

Guide d'utilisation: Projet MLOps - Déploiement automatisé d'une solution ML

Groupe :

- Ikram BENABBAS
- Manel CHENNA
- Yousra MBARKI

1. Infrastructure :

1.1 Création de l'infrastructure avec Terraform

Pour commencer, on a utilisé Terraform pour déployer une machine virtuelle (VM) sur **Google Cloud Platform (GCP)**. Voici les étapes principales que nous avons suivies

Étape 1 : Installation de Docker

Pour pouvoir utiliser des conteneurs dans notre projet, on a installé Docker sur la VM. Voici comment on s'y est pris :

1. Mise à jour des paquets : `sudo apt update`
2. Installation de Docker et Docker Compose : `sudo apt install -y docker.io docker-compose`
3. Vérification des installations :
 - a. Version de Docker : `docker --version`
 - b. Version de Docker Compose : `docker-compose --version`

Docker est maintenant prêt à l'emploi sur la VM.

Installe Docker et Docker Compose :

```
bash
```

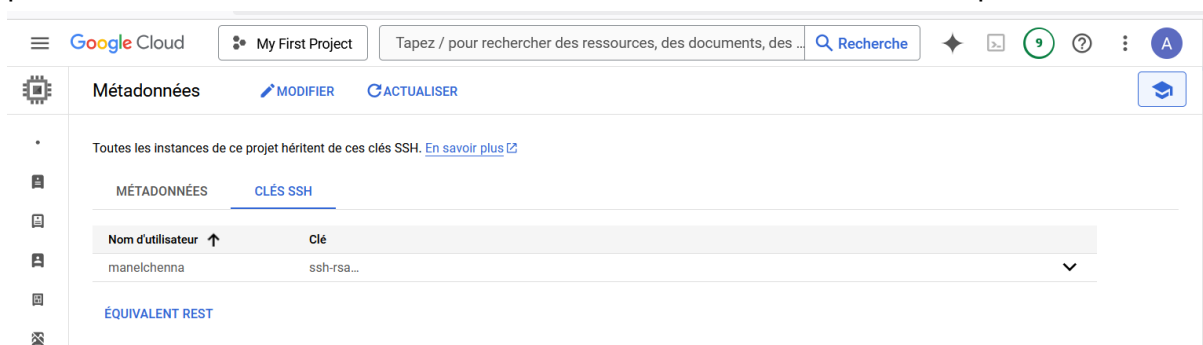
```
sudo apt update
```

```
sudo apt install -y docker.io docker-compose
```

```
manelchenna@ubuntu-vm: ~  
Setting up containerd (1.7.12-0ubuntu2~20.04.1) ...  
Created symlink /etc/systemd/system/multi-user.target.wants/containerd.service → /lib/systemd/system/containerd.service  
Setting up python3-websocket (0.53.0-2ubuntu1) ...  
update-alternatives: using /usr/bin/python3-wsdump to provide /usr/bin/wsdump (wsdump) in auto mode  
Setting up python3-dockerpty (0.4.1-2) ...  
Setting up python3-docker (4.1.0-1) ...  
Setting up docker.io (24.0.7-0ubuntu2~20.04.1) ...  
Adding group `docker' (GID 121) ...  
Done.  
Created symlink /etc/systemd/system/multi-user.target.wants/docker.service → /lib/systemd/system/docker.service.  
Created symlink /etc/systemd/system/sockets.target.wants/docker.socket → /lib/systemd/system/docker.socket.  
Setting up dnsmasq-base (2.90-0ubuntu0.20.04.1) ...  
Setting up ubuntu-fan (0.12.13ubuntu0.1) ...  
Created symlink /etc/systemd/system/multi-user.target.wants/ubuntu-fan.service → /lib/systemd/system/ubuntu-fan.service  
Setting up docker-compose (1.25.0-1) ...  
Processing triggers for systemd (245.4-4ubuntu3.24) ...  
Processing triggers for man-db (2.9.1-1) ...  
Processing triggers for dbus (1.12.16-2ubuntu2.3) ...  
Processing triggers for libc-bin (2.31-0ubuntu9.16) ...  
manelchenna@ubuntu-vm:~$
```

Étape 2 : Création d'une clé SSH

On a créé notre clé publique SSH dans les métadonnées du projet sur GCP. Cela permet une connexion sécurisée sans avoir besoin de saisir un mot de passe.



Architecture du Code : le code est séparé en section **infrastructure**, **application ML**, **pipeline CI/CD**, et **monitoring**.

Dossier/Fichier	Description
-----------------	-------------

.github/workflows/ci-cd.yml	Pipeline CI/CD pour automatiser les tests, la construction des images Docker et le déploiement.
ansible/	Contient les fichiers pour l'automatisation de la configuration des serveurs avec Ansible.
└─ inventory.ini	Inventaire Ansible définissant les hôtes à configurer.
└─ playbook.yml	Playbook Ansible pour installer Docker et configurer le serveur.
docker/	Fichiers liés à Docker pour le déploiement des conteneurs.
└─ docker-compose.yml	Fichier Docker Compose pour lancer plusieurs conteneurs en même temps.
└─ Dockerfile	Instructions pour construire l'image Docker de l'application ML.
ml_app/	Dossier contenant le code source de l'application ML.
└─ api/	Contient l'API Flask pour effectuer des prédictions.
│ └─ main.py	Code de l'API avec des routes pour prédire à partir d'un modèle entraîné.
│ └─ requirements.txt	Dépendances Python nécessaires pour l'API (Flask, joblib, prometheus-flask-exporter).
└─ test_predict.py	Script pour tester les prédictions de l'API.
mlflow/	Dossier pour le suivi des expérimentations avec MLflow.
model/	Contient le modèle ML entraîné et les scripts d'entraînement.
└─ model.pkl	Modèle ML sauvegardé avec joblib.
└─ train.py	Script d'entraînement du modèle ML.
monitoring/	Configuration pour le monitoring avec Prometheus et Grafana.
└─ prometheus/prometheus.yml	Configuration de Prometheus pour la collecte des métriques.
└─ grafana/dashboard/	Fichiers de configuration pour les dashboards Grafana.
terraform/	Infrastructure as Code pour déployer une VM sur GCP avec Terraform.
└─ main.tf	Configuration principale pour créer une VM.
└─ providers.tf	Configuration du provider GCP pour Terraform.
└─ variables.tf	Variables utilisées dans les configurations Terraform.
secrets/	Contient les fichiers ou informations pour la gestion des secrets (comme les clés GCP).
ui/	Interface utilisateur pour interagir avec l'API.
└─ src/app.py	Interface Streamlit permettant d'envoyer des requêtes à l'API.

tests/load_test/	Contient des tests de charge pour l'API (non inclus ici mais recommandé).
README.md	Documentation complète expliquant chaque étape du projet.

Étape 3 : Installation de Terraform

Terraform est l'outil qu'on a utilisé pour automatiser la création de notre infrastructure. Voici les étapes suivies pour l'installer sur la VM :

1. Téléchargement de Terraform : `wget`
https://releases.hashicorp.com/terraform/1.5.0/terraform_1.5.0_linux_amd64.zip
2. Décompression et déplacement dans un répertoire système : `unzip`
`terraform_1.5.0_linux_amd64.zip`
`sudo mv terraform /usr/local/bin/`
3. Vérification de l'installation : `terraform -v`

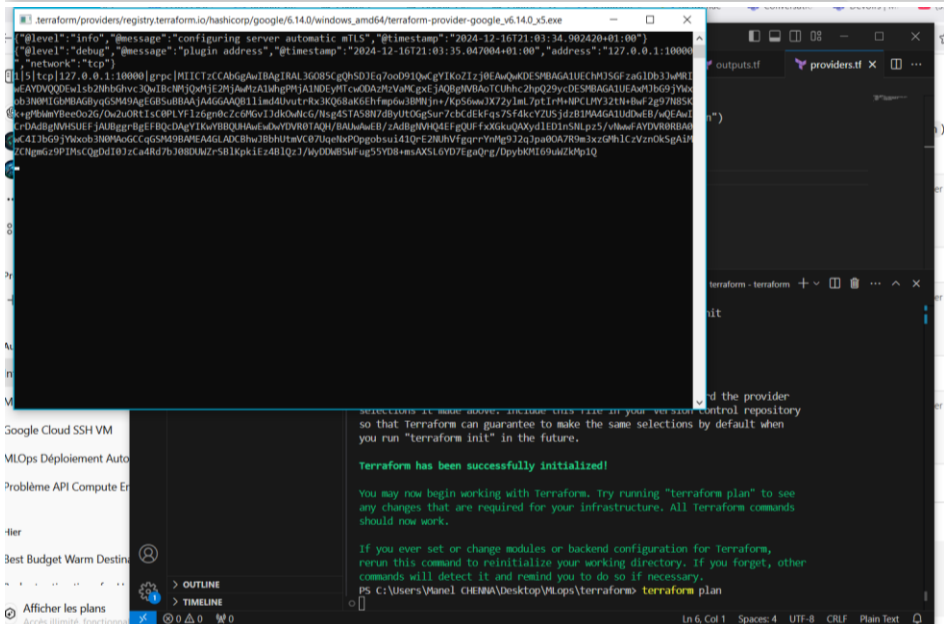
```
manelchenna@ubuntu-vm:~$ wget https://releases.hashicorp.com/terraform/1.5.0/terraform_1.5.0_linux_amd64.zip
--2024-12-16 18:22:46-- https://releases.hashicorp.com/terraform/1.5.0/terraform_1.5.0_linux_amd64.zip
Resolving releases.hashicorp.com (releases.hashicorp.com)... 18.245.199.100, 18.245.199.59, 18.245.199.105, ...
Connecting to releases.hashicorp.com (releases.hashicorp.com)|18.245.199.100|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 20957558 (20M) [application/zip]
Saving to: 'terraform_1.5.0_linux_amd64.zip'

terraform_1.5.0_linux_amd64.z 100%[=====] 19.99M ---KB/s in 0.09s

2024-12-16 18:22:46 (224 MB/s) - 'terraform_1.5.0_linux_amd64.zip' saved [20957558/20957558]

manelchenna@ubuntu-vm:~$ unzip terraform_1.5.0_linux_amd64.zip
Archive: terraform_1.5.0_linux_amd64.zip
  inflating: terraform
manelchenna@ubuntu-vm:~$ sudo mv terraform /usr/local/bin/
manelchenna@ubuntu-vm:~$ terraform -v
Terraform v1.5.0
on linux_amd64

Your version of Terraform is out of date! The latest version
is 1.10.2. You can update by downloading from https://www.terraform.io/downloads.html
manelchenna@ubuntu-vm:~$
```



Étape 4 : Création d'une VM avec Terraform

On a utilisé Terraform pour créer une VM. On a d'abord écrit deux fichiers principaux pour définir notre infrastructure.

1. **Fichier main.tf** : Ce fichier contient les configurations de la VM, comme le nom, le type de machine, et l'image système.

```
resource "google_compute_instance" "ubuntu_vm" {
  name          = "ubuntu-vm2"
  machine_type  = "e2-medium"
  zone          = "europe-west9-b"
  boot_disk {
    initialize_params {
      image = "ubuntu-os-cloud/ubuntu-2204-lts"
    }
  }
}
```

```
}  
}
```

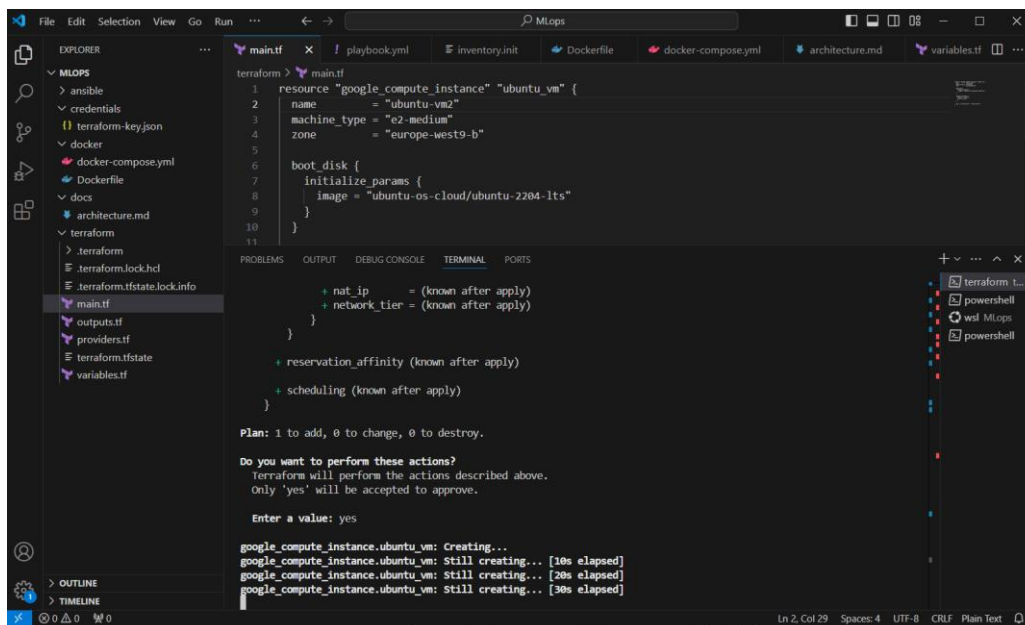
2. **Fichier providers.tf** : Ce fichier configure le fournisseur Google Cloud pour Terraform.

```
provider "google" {  
  project = "votre-projet"  
  region  = "europe-west9"  
}
```

Commandes Terraform :

Voici les commandes qu'on a utilisées pour déployer la VM :

1. **Initialisation** : terraform init
2. **Planification** : terraform plan
3. **Application** : terraform apply
On a confirmé avec yes pour finaliser la création de la VM.



The screenshot shows the VS Code editor with a Terraform configuration file named `main.tf`. The configuration defines a `google_compute_instance` resource named `ubuntu_vm` with the following properties:

```
resource "google_compute_instance" "ubuntu_vm" {
  name         = "ubuntu-vm2"
  machine_type = "e2-medium"
  zone         = "europe-west9-b"

  boot_disk {
    initialize_params {
      image = "ubuntu-os-cloud/ubuntu-2204-lts"
    }
  }
}
```

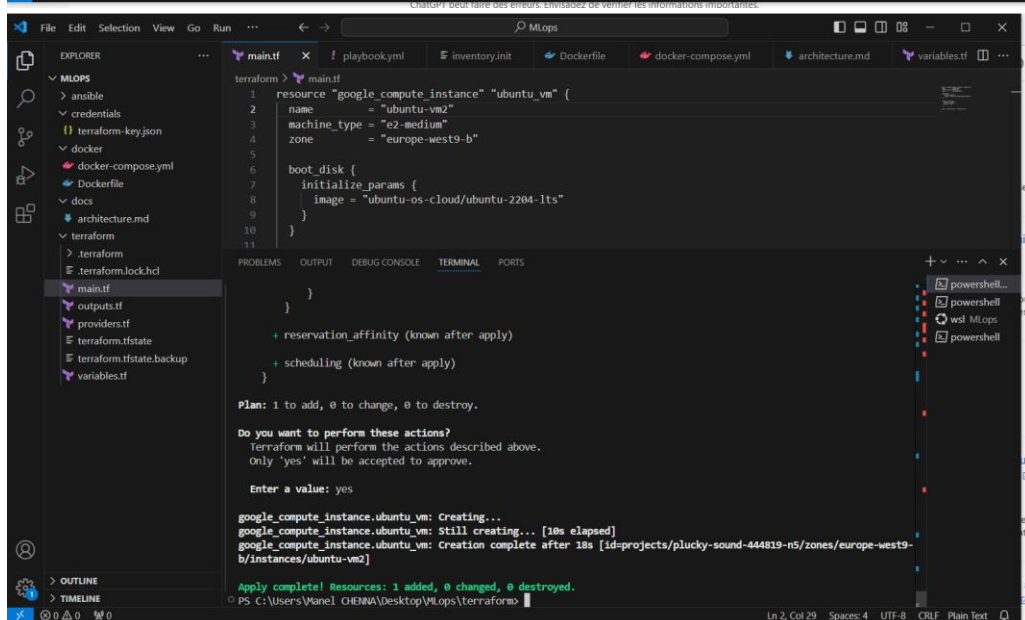
The terminal output shows the Terraform plan and the creation of the instance:

```
Plan: 1 to add, 0 to change, 0 to destroy.

Do you want to perform these actions?
Terraform will perform the actions described above.
Only 'yes' will be accepted to approve.

Enter a value: yes

google_compute_instance.ubuntu_vm: Creating...
google_compute_instance.ubuntu_vm: Still creating... [10s elapsed]
google_compute_instance.ubuntu_vm: Still creating... [20s elapsed]
google_compute_instance.ubuntu_vm: Still creating... [30s elapsed]
```



The screenshot shows the VS Code editor with the same Terraform configuration file `main.tf`. The terminal output shows the completion of the instance creation:

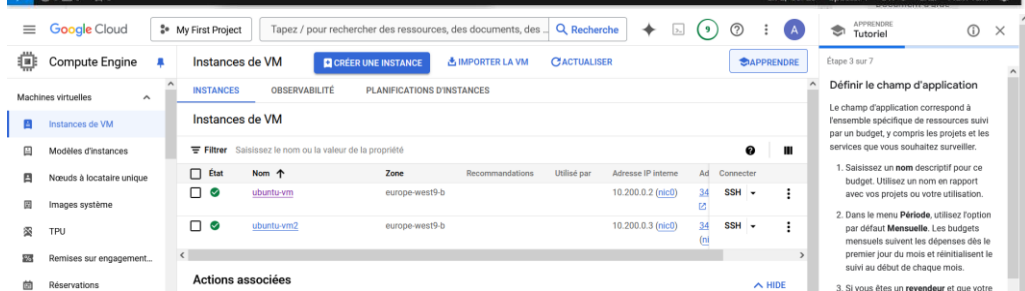
```
Plan: 1 to add, 0 to change, 0 to destroy.

Do you want to perform these actions?
Terraform will perform the actions described above.
Only 'yes' will be accepted to approve.

Enter a value: yes

google_compute_instance.ubuntu_vm: Creating...
google_compute_instance.ubuntu_vm: Still creating... [10s elapsed]
google_compute_instance.ubuntu_vm: Creation complete after 18s [id=projects/plucky-sound-444819-n5/zones/europe-west9-b/instances/ubuntu-vm2]

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```



1.2 Configuration des serveurs avec Ansible

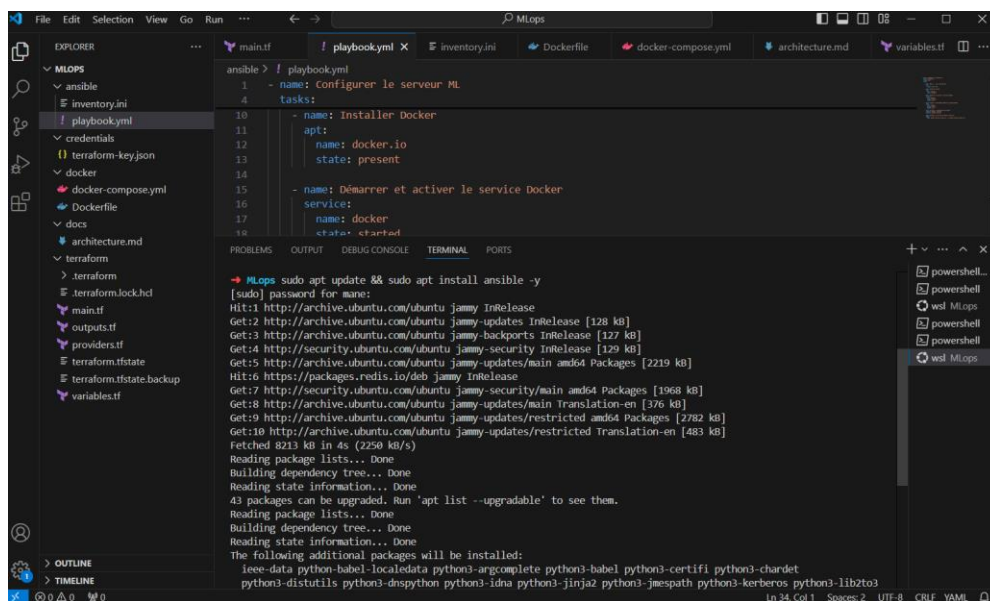
Une fois la VM déployée, on a utilisé Ansible pour configurer les outils nécessaires sur le serveur. Cela nous a permis d'installer **Docker** et **Terraform** de manière automatisée.

Étape 1 : Installation d'Ansible

On a commencé par installer Ansible sur notre machine locale pour gérer la configuration des serveurs. Voici la commande utilisée pour l'installation :

```
sudo apt update && sudo apt install ansible -y
```

Cette étape nous permet d'utiliser Ansible pour exécuter des tâches automatisées sur la VM.



Étape 2 : Tester la connexion à la VM via la clé SSH

Une fois Ansible installé, on a testé la connexion SSH à la VM pour vérifier que tout fonctionnait correctement. Voici la commande utilisée :

```
ssh -i "C:\Users\Mane1\CHENNA\Desktop\GCP\id_rsa_gcp"
ubuntu@34.155.38.80
```

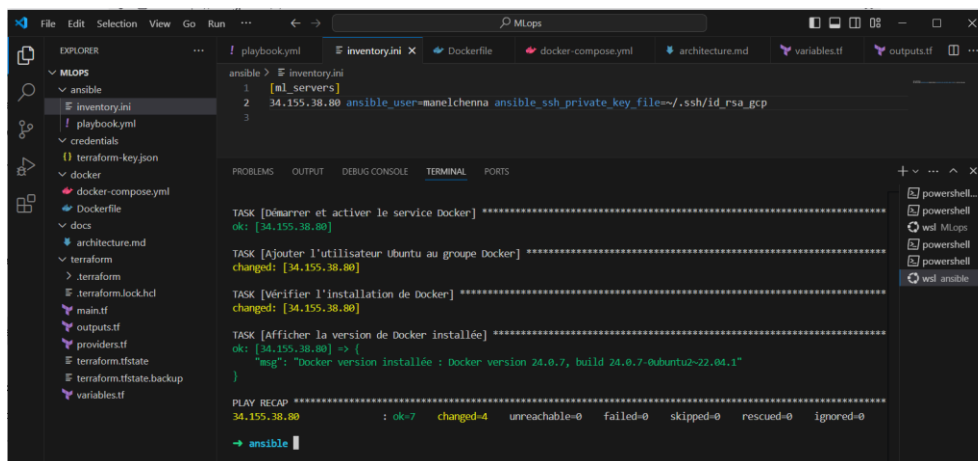
Cette étape était cruciale pour s'assurer qu'Ansible pourrait communiquer avec la VM.


```
service:
  name: docker
  state: started
```

- name: Ajouter l'utilisateur Ubuntu au groupe Docker
command: usermod -aG docker ubuntu
- name: Vérifier l'installation de Docker
command: docker --version

Pour exécuter ce playbook, on a utilisé la commande suivante :

```
ansible-playbook -i inventory.ini playbook.yml
```



1.3 Architecture Containerisée avec Docker

Dans cette section, on a confirmé que Docker était correctement installé et prêt à exécuter des conteneurs. Cela constitue une étape clé pour mettre en place une architecture containerisée.

Étape 1 : Test de Connexion Docker

Pour vérifier le bon fonctionnement de Docker, on a exécuté un conteneur de test en utilisant l'image officielle `hello-world`. Voici la commande utilisée :

```
bash
Copier le code
docker run hello-world
```

- Cette commande a téléchargé l'image hello-world depuis le registre Docker Hub et a exécuté un conteneur.
- Le conteneur a affiché un message confirmant que Docker fonctionne correctement, notamment que :
 - Le client Docker communique avec le daemon.
 - L'image a été téléchargée et exécutée avec succès.

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
manelchenna@ubuntu-vm2:~$ docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
c1ec31eb5944: Pull complete
Digest: sha256:5b3cc85e16e3058003c13b7821318369dad01dac3d0bb877aac3c28182255c724
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
1. The Docker client contacted the Docker daemon.
2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
   (amd64)
3. The Docker daemon created a new container from that image which runs the
   executable that produces the output you are currently reading.
4. The Docker daemon streamed that output to the Docker client, which sent it
   to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
$ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
https://hub.docker.com/

For more examples and ideas, visit:
https://docs.docker.com/get-started/

manelchenna@ubuntu-vm2:~$

```

```

manelchenna@ubuntu-vm2:~$ sudo systemctl status docker
● docker.service - Docker Application Container Engine
   Loaded: loaded (/lib/systemd/system/docker.service; enabled; vendor preset: enabled)
   Active: active (running) since Tue 2024-12-17 12:47:54 UTC; 28min ago
     TriggeredBy: ● docker.socket
       Docs: https://docs.docker.com
    Main PID: 6320 (dockerd)
      Tasks: 10
     Memory: 29.4M
        CPU: 1.280s
    CGroup: /system.slice/docker.service
            └─6320 /usr/bin/dockerd -H fd:// --containerd=/run/containerd/containerd.sock

Dec 17 12:47:53 ubuntu-vm2 systemd[1]: Starting Docker Application Container Engine...
Dec 17 12:47:53 ubuntu-vm2 dockerd[6320]: time="2024-12-17T12:47:53.430826328Z" level=info msg="Starting up"
Dec 17 12:47:53 ubuntu-vm2 dockerd[6320]: time="2024-12-17T12:47:53.439083206Z" level=info msg="detected 127.0.0.53 n
Dec 17 12:47:53 ubuntu-vm2 dockerd[6320]: time="2024-12-17T12:47:53.626081387Z" level=info msg="Loading containers: s
Dec 17 12:47:54 ubuntu-vm2 dockerd[6320]: time="2024-12-17T12:47:54.230439728Z" level=info msg="Loading containers: c
Dec 17 12:47:54 ubuntu-vm2 dockerd[6320]: time="2024-12-17T12:47:54.262678142Z" level=info msg="Docker daemon" commit
Dec 17 12:47:54 ubuntu-vm2 dockerd[6320]: time="2024-12-17T12:47:54.262890139Z" level=info msg="Daemon has completed
Dec 17 12:47:54 ubuntu-vm2 dockerd[6320]: time="2024-12-17T12:47:54.358245690Z" level=info msg="API listen on /run/do
Dec 17 12:47:54 ubuntu-vm2 systemd[1]: Started Docker Application Container Engine.
Dec 17 12:52:19 ubuntu-vm2 dockerd[6320]: time="2024-12-17T12:52:19.105461045Z" level=info msg="ignoring event" conta
lines 1-22/22 (END)

```

1.4 Documentation des Choix d'Architecture

Choix Techniques

1. Cloud Provider : Google Cloud Platform (GCP)

- On a choisi GCP pour ses performances, sa fiabilité, et sa facilité d'intégration avec Terraform.

- b. Les ressources sont facilement configurables via l'interface graphique et Terraform, ce qui simplifie le déploiement.

2. Infrastructure as Code (IaC) avec Terraform

- a. L'utilisation de Terraform garantit une gestion versionnée et répliquable de notre infrastructure.
- b. Les configurations sont codées dans des fichiers lisibles (`main.tf`, `providers.tf`), permettant une mise à jour et un suivi centralisé.

3. Configuration des Serveurs avec Ansible

- a. Ansible a été choisi pour automatiser la configuration des serveurs une fois la VM déployée. Cela inclut l'installation des dépendances comme Docker et Terraform.
- b. Sa syntaxe en YAML est simple et facilement intégrable à un pipeline CI/CD.

4. Docker pour Conteneurisation

- a. Docker nous permet d'encapsuler les applications et leurs dépendances dans des conteneurs. Cela garantit la portabilité entre environnements.

2. Application ML :

Voici une explication détaillée et claire pour le modèle ML, en suivant les étapes mentionnées avec une explication de chaque partie.

Étape 1 : Entraîner le Modèle

Le modèle a été entraîné à l'aide de Python et Scikit-Learn. Voici les étapes suivies :

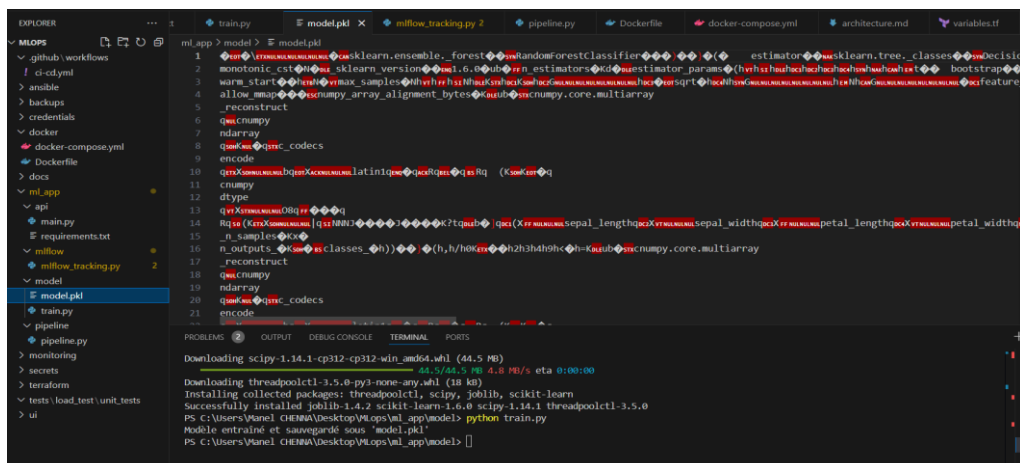
1. **Préparation des données** : Utilisation du dataset Iris.
2. **Entraînement** : Modèle RandomForestClassifier.
3. **Sauvegarde** : Le modèle est enregistré sous la forme d'un fichier `model.pkl`.

Commandes principales :

```
python train.py
```

```
PROBLEMS 2 OUTPUT DEBUG CONSOLE TERMINAL PORTS

Downloading scipy-1.14.1-cp312-cp312-win_amd64.whl (44.5 MB)
44.5/44.5 MB 4.8 MB/s eta 0:00:00
Downloading threadpoolctl-3.5.0-py3-none-any.whl (18 kB)
Installing collected packages: threadpoolctl, scipy, joblib, scikit-learn
Successfully installed joblib-1.4.2 scikit-learn-1.6.0 scipy-1.14.1 threadpoolctl-3.5.0
PS C:\Users\Manel CHENNA\Desktop\MLops\ml_app\model> python train.py
Modèle entraîné et sauvegardé sous 'model.pkl'
PS C:\Users\Manel CHENNA\Desktop\MLops\ml_app\model> []
```



Étape 2 : Lancer l'API avec Docker

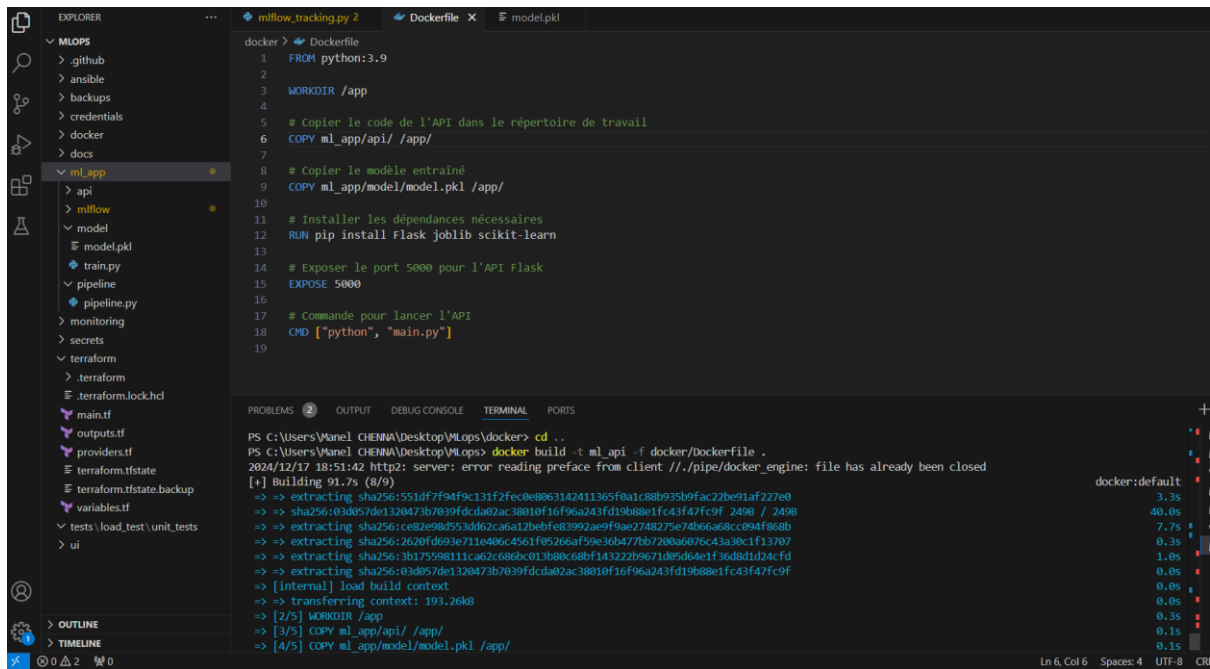
Pour rendre le modèle accessible, on a créé une API Flask. Ensuite, on l'a conteneurisée avec Docker grâce au Dockerfile.

Contenu du Dockerfile :

```
FROM python:3.9
WORKDIR /app
COPY ml_app/api/ /app/
COPY ml_app/model/model.pkl /app/
RUN pip install Flask joblib scikit-learn
EXPOSE 5000
CMD ["python", "main.py"]
```

Commande pour construire et exécuter l'image Docker :

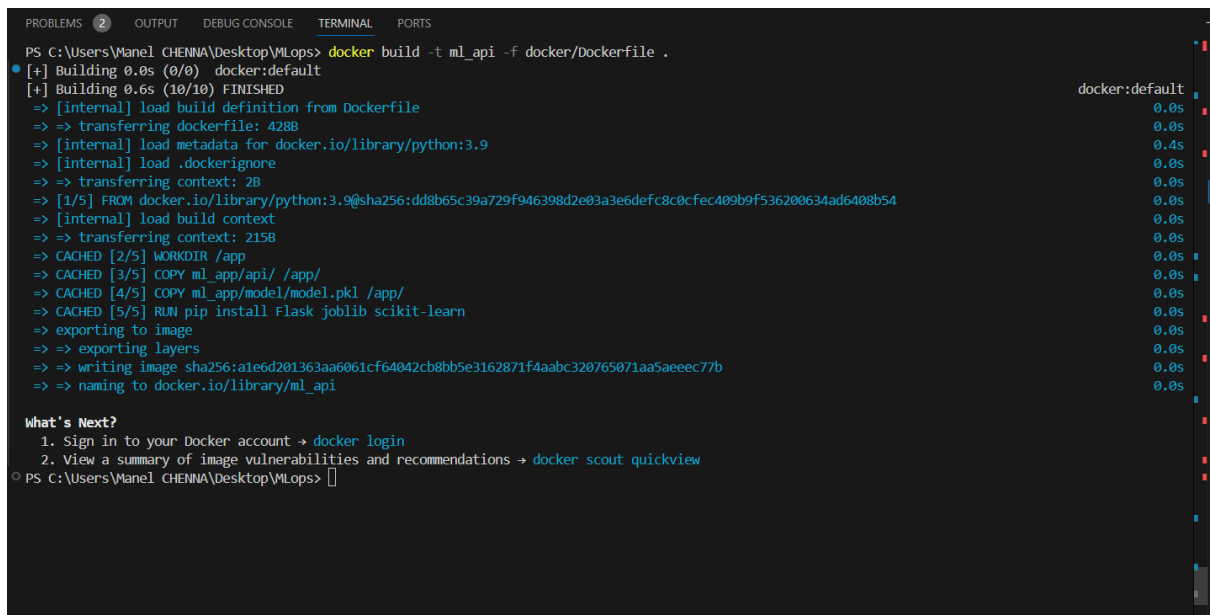
1. **Construire l'image Docker :** `docker build -t ml_api -f docker/Dockerfile .`
2. **Exécuter le conteneur :** `docker run -p 5000:5000 ml_api`



```
EXPLORER
  MLops
    .github
    ansible
    backups
    credentials
    docker
    docs
    ml_app
      api
      mlflow
      model
        model.pkl
        train.py
      pipeline
      pipeline.py
      monitoring
      secrets
      terraform
      .terraform
      .terraform.lock.hcl
      main.tf
      outputs.tf
      providers.tf
      terraform.tfstate
      terraform.tfstate.backup
      variables.tf
      tests
        load_test
        unit_tests
      ui

Dockerfile
1 FROM python:3.9
2
3 WORKDIR /app
4
5 # Copier le code de l'API dans le répertoire de travail
6 COPY ml_app/api/ /app/
7
8 # Copier le modèle entraîné
9 COPY ml_app/model/model.pkl /app/
10
11 # Installer les dépendances nécessaires
12 RUN pip install Flask joblib scikit-learn
13
14 # Exposer le port 5000 pour l'API Flask
15 EXPOSE 5000
16
17 # Commande pour lancer l'API
18 CMD ["python", "main.py"]
19

TERMINAL
PS C:\Users\Manel CHENNA\Desktop\MLops> cd ..
PS C:\Users\Manel CHENNA\Desktop\MLops> docker build -t ml_api -f docker/Dockerfile .
2024/12/17 18:51:42 http2: server: error reading preface from client //./pipe/docker_engine: file has already been closed
[+] Building 91.7s (8/9)
=> => extracting sha256:551df7f94f9c131f2fec0e8063142411365f0a1c88b935b9fac22be91af227e0 3.3s
=> => sha256:03d057de1320473b7039fdca02ac38010f16f96a243fd19b88e1fc43f47c9f 2498 / 2498 40.0s
=> => extracting sha256:c82e98d553dd62ca6a12bebf83992ae9f9ae2748275e74b66a68cc094f868b 7.7s
=> => sha256:2620fd693e711e486c461f65266af59e36b477bb7200a607ec43a30c1f13707 0.3s
=> => extracting sha256:3b175598111ca62c686bc013b80c68bf143222b9671d05d6e1f36d8d1d24cfd 1.0s
=> => extracting sha256:03d057de1320473b7039fdca02ac38010f16f96a243fd19b88e1fc43f47c9f 0.0s
=> [internal] load build context 0.0s
=> => transferring context: 193.2kB 0.0s
=> [2/5] WORKDIR /app 0.3s
=> [3/5] COPY ml_app/api/ /app/ 0.1s
=> [4/5] COPY ml_app/model/model.pkl /app/ 0.1s
```



```
PS C:\Users\Manel CHENNA\Desktop\MLops> docker build -t ml_api -f docker/Dockerfile .
[+] Building 0.0s (0/0) docker:default
[+] Building 0.6s (10/10) FINISHED
=> [internal] load build definition from Dockerfile 0.0s
=> => transferring dockerfile: 428B 0.0s
=> [internal] load metadata for docker.io/library/python:3.9 0.4s
=> [internal] load .dockerignore 0.0s
=> => transferring context: 2B 0.0s
=> [1/5] FROM docker.io/library/python:3.9@sha256:dd8b65c39a729f946398d2e0a3a3e6defc8c0cfc409b9f536200634ad6408b54 0.0s
=> [internal] load build context 0.0s
=> => transferring context: 215B 0.0s
=> CACHED [2/5] WORKDIR /app 0.0s
=> CACHED [3/5] COPY ml_app/api/ /app/ 0.0s
=> CACHED [4/5] COPY ml_app/model/model.pkl /app/ 0.0s
=> CACHED [5/5] RUN pip install Flask joblib scikit-learn 0.0s
=> exporting to image 0.0s
=> => exporting layers 0.0s
=> => writing image sha256:a1e6d201363aa6061cf64042cb8bb5e3162871f4aabc320765071aa5aeec77b 0.0s
=> => naming to docker.io/library/ml_api 0.0s

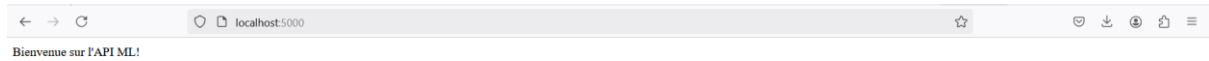
What's Next?
1. Sign in to your Docker account → docker login
2. View a summary of image vulnerabilities and recommendations → docker scout quickview
PS C:\Users\Manel CHENNA\Desktop\MLops>
```

Étape 3 : Tester le Port Localhost 5000

Une fois l'API déployée, on a testé son accessibilité sur <http://localhost:5000>.

Résultat attendu :

Un message indiquant que l'API est en ligne, comme : "Bienvenue sur l'API ML !".



Étape 4 : Tester une Méthode POST sur l'API

On a envoyé des données au point de terminaison /predict pour tester les prédictions.

Code utilisé pour tester la méthode POST :

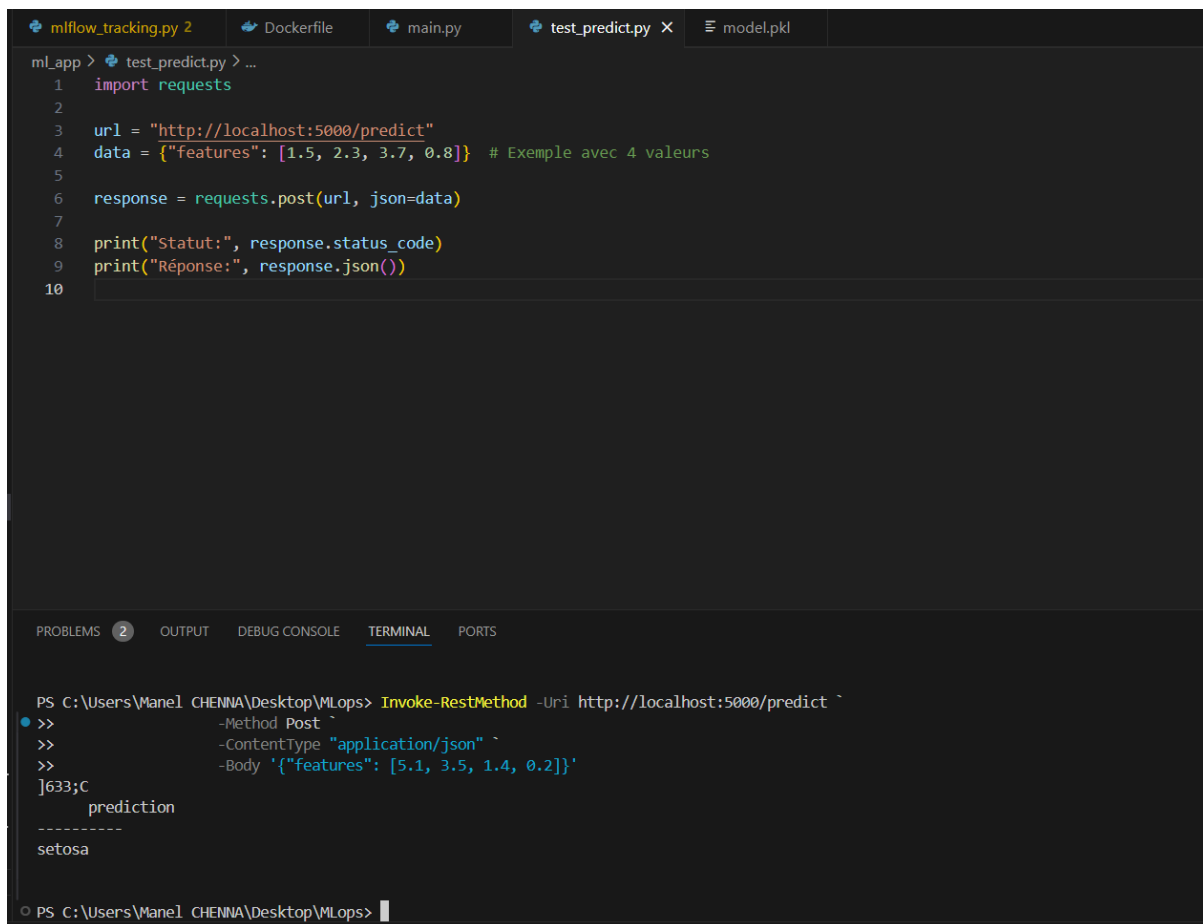
```
import requests

url = "http://localhost:5000/predict"
data = {"features": [5.1, 3.5, 1.4, 0.2]} # Exemple avec des
valeurs de test
response = requests.post(url, json=data)

print("Statut :", response.status_code)
print("Réponse :", response.json())
```

Résultat attendu :

Le modèle retourne une classe de prédiction, par exemple : setosa.



The image shows a VS Code editor with a Python file named `test_predict.py` open. The code is as follows:

```
1 import requests
2
3 url = "http://localhost:5000/predict"
4 data = {"features": [1.5, 2.3, 3.7, 0.8]} # Exemple avec 4 valeurs
5
6 response = requests.post(url, json=data)
7
8 print("Statut:", response.status_code)
9 print("Réponse:", response.json())
10
```

The terminal at the bottom shows the command `Invoke-WebRequest -Uri http://localhost:5000/predict -Method Post -ContentType "application/json" -Body '{"features": [5.1, 3.5, 1.4, 0.2]}'` being executed, which returns a JSON response: `{ "prediction": "setosa" }`.

Étape 5 : Interface Utilisateur pour l'API (Bonus)

On a développé une interface utilisateur simple avec Streamlit, permettant à l'utilisateur de saisir des caractéristiques et d'obtenir des prédictions.

Commande pour lancer l'interface :

```
streamlit run app.py
```




Étape 6 : Configuration et Suivi avec MLflow

Pour assurer le suivi des expériences, on a utilisé MLflow pour :

1. Enregistrer les métriques (précision, hyperparamètres, etc.).
2. Visualiser et comparer les résultats dans une interface web.

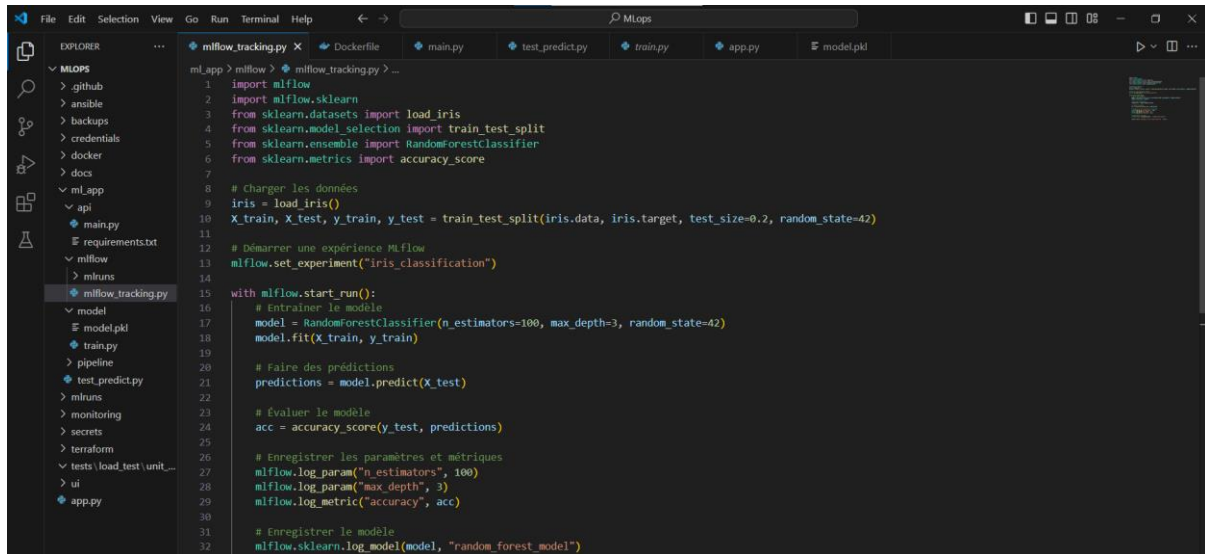
Commandes principales :

1. **Exécuter MLflow pour suivre les expérimentations :** `python mlflow_tracking.py`
2. **Lancer le serveur MLflow pour visualisation :** `mlflow ui`

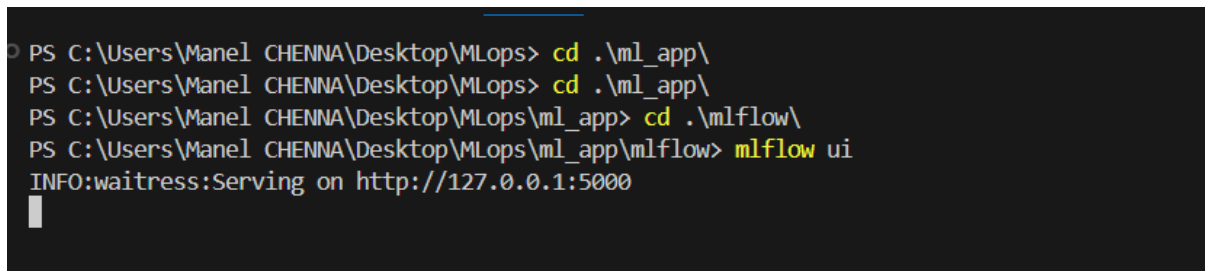
Résultat attendu :

Une interface web accessible à <http://localhost:5000> avec les détails des expérimentations.

Capture d'écran

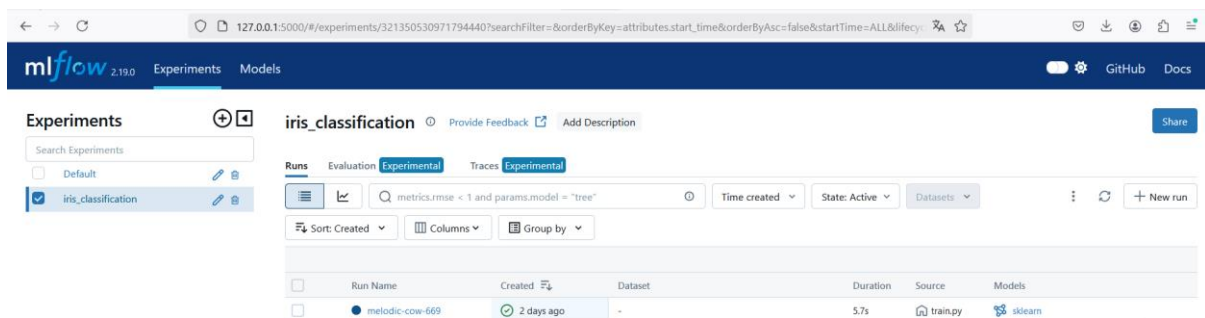


```
1 import mlflow
2 import mlflow.sklearn
3 from sklearn.datasets import load_iris
4 from sklearn.model_selection import train_test_split
5 from sklearn.ensemble import RandomForestClassifier
6 from sklearn.metrics import accuracy_score
7
8 # Charger les données
9 iris = load_iris()
10 X_train, X_test, y_train, y_test = train_test_split(iris.data, iris.target, test_size=0.2, random_state=42)
11
12 # Démarrer une expérience MLflow
13 mlflow.set_experiment("Iris_classification")
14
15 with mlflow.start_run():
16     # Entraîner le modèle
17     model = RandomForestClassifier(n_estimators=100, max_depth=3, random_state=42)
18     model.fit(X_train, y_train)
19
20     # Faire des prédictions
21     predictions = model.predict(X_test)
22
23     # Évaluer le modèle
24     acc = accuracy_score(y_test, predictions)
25
26     # Enregistrer les paramètres et métriques
27     mlflow.log_param("n_estimators", 100)
28     mlflow.log_param("max_depth", 3)
29     mlflow.log_metric("accuracy", acc)
30
31     # Enregistrer le modèle
32     mlflow.sklearn.log_model(model, "random_forest_model")
```



```
PS C:\Users\Manel CHENNA\Desktop\MLops> cd .\ml_app\
PS C:\Users\Manel CHENNA\Desktop\MLops> cd .\ml_app\
PS C:\Users\Manel CHENNA\Desktop\MLops\ml_app> cd .\mlflow\
PS C:\Users\Manel CHENNA\Desktop\MLops\ml_app\mlflow> mlflow ui
INFO: waitress: Serving on http://127.0.0.1:5000
```

Interface **MLflow** montrant une expérience nommée `iris_classification`, avec une exécution (run) appelée `melodic-cow-669`, créée il y a 2 jours, ayant duré 5,7 secondes, lancée via le script `train.py` et utilisant le modèle `sklearn`

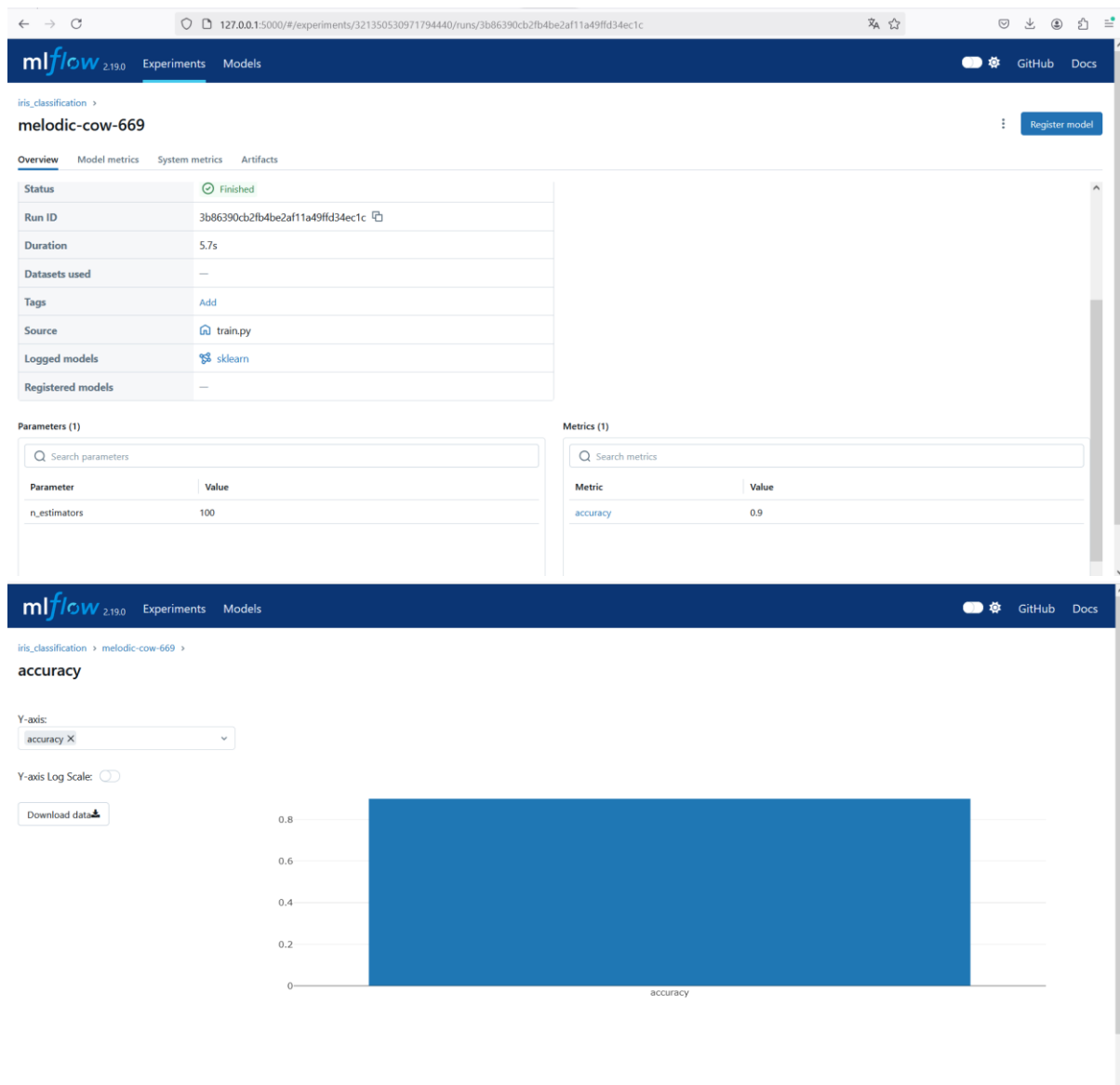


On observe dans le lien <http://127.0.0.1:5000> que notre API exécute correctement un modèle de machine learning :

- Le script **train.py** a été utilisé pour lancer le modèle.
- Le modèle enregistré provient de la bibliothèque **scikit-learn (sklearn)**.

- Le paramètre **n_estimators = 100** suggère l'utilisation d'un modèle de type **Random Forest**.
- La métrique d'évaluation indique une **accuracy de 0,9**, démontrant que le modèle a été exécuté avec succès.
- Le graphique confirme visuellement les performances obtenues par le modèle.

Ces éléments prouvent l'intégration réussie du modèle de machine learning dans



Résumé des Réalisations

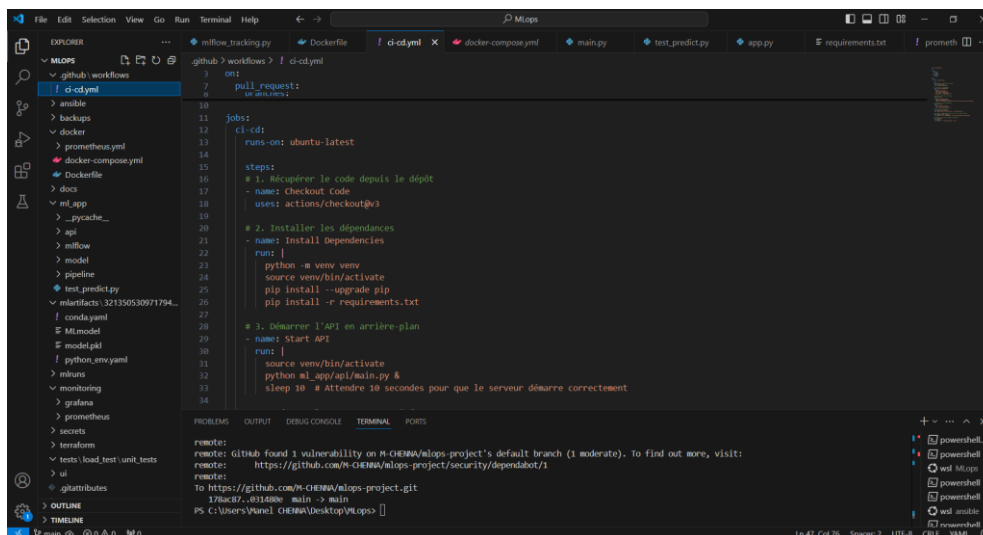
1. **Modèle entraîné et sauvegardé** : Modèle RandomForest entraîné avec succès et sauvegardé sous `model.pkl`.

2. **API Flask conteneurisée** : Déploiement d'une API avec Docker pour rendre le modèle accessible.
3. **Suivi avec MLflow** : Suivi des expérimentations et visualisation des performances.
4. **Interface utilisateur** : Interface simple avec Streamlit pour interagir avec le modèle.

3. Pipeline CI/CD :

Étape 1 : Configuration du Pipeline CI/CD

On a configuré un pipeline CI/CD à l'aide de GitHub Actions pour automatiser les processus d'intégration et de déploiement.



```
1 on:
2   pull_request:
3     branches: [main]
4
5 jobs:
6   ci-cd:
7     runs-on: ubuntu-latest
8
9     steps:
10      - name: Récupérer le code depuis le dépôt
11        uses: actions/checkout@v3
12
13      - name: Installer les dépendances
14        run: |
15          python -m venv venv
16          source venv/bin/activate
17          pip install --upgrade pip
18          pip install -r requirements.txt
19
20      - name: Démarrer l'API en arrière-plan
21        run: |
22          source venv/bin/activate
23          python ml_app/api/main.py &
24          sleep 10 # Attendre 10 secondes pour que le serveur démarre correctement
```

Étape 2 : Résultat de l'Exécution du Pipeline

On a déclenché le pipeline après un push sur la branche principale.

Résultat attendu :

1. Les tests sont exécutés automatiquement.
2. Une image Docker est construite via `docker build`.
3. L'application est déployée via `docker run`.

Résultat obtenu :

Le pipeline s'est terminé avec succès, indiquant que toutes les étapes ont été correctement exécutées.

The screenshot displays the GitHub Actions interface for a CI/CD pipeline. The top section shows the pipeline name "Ajout du Dockerfile et mise à jour du pipeline CI/CD #17" with a green checkmark indicating success. It was triggered via push 3 minutes ago and has a total duration of 1m 16s. The status is "Success".

On the left, a sidebar lists the jobs: "ci-cd" with a green checkmark. Below it, "Run details" are visible, including "Usage" and "Workflow file".

The main area shows the "ci-cd.yml" workflow file with the trigger "on: push". Below the file, a job card for "ci-cd" shows a green checkmark and a duration of 1m 6s.

Below the job card, the "Run Tests" step is expanded, showing the following log output:

```
1 ▶ Run source venv/bin/activate
5 ===== test session starts =====
6 platform linux -- Python 3.10.12, pytest-7.0.1, pluggy-1.5.0
7 rootdir: /home/runner/work/mlops-project/mlops-project
8 collected 1 item
9
10 ml_app/test_predict.py . [100%]
11
12 ===== 1 passed in 0.06s =====
```

4. Monitoring :

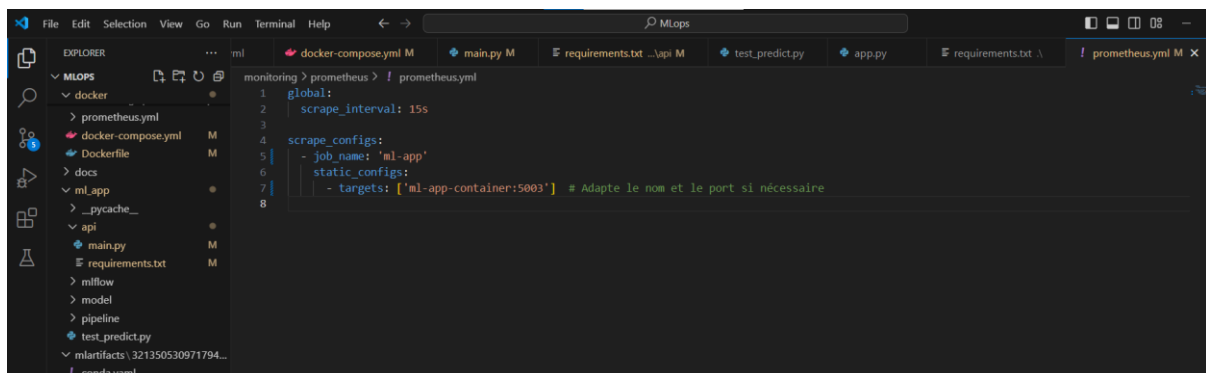
Étape 1 : Configuration de Prometheus

Pour permettre la collecte des métriques depuis notre application Flask, on a **configuré** le fichier `prometheus.yml` :

```
global:
  scrape_interval: 15s
```

```
scrape_configs:
  - job_name: 'ml-app'
    static_configs:
      - targets: ['ml-app-container:5003'] # Adapte le nom et le
port si nécessaire
```

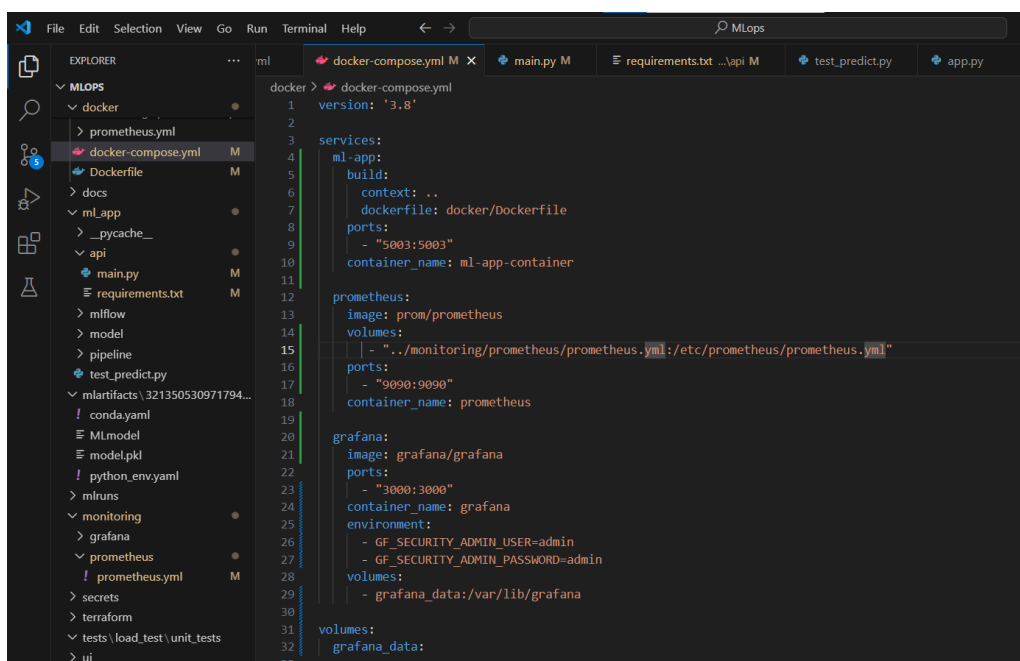
Cette configuration indique à Prometheus d'interroger l'endpoint /metrics de l'application Flask, exposée par le conteneur ml-app-container sur le port 5003.



```
monitoring > prometheus > ! prometheus.yml
1 global:
2   scrape_interval: 15s
3
4 scrape_configs:
5   - job_name: 'ml-app'
6     static_configs:
7       - targets: ['ml-app-container:5003'] # Adapte le nom et le port si nécessaire
8
```

Étape 2 : Mise à jour de docker-compose.yml

On a intégré Prometheus et Grafana dans le fichier docker-compose.yml pour orchestrer les conteneurs.



```
docker > docker-compose.yml
1 version: '3.8'
2
3 services:
4   ml-app:
5     build:
6       context: ..
7       dockerfile: docker/Dockerfile
8     ports:
9       - "5003:5003"
10    container_name: ml-app-container
11
12   prometheus:
13     image: prom/prometheus
14     volumes:
15       - ../monitoring/prometheus/prometheus.yml:/etc/prometheus/prometheus.yml
16     ports:
17       - "9090:9090"
18     container_name: prometheus
19
20   grafana:
21     image: grafana/grafana
22     ports:
23       - "3000:3000"
24     container_name: grafana
25     environment:
26       - GF_SECURITY_ADMIN_USER=admin
27       - GF_SECURITY_ADMIN_PASSWORD=admin
28     volumes:
29       - grafana_data:/var/lib/grafana
30
31 volumes:
32   grafana_data:
33
```

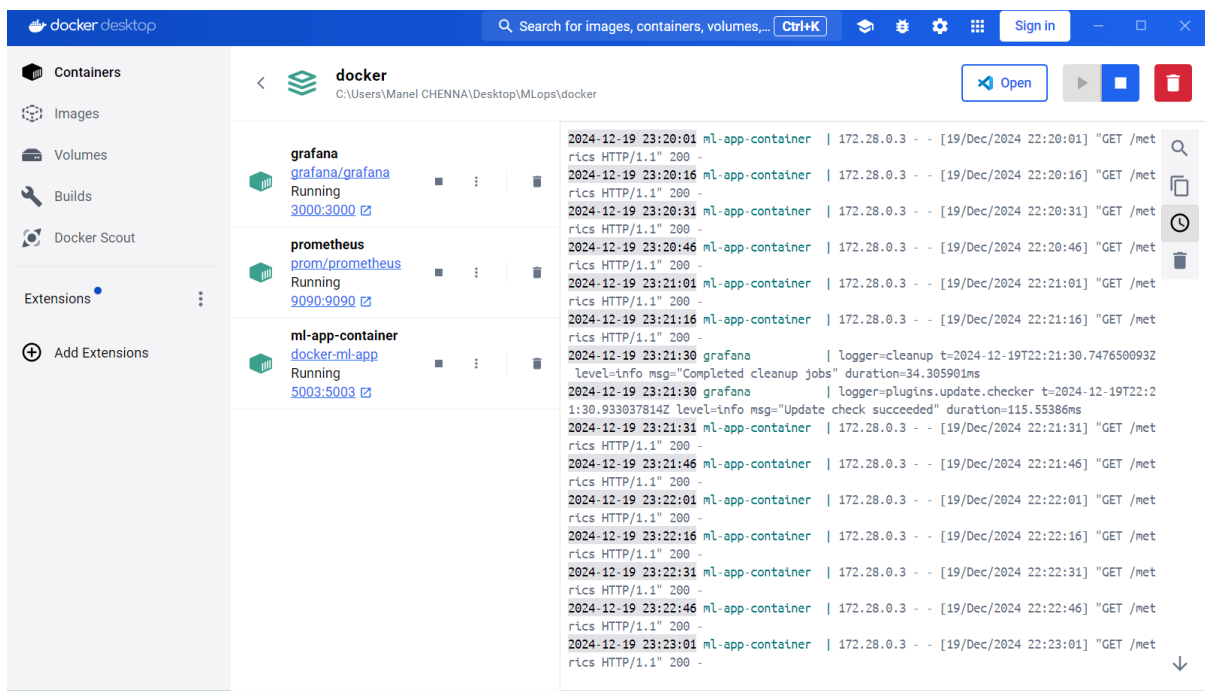
Étape 3 : Démarrage des Conteneurs

On a démarré les services en exécutant la commande suivante :

```
docker-compose up --build
```

Cette étape a permis de vérifier que tous les conteneurs étaient en cours d'exécution, y compris `ml-app-container`, `prometheus`, et `grafana`.

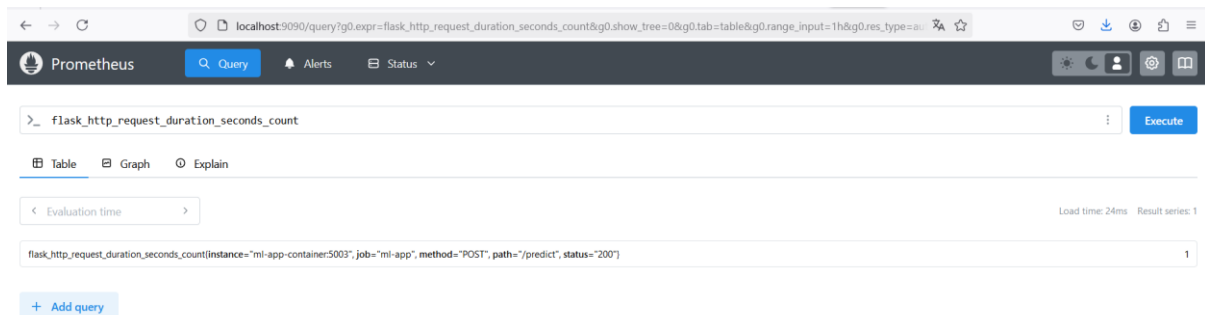
```
PS C:\Users\Manel_CHENNA\Desktop\MLops\docker> docker-compose down
time="2024-12-19T22:50:32+01:00" level=warning msg="C:\\Users\\Manel_CHENNA\\Desktop\\MLops\\docker\\docker-compose.yml: `version` is obsolete"
[+] Running 4/4
  ✓ Container ml-app-container   Removed      0.0s
  ✓ Container prometheus        Removed      0.7s
  ✓ Container grafana           Removed      0.5s
  ✓ Network docker_default      Removed      0.3s
PS C:\Users\Manel_CHENNA\Desktop\MLops\docker> docker-compose up --build
time="2024-12-19T22:50:42+01:00" level=warning msg="C:\\Users\\Manel_CHENNA\\Desktop\\MLops\\docker\\docker-compose.yml: `version` is obsolete"
[+] Building 0.0s (0/0)  docker:default
[+] Building 46.1s (10/10) FINISHED
=> [ml-app internal] load build definition from Dockerfile
=> => transferring dockerfile: 454B
=> [ml-app internal] load metadata for docker.io/library/python:3.9
=> [ml-app internal] load .dockerignore
=> => transferring context: 2B
=> [ml-app 1/5] FROM docker.io/library/python:3.9@sha256:dd8b65c39a729f946398d2e03a3e6defc8c0cfec409b9f536200634ad6408b54
=> [ml-app internal] load build context
=> => transferring context: 217B
=> CACHED [ml-app 2/5] WORKDIR /app
=> CACHED [ml-app 3/5] COPY ml_app/api/ /app/
=> CACHED [ml-app 4/5] COPY ml_app/model/model.pkl /app/
=> [ml-app 5/5] RUN pip install Flask joblib scikit-learn prometheus-flask-exporter
=> [ml-app] exporting to image
=> => exporting layers
=> => writing image sha256:e56a5524713b2dc99f2c323b94c36b791a33e7a9d39a08bc43aed6a2a348dd9
=> => naming to docker.io/library/docker-ml-app
[+] Running 4/4
  ✓ Network docker_default      Created      0.0s
  ✓ Container prometheus        Created      0.1s
  ✓ Container ml-app-container   Created      0.1s
  ✓ Container grafana           Created      0.1s
Attaching to grafana, ml-app-container, prometheus
```



Étape 4 : Test de Connexion des Services

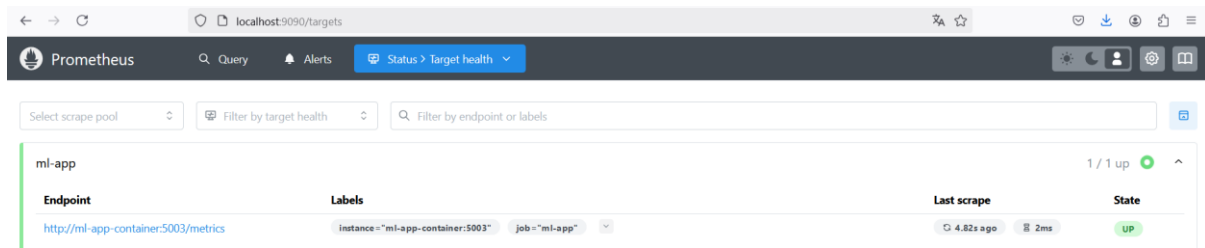
1. Test Prometheus :

- On a vérifié que l'endpoint `/metrics` de l'API Flask était accessible en ouvrant : <http://localhost:5003/metrics>.
- Prometheus collecte les métriques de cet endpoint toutes les 15 secondes.



2. Statut Prometheus :

- On a confirmé que Prometheus détectait bien la cible `ml-app-container` et que son statut était UP.



Étape 5 : Génération de Métriques

Pour enrichir les métriques, on a effectué plusieurs requêtes POST sur `/predict` :

```
Invoke-RestMethod -Uri "http://localhost:5003/predict" `
-Method POST `
-Headers @{ "Content-Type" = "application/json" } `
-Body '{"features": [5.1, 3.5, 1.4, 0.2]}'
```

Cela a généré des métriques visibles dans Prometheus, telles que :

- **flask_http_request_duration_seconds_sum** : Somme totale des durées des requêtes HTTP traitées.
- **flask_http_request_duration_seconds_count** : Nombre total de requêtes HTTP traitées.
- **flask_http_request_total** : Nombre total de requêtes HTTP.

Collecte des Métriques

- 1) En exécutant directement des commande Post depuis le terminal

```

Windows PowerShell

(venv) PS C:\Users\Manel CHENNA> Invoke-RestMethod -Uri "http://localhost:5003/predict" `
>> -Method POST `
>> -Headers @{ "Content-Type" = "application/json" } `
>> -Body '{"features": [7, 7, 7, 0]}'

prediction
-----
setosa

(venv) PS C:\Users\Manel CHENNA> Invoke-RestMethod -Uri "http://localhost:5003/predict" `
>> -Method POST `
>> -Headers @{ "Content-Type" = "application/json" } `
>> -Body '{"features": [7, 7, 7, 8]}'

prediction
-----
virginica

(venv) PS C:\Users\Manel CHENNA> Invoke-RestMethod -Uri "http://localhost:5003/predict" `
>> -Method POST `
>> -Headers @{ "Content-Type" = "application/json" } `
>> -Body '{"features": [7, 7, 7, 7]}'

prediction
-----
virginica

(venv) PS C:\Users\Manel CHENNA> Invoke-RestMethod -Uri "http://localhost:5003/predict" `
>> -Method POST `
>> -Headers @{ "Content-Type" = "application/json" } `
>> -Body '{"features": [7, 7, 7, 8]}'

prediction
-----
virginica

(venv) PS C:\Users\Manel CHENNA>

```

- 2) En utilisant l'interface utilisateur de l'API pour exécuter des commande POST

localhost:8501

Deploy

Prédiction avec l'API ML

Entrez les caractéristiques (ex: 5.1, 3.5, 1.4, 0.2)

3,2,7,7

Prédire

Prédiction : virginica

3) Accéder au lien : localhost :9090

Exécuter des requêtes Prometheus pour visualiser les métriques :

Prometheus

Query Alerts Status

Load time: 1119ms Result series: 1

flask_http_request_duration_seconds_count(instance="ml-app-container:5003", job="ml-app", method="POST", path="/predict", status="200")

7

>_ flask_http_request_duration_seconds_sum

Execute

Table Graph Explain

Load time: 9ms Result series: 1

flask_http_request_duration_seconds_sum(instance="ml-app-container:5003", job="ml-app", method="POST", path="/predict", status="200")

0.14311741199344397

>_ flask_http_request_duration_seconds_count

Execute

Table Graph Explain

Load time: 12ms Result series: 1

flask_http_request_duration_seconds_count(instance="ml-app-container:5003", job="ml-app", method="POST", path="/predict", status="200")

8

>_ flask_http_request_total

Execute

En accédant au lien : localhost :9090/metrics

On observe prometheus collecte et expose plusieurs métriques.

```
← → ↺ laptop-8e4b6mrh:9090/metrics

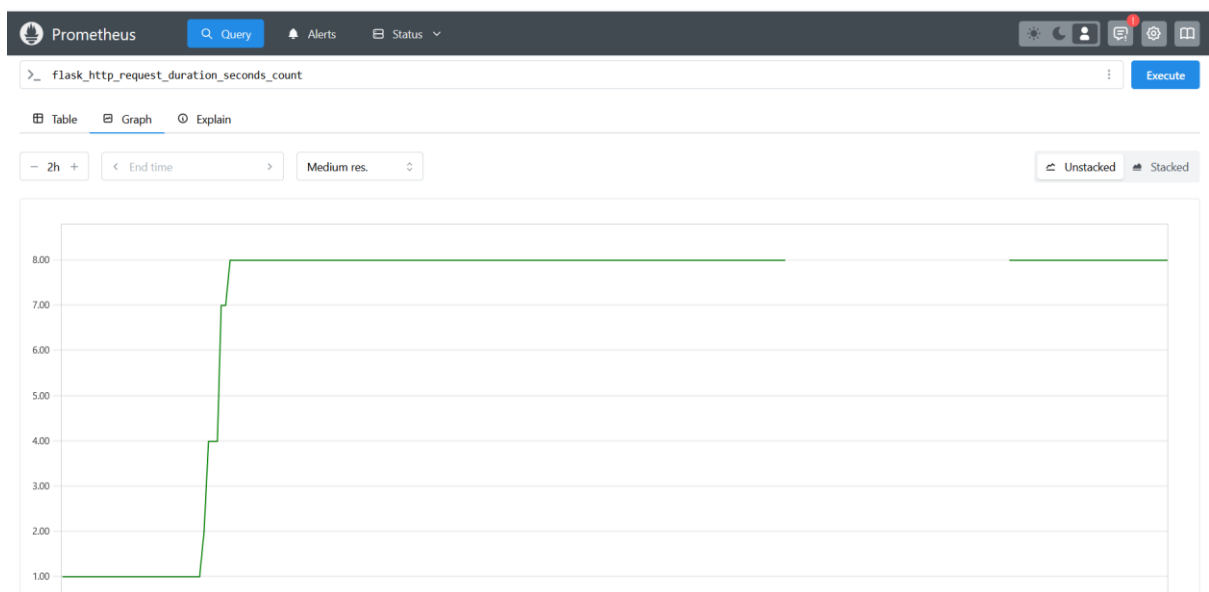
# HELP go_gc_cycles_automatic_gc_cycles_total Count of completed GC cycles generated by the Go runtime. Sourced from /gc/cycles/automatic:gc-cycles
# TYPE go_gc_cycles_automatic_gc_cycles_total counter
go_gc_cycles_automatic_gc_cycles_total 10
# HELP go_gc_cycles_forced_gc_cycles_total Count of completed GC cycles forced by the application. Sourced from /gc/cycles/forced:gc-cycles
# TYPE go_gc_cycles_forced_gc_cycles_total counter
go_gc_cycles_forced_gc_cycles_total 0
# HELP go_gc_cycles_total_gc_cycles_total Count of all completed GC cycles. Sourced from /gc/cycles/total:gc-cycles
# TYPE go_gc_cycles_total_gc_cycles_total counter
go_gc_cycles_total_gc_cycles_total 10
# HELP go_gc_duration_seconds A summary of the wall-time pause (stop-the-world) duration in garbage collection cycles.
# TYPE go_gc_duration_seconds summary
go_gc_duration_seconds{quantile="0"} 0
go_gc_duration_seconds{quantile="0.25"} 0
go_gc_duration_seconds{quantile="0.5"} 0
go_gc_duration_seconds{quantile="0.75"} 0.0007909
go_gc_duration_seconds{quantile="1"} 0.0016961
go_gc_duration_seconds_sum 0.0039924
go_gc_duration_seconds_count 10
# HELP go_gc_gogc_percent Heap size target percentage configured by the user, otherwise 100. This value is set by the GOGC environment variable, and the runtime/debug.SetGCPercent function. Sourced from /gc/gogc:percent
# TYPE go_gc_gogc_percent gauge
go_gc_gogc_percent 75
# HELP go_gc_gomemlimit_bytes Go runtime memory limit configured by the user, otherwise math.MaxInt64. This value is set by the GOMEMLIMIT environment variable, and the runtime/debug.SetMemoryLimit function. Sourced from /gc/gomemlimit:bytes
# TYPE go_gc_gomemlimit_bytes gauge
go_gc_gomemlimit_bytes 1.5306452582e+10
# HELP go_gc_heap_allocs_by_size_bytes Distribution of heap allocations by approximate size. Bucket counts increase monotonically. Note that this does not include tiny objects as defined by /gc/heap/tiny/allocs:objects, only tiny blocks. Sourced from /gc/heap/allocs-by-size:bytes
# TYPE go_gc_heap_allocs_by_size_bytes histogram
go_gc_heap_allocs_by_size_bytes_bucket{le="8.999999999999998"} 5629
go_gc_heap_allocs_by_size_bytes_bucket{le="24.999999999999996"} 41687
go_gc_heap_allocs_by_size_bytes_bucket{le="64.99999999999999"} 121650
go_gc_heap_allocs_by_size_bytes_bucket{le="144.99999999999997"} 158109
go_gc_heap_allocs_by_size_bytes_bucket{le="320.99999999999994"} 164893
go_gc_heap_allocs_by_size_bytes_bucket{le="704.9999999999999"} 166347
go_gc_heap_allocs_by_size_bytes_bucket{le="1536.9999999999998"} 167164
go_gc_heap_allocs_by_size_bytes_bucket{le="3200.9999999999995"} 167656
go_gc_heap_allocs_by_size_bytes_bucket{le="6528.999999999999"} 167980
go_gc_heap_allocs_by_size_bytes_bucket{le="13568.999999999998"} 168179
go_gc_heap_allocs_by_size_bytes_bucket{le="27264.999999999996"} 168270
go_gc_heap_allocs_by_size_bytes_bucket{le="+inf"} 168424
go_gc_heap_allocs_by_size_bytes_sum 6.0756984e+07
go_gc_heap_allocs_by_size_bytes_count 168424
# HELP go_gc_heap_allocs_bytes_total Cumulative sum of memory allocated to the heap by the application. Sourced from /gc/heap/allocs:bytes
# TYPE go_gc_heap_allocs_bytes_total counter
go_gc_heap_allocs_bytes_total 6.0756984e+07
# HELP go_gc_heap_allocs_objects_total Cumulative count of heap allocations triggered by the application. Note that this does not include tiny objects as defined by /gc/heap/tiny/allocs:objects, only tiny blocks. Sourced from /gc/heap/allocs:objects
```

Analyse de la métrique :

Le graphique de **Prometheus** représente l'évolution de la métrique **flask_http_request_duration_seconds_count** dans le temps.

Explication de la métrique :

- **flask_http_request_duration_seconds_count** :
 - C'est le nombre total de requêtes HTTP traitées par l'API Flask.
 - L'axe **Y** indique le nombre total de requêtes (count).
 - L'axe **X** représente le temps.



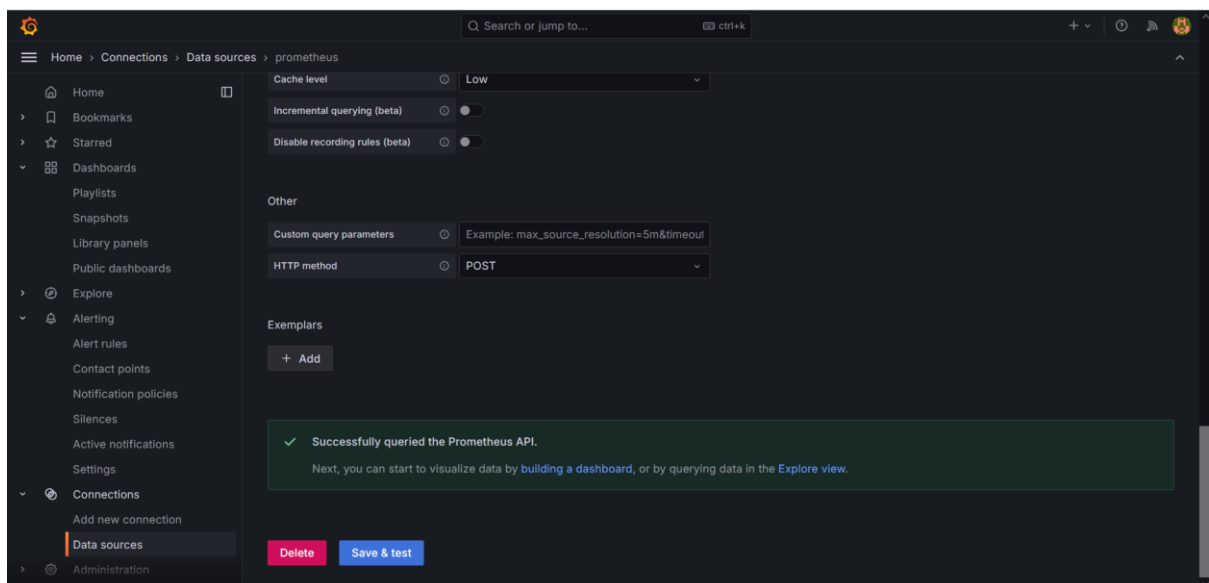
Étape 6 : Configuration de Grafana

1. Accès à Grafana :

- On s'est connecté à Grafana via <http://localhost:3000>.
- Identifiants par défaut : admin/admin.

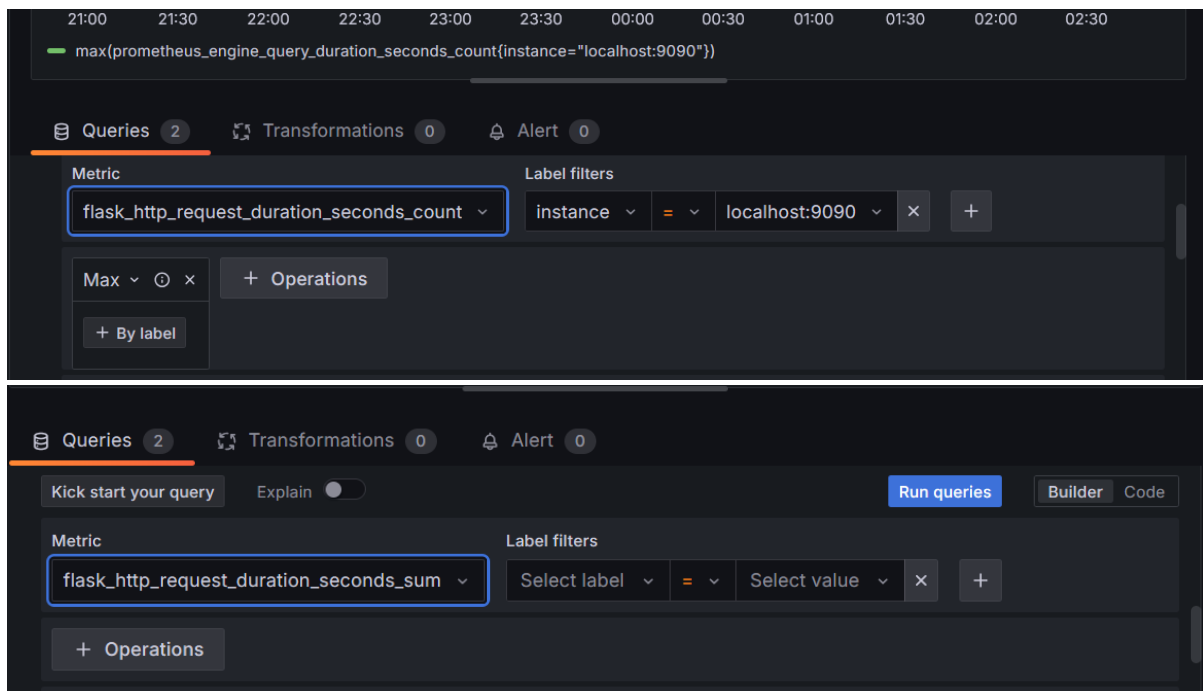
2. Lier Grafana à Prometheus :

- On se rend dans « **Data Sources** » depuis le menu des paramètres de Grafana.
- On clique sur « **Add Data Source** », puis on sélectionne « **Prometheus** ».
- On configure l'URL de Prometheus, généralement :



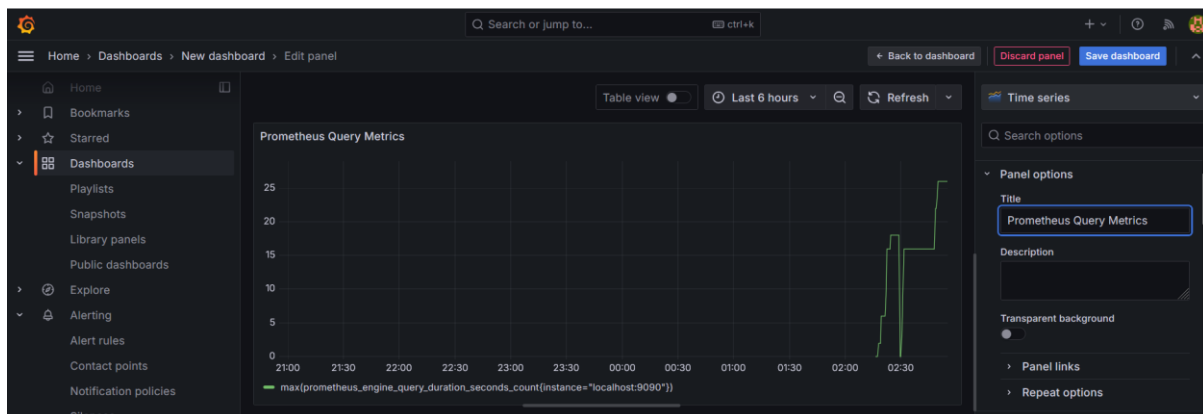
2. Création d'un Tableau de Bord :

- On a créé un tableau de bord avec des graphiques basés sur les métriques de Prometheus.
- Métriques utilisées :
 - `flask_http_request_duration_seconds_count` : Affichage du nombre de requêtes en temps réel.
 - `flask_http_request_duration_seconds_sum` : Somme des durées des requêtes.



Étape 7 : Interprétation des Résultats

- **Métriques Prometheus :**
 - Les incréments de `flask_http_request_duration_seconds_count` indiquent le nombre de requêtes POST traitées avec succès.
 - Le statut UP de `m1-app-container` dans Prometheus confirme que les métriques sont bien collectées.
- **Visualisations Grafana :**
 - Les graphiques affichent en temps réel les performances de l'API, notamment la latence des requêtes et leur volume.



Bonus:

Gestion des secrets

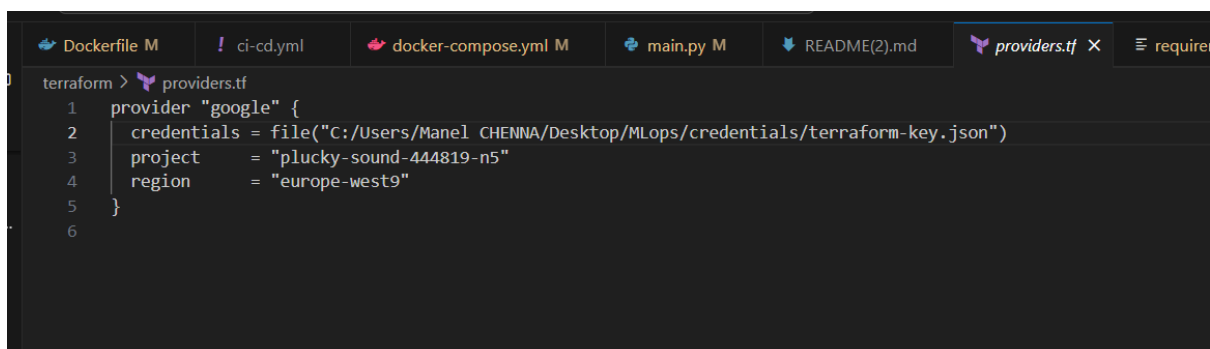
1. **Terraform :**
 - a. Utilisé pour la mise en place de l'infrastructure cloud.
 - b. Gestion des clés d'accès via des fichiers de credentials sécurisés.
2. **Ansible :**
 - a. Utilisé pour l'automatisation des configurations de serveurs.
 - b. Chiffrement des secrets avec **Ansible Vault**.
3. **Variables d'Environnement :**
 - a. Utilisées pour stocker et injecter les secrets de manière sécurisée dans le code.

3. Méthodes de Gestion des Secrets

3.1 Fichier de Credentials pour Terraform

- **Description :** Un fichier `terraform-key.json` contient les identifiants d'accès à Google Cloud Platform (GCP).
- **Utilisation :** Ce fichier est référencé dans le fichier `providers.tf` pour authentifier Terraform avec GCP.

Exemple de configuration dans `providers.tf` :



```
terraform > providers.tf
1 provider "google" {
2   credentials = file("C:/Users/Manel CHENNA/Desktop/MLops/credentials/terraform-key.json")
3   project     = "plucky-sound-444819-n5"
4   region     = "europe-west9"
5 }
6
```

Bonnes pratiques :

- Ajouter `terraform-key.json` au `.gitignore` pour éviter de le versionner dans Git

