



Université Paris cité

UFR des Sciences Fondamentales et Biomédicales

---

# Rapport du Projet de Programmation Distribuée

Parcours : Apprentissage Machine pour la Science des Données  
(AMSD)

*Préparé par :*

Mr. MAHMAHI Anis

Mr. HATTABI Ilyes

*Supervisé par :*

Mr. CHARROUX Benoit

Promotion : 2023/2024

# Table des matières

<b>1</b>	<b>Description de l'application</b>	<b>1</b>
1.1	Cas d'étude	1
1.2	Interface utilisateur	2
<b>2</b>	<b>L'architecture de l'application</b>	<b>3</b>
2.1	Architecture micro-services	3
2.2	Configuration de BDD	4
2.3	Dockerisation	5
2.4	Orchestration des conteneurs (kubernetes)	6
2.4.1	Deployments	6
2.4.2	Services	6
2.4.3	Ingress	7
2.5	Deployment dans le cloud	9
<b>3</b>	<b>Conclusion</b>	<b>11</b>

# Chapitre 1

## Description de l'application

### 1.1 Cas d'étude

La classification des cellules sanguines grâce au Deep Learning revêt une importance cruciale dans le domaine médical. Elle permet une identification et une distinction rapides et précises des différents types de cellules sanguines, ce qui améliore considérablement le processus de diagnostic.

Il existe quatre principaux types de cellules sanguines :

- Les neutrophiles.
- Les éosinophiles.
- Les lymphocytes.
- Les monocytes.

Les neutrophiles sont les globules blancs les plus abondants et jouent un rôle crucial dans la défense contre les infections bactériennes. Les éosinophiles interviennent dans les réactions allergiques et la lutte contre les parasites. Les lymphocytes, subdivisés en lymphocytes B et lymphocytes T, sont responsables de la réponse immunitaire adaptative. Les monocytes, quant à eux, se transforment en macrophages et jouent un rôle essentiel dans l'élimination des débris cellulaires et des agents pathogènes.

Cette classification présente de multiples avantages. Tout d'abord, elle joue un rôle clé dans la numération des cellules sanguines, permettant aux médecins d'estimer les quantités de globules rouges, de globules blancs et de plaquettes présentes dans un échantillon de sang. Une classification précise facilite la détection précoce de diverses affections, telles que les infections, les troubles auto-immuns et les cancers hématologiques.

### 1.2 Interface utilisateur

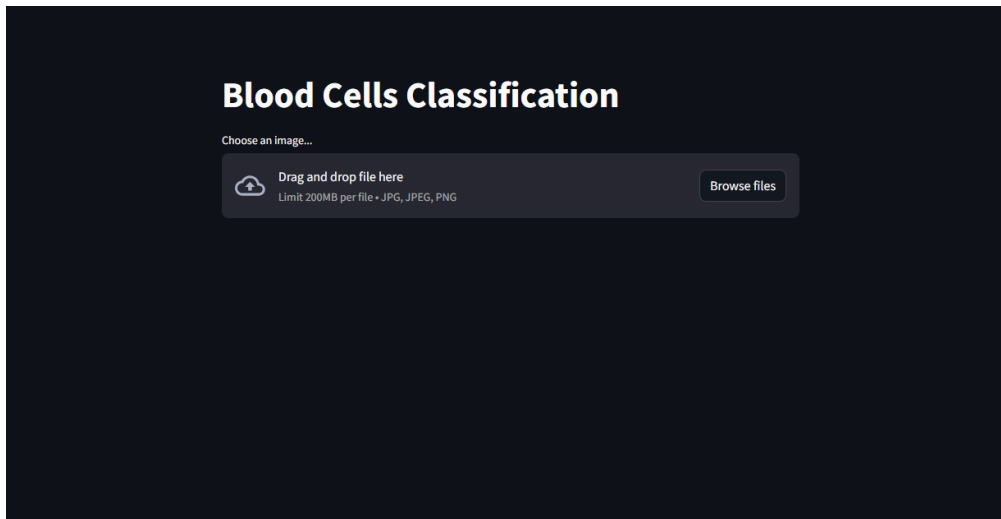


FIG. 1.1 : Téléchargement d'image de cellule sanguine

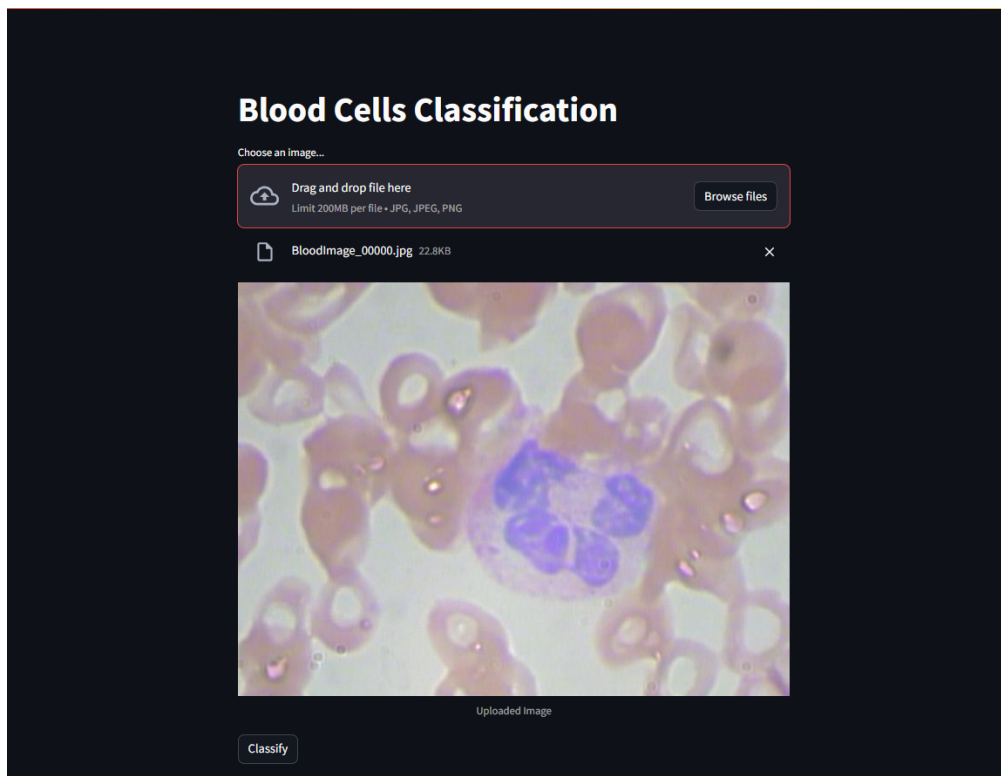


FIG. 1.2 : Affichage d'image téléchargée

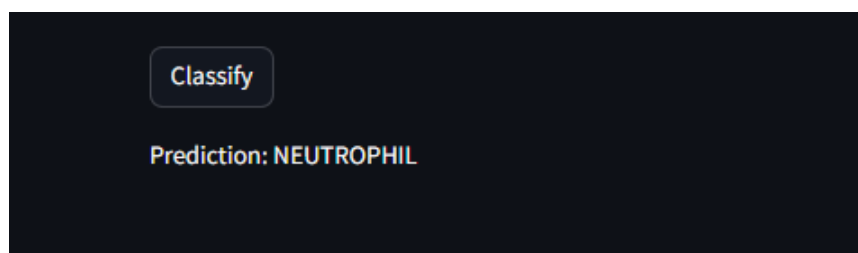


FIG. 1.3 : Résultat de la classification

# Chapitre 2

## L'architecture de l'application

### 2.1 Architecture micro-services

L'architecture de l'application est composée de trois microservices : UI, Predict et Preprocess et une passerelle (ingress).

- Le microservice **UI** est responsable de l'interaction avec l'utilisateur. Il permet de télécharger une image de cellule et de visualiser le résultat de la classification.

- Le microservice **Predict** est responsable de la classification des images. Il utilise un modèle de machine learning pour identifier le type de cellule présent dans l'image. Également il stocke les images saisies dans un bucket Amazon S3. Cela permet de les réutilisation ultérieurement par exemple pour entraîner un nouveau modèle.

- Le microservice **Preprocess** est responsable du prétraitement des images. Il effectue des opérations telles que la normalisation et redimensionnement pour la mettre sous format prévu par le modèle.

- La passerelle (ingress) est responsable du routage du trafic vers le microservice approprié.

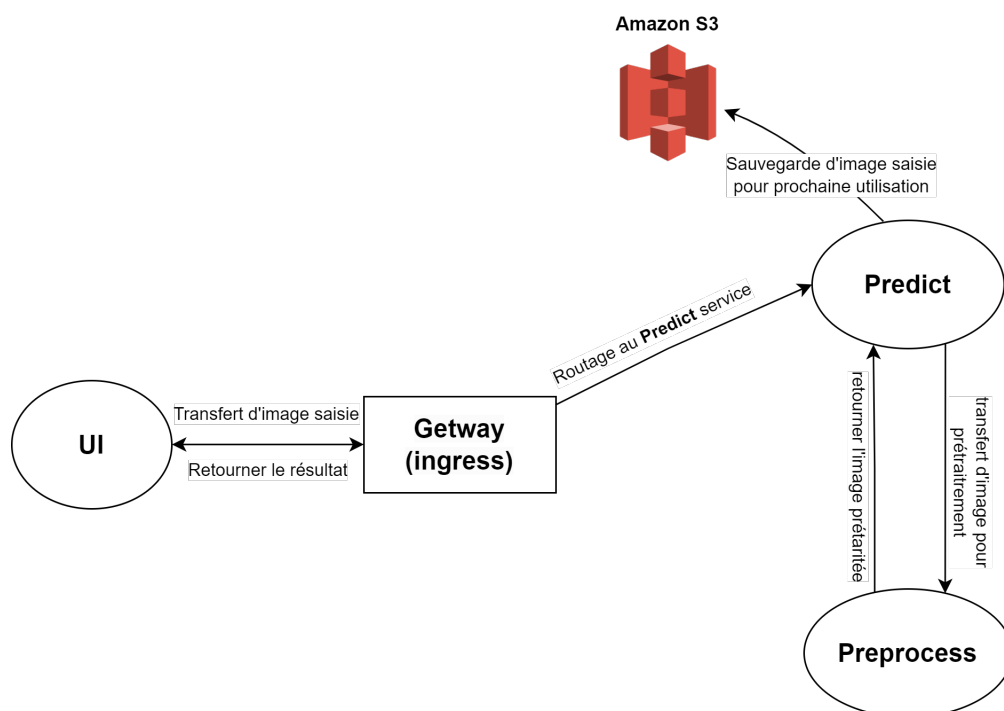


FIG. 2.1 : Architecture globale de l'application (code sur [Github](#))

## 2.2 Configuration de BDD

Pour pouvoir télécharger les fichiers dans un s3 bucket :

### 1- Créer un bucket S3

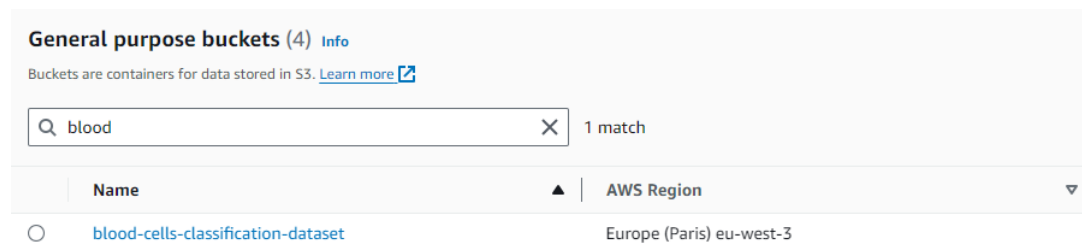


FIG. 2.2 : S3 bucket

### 2- Créer un utilisateur et générer une clé d'accès

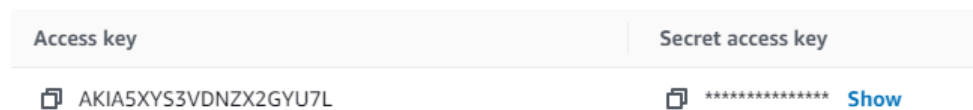


FIG. 2.3 : informations d'identification

### 3- Fournir les permissions demandées à l'utilisateur

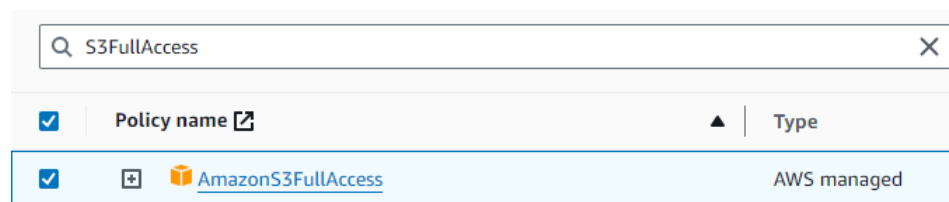


FIG. 2.4 : Permissions pour pouvoir lire/écrire

```
def upload_to_s3(local_file, bucket_name, s3_key, aws_access_key_id, aws_secret_access_key):
    s3 = boto3.client('s3', aws_access_key_id=aws_access_key_id, aws_secret_access_key=aws_secret_access_key)
    try:
        s3.upload_file(local_file, bucket_name, s3_key)
        print(f"File {local_file} uploaded to {bucket_name}/{s3_key}")
    except FileNotFoundError:
        print(f"The file {local_file} was not found")
    except NoCredentialsError:
        print("Credentials not available")
```

FIG. 2.5 : Code python pour télécharger un fichier dans s3 bucket

## 2.3 Dockerisation

Chaque microservice a été mis dans une image Docker. Les images Docker sont des fichiers compressés qui contiennent tout ce dont un microservice a besoin pour fonctionner, y compris le code source, les dépendances et les configurations. L'utilisation de Docker présente plusieurs avantages notamment : La portabilité.

Sous github chaque micro-service est principalement caractérisé par un répertoire sous la forme :

```
Service_name
├── code.py
├── Dockerfile
└── requirements.txt
```

Pour chaque service dockérisé, un port est exposé afin de lui permettre de communiquer avec le monde extérieur. Les ports exposés dans chaque service se retrouvent dans le fichier [Dockerfile](#) et sont les suivants :

Container	Port
UI	8501
Predict	5000
Preprocess	5001

TAB. 2.1 : Ports exposés

1- On crée une image docker pour le service associé :

```
$ docker build -t anismahmahi/predict_micro_service .
```

2- Push l'image dans docker-hub :

```
$ docker push anismahmahi/predict_micro_service:latest
```

Liens des images sur Docker-hub : [UI](#), [Preprocess](#), [Predict](#)

## 2.4 Orchestration des conteneurs (kubernetes)

Kubernetes permet de déployer les 3 microservices sur un cluster de machines, de les gérer et de les mettre à l'échelle pour répondre à l'augmentation de la demande.

Un fichier Kubernetes appelé **xxx-deployment.yaml** est utilisé pour définir les paramètres de déploiement des microservices. Ce fichier spécifie le nombre de pods à créer, l'image docker à exécuter, le port sur lequel chaque pod doit écouter et les ressources dont chaque pod a besoin.

Un fichier Kubernetes appelé **xxx-service.yaml** est utilisé pour définir la manière dont les pods doivent être accessibles. Ce fichier spécifie le port sur lequel le service doit écouter.

Les fichiers décrivent la configuration se trouvent sur [Github](#) regroupés tous dans un seul fichier **all.yaml**

1- On crée un cluster avec le nom amsd-cluster :

```
$ minikube start -p amsd-cluster
```

2- Configurer-le :

```
$ Kubectl apply -f all.yaml
```

### 2.4.1 Deployments

Chaque micro-service s'exécute sur un pod qui est géré par un deployment ([fichiers yaml](#)).

- Voir les deployments créées :

```
$ Kubectl get deployments
```

```
C:\Users\anis>kubectl get deployments
NAME                                READY    UP-TO-DATE    AVAILABLE    AGE
predict-microservice                1/1      1              1            30h
preprocess-microservice              1/1      1              1            30h
ui-microservice                     1/1      1              1            30h
```

```
C:\Users\anis>kubectl get pods
NAME                                READY    STATUS    RESTARTS    AGE
predict-microservice-6d67847bd9-dgnns  1/1      Running   2 (2m6s ago)  30h
preprocess-microservice-865d6487d5-grf6v  1/1      Running   2 (2m6s ago)  30h
ui-microservice-86cbdb6ccb-jqfzg        1/1      Running   2 (2m6s ago)  30h
```

### 2.4.2 Services

Il existe trois services, un pour chaque micro-service, afin de permettre aux pods de communiquer entre eux. Deux d'entre eux sont de type ClusterIP (predict-service et preprocess-service) car ils n'auront pas besoin de communiquer à l'extérieur du cluster. Le service ui-service est de type LoadBalancer car il sera exposé au public (sur l'internet) ([fichiers yaml](#)).



Les services sont exposés aux ports suivants :

Service	Port
ui-service	80
predict-service	5000
preprocess-service	5001

TAB. 2.2 : Ports des services

Comme les pods sont essentiellement des API Flask, la communication entre eux s'effectue par des requêtes POST au moyen des noms de services auxquels ils appartiennent :

**http ://NomDeService :port/EndPoint/**

- Voir les services créés :

```
$ Kubectl get services
```

```
C:\Users\anis>kubectl get services
NAME                TYPE          CLUSTER-IP      EXTERNAL-IP  PORT(S)          AGE
kubernetes           ClusterIP     10.96.0.1        <none>        443/TCP          30h
predict-service      ClusterIP     10.107.185.159   <none>        5000/TCP         30h
preprocess-service   ClusterIP     10.96.35.247     <none>        5001/TCP         30h
ui-service           LoadBalancer 10.105.35.35     <pending>     80:30458/TCP     30h
```

### 2.4.3 Ingress

un Ingress est une ressource qui gère l'accès externe aux services au sein du cluster. Il agit comme une couche d'abstraction pour gérer les règles de routage du trafic externe vers les services Kubernetes.

```
1  apiVersion: networking.k8s.io/v1
2  kind: Ingress
3  metadata:
4    name: ui-ingress
5  spec:
6    rules:
7      - host: blood-classification.com # Replace with your actual domain
8        http:
9          paths:
10           - path: /
11             pathType: Prefix
12             backend:
13               service:
14                 name: ui-service
15                 port:
16                   number: 80
```

FIG. 2.6 : Configuration de Ingress

1- Activer le NGINX Ingress controlleur :

```
$ minikube addons enable ingress
$ kubectl apply -f ingress.yml
```

2- Voir quelle adresse est associée :

```
$ kubectl get ingress
```

```
C:\Users\anis>kubectl get ingress
NAME                CLASS    HOSTS                ADDRESS      PORTS    AGE
example-ingress     nginx    myservice.info       192.168.49.2 80       21d
ui-ingress           nginx    blood-classification.com 192.168.49.2 80       17d
```

```
C:\Users\anis>kubectl get all
NAME                                READY   STATUS    RESTARTS   AGE
pod/predict-microservice-6d67847bd9-dgnns   1/1     Running   2 (10m ago)  30h
pod/preprocess-microservice-865d6487d5-grf6v  1/1     Running   2 (10m ago)  30h
pod/ui-microservice-86cbdb6ccb-jqfzg        1/1     Running   2 (10m ago)  30h

NAME                                TYPE                CLUSTER-IP      EXTERNAL-IP   PORT(S)          AGE
service/kubernetes                  ClusterIP           10.96.0.1       <none>        443/TCP          30h
service/predict-service              ClusterIP           10.107.185.159  <none>        5000/TCP          30h
service/preprocess-service            ClusterIP           10.96.35.247    <none>        5001/TCP          30h
service/ui-service                   LoadBalancer       10.105.35.35    <pending>     80:30458/TCP     30h

NAME                                READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/predict-microservice  1/1     1             1           30h
deployment.apps/preprocess-microservice  1/1     1             1           30h
deployment.apps/ui-microservice        1/1     1             1           30h

NAME                                DESIRED   CURRENT   READY   AGE
replicaset.apps/predict-microservice-6d67847bd9   1         1         1       30h
replicaset.apps/preprocess-microservice-865d6487d5 1         1         1       30h
replicaset.apps/ui-microservice-86cbdb6ccb         1         1         1       30h
```

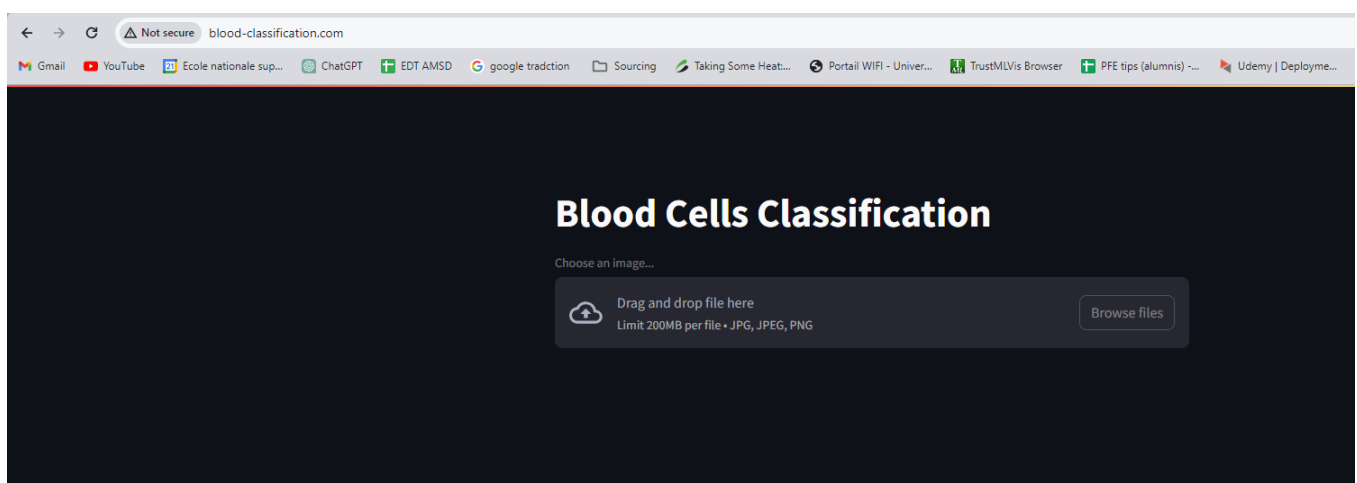
3- Changer `c :/windows/system32/drivers/etc/hosts` en ajoutant :

```
127.0.0.1 blood-classification.com
```

4- Activer un tunnel pour Minikube :

```
$ minikube addons enable ingress-dns
$ minikube tunnel
```

5- Enfin, vérifions dans navigateur Web : **blood-classification.com**



## 2.5 Déploiement dans le cloud

Le déploiement a été effectué dans le cloud GCP et le service Kubernetes Engine (GKE). Une petite [video sur Youtube](#) montre le bon fonctionnement. GKE est un service entièrement géré qui facilite le déploiement et la gestion des applications Kubernetes.

Pour déployer l'application sur GKE, on doit d'abord créer un cluster Kubernetes par l'interface graphique de GCP ou l'outil de ligne de commande gcloud.

Filter	Enter property name or value
<input type="checkbox"/> Status	Name <input type="text" value="↑"/>
<input type="checkbox"/> <input checked="" type="checkbox"/>	cluster-1
	europe-central2-a
	2
	4
	8 GB
	Notifications
	Labels

```

ja_mahmahi@cloudshell:~ (amsd-kubernetes)$ kubectl get node
NAME                                     STATUS   ROLES    AGE      VERSION
gke-cluster-1-default-pool-6fb9631d-ht33 Ready    <none>    7m14s    v1.27.3-gke.100
gke-cluster-1-default-pool-6fb9631d-tc1m Ready    <none>    7m14s    v1.27.3-gke.100
ja_mahmahi@cloudshell:~ (amsd-kubernetes)$

```

FIG. 2.7 : Création de cluster avec 2 noeuds

Une fois que le cluster est créé, on doit se connecter :

```

ja_mahmahi@cloudshell:~ (amsd-kubernetes)$ gcloud container clusters get-credentials cluster-1 --zone europe-central2-a --project amsd-kubernetes

```

FIG. 2.8 : Se connecter au cluster

Déploiement de l'application en appliquant le fichier de configuration **all.yaml**, qui peut être créé en utilisant n'importe quel éditeur de texte, avec la commande :

```
$ Kubectl apply -f all.yaml
```

```

ja_mahmahi@cloudshell:~ (amsd-kubernetes)$ kubectl get all
NAME                                     READY   STATUS    RESTARTS   AGE
pod/predict-microservice-648bdbb5c9-r272n 1/1     Running   0           32h
pod/preprocess-microservice-7db67d8659-jshq4 1/1     Running   0           32h
pod/ui-microservice-6c84f6d6f4-swkscc 1/1     Running   0           32h

NAME                                     TYPE          CLUSTER-IP      EXTERNAL-IP      PORT(S)          AGE
service/kubernetes                      ClusterIP      10.96.0.1        <none>            443/TCP          32h
service/predict-service                  ClusterIP      10.96.9.104      <none>            5000/TCP          32h
service/preprocess-service               ClusterIP      10.96.11.240     <none>            5001/TCP          32h
service/ui-service                       LoadBalancer  10.96.8.229      34.118.5.93      80:31768/TCP     32h

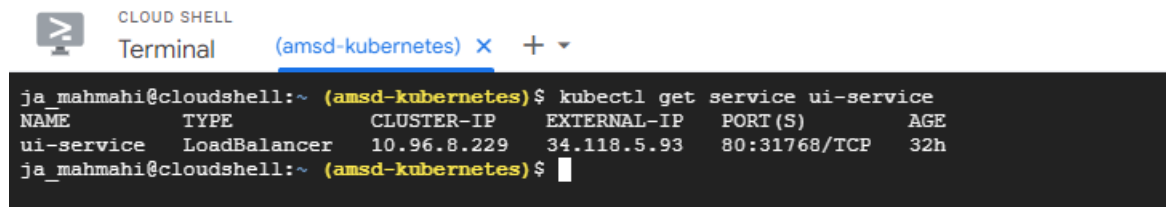
NAME                                     READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/predict-microservice      1/1     1             1           32h
deployment.apps/preprocess-microservice   1/1     1             1           32h
deployment.apps/ui-microservice            1/1     1             1           32h

NAME                                     DESIRED   CURRENT   READY   AGE
replicaset.apps/predict-microservice-648bdbb5c9 1         1         1       32h
replicaset.apps/preprocess-microservice-7db67d8659 1         1         1       32h
replicaset.apps/ui-microservice-6c84f6d6f4 1         1         1       32h
ja_mahmahi@cloudshell:~ (amsd-kubernetes)$

```

FIG. 2.9 : Configuration du cluster

Accédons à l'adresse publique de l'application



```
ja_mahmahi@cloudshell:~ (amsd-kubernetes) $ kubectl get service ui-service
NAME          TYPE          CLUSTER-IP    EXTERNAL-IP    PORT(S)          AGE
ui-service    LoadBalancer 10.96.8.229    34.118.5.93    80:31768/TCP     32h
ja_mahmahi@cloudshell:~ (amsd-kubernetes) $
```

FIG. 2.10 : Adresse publique de l'application

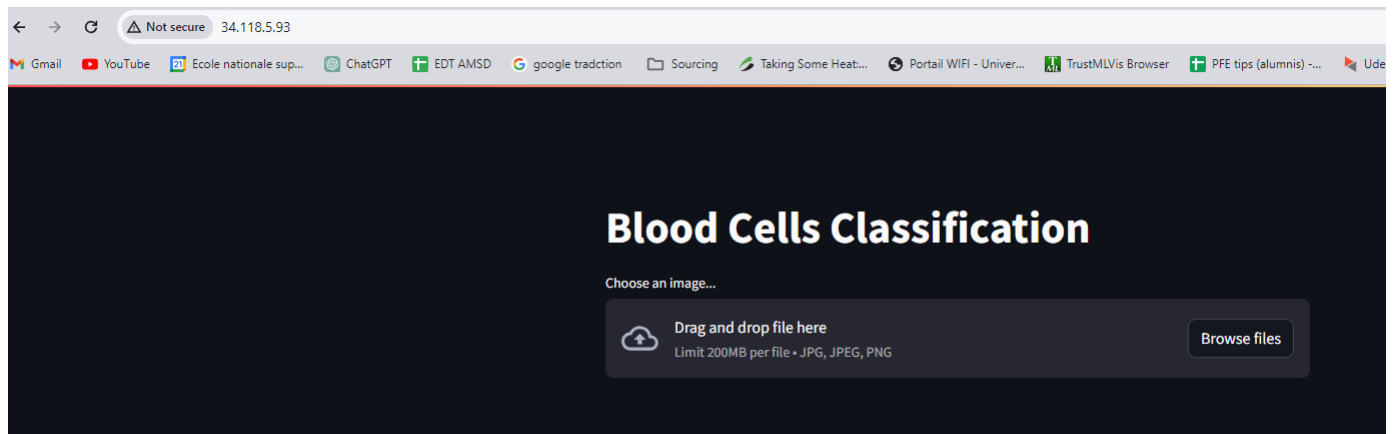


FIG. 2.11 : L'application sur internet accessible via <http://34.118.5.93/>

# Chapitre 3

## Conclusion

La mise en place de l'architecture microservices, la dockerisation des services, et l'orchestration minutieuse via Kubernetes garantissent que l'application est non seulement performante et fiable, mais également facile à maintenir et à faire évoluer. Le choix de GKE pour le déploiement souligne une volonté de profiter des meilleures ressources cloud disponibles, assurant ainsi une gestion optimale des ressources et une haute disponibilité de l'application.

En somme, ce projet reflète un parfait exemple de la manière dont les technologies modernes peuvent être appliquées pour déployer des modèles ML et ne les limite pas seulement aux jupyter notebooks, car le déploiement est un aspect crucial de la science des données, et souvent considéré comme l'étape finale du processus de développement où on transforme un modèle théorique en un outil pratique et utile.