Fiche 2

API Serving API, Dockerisation & Interface : - Entraîner et servir votre modèle ML : voir Fiche 1. - Concevoir l'API permettant de servir leur modèle ML. - Dockeriser leur application. - Développer une interface optionnelle (Streamlit / Flask / Next.js).

---

1. Implémentation de l'API

1. Choix du framework pour la création de l'API

- FastAPI (recommandé)
- Streamlit
- Flask-RESTful
- Django REST Framework
- ...

2. Intégration du Model Pipeline dans l'API

    - Préciser ce que l'API doit faire concrètement au moment de l'inférence (inférence = servir des prédictions), et ce qu'elle doit laisser au pipeline de données / modèle.
    - Par exemple, l'API ne doit pas relancer le pipeline d'entraînement ni exécuter l'ETL complet : ces tâches sont hors production real-time.
    - L'API doit charger et utiliser les artefacts d'inférence sérialisés : `model.joblib`, `KPI`, metadata (par exemple `features_order.json`) si nécessaire.

3. Les endpoints possibles

    - **GET** `/health` : renvoie l'état du service
    - **POST** `/predict` : reçoit une requête, renvoie la prédiction
    - **POST** `/train` (optionnel) : permet de réentraîner
    - **GET** `/metrics` : expose les métriques ML/system

4. Chargement du modèle

    - Parexemple (à intégrer dans `src/api/model_loader.py`)

- Centraliser ici le chargement du modèle entraîné pour que l'API puisse récupérer une instance unique (`get_model()`) sans recharger l'artefact à chaque requête.
- Ajouter une fonction `load_model(path: str)` qui lit le fichier (joblib/pickle) depuis `models/` ou, si besoin, depuis S3 en utilisant les credentials AWS déjà configurés.
- Prévoir un petit cache ou singleton (ex. variable module) et lever une erreur claire si le modèle est introuvable, afin que `/health` puisse signaler l'état `model_loaded.

5. Pré-calcul de KPIs

- Avant la prédiction, les KPIs importants doivent être chargés ou recalculés (ex. distribution des features, seuils de confiance, latence estimée). Ces informations peuvent servir à valider les entrées, enrichir la réponse (p.ex. confidence, score de qualité) et alimenter le monitoring et les alertes. Selon le besoin de latence et de partage entre instances, stockez ces KPIs en local (fichier/SQLite), dans S3, ou dans une table légère (DuckDB/Aurora) pour réutilisation.

- Si votre application prévoit d'autres KPIs pour un tableau de bord, pensez également à exposer ces métriques via l'API. Exemple d'adaptation de l'URL principale `/metrics` en sous-URLs :

```
GET /metrics                        -> résumé global (tous types)
GET /metrics/model                  -> métriques ML (accuracy, precision, recall,
f1)
GET /metrics/system                 -> métriques système (latence, error_rate,
throughput)
GET /metrics/kpis                   -> KPIs métier / business spécifiques
GET /metrics/feature-distribution   -> distribution des features (optionnel, paginé)
```

2. Dockerisation

1 Dockerfile type - Installation dépendances - Copie du code - Commande `uvicorn` ou `gunicorn`

2 Fichiers importants

- `Dockerfile`
- `.dockerignore`

- `requirements.txt`

3 Test local

- Installer Docker Desktop
- Construire et lancer l'image localement

```
docker build -t groupeX-mlops .
docker run -p 8000:8000 groupeX-mlops
```

---

3. Interface utilisateur (optionnel)

- Streamlit (simple) `app.py`
- Flask (minimal)

  ```
  templates/
  static/
  app.py
  ```

- Next.js / Angular (avancé)