MLOps & Industrialisation

Jour 5 — Matin

Julien Rolland

Formation M2 Développement Fullstack

Jour 5

Plan du module

1. Du notebook à la production
2. Sérialiser et optimiser
3. Serving avec FastAPI
4. Docker et CI/CD

Le constat

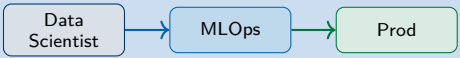
« *It works on my Notebook* » n'est pas une mise en production.

Contraintes du monde réel :

- **Disponibilité :** 1 000 req/s sans s'effondrer
- **Sécurité :** injection de données corrompues, vol de poids du modèle
- **Reproductibilité :** Python 3.10 vs 3.12, PyTorch 2.x vs 1.x

Le rôle du MLOps

Créer un **pont stable** entre le Data Scientist et le DevOps.



Au programme

Sérialisation ▪ Serving ▪ Docker ▪ Monitoring ▪ CI/CD

Sérialisation (1/2) — le danger de Pickle

Pickle — le réflexe Python

```python
import pickle

with open("model.pkl", "wb") as f:
    pickle.dump(model, f)

# Charger -- danger si source inconnue !
with open("model.pkl", "rb") as f:
    model = pickle.load(f)
```

Avantage

Simple, natif, aucune dépendance supplémentaire.

Risque de sécurité critique

Pickle peut **exécuter du code arbitraire** à la désérialisation.

Ne jamais charger un `.pkl` d'origine inconnue.

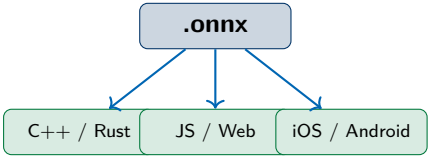
Rigidité = dette technique

- Nécessite le **même code source** et les mêmes versions de classes
- Incompatible entre versions de Python
- Inutilisable depuis Java, Go, Rust, etc.

ONNX — Open Neural Network Exchange

Modèle sous forme de **graphe statique figé** :

- **Standard universel :** portable partout
- **Vitesse :** souvent plus rapide que PyTorch brut




```python
import torch.onnx
torch.onnx.export(
    model, dummy_input, "model.onnx",
    input_names=["x"], output_names=["y"],
)
import onnxruntime as ort
sess = ort.InferenceSession("model.onnx")
out = sess.run(None, {"x": x_np})
```

Alternatives PyTorch

- `state_dict` — poids seuls
- `torch.jit.script` — TorchScript
- `safetensors` — sans exécution de code

L'IA « On-Edge » — zéro serveur, zéro latence

Concept

Faire tourner l'IA **directement chez l'utilisateur** : navigateur, mobile, IoT.

Outils :

- **ONNX Runtime Web** — modèle ONNX dans le navigateur
- **TensorFlow.js** — JavaScript natif
- **MediaPipe / CoreML** — vision sur mobile

Avantages stratégiques

- **Coût :** facture serveur proche de 0€ — le client fournit la puissance de calcul
- **Confidentialité :** la donnée ne quitte jamais l'appareil — RGPD-friendly par design
- **Réactivité :** pas de latence réseau, idéal pour la vidéo temps réel

Contrainte

Taille limitée par la RAM du device — d'où l'importance de la quantization.

Optimisation de l'inférence — « Speed is a Feature »

Quantization

Passer les poids de **Float32 à Int8**.

- Gain de vitesse $\times 3$
- Poids du modèle $-75\,\%$
- Perte de précision $< 1\,\%$

Pruning

Supprimer les connexions dont les poids sont proches de zéro.

- Réseau **creux** (*sparse*)
- 50–90 % des poids supprimables
- Fine-tuning nécessaire après

Compilation Hardware

Compiler le graphe pour la carte cible :

- **TensorRT** — GPU NVIDIA
- **OpenVINO** — CPU Intel
- **CoreML** — Apple Silicon

Ordre recommandé : **Quantization** → **Pruning** → **Compilation** — chaque étape s'applique indépendamment

Anti-pattern à bannir

Charger le modèle **dans** la route API → 500 Mo rechargés à chaque requête → latence + RAM saturée.

Le pattern Singleton

Le modèle est chargé **une seule fois** au démarrage, maintenu en mémoire partagée entre toutes les requêtes.

```python
from contextlib import asynccontextmanager
from fastapi import FastAPI

@asynccontextmanager
async def lifespan(app: FastAPI):
    app.state.model = load_model("model.onnx")
    yield
    app.state.model = None
app = FastAPI(lifespan=lifespan)
@app.post("/predict")
async def predict(data: InputData):
    return app.state.model.run(data)
```

Événements `lifespan`

- **Startup :** charger modèle, connexion BDD, warmup GPU
- **Shutdown :** libérer GPU, fermer connexions proprement

Pydantic — typage fort

Valider automatiquement les entrées JSON :

- Évite les crashs sur données malformées
- Erreurs 422 auto-générées avec détail
- Intégration native FastAPI

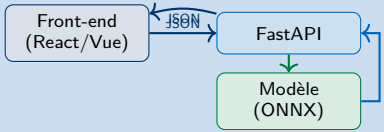
Asynchronisme

`async def` pour ne pas bloquer l'Event Loop pendant les I/O :

- Requêtes BDD, appels HTTP externes
- Inférence : déléguer au `ThreadPoolExecutor`

Swagger UI automatique

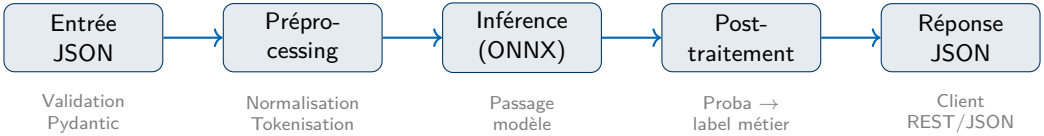
Documentation sur `/docs` — indispensable pour l'intégration Front-end.



Intérêt terrain

Vous connaissez déjà le paradigme REST — FastAPI l'applique à l'IA.

Le pipeline d'inférence complet



Point critique : cohérence du prétraitement

Le preprocessing doit être **strictement identique** au code d'entraînement — même scaler, même tokenizer, mêmes paramètres.

Bonne pratique

Empaqueter le prétraitement **dans** le modèle ONNX ou le versionner avec lui (ex: `sklearn Pipeline` sérialisé avec `safetensors`).

Pourquoi Docker en IA ?

- **Environnement figé :** Python, CUDA, drivers toujours identiques en dev, staging, prod
- **Isolation GPU :** NVIDIA Container Toolkit
- **Reproductibilité :** même image partout

Le problème du poids

Les images Data Science sont lourdes (plusieurs Go) à cause de PyTorch et CUDA.

Solution : images **slim** + ne copier que le nécessaire.

Bonnes pratiques

- Partir de `python:3.10-slim` (pas `:latest`)
- **Multi-stage build** : compiler dans une image lourde, exécuter dans une image légère
- Ne jamais commiter des poids ou des données dans l'image
- Utiliser `.dockerignore`

GPU en prod

`nvidia/cuda:12.1-base` comme base + `-gpus all` au `docker run`.

Dockerfile — optimisation pour l'ingénieur

```dockerfile
FROM python:3.10-slim AS base
WORKDIR /app

# Layer caching : deps avant le code
COPY requirements.txt .
RUN pip install --no-cache-dir -r
    requirements.txt

# Code source (invalide le cache si change)
COPY src/ ./src/

# Securite : pas de root en prod
RUN useradd --no-create-home appuser
USER appuser

EXPOSE 8000
CMD ["uvicorn", "src.main:app", "--host", "
    0.0.0.0"]
```

Layer caching

Les dépendances changent rarement : les installer **avant** de copier le code.

Docker réutilise le cache si `requirements.txt` n'a pas changé — rebuild quasi instantané.

Sécurité : non-root

Ne jamais lancer l'API en `root`. Un `useradd` + `USER` réduit drastiquement la surface d'attaque.

Monitoring & Dérives (*Drift*)

Pourquoi mon modèle « meurt » en prod ?

- **Data Drift :** les données changent (ex: nouvel argot sur les réseaux sociaux)
- **Concept Drift :** le monde change (ex: prédictions avant/après une crise)
- **Infrastructure :** mise à jour de dépendance, changement de format d'entrée

Pipeline de surveillance

1. **Logger** les prédictions en production
2. **Collecter** la vérité terrain
3. **Comparer** distributions et métriques
4. **Alerter** si dégradation $>$ seuil
5. **Ré-entraîner** sur les nouvelles données

Outils

Evidently ▪ **WhyLogs** ▪ **MLflow** ▪ **Grafana**

Automatisation : CI/CD ML

- **Tests unitaires :** preprocessing, postprocessing
- **Tests modèle :** accuracy $>$ seuil minimal sur un jeu de validation figé
- **Build & push** de l'image Docker
- **Déploiement :** staging puis prod

Stratégies de déploiement

- **Canary :** 5 % du trafic sur le nouveau modèle
- **A/B Testing :** comparaison contrôlée
- **Shadow mode :** logs sans impact client

Ce que vous avez appris

- Sérialiser avec ONNX pour la portabilité
- Servir avec FastAPI (pattern Singleton)
- Conteneuriser avec Docker (non-root, layer caching)
- Surveiller les dérives en production

Place au Challenge !

Vous avez le moteur, vous avez la boîte — maintenant mettez votre modèle en production.