


# TP : De Kedro à Kubeflow (via GitHub Actions & Minikube): parties 1 à 5

Owner	 louis reynouard
Tags	
Created time	@May 2, 2025 8:05 AM

## Contexte et objectifs

### 1. Mise en place du projet Kedro

- 1.1 Installation et création du squelette
- 1.2 Pipelines Kedro : YOLOv8-10 & OCR
  - Pipeline YOLOv8-10
  - Pipeline OCR (Tesseract)
- 1.3 Tests unitaires et coverage local (avant CI/CD)

### 2. Docker : Conteneuriser le projet

- 2.1 Fichier `Dockerfile`

### 3. Mise en place du Git et GitHub Actions

- 3.1 Initialiser un dépôt Git et pousser sur GitHub
- 3.2 Configuration GitHub Actions (CI/CD)

### 4. API Flask (avec mini-GUI)

### 5. Déploiement Minikube & Autoscaling

- 5.1 Installation et démarrage de Minikube
- 5.2 Manifests Kubernetes
  - 5.2.1 Probes : comment ça marche ?
  - 5.2.2 HPA (Horizontal Pod Autoscaler)

### 5.3 Écriture des manifests Kubernetes

- 5.3.1 `deployment.yaml`
- 5.3.2 `service.yaml`
- 5.3.3 `hpa.yaml`

### 5.4 Stress test & validation

- Script Python `stress_test.py`

## Contexte et objectifs

Nous allons créer un **projet d'industrialisation** complet intégrant deux IA ayant pour but la lecture de panneaux routiers :

1. **YOLOv8-10** pour la détection de panneaux routiers,
2. **Tesseract** (ou EasyOCR) pour la lecture du texte détecté.

Les étapes :

1. **Projet Kedro** : structurer les pipelines (un pipeline "détection", un pipeline "OCR", etc.).
2. **Git & GitHub Actions** : versionner le code, mettre en place un pipeline CI/CD avec seuil de couverture (80 %).
3. **API Flask** (avec un mini-GUI ) pour exposer `/predict` et manipuler rapidement des images.
4. **Docker** : conteneuriser le projet.
5. **Minikube & Kubernetes** : déployer l'API, ajouter l'autoscaling (HPA), et exécuter un stress test.
6. **Kubeflow** (optionnel mais vivement recommandé) : démontrer l'orchestration au sein du cluster, gérer l'entraînement distribué, et éventuellement le serving via KServe.

## 1. Mise en place du projet Kedro

Comme lors des UE précédentes, je vous recommande vivement de créer un environnement virtuel propre à ce TP.

### 1.1 Installation et création du squelette

1. **Installer kedro** :

```
pip install kedro
```

## 2. Créer un nouveau projet :

```
kedro new
```

- Nom : `kedro_road_sign` (ou autre).
- Vérifiez que l'arborescence suivante est générée :

```
kedro_road_sign/  
├── conf/  
├── src/  
│   ├── kedro_road_sign/  
│   │   ├── pipelines/  
│   │   └── ...  
└── ...
```

## 1.2 Pipelines Kedro : YOLOv8-10 & OCR

### Pipeline YOLOv8-10



Pour éviter tout problème de compatibilité, utilisez python 3.10 et `pip install kedro-datasets` en cas de problème de Dataset

Définissez un pipeline pour prendre en entrée des données (récupérable [ici](#) et/ou [ici](#)) de panneaux routiers et effectuer l'entraînement de ce modèle en respectant les bonnes pratiques vues dans les précédentes UE.

- **Nœuds** recommandés :
  - `load_data` + `preprocess_data` (pour charger et convertir en format YOLO).
  - `train_yolov8-10` (entraînement ou fine-tuning).
  - `evaluate_yolov8-10` (calcul de la mAP, etc.).
- **Modèle YOLOv8-10** :
  - Dans `catalog.yml`, définissez où sont stockés vos données, vos poids entraînés, etc.

### Pipeline OCR (Tesseract)

Ce pipeline aura pour but de "consommer" les detections issues du premier modèle pour renvoyer le texte du panneau.

- **Nœuds** recommandés :
  - `prepare_ocr_data` (éventuellement récupère des ROI "panneaux" depuis la détection YOLO),
  - `configure_tesseract` (installer/paramétrer Tesseract, ex. "fra.traineddata" si besoin du français),
  - `evaluate_ocr` (calcul du CER ou d'un indicateur de réussite).
- **Mise en place Tesseract** :
  - Installez `pytesseract`, configurez la variable d'env `TESSDATA_PREFIX` si nécessaire.
  - Indiquez dans `catalog.yml` l'emplacement des data de Tesseract si besoin.

Remarque : Vous pouvez créer d'autres pipelines (ex. "prediction\_pipeline", "model\_validation\_pipeline") si nécessaire.

## 1.3 Tests unitaires et coverage local (avant CI/CD)

- Dans `tests/` : écrivez vos **tests unitaires**. Par exemple,

- `test_preprocessing.py` : vérifier que le format YOLO est correct,
- `test_ocr.py` : simuler une image et s'assurer que Tesseract renvoie un résultat prévisible.
- Installez `pytest` et `pytest-cov` localement, ex.

```
pip install pytest pytest-cov
pytest --cov=src --cov-report=term-missing
```

## 2. Docker : Conteneuriser le projet

### 2.1 Fichier `Dockerfile`

Dans cette partie pratique, nous allons parcourir le processus de création d'une image Docker, avec `kedro-docker`, qui contiendra notre modèle de ML et les ressources nécessaires pour son exécution:

- Installation de docker et kedro-docker
  - Vérifier que Docker est installé

```
docker version
```

Si docker n'est pas installé, l'installer en suivant [ces instructions](#)

- Installation de kedro docker

```
pip install kedro-docker
```

- Création d'un fichier Dockerfile

Une fois le plugin `kedro-docker` installé, nous pouvons l'utiliser pour générer automatiquement un Dockerfile en utilisant les conventions recommandées par Kedro, en se basant sur votre configuration de projet existante (fichier `kedro.yml`, dépendances Python, etc.)

```
kedro docker build
```

Le Dockerfile généré inclura les étapes pour copier les fichiers nécessaires, installer les dépendances Python et exécuter le pipeline Kedro.

- Construction de l'image Docker
  - Dans le terminal, naviguez jusqu'au répertoire contenant le fichier Dockerfile et exécutez la commande suivante

```
docker build -t nom_image:tag .
```

Assurez-vous de remplacer `nom_image` par le nom souhaité pour l'image Docker, et `tag` par une version spécifique si nécessaire. Le `.` à la fin de la commande indique que le Dockerfile se trouve dans le répertoire actuel.

- Exécution du conteneur Docker

Une fois que l'image Docker est construite, nous pouvons exécuter un conteneur basé sur cette image en utilisant la commande suivante :

```
docker run -p port_local:port_conteneur nom_image:tag
```

Assurez vous de remplacer `port_local` par le port de votre machine locale sur lequel vous souhaitez exposer le modèle (ex: 5000), et `port_conteneur` par le port à travers lequel l'application dans le conteneur est accessible. Le `nom_image` et le `tag` doivent correspondre à ceux que vous avez spécifiés lors de la construction de l'image Docker.

## 3. Mise en place du Git et GitHub Actions

Dans cette UE, nous allons utiliser les principes du CI/CD vu précédemment mais avec un nouvel outil: GitHub Actions.

### 3.1 Initialiser un dépôt Git et pousser sur GitHub

1. Dans votre dossier `kedro_road_sign` :

```
git init
git add .
git commit -m "Initial commit"
```

2. Créez un repo GitHub.
3. Poussez-y votre code :

```
git remote add origin https://github.com/votre_compte/kedro_road_sign.git
git push -u origin main
```

### 3.2 Configuration GitHub Actions (CI/CD)

Créez le fichier `.github/workflows/ci.yml` et insérez y:

```
name: MLOpsPipeline

on:
  push:
    branches: ["main"]
  pull_request:
    branches: ["main"]

jobs:
  build-and-deploy:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout code
        uses: actions/checkout@v2

      - name: Setup Python
        uses: actions/setup-python@v2
        with:
          python-version: '3.10'

      - name: Install dependencies
        run: |
          sudo apt-get update
          pip install --upgrade pip
          pip install -r requirements.txt
          pip install pytest pytest-cov

      - name: Run tests with coverage
        run: |
          pytest tests/ --cov=src --cov-fail-under=80

      - name: Build Docker image
        run: |
          docker build -t youruser/yolov8-10_ocr_app:latest .

      - name: Login to DockerHub
        if: success() # on ne se connecte que si les tests ont réussi
        run: |
          echo "${{ secrets.DOCKERHUB_PASSWORD }}" | docker login -u "${{ secrets.DOCKERHUB_USERNAME }}" -
```

```
-password-stdin
```

```
- name: Push Docker image
if: success() # on ne push que si success
run: |
  docker push youruser/yolov8-10_ocr_app:latest
```



Reportez vous au cours pour l'explication de chacune des lignes ci-dessus. Elles peuvent vous être demandées à l'oral.... et demandent de faire évoluer votre projet !

#### Points clés :

- **Seuil de couverture** : `-cov-fail-under=80` impose 80 %.
- **Docker push** : n'oubliez pas de créer les secrets `DOCKERHUB_USERNAME` et `DOCKERHUB_PASSWORD` dans les paramètres du repo GitHub.

## 4. API Flask (avec mini-GUI)

Une fois les modèles entraînés et packagés (fichiers `.pt` YOLOv8-10, config Tesseract), créez un **service Flask** exposant une interface graphique permettant:

- L'upload d'une image avec renvoi d'une image annotée avec ROI + Texte du panneau
- L'upload d'une vidéo (Exemple disponible en tapant Dashcam france sur youtube) et renvoi de la vidéo annotée de la même manière que l'image

## 5. Déploiement Minikube & Autoscaling

### 5.1 Installation et démarrage de Minikube

#### 1. Installer les outils :

- `kubectl` (version  $\geq 1.25$ )
- `minikube` (version  $\geq 1.30$ )

#### 2. Démarrer Minikube avec des ressources garanties :

```
minikube start --cpus=4 --memory=8192 --driver=docker
```

#### 3. Vérifier le contexte :

```
kubectl config current-context # doit contenir "minikube"
```

### 5.2 Manifests Kubernetes

Chacune des configurations que nous allons mettre en place sert à garantir la robustesse, la disponibilité et l'évolutivité du service. Pour rappel:

Élément	Objectif
<b>resources.requests</b>	Garantit un minimum de CPU/mémoire pour chaque pod (évite qu'il soit évincé ou bloqué par un autre pod gourmand).
<b>resources.limits</b>	Empêche un conteneur de consommer excessivement les ressources du nœud (protection du cluster contre les "noisy neighbors").
<b>readinessProbe</b>	Vérifie que l'application est prête à recevoir du trafic ; le Service n'enverra pas de requêtes tant que la sonde échoue.

<b>livenessProbe</b>	Vérifie que le conteneur fonctionne toujours ; en cas d'échec, Kubernetes redémarre automatiquement le pod.
<b>Service LoadBalancer</b>	Expose votre application sur un point d'entrée stable (IP/port) et équilibre la charge entre tous les pods "Ready".

### 5.2.1 Probes : comment ça marche ?

- **Readiness probe**

- **Quoi ?** Un appel HTTP, TCP ou exécution de commande dans le conteneur.
- **Pourquoi ?** Empêche Kubernetes de diriger du trafic vers un pod qui n'est pas encore entièrement initialisé (ex : chargement de gros modèles).
- **Exemple :**

```
readinessProbe:
  httpGet:
    path: /health
    port: 5000
  initialDelaySeconds: 5 # attente avant la première vérification
  periodSeconds: 10    # intervalle entre vérifications
```

- **Liveness probe**

- **Quoi ?** Même principe qu'une readiness probe, mais vérifie que l'application n'est pas figée ou en erreur fatale.
- **Pourquoi ?** Redémarre automatiquement le pod en cas de "deadlock" ou de fuite mémoire, sans intervention manuelle.
- **Exemple :**

```
livenessProbe:
  httpGet:
    path: /health
    port: 5000
  initialDelaySeconds: 15
  periodSeconds: 20
```

### 5.2.2 HPA (Horizontal Pod Autoscaler)

- **Rôle :** Ajuste automatiquement le nombre de réplicas d'un Deployment en fonction des métriques (CPU, mémoire, custom).
- **Objectif pédagogique :**
  1. Comprendre comment Kubernetes peut scaler horizontalement un service afin de maintenir une performance constante sous charge variable.
  2. Expérimenter l'effet du scaling dynamique sur la latence et la stabilité de l'application.

```
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
  name: yolov8-10-ocr-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: yolov8-10-ocr-deployment
  minReplicas: 1
  maxReplicas: 5
  metrics:
    - type: Resource
```

```
resource:
  name: cpu
target:
  type: Utilization
  averageUtilization: 50
```

## 5.3 Écriture des manifests Kubernetes

Vous devez fournir trois fichiers YAML :

- 5.2.1 `deployment.yaml`
- 5.2.2 `service.yaml`
- 5.2.3 `hpa.yaml`

Chaque fichier contient des placeholders ( `<...>` ) que vous adapterez à votre projet.

### 5.3.1 `deployment.yaml`

#### Théorie & aides

- Un **Deployment** gère le cycle de vie de vos pods et garantit le nombre de réplicas désiré.
- Les **resources.requests** et **limits** assurent un usage contrôlé de la CPU/mémoire.
- Les **probes** (readiness & liveness) garantissent la santé et la disponibilité de chaque pod.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: yolov8-10-ocr-deployment    # À renommer si besoin
  labels:
    app: yolov8-ocr                # Label cohérent avec service & HPA
spec:
  replicas: 1                      # Nombre initial de pods
  selector:
    matchLabels:
      app: yolov8-ocr
  template:
    metadata:
      labels:
        app: yolov8-ocr
    spec:
      containers:
        - name: yolov8-ocr-container
          image: <VOTRE_UTILISATEUR>/<NOM_IMAGE>:<TAG> # Ex. youruser/yolov8-10_ocr_app:latest
          ports:
            - containerPort: 5000    # Port exposé par l'application Flask
          resources:
            requests:
              cpu: "200m"
              memory: "512Mi"
            limits:
              cpu: "500m"
              memory: "1Gi"
          readinessProbe:
            httpGet:
              path: /health
              port: 5000
```

```

initialDelaySeconds: 5    # attendre 5 s avant 1re vérif.
periodSeconds: 10        # vérification toutes les 10 s
livenessProbe:
  httpGet:
    path: /health
    port: 5000
initialDelaySeconds: 15   # attendre 15 s avant 1re vérif.
periodSeconds: 20         # vérification toutes les 20 s

```

- **Placeholders à remplir**

- `<VOTRE_UTILISATEUR>` , `<NOM_IMAGE>` , `<TAG>`
- Ajustez `replicas` selon votre validation initiale.

### 5.3.2 `service.yaml`

#### Théorie & aides

- Un **Service** de type **LoadBalancer** crée un point d'entrée unique, équilibre la charge et garantit que seules les instances "Ready" reçoivent du trafic.

```

apiVersion: v1
kind: Service
metadata:
  name: yolov8-10-ocr-service    # Doit matcher avec les vues externes
  labels:
    app: yolov8-ocr
spec:
  type: LoadBalancer           # Expose sur IP externe / NodePort en local
  selector:
    app: yolov8-ocr             # Lie le service aux pods marqués app=yolov8-ocr
  ports:
    - protocol: TCP
      port: 80                  # Port exposé à l'extérieur du cluster
      targetPort: 5000          # Port sur lequel tourne Flask dans le pod

```

- **Points de vigilance**

- En local Minikube, `type: LoadBalancer` est mappé en NodePort.
  - Sur la plupart des clouds (AWS, GCP, Azure), un Service de type **LoadBalancer** provisionne automatiquement un load-balancer externe. En revanche, Minikube ne dispose pas d'un LB cloud :
    - **Minikube** convertit toute déclaration `type: LoadBalancer` en **NodePort** sous le capot.
    - Concrètement, Kubernetes expose alors votre service sur un port élevé du nœud (généralement entre 30000 et 32767).
    - Vous pouvez voir ce port avec :

```
kubectl get svc yolov8-10-ocr-service
```

et accéder à votre appli via `http://<IP_MINIKUBE>:<NODE_PORT>` .

- Pour simplifier, utilisez :

```
minikube service yolov8-10-ocr-service --url
```

qui ouvre un tunnel et vous retourne directement une URL exploitable.



- Utiliser `minikube service yolov8-10-ocr-service --url` pour récupérer l'URL.

### 5.3.3 hpa.yaml

#### Théorie & aides

- Le **HorizontalPodAutoscaler** (HPA) ajuste dynamiquement le nombre de pods selon la consommation CPU (ou d'autres métriques).
- Permet d'assurer une réponse rapide sous forte charge et de réduire les coûts en période creuse.

```
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
  name: yolov8-10-ocr-hpa          # Nom du HPA
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: yolov8-10-ocr-deployment # Doit correspondre à metadata.name du Deployment
  minReplicas: 1
  maxReplicas: 5
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 50    # Vise 50 % d'utilisation CPU avant de scaler
```

#### • À tester

- Vérifier `kubectl get hpa yolov8-10-ocr-hpa` pour observer `CURRENT` vs `TARGET` CPU.
- Vous verrez une sortie semblable à :

NAME	REFERENCE	TARGETS	MINPODS	MAXPODS	REPLICAS	AGE
yolov8-10-ocr-hpa	Deployment/yolov8-10-ocr-deployment	75%/50%	1	5	2	10m

- **TARGETS** : `CURRENT%/DESIRED%` (ex. 75 % d'usage CPU pour un objectif à 50 %).
- **REPLICAS** : nombre de pods actuellement déployés.
- Forcer la charge et vérifier que `DESIRED` augmente jusqu'à `maxReplicas`.

#### Conseil pour l'avancement en autonomie :

1. **Validez** chaque manifest en local :

```
kubectl apply -f deployment.yaml
kubectl apply -f service.yaml
kubectl apply -f hpa.yaml
```

2. **Surveillez** l'état :

```
kubectl get deployments,svc,hpa,pods -o wide
```

3. **Debuggez** avec :

- `kubectl describe pod <nom-du-pod>`
- `kubectl logs <nom-du-pod>`

4. Utilisez **Kubernetes Dashboard** pour aider à visualiser:

```
minikube dashboard
```

## 5.4 Stress test & validation

### Script Python `stress_test.py`

Vous n'avez qu'à modifier la variable `URL` pour pointer vers votre service :

```
#!/usr/bin/env python3
"""
stress_test.py
Envoie N requêtes concurrentes à l'endpoint /predict d'une application Flask déployée sur Kubernetes.
"""

import argparse
import concurrent.futures
import time
import requests
from io import BytesIO
from PIL import Image

# -----
# Configuration à adapter
# -----
URL = "http://<VOTRE_URL>:80/predict" # Ex. obtenu via `minikube service --url`
IMAGE_PATH = "test_image.jpg"
NUM_REQUESTS = 100
MAX_WORKERS = 10
# -----

def send_request(_):
    """Charge l'image et poste vers l'API."""
    with Image.open(IMAGE_PATH) as img:
        buf = BytesIO()
        img.save(buf, format='JPEG')
        buf.seek(0)
        files = {'file': ('test_image.jpg', buf, 'image/jpeg')}
        start = time.time()
        resp = requests.post(URL, files=files)
        latency = time.time() - start
    return resp.status_code, latency

def main():
    parser = argparse.ArgumentParser()
    parser.add_argument('--url', default=URL, help="URL de l'API /predict")
    parser.add_argument('--requests', type=int, default=NUM_REQUESTS, help="Nombre total de requêtes")
    parser.add_argument('--workers', type=int, default=MAX_WORKERS, help="Nombre de threads")
    args = parser.parse_args()

    latencies = []
    statuses = []

    print(f"Lancement de {args.requests} requêtes vers {args.url} avec {args.workers} workers...")
```

```

with concurrent.futures.ThreadPoolExecutor(max_workers=args.workers) as exe:
    futures = [exe.submit(send_request, i) for i in range(args.requests)]
    for f in concurrent.futures.as_completed(futures):
        status, lat = f.result()
        statuses.append(status)
        latencies.append(lat)

print("Statuts reçus:", set(statuses))
print(f"Latence moyenne : {sum(latencies)/len(latencies):.3f}s")
print(f"Latence p95 : {sorted(latencies)[int(0.95*len(latencies))-1]:.3f}s")

if __name__ == "__main__":
    main()

```

### Consignes :

- Remplacez `URL` ou passez `-url http://...`.
- Fournissez une image `test_image.jpg` représentative de votre dataset.