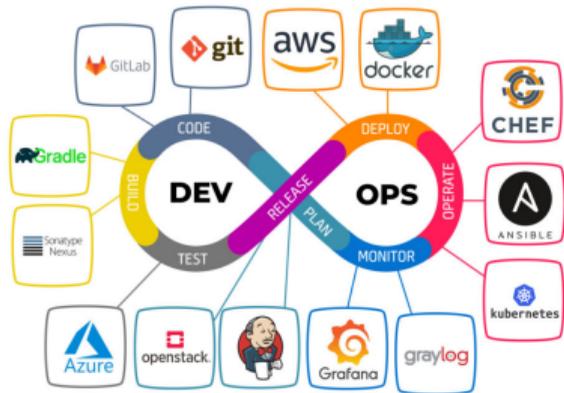
Table of contents

1. Introduction
2. Terraform Concepts
3. Good practices
4. Your turn...

Introduction

DevOps

DevOps practices tries to reduce the time of release cycles, make more flexible (agile) software development etc., by bridging the gap between development and operation.



Relationship between DevOps and Cloud computing?

Lots of applications are now migrated to micro-services architectures and are deployed in the Cloud because

- servers are operated by a tiers
- companies pay only what they consume
- easy elasticity
- etc.

→ DevOps cycles very often integrate Cloud operations.

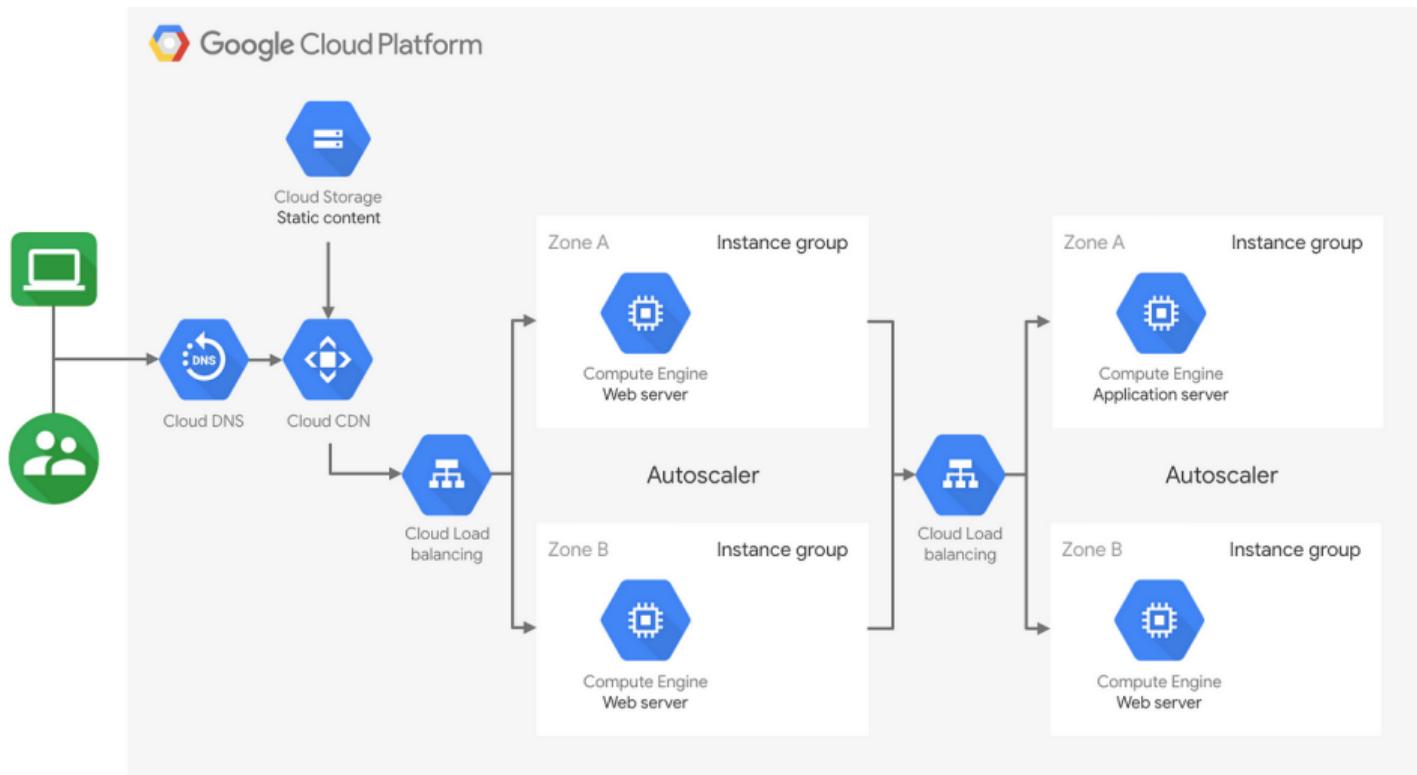
Infrastructures = APIs

The Cloud computing paradigm has transformed **infrastructure** (also platforms and software) to **external APIs** to request.

Nowadays infrastructures are like any piece of software offering services.

- request for a **bucket** to store content,
- request for a **GKE** cluster to host micro-services applications,
- request for **VMs** to install databases or larger software,
- etc.

Example



What is Infrastructure-as-Code?

Infrastructure-as-Code (IaC)

- Avoid manual or ad-hoc way of handling (create, update, delete) complex infrastructures,
- See infrastructure as standardized text files living next to the codebase and describing what and where to deploy,
- IaC scripts can then be shared, versioned, reviewed, etc. with the usual development tools.

Associated concepts

- imperative/declarative
- idempotence
- reversibility
- state management

Configuration management

Originally made to automate and make more reusable and flexible the configuration of servers, machines, virtual machines

- Puppet
- Chef
- Ansible

Provisioning tools

Originally made to automate and make more reusable, flexible and safe the management of Cloud infrastructures

- Heat (OpenStack)
- Cloud Formation (AWS)
- Terraform and OpenTofu
- Pulumi

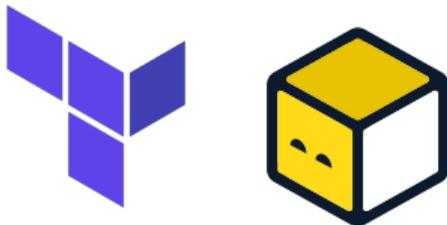
Orchestration tools

Made to orchestrate the lifespans of a large set of containers and their deployment on servers

- Docker Swarm
- Kubernetes
- Nomad

Provisioning

In this module we focus on one provisioning tool: **terraform** (and its fork **OpenTofu**)



- **state**: the current state of the infrastructure and the desired state specified by the user. Both can change over time.
- **reconciliation**: declarative IaC provisioning tools try to reconcile the current state with the desired state.
- **plan**: to reconcile, declarative IaC tools automatically generate a plan to execute/apply.

Why using a provisioning tool?

I can do that through the graphical interfaces of Cloud providers! Yes but...

- Long and error-prone manual procedures.
- Difficult and error-prone when collaborating.
- No clear/central vision of the state of your infrastructure.
- Not scalable.

I can do that with Cloud providers CLIs and scripts! Yes but...

- You have to know as many CLIs as the number of Cloud providers you are using.
- A script is less specialized and structured than IaC, more difficult to write/read and maintain.
- You have to manually handle the state of your infrastructure which is difficult and error prone.

Other Provisioning IaC Tools

Specific to Cloud Providers

- *CloudFormation* for AWS: YAML DSL or Typescript classes (API)
- *Azure Resource Manager (ARM)* for Azure: custom DSL
- *Heat* for OpenStack: YAML DSL

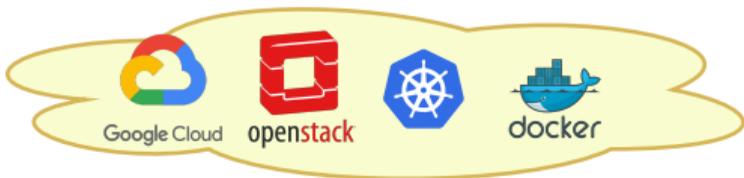
Comparatively, Terraform can handle *all* cloud providers within a single state file. Infrastructure resources described in a custom DSL.

Pulumi

- Competitor of Terraform. Same purpose.
- Reuses Terraform providers
- Agnostic of the Language: API in Python, NodeJS, .NET, Go or YAML

Terraform Concepts

Concepts of Terraform



Providers

- Schema for *Resources*, *data sources* and *provider credential*
- Implementation CRUD API calls

Users

- HCL *Resources* and *Providers* configuration
- CLI commands
- Managed State

A screenshot of the Google Cloud console interface. The main content area shows 'Gateways, Services & Ingress' for a 'Kubernetes Engine' cluster. It includes a table with columns for Name, Status, Type, Endpoints, Pods, Namespace, and Clusters. The table lists several services like 'redis', 'result', and 'vote' with their respective statuses and configurations.

Name	Status	Type	Endpoints	Pods	Namespace	Clusters
redis	OK	Cluster IP	34.118.231.15	1/1	default	tuto-terraform-405010-gke
result	OK	External load balancer	35.203.89.78:8000	1/1	default	tuto-terraform-405010-gke
vote	OK	External load balancer	34.47.21.158:8000	2/2	default	tuto-terraform-405010-gke

A screenshot of the Terraform documentation page for the 'google_compute_instance' resource. It shows the resource name, a brief description, and an 'Example Usage' section with HCL code snippets for creating a VM instance and a disk.

```
resource "google_compute_instance" "default" {
  name         = "my-instance"
  display_name = "Instance for VM example"
}
```

```
resource "google_compute_disk" "default" {
  name        = "my-disk"
  machine_type = "f1-micro"
  size        = "10GB"
}
```

```
# vm.tf

resource "google_compute_instance" "vm" {
  name           = "redis"
  machine_type   = "e2-medium"
  network_interface {
    network = var.network-name
  }
}
```

Automatic ordering

Declarative state: declare *what* not *how*

The desired state is written by the DevOps in `.tf` files. A DSL on top of HCL config. language.

- The **order** of provisionment is determined automatically, Terraform will **create**, **update** and **delete** resources in the right order.

Automatic ordering

Declarative state: declare *what* not *how*

The desired state is written by the DevOps in `.tf` files. A DSL on top of HCL config. language.

- The **order** of provisionment is determined automatically, Terraform will **create**, **update** and **delete** resources in the right order.
- The order is defined when resources **refer** to each other, creating a dependency graph.

Automatic ordering

Declarative state: declare *what* not *how*

The desired state is written by the DevOps in `.tf` files. A DSL on top of HCL config. language.

- The **order** of provisionment is determined automatically, Terraform will **create**, **update** and **delete** resources in the right order.
- The order is defined when resources **refer** to each other, creating a dependency graph.
- **Changes** in the declared state are compared against the state file to create the plan.

We can create multiple versions of the same replicated infrastructure which differ slightly, e.g. for different environments like *dev*, *prod*, *test*.

Resources

- A resource can represent anything. e.g. VM, docker image, virtual network, ip, user, account, role, etc.
- Providers furnish an API that lists
 1. Available resource **types**.
 2. For each of them their **parameters**.

Resources

- A resource can represent anything. e.g. VM, docker image, virtual network, ip, user, account, role, etc.
- Providers furnish an API that lists
 1. Available resource **types**.
 2. For each of them their **parameters**.

Terraform Core

- Configuration : every **.tf** files \Rightarrow provider and resource declarations.
- The current directory constitutes the **root module**.
- Modules can be **shared** and reused through a registry
- State file : contains the **current state** of resources under Terraform's management.
- Upon each CLI call, the state file is **refreshed** with the actual resources.

Basic Architecture

Resources

- A resource can represent anything. e.g. VM, docker image, virtual network, ip, user, account, role, etc.
- Providers furnish an API that lists
 1. Available resource **types**.
 2. For each of them their **parameters**.

Terraform Core

- Configuration : every **.tf** files \Rightarrow provider and resource declarations.
- The current directory constitutes the **root module**.
- Modules can be **shared** and reused through a registry
- State file : contains the **current state** of resources under Terraform's management.
- Upon each CLI call, the state file is **refreshed** with the actual resources.

Terraform detects changes in the configuration and **plan** API calls accordingly.

Terraform Workflow Illustration

User Configuration

```
# vm.tf (HCL format)

terraform {
  required_providers {
    google = {
      source = "hashicorp/google"
      version = "5.6.0"
    }
  }
}

provider "google" {
  project = var.gcp_project_id
  region = var.gcp_region
  credentials = file(var.gcp_key)
}

resource "google_compute_instance" "vm"
{
  name          = "redis"
  machine_type = "e2-medium"
  network_interface {
    network = "default"
  }
}
```

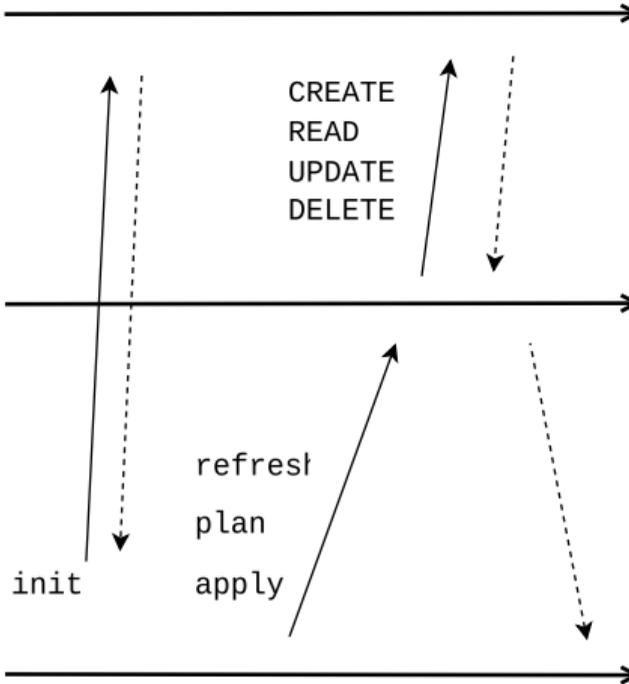
Distant API



Terraform Managed State

```
# terraform.tfstate (JSON format)

resource "google_compute_instance"
"vm" {
  name          = "redis"
  id            = "projects/tf-80/..."
  instance_id   = "3012364625718931000"
  network_interface {
    name        = "nic0"
    network     =
"https://www.googleapis.com/compute"
    network_ip  = "10.162.0.20"
  }
}
```



Commands for different stages

```
terraform init
```

Initialize the working directory by **downloading** providers and distant modules.

Commands for different stages

`terraform init`

Initialize the working directory by **downloading** providers and distant modules.

`terraform plan`

Produces an execution **plan** with details on which resources to create/update/replace/delete by "compiling" the `.tf` configuration and comparing it with the state file.

Plans can be stored to be applied in the future.

Commands for different stages

terraform init

Initialize the working directory by **downloading** providers and distant modules.

terraform plan

Produces an execution **plan** with details on which resources to create/update/replace/delete by "compiling" the `.tf` configuration and comparing it with the state file.

Plans can be stored to be applied in the future.

terraform apply

Produces a plan and **execute** it. An execution may fail depending on the provider's implementation and hidden infrastructure constraints.

Commands for different stages

`terraform init`

Initialize the working directory by **downloading** providers and distant modules.

`terraform plan`

Produces an execution **plan** with details on which resources to create/update/replace/delete by "compiling" the `.tf` configuration and comparing it with the state file.

Plans can be stored to be applied in the future.

`terraform apply`

Produces a plan and **execute** it. An execution may fail depending on the provider's implementation and hidden infrastructure constraints.

`terraform destroy`

Calls the provider to **delete all managed resources**.

HCL Language Syntax (1): Attributes

aka “argument”, “field”

Attributes are distinguished with the **equal** sign = meaning *assignment*.

The value can be any expressions: function calls, lists, objects, references, etc.

- `name = "redis server"`
- `credentials = file("./creds.json")`
- `labels = { app = "redis" }`
- `image = docker_image.redis.name`
- etc.

HCL Language Syntax (1): Attributes

aka “argument”, “field”

Attributes are distinguished with the **equal** sign = meaning *assignment*.

The value can be any expressions: function calls, lists, objects, references, etc.

- `name = "redis server"`
- `credentials = file("./creds.json")`
- `labels = { app = "redis" }`
- `image = docker_image.redis.name`
- etc.

Multiple definitions of an attribute are **forbidden**. They are *single assignment*.

In addition to arguments within a block, there are a few **meta attributes** that have special semantics: `count`, `for_each` and `depends_on` and `lifecycle`.

Blocks

e.g. `resource "docker_image" "redis" { ... }`

- Have a mandatory **key** identifier, here `resource`. It have a meaning in the context where it is defined, like standard attributes.
- Strings can be attached, here `docker_image` and `redis`.
- They can be **referenced** from another part of the configuration.

HCL Language Syntax (2): Blocks

Blocks

e.g. `resource "docker_image" "redis" { ... }`

- Have a mandatory **key** identifier, here `resource`. It have a meaning in the context where it is defined, like standard attributes.
- Strings can be attached, here `docker_image` and `redis`.
- They can be **referenced** from another part of the configuration.

Block can be **embedded**. e.g. in a container resource:

```
mounts { volume_options { no_copy = true }}
```

HCL Language Syntax (2): Blocks

Blocks

e.g. `resource "docker_image" "redis" { ... }`

- Have a mandatory **key** identifier, here `resource`. It have a meaning in the context where it is defined, like standard attributes.
- Strings can be attached, here `docker_image` and `redis`.
- They can be **referenced** from another part of the configuration.

Block can be **embedded**. e.g. in a container resource:

```
mounts { volume_options { no_copy = true }}
```

Multiple blocks with identical keyword are **allowed** and results in an **array** of objects. e.g. to refer to a particular mount option:

```
mounts[0].volume_options[0].no_copy
```

Kinds of top-level blocks

Terraform has concepts for each kind of block that can be declared at the **top-level**.

resource, data, provider and variables

The references for those blocks are found in the provider's documentation at <https://registry.terraform.io/providers/>.

- The *resource* block is the main state declarations of **managed resources**.

Kinds of top-level blocks

Terraform has concepts for each kind of block that can be declared at the **top-level**.

resource, data, provider and variables

The references for those blocks are found in the provider's documentation at <https://registry.terraform.io/providers/>.

- The *resource* block is the main state declarations of **managed resources**.
- The *data* block is for **read-only** resource.

Kinds of top-level blocks

Terraform has concepts for each kind of block that can be declared at the **top-level**.

resource, data, provider and variables

The references for those blocks are found in the provider's documentation at <https://registry.terraform.io/providers/>.

- The *resource* block is the main state declarations of **managed resources**.
- The *data* block is for **read-only** resource.
- The *provider* block sets configuration parameters for a provider.

Kinds of top-level blocks

Terraform has concepts for each kind of block that can be declared at the **top-level**.

resource, data, provider and variables

The references for those blocks are found in the provider's documentation at <https://registry.terraform.io/providers/>.

- The *resource* block is the main state declarations of **managed resources**.
- The *data* block is for **read-only** resource.
- The *provider* block sets configuration parameters for a provider.
- The *variable* blocks for **input** values set from outside of *.tf* files.

There are a few other top-level blocks, e.g. *module*, *check*, *import*.

Variables and References

Variables

There is **three kinds** of variables depending on how we **use** them:

Inputs, Outputs, and Locals.

Declared respectively with the top-level blocks *variable*, *output*, and *locals*.

Variables and References

Variables

There is **three kinds** of variables depending on how we **use** them:

Inputs, Outputs, and Locals.

Declared respectively with the top-level blocks *variable*, *output*, and *locals*.

References to resources, data sources and variables

- Resources attributes are referenced with **type** and **name** of the resource, e.g.
docker_image.redis.image_id
- To reference a *data source*, we use the keyword **data**, e.g.
data.docker_image.redis.image_id.
- For input and local variables, we use **var** and **local** keywords, e.g.
var.my_input_var
local.my_local_var
- Terraform has other such special variable keywords, e.g. *module*, *each*, *path*.

Good practices

Objective: avoid troubleshooting

- Read and understand carefully each declarations and plan.
- Version control your Terraform codes. *Beware not to commit **secrets**.*
- CI/CD on your Terraform infrastructure.
- Store the Terraform state files on remote storages with lock mechanisms.

Your turn...

Your turn!

- Tutorial with the Docker provider
 - TF calls the local Docker client
- Tutorial with the Kubernetes provider
 - TF calls a distant Kubernetes API server
- Project
 - Deploy the same Voting App
 - Use Terraform to deploy with
 - Docker only
 - Kubernetes only
 - Kubernetes + Proxmox for Redis