Cloud Computing : Infrastructure et Opérations

Hugo Blanc

Université Lyon 1

Outline

$ whoami

Hugo Blanc

$ whoami

Hugo Blanc

→ Platform Security Engineer @ Doctolib

$ whoami

Hugo Blanc

→ Platform Security Engineer @ Doctolib

→ Enseignant @ UCBL depuis 2022

$ whoami

Hugo Blanc

→ Platform Security Engineer @ Doctolib

→ Enseignant @ UCBL depuis 2022

Adepte du tutoiement :)

$ man platsec

- Linux and containerized workloads hardening
- Networking security, detection automation
- Kubernetes & Cloud security
- Blue team & forensics
- Incident management & response

$ history

- Site Reliability Engineer (aka Cloud sysadmin) @ Virtuo
- DevOps & Security @ DevOps.Works
- Étudiant @ LP ESSIR :)

En cas de questions sur les cours (ou Linux/infosec/cloud en général), ne pas hésiter:
hugo.blanc@univ-lyon1.fr

Best effort pour les réponses :)

Où trouver les cours ?

Slides:

⟹ https://syscall.cafe/t/

Évaluation

Ce cours sera évalué par :

- Un DS sur table de 2h en fin de cours

LLM & co.

- L'utilisation des LLMs (et autres outils) est **déconseillée** car ils ne participent pas à la réflexion et à l'apprentissage.
- Pas de sanctions si usage « modéré » des LLMs **pour les TP notés seulement**. Des questions orales de validation des acquis peuvent être posées lors de la restitution.
- Tout aide durant les contrôles sur table sera considérée comme de la triche, et mènera à une note de zéro.

Sommaire

Pré-requis

- Connaître les bases de l'utilisation d'un terminal (bash/zsh)
- Comprendre les concepts de base du networking (IP, port, protocole)
- Avoir Docker installé (pour LocalStack)

Outline

Des mainframes au cloud

L'infrastructure informatique a connu plusieurs révolutions majeures :

- **Années 1960-1970 : Mainframes**

Les machines centralisées partagées via des terminaux. Le temps de calcul était précieux et facturé à l'usage. IBM dominait ce marché avec ses System/360.

- **Années 1980 : Client-serveur**

L'arrivée du PC démocratise l'informatique. Les entreprises déploient des serveurs locaux connectés à des postes clients. Chaque département gère souvent sa propre infrastructure.

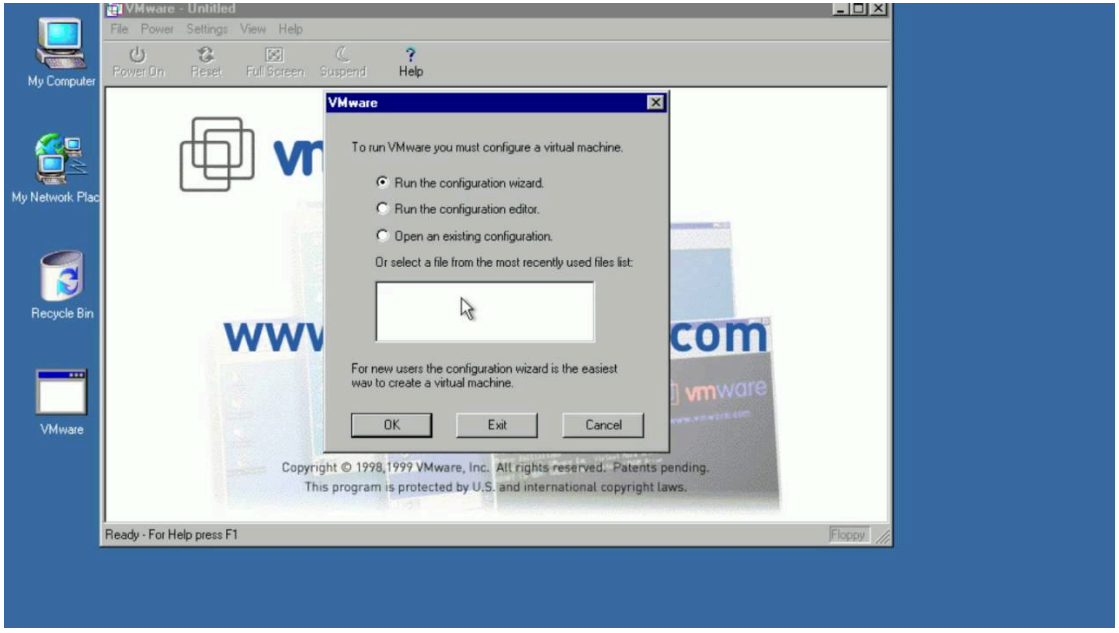
Évolution de l'infrastructure informatique

- **Années 1990 : Ère du Web**

Internet transforme les architectures. Les serveurs web deviennent critiques. Les datacenters se professionnalisent pour héberger les sites et applications.

- **Années 2000 : Virtualisation**

VMware[1] permet de faire tourner plusieurs systèmes sur une même machine physique. La consolidation des serveurs réduit les coûts matériels.



[1]VMware a été fondé en 1998. VMware Workstation 1.0 est sorti en 1999, et ESX Server (hyperviseur bare-metal) en 2001.

Évolution de l'infrastructure informatique

- **2006 et après : Cloud Computing**

Amazon lance EC2 et inaugure l'ère du cloud public. L'infrastructure devient programmable et disponible à la demande.

Datacenters traditionnels

Avant l'essor du cloud, chaque entreprise devait construire et maintenir sa propre infrastructure :

- **Investissement initial massif** : achat de serveurs, baies de stockage, équipements réseau
- **Délais de provisioning** : plusieurs semaines voire mois entre la commande et la mise en production
- **Surcapacité planifiée** : dimensionnement pour les pics de charge, entraînant un taux d'utilisation moyen de 10-15%
- **Équipes dédiées** : administrateurs systèmes, réseau, stockage, sécurité physique
- **Obsolescence** : renouvellement du matériel tous les 3-5 ans

Virtualisation : le précurseur

La virtualisation a posé les bases conceptuelles du cloud :

- **Hyperviseurs** : VMware ESX (2001), Xen (2003), KVM (2007) permettent d'abstraire le matériel
- **Consolidation** : plusieurs machines virtuelles sur un serveur physique optimisent l'utilisation des ressources
- **Isolation** : chaque VM fonctionne de manière indépendante avec son propre OS
- **Migration** : déplacement des VMs entre serveurs physiques sans interruption (vMotion)

La virtualisation reste cependant limitée à l'infrastructure interne de l'entreprise.

Les pionniers

Plusieurs acteurs ont façonné le cloud computing moderne :

- **Amazon Web Services (2006)** : lance S3 (Simple Storage Service) en mars 2006 puis EC2 (Elastic Compute Cloud) en août 2006[1], premiers services d'infrastructure à la demande
- **Google App Engine (2008)** : plateforme PaaS permettant de déployer des applications sans gérer l'infrastructure
- **Microsoft Azure (2010)** : entrée de Microsoft sur le marché cloud, forte intégration avec l'écosystème entreprise

[1] S3 a été lancé le 14 mars 2006. EC2 a suivi en version bêta le 25 août 2006.

L'innovation d'Amazon

Le lancement d'AWS en 2006 repose sur plusieurs innovations clés :

L'innovation d'Amazon

Le lancement d'AWS en 2006 repose sur plusieurs innovations clés :

Approche API-first : toute l'infrastructure est accessible via des API programmatiques. Plus besoin de tickets ou d'interventions manuelles pour provisionner un serveur.

L'innovation d'Amazon

Le lancement d'AWS en 2006 repose sur plusieurs innovations clés :

Approche API-first : toute l'infrastructure est accessible via des API programmatiques. Plus besoin de tickets ou d'interventions manuelles pour provisionner un serveur.

Modèle self-service : les développeurs peuvent créer des ressources en quelques minutes, sans passer par les équipes infrastructure. Cela change radicalement les cycles de développement.

L'innovation d'Amazon

Le lancement d'AWS en 2006 repose sur plusieurs innovations clés :

Approche API-first : toute l'infrastructure est accessible via des API programmatiques. Plus besoin de tickets ou d'interventions manuelles pour provisionner un serveur.

Modèle self-service : les développeurs peuvent créer des ressources en quelques minutes, sans passer par les équipes infrastructure. Cela change radicalement les cycles de développement.

Facturation à l'usage : paiement à l'heure (puis à la seconde) des ressources consommées. Fini les investissements initiaux massifs.

Cattle vs Pets

- **Pets** : Serveurs uniques, irremplaçables, soignés individuellement
- **Cattle** : Serveurs interchangeables, jetables, gérés en masse

Cette métaphore illustre un changement fondamental de mentalité :

Approche « Pets » (traditionnelle) :

- Chaque serveur a un nom unique
- En cas de panne, on diagnostique et répare
- Configuration manuelle et documentation détaillée
- La perte d'un serveur est un incident majeur

Philosophies fondamentales

Cette métaphore illustre un changement fondamental de mentalité :

Approche « Pets » (traditionnelle) :

- Chaque serveur a un nom unique
- En cas de panne, on diagnostique et répare
- Configuration manuelle et documentation détaillée
- La perte d'un serveur est un incident majeur

Approche « Cattle » (cloud native) :

- En cas de panne, on détruit et recrée automatiquement
- Configuration automatisée et reproductible
- La perte d'un serveur est un événement normal, géré automatiquement

Philosophies fondamentales

Cette métaphore illustre un changement fondamental de mentalité :

Approche « Pets » (traditionnelle) :

- Chaque serveur a un nom unique
- En cas de panne, on diagnostique et répare
- Configuration manuelle et documentation détaillée
- La perte d'un serveur est un incident majeur

Approche « Cattle » (cloud native) :

- En cas de panne, on détruit et recrée automatiquement
- Configuration automatisée et reproductible
- La perte d'un serveur est un événement normal, géré automatiquement

Exemple concret : Netflix détruit volontairement des serveurs en production (Chaos Monkey) pour s'assurer que son architecture supporte les pannes.

Infrastructure immutable

Le principe d'infrastructure immutable découle de l'approche « cattle » :

- **Jamais de modification en place** : on ne met pas à jour un serveur, on le remplace par une nouvelle version

Infrastructure immutable

Le principe d'infrastructure immutable découle de l'approche « cattle » :

- **Jamais de modification en place** : on ne met pas à jour un serveur, on le remplace par une nouvelle version
- **Images pré-construites** : les serveurs sont créés à partir d'images (AMI sur AWS) contenant toute la configuration

Infrastructure immutable

Le principe d'infrastructure immutable découle de l'approche « cattle » :

- **Jamais de modification en place** : on ne met pas à jour un serveur, on le remplace par une nouvelle version
- **Images pré-construites** : les serveurs sont créés à partir d'images (AMI sur AWS) contenant toute la configuration
- **Reproductibilité** : l'environnement de production peut être recréé à l'identique à tout moment

Infrastructure immutable

Le principe d'infrastructure immutable découle de l'approche « cattle » :

- **Jamais de modification en place** : on ne met pas à jour un serveur, on le remplace par une nouvelle version
- **Images pré-construites** : les serveurs sont créés à partir d'images (AMI sur AWS) contenant toute la configuration
- **Reproductibilité** : l'environnement de production peut être recréé à l'identique à tout moment
- **Rollback simplifié** : en cas de problème, retour à l'image précédente

Cette approche élimine le « configuration drift » où les serveurs divergent progressivement de leur état initial.

Modèles de service Cloud

Les services cloud se répartissent en trois niveaux, souvent représentés en pyramide :

IaaS : Infrastructure as a Service

Le niveau le plus bas, offrant les briques fondamentales :

- **Compute** : machines virtuelles (EC2, Compute Engine, Azure VMs)
- **Storage** : stockage bloc (EBS), objet (S3), fichiers (EFS)
- **Network** : réseaux virtuels, load balancers, firewalls

Le client gère : OS, middleware, applications, données

Le fournisseur gère : matériel, virtualisation, réseau physique

Usage typique : migration « lift and shift » d'applications existantes vers le cloud.

PaaS : Platform as a Service

Niveau intermédiaire orienté développeurs :

- **Runtime managé** : le client déploie son code, la plateforme gère l'infrastructure
- **Exemples** : Heroku, Google App Engine, Azure App Service, AWS Elastic Beanstalk

Le client gère : application et données

Le fournisseur gère : OS, runtime, scaling, haute disponibilité

Note : ce modèle est moins pertinent pour un cours orienté infrastructure, car il abstrait justement la couche infra.

SaaS : Software as a Service

Applications complètes accessibles via le web :

- **Exemples** : Gmail, Salesforce, Microsoft 365, Slack

Le client gère : ses données et la configuration applicative

Le fournisseur gère : tout le reste

Note : modèle utilisateur final, hors scope d'un cours infrastructure.

Cloud public

Infrastructure partagée entre de nombreux clients, opérée par un fournisseur tiers :

- **Avantages** : élasticité, pas d'investissement initial, innovation rapide
- **Inconvénients** : données chez un tiers, dépendance au fournisseur, coûts potentiellement élevés à grande échelle
- **Fournisseurs principaux** : AWS, Google Cloud, Microsoft Azure (voir chapitre dédié)

Modèles de déploiement

Cloud privé

Infrastructure dédiée à une seule organisation :

- **On-premise** : hébergée dans les datacenters de l'entreprise (OpenStack, VMware vSphere)
- **Hosted** : infrastructure dédiée chez un hébergeur

Cas d'usage :

Cloud privé

Infrastructure dédiée à une seule organisation :

- **On-premise** : hébergée dans les datacenters de l'entreprise (OpenStack, VMware vSphere)
- **Hosted** : infrastructure dédiée chez un hébergeur

Cas d'usage : contraintes réglementaires fortes, données sensibles, workloads prévisibles et stables.

Cloud hybride

Combinaison de cloud public et privé avec interconnexion :

- Permet de garder certaines données sensibles en interne
- Utilise le cloud public pour absorber les pics de charge (cloud bursting)
- **Solutions** : AWS Outposts, Azure Stack, Google Anthos

Multi-cloud

Utilisation simultanée de plusieurs fournisseurs cloud publics.

Avantages :

Multi-cloud

Utilisation simultanée de plusieurs fournisseurs cloud publics.

Avantages : éviter la dépendance à un fournisseur unique, exploiter les forces de chaque cloud.

Problèmes :

Multi-cloud

Utilisation simultanée de plusieurs fournisseurs cloud publics.

Avantages : éviter la dépendance à un fournisseur unique, exploiter les forces de chaque cloud.

Problèmes : multiplication des compétences requises, des outils, et des modèles de facturation. À réserver aux organisations matures.

Shared Responsibility Model

Le modèle de responsabilité partagée est un concept fondamental du cloud. Il sera détaillé dans le chapitre *Sécurité cloud*.

En résumé : le fournisseur sécurise **le cloud** (infrastructure physique, hyperviseurs), le client sécurise ce qu'il met **dans le cloud** (configuration, données, accès).

Régions et zones de disponibilité

Les fournisseurs cloud organisent leur infrastructure géographiquement :

- **Région** : zone géographique contenant plusieurs datacenters (ex: eu-west-1 = Irlande)
- **Zone de disponibilité (AZ)** : datacenter isolé au sein d'une région, avec alimentation et réseau indépendants
- **Latence** : quelques millisecondes entre AZs d'une même région, dizaines de ms entre régions

Carte des régions d'AWS:

Bonnes pratiques :

- Déployer sur plusieurs AZs pour la haute disponibilité
- Choisir une région proche des utilisateurs pour la latence
- Considérer les contraintes réglementaires (RGPD : données en Europe)

Élasticité et scalabilité

- **Scalabilité verticale (scale up)** : augmenter la puissance d'une machine (plus de CPU, RAM)
- **Scalabilité horizontale (scale out)** : ajouter des machines identiques

Élasticité et scalabilité

- **Scalabilité verticale (scale up)** : augmenter la puissance d'une machine (plus de CPU, RAM)
- **Scalabilité horizontale (scale out)** : ajouter des machines identiques

Le cloud favorise la scalabilité horizontale :

- Pas de limite physique
- Meilleure résilience (panne d'une machine = impact limité)
- Auto-scaling : ajustement automatique selon la charge

Haute disponibilité

La haute disponibilité et les SLA seront traités en détail dans le chapitre *Architecture Infrastructure Cloud*. **Principe clé** : éliminer les points de défaillance uniques (SPOF) en répartissant les ressources sur plusieurs zones de disponibilité.

Outline

Le marché du cloud computing

Les « Big Three »

Le marché du cloud est dominé par trois acteurs majeurs, souvent appelés hyperscalers[1] :

- **Amazon Web Services (AWS)** : Leader historique (lancé en 2006), 31% du marché
- **Microsoft Azure** : Second acteur, fort en entreprise, 20% du marché
- **Google Cloud Platform (GCP)** : Troisième, expertise Kubernetes native, 13% du marché

[1] Parts de marché du cloud public (IaaS + PaaS), Q3 2025, source : Synergy Research Group / Canalys.

Parts de marché et évolution

Le marché du cloud public représente plusieurs centaines de milliards de dollars annuels, avec une croissance soutenue.

Les parts de marché évoluent, mais AWS maintient son avance historique. Azure progresse grâce à son intégration avec l'écosystème Microsoft (Office 365, Active Directory). GCP se distingue par son expertise en data analytics et machine learning.

Historique et positionnement

AWS a été lancé en 2006 avec deux services fondateurs :

- **S3** (Simple Storage Service) : stockage objet
- **EC2** (Elastic Compute Cloud) : machines virtuelles

Cette approche « API-first » a révolutionné l'informatique en permettant de provisionner de l'infrastructure via du code.

Services principaux pour l'infrastructure

- **Compute** : EC2, Lambda, ECS, EKS
- **Storage** : S3, EBS, EFS
- **Networking** : VPC, Route 53, CloudFront, ELB
- **Database** : RDS, DynamoDB, ElastiCache

Régions et disponibilité

AWS dispose de plus de 30 régions dans le monde, chacune comprenant plusieurs zones de disponibilité (AZ). En Europe : Ireland (eu-west-1), Frankfurt (eu-central-1), Paris (eu-west-3), Stockholm, Milan, Zurich, Spain.

Historique et positionnement

Azure a été lancé en 2010 sous le nom « Windows Azure », rebaptisé en 2014. Son avantage principal est l'intégration avec l'écosystème Microsoft existant dans les entreprises.

Services principaux

- **Compute** : Virtual Machines, Functions, AKS
- **Storage** : Blob Storage, Managed Disks, Files
- **Networking** : Virtual Network, Load Balancer, Application Gateway
- **Database** : Azure SQL, Cosmos DB, Cache for Redis

Intégration entreprise

Azure se distingue par :

- Intégration native avec Active Directory (Entra ID)
- Licences hybrides Windows Server
- Support fort des environnements .NET
- Outils de migration depuis les datacenters on-premise

Historique et positionnement

GCP s'appuie sur l'infrastructure interne de Google, construite pour gérer des services comme Search, Gmail et YouTube. Lancé en 2008 avec App Engine (PaaS), il s'est étendu vers l'IaaS.

Google Cloud Platform (GCP)

Services principaux

- **Compute** : Compute Engine, Cloud Functions, GKE
- **Storage** : Cloud Storage, Persistent Disks, Filestore
- **Networking** : VPC, Cloud Load Balancing, Cloud CDN
- **Database** : Cloud SQL, Firestore, Memorystore

Google Cloud Platform (GCP)

Points forts

- **Kubernetes** : Google est le créateur de Kubernetes, GKE est considéré comme l'implémentation de référence
- **Réseau global** : Infrastructure réseau privée mondiale
- **BigQuery** : Data warehouse serverless très performant
- **Tarification** : Facturation à la seconde, remises automatiques pour usage soutenu

Alternatives européennes

Souveraineté des données

La souveraineté numérique est un enjeu majeur en Europe. Les données hébergées chez les hyperscalers américains sont soumises au Cloud Act, permettant aux autorités américaines d'y accéder.

Alternatives européennes

OVHcloud

Fournisseur français historique, OVHcloud propose :

- Datacenters en France et en Europe
- Conformité RGPD garantie
- Services IaaS compétitifs (Public Cloud, Bare Metal)
- Alternative européenne crédible pour la souveraineté

Alternatives européennes

Scaleway

Filiale du groupe Iliad (Free), Scaleway offre :

- Infrastructure 100% européenne
- Tarification agressive
- Services cloud modernes (Kubernetes, Serverless)
- Focus sur la simplicité d'utilisation

Alternatives européennes

Autres acteurs

- **Exoscale** (Suisse) : conformité stricte
- **Hetzner** (Allemagne) : excellent rapport qualité/prix
- **Infomaniak** (Suisse) : engagement écologique

Comparaison des services

Critères de choix

Le choix d'un provider dépend de plusieurs facteurs :

- **Existant technologique** : écosystème Microsoft -> Azure
- **Expertise Kubernetes** : GCP excelle
- **Maturité des services** : AWS a le catalogue le plus large
- **Conformité** : contraintes réglementaires (HDS, souveraineté)
- **Coûts** : varient selon les services et l'usage
- **Localisation** : présence de régions dans les zones cibles

Comparaison des services

Console web et interfaces

En production, chaque provider propose une console web :

- **AWS** : console.aws.amazon.com
- **GCP** : console.cloud.google.com
- **Azure** : portal.azure.com

Exercices

Exercice

Installer LocalStack et vérifier que les services sont fonctionnels. Lister les services disponibles via `curl http://localhost:4566/_localstack/health`.

Exercice

Comparer les prix d'une VM (2 vCPU, 8 Go RAM) entre AWS, Azure et GCP pour une région européenne.

Exercice

Identifier les équivalences de services entre AWS, GCP et Azure pour : compute, stockage objet, réseau virtuel, et load balancing.

Outline

Qu'est-ce que la virtualisation ?

La virtualisation est née dans les années 1960 chez IBM. Elle a connu un essor majeur dans les années 2000 avec VMware, devenant la base technique du cloud computing.

Hyperviseurs

L'hyperviseur est le logiciel qui crée et gère les machines virtuelles. Il abstrait le matériel physique et le partage entre plusieurs VMs.

Type 1 : Bare-metal

Les hyperviseurs de Type 1 s'exécutent directement sur le matériel, sans système d'exploitation hôte. Ils offrent de meilleures performances et sont utilisés en production.

Exemples :

- VMware ESXi
- Xen (historiquement AWS EC2, migré vers Nitro depuis 2017)[1]
- KVM (Google Cloud, via des adaptations propriétaires)
- Hyper-V (Azure)

Le Type 1 offre de meilleures performances car il n'y a pas d'OS intermédiaire. C'est le standard en production cloud.

[1] AWS a progressivement migré ses instances d'Xen vers son système Nitro, basé sur KVM, à partir de 2017. Aujourd'hui la quasi-totalité des instances AWS utilisent Nitro.

Types d'hyperviseurs

Type 2 : Hosted

Les hyperviseurs de Type 2 s'exécutent sur un système d'exploitation hôte. Ils sont plus simples à installer mais moins performants.

Exemples : VirtualBox, VMware Workstation. Principalement utilisés pour le développement local et les tests.

Types d'hyperviseurs

Paravirtualisation vs Full virtualization

- **Full virtualization** : le guest OS n'est pas modifié, l'hyperviseur émule tout le matériel
- **Paravirtualisation** : le guest OS est modifié pour communiquer directement avec l'hyperviseur (meilleures performances)

Conteneurs vs VMs

Une VM virtualise tout le matériel et nécessite un OS complet, tandis qu'un conteneur partage le kernel de l'hôte et isole uniquement l'application et ses dépendances.

- VMs : isolation forte, overhead important (Go de RAM par VM), démarrage en minutes.
- Conteneurs : isolation plus légère, overhead minimal (Mo), démarrage en secondes.

Technologies sous-jacentes Linux

Les conteneurs reposent sur :

- **Namespaces** : isolation des ressources
- **Cgroups** : limitation des ressources
- **Union filesystems** : layers et images

Types de namespaces

Les namespaces permettent d'isoler différents aspects du système pour chaque conteneur.

- PID : Process isolation
- NET : Network stack isolation
- MNT : Filesystem mount points
- UTS : Hostname and domain
- IPC : Inter-process communication
- USER : User and group IDs

Chaque conteneur a sa propre vue du système grâce aux namespaces. Commandes utiles : `unshare` (créer un namespace), `nsenter` (entrer dans un namespace existant).

Cgroups (Control Groups)

Gestion des ressources

Les cgroups permettent de limiter les ressources (CPU, mémoire, I/O) utilisées par un groupe de processus.

Les cgroups empêchent un conteneur de consommer toutes les ressources de l'hôte, garantissant l'isolation entre conteneurs.

Docker : Architecture

Composants Docker

Docker comprend trois composants principaux :

- **Docker daemon** : service qui gère les conteneurs
- **Docker CLI** : interface en ligne de commande
- **Registry** : stockage et distribution des images

Le daemon (dockerd) tourne en arrière-plan. Le CLI envoie des commandes au daemon. Le registry stocke les images.

Images et layers

Les images Docker sont construites en couches (layers). Chaque instruction du Dockerfile crée une nouvelle couche.

```
FROM alpine:3.14
RUN apk add --no-cache nginx
COPY nginx.conf /etc/nginx/
```

Le système de layers permet de réutiliser les couches communes entre images, accélérant les builds et réduisant le stockage.

Isolation et limites

Les conteneurs partagent le kernel de l'hôte ! Ce n'est pas une isolation aussi forte qu'une VM.

Un conteneur compromis peut potentiellement affecter l'hôte. L'isolation n'est pas aussi forte qu'une VM.

Bonnes pratiques de sécurité

Quelques règles simples pour des conteneurs plus sûrs :

- Ne pas exécuter en tant que root dans le conteneur
- Pinner les sha256 (pas comme dans l'exemple d'avant)
- Utiliser des images officielles et les maintenir à jour
- Limiter les ressources avec les options `--memory` et `--cpus`

Docker Compose

Docker Compose permet de définir et gérer des applications multi-conteneurs.

```yaml
version: '3'
services:
  web:
    image: nginx
    ports:
      - "80:80"
  db:
    image: postgres
    environment:
      POSTGRES_PASSWORD: secret
```

Docker Compose est idéal pour le développement et les environnements de test. Pour la production à grande échelle, on utilise des orchestrateurs comme Kubernetes.

Autres technologies de conteneurs

Il existe d'autres technologies qui offrent différents niveaux d'isolation :

Firecracker : microVMs utilisées par AWS Lambda

- Démarrage en 125ms
- Empreinte mémoire minimale (5MB)
- Isolation VM avec performance conteneur

Autres technologies de conteneurs

Il existe d'autres technologies qui offrent différents niveaux d'isolation :

Firecracker : microVMs utilisées par AWS Lambda

- Démarrage en 125ms
- Empreinte mémoire minimale (5MB)
- Isolation VM avec performance conteneur

Kata Containers : conteneurs basés sur des VMs légères

- Compatible OCI et CRI
- Isolation hardware via hyperviseur léger

Autres technologies de conteneurs

Il existe d'autres technologies qui offrent différents niveaux d'isolation :

Firecracker : microVMs utilisées par AWS Lambda

- Démarrage en 125ms
- Empreinte mémoire minimale (5MB)
- Isolation VM avec performance conteneur

Kata Containers : conteneurs basés sur des VMs légères

- Compatible OCI et CRI
- Isolation hardware via hyperviseur léger

gVisor : sandboxing au niveau applicatif

- Kernel utilisateur écrit en Go
- Intercepte les syscalls
- Utilisé par Google Cloud Run

Ces technologies sont utilisées quand on a besoin d'une isolation plus forte que les conteneurs classiques (multi-tenancy, workloads non fiables).

Qu'est-ce qu'un registry ?

Un registry est un service qui stocke et distribue des images de conteneurs.

Registries traditionelles :

- Docker Hub (public)
- Amazon ECR
- Google Container Registry
- Harbor (self-hosted)

Docker Hub est le registry public par défaut. `docker pull` télécharge, `docker push` publie une image.

Outline

Régions et géographie

Une **région** est un cluster de datacenters déployés dans un périmètre défini et connectés via un réseau dédié à faible latence.

Critères de choix d'une région :

- Proximité géographique des utilisateurs (latence)
- Conformité réglementaire (RGPD, localisation des données)
- Disponibilité des services requis
- Coût (varie selon les régions)

Zones de disponibilité (AZ)

Une AZ est un datacenter isolé physiquement au sein d'une région.

- Alimentation électrique et réseau indépendants
- Connexion haut débit entre AZ (quelques ms de latence)
- Design multi-AZ : répartir les ressources sur 2+ AZ pour la résilience

Edge locations et CDN

Les edge locations sont des points de présence pour servir le contenu au plus près des utilisateurs.

- Caching de contenu statique (images, vidéos, fichiers)
- Services : CloudFront (AWS), Cloud CDN (GCP), Azure CDN
- Réduction significative de la latence pour les utilisateurs distants

Types d'instances

Familles d'instances :

- General Purpose (t3, n2)
- Compute Optimized (c5, c2)
- Memory Optimized (r5, m2)
- Storage Optimized (i3, d2)
- Accelerated Computing (p4, a2)

Compute : Les fondations

- General Purpose : applications web, environnements de dev
- Compute Optimized : calcul intensif, batch processing
- Memory Optimized : bases de données in-memory, caches
- Storage Optimized : data warehouses, systèmes de fichiers distribués
- Accelerated : machine learning, rendu graphique

Sizing et rightsizing

Le surdimensionnement est la première source de gaspillage cloud ! (métiers FinOps)

Métriques à surveiller : CPU, mémoire, I/O réseau et disque. Outils : AWS Compute Optimizer, GCP Recommender, Azure Advisor.

Modèles de tarification

- **On-demand** : tarif standard, flexibilité maximale, aucun engagement
- **Reserved** : engagement 1-3 ans, réduction jusqu'à 72%
- **Spot** : capacité excédentaire, jusqu'à 90% de réduction, peut être interrompu
- **Savings Plans** : engagement de dépense horaire, plus flexible que Reserved

Auto Scaling et élasticité

Horizontal scaling

- **Scale-out** : ajouter des instances pour absorber la charge
- **Scale-in** : retirer des instances quand la charge diminue
- Prérequis : applications stateless (sans état local persistant)
- L'état doit être externalisé (base de données, cache distribué)

Instance metadata et user-data

- Instance Metadata Service : API locale (169.254.169.254) pour accéder aux infos de l'instance
- User-data : script exécuté au démarrage pour configurer l'instance
- IMDS v2 : version sécurisée avec token obligatoire (protection contre SSRF)

Types de load balancers

- **Layer 4 (TCP/UDP)** : Network Load Balancer, très performant, routage par IP/port
- **Layer 7 (HTTP/S)** : Application Load Balancer, routage par URL, headers, host
- NLB : millions de requêtes/seconde, latence minimale
- ALB : routing intelligent, terminaison SSL, intégration WAF

Load Balancing

Algorithmes de distribution

- **Round-robin** : distribution équitable en rotation
- **Least connections** : envoie vers l'instance la moins chargée
- **IP hash** : même client toujours vers même serveur (sticky sessions)

Cross-zone load balancing

Distribue le trafic équitablement entre toutes les AZ, même si le nombre d'instances diffère.

- Avantage : meilleure répartition de charge
- Inconvénient : coût de transfert inter-AZ
- Activé par défaut sur ALB, optionnel sur NLB

Patterns de HA

- **Active-Active** : toutes les instances traitent le trafic, meilleure utilisation des ressources
- **Active-Passive** : instances de secours en standby, basculement en cas de panne
- Trade-off : Active-Active plus complexe mais meilleur RTO

SLA et disponibilité

Définition

- 99% = 3.65 jours d'indisponibilité/an
- 99.9% = 8.76 heures/an
- 99.99% = 52.56 minutes/an
- 99.999% = 5.26 minutes/an

Concepts de base

- **RTO** (Recovery Time Objective) : temps maximum acceptable pour restaurer le service
- **RPO** (Recovery Point Objective) : perte de données maximum acceptable (en temps)

Exemples :

- Site e-commerce : RTO 1h, RPO 15min (perte de quelques commandes acceptable)
- Application bancaire : RTO 5min, RPO 0 (aucune perte de transaction)

Tests de DR

Un plan de DR non testé est un plan qui ne fonctionne pas. Planifiez des tests réguliers :

- **Tabletop exercises** : simulation sur papier avec l'équipe
- **Failover tests** : basculement réel vers le site DR
- **Chaos engineering** : injection de pannes contrôlées

Spot Instances

Use cases adaptés aux Spot Instances :

- Batch processing, CI/CD, tests
- Traitement de données parallélisable
- Workloads tolérants aux interruptions

Gestion des interruptions : notification 2 min avant, checkpointing, graceful shutdown.

Cost allocation et tagging

```
tags = {
  Environment = "production"
  Team        = "infrastructure"
  CostCenter  = "IT-OPS"
}
```

Outline

Concepts fondamentaux

Définition

Le SDN sépare le plan de contrôle (décisions de routage) du plan de données (forwarding), permettant une gestion centralisée et programmable du réseau.

- **Architecture** : Contrôleur central (cerveau) + switches (exécution)
- **OpenFlow** : Protocole standard de communication contrôleur-switch
- **Contrôleurs** : ONOS, OpenDaylight, ou solutions propriétaires cloud

SDN dans le cloud public

Chaque provider implémente son propre SDN :

- AWS VPC utilise un hyperviseur réseau propriétaire
- Google Andromeda pour le réseau virtuel GCP
- Azure utilise un SDN basé sur VXLAN

Virtual Private Cloud (VPC)

Isolation réseau

Un VPC est un réseau virtuel isolé logiquement dans le cloud, donnant un contrôle total sur l'environnement réseau.

- **Multi-tenancy** : Plusieurs clients partagent l'infrastructure physique
- **Isolation** : VXLAN, tunnels encapsulés, tables de routage séparées
- Chaque VPC est complètement isolé des autres par défaut

CIDR et adressage IP

```
resource "aws_vpc" "main" {
  cidr_block = "10.0.0.0/16"
  enable_dns_hostnames = true
}
```

- **RFC1918** : Plages privées (10.0.0.0/8, 172.16.0.0/12, 192.168.0.0/16)
- Planifier large : difficile d'étendre un VPC après création
- Éviter les chevauchements avec le réseau on-premise

Subnets et segmentation

Subnets publics vs privés

- **Subnet public** : Route vers Internet Gateway, IPs publiques possibles
- **Subnet privé** : Pas d'accès direct Internet, utilise NAT pour sortir
- Best practice : bases de données et backends en subnet privé

Calcul CIDR et subnetting

```
VPC: 10.0.0.0/16 (65,536 IPs)
├── Public:  10.0.0.0/24  (256 IPs)
├── Private: 10.0.1.0/24  (256 IPs)
└── Database: 10.0.2.0/24 (256 IPs)
```

- AWS réserve 5 IPs par subnet (network, router, DNS, future, broadcast)
- Outils : ipcalc, Visual Subnet Calculator
- Attention aux overlapping lors du peering VPC

Multi-AZ networking

- Répartir les subnets sur plusieurs Availability Zones
- Une AZ = un datacenter logique (domaine de panne isolé)
- Haute disponibilité : déployer dans minimum 2 AZs

Route tables

```
Destination      Target
0.0.0.0/0        igw-xxxxx     # Internet Gateway
10.0.0.0/16      local         # VPC local routing
192.168.0.0/16   pcx-xxxxx     # Peering connection
```

- Route la plus spécifique gagne (longest prefix match)
- Route « local » toujours prioritaire pour le trafic intra-VPC
- Propagation BGP automatique depuis VPN Gateway

Internet Gateway (IGW)

- Un IGW par VPC, attaché explicitement
- Haute disponibilité native (géré par le provider)
- Pas de limite de bande passante, scale automatiquement

NAT Gateway vs NAT Instance

Couts Cloud

Routing dans le cloud

NAT Gateway : 45$/mois + data transfer
NAT Instance : coût de l'instance EC2

- NAT Gateway : managé, 45 Gbps, haute dispo par AZ
- NAT Instance : EC2 classique, personnalisable, single point of failure
- Pour HA : un NAT Gateway par AZ utilisée

VPN Site-to-Site

Infrastructure

IPSec VPN pour connexion sécurisée on-premise ↔ cloud

- Customer Gateway (on-premise)
- Virtual Private Gateway (cloud)

- Deux tunnels IPSec pour redondance (actif/passif)
- BGP recommandé pour le routage dynamique
- Latence variable selon Internet (non garantie)

AWS Direct Connect / Azure ExpressRoute

- Liaison dédiée physique (1 Gbps ou 10 Gbps)
- Latence stable et bande passante garantie
- Coût élevé : port + data transfer + colocation

VPC Peering

```
resource "aws_vpc_peering_connection" "peer" {
  vpc_id        = aws_vpc.main.id
  peer_vpc_id   = aws_vpc.peer.id
  peer_region   = "eu-west-1"
}
```

- Pas de peering transitif (A↔B et B↔C n'implique pas A↔C)
- Les CIDR ne doivent pas se chevaucher
- Cross-region possible mais avec coût de transfert inter-région

PrivateLink / Private Endpoints

Note

Les endpoints privés permettent d'accéder aux services cloud (S3, bases de données managées) sans passer par Internet. Cela améliore la sécurité et réduit la latence.

Layer 4 Load Balancing

Définition

Le load balancing L4 opère au niveau Transport (TCP/UDP), routant le trafic basé sur IP:port sans inspection du contenu.

- Network Load Balancer : millions de requêtes/sec, très faible latence
- Supporte TCP, UDP, TLS
- Connection draining : termine proprement les connexions existantes

Layer 7 Load Balancing

- Application Load Balancer : inspecte le contenu HTTP
- Routing basé sur URL, headers, hostname
- Supporte WebSocket et HTTP/2

Global Load Balancing

- Distribue le trafic entre plusieurs régions
- GeoDNS : route vers la région la plus proche de l'utilisateur
- Failover automatique si une région est indisponible

Health checks networking

- TCP check : vérifie que le port répond (rapide)
- HTTP check : vérifie une URL spécifique (plus précis)
- Ajuster interval et threshold selon la criticité

DNS dans le cloud

DNS privé

- Zones DNS privées : résolution uniquement depuis le VPC
- Permet des noms internes (db.internal, api.internal)
- Résolveur DNS à l'adresse VPC CIDR + 2

DNS public et CDN

- TTL court (60s) : changements rapides, plus de requêtes
- TTL long (3600s) : moins de charge, propagation lente
- CDN : souvent TTL très court pour flexibilité

Split-horizon DNS

- Même nom de domaine, réponses différentes selon la source
- Interne : IP privée (10.0.1.5)
- Externe : IP publique ou load balancer

Security Groups

```
// Stateful firewall rules
// Inbound:  TCP 443 from 0.0.0.0/0
// Outbound: ALL to 0.0.0.0/0
```

- Stateful : le retour est automatiquement autorisé
- Toutes les règles sont évaluées (pas d'ordre)
- Par défaut : tout sortant autorisé, tout entrant refusé

Network ACLs

- Stateless : il faut autoriser entrant ET sortant explicitement
- Appliqué au niveau subnet (pas instance)
- Règles évaluées par numéro croissant, première match gagne

WAF et DDoS Protection

- WAF : filtre les requêtes malveillantes (SQL injection, XSS)
- AWS Shield / CloudFlare : protection DDoS L3/L4/L7
- Rate limiting : limite le nombre de requêtes par IP

Network segmentation

- Microsegmentation : règles fines entre chaque workload
- Zero-trust : ne jamais faire confiance, toujours vérifier
- Principe du moindre privilège appliqué au réseau

Bandwidth et limites

Infrastructure

Limites de bande passante par instance :

- t3.micro : jusqu'à 5 Gbps
- c5n.18xlarge : 100 Gbps
- Placement groups : jusqu'à 10 Gbps entre instances

- Burst : débit temporaire élevé (crédits réseau)
- Sustained : débit garanti en continu
- Instances plus grandes = meilleure performance réseau

Enhanced networking

- SR-IOV : accès direct au hardware réseau (bypass hyperviseur)
- ENA : driver optimisé AWS pour haut débit
- DPDK : traitement packets en userspace (très haute performance)

Performance réseau

Jumbo frames

```
// MTU 9000 for better throughput
sudo ip link set dev eth0 mtu 9000
```

- MTU 9000 au lieu de 1500 : moins de fragmentation
- Tous les composants du chemin doivent supporter jumbo frames
- Utile pour transferts volumineux (backup, big data)

Network latency optimization

- Placement group cluster : instances physiquement proches
- Communication same-AZ : gratuite et plus rapide
- Cross-AZ : latence 1ms, coût de transfert

Content Delivery Networks

- Réseau de serveurs cache distribués mondialement
- Réduit la latence en servant depuis le point le plus proche
- Décharge le serveur origin (économie de bande passante)

Edge locations

- Points de présence (PoPs) sur tous les continents
- Cache-Control headers définissent la politique de cache
- Invalidation possible mais coûteuse (préférer versioning)

Lambda@Edge / CloudFlare Workers

> **Note**
>
> Focus infrastructure : routing, caching, pas le code applicatif

- Exécution de code au edge (proche de l'utilisateur)
- Manipulation des headers, redirections, A/B testing
- Geo-routing : contenu différent selon le pays

VPC Flow Logs

Définition

Les Flow Logs enregistrent le trafic réseau entrant et sortant des interfaces réseau dans un VPC. Ils permettent de diagnostiquer les problèmes de connectivité et d'auditer le trafic.

```
{
  "srcaddr": "10.0.1.5",
  "dstaddr": "10.0.2.10",
  "srcport": 45928,
  "dstport": 443,
  "protocol": 6,
  "packets": 20,
  "bytes": 4000
}
```

Note

Monitoring réseau

Les Flow Logs peuvent être stockés dans des services de logs cloud (CloudWatch, Cloud Logging) pour analyse ultérieure.

IPv6 dans le cloud

Note

Les VPC cloud supportent le dual-stack IPv4/IPv6. IPv6 offre un espace d'adressage quasi illimité et simplifie certaines configurations réseau. La plupart des services cloud modernes supportent IPv6, mais IPv4 reste le standard pour les déploiements classiques.

Exercices

Exercice

Designer un VPC simple avec un subnet public et un subnet privé. Expliquer le rôle de chaque composant (Internet Gateway, route tables, NAT Gateway).

Exercice

Calculer un plan d'adressage CIDR pour un VPC hébergeant 3 environnements (dev, staging, prod) avec 50 machines chacun.

Exercice

Configurer les Security Groups pour une application web 3-tiers : frontend (HTTP/HTTPS depuis Internet), backend (accès depuis frontend uniquement), base de données (accès depuis backend uniquement).

Outline

Types de stockage

Le cloud propose trois paradigmes de stockage fondamentaux, chacun adapté à des cas d'usage spécifiques. Comprendre leurs différences est essentiel pour concevoir une architecture performante et économique.

Block Storage

Définition

Block storage : volumes attachables aux instances, comme des disques durs virtuels. Les données sont organisées en blocs de taille fixe, accessibles via des protocoles de bas niveau.

Types de stockage

Services par provider :

- **AWS** : Elastic Block Store (EBS)
- **GCP** : Persistent Disks
- **Azure** : Managed Disks

Caractéristiques communes :

- Attachement à une seule instance (= VM)
- Persistance indépendante du cycle de vie de l'instance (VM détruite = stockage intact)
- Possibilité de redimensionnement

Cas d'usage principaux :

Types de stockage

Services par provider :

- **AWS** : Elastic Block Store (EBS)
- **GCP** : Persistent Disks
- **Azure** : Managed Disks

Caractéristiques communes :

- Attachement à une seule instance (= VM)
- Persistance indépendante du cycle de vie de l'instance (VM détruite = stockage intact)
- Possibilité de redimensionnement

Cas d'usage principaux :

- **Boot volumes** : Disques système des instances
- **Bases de données** : MySQL, PostgreSQL, MongoDB (accès aléatoire rapide requis)
- **Applications transactionnelles** : Tout système nécessitant des I/O à faible latence

Object Storage

Définition

Object storage : système de stockage distribué où chaque fichier (objet) est identifié par une clé unique dans un espace de noms plat (bucket). Les objets sont accessibles via HTTP/HTTPS.

Types de stockage

Services par provider :

- **AWS** : Simple Storage Service (S3)
- **GCP** : Cloud Storage (GCS)
- **Azure** : Blob Storage

Caractéristiques communes :

- Scalabilité quasi-illimitée (exaoctets)
- Durabilité 11 nines (99.999999999%)
- Accès via API REST
- Pas de hiérarchie de répertoires (simulation par préfixes)

Cas d'usage principaux :

Types de stockage

Services par provider :

- **AWS** : Simple Storage Service (S3)
- **GCP** : Cloud Storage (GCS)
- **Azure** : Blob Storage

Caractéristiques communes :

- Scalabilité quasi-illimitée (exaoctets)
- Durabilité 11 nines (99.999999999%)
- Accès via API REST
- Pas de hiérarchie de répertoires (simulation par préfixes)

Cas d'usage principaux :

- **Fichiers statiques** : Images, vidéos, assets web
- **Backups et archives** : Sauvegardes de bases de données, logs
- **Data lakes** : Stockage de données brutes pour analytics
- **Distribution de contenu** : Intégration avec CDN

```
aws s3 cp backup.sql s3://mon-bucket/backups/2024/backup.sql
```

File Storage

Définition

File storage : système de fichiers partagé accessible via des protocoles réseau standards (NFS, SMB/CIFS). Permet l'accès concurrent depuis plusieurs instances.

Types de stockage

Services par provider :

- **AWS** : Elastic File System (EFS), FSx
- **GCP** : Filestore
- **Azure** : Azure Files

Caractéristiques communes :

- Montage simultané sur plusieurs instances
- Sémantique POSIX (permissions, locks)
- Scalabilité automatique ou provisionnée

Cas d'usage principaux :

Types de stockage

Services par provider :

- **AWS** : Elastic File System (EFS), FSx
- **GCP** : Filestore
- **Azure** : Azure Files

Caractéristiques communes :

- Montage simultané sur plusieurs instances
- Sémantique POSIX (permissions, locks)
- Scalabilité automatique ou provisionnée

Cas d'usage principaux :

- **Accès partagé** : Répertoires home, fichiers de configuration partagés
- **Applications legacy** : Logiciels conçus pour NFS/SMB
- **CMS et web** : WordPress, Drupal avec plusieurs serveurs web
- **Environnements de développement** : Code source partagé entre instances

Attention

Le file storage a généralement des performances inférieures au block storage pour les opérations à faible latence. Ne l'utilisez pas pour des bases de données transactionnelles.

Types de volumes

Les providers cloud proposent différents types de volumes optimisés pour des charges de travail spécifiques.

Définition

IOPS (Input/Output Operations Per Second) : Nombre d'opérations de lecture/écriture par seconde. Métrique clé pour les applications transactionnelles.

Throughput (débit) : Volume de données transféré par seconde (Mo/s). Métrique clé pour les transferts séquentiels.

Block Storage en détail

Types de volumes AWS EBS :

- gp3 (SSD) : Usage general, 16000 IOPS max, 1000 Mo/s
- io2 (SSD) : Bases critiques, 256000 IOPS max, 4000 Mo/s
- st1 (HDD) : Big data, logs, 500 IOPS, 500 Mo/s
- sc1 (HDD) : Archivage froid, 250 IOPS, 250 Mo/s

Block Storage en détail

Types de volumes AWS EBS :

- gp3 (SSD) : Usage general, 16000 IOPS max, 1000 Mo/s
- io2 (SSD) : Bases critiques, 256000 IOPS max, 4000 Mo/s
- st1 (HDD) : Big data, logs, 500 IOPS, 500 Mo/s
- sc1 (HDD) : Archivage froid, 250 IOPS, 250 Mo/s

GCP Persistent Disks :

- pd-balanced : SSD équilibré (80000 IOPS)
- pd-ssd : SSD haute performance (100000 IOPS)
- pd-standard : HDD économique
- pd-extreme : Ultra-haute performance (120000 IOPS)

SSD vs HDD - Critères de choix :

- **SSD** : Applications nécessitant des accès aléatoires rapides (bases de données, boot volumes). Latence inférieure à 1ms.
- **HDD** : Charges de travail séquentielles à fort débit (analytics, logs). Coût par Go inférieur.

```
aws ec2 create-volume --availability-zone eu-west-1a \
    --size 500 --volume-type gp3 --iops 10000
```

Snapshots et backups

Définition

Snapshot : Copie instantanée d'un volume à un instant T. Les snapshots sont incrémentaux : seuls les blocs modifiés depuis le dernier snapshot sont stockés.

Fonctionnement des snapshots incrémentaux :

1. Premier snapshot : copie complète du volume
2. Snapshots suivants : uniquement les blocs modifiés (delta)
3. Chaque snapshot reste autonome pour la restauration
4. Suppression d'un snapshot intermédiaire : les blocs nécessaires sont préservés

```
aws ec2 create-snapshot --volume-id vol-1234567890abcdef0 \
    --description "Backup pre-mise-a-jour"

aws ec2 copy-snapshot --source-region eu-west-1 \
    --source-snapshot-id snap-1234567890abcdef0 \
    --destination-region us-east-1
```

Block Storage en détail

Important

Les snapshots sont stockés dans le object storage du provider (S3 pour AWS, GCS pour GCP). Ils bénéficient de la même durabilité 11 nines.

Performance tuning

Définition

IOPS provisionnées : Réservation d'un nombre garanti d'IOPS, indépendamment de la taille du volume. Permet de découpler performance et capacité.

Burst credits : Système de crédit permettant des pics de performance temporaires.

Calcul des performances de base (AWS gp3) :

- IOPS de base : 3000 (inclus)
- IOPS provisionnables : jusqu'à 16000 (avec coût supplémentaire)
- Throughput de base : 125 Mo/s (inclus)
- Throughput provisionnable : jusqu'à 1000 Mo/s

Attention

Attention aux burst credits épuisés. Un volume gp2 sans credits disponibles retombe à ses IOPS de base (3 IOPS/Go). Surveillez la métrique BurstBalance dans CloudWatch.

Buckets et organisation

Définition

Bucket : Conteneur de niveau supérieur pour les objets. Le nom du bucket doit être globalement unique (sur tout le provider) et respecter les conventions DNS.

Règles de nommage :

- Longueur : 3-63 caractères
- Caractères autorisés : lettres minuscules, chiffres, tirets
- Pas de points consécutifs ni de format IP
- Préfixe par organisation recommandé : monentreprise-projet-env

```
aws s3 mb s3://monentreprise-backups-prod --region eu-west-1
```

Organisation recommandée par préfixes :

```
monentreprise-data-prod/
- raw/2024/01/
- raw/2024/02/
- processed/daily/
- processed/weekly/
- backups/databases/
- backups/configs/
```

Les préfixes simulant des répertoires permettent une gestion fine des permissions IAM et des lifecycle policies.

Classes de stockage

Les providers proposent différentes classes de stockage optimisées pour la fréquence d'accès aux données.

Object Storage pour l'infrastructure

Classes de stockage AWS S3 :

- **Standard** (99.99% dispo) : Accès fréquent, latence ms
- **Standard-IA** (99.9% dispo) : Accès mensuel, latence ms
- **One Zone-IA** (99.5% dispo) : Backups reproductibles
- **Glacier Instant Retrieval** : Archives consultées, latence ms
- **Glacier Flexible Retrieval** : Archives rares, latence 1-5 min
- **Glacier Deep Archive** : Conformité long terme, latence 12h

Object Storage pour l'infrastructure

GCP Cloud Storage :

- Standard : Accès fréquent
- Nearline : Accès mensuel (30 jours minimum)
- Coldline : Accès trimestriel (90 jours minimum)
- Archive : Accès annuel (365 jours minimum)

Attention

Les classes Infrequent Access et Archive facturent des frais de récupération par Go. Calculez le coût total (stockage + récupération) avant de migrer des données fréquemment accédées.

Lifecycle policies

Définition

Lifecycle policy : Règle automatisant la transition des objets entre classes de stockage ou leur suppression après une période définie.

Exemple de configuration lifecycle S3 :

- Après 30 jours : transition vers STANDARD_IA
- Après 90 jours : transition vers GLACIER_IR
- Après 365 jours : transition vers DEEP_ARCHIVE
- Après 2555 jours (7 ans) : suppression

Une lifecycle policy bien configurée peut réduire les coûts de stockage de 70-80% pour des données de logs ou backups. Exemple : 1 To de logs à 0.023 euros/Go (Standard) vs 0.004 euros/Go (Glacier DA) = économie de 228 euros/mois.

Versioning et replication

Définition

Versioning : Conservation de toutes les versions d'un objet lors de modifications ou suppressions. Protège contre les erreurs humaines et permet la récupération.

Cross-Region Replication (CRR) : Réplication automatique des objets vers un bucket dans une autre région pour disaster recovery.

Object Storage pour l'infrastructure

```
aws s3api put-bucket-versioning \
    --bucket monentreprise-config-prod \
    --versioning-configuration Status=Enabled

aws s3api list-object-versions \
    --bucket monentreprise-config-prod \
    --prefix config.yaml
```

Prérequis pour la réplication cross-region :

1. Versioning activé sur source ET destination
2. Rôle IAM avec permissions de réplication
3. Buckets dans des régions différentes

Important

La réplication cross-region est asynchrone (quelques secondes à minutes). Pour une cohérence stricte, implémentez une vérification applicative.

Cache et stockage in-memory

Définition

Cache distribué : Système de stockage clé-valeur en mémoire (RAM) offrant des latences sub-milliseconde. Utilisé pour accélérer les accès aux données fréquentes.

Services managés :

- **AWS** : ElastiCache (Redis, Memcached)
- **GCP** : Memorystore (Redis, Memcached)
- **Azure** : Azure Cache for Redis

Cache et stockage in-memory

Définition

Cache distribué : Système de stockage clé-valeur en mémoire (RAM) offrant des latences sub-milliseconde. Utilisé pour accélérer les accès aux données fréquentes.

Services managés :

- **AWS** : ElastiCache (Redis, Memcached)
- **GCP** : Memorystore (Redis, Memcached)
- **Azure** : Azure Cache for Redis

Redis vs Memcached :

- Redis : Structures riches (listes, sets), persistance possible, réplication. Idéal pour sessions, leaderboards.
- Memcached : Simple clé-valeur, pas de persistance. Idéal pour cache HTTP, objets.

Pattern cache-aside : L'application tente d'abord de lire depuis le cache. En cas de miss, elle lit depuis la base de données et met à jour le cache avec un TTL (*Time To Live*).

Stratégies de backup

Définition

Règle 3-2-1 : Stratégie de backup recommandée :

- **3** copies des données
- **2** supports/technologies différents
- **1** copie hors site (autre région/provider)

Services de backup managés

AWS Backup :

- Service centralisé pour EBS, RDS, DynamoDB, EFS, S3
- Policies de retention configurables
- Cross-region et cross-account copy
- Vault Lock pour conformité (immutabilité)

Azure Backup :

- Support VMs, SQL, Files, Blobs
- Retention jusqu'à 9999 jours
- Restauration granulaire (fichiers individuels)

GCP Backup and DR :

- Backup centralisé pour Compute Engine, GKE, databases
- Intégration avec management console

Comparaison approximative des coûts de stockage (par Go/mois, région EU) :

- Block SSD : AWS 0.10 / GCP 0.17 / Azure 0.12 euros
- Block HDD : AWS 0.045 / GCP 0.04 / Azure 0.05 euros
- Object Standard : AWS 0.023 / GCP 0.020 / Azure 0.021 euros
- Object Archive : AWS 0.004 / GCP 0.004 / Azure 0.002 euros
- File (NFS) : AWS 0.30 / GCP 0.20 / Azure 0.06 euros

Note : Le file storage est significativement plus cher. Évaluez si le partage est réellement nécessaire avant de l'adopter.

Résumé : choisir le bon stockage

« Decision tree » simplifié :

1. **Besoin de partage entre instances ?**
   - Oui : File Storage (NFS/SMB)
   - Non : Question 2
2.

Résumé : choisir le bon stockage

« Decision tree » simplifié :

1. **Besoin de partage entre instances ?**
   - Oui : File Storage (NFS/SMB)
   - Non : Question 2
2. **Accès via API HTTP suffisant ?**
   - Oui : Object Storage (S3/GCS)
   - Non : Question 3
3.

Résumé : choisir le bon stockage

« Decision tree » simplifié :

1. **Besoin de partage entre instances ?**
   - Oui : File Storage (NFS/SMB)
   - Non : Question 2
2. **Accès via API HTTP suffisant ?**
   - Oui : Object Storage (S3/GCS)
   - Non : Question 3
3. **Besoin de faible latence / IOPS élevés ?**
   - Oui : Block Storage SSD
   - Non : Block Storage HDD ou Object Storage

Exercice 1 : Architecture de stockage multi-tiers

Une application web doit stocker :

- Images uploadées par les utilisateurs (accès fréquent la première semaine, puis rare)
- Logs applicatifs (accès fréquent 24h, consultation occasionnelle ensuite)
- Base de données PostgreSQL (10000 transactions/seconde)
- Sessions utilisateurs (latence inférieure à 10ms requise)

Concevez l'architecture de stockage en spécifiant :

1. Le type de stockage pour chaque besoin
2. Les classes/tiers de stockage
3. Les lifecycle policies appropriées

Exercice 2 : Plan de disaster recovery

Votre entreprise héberge une application critique avec les contraintes suivantes :

- RPO : 1 heure maximum
- RTO : 4 heures maximum
- Données : 2 To de base de données, 10 To d'objets statiques

Proposez :

1. La stratégie de backup (fréquence, rétention)
2. L'architecture de réplication
3. La procédure de failover

Outline

Pourquoi l'Infrastructure as Code ?

Le problème : l'infrastructure manuelle

Scénario classique : un serveur de production tombe. Il faut le reconstruire.

Pourquoi l'Infrastructure as Code ?

Le problème : l'infrastructure manuelle

Scénario classique : un serveur de production tombe. Il faut le reconstruire.

- Qui se souvient de la configuration exacte ?

Pourquoi l'Infrastructure as Code ?

Le problème : l'infrastructure manuelle

Scénario classique : un serveur de production tombe. Il faut le reconstruire.

- Qui se souvient de la configuration exacte ?
- La documentation est-elle à jour ? (spoiler : non)

Pourquoi l'Infrastructure as Code ?

Le problème : l'infrastructure manuelle

Scénario classique : un serveur de production tombe. Il faut le reconstruire.

- Qui se souvient de la configuration exacte ?
- La documentation est-elle à jour ? (spoiler : non)
- Combien de temps pour tout recréer manuellement ?

Pourquoi l'Infrastructure as Code ?

L'infrastructure manuelle accumule de la dette technique invisible :

Pourquoi l'Infrastructure as Code ?

L'infrastructure manuelle accumule de la dette technique invisible : configurations oubliées,

Pourquoi l'Infrastructure as Code ?

L'infrastructure manuelle accumule de la dette technique invisible : configurations oubliées, dépendances non documentées,

Pourquoi l'Infrastructure as Code ?

L'infrastructure manuelle accumule de la dette technique invisible : configurations oubliées, dépendances non documentées, modifications ad-hoc jamais tracées.

Pourquoi l'Infrastructure as Code ?

Problèmes récurrents :

- **Erreurs humaines** : un clic de travers peut supprimer des données
- **Documentation obsolète** : personne ne met à jour le wiki
- **« Ça marche sur mon serveur »** : impossible de reproduire l'environnement
- **Reconstruction après incident** : des heures ou des jours de travail

Pourquoi l'Infrastructure as Code ?

La solution : l'IaC

Avec l'Infrastructure as Code :

- Infrastructure reproductible en quelques minutes
- Historique Git complet de toutes les modifications
- Code review avant tout changement critique
- Disaster recovery trivial : `terraform apply` et c'est reparti

Pourquoi l'Infrastructure as Code ?

Déclaratif vs Impératif

- **Déclaratif** : on décrit l'état final souhaité (Terraform, CloudFormation)
- **Impératif** : on décrit les étapes pour atteindre cet état (scripts shell, Ansible)
- Préférer le déclaratif : plus simple à maintenir et à raisonner

Idempotence

Définition

Une opération idempotente produit le même résultat si elle est exécutée une ou plusieurs fois.

Exemple : « créer un utilisateur admin » vs « s'assurer que l'utilisateur admin existe »

Terraform vérifie l'état actuel avant d'agir : si la ressource existe déjà avec la bonne configuration, il ne fait rien.

Reproductibilité

- Garantit que dev, staging et prod sont identiques
- Détecte le **configuration drift** : écart entre l'état décrit dans le code et l'état réel de l'infrastructure, typiquement causé par une modification manuelle hors Terraform
- Permet de recréer un environnement complet en quelques minutes

Versioning et collaboration

- Stockage du code dans Git : historique complet des modifications
- Code review obligatoire avant tout changement d'infrastructure
- Pipelines CI/CD pour valider et appliquer automatiquement les changements

Terraform : Vue d'ensemble

Pourquoi Terraform ?

Chaque cloud propose son outil natif d'IaC (CloudFormation pour AWS, Cloud Deployment Manager pour GCP, ARM/Bicep pour Azure), mais ces outils sont mono-fournisseur. Terraform s'est imposé pour les raisons suivantes :

- Support multi-cloud : AWS, GCP, Azure, et bien d'autres
- Large écosystème de providers (3000+)
- Communauté active et documentation abondante
- Langage HCL lisible et expressif

Depuis 2023, HashiCorp a changé la licence de Terraform (BSL). Un fork communautaire, **OpenTofu**, est maintenu par la Linux Foundation sous licence MPL 2.0. Les commandes et fichiers `.tf` restent compatibles (`tofu init`, `tofu plan`, `tofu apply`).

Terraform : Vue d'ensemble

Architecture de Terraform

- **Core** : moteur principal qui gère le state et le graphe de dépendances
- **Providers** : plugins qui communiquent avec les APIs des services (google, aws, azurerm...)
- **State** : source de vérité sur l'état actuel de l'infrastructure

Organisation standard des fichiers

```
projet/
├── main.tf           # Ressources principales
├── variables.tf      # Déclarations de variables
├── outputs.tf        # Valeurs exposées
├── versions.tf       # Versions Terraform et providers
├── terraform.tfvars  # Valeurs des variables (ne pas commiter
si secrets)
└── modules/          # Modules locaux
```

Fichier versions.tf

```
terraform {
  required_version = ">= 1.5.0"
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 5.0"
    }
  }
}
```

Toujours épingler les versions pour garantir la reproductibilité.

Fichier .gitignore

Attention

Certains fichiers ne doivent jamais être commités !

```
*.tfstate
*.tfstate.*
.terraform/
*.tfvars  # si contient des secrets
```

Blocs, attributs, références

HCL est un langage déclaratif conçu pour être lisible.

- **Blocs** : `resource`, `variable`, `output`, `data`, `locals`
- **Attributs** : `key = value`
- **Références** : `resource_type.name.attribute`

Configuration du provider

```
provider "aws" {
  region = var.region
}

provider "google" {
  project = var.gcp_project
  region  = var.region
}

provider "azurerm" {
  features {}
}
```

L'authentification dépend du provider : variables d'environnement `AWS_ACCESS_KEY_ID` / `AWS_SECRET_ACCESS_KEY` côté AWS, application default credentials (`gcloud auth application-default login`) côté GCP, `az login` côté Azure.

Ressources

```
resource "aws_instance" "web" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"

  tags = {
    Name = "web-server"
  }
}
```

Terraform calcule automatiquement les dépendances entre ressources, à partir du DAG qu'il construit en analysant les références d'attributs (`aws_subnet.main.id` lie l'instance au subnet).

Data sources

Un bloc `data` lit une ressource existante sans la gérer. Utile pour référencer ce qui est créé en dehors du projet Terraform (AMI publique, VPC partagé, secret externe).

```
data "aws_ami" "debian" {
  most_recent = true
  owners      = ["136693071363"]
  filter {
    name   = "name"
    values = ["debian-12-*"]
  }
}

resource "aws_instance" "web" {
  ami = data.aws_ami.debian.id
}
```

Variables

Variables, Outputs et Locals

```hcl
variable "instance_type" {
  description = "Type d'instance AWS"
  type        = string
  default     = "t2.micro"
}

variable "environment" {
  description = "Environnement (dev, staging, prod)"
  type        = string
  validation {
    condition     = contains(["dev", "staging", "prod"],
var.environment)
    error_message = "Environnement invalide."
  }
}

variable "db_password" {
  description = "Mot de passe base de données"
  type        = string
  sensitive   = true  # Masqué dans les logs
}
```

Outputs

```
output "instance_ip" {
  description = "IP publique de l'instance"
  value       = aws_instance.web.public_ip
}
```

Les outputs permettent d'exposer des valeurs pour d'autres modules ou pour l'utilisateur.

Locals

```
locals {
  project_prefix = "${var.project}-${var.environment}"
  common_labels = {
    environment = var.environment
    managed_by  = "terraform"
  }
}

resource "aws_instance" "web" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
  tags = merge(local.common_labels, {
    Name = "${local.project_prefix}-web"
  })
}
```

Les locals permettent de définir des valeurs calculées réutilisables. Utilisez-les pour éviter la répétition (DRY).

Création de ressources multiples

count : itération par index

```
resource "aws_instance" "web" {
  count         = 3
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
  tags = { Name = "web-${count.index}" }
}
```

Crée `web-0`, `web-1`, `web-2`.

Attention

Problème avec count : si vous supprimez l'élément du milieu, Terraform renomme tous les suivants et les recrée !

for_each : itération par clé

```
variable "instances" {
  default = {
    web    = "t2.micro"
    api    = "t2.small"
    worker = "t2.medium"
  }
}

resource "aws_instance" "server" {
  for_each      = var.instances
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = each.value
  tags = { Name = each.key }
}
```

Avec `for_each`, chaque ressource est identifiée par sa clé. Supprimer « api » ne touche pas « web » ni « worker ».

Création de ressources multiples

Quand utiliser quoi ?

- **count** : ressources identiques et interchangeables (ex: pool de workers)
- **for_each** : ressources avec identité propre (ex: environnements, services distincts)

En cas de doute, préférez `for_each`.

Dépendances implicites

Terraform détecte automatiquement les dépendances via les références :

```
resource "aws_vpc" "main" {
  cidr_block = "10.0.0.0/16"
}

resource "aws_instance" "web" {
  # Dépendance implicite : le VPC doit exister avant
l'instance
  subnet_id = aws_subnet.main.id
}
```

depends_on : dépendances explicites

```
resource "aws_instance" "app" {
  # ...
  depends_on = [aws_iam_role_policy.app_policy]
}
```

Note

Utiliser `depends_on` uniquement quand Terraform ne peut pas détecter la dépendance. Un excès de `depends_on` rend le code difficile à maintenir.

Bloc lifecycle

```
resource "aws_instance" "critical" {
  # ...
  lifecycle {
    # Crée la nouvelle avant de détruire l'ancienne (zero-
downtime)
    create_before_destroy = true

    # Empêche la suppression accidentelle
    prevent_destroy = true

    # Ignore les modifications manuelles sur certains
attributs
    ignore_changes = [tags, user_data]
  }
}
```

Cycle de vie des ressources

Cas d'usage

- `create_before_destroy` : bases de données, load balancers critiques
- `prevent_destroy` : ressources de production irremplaçables
- `ignore_changes` : tags gérés par un autre processus

Le fichier terraform.tfstate

Le state file contient l'état actuel de l'infrastructure connue par Terraform.

- Permet de calculer les différences entre l'état désiré et l'état réel
- Contient potentiellement des données sensibles (mots de passe, IPs…)

Attention

Ne jamais commiter le state file dans Git !

Backends et remote state

Un **backend** est l'endroit où Terraform persiste son state. Par défaut, le backend est local (fichier sur disque). En équipe, on utilise un backend distant : S3 sur AWS, GCS sur GCP, Azure Storage sur Azure. Chacun fournit un mécanisme de lock équivalent pour éviter qu'un `apply` se chevauche avec un autre.

```
terraform {
  backend "s3" {
    bucket         = "mon-projet-tfstate"
    key            = "terraform/state"
    region         = "eu-west-1"
    dynamodb_table = "tf-locks"  # lock via DynamoDB
  }
}
```

État Terraform

Sur GCS, le lock est natif via l'object lock. Sur Azure Storage, via le lease du blob.

Avantages du remote state :

- Travail en équipe possible
- Locking automatique (évite les `apply` simultanés)
- Backup et versioning intégrés

Commandes state

```
# Lister toutes les ressources dans le state
terraform state list

# Afficher les détails d'une ressource
terraform state show aws_instance.web

# Déplacer une ressource (renommage)
terraform state mv aws_instance.old aws_instance.new

# Retirer une ressource du state (sans la détruire)
terraform state rm aws_instance.manual
```

Attention

`terraform state rm` ne supprime pas la ressource réelle ! Elle devient simplement « invisible » pour Terraform.

Concept de module

Un module est un répertoire contenant des fichiers `.tf` réutilisables.

- **Inputs** : variables passées au module
- **Outputs** : valeurs exposées par le module

Utilisation d'un module

```
module "vpc" {
  source  = "terraform-aws-modules/vpc/aws"
  version = "~> 5.0"

  name = "my-vpc"
  cidr = "10.0.0.0/16"
}

# Référencer les outputs du module
resource "aws_instance" "web" {
  subnet_id = module.vpc.public_subnets[0]
}
```

Bonnes pratiques modules

- Un module par composant logique (réseau, compute, database...)
- Toujours épingler une version spécifique
- Documenter les variables avec leur description

Commandes essentielles

```
terraform init    # Télécharge providers et modules
terraform plan    # Prévisualise les changements
terraform apply   # Applique les changements
terraform destroy # Détruit toute l'infrastructure
```

Toujours lire attentivement le plan avant `apply` :

- `+` création, `-` destruction, `~` modification
- Attention aux destructions/recréations non prévues

Outils de validation

```
# Formate automatiquement le code HCL
terraform fmt -recursive

# Vérifie la syntaxe sans accéder au state
terraform validate
```

Debugging

```
# Activer les logs détaillés
export TF_LOG=DEBUG  # TRACE, DEBUG, INFO, WARN, ERROR
export TF_LOG_PATH="terraform.log"

terraform apply
```

Import de ressources existantes

```
# Importer une ressource existante dans le state
terraform import aws_instance.web i-1234567890abcdef0
```

Utile pour migrer une infrastructure manuelle vers IaC progressivement.

Concept

Les workspaces permettent de gérer plusieurs environnements avec le même code.

```
terraform workspace new staging
terraform workspace new prod
terraform workspace select staging
terraform workspace list
```

```
resource "aws_instance" "web" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
  tags = { Name = "web-${terraform.workspace}" }
}
```

Limitations

- Même backend, mêmes credentials pour tous les workspaces
- Pas d'isolation du state entre environnements
- Risque d'appliquer sur le mauvais workspace

Note

Pour des environnements vraiment isolés (credentials différents, comptes séparés), préférez des répertoires séparés par environnement.

Pourquoi automatiser les déploiements ?

L'IaC prend tout son sens avec l'automatisation :

- Code review obligatoire avant tout changement
- Plan visible dans la pull request
- Apply automatique après merge
- Traçabilité complète des modifications

Pipeline Terraform typique

À chaque push sur une branche : `fmt -check`, `validate`, `plan`. Le plan est posté en commentaire sur la pull request pour revue. Après approbation et merge sur `main`, le pipeline déclenche `apply` automatiquement. La séparation push / plan / review / apply garantit qu'aucun changement n'arrive en production sans avoir été lu et validé.

Exemple GitHub Actions (simplifié)

```yaml
# .github/workflows/terraform.yml
name: Terraform
on:
  push: { branches: [main] }
  pull_request: { branches: [main] }
jobs:
  terraform:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: hashicorp/setup-terraform@v3
      # fmt -check, init, validate, plan, apply...
```

En production, le pipeline complet inclut : authentification AWS (OIDC / IAM Roles), commentaire automatique du plan sur les PR, permissions granulaires, et apply conditionnel sur merge uniquement.

Bonnes pratiques CI/CD

Sécurité des pipelines :

- Utiliser Workload Identity Federation (pas de clés de service account)
- Séparer les credentials par environnement
- Limiter les permissions du service account au strict nécessaire
- Activer les branch protection rules

CI/CD pour l'infrastructure

Ne jamais faire :

- `terraform apply -auto-approve` sur une PR
- Stocker des secrets dans le code ou les logs
- Donner des droits admin au service account CI

Terraform Cloud / Terraform Enterprise

HashiCorp propose deux offres managées :

Terraform Cloud (SaaS, plan gratuit jusqu'à 500 ressources) : remote state intégré, exécution des `plan` et `apply` côté serveur, policy as code via Sentinel.

Terraform Enterprise (self-hosted) : même périmètre fonctionnel que Cloud, plus RBAC fin, SSO SAML/OIDC, audit log et déploiement on-premise pour les contraintes de souveraineté.

Patterns recommandés

Ressources conditionnelles avec `count = condition ? 1 : 0` permettent d'activer/désactiver une ressource selon une variable, sans dupliquer le code :

```
resource "aws_eip" "static" {
  count  = var.create_static_ip ? 1 : 0
  domain = "vpc"
}
```

Composition : factoriser les patterns récurrents (réseau, app stateless, base de données) dans des modules locaux ou un registre interne, et instancier avec `for_each` pour gérer les environnements.

Anti-patterns à éviter

- **Secrets en dur dans le code** : un secret commité dans Git reste dans l'historique même après suppression. Passer par un secret manager (AWS Secrets Manager, Vault, GCP Secret Manager) ou des variables d'environnement injectées au runtime.

Anti-patterns à éviter

- **Secrets en dur dans le code** : un secret commité dans Git reste dans l'historique même après suppression. Passer par un secret manager (AWS Secrets Manager, Vault, GCP Secret Manager) ou des variables d'environnement injectées au runtime.
- **Modifications manuelles sur des ressources gérées** : provoque du drift. Au prochain `apply`, Terraform écrasera la modification ou refusera de partir d'un état incohérent. Toute modification doit passer par le code.

Anti-patterns à éviter

- **Secrets en dur dans le code** : un secret commité dans Git reste dans l'historique même après suppression. Passer par un secret manager (AWS Secrets Manager, Vault, GCP Secret Manager) ou des variables d'environnement injectées au runtime.
- **Modifications manuelles sur des ressources gérées** : provoque du drift. Au prochain `apply`, Terraform écrasera la modification ou refusera de partir d'un état incohérent. Toute modification doit passer par le code.
- **Copy-paste entre environnements** : deux blocs `resource` quasi-identiques pour dev et prod divergent inévitablement. Factoriser en module avec des variables.

Anti-patterns à éviter

- **Secrets en dur dans le code** : un secret commité dans Git reste dans l'historique même après suppression. Passer par un secret manager (AWS Secrets Manager, Vault, GCP Secret Manager) ou des variables d'environnement injectées au runtime.
- **Modifications manuelles sur des ressources gérées** : provoque du drift. Au prochain `apply`, Terraform écrasera la modification ou refusera de partir d'un état incohérent. Toute modification doit passer par le code.
- **Copy-paste entre environnements** : deux blocs `resource` quasi-identiques pour dev et prod divergent inévitablement. Factoriser en module avec des variables.
- **State file commité dans Git** : conflits de merge ingérables, secrets potentiellement exposés, pas de locking. Utiliser un backend distant dès qu'on est plus d'un.

Anti-patterns à éviter

- **Secrets en dur dans le code** : un secret commité dans Git reste dans l'historique même après suppression. Passer par un secret manager (AWS Secrets Manager, Vault, GCP Secret Manager) ou des variables d'environnement injectées au runtime.
- **Modifications manuelles sur des ressources gérées** : provoque du drift. Au prochain `apply`, Terraform écrasera la modification ou refusera de partir d'un état incohérent. Toute modification doit passer par le code.
- **Copy-paste entre environnements** : deux blocs `resource` quasi-identiques pour dev et prod divergent inévitablement. Factoriser en module avec des variables.
- **State file commité dans Git** : conflits de merge ingérables, secrets potentiellement exposés, pas de locking. Utiliser un backend distant dès qu'on est plus d'un.
- **Abus de provisioners** (`local-exec`, `remote-exec`) : non idempotents, fragiles, ne s'intègrent pas au DAG. Préférer cloud-init / user-data pour la configuration au boot, ou un outil dédié (Ansible) en aval.

Anti-patterns à éviter

- **Secrets en dur dans le code** : un secret commité dans Git reste dans l'historique même après suppression. Passer par un secret manager (AWS Secrets Manager, Vault, GCP Secret Manager) ou des variables d'environnement injectées au runtime.
- **Modifications manuelles sur des ressources gérées** : provoque du drift. Au prochain `apply`, Terraform écrasera la modification ou refusera de partir d'un état incohérent. Toute modification doit passer par le code.
- **Copy-paste entre environnements** : deux blocs `resource` quasi-identiques pour dev et prod divergent inévitablement. Factoriser en module avec des variables.
- **State file commité dans Git** : conflits de merge ingérables, secrets potentiellement exposés, pas de locking. Utiliser un backend distant dès qu'on est plus d'un.
- **Abus de provisioners** (`local-exec`, `remote-exec`) : non idempotents, fragiles, ne s'intègrent pas au DAG. Préférer cloud-init / user-data pour la configuration au boot, ou un outil dédié (Ansible) en aval.

Sécurité spécifique à l'IaC

Les principes généraux (moindre privilège, gestion des secrets, séparation des environnements) sont traités dans le chapitre Sécurité Cloud. Trois sujets sont spécifiques à Terraform :

Scanning statique : **tfsec**, **Checkov** et **Trivy** analysent les fichiers `.tf` avant déploiement pour repérer les mauvaises configurations (bucket public, chiffrement manquant, port ouvert au monde, IAM trop permissif).

```
tfsec .
checkov -d .
```

Sécurité spécifique à l'IaC

Les principes généraux (moindre privilège, gestion des secrets, séparation des environnements) sont traités dans le chapitre Sécurité Cloud. Trois sujets sont spécifiques à Terraform :

Scanning statique : **tfsec**, **Checkov** et **Trivy** analysent les fichiers `.tf` avant déploiement pour repérer les mauvaises configurations (bucket public, chiffrement manquant, port ouvert au monde, IAM trop permissif).

```
tfsec .
checkov -d .
```

Drift detection : `terraform plan` détecte les écarts entre le state et la réalité. À exécuter régulièrement en CI sur les environnements critiques pour détecter les modifications manuelles non tracées.

Sécurité spécifique à l'IaC

Les principes généraux (moindre privilège, gestion des secrets, séparation des environnements) sont traités dans le chapitre Sécurité Cloud. Trois sujets sont spécifiques à Terraform :

Scanning statique : **tfsec**, **Checkov** et **Trivy** analysent les fichiers `.tf` avant déploiement pour repérer les mauvaises configurations (bucket public, chiffrement manquant, port ouvert au monde, IAM trop permissif).

```
tfsec .
checkov -d .
```

Drift detection : `terraform plan` détecte les écarts entre le state et la réalité. À exécuter régulièrement en CI sur les environnements critiques pour détecter les modifications manuelles non tracées.

Verrous au niveau du code : marquer les ressources critiques avec `lifecycle { prevent_destroy = true }`, marquer les variables sensibles avec `sensitive = true` pour éviter qu'elles apparaissent dans les logs de `plan`.

Alternatives à Terraform

Solutions natives cloud

- **CloudFormation** (AWS), **Cloud Deployment Manager** (GCP), **ARM/Bicep** (Azure)
- Avantage : intégration native, pas de state à gérer (le state est dans le service)
- Inconvénient : vendor lock-in, pas de multi-cloud

Alternatives à Terraform

Pulumi

Pulumi reprend le modèle Terraform (state, providers, DAG) mais remplace HCL par un vrai langage de programmation : TypeScript, Python, Go, C#. Permet d'utiliser boucles, classes, imports de bibliothèques. Plus puissant mais aussi plus facile à mal écrire.

CDK et CDK for Terraform

L'AWS CDK génère du CloudFormation à partir de code TypeScript/Python. CDKTF fait pareil pour Terraform. Approche utile quand l'équipe est déjà à l'aise avec un langage généraliste et veut éviter le DSL.

Crossplane

Crossplane déplace l'IaC dans Kubernetes : les ressources cloud deviennent des objets Kubernetes (CRD) réconciliés par des contrôleurs. Cohérent quand toute la plateforme tourne déjà sur K8s, mais ajoute la complexité de Kubernetes pour les équipes qui ne l'utilisent pas par ailleurs.

Alternatives à Terraform

Ansible

- **Terraform** : crée l'infrastructure (VMs, réseaux...)
- **Ansible** : configure ce qui tourne sur l'infrastructure
- Souvent utilisés ensemble : Terraform crée, Ansible configure

Outline

License du cours

Crédits

Cours créé par Hugo Blanc pour l'Université Lyon 1, ESSIR.

Contact : hugo.blanc@univ-lyon1.fr

Ressources et références

Les marques citées (AWS, GCP, Azure, etc.) appartiennent à leurs propriétaires respectifs.

Les exemples de code sont fournis à titre éducatif uniquement.