

Master Data Science et Big Data

Année Universitaire 2024-2025

Mémoire de projet de fin de Module :



Mise en place d'une infrastructure DevOps
locale avec CI/CD, Kubernetes, et Monitoring



Réaliser par :

OSSAMA EL OUADIH

BENICHE Mohamed Fadel

EL KAISSOUNI El Mehdi

Encadrer par : **Fouzia Benabbou**



Remerciement

La réalisation de ce projet de fin de module n'aurait pas été possible sans le soutien et l'aide précieuse de nombreuses personnes. À travers ces remerciements, nous tenons à exprimer notre gratitude envers tous ceux qui ont contribué, de près ou de loin, à la réussite de ce travail.

Nos remerciements les plus profonds vont à nos professeurs et encadrants, dont la compétence, la disponibilité et les précieux conseils ont été déterminants dans l'aboutissement de ce projet. Un remerciement particulier à Mme **Fouzia Benabbou**, pour son encadrement exceptionnel, sa patience et son soutien tout au long de ce projet.

Nous remercions également nos amis et collègues de promotion, pour leur camaraderie, leurs discussions enrichissantes et leur soutien moral. Ce semestre a été bien plus agréable grâce à vous.

Enfin, nous remercions toutes les personnes qui, de près ou de loin, ont apporté leur aide et leur soutien à la réalisation de ce projet. Votre contribution, même indirecte, a été précieuse et appréciée.

Tout d'abord, nous remercions sincèrement nos parents pour leur amour, leur soutien inconditionnel et leurs sacrifices. Leur confiance en nos capacités nous a permis de surmonter les obstacles et d'avancer avec détermination.

Nous tenons également à exprimer notre gratitude envers nos familles, nos frères et sœurs, pour leur patience et leur encouragement constant. Votre présence a été une source de réconfort et de motivation.

Remerciement	3
Chapitre1 : Contexte général du projet.....	9
I. Contexte général du Projet	10
II. Objectifs et Détails du Projet.....	10
III. Problématique du Projet	11
IV. Cahier de charges :	11
Description du Projet.....	11
I. Objectifs Visés	11
1. Création et déploiement d'une application conteneurisée	11
2. Mise en place d'une chaîne CI/CD locale	11
3. Orchestration avec Kubernetes.....	11
4. Utilisation de Terraform et Ansible	12
II. Étapes Principales du Projet.....	12
1. Architecture du projet.....	12
2. Création des machines virtuelles avec Terraform.....	12
3. Initialisation de l'infrastructure	12
4. Automatisation avec Terraform et Ansible	12
5. Développement et conteneurisation de l'application.....	12
6. Création du pipeline CI/CD	12
III-Livrables attendus :	12
Chapitre2 : Conception et Architecture	14
1. Introduction à l'Architecture	15
2. Composants Principaux	15
2.1 Développeur et GitHub.....	15
2.2 Jenkins et Docker.....	15
2.3 AWS et Kubernetes.....	16
2.4 Machine Locale et Outils d'Infrastructure.....	16
3. Flux de Travail.....	16
4. Avantages de l'Architecture	16
5. Conclusion	16
Chapitre 3 : Réalisation du projet.....	17
Introduction.....	18
Partie 1 : Configuration de l'environnement en local	18
Introduction.....	18
Pré-requis avant la mise en place.....	18
Mise en place avec Terraform et Ansible	19

1. Installation et configuration de Terraform.....	19
2. Installation et configuration d'Ansible	27
Exécution des codes	33
Partie 2 : Déploiement sur le cloud Azure.....	34
Introduction.....	34
Jenkins :	34
Configuration des plugins Jenkins :	34
Jenkinsfile :	36
Voici notre Jenkinsfile :.....	37
pipeline:.....	40
Docker :	40
Introduction.....	40
Note :	40
Docker file :.....	41
Kubernetes :.....	41
1. Introduction.....	41
2. <i>Prérequis</i>	42
3. configuration de Kubernetes.....	42
Vue de l'application apres le déploiement :.....	44
CONCLUSION GENERALE	45



Résumé du Projet

Ce projet vise à concevoir et déployer une infrastructure DevOps complète sur une machine cloud (AWS, GCP, AZURE) en utilisant les tiers gratuits disponibles. L'objectif principal est d'automatiser le déploiement d'une application conteneurisée à travers une chaîne CI/CD robuste et efficace.

Dans ce cadre, nous avons intégré plusieurs outils DevOps permettant de garantir l'automatisation des processus de développement, de test et de déploiement. Kubernetes a été utilisé pour orchestrer les conteneurs, assurant ainsi une scalabilité et une gestion optimisée des ressources. Des outils de monitoring ont été mis en place afin d'assurer une supervision en temps réel et de garantir la stabilité de l'infrastructure.

De plus, une attention particulière a été portée à la sécurisation de l'ensemble du pipeline DevOps. Des pratiques telles que le scanning des images de conteneurs, l'intégration de tests de sécurité automatisés et la mise en place d'une gestion des accès stricte ont été adoptées afin de minimiser les risques de vulnérabilité.

L'intégration et le déploiement continus (CI/CD) ont été optimisés en exploitant des solutions telles que Jenkins, GitHub Actions et GitLab CI/CD, permettant un suivi rigoureux et une automatisation avancée du cycle de vie des applications.

Ce projet permet d'expérimenter les bonnes pratiques DevOps et de simuler une infrastructure de bout en bout en environnement cloud. Il constitue une solution complète et efficace pour démontrer les avantages de l'automatisation et de l'orchestration des déploiements en entreprise.

Enfin, une documentation détaillée a été élaborée afin de faciliter la prise en main de l'infrastructure par de nouveaux utilisateurs. Cette documentation comprend des guides d'installation, des procédures de dépannage et des recommandations pour l'optimisation continue du système.

Introduction Générale

Dans un monde où l'agilité et l'automatisation sont devenues des éléments clés du développement logiciel, l'adoption des pratiques **DevOps** s'impose comme une nécessité pour les entreprises cherchant à optimiser leurs processus de déploiement et de gestion d'infrastructure. **DevOps** repose sur une collaboration étroite entre les équipes de développement et d'exploitation, permettant ainsi d'accélérer la mise en production des applications tout en garantissant leur fiabilité et leur scalabilité.

Ce projet s'inscrit dans cette dynamique en mettant en place une infrastructure **DevOps** locale basée sur des technologies cloud gratuites telles **qu'AWS, GCP et AZURE**. L'objectif est de concevoir **une chaîne d'intégration et de déploiement continu (CI/CD)** robuste et entièrement automatisée, facilitant le déploiement d'une application conteneurisée tout en assurant un monitoring efficace de l'environnement.

Chapitre1 : Contexte général du projet



Ce chapitre a pour objectif de positionner notre projet dans son cadre global. Nous commencerons par introduire **le contexte** qui justifie notre démarche, en mettant en avant **les problématiques et les enjeux** liés à l'adoption d'une infrastructure DevOps. Nous préciserons ensuite les **objectifs** visés ainsi que les principales fonctionnalités mises en place, tout en définissant les limites et les exigences du projet. Enfin, nous présenterons **le cahier de charge** employée pour sa réalisation, en expliquant les approches et outils utilisés afin d'assurer une mise en œuvre efficace et structurée.

I. Contexte général du Projet

Avec l'essor du cloud computing et la nécessité croissante d'une livraison rapide et fiable des logiciels, les entreprises modernisent leurs processus de développement et de déploiement. Les approches traditionnelles présentent des limites telles que des délais longs, un manque de flexibilité et une gestion inefficace des ressources.

Le paradigme DevOps émerge alors comme une solution incontournable. Il favorise une collaboration étroite entre les équipes de développement et d'exploitation afin d'accélérer le déploiement des applications tout en garantissant leur qualité et leur résilience. L'automatisation des processus, l'orchestration des infrastructures et la surveillance continue deviennent ainsi des éléments essentiels pour assurer l'efficacité et la stabilité des services informatiques.

Dans ce contexte, notre projet vise à mettre en place une infrastructure DevOps complète sur une machine cloud gratuite (AWS, GCP, AZURE). L'objectif est de concevoir un pipeline CI/CD automatisé permettant le déploiement d'une application conteneurisée avec Kubernetes et un système de monitoring performant.

II. Objectifs et Détails du Projet

Ce projet consiste à créer une infrastructure DevOps locale intégrant des solutions modernes pour l'intégration et le déploiement continu (CI/CD), la gestion avancée des conteneurs et la supervision des services. Kubernetes est utilisé pour l'orchestration des conteneurs, garantissant scalabilité et gestion optimisée des ressources. Des outils de monitoring permettent un suivi en temps réel afin d'assurer la stabilité du système.

Nous avons également mis en place des mesures de sécurité, incluant le scanning des images de conteneurs, l'intégration de tests automatisés et une gestion stricte des accès. L'automatisation du pipeline CI/CD repose sur des outils tels que Jenkins, GitHub Actions et GitLab CI/CD, permettant un suivi rigoureux et une exécution fluide des processus.

Cette infrastructure DevOps offre une simulation complète d'un environnement de production en exploitant les solutions cloud gratuites, facilitant ainsi la compréhension et l'adoption des meilleures pratiques DevOps. Enfin, une documentation détaillée a été réalisée pour accompagner les utilisateurs dans l'installation et l'exploitation du système.

III. Problématique du Projet

L'adoption de DevOps est essentielle pour garantir un développement logiciel rapide et efficace. Cependant, la mise en place d'une infrastructure DevOps complète présente plusieurs défis, notamment l'automatisation des déploiements, l'orchestration des conteneurs, la gestion des ressources cloud et la surveillance en temps réel des applications. De plus, l'utilisation de solutions cloud gratuites impose des contraintes en termes de performances et de sécurité.

Comment concevoir une infrastructure DevOps optimisée qui répond à ces exigences tout en étant accessible et reproductible pour différents environnements de développement ?

IV. Cahier de charges :

Description du Projet

L'objectif de ce projet est de concevoir et déployer une infrastructure DevOps complète sur une machine cloud (AWS, GCP, Azure) en utilisant les ressources gratuites fournies par ces fournisseurs. Cette infrastructure permettra de déployer une application conteneurisée en utilisant une chaîne CI/CD automatisée. Le projet vise à appliquer les principes DevOps et à simuler une infrastructure de bout en bout.

I. Objectifs Visés

1. Création et déploiement d'une application conteneurisée

- **Application simple** : Développement d'une application web
- **Conteneurisation** : Dockeriser l'application pour faciliter son déploiement dans un environnement cloud.

2. Mise en place d'une chaîne CI/CD locale

- **Choix d'un outil CI** : Sélection d'un outil CI adapté pour automatiser le processus de build, test et déploiement. Par exemple Jenkins ou GitLab.
- **Automatisation des processus** : Mise en place d'un pipeline d'intégration et de déploiement continu pour l'application conteneurisée.
-

3. Orchestration avec Kubernetes

- **Installation d'un cluster Kubernetes** : Configuration d'un cluster Kubernetes (master node et worker node).
- **Déploiement de l'application** : Déploiement de l'application conteneurisée dans le cluster Kubernetes pour l'orchestration.

4. Utilisation de Terraform et Ansible

- **Terraform** : Automatisation de l'installation et de la configuration du cluster Kubernetes et des ressources cloud nécessaires (ex: machines virtuelles).
- **Ansible** : Automatisation du déploiement des configurations sur les nœuds du cluster. Utilisation d'un inventaire dynamique si nécessaire.

II. Étapes Principales du Projet

1. Architecture du projet

- **Conception de l'architecture** : Réalisation d'un diagramme d'architecture du projet, pour définir les composants et la communication entre eux.

2. Création des machines virtuelles avec Terraform

- **Utilisation d'un provider cloud** : Création des machines virtuelles en utilisant AWS.

3. Initialisation de l'infrastructure

- **Installation de Docker et Kubernetes** : Déploiement de Docker et Kubernetes sur les machines cloud (prévoir un nœud master et un nœud worker pour Kubernetes).
- **Installation de Terraform et Ansible** : Installation de Terraform et Ansible sur la machine locale pour gérer l'infrastructure.

4. Automatisation avec Terraform et Ansible

- **Scripts Terraform** : Écriture des scripts Terraform pour provisionner l'infrastructure, gérer les ressources cloud et configurer le cluster Kubernetes.
- **Playbooks Ansible** : Création des playbooks Ansible pour configurer les nœuds (ex: mise à jour des logiciels, déploiement de l'application).

5. Développement et conteneurisation de l'application

- **Création de l'application** : Développement d'une application simple.
- **Conteneurisation avec Docker** : Écriture d'un Dockerfile pour créer l'image de l'application et faciliter son déploiement.
-

6. Création du pipeline CI/CD

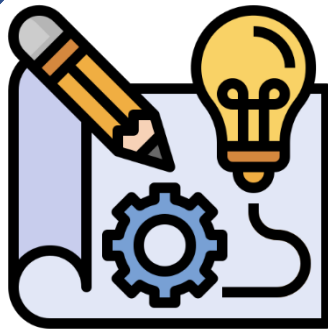
- **Configuration du pipeline CI/CD** : Configuration d'un pipeline CI/CD avec un outil tel que GitLab CI ou Jenkins.
- **Automatisation du déploiement** : Automatisation du déploiement de l'application dans le cluster Kubernetes après les tests via le pipeline CI/CD.

III-Livrables attendus :

1. Code source de l'application et Dockerfile.
2. Configuration du pipeline CI/CD.

3. Scripts Terraform et playbooks Ansible.
4. Manifests Kubernetes (YAML) pour le déploiement de l'application.
5. Monitoring Optionnel (grafana, loki, Prometheus, promtail...)
6. Documentation détaillée pour reproduire l'infrastructure.

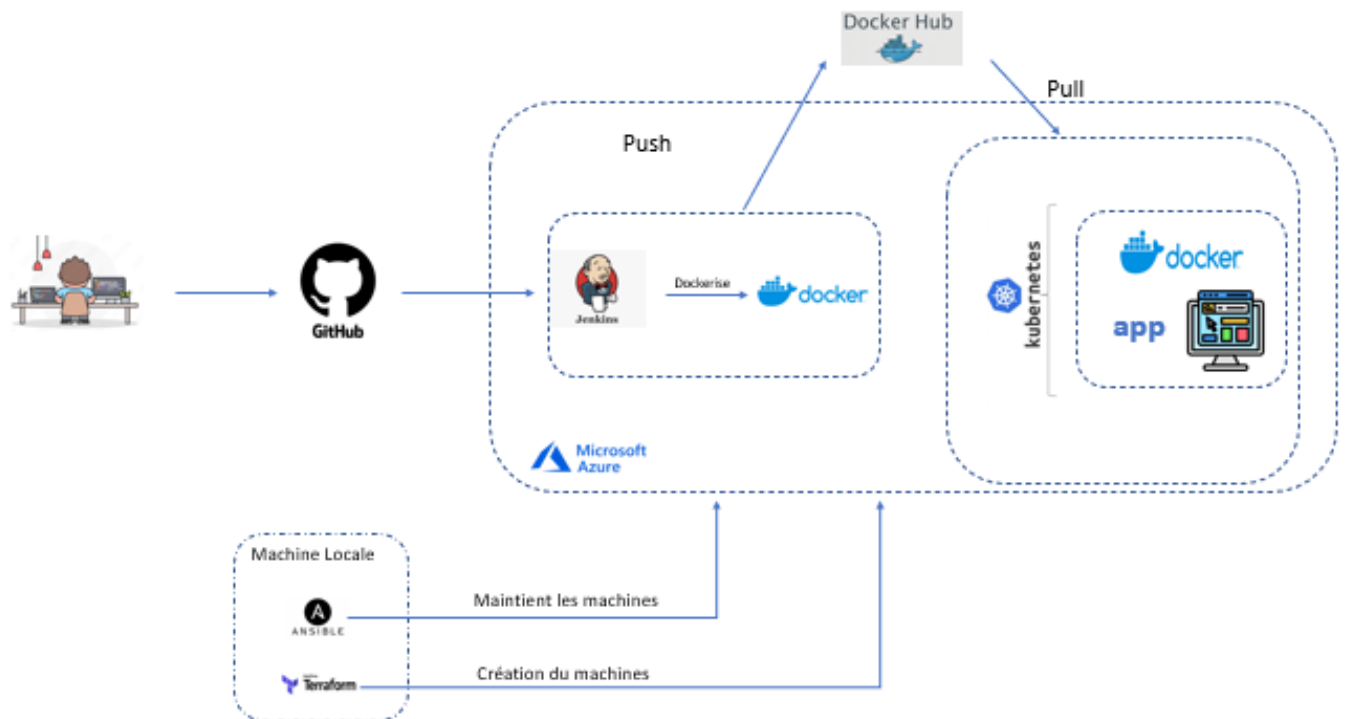
Chapitre2 : Conception et Architecture



Une architecture bien conçue est essentielle pour garantir la performance, la scalabilité et la maintenabilité d'un système. Dans ce chapitre, nous allons détailler **l'architecture générale** de notre infrastructure DevOps, en expliquant le rôle et l'interaction de chaque composant clé. Nous décrirons également **le flux de travail**, qui illustre le cheminement des processus depuis le développement jusqu'au déploiement. Enfin, nous mettrons en avant **les avantages** de cette architecture, notamment en termes d'**automatisation**, de **flexibilité** et d'**efficacité opérationnelle**.

1. Introduction à l'Architecture

L'architecture du projet est conçue pour assurer une intégration continue et une livraison continue (CI/CD) efficace, tout en garantissant la scalabilité, la fiabilité et la maintenance facile des applications. Le schéma ci-dessous illustre les différents composants et leur interaction.



2. Composants Principaux

2.1 Développeur et GitHub

- **Développeur** : Les développeurs travaillent sur leurs machines locales pour coder, tester et valider les fonctionnalités.
- **GitHub** : Le code source est versionné et stocké sur GitHub. Cela permet une collaboration facile entre les développeurs et assure un suivi précis des modifications.

2.2 Jenkins et Docker

- **Jenkins** : Jenkins est utilisé comme outil d'intégration continue. Il exécute automatiquement des tests et construit les images Docker dès qu'un nouveau commit est pushé sur GitHub.
- **Docker** : Les applications sont containerisées avec Docker pour assurer une cohérence environnementale et faciliter le déploiement.

2.3 AWS et Kubernetes

- **Azure** : Microsoft Azure est utilisé pour héberger les ressources nécessaires au déploiement et à l'exécution des applications. Cela comprend la gestion des machines virtuelles et des services cloud.
- **Kubernetes** : Kubernetes orchestre les conteneurs Docker déployés sur AWS. Il gère l'équilibrage de charge, la redondance et la mise à l'échelle automatique des applications.

2.4 Machine Locale et Outils d'Infrastructure

- **Machine Locale** : La machine locale du développeur utilise Ansible et Terraform pour automatiser la création et la maintenance des machines sur AWS.
- **Ansible** : Ansible est utilisé pour configurer et maintenir les machines virtuelles.
- **Terraform** : Terraform est utilisé pour créer et gérer les infrastructures cloud sur AWS.

3. Flux de Travail

Le flux de travail commence lorsque le développeur pousse un nouveau commit sur GitHub. Jenkins détecte ce changement, exécute les tests et construit une nouvelle image Docker. Cette image est ensuite poussée vers Docker Hub. Kubernetes tire cette image mise à jour depuis Docker Hub et déploie les nouvelles versions des applications sur les machines gérées par Azure. En parallèle, Ansible et Terraform s'occupent de la création et de la maintenance des machines sur Azure.

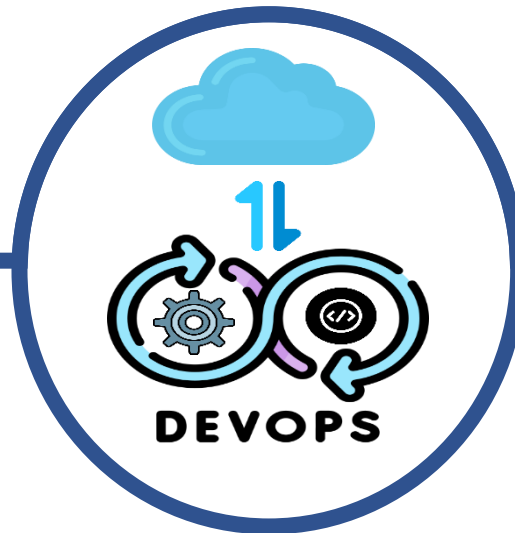
4. Avantages de l'Architecture

- **Scalabilité** : L'utilisation de Kubernetes permet une mise à l'échelle automatique des applications selon la charge.
- **Fiabilité** : L'orchestration par Kubernetes assure la redondance et la haute disponibilité des services.
- **Maintenance Facile** : L'automatisation via Ansible et Terraform simplifie la gestion de l'infrastructure.
- **Consistance Environnementale** : Docker garantit que les applications fonctionnent de manière identique sur toutes les machines.

5. Conclusion

Cette architecture offre une solution robuste et flexible pour le développement, le déploiement et la maintenance des applications. Elle optimise les processus de CI/CD tout en assurant une performance optimale et une maintenance aisée.

Chapitre 3 : Réalisation du projet



La mise en place d'une infrastructure DevOps passe par plusieurs étapes essentielles, allant de la configuration initiale à son déploiement final. Dans ce chapitre, nous détaillerons la réalisation du projet, qui se divise en deux parties principales. La première concerne l'installation locale, où nous préparerons l'environnement en installant et configurant les outils nécessaires. La seconde partie portera sur le déploiement sur le cloud, où nous provisionnerons les ressources et automatiserons le déploiement de l'application. Cette approche permet d'assurer une transition fluide entre le développement local et un environnement cloud optimisé, garantissant ainsi une infrastructure performante et reproductible.

Introduction

Dans ce chapitre, nous allons détailler les étapes permettant de reproduire l'infrastructure DevOps mise en place. Ces étapes sont divisées en deux parties principales :

1. **Configuration de l'environnement en local** : Mise en place des outils nécessaires sur une machine locale pour préparer l'infrastructure et l'automatisation.
2. **Déploiement sur le cloud Azure** : Création et orchestration des ressources sur Azure afin d'héberger l'application dans un environnement scalable et fiable.

Partie 1 : Configuration de l'environnement en local

Introduction

Avant de déployer l'infrastructure sur Azure, il est essentiel de configurer l'environnement en local. Cette étape comprend l'installation et la configuration des outils Terraform et Ansible, qui permettront d'automatiser le provisionnement et la gestion de l'infrastructure.

- **Terraform** : Un outil d'Infrastructure as Code (IaC) permettant de définir et de provisionner une infrastructure cloud de manière déclarative.
- **Ansible** : Un outil d'automatisation utilisé pour la gestion de configuration, l'installation de logiciels et l'orchestration des serveurs.

Pré-requis avant la mise en place

Avant de commencer à utiliser Terraform et Ansible, il est nécessaire de remplir certains pré-requis :

1. **Création d'un compte Azure** : Un compte AWS est indispensable pour provisionner des instances et gérer l'infrastructure.
2. **Installation de l'interface de connexion Azure CLI** : Cet outil permet d'interagir avec AWS directement via la ligne de commande, en établissant une connexion entre Terraform et Azure .
3. **Établissement de la connexion SSH** : Afin d'utiliser Ansible pour la gestion des configurations, il est essentiel que votre machine locale puisse se connecter aux instances via SSH.



Terraform est utilisé pour définir l'infrastructure sur AWS sous forme de code. En utilisant des fichiers de configuration HCL (HashiCorp Configuration Language), vous pouvez déclarer les ressources nécessaires à l'infrastructure.

Code Terraform pour la création des machines

Le code suivant crée deux machines : une pour le master de Kubernetes et une pour le worker, tout en ajoutant une clé SSH pour sécuriser la connexion à ces instances.

```
terraform {  
  
  required_providers {  
  
    azurerm = {  
  
      source  = "hashicorp/azurerm"  
  
      version = "~>3.0"  
  
    }  
  
  }  
  
}  
  
provider "azurerm" {  
  
  features {}  
  
  
  subscription_id = var.subscription_id  
  
  client_id       = var.client_id  
  
  client_secret   = var.client_secret
```

```

    tenant_id      = var.tenant_id
}

# Define sensitive variables

variable "subscription_id" { default = "5819eb20-62fc-4335-834c-
0b29fe7da229" }

variable "client_id" { default = "ccbd9d71-90b5-4cae-88c8-87ab9f4b5059" }

variable "client_secret" { default =
"m9I8Q~8Q2nFzGvOUuPlFveUBfevxFneRsayGTdr2" }

variable "tenant_id" { default = "7dbac5dd-7698-436f-86c6-d770d076a516" }

# Create Resource Group (Updated Name)

resource "azurerm_resource_group" "rg" {

    name      = "rg-devops-new" # Changed from "rg-devops" to "rg-devops-
new"

    location = "East US"
}

# Create Virtual Network

resource "azurerm_virtual_network" "vnet" {

    name                = "vnet-devops"

    resource_group_name = azurerm_resource_group.rg.name

    location            = azurerm_resource_group.rg.location

    address_space       = ["10.0.0.0/16"]
}

# Create Subnet

resource "azurerm_subnet" "subnet" {

    name                = "subnet-devops"

```

```

    resource_group_name = azurerm_resource_group.rg.name

    virtual_network_name = azurerm_virtual_network.vnet.name

    address_prefixes      = ["10.0.1.0/24"]
}

# Create Network Security Group (NSG)

resource "azurerm_network_security_group" "nsg" {

    name                = "devops-nsg"

    resource_group_name = azurerm_resource_group.rg.name

    location            = azurerm_resource_group.rg.location

# Allow SSH (Port 22)

    security_rule {

        name                = "AllowSSH"

        priority            = 100

        direction          = "Inbound"

        access              = "Allow"

        protocol            = "Tcp"

        source_port_range   = "*"

        destination_port_range = "22"

        source_address_prefix = "*"

        destination_address_prefix = "*"

    }

}

# Create Public IPs (Static)

resource "azurerm_public_ip" "pip_master" {

    name                = "pip-master"

```

```

resource_group_name = azurerm_resource_group.rg.name

location            = azurerm_resource_group.rg.location

allocation_method   = "Static"
}

resource "azurerm_public_ip" "pip_worker" {

  name                = "pip-worker"

  resource_group_name = azurerm_resource_group.rg.name

  location            = azurerm_resource_group.rg.location

  allocation_method   = "Static"
}

# Create Network Interfaces (Attach to NSG)

resource "azurerm_network_interface" "nic_master" {

  name                = "nic-master"

  resource_group_name = azurerm_resource_group.rg.name

  location            = azurerm_resource_group.rg.location

  ip_configuration {

    name                = "internal"

    subnet_id           = azurerm_subnet.subnet.id

    private_ip_address_allocation = "Static"

    private_ip_address    = "10.0.1.10"

    public_ip_address_id  = azurerm_public_ip.pip_master.id

  }

}

resource "azurerm_network_interface" "nic_worker" {

```

```

name                = "nic-worker"

resource_group_name = azurerm_resource_group.rg.name

location            = azurerm_resource_group.rg.location

ip_configuration {
  name                = "internal"

  subnet_id          = azurerm_subnet.subnet.id

  private_ip_address_allocation = "Static"

  private_ip_address      = "10.0.1.11"

  public_ip_address_id    = azurerm_public_ip.pip_worker.id
}
}

# Attach NSG to Network Interfaces

resource "azurerm_network_interface_security_group_association"
"master_nsg" {

  network_interface_id      = azurerm_network_interface.nic_master.id

  network_security_group_id = azurerm_network_security_group.nsg.id
}

resource "azurerm_network_interface_security_group_association"
"worker_nsg" {

  network_interface_id      = azurerm_network_interface.nic_worker.id

  network_security_group_id = azurerm_network_security_group.nsg.id
}

# Create Master Node (2 vCPUs, 4GB RAM)

resource "azurerm_linux_virtual_machine" "vm_master" {

  name                = "k8s-master"

```

```

resource_group_name = azurerm_resource_group.rg.name

location            = azurerm_resource_group.rg.location

size                = "Standard_B2ms" # 2 vCPUs, 8GB RAM

admin_username      = "azureuser"


network_interface_ids = [azurerm_network_interface.nic_master.id]


admin_ssh_key {
    username    = "azureuser"

    public_key = file("${path.module}/ssh_key/id_rsa.pub") # Correct path
}


os_disk {
    caching          = "ReadWrite"

    storage_account_type = "Standard_LRS"
}


source_image_reference {
    publisher = "Canonical"

    offer     = "UbuntuServer"

    sku       = "18.04-LTS"

    version   = "latest"
}


tags = {
    Role          = "Master"

    Environment    = "Dev"

    Project        = "DevOps"
}

```



```

    }
}

# Create Worker Node (1 vCPU, 2GB RAM)

resource "azurerm_linux_virtual_machine" "vm_worker" {

  name          = "k8s-worker"

  resource_group_name = azurerm_resource_group.rg.name

  location      = azurerm_resource_group.rg.location

  size          = "Standard_B2s" # 1 vCPU, 2GB RAM

  admin_username = "azureuser"

  network_interface_ids = [azurerm_network_interface.nic_worker.id]

  admin_ssh_key {

    username = "azureuser"

    public_key = file("${path.module}/ssh_key/id_rsa.pub") # Correct path
  }

  os_disk {

    caching          = "ReadWrite"

    storage_account_type = "Standard_LRS"
  }

  source_image_reference {

    publisher = "Canonical"

    offer     = "UbuntuServer"

    sku       = "18.04-LTS"

    version   = "latest"
  }
}

```

```

}

tags = {

  Role          = "Worker"

  Environment = "Dev"

  Project       = "DevOps"

}

}

```

Explication du code Terraform :

1. **provider "azurerm" :**
 Cette section définit le fournisseur **Azure** en utilisant le provider **azurerm** de HashiCorp, qui permet de gérer des ressources Azure avec Terraform. Les informations d'identification (subscription_id, client_id, client_secret, tenant_id) sont fournies sous forme de variables pour assurer la connexion sécurisée.
2. **resource "azurerm_resource_group" :**
 Cette ressource crée un **groupe de ressources** nommé **rg-devops-new** dans la région **East US**. Le groupe de ressources est l'entité logique qui permet d'organiser et de gérer toutes les ressources Azure associées.
3. **resource "azurerm_virtual_network" :**
 Cette ressource définit le **réseau virtuel (VNet)** avec l'adresse IP de plage **10.0.0.0/16**, qui sert de base pour la communication réseau entre les machines.
4. **resource "azurerm_subnet" :**
 Une **subnet** est créée pour allouer une partie du réseau virtuel avec l'adresse **10.0.1.0/24**.
5. **resource "azurerm_network_security_group" (NSG) :**
 Le **groupe de sécurité réseau** permet de définir des règles de sécurité. Une règle **AllowSSH** est configurée pour autoriser le trafic entrant sur le port **22** (SSH).
6. **resource "azurerm_public_ip" :**
 Cette ressource crée des **adresses IP publiques** pour le master et le worker afin de permettre l'accès à distance.
7. **resource "azurerm_network_interface" :**
 Deux **interfaces réseau (NIC)** sont créées, une pour chaque machine virtuelle. Chaque interface est associée à la subnet et à une adresse IP publique.
8. **resource "azurerm_network_interface_security_group_association" :**
 Cette ressource associe le groupe de sécurité réseau (NSG) à chaque interface réseau, afin d'appliquer les règles de sécurité.
9. **resource "azurerm_linux_virtual_machine" :**
 Cette section crée deux machines virtuelles Linux basées sur Ubuntu 18.04 :
 - Le **Master Node** utilise une machine de type **Standard_B2ms** (2 vCPUs, 8GB RAM).
 - Le **Worker Node** utilise une machine de type **Standard_B2s** (1 vCPU, 2GB RAM).
 Les machines sont accessibles via SSH avec la clé publique stockée dans le fichier **id_rsa.pub**.

Remarque :

Après l'exécution de ce code, l'infrastructure Azure sera créée avec :

- 1 machine virtuelle **Master Node**
- 1 machine virtuelle **Worker Node**
- Réseau virtuel avec une subnet
- Sécurité réseau pour autoriser les connexions SSH
- Adresses IP publiques pour accéder aux machines

2. Installation et configuration d'Ansible



Ansible est un outil puissant permettant de gérer la configuration des serveurs. Il utilise des **playbooks** écrits en YAML pour automatiser des tâches comme l'installation de logiciels, la configuration de services, etc.

Fichier d'inventaire Ansible

Le fichier d'inventaire spécifie les hôtes (les machines) sur lesquels Ansible va exécuter les tâches. Voici un exemple de fichier d'inventaire pour les nœuds Kubernetes.

```
[master]

k8s-master ansible_host=52.170.19.17 ansible_user=azureuser
ansible_ssh_private_key_file=/mnt/c/devops-project/ssh_key/id_rsa

[worker]

k8s-worker ansible_host=52.170.23.129 ansible_user=azureuser
ansible_ssh_private_key_file=/mnt/c/devops-project/ssh_key/id_rsa

[k8s: children]

master

worker
```

Explication du fichier d'inventaire:

- `[k8s_master]` : Ce groupe contient l'IP de l'instance master.
- `[k8s_worker]` : Ce groupe contient l'IP de l'instance worker.
- `ansible_ssh_private_key_file` : Spécifie le chemin du fichier de clé privée pour établir une connexion SSH sécurisée.

Playbook Ansible pour l'installation des dépendances

Le playbook suivant installe Docker, Kubernetes et Jenkins (sur le master uniquement) sur les instances définies dans l'inventaire.

```
---

- name: Suppression et Réinstallation de Kubernetes et Docker

  hosts: all

  become: yes

  tasks:

    # =====

    #  SUPPRESSION DE DOCKER & KUBERNETES

    # =====

- name: Arrêter et désactiver Docker et Kubelet

  service:

    name: "{{ item }}"

    state: stopped

    enabled: no

  loop:

    - docker

    - kubelet

  ignore_errors: yes

- name: Désinstaller Docker et Kubernetes

  apt:

    name:

      - docker.io

      - kubelet
```

```

        - kubeadm

        - kubect1

state: absent

- name: Supprimer les fichiers et configurations Docker & Kubernetes

file:

    path: "{{ item }}"

    state: absent

loop:

    - /var/lib/docker

    - /var/lib/kubelet

    - /etc/docker

    - /etc/kubernetes

    - /etc/apt/sources.list.d/kubernetes.list

    - /etc/apt/sources.list.d/docker.list

ignore_errors: yes

- name: Nettoyer les paquets inutiles

shell: apt autoremove -y && apt autoclean -y

# =====

#  INSTALLATION DE DOCKER

# =====

- name: Mettre à jour APT et installer les dépendances

apt:

    name:

        - apt-transport-https

```

```

    - ca-certificates

    - curl

    - gnupg

    - lsb-release

state: present

update_cache: yes


- name: Ajouter la clé GPG de Docker

    shell: curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo
tee /etc/apt/trusted.gpg.d/docker.asc


- name: Ajouter le dépôt Docker

    shell: add-apt-repository "deb [arch=amd64]
https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable"


- name: Installer Docker

apt:

    name: docker.io

    state: present

    update_cache: yes


- name: Activer et démarrer Docker

service:

    name: docker

    state: started

    enabled: yes


# =====

#  INSTALLATION DE KUBERNETES

```

```
# =====

- name: Ajouter la clé GPG de Kubernetes

    shell: curl -fsSL
https://pkgs.k8s.io/core:/stable:/v1.28/deb/Release.key | sudo tee
/etc/apt/trusted.gpg.d/kubernetes.asc

- name: Ajouter le dépôt Kubernetes

    shell: echo "deb https://pkgs.k8s.io/core:/stable:/v1.28/deb/ /" |
sudo tee /etc/apt/sources.list.d/kubernetes.list

- name: Mettre à jour APT et installer Kubernetes

apt:

    name:

        - kubelet

        - kubeadm

        - kubectl

    state: present

    update_cache: yes

- name: Activer et démarrer Kubelet

service:

    name: kubelet

    state: started

    enabled: yes

# =====

# VÉRIFICATION DES INSTALLATIONS

# =====
```

```
- name: Vérifier que Docker est installé

  command: which docker

  register: docker_installed

  failed_when: docker_installed.rc != 0

  changed_when: false


- name: Vérifier que kubelet est installé

  command: which kubelet

  register: kubelet_installed

  failed_when: kubelet_installed.rc != 0

  changed_when: false


- name: Vérifier que kubeadm est installé

  command: which kubeadm

  register: kubeadm_installed

  failed_when: kubeadm_installed.rc != 0

  changed_when: false


- name: Vérifier que kubectl est installé

  command: which kubectl

  register: kubectl_installed

  failed_when: kubectl_installed.rc != 0

  changed_when: false


- name: Vérifier l'état du cluster Kubernetes

  command: kubectl get nodes

  register: cluster_status
```



```

    failed_when: false

    changed_when: false

    when: kubectl_installed.rc == 0

- name: Afficher l'état du cluster

  debug:

    msg: "{{ cluster_status.stdout_lines }}"

    when: kubectl_installed.rc == 0

```

Explication du playbook :

- **Suppression de Docker et Kubernetes** : Arrête les services, désinstalle les paquets et supprime les fichiers de configuration.
- **Nettoyage des paquets inutiles** : Supprime les dépendances obsolètes pour libérer de l'espace.
- **Installation de Docker** : Met à jour APT, ajoute le dépôt Docker, installe Docker et le lance.
- **Installation de Kubernetes** : Ajoute le dépôt Kubernetes, installe kubelet, kubeadm et kubectl, puis active Kubelet.
- **Vérifications finales** : Vérifie que Docker et les outils Kubernetes sont installés et fonctionnels, puis affiche l'état du cluster si disponible.

Ce playbook permet de réinitialiser un système en supprimant puis réinstallant Docker et Kubernetes proprement.

Exécution des codes

Pour exécuter les codes Terraform et Ansible, voici les étapes :

1. **Exécuter Terraform** :
 - Initialisez Terraform : `terraform init`
 - Planifiez le déploiement : `terraform plan`
 - Appliquez les changements : `terraform apply`
2. **Exécuter Ansible** :
 - Vérifiez la connectivité SSH avec les instances : `ansible -m ping all`
 - Lancez le playbook : `ansible-playbook -i inventaire playbook.yml`

Partie 2 : Déploiement sur le cloud Azure

Introduction

Dans cette section, nous allons décrire comment automatiser et orchestrer le processus de déploiement de l'application en utilisant **Jenkins**, **Docker** et **Kubernetes**. Ces outils sont essentiels pour gérer l'intégration continue (CI), la livraison continue (CD) et le déploiement scalable de l'application. Nous allons également aborder la configuration des pipelines Jenkins pour automatiser la création des images Docker et leur déploiement sur un cluster Kubernetes.

Jenkins :



Jenkins est une plateforme open-source d'intégration continue (CI) et de livraison continue (CD) qui automatisent les processus de build, test et déploiement. Grâce à ses nombreux plugins, Jenkins s'adapte à divers environnements de développement, facilitant la collaboration entre équipes et assurant une livraison de code plus rapide et fiable.

Configuration des plugins Jenkins :

Pour travailler avec une application Spring Boot dans Jenkins, vous pouvez utiliser plusieurs plugins pour automatiser les différentes étapes du processus de build, test et déploiement. Voici quelques plugins couramment utilisés :

Pipeline Plugin

Le Pipeline Plugin (aussi connu sous le nom de Workflow Plugin) permet de définir des pipelines Jenkins en tant que code. Vous pouvez écrire des scripts Groovy pour automatiser complètement le processus de CI/CD. Cela inclut la compilation, l'exécution des tests, la construction de l'image Docker, et le déploiement sur Kubernetes ou tout autre environnement cible.

Docker Pipeline Plugin

Ce plugin facilite l'utilisation de Docker dans les pipelines Jenkins. Il permet de construire, de pousser et de tirer des images Docker directement depuis les pipelines Jenkins.

JUnit Plugin

Le JUnit Plugin permet d'agrégier les résultats des tests unitaires et d'intégration exécutés par Maven ou Gradle. Les rapports de tests sont affichés dans l'interface Jenkins, ce qui facilite le suivi des problèmes potentiels.

Publish Over SSH Plugin

Ce plugin peut être utile pour les applications sur un serveur distant via SSH. Il permet d'envoyer des fichiers et d'exécuter des commandes sur un serveur distant via SSH.

Git Plugin

Le Git Plugin est nécessaire. Il permet à Jenkins de cloner le dépôt Git, de récupérer les dernières modifications, et de gérer les branches et les tags.

Kubernetes Continuous Deploy Plugin

Ce plugin peut simplifier le processus de déploiement. Il permet de créer, mettre à jour ou supprimer des ressources Kubernetes directement depuis Jenkins.

La méthode la plus courante pour installer des plugins est d'utiliser l'interface web de Jenkins. Suivez ces étapes :

- Connectez-vous à votre instance Jenkins.
- Allez dans **Gérer Jenkins > Gérer les plugins**.
- Cliquez sur l'onglet **Disponibles**.
- Utilisez la barre de recherche pour trouver les plugins que vous souhaitez installer .
- Cochez les plugins que vous voulez installer et cliquez sur "Télécharger maintenant et installer après redémarrage".
- Une fois l'installation terminée, Jenkins vous demandera de redémarrer pour que les nouveaux plugins soient activés.

Voici notre script Groovy qui peut être exécuté via l'interface de gestion de script Jenkins (**Gérer Jenkins > Script Console**) :

```
def pluginList = [  
    "pipeline",  
    "docker-workflow",
```

```

    "junit",
    "publish-over-ssh",
    "git",
    "sonar",
    "kubernetes"
]

def jenkins = Jenkins.instance

pluginList.each { plugin ->
    if (!jenkins.getPluginManager().getPlugin(plugin)) {
        println "Installing ${plugin}..."
        def updateCenter = jenkins.getUpdateCenter()
        def pluginInfo = updateCenter.getPlugin(plugin)
        if (pluginInfo != null) {
            def installFuture = pluginInfo.deploy()
            while (!installFuture.isDone()) {
                sleep(1000)
            }
        } else {
            println "Plugin ${plugin} not found."
        }
    } else {
        println "Plugin ${plugin} already installed."
    }
}

jenkins.save()

```

Jenkinsfile :

Le Jenkinsfile joue un rôle crucial dans l'automatisation des processus de construction, de test et de déploiement en définissant le pipeline CI/CD comme code. Il s'agit d'un fichier texte qui contient les instructions Groovy pour exécuter les différentes étapes du pipeline Jenkins. En utilisant le Jenkinsfile, les développeurs peuvent versionner et gérer facilement les configurations du pipeline, assurant ainsi une cohérence et une reproductibilité des builds. De plus, il permet une flexibilité accrue dans la conception des pipelines complexes, en facilitant

l'intégration de divers outils et technologies, tels que Docker ou Kubernetes, tout en offrant une visibilité claire sur le flux de travail de développement continu.

Voici notre Jenkinsfile :

```
pipeline {
    agent any

    environment {
        GITHUB_PAT = credentials('github-pat') // GitHub Personal Access
Token
        DOCKER_CREDENTIALS = credentials('docker-hub-credentials') //
        Docker Hub Credentials
        // Ensure the image name includes your Docker Hub username.
        DOCKER_IMAGE = "${DOCKER_CREDENTIALS_USR}/age-calculator:latest"

        // Now the YAML files are stored in your repository
        K8S_DEPLOYMENT_FILE = 'age-calculator/deployment.yaml'
        K8S_SERVICE_FILE = 'age-calculator/service.yaml'
        K8S_DEPLOYMENT_NAME = 'age-calculator'
    }

    stages {
        stage('Clone Repository') {
            steps {
                sh """
                if [ -d "age-calculator/.git" ]; then
                    cd age-calculator
                    git reset --hard
                    git pull origin main
                else
                    git clone
https://${GITHUB_PAT}@github.com/goldentrader/age-calculator
                fi
                """
            }
        }
    }
}
```

```

    }
}

stage('Build Docker Image') {
    steps {
        dir('age-calculator') {
            sh "docker build -t ${DOCKER_IMAGE} ."
        }
    }
}

stage('Push to Docker Hub') {
    steps {
        script {
            sh """
                echo ${DOCKER_CREDENTIALS_PSW} | docker login -u
${DOCKER_CREDENTIALS_USR} --password-stdin
                docker push ${DOCKER_IMAGE}
            """
        }
    }
}

stage('Deploy to Kubernetes Worker') {
    steps {
        script {
            sshagent(['k8s-worker-ssh']) {
                sh """
                    # Copy the Deployment and Service YAML files to the
remote worker

                    scp -o StrictHostKeyChecking=no
${K8S_DEPLOYMENT_FILE} azureuser@k8s-worker:/tmp/deployment.yaml

                    scp -o StrictHostKeyChecking=no ${K8S_SERVICE_FILE}
azureuser@k8s-worker:/tmp/service.yaml

                    ssh -o StrictHostKeyChecking=no azureuser@k8s-
worker <<EOF

```

```

# Pull the latest Docker image on the worker node
docker pull ${DOCKER_IMAGE}

# Clean up old resources if they exist
kubectl delete deployment ${K8S_DEPLOYMENT_NAME} --
ignore-not-found
kubectl delete service age-calculator-service --
ignore-not-found

# Apply the new Deployment and Service
configurations
kubectl apply -f /tmp/deployment.yaml
kubectl apply -f /tmp/service.yaml

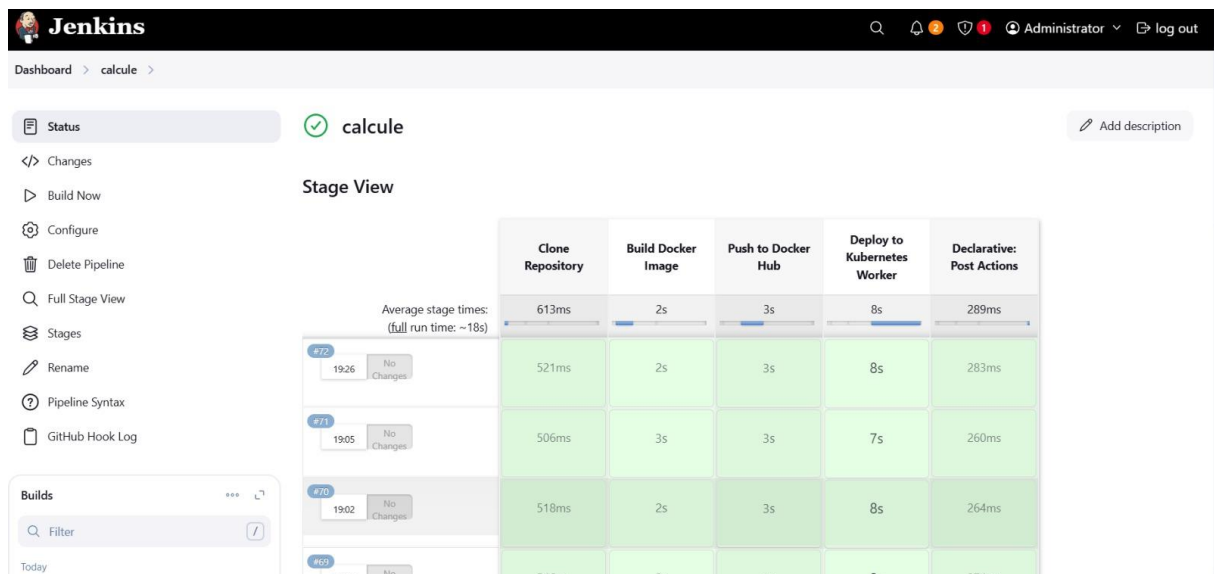
# Restart the deployment to pick up any changes and
wait for rollout
kubectl rollout restart
deployment/${K8S_DEPLOYMENT_NAME}
kubectl rollout status
deployment/${K8S_DEPLOYMENT_NAME} --timeout=180s
EOF

"""
    }
  }
}

post {
  success {
    echo 'Deployment successful! 🚀'
  }
  failure {
    echo 'Deployment failed! ❌'
  }
}

```

pipeline:



Docker :

Introduction

Docker est une plateforme open-source qui permet de créer, déployer et exécuter des applications dans des conteneurs isolés. Ces conteneurs encapsulent le code et toutes ses dépendances, garantissant que l'application fonctionne de manière cohérente sur n'importe quel environnement, qu'il s'agisse d'un ordinateur local, d'un serveur ou du cloud. Docker simplifie ainsi le développement, le test et le déploiement d'applications en offrant une portabilité, une isolation et une scalabilité optimales.



Note :

Notre Dockerfiles est conçu avec une approche méticuleuse visant à réduire la taille des images et à optimiser le déploiement. En utilisant des techniques telles que la minimisation des couches, l'élimination des fichiers inutiles et l'utilisation d'images de base allégées, nous avons réussi à créer des images compactes qui ne contiennent que les éléments essentiels nécessaires au fonctionnement de nos applications. Cette approche non seulement accélère le processus de build et de déploiement.

Docker file :

```
FROM nginx:alpine

# Set the working directory
WORKDIR /usr/share/nginx/html

# Copy the application files to the container
COPY . /usr/share/nginx/html

# Expose custom ports
EXPOSE 3000 5000
```

Kubernetes :



kubernetes

Installation de Kubernetes sur les machines (Master et Worker)

1. Introduction

Kubernetes est un système d'orchestration de conteneurs qui permet d'automatiser le déploiement, la mise à l'échelle et la gestion d'applications conteneurisées. Dans notre projet, nous avons installé un cluster Kubernetes sur Azure en utilisant deux machines virtuelles.

Une pour le nœud master et une autre pour le nœud worker.

2. Prérequis

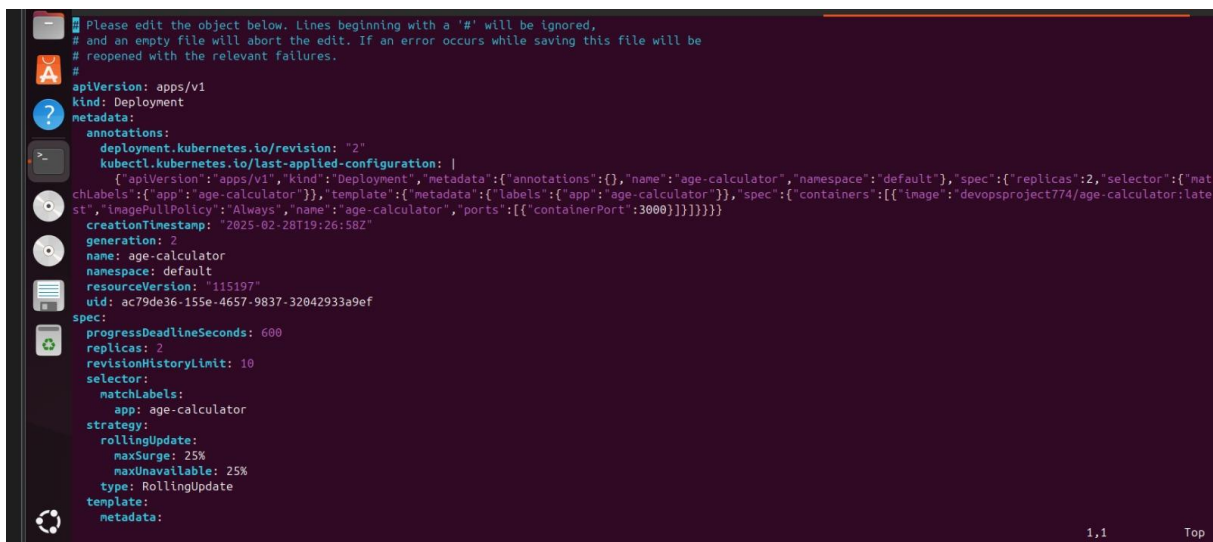
Avant l'installation de Kubernetes, il est nécessaire de s'assurer que les conditions suivantes sont remplies :

- Deux instances k8s fonctionnant sous Ubuntu 18.
- Accès SSH configuré entre les instances.
- Docker installé sur chaque machine (Docker est un prérequis pour Kubernetes).
- Désactivation du swap (Kubernetes ne fonctionne pas avec le swap activé).
- Activation du module `br_netfilter` pour permettre la communication réseau entre les pods.

3. configuration de Kubernetes

3.1 configuration du fichier `deployment.yaml`

Le worker node doit avoir la configuration suivante au niveau du fichier de déploiement :



```
# Please edit the object below. Lines beginning with a '#' will be ignored,
# and an empty file will abort the edit. If an error occurs while saving this file will be
# reopened with the relevant failures.
#
apiVersion: apps/v1
kind: Deployment
metadata:
  annotations:
    deployment.kubernetes.io/revision: "2"
    kubectl.kubernetes.io/last-applied-configuration: |
      {"apiVersion":"apps/v1","kind":"Deployment","metadata":{"annotations":{},"name":"age-calculator","namespace":"default"},"spec":{"replicas":2,"selector":{"matchLabels":{"app":"age-calculator"}},"template":{"metadata":{"labels":{"app":"age-calculator"}},"spec":{"containers":[{"image":"devopsproject774/age-calculator:late
st","imagePullPolicy":"Always","name":"age-calculator","ports":[{"containerPort":3000}]}]}}}}
  creationTimestamp: "2025-02-28T19:26:58Z"
  generation: 2
  name: age-calculator
  namespace: default
  resourceVersion: "115197"
  uid: ac79de36-155e-4657-9837-32042933a9ef
spec:
  progressDeadlineSeconds: 600
  replicas: 2
  revisionHistoryLimit: 10
  selector:
    matchLabels:
      app: age-calculator
  strategy:
    rollingUpdate:
      maxSurge: 25%
      maxUnavailable: 25%
    type: RollingUpdate
  template:
    metadata:
```

3.2 configuration du fichier `age-calculator-service.yaml`

Sur la machine worker, configurer les éléments suivants dans le fichier de service :

```
azureuser@k8s-worker: ~  
Please edit the object below. Lines beginning with a '#' will be ignored,  
# and an empty file will abort the edit. If an error occurs while saving this file will be  
# reopened with the relevant failures.  
#  
apiVersion: v1  
kind: Service  
metadata:  
  annotations:  
    kubernetes.io/last-applied-configuration: |  
      {"apiVersion":"v1","kind":"Service","metadata":{"annotations":{},"name":"age-calculator-service","namespace":"default"},"spec":{"ports":[{"port":3000,"protocol":"TCP","targetPort":80}], "selector":{"app":"age-calculator"},"type":"LoadBalancer"}}  
  creationTimestamp: "2025-02-28T19:26:58Z"  
  name: age-calculator-service  
  namespace: default  
  resourceVersion: "115102"  
  uid: 28927a5f-868e-44ae-a60c-2de5318dd949  
spec:  
  allocateLoadBalancerNodePorts: true  
  clusterIP: 10.111.135.161  
  clusterIPs:  
    - 10.111.135.161  
  externalTrafficPolicy: Cluster  
  internalTrafficPolicy: Cluster  
  ipFamilies:  
    - IPv4  
  ipFamilyPolicy: SingleStack  
  ports:  
    - nodePort: 30989  
      port: 3000  
      protocol: TCP  
      targetPort: 80  
  selector:  
    app: age-calculator  
"/tmp/kubectrl-edit-337502958.yaml" 38L, 1181C 1,1 Top
```

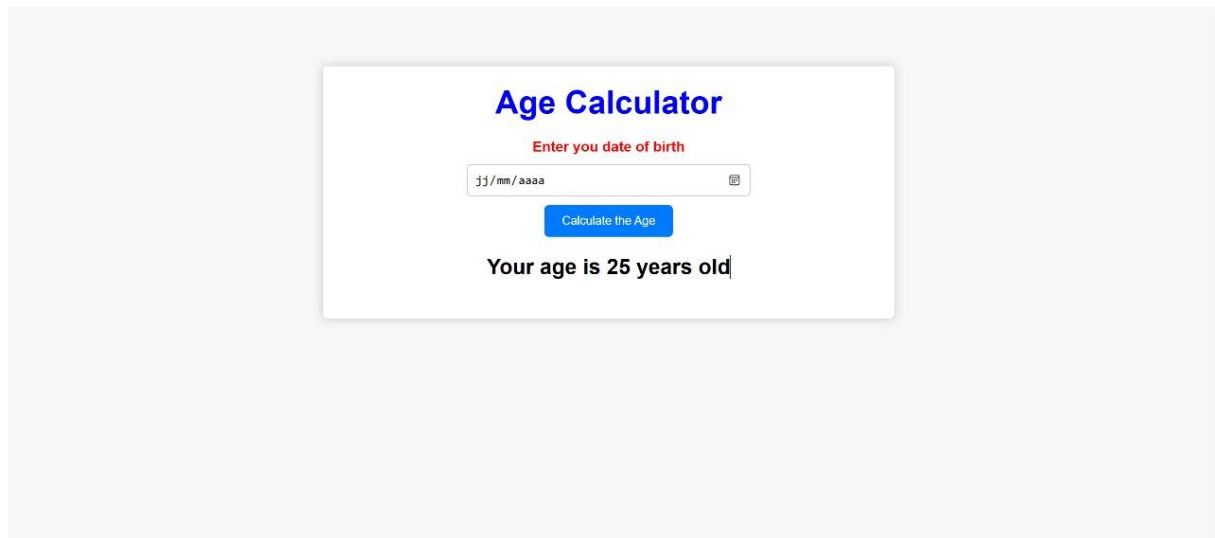
3.3 Vérification du cluster

Vérifiez que les nœuds et le pods sont bien connectés avec la commande :

```
Last login: Fri Feb 28 16:57:34 2025 from 196.217.181.49  
azureuser@k8s-worker: ~$ kubectl get nodes  
NAME           STATUS    ROLES           AGE   VERSION  
k8s-worker     Ready     control-plane   23h   v1.28.15  
azureuser@k8s-worker: ~$ kubectl get pods  
NAME                                                    READY   STATUS    RESTARTS   AGE  
age-calculator-5dd65b7975-h2dn8                        1/1     Running   0           27m  
age-calculator-5dd65b7975-ksc99                       1/1     Running   0           27m  
azureuser@k8s-worker: ~$
```

Si tout est configuré correctement, vous devriez voir le Worker avec le statut Ready.

Vue de l'application apres le déploiement :



The screenshot shows a web application titled "Age Calculator" in a bold blue font. Below the title, there is a red prompt "Enter you date of birth". A text input field contains the placeholder "jj/mm/aaaa" and a calendar icon on the right. Below the input field is a blue button labeled "Calculate the Age". At the bottom, the result is displayed as "Your age is 25 years old". The entire form is centered on a light gray background.

CONCLUSION GENERALE

La mise en place de notre infrastructure DevOps a permis d'automatiser l'intégration et le déploiement continu d'une application conteneurisée en exploitant les outils modernes tels que Kubernetes, Jenkins, Docker, et Terraform. Grâce à Azure, nous avons pu provisionner une infrastructure scalable et flexible tout en respectant les contraintes des environnements cloud gratuits.

L'installation et la configuration d'un cluster Kubernetes ont été des étapes clés, assurant l'orchestration et la gestion efficace des conteneurs sur notre infrastructure cloud. L'utilisation d'Ansible et de Terraform nous a permis d'automatiser l'ensemble du processus de déploiement, garantissant ainsi une gestion efficace des ressources et une reproductibilité accrue du projet.

Ce projet nous a permis d'explorer en profondeur les bonnes pratiques DevOps et de comprendre l'importance de l'automatisation et du monitoring dans un environnement cloud. À travers cette implémentation, nous avons pu constater que l'approche DevOps favorise une meilleure collaboration entre les équipes de développement et d'exploitation, tout en réduisant les délais de mise en production.

En perspective, des améliorations peuvent être apportées, notamment :

Renforcer la sécurité en intégrant des solutions comme Vault pour la gestion des secrets.

Optimiser l'infrastructure avec l'utilisation de Kubernetes autoscaler et des Load Balancers.

Ajouter du monitoring avancé avec Prometheus et Grafana pour suivre les performances du cluster.

En conclusion, ce projet constitue une première étape vers la mise en place d'une infrastructure DevOps robuste et nous ouvre de nombreuses possibilités d'amélioration