# Introduction au DevOps

## Infrastructure-as-Code

Jolan PHILIPPE

16 Septembre 2026

Université d'Orléans

# Infrastructure-as-Code

**Virtualization et Conteneurisation**

Customiser, configurer
tester l'application
et la conteneuriser

**Provisionnement d'infrastructure**

Demander ressources
physiques ou virtuelles;
configurer le réseau;
et règles sécurité

**Installation et Configuration**

Installer les services
(app + deps)
configurer les services;
et les intégrer

**Orchestration de cycle de vie**

Upgrades auto;
Backup et recovery;
Surveillance;
Passage à l'échelle

# Technologie d'Infrastructure-as-Code

| Technology | Conteneurisation | Provisioning | Conf. Mgmt | Cluster. Mgmt | Target |
|---|---|---|---|---|---|
| containerd | ✔ | ✖ | ✖ | ✖ | Agnostic |
| Docker | ✔ | ✖ | ✖ | ✖ | Agnostic |
| LXC | ✔ | ✖ | ✖ | ✖ | Dedicated (Linux) |
| Podman | ✔ | ✖ | ✖ | ✖ | Agnostic |
| CRI-O | ✔ | ✖ | ✖ | ✖ | Dedicated (Kubernetes) |
| Azure Resource Manager | ✖ | ✔✔ | ✔ | ✔ | Dedicated (Azure) |
| CloudFormation | ✖ | ✔✔ | ✔ | ✔ | Dedicated (AWS) |
| Heat | ✖ | ✔✔ | ✔ | ✔ | Dedicated (OpenStack) |
| Pulumi | ✖ | ✔✔ | ✔ | ✖ | Agnostic |
| Terraform | ✖ | ✔✔ | ✔ | ✖ | Agnostic |
| Ansible | ✖ | ✔ | ✔✔ | ✖ | Agnostic |
| CFEngine | ✖ | ✔ | ✔✔ | ✖ | Agnostic |
| Chef | ✖ | ✔ | ✔✔ | ✖ | Agnostic |
| Puppet | ✖ | ✔ | ✔✔ | ✖ | Agnostic |
| SaltStack | ✖ | ✔ | ✔✔ | ✖ | Agnostic |
| Kubernetes | ✖ | ✔✔ | ✖ | ✔✔ | Hybrid (Containers) |
| Nomad | ✖ | ✔✔ | ✖ | ✔✔ | Agnostic |
| Docker Swarm | ✖ | ✔✔ | ✔ | ✔✔ | Dedicated (Docker) |
| Juju | ✖ | ✔✔ | ✔✔ | ✔✔ | Agnostic |
| TOSCA | ✖ | ✔✔ | ✔✔ | ✔✔ | Agnostic |

# Plan

Conteneurisation avec Docker

Provisionnement avec Terraform

(Go dans l'Infrastructure-as-Code)

Installation et configuration avec Ansible

Orchestration avec Kubernetes

"Quality Assurance in Infrastructure as Code: Issues, Approaches, and Open Challenges"

# Plan

Conteneurisation avec Docker

Provisionnement avec Terraform

(Go dans l'Infrastructure-as-Code)

Installation et configuration avec Ansible

Orchestration avec Kubernetes

"Quality Assurance in Infrastructure as Code: Issues, Approaches, and Open Challenges"

# Les rôles des outils d'IaC

**Virtualization et Conteneurisation**

Customiser, configurer
tester l'application
et la conteneuriser

**Provisionnement d'infrastructure**

Demander ressources
physiques ou virtuelles;
configurer le réseau;
et règles sécurité

**Installation et Configuration**

Installer les services
(app + deps)
configurer les services;
et les intégrer

**Orchestration de cycle de vie**

Upgrades auto;
Backup et recovery;
Surveillance;
Passage à l'échelle

`https://www.docker.com/blog/docker-for-devops/`

# Un conteneur c'est quoi ?

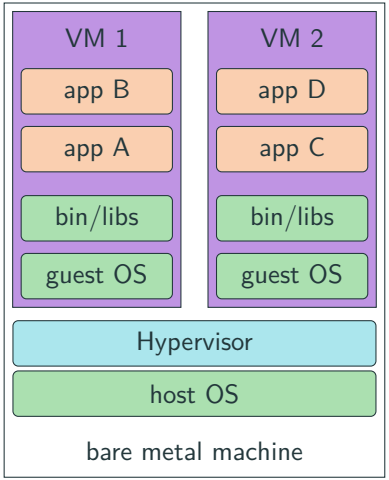
# A propos du kernel Linux

Le kernel c'est le cœur d'un système d'exploitation (OS)

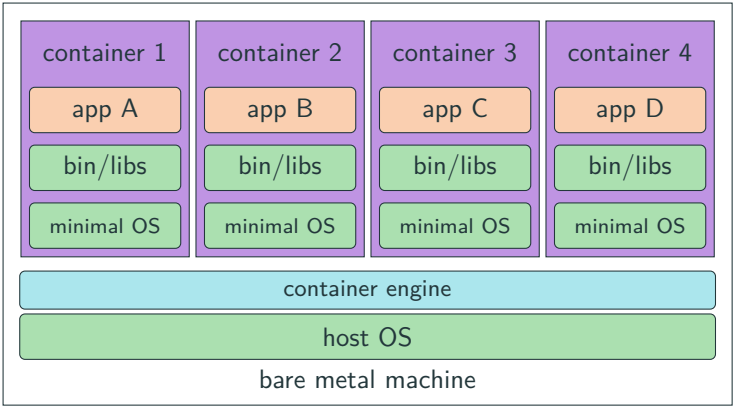
- C'est une partie de l'OS toujours chargé en mémoire.
- Il contrôle les ressources physiques, ou hardware, (e.g., I/O, mémoire, cryptographie, CPU) en utilisant des pilotes, ou drivers.
- Il arbitre les conflits et la concurrence entre les processus.
- Il optimise l'utilisation de ressources (e.g., cache, mémoire, CPU, système de fichiers, réseau)

Le kernel est l'un des premiers programmes chargé au démarrage.

# Comparaison à gros grains entre les VMs et les conteneurs




VMs avec un hyperviseur




Containers

# Conteneurs

## Avantages des conteneurs

- Isolation
- Portabilité
- Limitations des ressources dupliquées
- Impact limité sur les performances
- Démarrage rapide

Le modèle est différent et la façon dont les applications sont déployées est différente !

Que se cache-t-il dessous ?

# Un peu plus à propos de Linux

Dans Unix/Linux, tout est fichier : une fichier, un dossier, un périphérique, etc.

## Système de fichiers

- Organisation des fichiers hierarchique sous la forme d'un arbre
- `/` est la racine du système de fichiers
- `/sys` contient les fichiers du système
- `/etc` contient les fichiers de configuration et les scripts
- `/media` contient les partitions de disques, périphériques, etc.
- etc.

Un conteneur est aussi un ensemble de fichiers

# Images

Une image de conteneur est une archive de fichiers, incluant:

- une racine de système de fichiers
- des bibliothèques, des packages, etc. (i.e., des dépendances)
- l'application, ou le service, à démarre
- etc.

L'image contient l'environnement requis pour exécuter l'application ou le service sur le noyau hôte.

# Portabilité des images

Cet environnement est portable d'un hôte à un autre, si un kernel compatible est présent (WSL2 sur Windows, ou VM sur MacOS)

Attention: La limite, c'est l'architecture matérielle : par exemple une image construite pour amd64 ne tournera pas nativement sur un hôte arm64 (Mac M1/M2) sans émulation (e.g., qemu).

# Registre d'images

- Les images Docker sont stockées dans des **registres** (e.g., *Docker Hub* `https://hub.docker.com/`).
- La commande `docker pull` permet de **télécharger une image** vers la machine locale.
- Exemple :
  `> docker pull MyImage`
- Résultat :
  - Télécharge l'image `MyImage`
  - Rend l'image disponible localement pour créer des conteneurs.

# Example: Alpine

L'image docker de Linux Alpine est souvent utilisée par les conteneurs.

- Distribution très légère de Linux
- `https://hub.docker.com/_/alpine`

Télécharger une image alpine

```
> docker pull alpine
```

Démarre un conteneur en utilisant l'image alpine et démarre une invite de commande sh interactive dans celui-ci

```
> docker run -it alpine /bin/sh
```

Une fois démarrer, on peut intéragir avec le système de fichier:

```
in alpine > ls -al
```

Mais il n'y a pas de kernel:

```
in alpine > ls /boot
```

# D'une image vers un conteneur

Comme vu précédemment, une image est une archive d'un système de fichiers. Créer un conteneur consiste à:

- Donner une quantité limité de ressources au conteneur
- Créer un environnement isolé au processus du conteneur
- assigner la racine du système de fichiers de l'image à la la racine du système de fichiers du conteneur

# D'une image vers un conteneur

**`cgroups`**

Linux Control Groups (**`cgroups`**) limite la quantité de ressource qu'un processus peut utiliser (CPU, mémoire, latence, etc).

**`namespaces`**

Linux Namespaces assure que chaque processus voit sa propre vue personnelle du système (fichiers, processus, interfaces réseaux, hostname, etc).

**`chroot, pivot_root`**

Change la racine du système de fichiers d'un processus.

# Docker en détails

# Vue d'ensemble de Docker

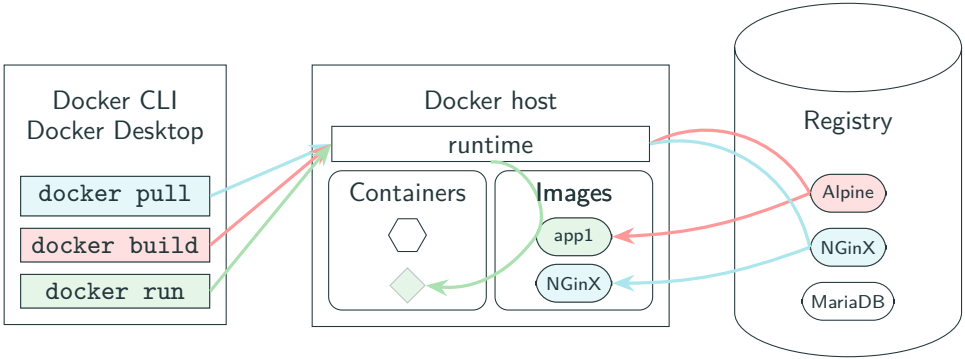
Les différents éléments de Docker

- CLI (Command Line Interface)
- Docker runtime
- Les images et le registre
- Les conteneurs

**Les rôles du Docker runtime**

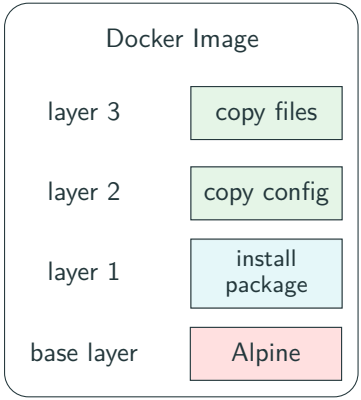
- Démarrer (start) et arrêter (stop) des conteneurs
- Manager des images
- Manager les réseaux
- Manager les volumes
- etc.

# Vue d'ensemble de Docker



# Structure d'une image Docker

Une image Docker est construite en assemblant différentes couches.




Optimisation de stockage

- Les couches sont partagées par différentes images pour optimiser le stockage;
- Pour se faire, chaque couche est identifiée par une fonction de hashage.

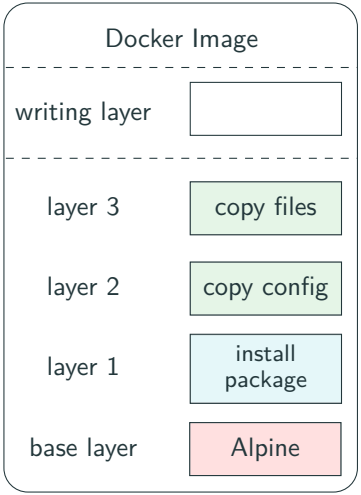
# Écrire dans un conteneur ?

Une image Docker est immuable! C'est à dire qu'elle ne peut pas être modifiée.
Au runtime, une couche virtuelle est créée dans le conteneur, au dessus de l'image

- Il est possible d'écrire dans cette couche;
- Cette couche n'est pas partagée par les autres conteneurs;
- Cette couche est détruite avec le conteneur.

**Volumes**

Si les données doivent être persistantes et partagées entre les conteneurs, un volume doit être utilisé. Voir `https://docs.docker.com/engine/storage/volumes/`



# Créer une image avec un Dockerfile

# Principes du Dockerfile

Un Dockerfile contient un ensemble de commandes pour créer une image Docker.

- Empêche de construire des images manuellement, “from scratch”;
- Offre une manière pour Docker de construire des couches et empêcher les commandes inutiles;
- Un Dockerfile ressemble à un fichier bash avec des instructions à appliquer.

# Exemple de Dockerfile

- `FROM` pour indiquer quelle image de base utiliser pour construire notre image;
- `RUN` pour exécuter une commande par dessus l'image de base;
- `ENV` pour déclarer des variables d'environnement;
- `ENTRYPOINT` pour définir quelle commande doit être exécutées au démarrage du conteneur.

```
1    FROM alpine
2    RUN apt update
3    RUN apt install -y htop
4    ENV TERM=xterm
5    ENTRYPOINT /bin/htop
```

La documentation complète est accessible ici:
`https://docs.docker.com/reference/dockerfile/`

# Exemple de Dockerfile

- `FROM` avec une version d'image;
- `WORKDIR` pour indiquer le dossier de travail dans lequel se placer quand le conteneur démarrera;
- `ADD` pour ajouter des fichiers depuis la machine locale (i.e., hôte) à l'image du conteneur;
- `CMD` pour définir quelles commandes doivent être exécutées au démarrage du conteneur.

```
1   FROM python:3.8-alpine
2   RUN apt updateWORKDIR /app
3   ADD . /app/
4   RUN pip install -r requirements.txt
5   CMD ["python", "movie.py"]
```

# Quelques subtilités des commandes Dockerfile

- `ADD` ≠ `COPY`: `ADD` fait la même chose que `COPY` mais permet aussi de décompresser automatiquement des archives locales, et télécharger une ressource depuis une URL

- `ENTRYPOINT` ≠ `CMD`: `ENTRYPOINT` sert à démarrer un programme principal fixe (e.g., python, java) alors que `CMD` sert à remplacer les argumments passé après `docker run image` (e.g., `/bin/sh` dans le premier exemple)

# Construire une image à partir d'un Dockerfile

**Commande**

```
docker build [OPTIONS] PATH
```

```
> docker build .  -t "monapp:latest"
```

- `docker build` est la commande pour construire une image Docker;
- `.` est le chemin vers le Dockerfile;
- `-t` est une option pour nommer cette image (ici `"monapp:latest"` se réfère à un nom et une version);
- Par défaut, le Dockerfile est `PATH/Dockerfile`, mais vous pouvez donner un autre nom en utilisant l'option `-f`.

# Docker CLI

> `docker -h` donne l'aide racine de la CLI
> `docker images -h` donne l'aide pour les images

> `docker pull hello-world` télécharge l'image `hello-world`
> `docker images`, ou `docker images ls`, donne une liste des images téléchargées
> `docker image inspect id` donne des détail sur l'image correspondante à l'`id`

> `docker run hello-world` créer et démarre un conteneur à partir d'une image
> `docker ps` affiche les conteneurs en cours d'exécution
> `docker ps -a` affiche tous les conteneurs, y compris ceux aux statut `exited`
> `docker container inspect id` donne des détail sur le conteneur correspondant à l'`id`

> `docker container prune` supprime tous les conteneurs qui ne sont plus utilisés
> `docker image rm hello-world` supprime l'image locale `hello-world`

# Quelques bonnes pratiques

# Penser aux couches

Dans les versions plus anciennes de Docker, chaque ligne dans le Dockerfile créait une couche. En conséquences:

- Trop de couches intermédiaires, pouvant être coûteuses;
- Le temps de build pouvait être impacté;
- Les optimisations de stockage pouvaient être impossibles;
- Aujourd'hui, `RUN`, `COPY` et `ADD` uniquement créent de nouvelles couches.

**Bonne pratique 1**

Pensez à vos couches quand vous utiliser `RUN`, `COPY` et `ADD` dans votre Dockerfile

# Réduire la taille d'une image

**Bonne pratique 2**

N'installez uniquement que les dépendances nécessaires dans votre Dockerfile

- Si vous utilisez `apt` pour installer des packages, utilisez l'option `-no-install-recommends`
- Si possible, supprimez les fichiers intermediaires non requis quand vous appliquez `RUN`

# Construction en plusieurs étapes

## Bonne pratique 3

- Réduisez la taille des images en supprimant les dépendances de compilation dans l'image finale.
- L'image finale contient uniquement les dépendances nécessaires à l'exécution du service.
- Une image de base bien adaptée aux fichiers exécutables uniquement est **`scratch`** ou **`alpine`**.

```
1  FROM golang as builder
2  RUN apt update && apt install -y git protobuf-compiler golang-goprotobuf-def && \
3      git clone https://gitlab.imt-atlantique.fr/url && \
4      cd productcatalogservice && \
5      go mod download && \
6      mkdir genproto && \
7      protoc --go_out=plugins=grpc:genproto -I . productcatalogservice.proto && \
8      CGO_ENABLED=0 go builder
9
10 FROM scratch
11 COPY --from=builder /go/productcatalogservice/productcatalogservice /
12 COPY --from=builder /go/productcatalogservice/products.json /
13 ENTRYPOINT ["/productcatalogservice"]
```

N'importe qui peut déposer une image Docker sur Docker Hub !

**Bonne pratique 4**

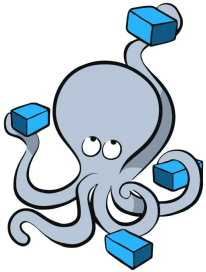
- Préférez toujours les images Docker officielles.
- Vérifiez que l'image Docker est régulièrement mise à jour.
- Soyez sur que l'image contient ce que vous pensez en utilisant
  - `> docker history image_name`
  - des outils comme `dive`
- Soyez sure de mettre à jour les images que vous utilisez.

# Un peu plus dans les Dockerfiles

- Exposer ses ports dans un Dockerfile
  - `EXPOSE 80` pour exposer un numéro de port
  - `EXPOSE 56/udp` pour exposer un numéro de port pour un protocole précis
- Ajouter des données purement informatives
  - `LABEL maintainer="Jolan PHILIPPE"`
- Ajouter des variables d'environnement
  - `ENV ADMIN_USER="Alice"`
  - ou directement via la CLI: `docker run -e ADMIN_USER="Bob"`
- Ajouter des volumes depuis l'hôte
  - `VOLUME /myapp/data`

# Déployer une pile logicielle avec Docker Compose

# Automatiser le déploiement d'applications dans des conteneurs



docker Compose

- Déploie facilement une pile logicielle conteneurisée
- Défini votre déploiement avec un seul fichier YAML (conteneurs, volumes, réseaux, etc.)
- Les fichiers de déploiement deviennent faciles à partager, à contrôler leurs versions, etc. (c'est de l'IaC :wink:)

# Structure de `compose.yaml`

Spécification complète: `https://docs.docker.com/reference/compose-file/`

- **services**
  - Nom du service
    - **image** Docker ou chemin de **build** pour le Dockerfile
    - **ports** exposés par le service
    - **networks** utilisés par le service
    - **volumes** utilisés par le service
    - variables d'**environment** utilisés par le service avec sa valeur
    - **depends_on** un autre service
- **volumes**
- **networks**

```yaml
version: '3.3'

services:
  lychee:
    image: linuxserver/lychee:4.7.0
    container_name: Gdsn-photos-web
    restart: always
    networks:
      - web
      - default
    volumes:
      - ./conf:/config
      - /srv/gdsn_photos_lychee:/pictures
    labels:
      - "traefik.http.routers.gdsn_photos.rule=Host(`photos.gdsn.fr`)"
  db:
    image: mysql:5.7
    container_name: Gdsn-photos-db
    restart: always
    networks:
      - default
    volumes:
      - /srv/gdsn_photos_db_data:/var/lib/mysql
    environment:
      MYSQL_ROOT_PASSWORD: secret
      MYSQL_DATABASE: lychee
      MYSQL_USER: lychee
      MYSQL_PASSWORD: secret
    labels:
      - "traefik.enable=false"

networks:
  web:
    name: traefik_web
    external: true
```

Il est très important de comprendre que **`Docker compose`** crée un DNS pour que les conteneurs puissent s'appeler les uns les autres sans connaître leurs adresses IP.

# CLI

Quelques commandes importantes :

- `> docker-compose build` pour construire, ou reconstruire, des images
- `> docker-compose up` pour créer et démarrer des conteneurs, réseaux, etc.
- `> docker-compose stop` pour arrêter des conteneurs, réseaux, etc.
- `> docker-compose down` pour arrêter et supprimer des conteneurs, réseaux, etc.

Toute la documentation : `https://docs.docker.com/compose/reference/`

Quelques exemples : `https://github.com/docker/awesome-compose`

# Démonstration

# Plan

Conteneurisation avec Docker

Provisionnement avec Terraform

(Go dans l'Infrastructure-as-Code)

Installation et configuration avec Ansible

Orchestration avec Kubernetes

"Quality Assurance in Infrastructure as Code: Issues, Approaches, and Open Challenges"

# Les rôles des outils d'IaC

**Virtualization et Conteneurisation**

Customiser, configurer
tester l'application
et la conteneuriser

**Provisionnement d'infrastructure**

Demander ressources
physiques ou virtuelles;
configurer le réseau;
et règles sécurité

**Installation et Configuration**

Installer les services
(app + deps)
configurer les services;
et les intégrer

**Orchestration de cycle de vie**

Upgrades auto;
Backup et recovery;
Surveillance;
Passage à l'échelle

`https://developer.hashicorp.com/terraform`

# Une infrastructure, c'est quoi ?

# Définitions

**Infrastructure**

L'infrastructure fait référence au logiciel, à la plate-forme ou au matériel qui fournit ou déploie des applications. *Source: Infrastructure-as-code, patterns and practices*

**Infrastructure as a Service (IaaS)**

L'IaaS (infrastructure as a service) est un service de cloud computing offrant des ressources informatiques matérielles (stockage, réseau, baies de serveurs). *Source: CNIL*

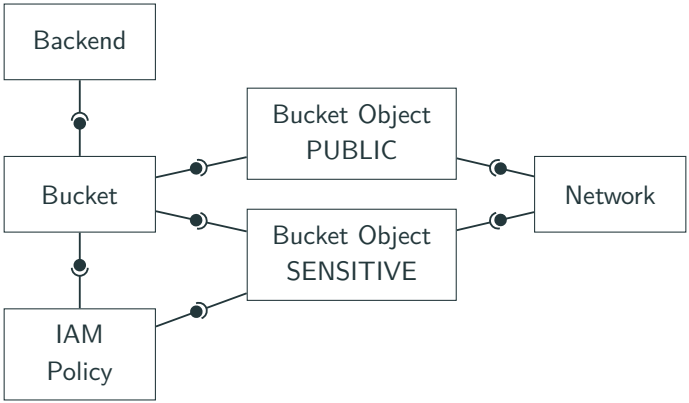
# Dans le Cloud: Infrastructure ⇒ APIs

Dans un contexte de IaaS, le paradigme du Cloud computing a transformé l'infrastructure en APIs externes (e.g., REST) sur lesquelles on peut envoyer des requêtes.

Aujourd'hui, dans le Cloud, les infrastructures sont comme n'importe quel logiciel offrant des services.

- Une requête pour un bucket pour stocker du contenu (e.g., BDD)
- Une requête pour un cluster GKE (Google Kubernetes Engine) pour acquerir des applciations micro-services
- Une requête pour des machines virtuelles, sur lesquelles on installera une application ou des logiciels
- etc.

# Exemple


Backend
Bucket
Bucket Object
PUBLIC
Bucket Object
SENSITIVE
IAM
Policy
Network

# Provisionner une infrastructure

**Définition**

Le provisionnement est l'opération de création et de gestion d'un ensemble de ressources dans une infrastructure dans un Cloud public, un data center, ou solution de monitoring hébergée.

Concrètement, le provisionnement c'est demander des machines et des ressources.

# Provisionner une infrastructure

**Définition**

Le provisionnement est l'opération de création et de gestion d'un ensemble de ressources dans une infrastructure dans un Cloud public, un data center, ou solution de monitoring hébergée.

Concrètement, le provisionnement c'est demander des machines et des ressources.

**Ressources**

Dans le provisionnement, TOUT est une ressource.

- Une machine virtuelle
- Un stockage (e.g., BDD, services de stockage, etc.)
- Un réseau
- Un secret manager
- etc.

# Provisionner une infrastructure

Les outils de provisionnement ont été fait pour rendre plus réutilisable, reproductible, flexible et sûr la gestion d'infrastructure dans le Cloud.

## Solution spécifique

- Heat pour OpenStack
- Cloud Formation (CFN) pour Amazon Web Service (AWS)
- Azure Resource Manager (ARM) pour Microsoft Azure
- GCP pour Google Cloud (déprécié, Google utilise maintenant Terraform)

## Solution multi-Cloud

- Terraform
- Pulumi

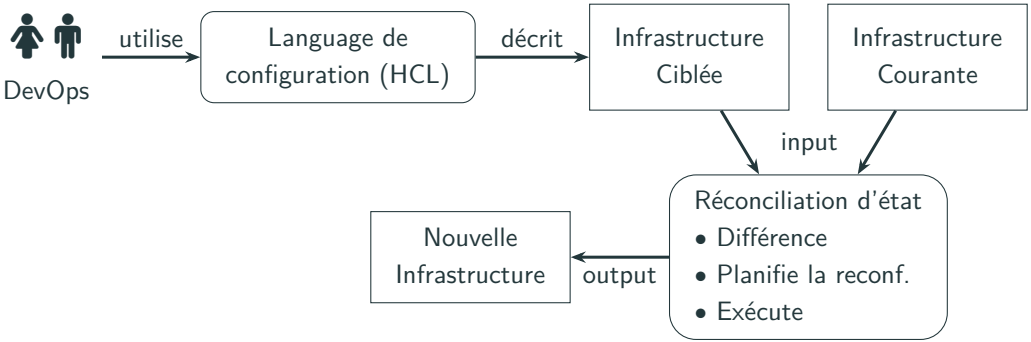
# Terraform

Dans ce module, nous utiliserons un outil de provisionnement: Terraform


Terraform est un outil

- Déclaratif: l'état souhaité est décrit dans un fichier, en utilisant le langage HCL (HashiCorp Configuration Language).
- Stateful: l'état courant de l'infrastructure est conservé dans un fichier.
- De reconciliation: l'état souhaité est comparé avec l'état courant pour définir quelle opérations (CRUD) doivent être réalisées. Ces actions sont ensuite exécutées.
- Conccurent: les opérations de réconcialtion peuvent être exécutée en parallel, tant que les dépendences sont respectées.

# Boucle de reconciliation



# Terraform

# Vue d'ensemble de Terraform

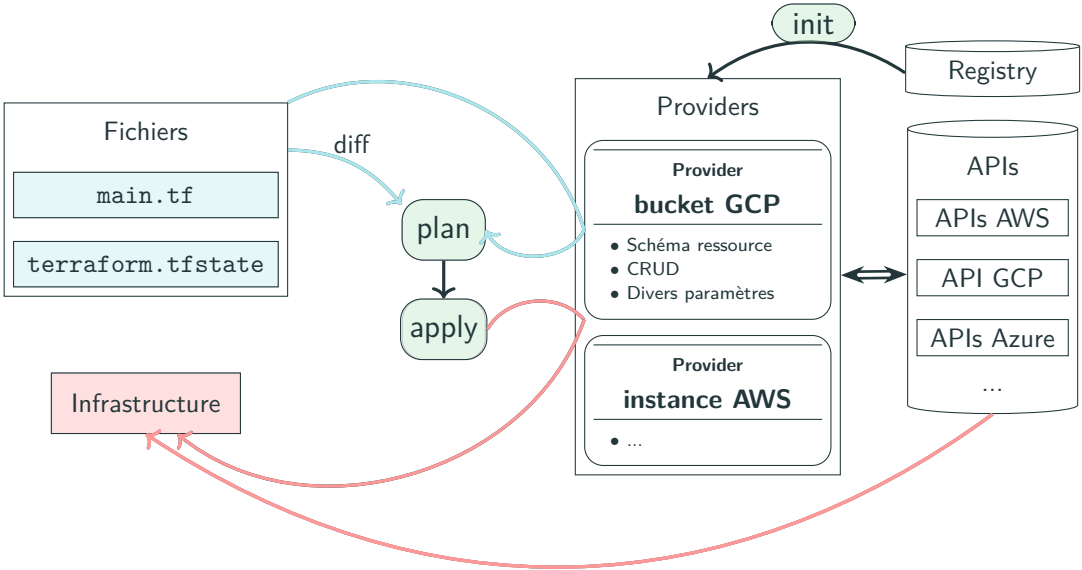
Les différents éléments de Terraform

- CLI (Command Line Interface)
- Le langage déclaratif HCL (HashiCorp Configuration Language)
- Les fichiers de configuration (`.tf`)
- Les providers
- Le state (fichier d'état)

## Les rôles de Terraform

- Provisionner des ressources (VM, réseaux, bases, etc.)
- Gérer le cycle de vie de l'infrastructure (create, update, delete)
- Maintenir l'infrastructure en cohérence avec le code
- Fournir un plan d'exécution avant application
- Supporter de multiples fournisseurs cloud

# Vue d'ensemble de Terraform



# CLI

```
terraform init
```

Initialise le repertoire de travail, et télécharge les providers.

# CLI

**`terraform init`**

Initialise le repertoire de travail, et télécharge les providers.

**`terraform plan`**

Produit un plan d'exécution pour reconcilier l'infrastructure avec le fichier de configuration (**.tf**). Le plan consiste en un ensemble de create/delete/update/replace.

# CLI

**`terraform init`**

Initialise le repertoire de travail, et télécharge les providers.

**`terraform plan`**

Produit un plan d'exécution pour reconcilier l'infrastructure avec le fichier de configuration (**.tf**). Le plan consiste en un ensemble de create/delete/update/replace.

**`terraform apply`**

Produit un plan et l'exécute.

# CLI

**`terraform init`**

Initialise le repertoire de travail, et télécharge les providers.

**`terraform plan`**

Produit un plan d'exécution pour reconcilier l'infrastructure avec le fichier de configuration (**.tf**). Le plan consiste en un ensemble de create/delete/update/replace.

**`terraform apply`**

Produit un plan et l'exécute.

**`terraform destroy`**

Supprime toutes les ressources.

# Ressources et Providers

## Ressource

- Une “**Resource**” est l’élément de base dans Terraform.
- Elle représente un objet de l’infrastructure (ex: VM, base de données, VPC).
- Elle est définie par :
  - un **nom** (identifiant local dans Terraform),
  - un **type** (ex: `aws_instance`, `google_storage_bucket`),
  - des **attributs/valeurs** (paramètres de configuration).

# Ressources et Providers

## Provider

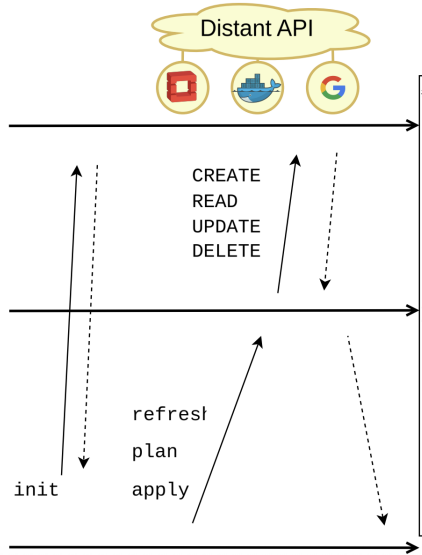
- Chaque resource est implémentée dans un **Provider** (fichier binaire écrit en Go).
- Le provider contient :
  - le **schéma** des ressources (attributs requis et optionnels; ainsi que leurs types),
  - les opérations **CRUD** (Create, Read, Update, Delete),
  - divers paramètres et règles de validation.
- Exemple : la resource `aws_instance` est définie dans le provider AWS.

# Terraform

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"
  }
}
```




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"
}
```

# Ecrire une configuration en HCL

# HCL: un langage de configuration déclaratif

**Définition en blocs**

En HCL, on définie des blocs, contenants des attributs sous la forme de clé-valeur. Un bloc peut lui même contenir un bloc. On parle alors de blocs imbriqués.

# HCL: un langage de configuration déclaratif

## Définition en blocs

En HCL, on définie des blocs, contenants des attributs sous la forme de clé-valeur. Un bloc peut lui même contenir un bloc. On parle alors de blocs imbriqués.

## Le bloc `terraform`

Définit les paramètres globaux du projet Terraform : version requise, configuration du backend pour le state, et gestion des providers.

# HCL: un langage de configuration déclaratif

## Définition en blocs

En HCL, on définie des blocs, contenants des attributs sous la forme de clé-valeur. Un bloc peut lui même contenir un bloc. On parle alors de blocs imbriqués.

## Le bloc `terraform`

Définit les paramètres globaux du projet Terraform : version requise, configuration du backend pour le state, et gestion des providers.

## Le bloc `provider`

Configure l'accès à un service cloud ou une API : informations de connexion, région, options spécifiques au fournisseur. Ce bloc est nommé.

# HCL: un langage de configuration déclaratif

### Définition en blocs

En HCL, on définie des blocs, contenants des attributs sous la forme de clé-valeur. Un bloc peut lui même contenir un bloc. On parle alors de blocs imbriqués.

### Le bloc `terraform`

Définit les paramètres globaux du projet Terraform : version requise, configuration du backend pour le state, et gestion des providers.

### Le bloc `provider`

Configure l'accès à un service cloud ou une API : informations de connexion, région, options spécifiques au fournisseur. Ce bloc est nommé.

### Le bloc `resource`

Décrit une ressource de l'infrastructure (VM, réseau, bucket...). Contient un type, un nom et des attributs configurables.

# HCL: un langage de configuration déclaratif

## Attributs / Arguments / Champs

Les attributs en HCL sont définis avec le signe égal (=).

La valeur des attributs peut être n'importe quelle expression: constante, appel de fonction, une liste, un objet, un référence, etc.

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

Les définitions multiples d'un attribut sont interdites. On ne fait qu'une seule attribution.

# HCL: un langage de configuration déclaratif

```
terraform {
    required_providers {
        google = {
            source  = "hashicorp/google"
            version = "5.6.0"
        }
    }
}
provider "google" {
    project = "mon_projet"
    region = "europe-west1"
    credentials = file(var.gcp_key)
}
resource "google_compute_instance" "vm" {
    name = "redis-${provider::google.project}"
    machine_type = "e2-medium"
    network_interface {
        network = "default"
    }
}
```

## Bloc `terraform`

- Le bloc **`terraform`** définit des paramètres globaux.
- Ici, on indique que le provider utilisé est `google`.
- Le provider est téléchargé depuis le registry `hashicorp`.
- La version requise est fixée à `5.6.0`.

# HCL: un langage de configuration déclaratif

```
terraform {
    required_providers {
        google = {
            source  = "hashicorp/google"
            version = "5.6.0"
        }
    }
}
provider "google" {
    project = "mon_projet"
    region = "europe-west1"
    credentials = file(var.gcp_key)
}
resource "google_compute_instance" "vm" {
    name = "redis-${provider::google.project}"
    machine_type = "e2-medium"
    network_interface {
        network = "default"
    }
}
```

**Bloc `provider`**

- `project = "mon_projet"` : identifiant du projet
- `region = "europe-west1"` : région utilisée pour déployer les ressources.
- `credentials = file(gcp_key.json)` : chemin vers la clé de service GCP

# HCL: un langage de configuration déclaratif

```
terraform {
    required_providers {
        google = {
            source  = "hashicorp/google"
            version = "5.6.0"
        }
    }
}
provider "google" {
    project = "mon_projet"
    region = "europe-west1"
    credentials = file(var.gcp_key)
}
resource "google_compute_instance" "vm" {
    name = "redis-${provider::google.project}"
    machine_type = "e2-medium"
    network_interface {
        network = "default"
    }
}
```

**Bloc `resource`**

- `resource "..."""vm"` : définit une machine virtuelle avec l'id `vm`.
- `name = "redis-${provider::google.project}"` : le nom de la VM sera construit à partir de redis- suivi du nom de projet dans le provider.
- `machine_type = "e2-medium"` spécifie le type de machine.
- `network_interface { network = "default"}` rattache l'instance au réseau par défaut.

# HCL: un langage de configuration déclaratif

Quelques autres blocs

**Le bloc `data`**

Permet de récupérer des informations existantes depuis un provider. Le contenu est en lecture-seule.

**Le bloc `module`**

Regroupe plusieurs ressources et variables dans une unité réutilisable. Facilite la factorisation et la réutilisation du code Terraform. Un fichier **.tf** est lui même un module.

**Le bloc `check`**

Permet de définir des conditions de validation dans la configuration. Terraform échoue si la condition n'est pas respectée ⇒ utile pour garantir des contraintes métiers.

**Le bloc `import`**

Relie une ressource réelle déjà existante dans le cloud à une ressource définie dans le code Terraform. Important : permet d'intégrer de l'infra existante sans la recréer.

# HCL: un langage de configuration déclaratif

## Variables

Les modules peuvent avoir trois types de variables:

- Inputs
- Outputs
- Locals

Ces variables sont définies avec les blocs: `variable`, `output` et `locals`.

# HCL: un langage de configuration déclaratif

## Références

- Les attributs d'une ressource sont référencées par le type et le nom de la ressource, suivi du nom de la ressource. Par exemple: `docker_image.redis.image_id`
- Pour référencer depuis un bloc **`data`**, on utilise le mot clé **`data`**. Par exemple: **`data`**`.docker_image.redis.image_id`
- Pour les variables de type input et local, on utilse les mots clés `var` et `local`. Par exemple:
  `var.my_input_var`
  `local.my_local_var`

# HCL: un langage de configuration déclaratif

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

provider "google" {
  project = "my-gcp-project-id"
  region  = "europe-west1"
  credentials = file(gcp_key.json)
}

data "google_compute_image" "debian" {
  family  = "debian-11"
  project = "debian-cloud"
}

locals {
  prefix = "app"
}
```

```
resource "google_compute_instance" "redis" {
  name = "redis-01"
  machine_type = "e2-micro"
  boot_disk {
    initialize_params {
      image = data.google_compute_image.debian.
          self_link
    }
  }
  network_interface {
    network = "default"
    access_config {}
  }
}
resource "google_compute_instance" "app" {
  name = "${local.prefix}-${
      google_compute_instance.redis.name}"
  machine_type = "e2-medium"
  boot_disk {
    initialize_params {
      image = data.google_compute_image.debian.
          self_link
    }
```

# Quelques bonnes pratiques

# Convention de nommage

Un code d'IaC est aussi source de documentation.

**Bonne pratique 1**

- Donner toujours une description de la ressource au sein de son nom
  - Environnement
  - Type de la ressource
  - But, usage, intention
  - Exemple: `dev-firewall-rule-allow-helloworld-to-database`

# Variables et constantes

**Bonne pratique 2**

- De manière générale: Privilégiez les constantes plutôt que les variables
- Si le changement d'une valeur affecte les dépendances de l'infrastructure ou compromets des informations sensibles, utilisez une variable

# Avoid troubleshooting

**Bonne pratique 3**

- Bien lire et comprendre chaque déclarations et le plan avant de faire un `terraform apply`
- Versionner le code Terraform, sans commit de secrets
- Intégrer du CI/CD sur votre infrastructure Terraform
- Stocker vos fichiers d'état (.tfstate) sur des stockages distants avec des mecanismes de lock

# Démonstration

# Plan

Conteneurisation avec Docker

Provisionnement avec Terraform

(Go dans l'Infrastructure-as-Code)

Installation et configuration avec Ansible

Orchestration avec Kubernetes

"Quality Assurance in Infrastructure as Code: Issues, Approaches, and Open Challenges"

# Go lang

- Créé chez **Google** en **2007**.
- Rendu public en **2009**.
- Objectif : créer un langage simple, rapide à compiler et adapté à tous les systèmes logiciels.
- Pensé pour le réseau, la concurrence et les services distribués.

**Quel rapport avec l'IaC ?**

Go est souvent utilisé pour construire les moteurs, plugins, providers et controleurs dans les outils d'infrastructure-as-code.

# Où trouve-t-on Go dans l'IaC ?

- **Terraform** : providers écrits en Go.
- **OpenTofu** : utilise aussi le modèle de providers compatibles Terraform.
- **Pulumi** : permet d'écrire de l'IaC en Go.
- **Crossplane** : providers et contrôleurs Kubernetes majoritairement écrits en Go.
- **Kubernetes** : API, contrôleurs et opérateurs reposent fortement sur Go.

**Go pour le cloud :** peu visible côté utilisateur final, mais partout dans l'ingenierie du Cloud.

# Rappel sur les providers dans Terraform

Configuration Terraform (fichier `.tf` en HCL)

⇓

Terraform Core (en Go)

⇓

Provider (en Go)

⇓

API distante (par ex. REST)

- Terraform Core lit le plan.
- Le provider connaît une API spécifique : AWS, GitHub, Docker, Proxmox, etc.
- Le provider expose des **resources** et des **data sources**.
- Terraform appelle le provider via un protocole de plugin.

# Anatomie d'un provider Terraform

Un provider Terraform contient généralement :

- une configuration globale : endpoint, token, région ;
- des **resources** : objets créés, modifiés ou supprimés ;
- des **data sources** : objets lus depuis une API ;
- un client API partagé ;
- des tests unitaires et des tests d'acceptance.
- etc.

# Terraform Plugin Framework

HashiCorp recommande le **Terraform Plugin Framework** pour construire de nouveaux providers.

- Il fournit les interfaces Go attendues par Terraform.
- Il gère le dialogue entre Terraform Core et le provider.
- Il structure le code autour de providers, resources, data sources et schemas.

# On écrit quoi en Go ?

Un **Schéma** pour un provider ou une ressource avec son CRUD.

- **Create** : créer l'objet distant ;
- **Read** : relire l'objet et synchroniser l'état ;
- **Update** : modifier l'objet distant ;
- **Delete** : supprimer l'objet distant.

# Un provider en Go

Un **Schéma** pour un provider.

```
package provider

import (...)

type myProviderModel struct {...}

func (p *myProvider) Schema(
    ctx context.Context,
    req provider.SchemaRequest,
    resp *provider.SchemaResponse,
) {...}
```

# Le schema pour un provider décrit sa configuration globale

```
type myProviderModel struct {
    Endpoint types.String `tfsdk:"endpoint"`
    Token    types.String `tfsdk:"token"`
}

func (p *myProvider) Schema(ctx context.Context,
                            req provider.SchemaRequest,
                            resp *provider.SchemaResponse,) {
    resp.Schema = schema.Schema{
        Attributes: map[string]schema.Attribute{
            "endpoint": schema.StringAttribute{
                Required: true,
            },
            "token": schema.StringAttribute{
                Required:  true,
                Sensitive: true,
            },
        },
    }
}
```

# Une ressource en Go

Un **Schéma** pour une ressource avec son CRUD.

- **Create** : créer l'objet distant ;
- **Read** : relire l'objet et synchroniser l'état ;
- **Update** : modifier l'objet distant ;
- **Delete** : supprimer l'objet distant.

# Une ressource en Go

```go
package provider

import (...)

type myResourceModel struct {...}

func (r *myResource) Schema(
    ctx context.Context,
    req resource.SchemaRequest,
    resp *resource.SchemaResponse,
) { ... }

func (r *myResource) Create(...) { ... }
func (r *myResource) Read(...)   { ... }
func (r *myResource) Update(...) { ... }
func (r *myResource) Delete(...) { ... }
```

# Le schema pour une ressource décrit les attributs manipulés par Terraform

```
type myResourceModel struct {
    ID   types.String `tfsdk:"id"`
    Name types.String `tfsdk:"name"`
}

func (r *myResource) Schema(ctx context.Context,
                            req resource.SchemaRequest,
                            resp *resource.SchemaResponse,) {
    resp.Schema = schema.Schema{
        Attributes: map[string]schema.Attribute{
            "id": schema.StringAttribute{
                Computed: true,
            },
            "name": schema.StringAttribute{
                Required: true,
            },
        },
    }
}
```

# CRUD pour une ressource

Une ressource implémente son **cycle de vie** :

```
func (r *myResource) Create(
    ctx context.Context,
    req resource.CreateRequest,
    resp *resource.CreateResponse
        ,
) { ... }
```

```
func (r *myResource) Update(
    ctx context.Context,
    req resource.UpdateRequest,
    resp *resource.UpdateResponse
        ,
) { ... }
```

```
func (r *myResource) Read(
    ctx context.Context,
    req resource.ReadRequest,
    resp *resource.ReadResponse,
) { ... }
```

```
func (r *myResource) Delete(
    ctx context.Context,
    req resource.DeleteRequest,
    resp *resource.DeleteResponse
        ,
) { ... }
```

# CRUD pour une ressource

```
func (r *myResource) Create(...)
func (r *myResource) Read(...)
func (r *myResource) Update(...)
func (r *myResource) Delete(...)
```

- lire le **plan** ou le **state** Terraform ;
- convertir les données vers un modèle Go ;
- appeler l'**API distante** ;
- convertir la réponse de l'API ;
- mettre à jour le **state Terraform**.

# Plan

Conteneurisation avec Docker

Provisionnement avec Terraform

(Go dans l'Infrastructure-as-Code)

Installation et configuration avec Ansible

Orchestration avec Kubernetes

"Quality Assurance in Infrastructure as Code: Issues, Approaches, and Open Challenges"

# Les rôles des outils d'IaC

**Virtualization et Conteneurisation**

Customiser, configurer
tester l'application
et la conteneuriser

**Provisionnement d'infrastructure**

Demander ressources
physiques ou virtuelles;
configurer le réseau;
et règles sécurité

**Installation et Configuration**

Installer les services
(app + deps)
configurer les services;
et les intégrer

**Orchestration de cycle de vie**

Upgrades auto;
Backup et recovery;
Surveillance;
Passage à l'échelle

`https://docs.ansible.com/`

# Secure Shell (SSH)

# Rappel

**Protocole**

SSH (*Secure Shell*) est un protocole réseau chiffré permettant une connexion sécurisée entre un client et un serveur. Il repose généralement sur TCP et utilise le port **22** par défaut.

**Utilisation courante**

Connexion distante à un serveur pour exécuter des commandes, transfert de fichiers (`scp`, `sftp`), tunneling de ports, etc.

```
ssh utilisateur@serveur.example.com
```

# Démarrer une application

Dans un terminal :

```
> ssh user@mon-serveur
> sudo apt update && sudo apt install -y nginx
> nohup nginx -g "daemon off;" &
```

Cette suite de commandes :

- se connecte en SSH,
- installe Nginx avec privilèges sudo,
- lance Nginx en arrière-plan grâce à `nohup` et `&`.

# Multi-SSH

Pour communiquer avec plusieurs machines en parallèle :

- Utiliser `ssh` dans des boucles shell (for/parallel),
- Outils spécialisés comme `pssh` ou `parallel-ssh` (ou Ansible lui-même).

**Gestion de la concurrence**

Attention ! Il faut gérer l'ordre d'exécution, la tolérance aux pannes et le parallélisme afin d'éviter les conflits d'accès ou la surcharge réseau.

# Le configuration management, c'est quoi ?

# Configuration management

**Définition**

La **gestion de configuration** (*Configuration Management*) est la pratique qui consiste à automatiser l'installation, la configuration, la mise à jour et la maintenance d'applications ou de systèmes.

# Configuration management

**Définition**

La **gestion de configuration** (*Configuration Management*) est la pratique qui consiste à automatiser l'installation, la configuration, la mise à jour et la maintenance d'applications ou de systèmes.

**Multitude de stratégies**

Chaque solution de configuration management adopte son propre paradigme en composant avec différents stratégies. Il n'y en a pas une meilleure que l'autre, juste une plus adaptée en fonction de votre cas d'utilisation.

# Différentes approches

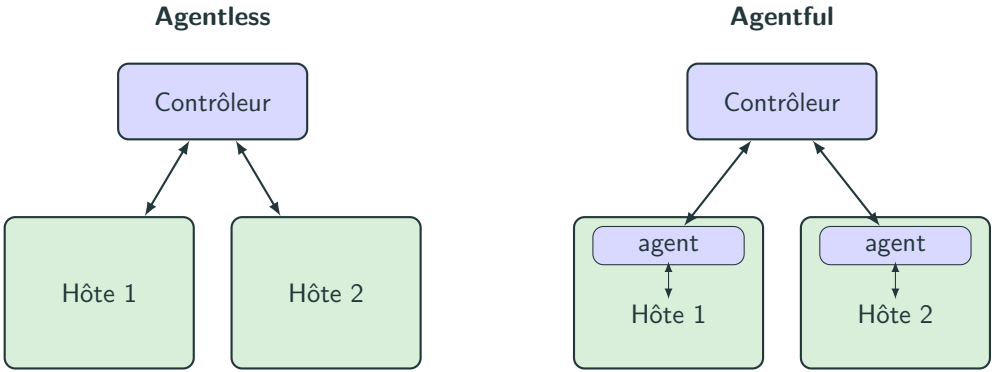
## Pull- ou Push- based

- **Pull** : les nœuds gérés récupèrent régulièrement leur configuration depuis un serveur central;
- **Push** : un contrôleur envoie directement la configuration vers les nœuds.

# Différentes approches

## Agentless vs. Agentful

- **Agentful** : chaque machine gérée exécute un agent dédié, installé sur la machine cible;
- **Agentless** : aucune installation côté cible, on utilise des protocoles standards (ex: SSH).



# Différentes approches

**Static vs. Event-driven**

- **Static** : la configuration est appliquée de manière periodique ou à la demande;
- **Event-driven** : les changements sont déclenchés par des événements (alertes, hooks, messages).

# Configurer, démarrer, et maintenir son application

## Pull-based et agent-driven solutions

- Chef (à la demande + periodic)
- Puppet (periodic)
- CFEngine (periodic)

## Event-driven

- SaltStack (à la demande + periodic + event)
- Juju (à la demande + periodic + event)

## Push-based et agentless solution

- Ansible (à la demande)

# Ansible

Dans ce module, nous utiliserons un outil de configuration management: Ansible.


Ansible est un outil

- Semi-déclaratif : on décrit l'état souhaité avec des **Tasks** et **Playbooks**, tout en pouvant créer ses propres modules si besoin.
- Stateless: l'état courant de l'infrastructure et des applications déployées n'est pas conservé entre deux exécutions.
- Conccurrent : exécute des tâches en parallèle sur plusieurs hôtes via SSH.

# Ansible

# Vue d'ensemble d'Ansible

Les différents éléments de Ansible

- **CLI** (`ansible`, `ansible-playbook`, etc.)
- **Inventaire** des hôtes (fichier INI ou YAML)
- Langage **YAML** pour décrire les tâches
- **Tasks** : appels à des actions atomiques (ex: apt, copy, service)
- **Playbooks** : regroupements ordonnés de tasks

# Vue d'ensemble d'Ansible

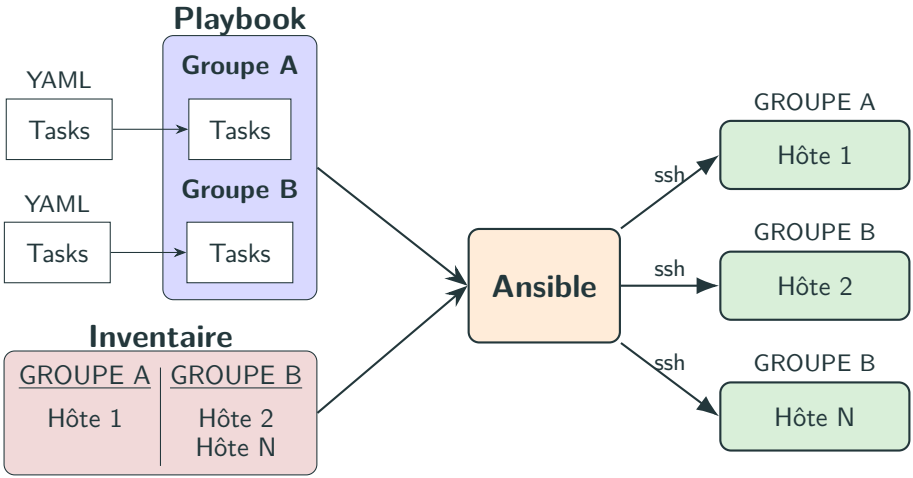
Les différents éléments de Ansible

- **CLI** (`ansible`, `ansible-playbook`, etc.)
- **Inventaire** des hôtes (fichier INI ou YAML)
- Langage **YAML** pour décrire les tâches
- **Tasks** : appels à des actions atomiques (ex: apt, copy, service)
- **Playbooks** : regroupements ordonnés de tasks

## Les rôles d'Ansible

- Définir un ensemble de tâches à éxécuter sur des hôtes distants
  - Commandes à éxécuter
  - Installation de dépendances
  - Installation d'application
  - etc.
- Grouper les hôtes par rôles pour limiter les répétitions
- Assurer la gestion de la concurrence dans le processus de reconfiguration

# Vue d'ensemble d'Ansible



# Inventaire

L'inventaire définit les machines cibles et leurs groupes. Il peut être au format INI ou YAML.

```
# inventory.ini
[web]
web1.example.com
web2.example.com

[db]
db1.example.com ansible_user=admin
```

Note : On peut aussi utiliser des inventaires dynamiques (scripts, plugins cloud).

# Tasks et Playbooks

- **Task** : unité de travail, appel d'une instruction (appelée module) avec des paramètres.
- **Play** : ensemble de tasks, souvent stocké dans un fichier YAML.
- **Playbook** : fichier YAML qui regroupe des plays. Chaque play associe des hôtes, des variables et une liste de tasks.

**Note**: Il existe un registre de rôles et de tâches: `https://galaxy.ansible.com/`

# Exemple de Tasks

Exemple de Play:

```yaml
# taches.yaml
- name: Installer curl
  become: yes          # sudo
  apt:                 # module apt pour Debian
    name: curl         # package a installer
    state: present     # doit etre present

- name: Creer un fichier de test
  file:                # module pour creer un fichier
    path: /tmp/hello.txt   # chemin du fichier
    state: touch           # creer vide
    # content: "Hello"     # remplace le contenu
    # line: "Hello"        # ajoute le contenu
```

# Exemple de Playbook

A partir du fichier `taches.yaml`, je peux créer un playbook:

```yaml
# site.yaml
- hosts: web
  become: yes # sudo
  tasks:
    # on appelle les taches du fichier externe
    - include_tasks: taches.yaml
    - name: Ecrire un message dans le fichier
      copy:
        content: "Bonjour depuis Ansible !\n"
        dest: /tmp/hello.txt
- hosts: db
  ...
```

# Quelques modules

# Module apt

Options principales :

- `name` : nom du paquet
- `state` : état désiré (`present`, `absent`, `latest`)
- `update_cache` : mettre à jour l'index (`yes`/`no`)

```
- name: Installer nginx
  apt:
    name: nginx
    state: present
    update_cache: yes
```

# Module git

Options principales :

- `repo` : URL du dépôt
- `dest` : répertoire de destination
- `version` : branche/tag/commit
- `update` : mettre à jour si déjà cloné

```
- name: Cloner un depot
  git:
    repo: "https://github.com/ansible/ansible-examples.git"
    dest: /opt/ansible-examples
    version: master
    update: yes
```

# Module shell

Options principales :

- `cmd` : commande à exécuter
- `chdir` : répertoire d'exécution
- `creates` : exécuter seulement si fichier n'existe pas
- `removes` : exécuter seulement si fichier existe

```
- name: Lancer un script shell
  shell: ./install.sh
  args:
    chdir: /opt/scripts
```

# Module copy

Options principales :

- `src` : chemin local (copié depuis la machine de contrôle)
- `dest` : chemin de destination
- `content` : contenu direct à écrire
- `owner`, `group`, `mode` : permissions

```
- name: Creer un fichier avec contenu
  copy:
    dest: /tmp/hello.txt
    content: "Bonjour Ansible!\n"
    owner: root
    group: root
    mode: '0644'
```

# Module file

Options principales :

- `path` : chemin du fichier ou dossier
- `state` : file, directory, absent, touch, link
- `owner`, `group`, `mode`

```
- name: Creer un repertoire
  file:
    path: /opt/app
    state: directory
    mode: '0755'
```

# Module lineinfile

Options principales :

- `path` : chemin du fichier
- `line` : ligne à ajouter
- `regexp` : expression régulière pour remplacer
- `insertafter`, `insertbefore` : position

```
- name: Ajouter une ligne dans /etc/hosts
  lineinfile:
    path: /etc/hosts
    line: "192.168.1.10 myapp.local"
```

# CLI

**`ansible`**

Exécuter une commande ad-hoc sur un ou plusieurs hôtes

```
> ansible all -m ping
```

# CLI

**`ansible`**

Exécuter une commande ad-hoc sur un ou plusieurs hôtes

```
> ansible all -m ping
```

**`ansible-playbook`**

Lancer un Playbook YAML complet

```
> ansible-playbook site.yaml
```

# CLI

**ansible**

Exécuter une commande ad-hoc sur un ou plusieurs hôtes

```
> ansible all -m ping
```

**ansible-playbook**

Lancer un Playbook YAML complet

```
> ansible-playbook site.yaml
```

**ansible-inventory**

Lister ou valider l'inventaire

```
> ansible-inventory -list -y
```

# Structure d'un projet

Exemple d'un projet très simple:

```
ansible-project/
├── inventory.ini
├── playbook.yaml
└── taches.yaml
```

# Démonstration

# Plan

# Les rôles des outils d'IaC

**Virtualization et Conteneurisation**

Customiser, configurer tester l'application et la conteneuriser

**Provisionnement d'infrastructure**

Demander ressources physiques ou virtuelles; configurer le réseau; et règles sécurité

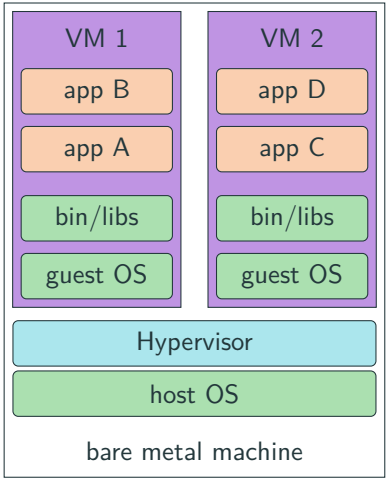
**Installation et Configuration**

Installer les services (app + deps) configurer les services; et les intégrer

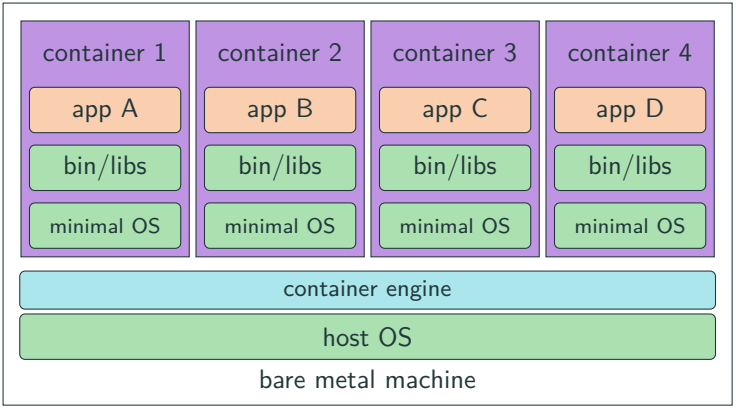
**Orchestration de cycle de vie**

Upgrades auto; Backup et recovery; Surveillance; Passage à l'échelle

# Rappel - Conteneurs




VMs avec un hyperviseur




Containers

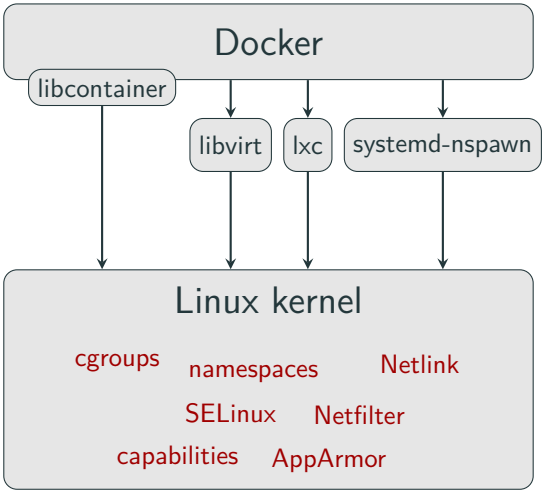
# Rappel - Conteneurs

## Tirer parti des fonctionnalités du kernel

- `Cgroups`: qu'est ce que mon processus peut consommer ?
- `Namespaces`: qu'est ce que mon processus peut voir ? Et donc faire ?

## Et d'autres choses ...

- `Capabilities`: gestion fine des permissions
- `SELinux/AppArmor`: tout contrôler
- `Netlink/Netfilter`: communication breakdown



# Kubernetes

## A brief history

- **2014** : Annonce initiale du projet par Google (développé en Go)
- **2017** : Devient le standard *de facto* face à ses concurrents (Docker Swarm, Apache Mesos)
- **2018** : Premier projet de la CNCF (Cloud Native Computing Foundation) à obtenir le statut de maturité maximale (“Graduated”)
- **Aujourd’hui** : Adopté massivement et proposé en service managé par tous les géants du Cloud (AWS, Azure, GCP)

# Vue d'ensemble de Kubernetes

Les différents éléments de Kubernetes

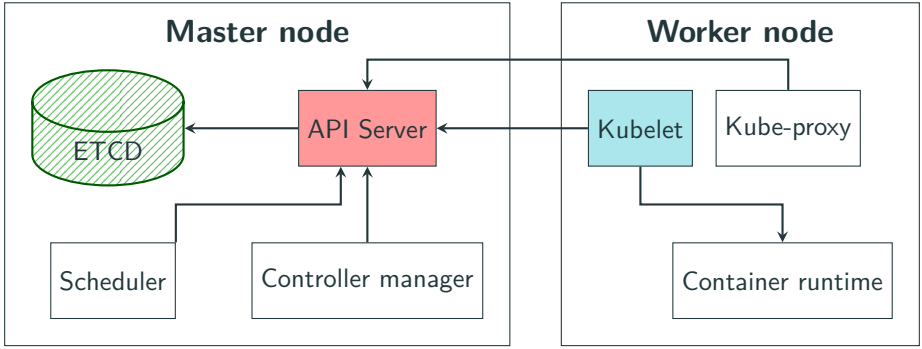
- kubectl (Command Line Interface)
- Le plan de contrôle (Control Plane)
- Les noeuds (Nodes)
- Les pods

**Les rôles du Control Plane**

- Planifier (scheduling) les pods sur les noeuds
- Gérer l'état désiré (desired state)
- Superviser (monitoring) les déploiements
- Gérer le réseau et les services
- Assurer la scalabilité et la haute disponibilité
- etc.

# Vue d'ensemble de Kubernetes

Un cluster Kubernetes est composé de plusieurs noeuds: un noeud **master**, en charge du Control plane et des noeuds **workers**.



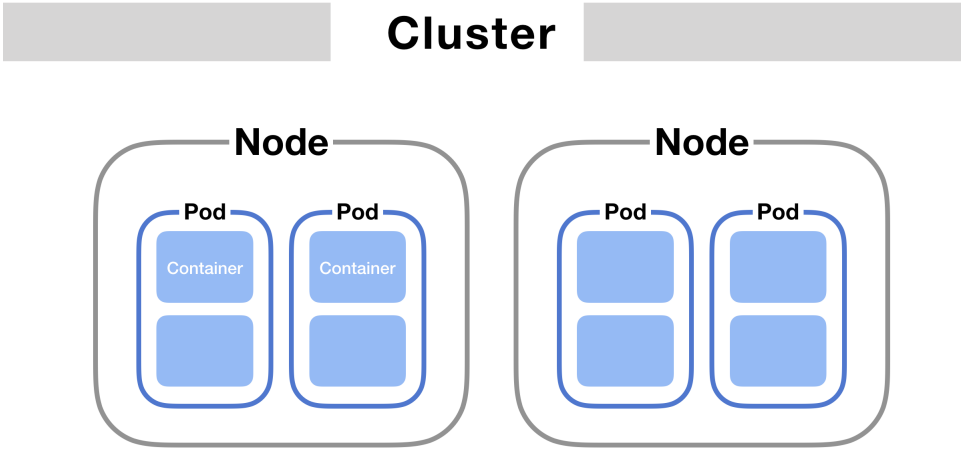
# Différents types de noeuds

## Master node - Control plane

- Kunernetes API server: Un utilisateur peut communiquer avec cette API via **kubectl**. Les services internes peuvent également communiquer avec cette API.
- Scheduler: planifie les applications sur les workers
- Controller manager: controle toutes les ressources d'un cluster Kubernetes (y compris les ressources intégrées telles que les pods)
- ETCD (`etc/` distribué): système de stockage de données distribué et fiable utilisé par Kubernetes pour stocker l'état des ressources et la configuration du cluster.

## Worker nodes

- Container runtime: Docker ou un autre
- Kubelet: communique avec l'API server et gère les conteneurs sur le noeud
- Kube-proxy: répartit la charge du trafic réseau entre les composants de l'application


Cluster
Node
Pod
Container
Pod
Container
Node
Pod
Pod

# Les pods dans Kubernetes

## La plus petite unité de Kubernetes

- Un Pod contient **un ou plusieurs conteneurs** (comme Docker) fortement couplés.
- Tous les conteneurs d'un même Pod partagent la même adresse IP (ils se parlent sur *localhost*) et le même stockage.
- Les conteneurs d'un Pod sont toujours planifiés et exécutés ensemble sur le même serveur (*Worker node*).
- Les pods sont ephemeres. S'il plante ou si le serveur tombe en panne, Kubernetes le détruit et en recrée un nouveau (nouvelle IP).

# Dimensionner un Pod

Pour éviter qu'un Pod ne consomme toutes les ressources d'une machine au détriment des autres, on configure deux seuils (CPU et Mémoire) :

### *Requests* (Le minimum garanti)

- C'est ce dont le Pod a **strictement besoin** pour fonctionner.
- Le *Scheduler* utilise cette valeur pour trouver un Noeud ayant assez de place disponible pour l'accueillir.

### *Limits* (Le plafond maximum)

- C'est la limite **à ne pas dépasser**.
- Protège le Noeud et les autres Pods contre une application qui s'emballe (ex: fuite de mémoire).

# Dimensionner un Pod

## Que se passe-t-il si un Pod dépasse ses limites ?

- **Pour le CPU** : Le conteneur est "bridé" (*Throttling*). L'application ralentit, mais elle ne plante pas.
- **Pour la Mémoire (RAM)** : Le conteneur est tué et redémarré brutalement (**Erreur OOMKilled** - *Out Of Memory*).

## Les règles d'or

- **Ne jamais deviner** : Mesurez d'abord la consommation réelle de l'application en charge avant de fixer vos valeurs.
- **Toujours définir des limites** : Un Pod sans limite est un danger pour l'ensemble du cluster Kubernetes.
- **Unités** : Le CPU se mesure en *millicores* (ex: 500m = 0.5 CPU) et la RAM en Mébioctets/Gibioctets (ex: 256Mi, 1Gi).

# Les manifestes

# Manifeste Kubernetes

## Infrastructure as Code (IaC)

Un manifeste est un fichier YAML qui décrit l'**état désiré** de votre application. Vous donnez ce fichier à Kubernetes, et il se débrouille pour que la réalité corresponde à ce fichier.

## Les 4 piliers obligatoires

Tout manifeste Kubernetes valide (sans exception) possède ces 4 champs principaux à sa racine :

1. `apiVersion` : La version de l'API Kubernetes à utiliser.
2. `kind` : Le type d'objet que l'on veut créer (Pod, Service, etc.).
3. `metadata` : Les données pour identifier l'objet.
4. `spec` : Les spécifications techniques (le cœur du sujet).

# Étape 1 : Quoi et Qui ?

**L'en-tête (apiVersion et kind)**

On commence par dire à Kubernetes à quelle partie de son cerveau s'adresser, et quel type d'objet on souhaite construire.

```yaml
apiVersion: v1
kind: Pod
```

**L'identité (metadata)**

Ensuite, on donne un nom unique à notre objet pour pouvoir le retrouver plus tard. On peut aussi lui coller des étiquettes (*labels*).

```yaml
metadata:
  name: mon-premier-pod
  labels:
    environnement: dev
```

# Étape 2 : Le comportement (Spec)

**Le bloc `spec`**

C'est ici que l'on décrit ce que notre ressource doit faire concrètement. Le contenu de ce bloc change complètement selon le `kind` choisi.

**Exemple pour un Pod**

Pour un Pod, la seule chose vraiment obligatoire est de fournir la liste des conteneurs à lancer (souvent un seul), avec leur nom et l'image Docker à utiliser.

```yaml
spec:
  containers:
  - name: mon-conteneur-web
    image: nginx:latest
    ports:
    - containerPort: 80
```

# Étape 2 : Le comportement (Spec)

**Le bloc `spec`**

C'est ici que l'on décrit ce que notre ressource doit faire concrètement. Le contenu de ce bloc change complètement selon le `kind` choisi.

**Exemple pour un Pod**

Pour un Pod, la seule chose vraiment obligatoire est de fournir la liste des conteneurs à lancer (souvent un seul), avec leur nom et l'image Docker à utiliser.

```yaml
spec:
  containers:
  - name: mon-conteneur-web
    image: nginx:latest
    ports:
    - containerPort: 80
```

# Le manifeste final complet

**Le puzzle assemblé (`pod.yaml`)**

En collant nos 3 morceaux en respectant l'indentation de 2 espaces, nous obtenons notre manifeste finalisé :

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: mon-premier-pod
  labels:
    environnement: dev
spec:
  containers:
  - name: mon-conteneur-web
    image: nginx:latest
    ports:
    - containerPort: 80
```

# kubectl (CLI de Kubernetes)

# Une commande pour les gouverner toutes : kubectl

`kubectl [ACTION] [RESSOURCE] [NOM] [OPTIONS]`

- **[ACTION]** : `get`, `describe`, `create`, `delete`, `apply`
- **[RESSOURCE]** : Le nom de l'objet ciblé. (`pods`, `services`, `deployments`, `nodes`...)
- **[NOM]** : L'identifiant précis (optionnel, sinon l'action s'applique à tous les objets du même type).
- **[OPTIONS]** : (ex: `-n` pour cibler un namespace, `-o yaml` pour changer le format de sortie).

# kubectl

## Observer et comprendre

- `$ kubectl get pods` : Liste les pods avec leur statut résumé (e.g., *Running*).
- `$ kubectl describe pod <nom>` : Affiche l'historique complet d'un pod. Utile pour lire la section Events et debugger
- `$ kubectl logs <nom>` : Affiche les logs générés par l'application à l'intérieur du conteneur.
- `$ kubectl exec -it <nom> - /bin/sh` : Ouvre un terminal à l'interieur du pod (l'équivalent d'une connexion SSH).

## Run - approche imperative

- `$ kubectl run mon-serveur -image=nginx`: Ordre direct à Kubernetes

## Apply - approche déclarative

- `$ kubectl apply -f application.yaml`: Reconcilier avec le manifeste YAML

# Plan

Conteneurisation avec Docker

Provisionnement avec Terraform

(Go dans l'Infrastructure-as-Code)

Installation et configuration avec Ansible

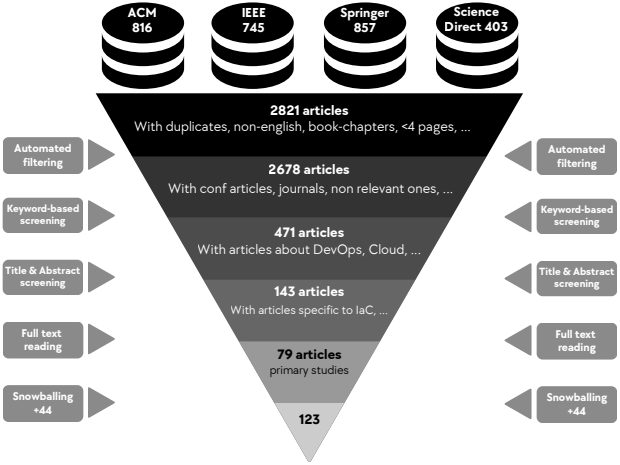
Orchestration avec Kubernetes

"Quality Assurance in Infrastructure as Code: Issues, Approaches, and Open Challenges"

On a fait une Systematic Litterature Review (SLR) autour de l'IaC

# Une communauté scientifique active

But de la SLR: analyser les productions / recherche pour améliorer la qualité de l'IaC.



# Une activité en forte croissance

La SLR observe une augmentation globale du nombre de publications sur la qualité de l'IaC.

- 2015 : **1** étude.
- 2020 : **14** études.
- 2023 : **20** études.
- 2024 : **31** études.
- 2025 : **20** études, malgré une année encore incomplète au moment de la publication.

La tendance montre que l'IaC devient un objet central pour la qualité logicielle et DevOps.

# Quels problèmes sont étudiés ?

Les travaux se concentrent surtout sur les problèmes qui peuvent casser ou fragiliser l'infrastructure.

- **Sécurité** : secrets exposés, permissions faibles, mauvaises configurations réseau.
- **Opérationnel** : idempotence, ordre de déploiement, état incohérent.
- **Structurel** : modularité, architecture, organisation du code.
- **Syntaxique** : conventions, erreurs de langage, mauvais usages.

**Observation**

La recherche est encore très défensive : elle cherche surtout à éviter les pannes, les failles et les erreurs de configuration.

# Comment la qualité est-elle vérifiée ?

La majorité des approches proposées sont des outils d'analyse.

- **Règles et patterns** : détecter des erreurs connues.
- **Graphes** : analyser les dépendances entre ressources.
- **Machine Learning / LLMs** : apprendre à repérer des anomalies.
- **Méthodes formelles** : vérifier des propriétés avec des contraintes.
- **Tests et exécution** : valider le comportement réel de l'infrastructure.

**Aujourd'hui :** beaucoup de statique, peu de dynamique.

# Ce qui reste ouvert

La SLR identifie plusieurs directions de recherche encore peu explorées.

- Passer de la **sécurité** à la **performance**, au coût et à la durabilité.
- Construire des approches **agnostiques aux outils**.
- Aller au-delà de l'analyse statique avec des approches **dynamiques** ou **hybrides**.
- Étudier l'IaC au niveau **système**, pas seulement fichier par fichier.
- Relier les scripts IaC aux logs, déploiements, incidents et dérives de configuration.

# Ressources supplémentaires


1. H. El Hayani et al. **“Quality assurance in infrastructure as code: Issues, approaches, and open challenges”.** In: Journal of Systems and Software (2026)
2. R. Wang. **“Infrastructure as Code, Patterns and Practices”.** In: Manning, O’Reilly (2022)
3. H. El Hayani, J. Philippe, and S. Challita. **“Poliac: Production Observability for LLM-based Infrastructure as Code”.** In: ASEW ’26. 2026
4. O. Proust et al. **“Scopeannon: A Static Analysis of Ansible’s Scopes Ambiguities”.** In: ASEW ’26. 2026
5. J. Philippe et al. **“Fast Choreography of Cross-DevOps Reconfiguration with Ballet: A Multi-Site OpenStack Case Study”.** In: SANER’24. 2024