Déployer Automatiquement une Infrastructure

Table des matières

Module 1 : Introduction au Cloud Computing

Objectifs pédagogiques

À la fin de ce module, l'apprenant sera capable de :

- Comprendre ce qu'est le Cloud Computing.
- Identifier les modèles de services (IaaS, PaaS, SaaS).
- Distinguer les types de déploiement Cloud : Public, Privé, Hybride.
- Évaluer les bénéfices et les limites du Cloud Computing.

1.1 Qu'est-ce que le Cloud Computing ?

Définition :
Le *Cloud Computing* (informatique en nuage) est un modèle permettant l'accès à des ressources informatiques (serveurs, stockage, bases de données, mise en réseau, logiciels) à la demande, via Internet, et facturées à l'usage.

Caractéristiques principales :

- **Élasticité :** ajout/suppression de ressources selon les besoins.
- **Accès à distance :** via Internet, de n'importe où.
- **Paiement à l'usage :** pas de gros investissements initiaux.
- **Provisionnement rapide :** déploiement en quelques clics ou lignes de commande.

1.2 Les modèles de service Cloud

Modèle	Description	Exemples
IaaS ***(Infrastructure as a Service)***	Fourniture de ressources informatiques de base (VM, stockage, réseau).	AWS EC2, Azure VM, Google Compute Engine
PaaS ***(Platform as a Service)***	Fourniture d'un environnement de développement complet (OS + serveur + runtime).	Heroku, Google App Engine, Azure App Service
SaaS ***(Software as a Service)***	Accès à des applications via un navigateur, sans gestion d'infrastructure.	Gmail, Microsoft 365, Salesforce

Illustration :

```
Utilisateur
│
├── SaaS : utilise l'application (ex : Gmail)
├── PaaS : développe une appli sans gérer les serveurs (ex : App Engine)
└── IaaS : déploie sa propre infra (ex : VM Linux sur AWS EC2)
```

🌍 1.3 Types de déploiement Cloud

Type de Cloud	Description	Avantages	Exemple
Public	Mutualisé entre plusieurs clients, géré par un fournisseur.	Coût réduit, maintenance gérée.	AWS, GCP
Privé	Infrastructure dédiée à une organisation.	Sécurité, personnalisation.	OpenStack sur site
Hybride	Mix de cloud privé + public.	Flexibilité, meilleure gestion des données sensibles.	VM on-prem + AWS

✅ 1.4 Avantages et inconvénients du Cloud

✅ Avantages :

- Réduction des coûts (CAPEX → OPEX)
- Évolutivité rapide
- Sauvegardes automatiques et haute disponibilité
- Accès global

❌ Inconvénients :

- Dépendance à Internet
- Sécurité et confidentialité à gérer
- Coût cumulatif si mal contrôlé
- Verrouillage fournisseur (*vendor lock-in*)

💡 1.5 Exemples concrets

Cas 1 : Startup web

Une startup déploie son site e-commerce via **Heroku (PaaS)**, avec une base **PostgreSQL** managée. Elle monte en charge sans se soucier de l'infrastructure.

Cas 2 : Multinationale

Une entreprise utilise **AWS EC2 (IaaS)** pour héberger ses applications métiers, avec un réseau privé virtuel (VPC) pour garantir la sécurité des données.

🧠 Exercice pratique

But : Identifier les types de service Cloud dans un contexte réel.

Énoncé :
Vous êtes consultant pour une PME. Elle utilise :

- Dropbox pour le stockage collaboratif.
- WordPress.com pour son blog d'entreprise.
- Une application maison déployée sur Azure App Service.

Question : Associez chaque service à son modèle :

1. Dropbox → ?
2. WordPress.com → ?
3. Azure App Service → ?

Correction :

1. **Dropbox** → SaaS
2. **WordPress.com** → SaaS
3. **Azure App Service** → PaaS

Synthèse du Module

- Le Cloud Computing permet de consommer des ressources informatiques via Internet.
- Trois modèles de services : IaaS, PaaS, SaaS.
- Trois types de déploiement : Public, Privé, Hybride.
- Il offre élasticité, flexibilité, mais demande une bonne gestion de la sécurité et des coûts.

Module 2 : Les solutions de Cloud Computing

Objectifs pédagogiques

À la fin de ce module, l'apprenant sera capable de :

- Identifier et comparer les principaux fournisseurs de services Cloud.
- Connaître les solutions proposées par chaque fournisseur.
- Comprendre les cas d'utilisation typiques pour chaque solution.
- Choisir la meilleure solution Cloud en fonction d'un besoin précis.

2.1 Introduction aux principaux fournisseurs de services Cloud

Les principaux acteurs du marché :

1. **Amazon Web Services (AWS)**
   - **Leader du marché**, propose une large gamme de services Cloud.
   - **Historique :** Lancement en 2006.
   - **Forces :** Évolutivité, large écosystème, services variés (IA, Big Data, IoT, etc.).
   - **Exemples de services :** EC2 (IaaS), S3 (stockage), RDS (bases de données), Lambda (serverless).
2. **Microsoft Azure**
   - **Second acteur majeur**, souvent préféré des entreprises ayant des infrastructures Windows.
   - **Historique :** Lancé en 2010.
   - **Forces :** Intégration avec les produits Microsoft (Windows Server, Active Directory, etc.), solutions hybrides.
   - **Exemples de services :** Azure VMs (IaaS), Azure Blob Storage (stockage), Azure Functions (serverless).
3. **Google Cloud Platform (GCP)**
   - **Expertise en Data Analytics et IA**.
   - **Historique :** Lancé en 2008.
   - **Forces :** Excellente infrastructure de données, Kubernetes Engine (GKE), Google BigQuery (analyse de données).
   - **Exemples de services :** Compute Engine (IaaS), Cloud Functions (serverless), BigQuery (analyse de données).

4. **IBM Cloud**
    - Propose des solutions en **cloud hybride** et des services d'**intelligence artificielle**.
    - **Forces :** Solutions pour entreprises, Watson AI, cloud privé.
    - **Exemples de services :** IBM Cloud VMs, IBM Watson (IA).
5. **Oracle Cloud**
    - Spécialisé dans les **bases de données** et les **solutions d'entreprise**.
    - **Forces :** Bases de données optimisées pour les applications d'entreprise.
    - **Exemples de services :** Oracle Autonomous Database, Oracle Cloud Infrastructure (OCI).

🔍 2.2 Comparaison des solutions Cloud

Critère	AWS	Azure	Google Cloud
Market Share	Leader	Deuxième	Troisième
Force	Évolutivité, large écosystème	Intégration Microsoft, hybridité	Analyse de données, IA, GKE
Support de conteneurs	ECS, EKS	AKS	GKE
Services de stockage	S3, EBS, Glacier	Blob Storage, Disk	Cloud Storage
Compute	EC2, Lambda	Azure VM, Azure Functions	Compute Engine, Cloud Functions
Base de données	RDS, Aurora	SQL Database, Cosmos DB	Cloud SQL, BigQuery

---

💡 2.3 Cas d'utilisation typiques pour chaque fournisseur

1. AWS (Amazon Web Services)

- **Cas d'utilisation :**
    - **Applications évolutives** : Déploiement d'une application web à grande échelle.
    - **Stockage** : Stockage de données non structurées avec S3.
    - **Big Data & IA** : Traitement de données massives avec Amazon EMR et SageMaker.
- **Exemple concret :** Une entreprise de e-commerce utilise AWS pour héberger son application (EC2), stocker ses images produits (S3), et analyser ses données de clients (Redshift, Athena).

2. Microsoft Azure

- **Cas d'utilisation :**
    - **Solutions hybrides** : Connexion de data centers privés avec Azure (via ExpressRoute).
    - **Applications Windows** : Intégration fluide avec Windows Server et SQL Server.
    - **Développement d'applications Web** : Utilisation de **Azure App Service** pour héberger des applications web.
- **Exemple concret :** Une entreprise avec une infrastructure Windows utilise Azure pour gérer sa messagerie (Exchange), déployer des applications internes sur **App Service**, et héberger des machines virtuelles (VM).

3. Google Cloud Platform

- **Cas d'utilisation :**
    - **Big Data** : Traitement des données à grande échelle avec **BigQuery**.
    - **Machine Learning & IA** : Développement de modèles IA avec **TensorFlow** et **AI Platform**.
    - **Conteneurs** : Orchestration de conteneurs avec **Kubernetes Engine (GKE)**.

- **Exemple concret :** Une entreprise de marketing utilise **BigQuery** pour analyser de grandes quantités de données clients, et **GKE** pour déployer des applications conteneurisées.

⚖️ 2.4 Choisir la meilleure solution Cloud

Le choix du fournisseur dépend de plusieurs critères :

- **Budget :** Comparer les coûts (paiement à l'usage).
- **Besoins techniques :** Fonctionnalités spécifiques comme l'IA, les conteneurs, le Big Data, etc.
- **Infrastructure existante :** Si l'entreprise utilise déjà des solutions Microsoft, Azure peut être plus simple à intégrer.
- **Sécurité et conformité :** Choisir un fournisseur qui respecte les normes de sécurité et de conformité nécessaires pour l'entreprise (ex : GDPR, ISO).

❓ Exercice pratique

But : Analyser un besoin et choisir un fournisseur Cloud adapté.

Énoncé :
Une entreprise de développement d'applications mobiles cherche à déployer son backend sur le Cloud. Elle a besoin :

- D'un environnement évolutif.
- D'un service de base de données relationnelle.
- D'une solution de stockage d'images et vidéos.
- D'une API de machine learning.

Question :

- Quel fournisseur Cloud recommanderiez-vous et pourquoi ?
- Quelles solutions spécifiques proposeriez-vous ?

Correction :

- **Fournisseur recommandé : AWS**
    - **Services proposés :**
        - **EC2 (IaaS)** pour héberger l'application backend.
        - **RDS** pour la base de données relationnelle.
        - **S3** pour stocker les images et vidéos.
        - **SageMaker** pour l'API machine learning.

Module 3 : L'architecture système et son déploiement

Objectifs pédagogiques

À la fin de ce module, l'apprenant sera capable de :

- Comprendre les principes de base d'une architecture système.
- Identifier les composants d'une architecture système typique.
- Concevoir une architecture adaptée aux besoins d'une entreprise ou d'un projet.
- Déployer et gérer l'architecture en utilisant des outils Cloud.

3.1 Introduction à l'architecture système

Qu'est-ce qu'une architecture système ?

Une **architecture système** désigne la structure globale d'un système informatique, qui comprend les différents composants matériels, logiciels et réseaux qui interagissent pour accomplir des tâches spécifiques. Cette architecture doit répondre aux exigences de performance, de sécurité, de disponibilité et de maintenabilité du système.

3.2 Composants d'une architecture système

Une architecture typique est composée de plusieurs éléments qui travaillent ensemble pour offrir des services complets.

1. Composants matériels (Infrastructure) :

- **Serveurs** : Machines physiques ou virtuelles sur lesquelles les applications sont déployées.
- **Stockage** : Disques durs ou solutions cloud (ex : AWS S3, Google Cloud Storage) pour le stockage de données.
- **Réseau** : Composants réseaux (switches, routeurs, VPN) pour la communication entre les serveurs.

2. Composants logiciels :

- **Systèmes d'exploitation** : Windows, Linux, ou autres systèmes de gestion d'infrastructure.
- **Serveurs d'applications** : Serveurs web (Apache, Nginx), serveurs d'applications (Tomcat, JBoss).
- **Bases de données** : MySQL, PostgreSQL, MongoDB, etc.
- **Middleware** : Logiciels qui assurent la communication entre différents systèmes ou services.

3. Composants de réseau :

- **Équilibrage de charge (Load balancing)** : Répartition du trafic réseau entre plusieurs serveurs pour améliorer la performance.
- **Pare-feu** : Sécurisation du réseau en filtrant les connexions non autorisées.
- **VPC (Virtual Private Cloud)** : Réseau virtuel isolé dans le Cloud (ex : AWS VPC).

🔍 3.3 Types d'architectures système

1. Architecture 3-tiers

Composants :

- **Tier 1 : Présentation (Front-end)** – Ce niveau gère l'interface utilisateur. Il peut s'agir d'un site web, d'une application mobile, etc.
- **Tier 2 : Logique (Middle-end)** – Ce niveau comprend les serveurs d'applications qui traitent la logique métier.
- **Tier 3 : Données (Back-end)** – Ce niveau regroupe les bases de données qui stockent les informations nécessaires.

Avantages :

- Séparation claire des responsabilités.
- Scalabilité et maintenance facilitées.

Exemple : Une application de e-commerce où le front-end est géré par un serveur web (NGINX), la logique métier par une API, et la base de données par MySQL.

---

2. Architecture microservices

Composants :

- Chaque service est déployé indépendamment avec ses propres ressources (base de données, API, etc.).
- Communication via des APIs (REST, gRPC).
- Chaque microservice peut être scalé indépendamment.

Avantages :

- Flexibilité et scalabilité.
- Déploiement indépendant des services.
- Résilience : si un service échoue, les autres restent fonctionnels.

Exemple : Une plateforme de streaming vidéo avec des microservices dédiés à la gestion des utilisateurs, du catalogue de vidéos, des paiements, etc.

---

3. Architecture serverless

Composants :

- Les applications sont composées de fonctions qui s'exécutent en réponse à des événements (ex : AWS Lambda).
- Pas besoin de gérer l'infrastructure.

Avantages :

- Pas de gestion d'infrastructure.
- Pay-as-you-go, ce qui réduit les coûts.
- Rapidité de déploiement.

🛠 3.4 Déploiement de l'architecture système

1. Conception du déploiement

La conception du déploiement inclut :

- **Choix de l'infrastructure** (cloud public, cloud privé, ou hybride).
- **Choix des services** : comme EC2 (AWS), VMs (Azure), ou Kubernetes (pour des microservices).
- **Décision sur la résilience** : mise en place de stratégies de haute disponibilité (multi-zone, multi-régions).
- **Surveillance et alertes** : configuration de CloudWatch (AWS) ou Azure Monitor pour surveiller les systèmes.

---

2. Processus de déploiement

Le déploiement d'une architecture nécessite généralement les étapes suivantes :

1. **Préparation de l'environnement** :
   - Création des VPC, sous-réseaux, groupes de sécurité.
   - Configuration des instances de calcul et des services nécessaires (bases de données, stockage, etc.).
2. **Automatisation** :
   - Utilisation de **Terraform** ou **CloudFormation** pour décrire l'infrastructure comme du code.
   - Déploiement des configurations avec **Ansible** ou **Chef** pour automatiser la gestion des serveurs.
3. **Test** :
   - Déploiement dans un environnement de staging pour tester la scalabilité, la sécurité et les performances.
   - Utilisation d'outils comme **JMeter** ou **Gatling** pour tester la charge du système.
4. **Production** :
   - Déploiement de l'architecture sur l'environnement de production.
   - Mise en place des mécanismes de mise à l'échelle (autoscaling) et de mise à jour continue (CI/CD).

🔧 3.5 Outils et technologies pour le déploiement

- **Terraform** : Infrastructure as Code pour créer et gérer des ressources Cloud (ex : VMs, bases de données, VPCs).
- **Ansible** : Automatisation des tâches de configuration des serveurs (ex : déploiement d'une stack LAMP).
- **Docker** et **Kubernetes** : Déploiement de microservices conteneurisés.
- **CI/CD** : Intégration continue et déploiement continu avec **Jenkins**, **GitLab CI**, **CircleCI** pour automatiser le processus de mise à jour de l'architecture.

❓ Exercice pratique

But : Concevoir et déployer une architecture 3-tiers pour une application web simple.

Énoncé :
Une entreprise souhaite héberger une application web de gestion de tâches en ligne. L'application doit être composée des trois niveaux suivants :

- **Front-end** : Interface web simple.
- **Back-end** : API REST pour la gestion des tâches.
- **Base de données** : Stockage des informations sur les tâches et les utilisateurs.

Tâches à réaliser :

1. Décrire l'architecture en termes de services Cloud (choisir AWS, Azure, ou GCP).
2. Concevoir le déploiement en utilisant Terraform ou CloudFormation.
3. Configurer l'autoscaling pour les services web.

Correction possible :

- **Front-end** : Serveur web statique sur S3 (AWS).
- **Back-end** : API REST sur **EC2** (AWS) avec un load balancer **ELB**.
- **Base de données** : **RDS** avec **MySQL**.

Contexte :

Une entreprise souhaite développer une application de gestion de tâches en ligne qui sera accessible via un navigateur web. Cette application doit permettre aux utilisateurs de créer, consulter, modifier et supprimer des tâches.
L'architecture doit être divisée en trois niveaux (tiers) :

- **Front-end** (interface utilisateur)
- **Back-end** (logique métier)
- **Base de données** (stockage des données des tâches et utilisateurs)

Objectifs de l'exercice :

1. Concevoir l'architecture de l'application en utilisant une architecture **3-tiers**.
2. Déployer l'architecture en utilisant des services Cloud (par exemple AWS, Azure ou GCP).
3. Configurer l'autoscaling pour la montée en charge du front-end et du back-end.
4. Créer une base de données pour stocker les informations de l'application.

Étape 1 : Conception de l'architecture

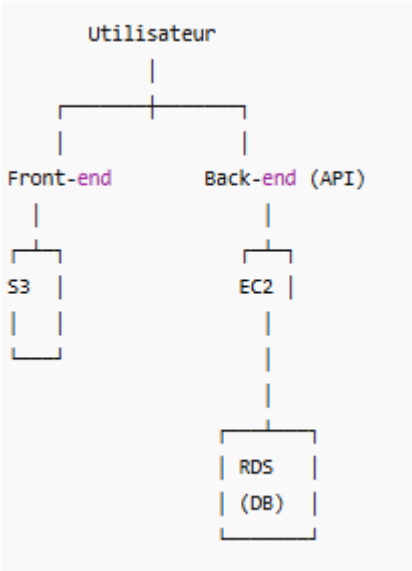
1.1 Structure de l'architecture

Une architecture **3-tiers** consiste en trois couches séparées :

- **Tier 1 : Front-end (Interface Utilisateur)**
  - Serveur web qui servira des fichiers statiques pour l'application web.
  - Technologie : par exemple, des pages HTML et CSS avec JavaScript (React, Angular, Vue.js).
  - Hébergement : Utilisation de **S3 (AWS)** ou **Blob Storage (Azure)** pour l'hébergement des fichiers statiques, ou un **EC2 (AWS)** pour déployer une application plus dynamique.
- **Tier 2 : Back-end (API et logique métier)**
  - Une API REST pour gérer la logique métier et les interactions avec la base de données.
  - Technologie : Node.js, Python (Flask/Django), Ruby on Rails, etc.
  - Hébergement : Utilisation de **EC2 (AWS)** ou **App Service (Azure)** pour héberger l'application back-end.
  - **Autoscaling** : Configurer un **Elastic Load Balancer (ELB)** et un **Auto Scaling Group (ASG)** (AWS) ou **Azure App Service Scaling** pour gérer la montée en charge automatique.
- **Tier 3 : Base de données (Stockage des données)**
  - Base de données relationnelle pour stocker les informations des tâches et des utilisateurs.
  - Technologie : MySQL, PostgreSQL, etc.
  - Hébergement : Utilisation de **RDS (AWS)**, **Azure SQL Database**, ou **Cloud SQL (GCP)**.

1.2 Diagramme de l'architecture

Voici un diagramme de l'architecture à concevoir :



Étape 2 : Déploiement de l'architecture sur le Cloud

2.1 Choix du fournisseur Cloud

Choisissez un fournisseur Cloud pour déployer votre architecture. Par exemple, pour cet exercice, nous utiliserons **AWS**. Le même principe peut être appliqué à **Azure** ou **GCP**.

2.2 Détails des services à utiliser :

- **Front-end :**
  - **Hébergement des fichiers statiques** : Utilisation de **S3 (Simple Storage Service)** d'AWS.
  - Si l'application nécessite une logique front-end plus complexe, vous pouvez utiliser un **EC2** pour exécuter un serveur web (ex : Nginx ou Apache).
- **Back-end :**
  - **Serveur d'API** : Déploiement sur **EC2** avec une application Node.js (par exemple).
  - **Autoscaling et équilibrage de charge** : Utilisation d'un **Elastic Load Balancer (ELB)** pour répartir les requêtes HTTP entre les instances EC2 et mise en place d'un **Auto Scaling Group (ASG)** pour ajouter ou supprimer des instances en fonction de la demande.
- **Base de données :**
  - **Base de données relationnelle** : Utilisation de **RDS** (Amazon Relational Database Service) pour gérer une base de données **MySQL** ou **PostgreSQL**.

2.3 Création des ressources sur AWS :

- **Step 1** : Créer un bucket **S3** pour héberger les fichiers front-end.
- **Step 2** : Créer une instance **EC2** avec Node.js ou Python comme back-end, déployer l'API.
- **Step 3** : Configurer un **Elastic Load Balancer (ELB)** pour distribuer les requêtes.
- **Step 4** : Créer une base de données **RDS** et connecter votre API à cette base de données.

Étape 3 : Automatisation du déploiement avec Terraform

3.1 Écrire des scripts Terraform pour déployer l'architecture automatiquement.

Terraform permet de définir l'infrastructure comme du code. Voici un exemple de base pour déployer une instance EC2, un bucket S3, et une base de données RDS.

```
# Déploiement d'un Bucket S3 pour le front-end
resource "aws_s3_bucket" "frontend" {
  bucket = "my-app-frontend-bucket"
  acl    = "public-read"
}

# Déploiement d'une instance EC2 pour le back-end
resource "aws_instance" "backend" {
  ami           = "ami-12345678" # Remplacer par l'AMI de votre choix
  instance_type = "t2.micro"

  # Connexion à un VPC et sous-réseau
  network_interface {
    network_interface_id = aws_network_interface.example.id
    device_index         = 0
  }

  tags = {
    Name = "Backend Server"
  }
}

# Déploiement d'une base de données RDS (MySQL)
resource "aws_db_instance" "db" {
  allocated_storage = 20
  db_instance_class = "db.t2.micro"
  engine            = "mysql"
  username          = "admin"
  password          = "password"
  db_name           = "tasks_db"
  multi_az          = true
  publicly_accessible = true
}
```

3.2 Exécution de Terraform :

Exécutez les commandes Terraform pour déployer l'architecture :

1. **`terraform init`** pour initialiser le projet.
2. **`terraform plan`** pour visualiser le plan de déploiement.
3. `terraform apply` pour créer les ressources dans le Cloud.

Étape 4 : Tests et mise en production

4.1 Tester l'architecture en environnement de staging :

- Vérifiez que le front-end est accessible via le bucket S3 ou EC2.
- Testez l'API back-end pour vous assurer que les appels à la base de données fonctionnent correctement.
- Effectuez un test de montée en charge pour vérifier l'autoscaling en augmentant le trafic vers l'API.

4.2 Mise en production :

Une fois les tests terminés et validés, déployez l'architecture en production et configurez la surveillance et les alertes (par exemple avec **CloudWatch** pour AWS).

Résultat attendu :

À la fin de cet exercice, vous devriez avoir une architecture Cloud fonctionnelle composée de :

- Un front-end hébergé sur **S3** ou **EC2**.
- Un back-end déployé sur une instance **EC2** avec un équilibrage de charge (ELB) et autoscaling configuré.
- Une base de données **RDS** gérant les données des utilisateurs et des tâches.

Correction possible :

- **Architecture** : Front-end sur **S3**, Back-end sur **EC2**, Base de données sur **RDS**.
- **Scalabilité** : Configuration de **Auto Scaling** et **Elastic Load Balancer** pour gérer la montée en charge.
- **Terraform** : Scripts permettant de créer les ressources nécessaires pour chaque composant.

Module 4 : La configuration des serveurs

Objectifs pédagogiques

À la fin de ce module, l'apprenant sera capable de :

- Comprendre les étapes nécessaires à la configuration d'un serveur.
- Mettre en place une configuration initiale (réseau, pare-feu, comptes).
- Installer les logiciels nécessaires selon le rôle du serveur.
- Sécuriser un serveur de manière basique.
- Automatiser la configuration à l'aide d'outils (Ansible, scripts shell, cloud-init).

4.1 Qu'est-ce que la configuration d'un serveur ?

La configuration d'un serveur consiste à préparer une machine (physique ou virtuelle) pour qu'elle puisse remplir son rôle dans une infrastructure. Cela inclut :

- L'installation du système d'exploitation,
- Le paramétrage du réseau,
- La gestion des utilisateurs et des droits,
- La configuration des logiciels et des services (web, base de données, etc.),
- La mise en place de la sécurité de base.

4.2 Types de serveurs courants

Chaque serveur a un rôle spécifique, ce qui influence sa configuration.

Type de serveur	Description
Serveur Web	Sert des pages web via HTTP/HTTPS (ex : Apache, NGINX)
Serveur d'applications	Exécute la logique métier d'une application (ex : Node.js, Tomcat)
Serveur de base de données	Gère les données et permet l'accès aux applications (ex : MySQL, PostgreSQL)
Serveur de fichiers	Partage des fichiers dans un réseau (ex : Samba, NFS)
Serveur DNS / DHCP	Gère la résolution de noms ou la distribution des adresses IP

4.3 Étapes de la configuration d'un serveur

4.3.1 Installation du système d'exploitation

- Choisir un OS selon le besoin : Ubuntu Server, CentOS, Debian, Windows Server, etc.
- Automatiser l'installation avec des **images customisées** ou des outils comme **cloud-init**, **Kickstart**, ou **PXE boot**.

4.3.2 Configuration réseau

- Attribuer une **adresse IP fixe** (souvent dans le cloud, c'est via DHCP statique).
- Définir la **passerelle**, **serveurs DNS**, et les **routes** éventuelles.
- Outils : `nmcli` (Linux), `netplan` (Ubuntu), ou panneau de configuration Windows.

4.3.3 Gestion des utilisateurs et accès

- Création des utilisateurs avec des rôles spécifiques (`adduser`, `usermod`).
- Mise en place de l'accès SSH :
  - Désactivation de l'accès root direct,
  - Ajout de clés publiques (authentification par clé SSH),
  - Restriction d'accès avec `AllowUsers` dans `/etc/ssh/sshd_config`.

4.3.4 Sécurité de base

- Mise à jour du système (`apt upgrade`, `yum update`, `Windows Update`).
- Configuration du **pare-feu** :
  - Outils : `ufw` (Ubuntu), `firewalld`, `iptables`, ou pare-feu Windows.
  - Ouverture uniquement des ports nécessaires (HTTP, HTTPS, SSH, etc.).
- Installation de **Fail2Ban** ou équivalent pour bloquer les attaques par force brute.
- Configuration du **journal système** (journald, syslog) et surveillance des logs.

⚙ 4.3.5 Installation des services et logiciels

Exemples :

- **Serveur web** : `sudo apt install nginx` ou `yum install httpd`
- **Base de données** : `apt install mysql-server` puis sécurisation via `mysql_secure_installation`
- **Langages/Environnements** : `python3`, `nodejs`, `php`, etc.
- **Services systèmes** : Configuration avec `systemctl` pour activer les services au démarrage.

4.4 Automatisation de la configuration

4.4.1 Scripts shell

```
#!/bin/bash
apt update && apt install -y nginx
ufw allow 'Nginx HTTP'
systemctl enable nginx && systemctl start nginx
```

4.4.2 cloud-init (Cloud)

Utilisé pour configurer automatiquement une VM à son lancement dans le cloud :

```yaml
#cloud-config
users:
  - name: devops
    ssh-authorized-keys:
      - ssh-rsa AAAAB3... user@host
    sudo: ['ALL=(ALL) NOPASSWD:ALL']
    shell: /bin/bash

packages:
  - nginx

runcmd:
  - systemctl enable nginx
  - systemctl start nginx
```

4.4.3 Ansible : Déploiement et configuration à distance de plusieurs serveurs avec des playbooks :

```yaml
- name: Configurer un serveur web
  hosts: webservers
  become: yes
  tasks:
    - name: Installer Nginx
      apt:
        name: nginx
        state: present
        update_cache: yes
    - name: Démarrer le service
      service:
        name: nginx
        state: started
        enabled: yes
```

⍰ Exercice pratique

Objectif :

Configurer un serveur Linux pour héberger un site web statique avec **Nginx**, le sécuriser, et automatiser cette configuration.

Étapes :

1. Créer une machine virtuelle (localement ou sur AWS/Azure/GCP).
2. Installer et configurer Nginx.
3. Ouvrir les ports nécessaires (HTTP/SSH).
4. Créer un utilisateur non root avec accès SSH.
5. Rédiger un **script bash** ou **playbook Ansible** pour automatiser cette configuration.

🖥 1. Création d'une machine virtuelle

Option 1 : En local (VirtualBox + Ubuntu Server)

- Créez une VM avec Ubuntu Server 22.04 LTS.
- Attribuez 1 à 2 Go de RAM, 1 CPU, et 10 Go de disque.
- Branchez une ISO d'installation Ubuntu Server et suivez l'installation.

Option 2 : Dans le Cloud (ex. AWS EC2)

- Lancez une instance EC2 (Ubuntu 22.04) avec un groupe de sécurité autorisant :
  - Port 22 (SSH)
  - Port 80 (HTTP)

⚙ 2. Configuration manuelle du serveur

Connexion SSH

```
ssh -i "votre-cle.pem" ubuntu@IP_PUBLIC
```

Mise à jour du système

```
sudo apt update && sudo apt upgrade -y
```

Installation de Nginx

```
sudo apt install nginx -y
```

Création de la page HTML

```
echo "<h1>Bienvenue sur mon site web !</h1>" | sudo tee /var/www/html/index.html
```

Activation et démarrage de Nginx

```
sudo systemctl enable nginx
sudo systemctl start nginx
```

Configuration du pare-feu (si UFW actif)

```
sudo ufw allow 'Nginx HTTP'
sudo ufw allow OpenSSH
sudo ufw enable
```

---

3. Création d'un utilisateur non-root avec accès SSH

```
sudo adduser devops
sudo usermod -aG sudo devops
sudo mkdir -p /home/devops/.ssh
sudo cp ~/.ssh/authorized_keys /home/devops/.ssh/
sudo chown -R devops:devops /home/devops/.ssh
sudo chmod 700 /home/devops/.ssh
sudo chmod 600 /home/devops/.ssh/authorized_keys
```

---

4. Script Bash – Automatisation

setup_nginx.sh

```
#!/bin/bash

# Mise à jour
apt update && apt upgrade -y

# Installation Nginx
apt install nginx -y

# Page web par défaut
echo "<h1>Serveur Nginx prêt à l'emploi !</h1>" > /var/www/html/index.html

# Activation du service
systemctl enable nginx
systemctl start nginx

# Pare-feu (si UFW actif)
ufw allow 'Nginx HTTP'
ufw allow OpenSSH
ufw --force enable
```

Exécution :

```
chmod +x setup_nginx.sh
sudo ./setup_nginx.sh
```

📄 5. Playbook Ansible – Automatisation distante

nginx-playbook.yml

```yaml
- name: Configurer un serveur web avec Nginx
  hosts: webservers
  become: true
  tasks:
    - name: Mettre à jour le système
      apt:
        update_cache: yes
        upgrade: dist

    - name: Installer Nginx
      apt:
        name: nginx
        state: present

    - name: Créer une page web
      copy:
        dest: /var/www/html/index.html
        content: "<h1>Bienvenue depuis Ansible !</h1>"

    - name: S'assurer que Nginx est activé et démarré
      service:
        name: nginx
        state: started
        enabled: yes

    - name: Autoriser le trafic HTTP via UFW
      ufw:
        rule: allow
        name: "Nginx HTTP"
```

💡 Commande de lancement :

```
ansible-playbook -i hosts nginx-playbook.yml
```

❓ Tests à réaliser

- Visitez l'adresse IP publique de votre serveur → vous devez voir la page HTML.
- Vérifiez que Nginx est actif :

  ```
  systemctl status nginx
  ```

- Vérifiez que les ports sont bien ouverts :

  ```
  sudo ufw status
  ```

- Testez la connexion SSH avec le nouvel utilisateur `devops`.

Module 5 : Description de la configuration des serveurs à déployer

Objectifs pédagogiques

À l'issue de ce module, vous serez capable de :

- Identifier les types de serveurs nécessaires pour une architecture donnée.
- Définir les caractéristiques techniques et logicielles de chaque serveur.
- Documenter la configuration cible avant le déploiement.
- Organiser et standardiser les configurations dans un contexte multi-environnements (dev, test, prod).
- Préparer la base pour les automatisations à l'aide d'outils comme Ansible, Terraform ou des templates cloud.

5.1 Pourquoi décrire la configuration des serveurs ?

Avant tout déploiement automatisé, il est essentiel de :

- **Anticiper les besoins** : type de serveur, ressources, rôle, sécurité.
- **Standardiser** : garantir l'homogénéité des configurations.
- **Communiquer** : rendre la configuration compréhensible pour les développeurs, ops, et managers.
- **Réutiliser** : créer des modèles pour éviter les duplications et faciliter la maintenance.

5.2 Éléments à décrire pour chaque serveur

Élément	Description
Nom du serveur	Nom logique (ex : `web-01`, `db-prod-01`)
Rôle / Fonction	Web, application, base de données, proxy, monitoring
Système d'exploitation	Ubuntu 22.04, CentOS 8, Windows Server 2022
Ressources	CPU, RAM, disque, type de stockage (SSD/HDD), réseau
Adresse IP / DNS	IP statique ou dynamique, nom DNS associé
Services à installer	Apache, NGINX, PostgreSQL, Redis, Docker, etc.
Utilisateurs	Comptes à créer, droits sudo, clés SSH
Règles de sécurité	Ports ouverts, pare-feu, groupe de sécurité cloud
Backups / Monitoring	Méthode de sauvegarde, outils de surveillance (Prometheus, Zabbix)

📝 5.3 Exemple de fiche de configuration serveur

```yaml
serveur: web-01
environnement: production
os: Ubuntu 22.04 LTS
cpu: 2 vCPU
ram: 4 GB
disque: 50 GB SSD
ip: 192.168.10.11
dns: web-prod.example.com
services:
  - nginx
  - certbot (Let's Encrypt)
firewall:
  - allow: 22/tcp
  - allow: 80/tcp
  - allow: 443/tcp
utilisateurs:
  - nom: devops
    sudo: true
    ssh_key: ssh-rsa AAAAB3Nza...
```

Ce genre de description peut être codée dans un fichier YAML ou JSON pour être utilisée ensuite dans des scripts ou des outils comme **Terraform**, **Ansible**, **Packer**, etc.

❔ 5.4 Bonnes pratiques

- **Centraliser les configurations** : stocker dans un dépôt Git versionné.
- **Utiliser des modèles** : créer des templates de serveurs selon les rôles.
- **Adapter selon l'environnement** : développement ≠ production (sécurité, perf).
- **Préparer pour l'automatisation** : écrire les fichiers de configuration dans des formats exploitables par les outils (YAML, JSON).
- **Tracer les modifications** : toute évolution doit être documentée.

⚒ 5.5 Outils utiles pour la description standardisée

Outil	Usage
Terraform variables	Définir les caractéristiques des serveurs cloud
Ansible inventory	Spécifier les hôtes, groupes et variables
Packer templates	Définir des images de VM prêtes à l'emploi
cloud-init	Automatiser la config initiale à partir d'un fichier
Excel/Notion/Markdown	Pour la documentation manuelle ou semi-structurée

---

⍰ Exercice pratique (à venir dans le module suivant)

- Créez une fiche de description de **3 serveurs différents** :
  - 1 serveur web
  - 1 serveur de base de données
  - 1 serveur de monitoring
- Choisissez un format (table, YAML ou Markdown).
- Prévoyez les IP, services, utilisateurs, sécurité, etc.
- Bonus : convertissez cette description en un **inventaire Ansible**.

✓ 1. Format YAML – Standard pour Infrastructure as Code

```yaml
serveurs:
  - nom: web-01
    role: serveur web
    os: Ubuntu 22.04
    cpu: 2 vCPU
    ram: 4GB
    disque: 50GB SSD
    ip: 192.168.10.11
    dns: web01.example.local
    services:
      - nginx
      - certbot
    utilisateurs:
      - nom: devops
        sudo: true
        ssh_key: ssh-rsa AAAAB3...

  - nom: db-01
    role: serveur de base de données
    os: Ubuntu 22.04
    cpu: 4 vCPU
    ram: 8GB
    disque: 100GB SSD
    ip: 192.168.10.21
    dns: db01.example.local
    services:
      - postgresql
      - fail2ban
    utilisateurs:
      - nom: dba
        sudo: false
        ssh_key: ssh-rsa AAAAB3...

  - nom: mon-01
    role: serveur de supervision
    os: Ubuntu 22.04
    cpu: 2 vCPU
    ram: 4GB
    disque: 40GB SSD
    ip: 192.168.10.31
    dns: mon01.example.local
    services:
      - prometheus
      - grafana
    utilisateurs:
      - nom: monitor
        sudo: true
        ssh_key: ssh-rsa AAAAB3...
```

2. Format Tableau – Pour documentation manuelle ou outil bureautique

Nom	Rôle	OS	CPU	RAM	Disque	IP	Services	Utilisateurs
web-01	Web	Ubuntu 22.04	2 vCPU	4GB	50GB	192.168.10.11	nginx, certbot	devops (sudo)
db-01	Base de données	Ubuntu 22.04	4 vCPU	8GB	100GB	192.168.10.21	postgresql, fail2ban	dba (no sudo)
mon-01	Monitoring	Ubuntu 22.04	2 vCPU	4GB	40GB	192.168.10.31	prometheus, grafana	monitor (sudo)

3. Inventaire Ansible

```ini
[webservers]
web-01 ansible_host=192.168.10.11 ansible_user=devops

[dbservers]
db-01 ansible_host=192.168.10.21 ansible_user=dba

[monitoring]
mon-01 ansible_host=192.168.10.31 ansible_user=monitor
```

Module 6 : Conception des scripts nécessaires aux déploiements

Objectifs pédagogiques

À la fin de ce module, vous serez capable de :

- Comprendre le rôle des scripts dans le déploiement d'infrastructure.
- Concevoir des scripts clairs, modulaires et réutilisables.
- Choisir les bons langages et outils selon le contexte (Bash, Python, PowerShell, Ansible, etc.).
- Éviter les erreurs classiques de scripting.
- Préparer les scripts pour l'automatisation (CI/CD, cloud-init, etc.).

6.1 Qu'est-ce qu'un script de déploiement ?

Un script de déploiement est un ensemble d'instructions automatisées qui permet de :

- Configurer un serveur ou une VM
- Installer des logiciels et dépendances
- Définir des utilisateurs, permissions, pare-feu
- Démarrer des services
- Appliquer des fichiers de configuration

Exemples :

- `install_web.sh` (Bash) pour installer Apache/Nginx
- `configure_db.yml` (Ansible) pour configurer PostgreSQL
- `cloud_init.yaml` pour initialiser une VM dans le cloud

6.2 Choisir la bonne technologie

Contexte	Langage / outil recommandé
Linux local / SSH	Bash, Python
Windows Server	PowerShell

Contexte	Langage / outil recommandé
Automatisation cloud / CI	Ansible, Terraform
Déploiement cloud initial	cloud-init, Packer
Déploiement applicatif	Docker Compose, Helm (Kubernetes)

🔁 6.3 Structure d'un bon script

Exemple générique (Bash)

```bash
#!/bin/bash

# --- Variables
WEB_ROOT="/var/www/html"

# --- Mise à jour
apt update && apt upgrade -y

# --- Installation
apt install -y nginx

# --- Configuration
echo "<h1>Déploiement automatique réussi</h1>" > $WEB_ROOT/index.html

# --- Sécurité
ufw allow 'Nginx HTTP'
ufw allow OpenSSH
ufw enable

# --- Démarrage
systemctl enable nginx
systemctl restart nginx
```

Bonnes pratiques :

- Commenter chaque section
- Utiliser des **variables**
- Écrire de façon **idempotente** (éviter de casser en cas de re-lancement)
- Gérer les erreurs (`set -e`, `trap` en Bash)
- Tester chaque étape dans un environnement de pré-production

---

📁 6.4 Organisation des fichiers

Pour un projet de déploiement automatisé :

```
infrastructure/
|
├── scripts/
|   ├── install_nginx.sh
|   ├── init_db.sh
|
├── ansible/
|   ├── playbook.yml
|   ├── roles/
|   ├── inventory/
|
├── cloud/
|   ├── cloud_init.yaml
|   ├── terraform.tf
```

📦 6.5 Exemples de tâches automatisables

Tâche	Script Bash/Ansible typique
Installation de paquets	`apt install`, `yum install`, `ansible apt module`
Création de dossiers/fichiers	`mkdir`, `touch`, `copy module`
Déploiement d'une app web	`git clone`, `npm install`, `systemctl`
Gestion des services	`systemctl`, `ansible service:`
Gestion utilisateurs	`useradd`, `adduser`, `ansible user:`

⚠ 6.6 Erreurs fréquentes à éviter

- Scripts non testés dans des environnements variés
- Hardcoding des chemins ou mots de passe
- Aucun log ni retour d'erreur
- Aucun contrôle de version
- Dépendances non documentées

✅ 6.7 Objectif de ce module

À la fin du module, vous devez être capable de **concevoir vos propres scripts de déploiement**, adaptés à :

- Un ou plusieurs serveurs
- Un rôle spécifique (web, base de données, monitoring…)
- Une utilisation manuelle ou automatisée (CI/CD)

⍰ Exercice pratique proposé

🎯 Objectif : Concevoir un script Bash (ou Ansible) qui :

- Installe Docker
- Crée un utilisateur `devops`
- Active le service Docker au démarrage
- Ajoute l'utilisateur au groupe Docker
- Configure le pare-feu pour autoriser le port 2375

```
#!/bin/bash

# Variables
USER="devops"
PORT=2375

# Mise à jour du système
apt update && apt upgrade -y

# Installation de Docker
apt install -y docker.io

# Création de l'utilisateur devops
id -u $USER &>/dev/null || useradd -m -s /bin/bash $USER

# Ajout de l'utilisateur au groupe docker
usermod -aG docker $USER

# Activation de Docker au démarrage
systemctl enable docker
systemctl start docker

# Configuration du pare-feu
ufw allow OpenSSH
ufw allow $PORT/tcp
ufw --force enable

# Confirmation
echo "Docker installé, utilisateur $USER ajouté au groupe docker, port $PORT ouvert."
```

Module 7 : Écriture des scripts nécessaires pour chaque serveur

🎯 Objectifs pédagogiques

À l'issue de ce module, vous serez capable de :

- Écrire des scripts adaptés au rôle de chaque serveur (web, DB, monitoring, etc.)
- Appliquer les bonnes pratiques pour un scripting maintenable
- Générer des scripts Bash, Ansible ou PowerShell réutilisables
- Organiser vos scripts par rôles ou groupes dans une architecture automatisée

7.1 Principe de séparation par rôle

Chaque serveur dans une architecture a un **rôle distinct** (web, DB, cache, proxy, monitoring, etc.), donc nécessite une configuration **spécifique**.
Il est important d'écrire un **script ou playbook par rôle**, pour garantir :

- Une meilleure lisibilité
- Une maintenance facilitée
- Une automatisation granulaire (par cible)

7.2 Structure d'organisation recommandée

```
scripts/
│
├── web/
│   └── install_nginx.sh
│   └── web_playbook.yml
│
├── db/
│   └── install_postgres.sh
│   └── db_playbook.yml
│
├── monitoring/
│   └── install_prometheus.sh
│   └── monitoring_playbook.yml
```

7.3 Exemples de scripts par serveur

Serveur Web – `install_nginx.sh`

```bash
#!/bin/bash

apt update && apt install -y nginx
systemctl enable nginx
systemctl start nginx
ufw allow 'Nginx Full'
```

Serveur Base de données – `install_postgres.sh`

```bash
#!/bin/bash

apt update && apt install -y postgresql postgresql-contrib
systemctl enable postgresql
systemctl start postgresql

# Création d'un utilisateur
sudo -u postgres createuser -s appuser
sudo -u postgres createdb appdb
```

Serveur Monitoring – `install_prometheus.sh`

```
#!/bin/bash

useradd --no-create-home --shell /bin/false prometheus

mkdir -p /etc/prometheus /var/lib/prometheus

# Télécharger Prometheus
wget https://github.com/prometheus/prometheus/releases/download/v2.50.0/prometheus-2.50.0.lin
tar xvf prometheus-*.tar.gz
cp prometheus-*/prometheus /usr/local/bin/
cp prometheus-*/promtool /usr/local/bin/

# Config minimal
echo 'global:\n  scrape_interval: 15s' > /etc/prometheus/prometheus.yml

# Service
cat <<EOF > /etc/systemd/system/prometheus.service
[Unit]
Description=Prometheus
[Service]
ExecStart=/usr/local/bin/prometheus --config.file=/etc/prometheus/prometheus.yml
[Install]
WantedBy=multi-user.target
EOF

systemctl daemon-reexec
systemctl enable prometheus
systemctl start prometheus
```

wget https://github.com/prometheus/prometheus/releases/download/v2.50.0/prometheus-2.50.0.linux-amd64.tar.gz

---

7.4 Bonnes pratiques

- **Modularisez** vos scripts (petites fonctions par tâche)
- **Testez** indépendamment chaque script sur une VM de dev
- **Commentez** chaque section
- **Protégez** les scripts sensibles (mot de passe, clés)
- **Versionnez** les scripts dans un dépôt Git
- **Générez un log de sortie** (`tee`, `logger`...)

7.5 Adaptation avec Ansible (optionnel)

Si vous utilisez Ansible, vous pouvez transformer vos scripts Bash en **playbooks**.
Exemple : Playbook pour serveur web

```yaml
- hosts: webservers
  become: yes
  tasks:
    - name: Installer NGINX
      apt:
        name: nginx
        state: present
        update_cache: yes

    - name: Autoriser le trafic HTTP/HTTPS
      ufw:
        rule: allow
        name: 'Nginx Full'
```

Exercice pratique

Objectif : Écrire un script **spécifique pour un serveur de cache (Redis)** :

- Installer Redis
- Lancer le service
- Activer le service au démarrage
- Configurer la sécurité de base (bind IP, port)
- Ajouter une règle UFW (port 6379)

Contenu de l'archive :

- install_redis.sh – Script Bash :
  - Installation de Redis
  - Configuration de la sécurité
  - Activation du service et ouverture du port 6379
- redis_playbook.yml – Playbook Ansible équivalent

```bash
#!/bin/bash

# Variables
REDIS_PORT=6379
BIND_ADDRESS=127.0.0.1

# Mise à jour système
apt update && apt install -y redis-server

# Configuration de base
sed -i "s/^bind .*/bind ${BIND_ADDRESS}/" /etc/redis/redis.conf
sed -i "s/^# requirepass .*/requirepass changeme123/" /etc/redis/redis.conf
sed -i "s/^supervised no/supervised systemd/" /etc/redis/redis.conf

# Activation du service
systemctl enable redis-server
systemctl restart redis-server

# Sécurisation pare-feu
ufw allow ${REDIS_PORT}/tcp

echo "Redis installé et configuré. Accessible uniquement sur ${BIND_ADDRESS}:${REDIS_PORT}"
```

```yaml
- hosts: redis
  become: yes
  tasks:
    - name: Installer Redis
      apt:
        name: redis-server
        state: present
        update_cache: yes

    - name: Modifier la configuration de Redis (bind address)
      lineinfile:
        path: /etc/redis/redis.conf
        regexp: '^bind'
        line: 'bind 127.0.0.1'

    - name: Définir le mot de passe Redis
      lineinfile:
        path: /etc/redis/redis.conf
        regexp: '^# requirepass'
        line: 'requirepass changeme123'

    - name: Activer le service Redis
      service:
        name: redis-server
        enabled: yes
        state: restarted

    - name: Ouvrir le port Redis dans UFW
      ufw:
        rule: allow
        port: 6379
        proto: tcp
```

Module 8 : Test des scripts nécessaires pour chaque serveur

Objectifs pédagogiques

À l'issue de ce module, vous serez capable de :

- Exécuter vos scripts dans un environnement de test (VM locale, cloud, conteneur)
- Détecter et corriger les erreurs avant déploiement réel
- Automatiser les tests de vos scripts (lint, dry-run, CI)
- Garantir que vos scripts sont **idempotents** (réexécutables sans effet négatif)

8.1 Pourquoi tester les scripts ?

« Un script non testé est une erreur en attente de se produire. »

Les erreurs de script peuvent :

- Planter un serveur
- Créer des failles de sécurité
- Perturber l'accès à des services critiques
- Être difficiles à détecter sans test

But du test : garantir que les scripts sont **fonctionnels, sûrs, cohérents**.

🛠 8.2 Méthodes de test

Méthode	Description	Outils
Exécution locale	Lancer le script dans une VM locale (VirtualBox, KVM)	Vagrant, Multipass
Conteneurisation	Tester le script dans un conteneur jetable	Docker
Environnement cloud isolé	Déploiement temporaire dans un cloud public ou privé	AWS EC2, Azure, Proxmox
Dry-run (simulation)	Valide la syntaxe sans appliquer les changements	`ansible-playbook --check`, `terraform plan`
CI/CD pipeline	Intègre le test dans un processus de livraison automatisé	GitHub Actions, GitLab CI

🧪 8.3 Outils de test Bash

1. Tester manuellement :

```
bash -n script.sh          # Vérifie la syntaxe
shellcheck script.sh       # Analyse statique (bonnes pratiques, sécurité)
```

2. Exemple avec Docker :

```
docker run -it --rm ubuntu bash
apt update && apt install -y curl
curl -sSL http://yourserver/install_script.sh | bash
```

🧪 8.4 Outils de test Ansible

1. Dry-run :

```
ansible-playbook playbook.yml --check –diff
```

2. Linting :

```
ansible-lint playbook.yml
```

3. Test automatisé avec Molecule :

```
pip install molecule docker
molecule init role -r nginx
cd nginx
molecule test
```

🔁 8.5 Tester l'idempotence

Un bon script peut être relancé sans provoquer d'erreurs ou réinitialisations indésirables.

Exemple :

```
bash install_nginx.sh
bash install_nginx.sh   # Ne casse rien au 2e passage
```

Avec Ansible :

```
ansible-playbook nginx.yml
ansible-playbook nginx.yml --check  # ne devrait proposer aucune modification
```

Module 9 : Lancement du déploiement

Objectifs pédagogiques

À l'issue de ce module, l'apprenant sera capable de :

- Lancer un déploiement automatisé d'une infrastructure complète.
- Enchaîner les outils de provisioning (ex : Terraform) et de configuration (ex : Ansible).
- Surveiller et vérifier le bon déroulement du déploiement.
- Identifier et corriger les erreurs éventuelles.

---

1. Concepts clés à comprendre

Terme	Définition
Provisioning	Création des ressources (VMs, réseaux, stockage, etc.)
Configuration	Installation et paramétrage des services sur les machines
Orchestration	Coordination de plusieurs étapes d'un déploiement
Idempotence	Capacité d'un script à être relancé sans effet secondaire

---

2. Enchaînement typique du déploiement

Voici les **étapes standards** dans un processus de déploiement :

Étape 1 : Initialiser les outils

```
terraform init
ansible-galaxy install -r requirements.yml
```

Étape 2 : Lancer le provisioning (Terraform)

```
terraform plan   # Visualiser ce qui va être créé
terraform apply  # Créer les ressources Cloud
```

Étape 3 : Récupérer les IPs publiques/privées

- Terraform peut générer un fichier `inventory.ini` ou `inventory.yml` pour Ansible.

```
terraform output -json > inventory.json
```

Étape 4 : Configurer les machines avec Ansible

```
ansible-playbook -i inventory.json site.yml
```

---

3. Vérifications post-déploiement

Éléments à vérifier	Méthode
Machines accessibles	`ping`, `ssh`, `ansible -m ping`
Services en écoute	`netstat`, `ss -tuln`, `curl`

Éléments à vérifier	Méthode
Logs d'exécution	Ansible (`.retry`, stdout), Terraform logs
Monitoring	Tableau de bord Prometheus, Grafana ou Cloud Provider

---

📌 4. Exemple pratique : déploiement d'un serveur web dans AWS

1. Provisioning avec Terraform

```
resource "aws_instance" "web" {
  ami           = "ami-0abcdef12345"
  instance_type = "t2.micro"
  tags = {
    Name = "WebServer"
  }
}
```

2. Configuration avec Ansible

```
- hosts: all
  become: yes
  tasks:
    - name: Installer NGINX
      apt:
        name: nginx
        state: present
```

3. Lancement complet (Makefile ou script bash)

```
terraform apply -auto-approve
terraform output -json > inventory.json
ansible-playbook -i inventory.json playbook.yml
```

⚠ 5. Gestion des erreurs et solutions

Problème	Cause fréquente	Solution
SSH timeout	Groupe de sécurité mal configuré	Ouvrir le port 22
Ansible "UNREACHABLE"	Mauvais inventaire ou IP	Corriger les IPs ou DNS
Erreur Terraform `already exists`	Ressource déjà présente	Appliquer `terraform destroy`
Déploiement partiel	Scripts non idempotents	Revoir les tâches pour qu'elles soient relançables

🕮 6. Bonnes pratiques

- Toujours **versionner les scripts** (Git).
- Utiliser des **tags** pour cibler des tâches précises dans Ansible :

```
ansible-playbook -t webserver playbook.yml
```

- Planifier les déploiements dans des environnements **sandbox/test** avant la prod.
- Utiliser **des outils de CI/CD** pour déclencher les déploiements automatiquement.

7. Outils complémentaires

Outil	Utilité
Jenkins / GitLab CI	Automatiser le pipeline de déploiement
Packer	Créer des images VM préconfigurées
Vault	Gérer les secrets pendant le déploiement
Consul	Découverte de service post-déploiement

✓ 8. Évaluation

📌 Exercice pratique :

Déployer une stack Web (NGINX + PostgreSQL) via Terraform + Ansible sur AWS

Ce projet inclut :

- Terraform pour le provisioning sur AWS
- Ansible pour la configuration (NGINX)
- Un Makefile pour tout orchestrer
- Un pipeline CI/CD avec GitLab CI

1. Structure du projet

```
css

infra-deploy/
├── terraform/
│   ├── main.tf
│   └── variables.tf
├── ansible/
│   ├── playbook.yml
│   ├── roles/
│   └── inventory_generator.sh
├── inventory.json
├── Makefile
└── .gitlab-ci.yml
```

🛠️ 2. `Makefile` – Orchestration des commandes

Makefile

```
# Fichier Makefile

init:
    cd terraform && terraform init

plan:
    cd terraform && terraform plan

apply:
    cd terraform && terraform apply -auto-approve
    cd terraform && terraform output -json > ../inventory.json

ansible:
    ansible-playbook -i inventory.json ansible/playbook.yml

deploy: init apply ansible

destroy:
    cd terraform && terraform destroy -auto-approve
```

📦 3. Exemple `main.tf` – Terraform AWS (simplifié)

hcl

```
provider "aws" {
  region = "us-east-1"
}

resource "aws_instance" "web" {
  ami           = "ami-0c02fb55956c7d316" # Ubuntu 20.04 LTS
  instance_type = "t2.micro"

  tags = {
    Name = "WebServer"
  }

  provisioner "local-exec" {
    command = "echo '[web]\n${self.public_ip}' > ../inventory.json"
  }
}
```

📜 4. `playbook.yml` – Ansible : installer NGINX

yaml

```yaml
- name: Configuration des serveurs
  hosts: all
  become: yes

  tasks:
    - name: Installer NGINX
      apt:
        name: nginx
        state: present
        update_cache: yes

    - name: S'assurer que NGINX est démarré
      service:
        name: nginx
        state: started
        enabled: true
```

⚙️ 5. `.gitlab-ci.yml` – CI/CD GitLab

yaml

```yaml
stages:
  - deploy

variables:
  TF_IN_AUTOMATION: "true"

deploy_infra:
  stage: deploy
  image: hashicorp/terraform:latest
  script:
    - cd terraform
    - terraform init
    - terraform apply -auto-approve
    - terraform output -json > ../inventory.json

config_serveurs:
  stage: deploy
  image: python:3.10
  before_script:
    - apt-get update && apt-get install -y ansible sshpass
  script:
    - ansible-playbook -i inventory.json ansible/playbook.yml
```

▶️ 6. Pour exécuter en local

```bash
make deploy     # Lancer tout le pipeline (Terraform + Ansible)
make destroy    # Supprimer toute l'infrastructure
```

Voici la version adaptée pour **Microsoft Azure** :

Cette version inclut :

- Provisioning Azure (via Terraform)
- Déploiement d'une VM Ubuntu + IP publique
- Playbook Ansible pour installer NGINX
- Makefile pour tout orchestrer
- Pipeline CI/CD (GitLab)

Pour qu'**Ansible puisse se connecter automatiquement** à la machine déployée sur Azure, il faut générer un fichier `inventory.json` (ou `.ini`) à partir de la **sortie Terraform** (l'adresse IP publique).

✅ Étapes automatiques

1. Ajout dans `outputs.tf` de Terraform

Dans `terraform/outputs.tf`, ajoute ceci (ou mets-le dans `main.tf` si pas de fichier dédié) :

hcl Copier

```hcl
output "public_ip" {
  value = azurerm_public_ip.pip.ip_address
}
```

2. Script Bash pour générer `inventory.json`

Crée un script `generate_inventory.sh` à la racine du projet (`infra-deploy/`) :

```bash
#!/bin/bash

# Extraire l'IP publique depuis Terraform
PUBLIC_IP=$(cd terraform && terraform output -raw public_ip)

# Générer un inventaire dynamique JSON
cat <<EOF > inventory.json
{
  "all": {
    "hosts": ["$PUBLIC_IP"],
    "vars": {
      "ansible_user": "azureuser",
      "ansible_ssh_private_key_file": "~/.ssh/id_rsa",
      "ansible_python_interpreter": "/usr/bin/python3"
    }
  }
}
EOF

echo "Inventaire généré avec succès : inventory.json"
```

🔐 Assure-toi que la **clé SSH privée** utilisée (`~/.ssh/id_rsa`) correspond à celle utilisée dans Terraform (`admin_ssh_key`).

3. Appel du script dans le Makefile

Ajoute une ligne à la cible `apply` de ton `Makefile` :

```make
apply:
    cd terraform && terraform apply -auto-approve
    ./generate_inventory.sh
```

✅ Exemple de `inventory.json` généré

```json
{
  "all": {
    "hosts": ["52.140.123.45"],
    "vars": {
      "ansible_user": "azureuser",
      "ansible_ssh_private_key_file": "~/.ssh/id_rsa",
      "ansible_python_interpreter": "/usr/bin/python3"
    }
  }
}
```

Module 10 : Maintien et mise en œuvre de l'évolution de la documentation technique

Objectifs du module :

- Comprendre le rôle crucial de la documentation dans le cycle de vie d'une infrastructure.
- Savoir créer, structurer et maintenir une documentation technique à jour.
- Mettre en place des bonnes pratiques et outils pour faciliter les mises à jour collaboratives.

---

1. Pourquoi documenter une infrastructure ?

Objectif	Description
Maintenabilité	Faciliter la compréhension de l'architecture, des processus de déploiement, de la configuration
Collaboration	Permettre à plusieurs équipes (Dev, Ops, Sécurité, Support) de travailler efficacement
Dépannage rapide	Accélérer la résolution d'incidents grâce à des schémas et procédures claires
Transfert de compétence	Accompagner l'intégration de nouveaux collaborateurs ou prestataires

---

2. Types de documentation à produire

Type	Contenu typique
Architecture	Schémas réseau, cartes de dépendance, composants cloud, etc.
Déploiement	Instructions Terraform/Ansible, pipeline CI/CD, Makefile
Configuration	Paramètres serveurs, secrets (vault), chemins, ports, firewalls
Exploitation / Monitoring	Logs, dashboards, alertes, tests automatisés
Procédures	Démarrage, rollback, mises à jour, réponses à incident

---

3. Outils recommandés

Outil	Usage
Markdown **+ Git**	Documentation versionnée, lisible, modifiable via pull request
MkDocs **ou** Docusaurus	Génération de sites de documentation statique
Diagrams.net **(anciennement draw.io)**	Schémas d'architecture clairs et modifiables
GitLab/GitHub Wiki	Documentation collaborative en ligne
Checklists automatisées	Intégrées aux CI/CD pour rappeler de documenter les changements

---

📄 4. Exemple de structure de documentation

```
docs/
├── 01-architecture/
│   └── infrastructure.md
├── 02-deploiement/
│   └── terraform-ansible.md
├── 03-configuration/
│   └── nginx-firewall.md
├── 04-exploitation/
│   └── supervision-alerting.md
├── 05-procedures/
│   └── rollback-reprise.md
└── index.md
```

🔁 5. Stratégies pour maintenir la doc à jour

✅ Intégration dans le workflow DevOps :

- Toute **merge request** (MR) liée à une modification d'infra doit inclure une **mise à jour de la doc**.
- Ajout de **checklists dans GitLab/GitHub** :

```markdown
- [ ] Les modifications sont documentées dans `docs/`
- [ ] Le schéma d'architecture a été mis à jour
```

🧪 Exemple de vérification CI

Ajoute dans `.gitlab-ci.yml` une étape de vérification :

```yaml
check_docs:
  script:
    - test -f docs/$(date +%Y)/$(date +%m)-update.md || echo "⚠ Documentation manquante"
```

🦾 6. Automatiser la documentation technique

- Utiliser **Terraform-docs** pour générer automatiquement la documentation des modules Terraform.
- Utiliser des outils comme **Swagger/OpenAPI** pour documenter des APIs si présentes.
- Utiliser **Ansible-doc** pour résumer les rôles Ansible.

🛠️ Exercice pratique proposé

📝 **Créer un fichier `docs/terraform-ansible.md` :**

1. Expliquer l'objectif de chaque dossier du projet (`terraform`, `ansible`, `Makefile`, etc.)
2. Documenter la procédure `make deploy`
3. Capturer une sortie de `terraform output` pour illustrer les IPs créées
4. Ajouter un exemple de playbook

Documentation : Déploiement avec Terraform & Ansible

1. Structure du projet

Le projet est organisé en plusieurs dossiers et fichiers ayant chacun un rôle précis :

- **`terraform/`**
  Contient les fichiers de configuration Terraform permettant de provisionner l'infrastructure (machines virtuelles, réseaux, etc.) sur un fournisseur cloud (comme AWS, GCP, etc.).
- **`ansible/`**
  Contient les playbooks et inventaires utilisés pour configurer les serveurs une fois qu'ils sont créés. Ansible est utilisé ici pour installer des logiciels, déployer des fichiers de configuration, etc.
- **`Makefile`**
  Automatise les commandes répétitives. Par exemple, `make deploy` enchaîne les étapes de déploiement Terraform + configuration Ansible.
- **`docs/`**
  Contient la documentation du projet (comme ce fichier).

2. Procédure : make deploy

La commande `make deploy` exécute l'enchaînement suivant :

```
deploy:
	terraform -chdir=terraform init
	terraform -chdir=terraform apply -auto-approve
	terraform -chdir=terraform output -json > ansible/inventory.json
	ansible-playbook -i ansible/inventory.json ansible/playbook.yml
```

Étapes détaillées :

1. **Initialisation de Terraform**
   `terraform init` initialise le dossier Terraform et télécharge les plugins nécessaires.
2. **Création de l'infrastructure**
   `terraform apply` applique les fichiers `.tf` pour créer ou mettre à jour les ressources.
3. **Extraction des IPs des machines**
   `terraform output -json` exporte les IPs publiques/privées des machines dans un fichier JSON utilisable par Ansible comme inventaire dynamique.
4. **Configuration des machines avec Ansible**
   `ansible-playbook` exécute le playbook principal qui configure les machines via SSH.

3. Exemple de sortie `terraform output`

Une fois le déploiement terminé, voici un exemple de ce que retourne la commande `terraform output` :

```json
{
  "web_ip": {
    "value": "34.239.123.45",
    "type": "string"
  },
  "db_ip": {
    "value": "10.0.1.5",
    "type": "string"
  }
}
```

Ce fichier est ensuite transformé pour être utilisé comme inventaire par Ansible.

4. Exemple de playbook Ansible

Voici un exemple de playbook simple pour installer Nginx sur les machines `web` :

```yaml
---
- name: Configure web servers
  hosts: all
  become: true

  tasks:
    - name: Update apt cache
      apt:
        update_cache: yes

    - name: Install nginx
      apt:
        name: nginx
        state: present

    - name: Start nginx
      service:
        name: nginx
        state: started
        enabled: true
```

Module 11 : Le dialogue avec les fournisseurs de services

Objectifs pédagogiques

- Comprendre les enjeux de la relation avec un fournisseur Cloud ou d'infrastructure
- Savoir formuler une demande ou un ticket de support technique
- Utiliser les bons canaux pour une communication efficace
- Connaître les bonnes pratiques en matière de suivi et d'escalade

---

1. Pourquoi dialoguer avec les fournisseurs ?

Dans un environnement Cloud ou hybride, les équipes IT dépendent d'un ou plusieurs prestataires pour des services critiques : hébergement, réseau, stockage, sécurité, etc.

Exemples de situations où le dialogue est essentiel :

- Problèmes de performance ou d'indisponibilité
- Demande d'augmentation de quota (par ex. nombre de VM)
- Facturation anormale ou mal comprise
- Accès API refusé
- Questions sur la roadmap produit ou nouvelles fonctionnalités

---

2. Moyens de communication disponibles

Canal	Cas d'usage	Avantages
Console Web (ex: AWS, Azure)	Création de ticket standard	Rapide, intégré à l'environnement
Support par e-mail	Questions ou réclamations détaillées	Archivage facile
Support téléphonique / chat live	Urgences, clarification en direct	Temps réel
Technical Account Manager (TAM)	Suivi dédié (si souscrit)	Personnalisation, suivi de dossier
Forum ou communauté	Questions générales ou bonnes pratiques	Échange avec la communauté

---

3. Types de contrats et niveaux de support

Les fournisseurs proposent plusieurs **niveaux de support (SLA)** :

Niveau	Délai de réponse typique	Exemple fournisseur
Gratuit / Basique	24-48h pour les tickets simples	AWS Basic, Azure Free
Standard	Quelques heures (24/7 en option)	GCP Standard Support
Premium	Moins de 1h, accès TAM	AWS Enterprise, Azure Premier

📝 **SLA = Service Level Agreement** : contrat qui définit les garanties (disponibilité, réactivité…).

---

4. Rédiger un ticket efficace

🔍 Bonnes pratiques :

- Être **clair, précis et structuré**
- Donner les **informations techniques complètes**
- Joindre des **logs ou captures d'écran**
- Décrire la **reproduction de l'erreur**, si possible

📄 Modèle de ticket :

```
Titre : Problème de déploiement EC2 – Erreur "Insufficient capacity"

Bonjour,

Nous rencontrons un problème lors de la création d'une instance EC2 t3.medium dans la région eu-w
`"Insufficient capacity error. Try a different AZ or instance type."`

Détails :
- Région : eu-west-1
- AZ : eu-west-1a
- Instance type : t3.medium
- Tentatives : 4 (entre 10h20 et 10h45 UTC)
- ID du compte : 123456789012

Pouvez-vous confirmer s'il y a une saturation temporaire ? Une solution alternative à proposer ?

Merci par avance,

[Nom, société]
```

Nous rencontrons un problème lors de la création d'une instance EC2 t3.medium dans la région eu-west-1a. Voici le message retourné :

`"Insufficient capacity error. Try a different AZ or instance type."`

📈 5. Suivi, relances et escalade

Suivre un ticket :

- Numéro de ticket + lien
- Historique des échanges
- Mises à jour automatiques par mail

Escalader un problème :

- Utiliser un champ "priorité" (Urgent, Haute, Normale)
- Justifier avec **impact métier**
- Contacter un TAM si disponible

💬 6. Dialogue technique : conseils

Quand vous discutez avec un **ingénieur support** :

- Restez **technique et factuel** : IP, logs, ID de ressource
- Soyez **coopératif**, ils ne sont pas décisionnaires mais intermédiaires
- Préparez vos demandes **à l'avance**, évitez les allers-retours flous

---

7. Exemple concret

Cas réel : augmentation de quota

Problème : Vous ne pouvez pas créer plus de 20 vCPUs dans Azure pour vos VM.

✅ **Étapes** :

1. Aller dans “Help + Support” > “New Support Request”
2. Catégorie : **Quota increase**
3. Choisir : vCPUs – Standard – région concernée
4. Rédiger : “Need to deploy a Kubernetes cluster with 50 vCPUs for staging.”

📬 Réponse d’Azure : Quota augmenté en 2 heures.

---

8. Relation fournisseur sur le long terme

- Maintenir un **contact régulier** : newsletters, TAM, formations
- Participer à des **bêtas de services**
- Faire remonter vos **cas d’usage métier** (important pour roadmap produit)

---

9. Points de vigilance

- Ne partagez jamais de **données sensibles** sans chiffrement
- **Notez les réponses techniques importantes** pour votre documentation interne
- **Vérifiez la facturation** après toute intervention du support

Module 12 : La veille technologique sur le Cloud hybride

Objectifs pédagogiques

- Comprendre ce qu’est un Cloud hybride et son importance stratégique
- Identifier les enjeux liés à la veille technologique dans ce contexte
- Connaître les outils, méthodes et ressources pour effectuer une veille efficace
- Être capable d’anticiper l’évolution des technologies Cloud et hybrides

---

1. Définition du Cloud hybride

Le **Cloud hybride** combine des **infrastructures locales (on-premise)** avec des **services de cloud public** (ex : AWS, Azure, GCP) via des **liens réseau, systèmes de gestion et outils communs**.

Exemple : Une entreprise garde ses bases de données critiques en local mais héberge ses applications web dans le cloud public.

🏗️ 2. Pourquoi la veille est cruciale dans un environnement hybride ?

Le Cloud hybride est en évolution constante. Voici pourquoi une veille est indispensable :

- Nouvelles fonctionnalités ou services Cloud
- Évolutions de sécurité (nouveaux standards, correctifs)
- Intégration de technologies émergentes (edge computing, IA, conteneurs)
- Décisions stratégiques sur le **choix d'un fournisseur ou d'une architecture**

🧭 3. Méthodologie de veille technologique

🔄 Cycle de veille :

1. **Collecte d'information** (flux, newsletters, blogs)
2. **Analyse** : tri, validation, pertinence
3. **Diffusion** : partages en équipe, documentation interne
4. **Exploitation** : test, benchmark, proof of concept (PoC)

🧰 4. Outils et sources pour la veille

Type	Exemples
Newsletters	The New Stack, AWS What's New, Azure Updates
Blogs officiels	AWS Blog, Google Cloud Blog, Azure Blog
Flux RSS	Feedly, Inoreader
Réseaux sociaux pro	LinkedIn, X (Twitter), Reddit r/devops
Plateformes vidéos	YouTube (AWS re:Invent, Microsoft Ignite)
Communautés	Stack Overflow, GitHub Discussions, Dev.to
Outils internes	Microsoft Viva, Notion, Confluence pour centraliser

---

🧰 5. Suivi des fournisseurs et technologies hybrides

Fournisseur	Solution hybride	Cas d'usage
Microsoft Azure	Azure Arc	Gérer des serveurs on-premises comme des ressources Azure
Amazon Web Services	AWS Outposts	Hardware AWS dans vos locaux
Google Cloud	Anthos	Gérer des clusters Kubernetes sur plusieurs environnements
VMware	VMware Cloud on AWS / Azure	Migration d'infrastructure virtualisée vers le cloud
Red Hat	OpenShift Hybrid Cloud	Orchestration Kubernetes hybride

---

6. Thèmes à surveiller dans le Cloud hybride

Thème	Pourquoi c'est important
Sécurité	VPN, zero trust, IAM multi-cloud
Interopérabilité	APIs, formats communs
Souveraineté des données	Règlementation (RGPD, localisation des données)
Coût	Outils de facturation multi-cloud, optimisation
Outils DevOps compatibles hybride	Terraform, Ansible, GitOps multi-environnements
Conteneurs	Kubernetes + outils d'observabilité hybrides

---

7. Bonnes pratiques pour organiser une veille efficace

- **Planifier un moment dédié chaque semaine**
- **Centraliser la veille dans un outil** (ex: Notion, Trello, SharePoint)
- **Faire des tests en sandbox** à chaque nouveauté majeure
- **Partager avec l'équipe** (réunion mensuelle, canal Teams ou Slack)
- **Créer une base de connaissances interne** (wiki, runbook technique)

8. Exemple concret de veille appliquée

Cas : Découverte d'AWS Outposts via une conférence AWS re:Invent

1. Lecture de la documentation officielle + vidéo de présentation
2. Analyse des cas d'usage pour l'entreprise (latence, confidentialité)
3. Benchmark avec Azure Stack
4. Rédaction d'un **rapport de veille** avec recommandations
5. Présentation à l'équipe pour une éventuelle expérimentation

9. Modèle de fiche de veille : Markdown

```
**Sujet** : Microsoft Azure Arc – Gestion de serveurs hybrides

Date : 25 avril 2025
Rédacteur : Jean Dupont
Source : Azure Blog, Ignite 2024, GitHub Azure Arc
Résumé : Azure Arc permet d'enregistrer des machines locales dans Azure, de les superviser, mettre à jour et sécuriser à distance.

Avantages :
- Intégration avec Azure Security Center
- Déploiement centralisé de politiques
- Support de serveurs Linux et Windows

Impact pour notre infra actuelle :
- Potentiel fort pour gérer le parc existant sans migration
- À tester sur un périmètre pilote

Actions proposées :
- Lancer un PoC en mai 2025 sur 5 serveurs locaux
- Impliquer les équipes sécurité et réseau
```

Module 13 : L’outil d’automatisation de déploiement d’infrastructure : Ansible

Objectifs pédagogiques

- Comprendre le rôle d’Ansible dans l'automatisation des infrastructures
- Maîtriser la structure et la syntaxe des fichiers Ansible (YAML, playbooks, rôles)
- Apprendre à écrire, tester et exécuter des playbooks simples à avancés
- Être capable de déployer et configurer automatiquement plusieurs serveurs

1. Introduction à Ansible

Ansible est un outil **open source** d’automatisation IT développé par Red Hat. Il permet de :

- Déployer des applications
- Gérer la configuration des serveurs
- Orchestrer des tâches complexes

Atouts d’Ansible :

- Sans agent (utilise SSH)
- Simple (fichiers YAML)
- Extensible (rôles, modules)
- Idéal pour les environnements Cloud ou hybrides

2. Architecture d'Ansible

Élément	Description
Control Node	Machine d’où on exécute les commandes Ansible
Managed Nodes	Machines distantes à configurer
Inventory	Liste des serveurs à gérer
Playbook	Script de tâches automatisées
Modules	Actions prédéfinies (ex: installer un paquet, copier un fichier)

Exemple d’inventory (`hosts.ini`) :

```
[webservers]
192.168.1.101
192.168.1.102

[dbservers]
192.168.1.201
```

3. Structure d'un playbook Ansible

Un **playbook** est écrit en YAML, avec une structure claire :

```yaml
- name: Installer un serveur web
  hosts: webservers
  become: yes
  tasks:
    - name: Installer Apache
      apt:
        name: apache2
        state: present
    - name: Démarrer Apache
      service:
        name: apache2
        state: started
        enabled: yes
```

✅ Ce playbook :

- cible le groupe `webservers`
- installe Apache et le démarre

---

4. Principaux modules utilisés

Module	Description	Exemple
`apt / yum`	Installer des paquets	Installer nginx
`copy`	Copier un fichier	Copier une config nginx
`template`	Fichier avec variables	Générer un `nginx.conf` dynamique
`service`	Gérer un service	Démarrer ou stopper un service
`user`	Créer des utilisateurs	Ajouter `devops`
`file`	Gérer les permissions	Créer un répertoire avec droits

---

5. Variables, templates et handlers

Variables :

Définies dans le playbook ou un fichier **`vars.yml`**

```yaml
vars:
  http_port: 80
```

Templates :

Fichier `.j2` (Jinja2) dynamique

`index.html.j2 :`

```
html
Hello {{ ansible_hostname }} !
```

🔁 Handlers :

Tâches déclenchées si une modification a lieu : Yaml

```yaml
- name: Reload nginx
  service:
    name: nginx
    state: reloaded
  listen: "restart nginx"
```

📁 6. Rôles Ansible : organisation modulaire

Un **rôle** est une structure réutilisable qui contient : Css

```
roles/
  webserver/
    tasks/
      main.yml
    templates/
      index.html.j2
    vars/
      main.yml
```

🔄 On appelle un rôle dans un playbook comme ceci : Yaml

```yaml
- hosts: all
  roles:
    - webserver
```

🧑‍💻 7. Exemple complet de déploiement

Cas : Déployer un site statique sur deux serveurs web

Inventory :

```yaml
- hosts: webservers
  become: yes
  tasks:
    - name: Installer nginx
      apt:
        name: nginx
        state: present

    - name: Copier la page HTML
      copy:
        src: index.html
        dest: /var/www/html/index.html

    - name: Démarrer nginx
      service:
        name: nginx
        state: started
        enabled: yes
```

---

8. Tests et exécution

Commandes de base :

Commande	Description
`ansible all -m ping`	Tester la connexion aux nœuds
`ansible-playbook playbook.yml`	Exécuter le playbook
`ansible-playbook -i inventory playbook.yml`	Utiliser un inventaire personnalisé
`ansible-doc <module>`	Lire la documentation d'un module

---

9. Gestion des secrets : Ansible Vault

Chiffrer un fichier contenant un mot de passe :

```
ansible-vault encrypt secrets.yml
```

Utilisation dans un playbook :

```
vars_files:
  - secrets.yml
```

---

10. Bonnes pratiques avec Ansible

- Structurer les projets avec **rôles**
- Documenter chaque tâche
- Utiliser des **tags** pour exécuter une partie seulement du playbook
- Gérer les versions avec Git
- Créer des **environnements de staging**

Quiz Interactif – Révision des commandes Ansible (10 questions)

---

1. Quelle commande permet de tester la connectivité SSH entre le control node et les machines distantes ?

A. `ansible-inventory`
B. `ansible all -m ping`
C. `ansible-playbook -m test`
D. `ansible-connect`

---

2. Comment exécuter un playbook nommé `site.yml` avec un fichier d'inventaire personnalisé `hosts.ini` ?

A. `ansible-playbook site.yml hosts.ini`
B. `ansible-playbook -i site.yml hosts.ini`
C. `ansible-playbook -i hosts.ini site.yml`
D. `ansible-playbook site.yml -inventory hosts.ini`

3. Quelle commande affiche la documentation d'un module Ansible comme `apt` ou `copy` ?

A. `ansible --help apt`
B. `ansible-doc apt`
C. `ansible-playbook apt`
D. `ansible apt -doc`

---

4. Quelle commande permet de chiffrer un fichier contenant des variables sensibles ?

A. `ansible-vault secure secrets.yml`
B. `ansible encrypt secrets.yml`
C. `ansible-vault encrypt secrets.yml`
D. `ansible-playbook --encrypt secrets.yml`

---

5. Quelle commande permet d'afficher le contenu d'un fichier chiffré avec Vault ?

A. `ansible-vault view secrets.yml`
B. `ansible-vault show secrets.yml`
C. `ansible-vault list secrets.yml`
D. `ansible-vault decrypt-view secrets.yml`

---

6. Quelle commande exécute uniquement les tâches portant le tag `web` dans un playbook ?

A. `ansible-playbook site.yml --only web`
B. `ansible-playbook site.yml --tags web`
C. `ansible-playbook site.yml --filter web`
D. `ansible-playbook --tag web site.yml`

---

7. Quelle commande permet de créer une nouvelle structure de rôle Ansible ?

A. `ansible-role init myrole`
B. `ansible-role-create myrole`
C. `ansible-galaxy init myrole`
D. `ansible init-role myrole`

---

8. Comment exécuter un playbook sans effectuer réellement les changements (mode dry run) ?

A. `ansible-playbook site.yml --test`
B. `ansible-playbook site.yml --pretend`
C. `ansible-playbook site.yml --check`
D. `ansible-playbook --simulate site.yml`

---

9. Quel fichier est utilisé pour définir des groupes de machines à gérer ?

A. `inventory.yml`
B. `ansible.cfg`

C. `playbook.yml`
D. `hosts`

---

10. Quel module est utilisé pour copier un fichier depuis le control node vers un serveur ?

A. `file`
B. `copy`
C. `fetch`
D. `move`

🚀 Quiz Bonus – Cas d'usage avancés (5 questions)

---

11. Tu veux redémarrer Nginx *uniquement si sa configuration a changé*. Que dois-tu utiliser dans ton playbook ?

A. Une tâche avec un module `service` classique
B. Un rôle avec `become: yes`
C. Un handler associé à une tâche notifiée
D. Un tag spécifique `restart`

---

12. Tu veux gérer plusieurs environnements (staging, production) avec le même playbook. Quelle est la méthode recommandée ?

A. Créer deux playbooks identiques
B. Utiliser un fichier `hosts` différent et des variables d'environnement
C. Fusionner toutes les configs dans un seul fichier YAML
D. Modifier manuellement les IPs dans le playbook

---

13. Tu veux injecter dynamiquement des variables dans un template Jinja2 (`.j2`) utilisé par Ansible. Que fais-tu ?

A. Écris les variables directement dans le fichier HTML
B. Utilise les blocs `with` dans le playbook
C. Définis des variables dans `vars/` et références-les avec `{{ var_name }}` dans le template
D. Ce n'est pas possible avec Ansible

---

14. Tu dois créer un utilisateur uniquement s'il n'existe pas déjà. Quel module Ansible utiliser et comment ?

A. `file`, avec la condition `when: not exists`
B. `adduser`, avec une commande shell
C. `user`, Ansible gère l'état souhaité nativement
D. `apt`, avec un package spécial utilisateur

---

15. Tu veux exécuter une commande shell seulement si un fichier `.env` existe dans `/opt/app`. Comment fais-tu ?

A. En utilisant `creates` dans un bloc `command`
B. En utilisant une variable `optional_file: true`
C. Avec `when: ansible_env.env == true`
D. Avec `when: ansible_facts['os'] == 'Ubuntu'`

Ces questions touchent à la logique **idempotente** d'Ansible, la **gestion des rôles/environnements**, et l'usage avancé des **handlers, conditions et templates**.

✅ Correction – Quiz principal (questions 1 à 10)

#	Réponse attendue	Explication
1	**B**	`ansible all -m ping` utilise le module `ping` pour tester la connectivité SSH avec tous les hôtes.
2	**C**	La syntaxe correcte est `ansible-playbook -i hosts.ini site.yml` : `-i` pour spécifier l'inventaire.
3	**B**	`ansible-doc apt` affiche la documentation du module `apt`.
4	**C**	`ansible-vault encrypt secrets.yml` permet de chiffrer un fichier YAML avec des données sensibles.
5	**A**	`ansible-vault view secrets.yml` permet de voir un fichier chiffré sans le déchiffrer définitivement.
6	**B**	`--tags` permet de cibler des tâches portant des tags spécifiques comme `web`.
7	**C**	`ansible-galaxy init myrole` est la commande officielle pour créer un squelette de rôle.
8	**C**	`--check` active le mode dry run pour simuler l'exécution sans appliquer les changements.
9	**D**	Le fichier `hosts` (ou `hosts.ini`) est utilisé comme inventaire d'hôtes à gérer.
10	**B**	Le module `copy` sert à transférer un fichier local vers une machine distante.

🔥 Correction – Quiz BONUS (questions 11 à 15)

#	Réponse attendue	Explication
11	**C**	Les `handlers` sont appelés uniquement si une tâche notifie un changement – idéal pour redémarrer un service **si la config change**.
12	**B**	On gère plusieurs environnements avec **des fichiers d'inventaire différents** et des variables spécifiques à chaque groupe.
13	**C**	On utilise des fichiers `.j2` avec des `{{ variables }}` injectées depuis `vars`, `group_vars`, etc.
14	**C**	Le module `user` d'Ansible est **idempotent**, il ne créera pas un utilisateur s'il existe déjà.
15	**A**	`creates` (ou `removes`) dans les modules `command` ou `shell` permet d'exécuter une commande **seulement si** un fichier n'existe/existe.

Module 14 : L'outil d'automatisation de déploiement d'infrastructure : Terraform

Objectifs pédagogiques

- Comprendre le fonctionnement de Terraform dans l'automatisation d'infrastructure
- Apprendre à écrire des fichiers de configuration en HCL (HashiCorp Configuration Language)
- Créer, modifier et supprimer des ressources Cloud de manière déclarative
- Gérer l'état de l'infrastructure et les cycles de vie
- Être capable de déployer une infrastructure reproductible, versionnable et modulaire

1. Introduction à Terraform

Terraform est un **outil d'Infrastructure as Code (IaC)** développé par **HashiCorp**. Il permet de décrire une infrastructure (machines, réseaux, volumes, etc.) sous forme de code, et de la déployer automatiquement.

Avantages clés :

- **Multi-cloud** : AWS, Azure, GCP, etc.
- **Déclaratif** : on décrit *l'état désiré*
- **Versionnable** : code suivi dans Git
- **Modulaire** : réutilisation des composants
- **Idempotent** : pas de duplication d'état

2. Architecture et fonctionnement

Les composants principaux :

Composant	Rôle
Fichiers `.tf`	Contiennent la description de l'infrastructure
Providers	Connecteurs vers les services (AWS, Azure, etc.)
State file (`terraform.tfstate`)	Fichier local ou distant qui stocke l'état actuel de l'infrastructure
Modules	Blocs réutilisables de configuration
Terraform CLI	Interface en ligne de commande pour init, plan, apply, etc.

3. Première configuration Terraform – Exemple basique

Objectif : Déployer une instance EC2 sur AWS

Étapes :

`main.tf`

```
provider "aws" {
  region = "us-east-1"
}
```

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

💡 Ici :

- `provider` déclare la région AWS
- `resource` décrit une machine EC2 à créer

🧱 4. Commandes de base Terraform

Commande	Fonction
`terraform init`	Initialise le projet et télécharge les providers
`terraform plan`	Simule les changements
`terraform apply`	Applique les changements
`terraform destroy`	Supprime toute l'infrastructure
`terraform fmt`	Formate le code
`terraform validate`	Valide la syntaxe des fichiers `.tf`

🔁 Terraform garde un **état** de ton infra pour gérer les différences futures.

🧱 5. Variables et outputs

🎁 `variables.tf`

```
variable "instance_type" {
  default = "t2.micro"
}
```

📄 `main.tf`

```
resource "aws_instance" "web" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = var.instance_type
}
```

📤 `outputs.tf`

```
output "ip_public" {
  value = aws_instance.web.public_ip
}
```

✅ Résultat : après un `terraform apply`, l'adresse IP s'affiche automatiquement.

🧱 6. Modularité avec les modules

Créer un module dans un dossier :

📁 Structure : css

```
modules/
  instance/
    main.tf
    variables.tf
    outputs.tf
```

Et on l'utilise ainsi : hcl

```
module "webserver" {
  source = "./modules/instance"
  instance_type = "t3.micro"
}
```

✅ **Avantage** : Tu réutilises le même bloc pour différents environnements (dev, prod...).

🌍 7. Gestion de l'état distant

Par défaut, l'état (`terraform.tfstate`) est local.

En entreprise, il est **recommandé** de le stocker dans un backend distant (ex : S3 + DynamoDB pour le verrouillage).

```
terraform {
  backend "s3" {
    bucket = "mon-bucket-tfstate"
    key    = "dev/terraform.tfstate"
    region = "us-west-2"
  }
}
```

🔐 8. Sécurité et gestion des secrets

Terraform ne chiffre pas les variables sensibles par défaut. Il faut :

- **Ne jamais commit le `.tfstate`**
- Utiliser des outils comme **Vault**, **SSM**, ou des fichiers `*.tfvars` exclus du Git
- Stocker les secrets dans des **variables d'environnement**

---

9. Cas concret – Infrastructure Web

Objectif :

Déployer automatiquement :

- 1 instance EC2 (backend)
- 1 bucket S3 (stockage statique)
- 1 base de données RDS

`main.tf` (extrait simplifié) :

```
module "backend" {
  source        = "./modules/ec2"
  instance_type = "t3.medium"
}

module "db" {
  source     = "./modules/rds"
  db_engine  = "postgres"
  db_version = "12"
}

module "storage" {
  source = "./modules/s3"
  bucket_name = "mon-site-statique"
}
```

---

10. Bonnes pratiques Terraform

✓ Versionner ton code (Git)
✓ Utiliser `.tfvars` pour les environnements
✓ Documenter tous les modules
✓ Organiser par environnement (`/dev`, `/prod`)
✓ Appliquer le principe **DRY** avec des modules réutilisables
✓ Toujours faire un `terraform plan` avant `apply`
✓ Stocker l'état à distance et le verrouiller

Quiz Terraform – Niveau Fondamental à Intermédiaire (10 questions)

---

1. Quel est le rôle du fichier `terraform.tfstate` ?

A. Il contient les logs de déploiement
B. Il stocke l'état actuel de l'infrastructure
C. Il contient la liste des erreurs Terraform
D. Il définit les variables globales

---

2. Quelle commande Terraform initialise un nouveau projet ?

A. `terraform start`
B. `terraform new`

C. `terraform init`
D. `terraform create`

---

3. Quelle commande permet de simuler les modifications avant de les appliquer ?

A. `terraform preview`
B. `terraform simulate`
C. `terraform plan`
D. `terraform dry-run`

---

4. Quel fichier contient la description principale de l'infrastructure ?

A. `main.tf`
B. `terraform.tfstate`
C. `terraform.cfg`
D. `infra.conf`

---

5. Quelle extension est utilisée pour les fichiers de configuration Terraform ?

A. `.yaml`
B. `.json`
C. `.tf`
D. `.hcl`

---

6. Comment appelle-t-on un ensemble réutilisable de ressources Terraform ?

A. Un bloc
B. Un module
C. Un package
D. Un provider

---

7. Quelle commande supprime toutes les ressources gérées par Terraform ?

A. `terraform clean`
B. `terraform delete`
C. `terraform destroy`
D. `terraform down`

---

8. Quel mot-clé est utilisé pour déclarer une variable dans Terraform ?

A. `var:`
B. `define`
C. `let`
D. `variable`

---

9. Quel fichier Terraform est le plus souvent ignoré dans un dépôt Git ?

A. `variables.tf`
B. `main.tf`
C. `terraform.tfstate`
D. `provider.tf`

---

10. Quelle commande est utilisée pour mettre en forme le code Terraform ?

A. `terraform clean`
B. `terraform format`
C. `terraform fmt`
D. `terraform style`

Quiz Terraform – Niveau BONUS (10 questions)

---

1. Que permet de faire le backend S3 dans Terraform ?

A. Chiffrer les variables sensibles
B. Gérer les versions de modules
C. Stocker l'état de manière centralisée et partagée
D. Lancer des déploiements multi-cloud

---

2. Quelle option est indispensable pour activer le verrouillage d'état sur AWS ?

A. Créer un bucket avec versioning
B. Utiliser un fichier `.lock` dans le dépôt Git
C. Configurer un verrouillage DynamoDB en plus de S3
D. Ajouter `force_unlock = true`

---

3. Quel mot-clé est utilisé pour charger un module externe ?

A. `module_path`
B. `include`
C. `import`
D. `source`

---

4. Quelle commande permet de changer d'environnement dans Terraform (dev, prod...) ?

A. `terraform env select`
B. `terraform workspace select`
C. `terraform switch`
D. `terraform setenv`

---

5. Que fait `terraform taint` ?

A. Marque une ressource pour recréation au prochain apply
B. Supprime une ressource manuellement
C. Ignore une ressource lors du plan
D. Ajoute une variable sensible au vault

---

6. Comment transmettre des valeurs aux variables lors d'un apply ?

A. `terraform apply -set var="..."`
B. `terraform apply -vars=filename`
C. `terraform apply -var="name=value"`
D. `terraform apply --env-file`

---

7. Quelle structure te permet de créer plusieurs ressources dynamiquement ?

A. `for_each` ou `count`
B. `loop`
C. `instance_group`
D. `multi_resource`

---

8. Quelle est la meilleure pratique pour gérer les secrets dans Terraform ?

A. Les stocker dans `terraform.tfvars` et versionner le fichier
B. Les injecter via des variables d'environnement non committées
C. Les écrire dans `main.tf` en clair pour plus de lisibilité
D. Les crypter à la main et les coller dans les fichiers `.tf`

---

9. Quel outil est souvent utilisé en complément de Terraform pour gérer les secrets ?

A. Jenkins
B. Packer
C. Vault
D. Docker

---

10. Tu dois partager un module interne dans ton entreprise. Quelle méthode est recommandée ?

A. Le stocker dans un bucket S3 ou un repo Git avec `source =`
B. L'inclure dans un fichier `.zip` dans le projet
C. Le copier-coller dans chaque projet manuellement
D. L'ajouter en tant que provider personnalisé

✓ Correction – Quiz Terraform – Niveau Fondamental à Intermédiaire

#	Réponse attendue	Explication
1	B	Le fichier `terraform.tfstate` contient l'état actuel de l'infrastructure déployée. Ce fichier est essentiel pour suivre les ressources et déterminer les actions nécessaires lors des déploiements futurs.
2	C	La commande `terraform init` initialise un projet Terraform, télécharge les providers et configure les fichiers nécessaires.
3	C	`terraform plan` permet de simuler l'application des modifications, ce qui permet de voir les changements avant de les appliquer.
4	A	Le fichier principal où l'infrastructure est définie est `main.tf`. C'est ici que tu décris les ressources, les variables, les providers, etc.
5	C	L'extension des fichiers Terraform est `.tf`. Cela permet à Terraform de reconnaître les fichiers de configuration.
6	B	Un **module** dans Terraform est un ensemble réutilisable de ressources. Les modules permettent d'organiser le code et de le rendre plus modulaire et réutilisable.
7	C	`terraform destroy` supprime toutes les ressources gérées par Terraform. Cela permet de détruire l'infrastructure que tu as créée avec Terraform.
8	D	Le mot-clé pour déclarer une variable dans Terraform est `variable`. Il permet de définir les valeurs qui peuvent être dynamiquement ajustées lors de l'exécution.
9	C	Le fichier `terraform.tfstate` contient des informations sensibles sur l'infrastructure et ne doit **pas** être versionné dans un dépôt Git. Il est conseillé de l'ajouter à `.gitignore`.
10	C	`terraform fmt` est utilisé pour formater le code Terraform afin de le rendre lisible et cohérent, ce qui facilite la gestion du code à long terme.

🔥 Correction – Quiz Terraform – Niveau BONUS (Avancé)

#	Réponse attendue	Explication
1	C	Le **backend S3** permet de stocker l'état Terraform dans un bucket S3. Cela permet de partager l'état entre plusieurs utilisateurs ou systèmes tout en centralisant la gestion des ressources.
2	C	Pour activer le **verrouillage d'état** avec DynamoDB en complément de S3, tu dois configurer DynamoDB pour qu'il verrouille l'état lorsque plusieurs utilisateurs ou systèmes tentent de modifier l'infrastructure simultanément.
3	D	Le mot-clé `source` est utilisé pour charger un module externe dans Terraform. Cela permet de réutiliser des ressources définies ailleurs, par exemple sur le Terraform Registry.
4	B	Pour changer d'environnement dans Terraform, on utilise la commande `terraform workspace select`. Les workspaces permettent de gérer des environnements séparés (par exemple, dev, prod).
5	A	La commande `terraform taint` marque une ressource comme "endommagée" afin qu'elle soit recréée lors de l'application suivante. Cela permet de forcer la recréation d'une ressource sans supprimer toute l'infrastructure.
6	C	On transmet des valeurs aux variables avec `-var="name=value"`. Cela permet de modifier des valeurs dans les configurations sans avoir à les définir dans les fichiers `.tfvars` ou `main.tf`.

#	Réponse attendue	Explication
7	A	Les structures `for_each` et `count` sont utilisées pour créer des ressources dynamiquement. `count` permet de créer un nombre spécifié de ressources, tandis que `for_each` permet de créer des ressources pour chaque élément d'une liste ou d'un mappage.
8	B	La meilleure pratique pour gérer les secrets dans Terraform est de les injecter via des **variables d'environnement non committées**. Il est essentiel de ne jamais laisser des secrets en clair dans les fichiers de configuration.
9	C	**Vault** est un outil conçu pour la gestion des secrets, et il est souvent utilisé avec Terraform pour injecter des informations sensibles de manière sécurisée dans les configurations.
10	A	Pour partager un module interne, la meilleure pratique est de le stocker dans un bucket S3 ou un dépôt Git et de l'importer dans d'autres projets avec `source = "path/to/module"`. Cela permet une gestion centralisée et réutilisable des modules.

Récapitulatif

- **Niveau Fondamental à Intermédiaire** : Ce quiz a couvert les bases de Terraform, comme l'initialisation, la configuration des fichiers, les commandes de gestion de l'infrastructure et les bonnes pratiques.
- **Niveau Bonus (Avancé)** : Ici, tu as été confronté à des situations plus complexes, comme l'utilisation des backends, la gestion des secrets, les workspaces et la gestion dynamique des ressources.

Module 15 : Définition de l'Hyper-V

Objectifs pédagogiques

À la fin de ce module, vous serez en mesure de :

- Comprendre ce qu'est **Hyper-V** et comment il fonctionne.
- Expliquer les cas d'utilisation et les avantages d'Hyper-V dans un environnement de virtualisation.
- Savoir configurer un **hôte Hyper-V** et déployer des **machines virtuelles (VM)**.
- Gérer les ressources d'Hyper-V et les machines virtuelles à l'aide des outils de gestion.

---

1. Qu'est-ce qu'Hyper-V ?

Hyper-V est une plateforme de virtualisation développée par **Microsoft**. Elle permet de créer, gérer et exécuter des machines virtuelles (VM) sur des serveurs physiques, et ce, sur des systèmes d'exploitation **Windows Server** et **Windows 10/11** (avec les éditions professionnelles ou Entreprise).

Hyper-V dans un environnement professionnel :

Hyper-V permet de virtualiser des serveurs, des applications et des environnements de test en permettant l'exécution de multiples systèmes d'exploitation **(Windows, Linux, etc.)** sur un même matériel physique. Il fournit une isolation des environnements et permet une gestion plus flexible des ressources.

🔑 2. Fonctionnement d'Hyper-V

Hyper-V fonctionne sur un modèle **type hyperviseur de type 1** (hyperviseur natif), ce qui signifie qu'il fonctionne directement sur le matériel sans nécessiter de système d'exploitation hôte. Cela le rend plus performant et mieux adapté à un environnement de production.

Principaux composants d'Hyper-V :

1. **Hyper-V Hypervisor** : Le composant central qui gère la virtualisation. Il est responsable de l'allocation des ressources matérielles aux VM.
2. **Hôte Hyper-V** : L'ordinateur physique qui exécute l'hyperviseur. Il est configuré pour accueillir des machines virtuelles.
3. **Machines Virtuelles (VM)** : Les instances logicielles qui exécutent les systèmes d'exploitation invités, comme Windows, Linux, ou d'autres.
4. **Virtual Switch** : Un composant qui permet de gérer les connexions réseau des VM. Il fonctionne comme un commutateur réseau virtuel permettant aux machines virtuelles de communiquer entre elles et avec l'extérieur.
5. **Gestionnaire Hyper-V (Hyper-V Manager)** : Un outil graphique permettant de gérer les hôtes Hyper-V et les VM, incluant la création, la configuration, et la gestion des VM.
6. **PowerShell Hyper-V** : Une interface en ligne de commande pour automatiser les tâches d'administration liées à Hyper-V.

🛠 3. Installation et Configuration d'Hyper-V

A. Installation d'Hyper-V sur Windows Server

1. Ouvrir **le Gestionnaire de serveur**.
2. Aller dans **"Ajouter des rôles et fonctionnalités"**.
3. Sélectionner **"Hyper-V"** dans les rôles disponibles et suivre les étapes de l'assistant d'installation.
4. Après l'installation, il est nécessaire de redémarrer l'hôte.

B. Installation d'Hyper-V sur Windows 10/11

1. Aller dans **Panneau de configuration > Programmes > Activer ou désactiver des fonctionnalités Windows**.
2. Cochez la case **Hyper-V** et cliquez sur **OK**.
3. Redémarrer le système pour que la fonctionnalité soit activée.

⚙ 4. Créer une Machine Virtuelle (VM) avec Hyper-V

Étapes de création d'une machine virtuelle via Hyper-V Manager :

1. Ouvrir **Hyper-V Manager**.
2. Cliquer sur **"Nouvelle Machine Virtuelle"**.
3. Spécifier le nom, la mémoire allouée et le disque dur virtuel.
4. Choisir un **fichier ISO** ou un disque de démarrage pour installer un système d'exploitation.
5. Configurer le réseau virtuel pour connecter la VM à l'extérieur (via un **Virtual Switch**).
6. Lancer la machine virtuelle et suivre les étapes d'installation du système d'exploitation.

---

5. Gestion des ressources Hyper-V

Gestion des ressources allouées à une VM :

1. **Processeur** : Vous pouvez allouer des **cores CPU** spécifiques aux VM, en les limitant ou en leur attribuant des priorités.
2. **Mémoire RAM** : Hyper-V permet de configurer la **mémoire dynamique**, ajustant la quantité de mémoire disponible pour chaque VM en fonction des besoins en temps réel.
3. **Disques durs virtuels (VHD/VHDX)** : Ce sont des fichiers qui agissent comme des disques durs physiques pour les machines virtuelles. Ils peuvent être de type **dynamique** (augmentation de la taille à la demande) ou **fixe** (taille définie dès le début).
4. **Réseau virtuel (Virtual Switch)** : Vous pouvez configurer des commutateurs virtuels pour connecter les machines virtuelles au réseau interne ou externe. Il existe trois types de commutateurs :
    - **Externe** : permet aux VMs de se connecter au réseau physique.
    - **Interne** : permet aux VMs de communiquer entre elles et avec l'hôte.
    - **Privé** : permet une communication uniquement entre les VMs, sans accès à l'hôte.

---

6. Sécurité dans Hyper-V

- **Isolation des VM** : Chaque machine virtuelle est isolée des autres, ce qui permet de créer des environnements de test sûrs et des applications isolées.
- **Contrôle d'accès** : Les administrateurs peuvent restreindre l'accès aux machines virtuelles en utilisant des groupes de sécurité ou des permissions spécifiques dans **Active Directory**.
- **Cryptage des disques VHDX** : Les disques durs virtuels peuvent être chiffrés pour une sécurité accrue, en particulier dans les environnements sensibles.

---

7. Hyper-V et Intégration avec d'autres outils

Gestion avec PowerShell :

Hyper-V peut être géré en ligne de commande avec **PowerShell**, permettant d'automatiser la création, la gestion et la suppression des machines virtuelles. Quelques commandes courantes :

- `New-VM` : Créer une nouvelle machine virtuelle.
- `Start-VM` : Démarrer une machine virtuelle.
- `Stop-VM` : Arrêter une machine virtuelle.
- `Get-VM` : Obtenir des informations sur les VM existantes.

Gestion avec System Center Virtual Machine Manager (SCVMM) :

Pour des déploiements plus grands et plus complexes, **SCVMM** peut être utilisé pour gérer plusieurs hôtes Hyper-V, effectuer des migrations de machines virtuelles entre hôtes, gérer les ressources, et automatiser certaines tâches.

---

8. Cas d'Utilisation d'Hyper-V

1. **Virtualisation des serveurs** : Les entreprises utilisent Hyper-V pour créer des environnements de serveurs virtuels, réduisant ainsi le nombre de serveurs physiques nécessaires.
2. **Environnements de test et développement** : Hyper-V permet de créer rapidement des machines virtuelles pour tester des applications, des systèmes d'exploitation, ou des configurations sans risquer de perturber l'environnement principal.

3. **Cloud privé avec Hyper-V** : En utilisant Hyper-V avec **System Center**, les entreprises peuvent créer des **clouds privés** pour déployer des machines virtuelles à la demande et gérer leurs ressources informatiques de manière flexible.
4. **Disaster Recovery et haute disponibilité** : En utilisant des fonctionnalités telles que la réplication Hyper-V (Hyper-V Replica), les entreprises peuvent assurer la récupération après sinistre en répliquant les VMs sur un site distant.

---

9. Ressources complémentaires

- **Documentation officielle de Microsoft Hyper-V** : Hyper-V Overview
- **Cours vidéo** : YouTube, Microsoft Learn – Tutoriels pratiques pour configurer et administrer Hyper-V
- **Forum et communauté** : Stack Overflow, Microsoft TechNet pour poser des questions et consulter les problèmes courants.

Résumé du Module

- **Hyper-V** est une plateforme de virtualisation native pour Windows Server et Windows 10/11, permettant de créer, gérer et automatiser des machines virtuelles.
- Elle offre une gestion fine des ressources, une sécurité renforcée et une intégration avec des outils comme PowerShell et SCVMM pour des déploiements complexes.
- **Virtual Switch**, **réseaux privés** et **réplication Hyper-V** sont des fonctionnalités clés pour une gestion efficace de l'infrastructure virtuelle.

Module 16 : Révision des principes du réseau IP pour le Cloud

---

Objectifs pédagogiques

À la fin de ce module, vous serez capable de :

- Comprendre les concepts de base du réseau IP appliqués dans un environnement Cloud.
- Expliquer la gestion des sous-réseaux et des adresses IP publiques et privées.
- Identifier les différents types de réseaux utilisés dans le Cloud (réseau privé, public, hybride, etc.).
- Savoir configurer un réseau IP dans un environnement Cloud, que ce soit sur AWS, Azure, ou Google Cloud.
- Gérer la connectivité entre des ressources cloud, des machines virtuelles et des réseaux externes.

---

1. Introduction au réseau IP pour le Cloud

Qu'est-ce qu'un réseau IP ?

Un **réseau IP (Internet Protocol)** est un ensemble de règles qui régissent l'adressage, le routage, et la gestion de la communication entre les ordinateurs sur un réseau, que ce soit dans un **réseau local (LAN)** ou sur l'**internet**.

Les principes de base des réseaux IP sont applicables dans tous les environnements, y compris les infrastructures **Cloud**. Dans un Cloud, les réseaux sont souvent virtuels et peuvent être configurés dynamiquement, mais les principes sous-jacents restent les mêmes.

🌍 2. Types d'adresses IP : Publiques et Privées

A. Adresses IP privées

Les **adresses IP privées** sont utilisées dans un réseau interne (local) et ne sont pas routables sur Internet. Elles sont définies par des plages spécifiques par l'IETF (Internet Engineering Task Force), telles que :

- **10.0.0.0 à 10.255.255.255**
- **172.16.0.0 à 172.31.255.255**
- **192.168.0.0 à 192.168.255.255**

Dans un environnement Cloud, les machines virtuelles ou les ressources sont souvent attribuées des adresses IP privées pour communiquer entre elles au sein d'un réseau privé.

B. Adresses IP publiques

Les **adresses IP publiques** sont routables sur Internet. Elles sont attribuées par un fournisseur de services Internet (ISP) et permettent de rendre une ressource, comme une machine virtuelle (VM), accessible depuis l'extérieur du réseau.

Les services Cloud, comme AWS, Azure, ou Google Cloud, peuvent attribuer des adresses IP publiques aux ressources afin qu'elles puissent communiquer avec d'autres ressources externes.

🔌 3. Sous-réseaux (Subnets) et Masques de Sous-réseaux

Un **sous-réseau** (ou **subnet**) est une division logique d'un réseau plus grand en plusieurs segments. Dans un environnement Cloud, il est courant de diviser les réseaux en sous-réseaux pour mieux contrôler le routage et la gestion des ressources.

A. Masques de sous-réseaux

Le **masque de sous-réseau** définit la taille du réseau et détermine quelles parties de l'adresse IP correspondent au réseau et quelles parties correspondent à l'hôte. Par exemple :

- **255.255.255.0** permet de créer un sous-réseau de 256 adresses IP (dont 2 sont réservées pour l'adresse réseau et l'adresse de diffusion).

B. Création de sous-réseaux dans le Cloud

Dans un Cloud comme **AWS VPC (Virtual Private Cloud)** ou **Azure Virtual Network**, vous devez définir les plages d'adresses IP et diviser ces plages en sous-réseaux pour isoler et sécuriser vos ressources.

🏢 4. Réseaux privés et publics dans le Cloud

A. Réseau privé (Private Network)

Un **réseau privé** dans le Cloud est une section isolée du Cloud, accessible uniquement aux ressources qui se trouvent à l'intérieur du même réseau ou à travers des connexions VPN (Virtual Private Network). Cela permet de mieux sécuriser les ressources sensibles et de contrôler l'accès.

Exemples d'outils :

- **Amazon VPC** : Vous pouvez définir des sous-réseaux privés dans une VPC.
- **Azure Virtual Network** : Permet la création de réseaux privés pour vos machines virtuelles.

B. Réseau public (Public Network)

Un **réseau public** permet aux ressources d'être accessibles depuis l'extérieur du Cloud. Par exemple, une **machine virtuelle publique** peut avoir une **adresse IP publique** pour être accessible depuis Internet.

Les ressources comme les **load balancers**, les **serveurs web**, et les **bases de données** peuvent être mises dans des sous-réseaux publics pour offrir un accès direct.

C. Réseau hybride (Hybrid Network)

Les réseaux hybrides combinent des éléments de réseaux privés et publics. Par exemple, une entreprise peut avoir des ressources dans un Cloud privé, tout en utilisant une **connexion VPN** ou une **connexion directe** pour relier son réseau local au Cloud.

---

5. Configuration des connexions réseau dans le Cloud

A. Routage dans le Cloud

Le **routage** détermine comment les paquets de données sont envoyés entre les sous-réseaux dans un Cloud. En fonction de la configuration de vos sous-réseaux, vous pouvez définir des tables de routage pour diriger le trafic vers les destinations appropriées.

- **Routage privé** : Le trafic entre les VM d'un réseau privé est routé uniquement au sein de ce réseau.
- **Routage public** : Le trafic vers Internet est routé via une **gateway** (passerelle) ou un **Internet Gateway**.

B. NAT (Network Address Translation)

Le **NAT** est utilisé pour permettre à des ressources privées (n'ayant pas d'adresse IP publique) d'accéder à Internet. En utilisant un **NAT Gateway** ou un **NAT instance**, des ressources privées peuvent envoyer du trafic vers Internet tout en restant non accessibles directement depuis l'extérieur.

- **NAT Gateway** (Cloud AWS) : Permet aux instances dans un sous-réseau privé d'accéder à Internet sans exposer leur adresse IP privée.

---

6. Sécurité des réseaux IP dans le Cloud

A. Contrôle d'accès (Security Groups et NACL)

Dans un environnement Cloud, vous pouvez configurer des règles de sécurité pour contrôler l'accès aux ressources en fonction de l'adresse IP, du port et du protocole. Deux éléments essentiels sont souvent utilisés pour gérer la sécurité :

- **Security Groups** (SG) : Ce sont des pare-feu virtuels pour contrôler l'accès entrant et sortant des ressources (VM, bases de données, etc.).
- **Network ACLs** : Des listes de contrôle d'accès pour gérer le trafic entrant et sortant au niveau du sous-réseau.

B. VPN et Direct Connect

- **VPN (Virtual Private Network)** : Vous pouvez configurer un tunnel VPN pour sécuriser les connexions entre votre réseau local et un réseau privé dans le Cloud.
- **Direct Connect** : Un service qui permet de créer une connexion réseau dédiée, privée et à haut débit entre vos ressources locales et le Cloud.

💻 7. Cas d'Utilisation des réseaux IP dans le Cloud

A. Hébergement de sites web (réseaux publics et privés)

- Les **serveurs web** et les **load balancers** sont souvent placés dans des sous-réseaux publics, tandis que les **bases de données** ou **serveurs d'application** peuvent être placés dans des sous-réseaux privés pour plus de sécurité.

B. Applications multi-tier (multi-niveaux)

Les applications **multi-tier** (par exemple, une application web avec une base de données) peuvent être configurées dans des sous-réseaux séparés. Par exemple :

- **Tier 1** : Serveurs web dans un sous-réseau public.
- **Tier 2** : Serveurs d'application dans un sous-réseau privé.
- **Tier 3** : Base de données dans un sous-réseau privé et sécurisé.

📚 8. Ressources complémentaires

- **AWS VPC Documentation** : AWS VPC Overview
- **Azure Networking** : Azure Networking Concepts
- **Google Cloud Networking** : Google Cloud VPC

🎯 Résumé du Module

- Le **réseau IP** dans le Cloud repose sur des concepts classiques de l'adressage, de la gestion des sous-réseaux et des routes.
- Vous pouvez configurer des **réseaux privés** et **publics**, gérer les **connexions sécurisées** via VPN et Direct Connect, et **contrôler l'accès** avec des outils comme les **Security Groups**.
- **NAT**, **routage privé/public** et l'utilisation de **NACL** et **Security Groups** permettent une gestion avancée des communications réseau dans un environnement Cloud.