Comment déployer un modèle ML

Master TRIED

Connaissance Entreprise

28/01/2026

pass Culture

whoami

Carole IBRAHIM

Data Scientist
pass Culture

CV Rapidos

Expérience Professionnelle:

- *2024 - présent* : Data Scientist au **pass Culture**, plateforme du ministère de la Culture pour les jeunes de 15-21 ans
- *2023-2024 :* ingénieure IA chez **Lusis** (leader français des systèmes de paiements)

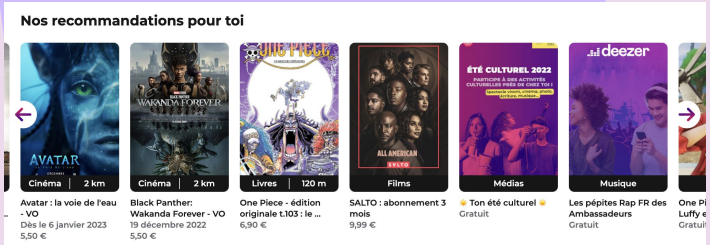
Formation:

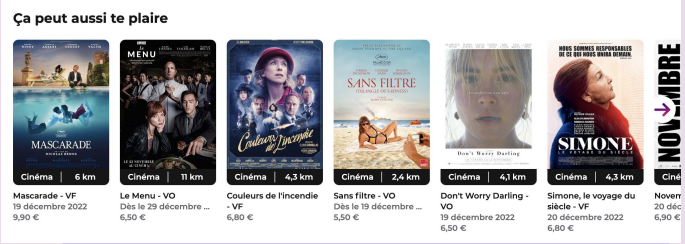
- *2023:* master **TRIED** + diplôme d'ingénieur de **TSP**

Personal Website | Github | Linkedin | Scholar

whoarewe

we are Pass Culture





pass Culture est un dispositif du gouvernement français qui offre aux jeunes de 15-21 ans une somme à dépenser sur notre app.

L'app a un grand catalogue d'offres culturelles (cinéma, livres, théâtre, concerts, ..)

L'équipe DS du pass Culture crée (entre autres) :

- Système de recommandations
- Modèles ML (artistes similaires, offres similaires, prédiction finance..)
- Chatbot et agents IA

ps aux | grep “Objectifs de l’intervention”

1. **Comprendre les différents métiers de la data en industrie et les responsabilités de chacun ;**

2. **Comprendre le cycle de développement et de vie d’un modèle ML en entreprise;**

3. **Comprendre les pratiques de déploiement et *serving* de modèle ML;**

4. **Mettre la main à la pâte 🍝 en :**
   a. Créer un modèle ML dummy
   b. Créer une API avec FastAPI
   c. encapsuler l’API dans une image/conteneur Docker

→ ***Le tout dans un environnement <u>uv</u> et avec les standards de clean code de l'industrie.***
→ 🍒***Cerise sur le gâteau : nouveau repo dans votre git perso pour séduire les recruteurs*** 👄

Petit jeu : J’ai jamais

1. Qui a déjà prédit iris avec un RF ?
2. Qui a déjà créé une API ?
3. Qui a déjà déployé un modèle ML en prod?
4. Qui a déjà utilisé MLflow ?
5. Qui a déjà fait du FastAPI ?
6. Qui connaît Pydantic ?
7. Qui a déjà fait du Docker ?
8. Qui a déjà utilisé uv ?
9. Qui est sous Windows 🤬 ?

Plan de la journée

Matin

09h00 – 09h30 · CM1 — Présentation du cycle de vie d'un projet ML en entreprise

09h30 – 10h30 · CM2 — Model Serving et APIs

☕ **Pause (10h30 – 10h45)**

10h45 – 11h30 · CM3 — Coder une API en Python

11h30 – 12h · TP1 — Setup du projet

12h – 12h15 · TP2 — Entraîner un modèle ML dummy

Après-midi

14h – 15h · TP3 — Prise en main de FastAPI

15h – 15h45 · TP4 — Développer une API pour servir un modèle ML

☕ **Pause (15h45 – 16h)**

16h – 16h30 · CM4 — Model deployment

16h30 – 17h · TP5 — Docker pour encapsuler API

BONUS: **17h – 17h30 · CM 5 —** MLflow

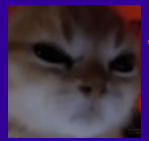
🎲 Règles du jeu :

👩🏻‍🏫 Pendant les CMs :

- Interrompez-moi dès que vous ne comprenez pas ou si je vais trop vite.
- Après chaque slide chargée, je vous laisse 10 sec pour relire seul la slide

💻 Pendant les TPs :

- Lire chaque ligne de code et commentaire
- Aller voir la documentation pour comprendre les différentes fonctions
- Soyez curieux, testez vous-même des bouts de code non prévus dans le TP
- Évitez d'utiliser les IA le plus possible !



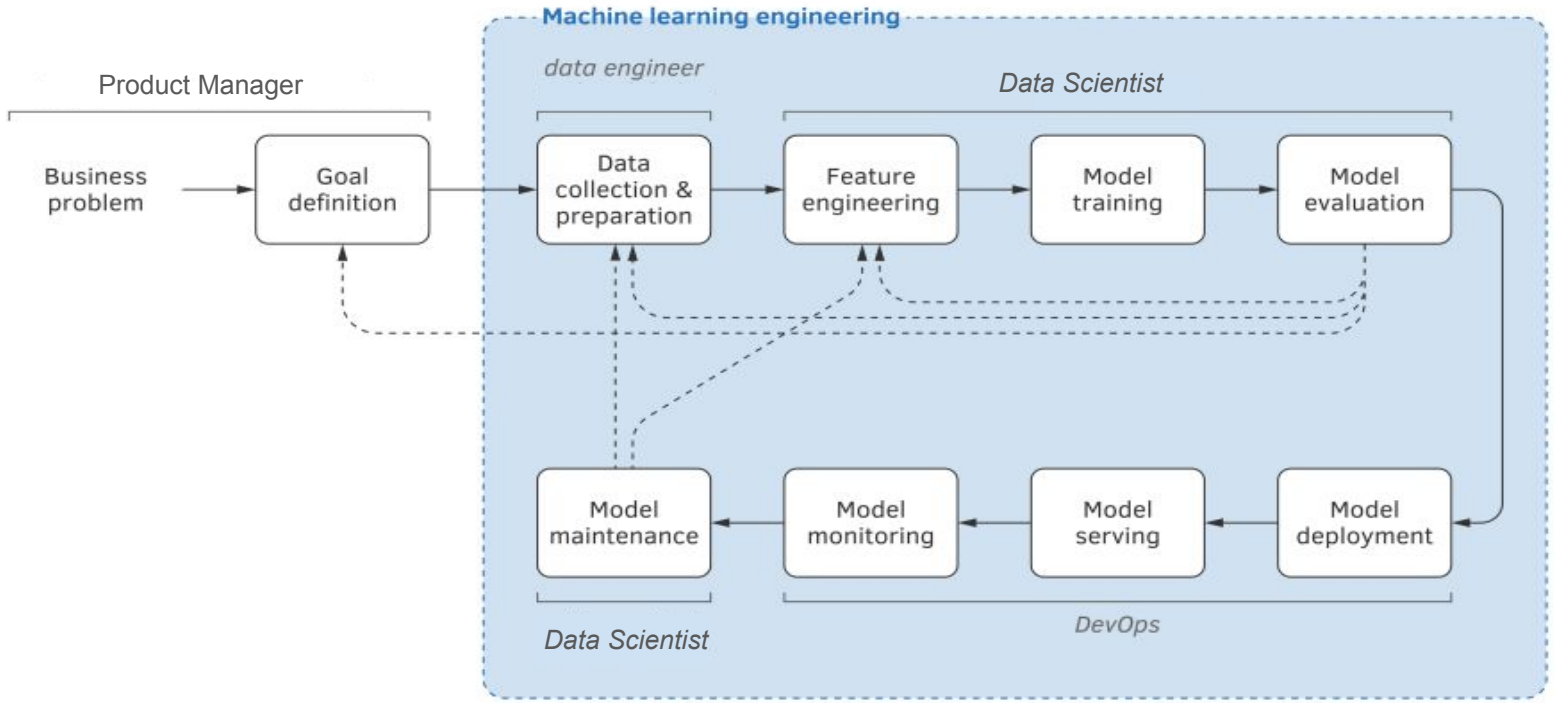
<u>SURTOUT :</u> NE REGARDEZ PAS AUTRE CHOSE PENDANT LE COURS (RÉPONDEZ À VOTRE CRUSH PLUS TARD)

Arrrr You Ready Kids?!?

9h00
CM 1:
Présentation
du cycle de vie
d'un projet ML
en entreprise
9h30
kwejk.pl

Part 1 of 10

Le cycle de vie d'un modèle ML en entreprise + les différents métiers de la data



source : Machine Learning Engineering book by Andriy Burkov

Étape 1 : Cadrage du problème (business problem)

🎯 **Objectif :** Transformer un besoin métier en problème ML clair et mesurable.

⚒️ **How To :**

- Comprendre le **besoin** business et les **KPI** associés
- Vérifier que le ML est pertinent (vs règles, heuristiques)
- Définir : les **features** candidates, les **métriques de succès** (offline / online)
- Identifier les contraintes : latence, **coût**, explicabilité, **réglementation, deadlines**

👪 **Métiers :**

- **Product Manager**
- Data Scientist
- **Stakeholders métier**
- ML Engineer (faisabilité technique)

📦 **livrables :**

- Problématique **ML formalisée (**si agile, fiche projet cadrée, propositions de tickets, roadmap**)**
- **Métriques** de succès définies

Étape 2 : collecte des données

🎯 **Objectif :** Vérifier que les données à disposition permettent de **résoudre** le problème ML.

⚒️ **How To :**

- Identifier les **sources** de données (DB, logs, APIs, ..)
- Vérifier : la **qualité**, le volume, le **biais**, la fraîcheur
- Analyse des données (données manquantes, outliers, ..)
- Demander la création de tables de données s'il le faut

👪 **Métiers:**

- Data Engineer
- Data Scientist

📦 **livrables :**

- Dataset exploitable

Étape 3: Exploration & préparation des données

🎯 **Objectif : Comprendre les données et les transformer en features exploitables.**

⚒️ **How To :**

- Analyse exploratoire (**EDA**) (outliers, analyse statistique, univariée, multivariée, etc)
- **Feature engineering**
- **Nettoyage**, normalisation
- **Split** train / validation / test (ou split temporel pour les séries temporelles)

👪 **Métiers :**

- Data Scientist (principal)

📦 **Livrables :**

- **Pipeline** de préparation de données reproductible
- **Jeu de données prêt** pour l’entraînement

Étape 4: Prototypage & entraînement (POC)

🎯 **Objectif : Prouver** rapidement la valeur (ou non) du ML pour résoudre ce problème.

⚒️ **How To :**

- Modèles simples en priorité (**baseline**)
- Comparaison de plusieurs approches
- Entraînement offline
- **Tracking** des expériences (MLflow, W&B…)

👪 **Métiers :**

- Data Scientist
- ML Engineer
- Product Manager (lecture des résultats, sync avec les stakeholders)

📦 **livrables :**

- **Modèle** POC
- Résultats de **performance** mesurés

Étape 5: Évaluation & validation

🎯 **Objectif** : **décider si le modèle peut aller en production.**

🛠️ **How To :**

- Évaluation sur jeux de test
- Analyse d’erreurs
- Tests de **robustesse**
- **Validation métier** des résultats

👪 **Métiers :**

- Data Scientist
- Product Manager
- Stakeholders métier

📦 **livrables :**

- **Rapport** d’évaluation
- **Décision Go / No-Go**

Étape 6: Industrialisation du modèle

🎯 **Objectif : Transformer un notebook en composant industriel.**

⚒️ **How To :**

- **Refactorisation** du code
- Versioning :
  - code (git)
  - données (dvc, mlflow)
  - modèles (mlflow, registry)

👪 **Métiers :**

- Data Scientist
- ML Engineer
- Software / Backend Engineer (si code à mettre dans le produit)

📦 **livrables :**

- Code **production-ready**, performant, propre, maintenable
- Pipeline ML industrialisé

Étape 7: Déploiement et Serving

🎯 **Objectif : Rendre le modèle accessible aux utilisateurs ou systèmes dans un environnement de production.**

🛠️ **How To :**

- **Serving** :
  - création d'API
- Choix de **l'infrastructure** :
  - VM
  - container + Kubernetes
  - Serverless
- **CI/CD** ML

👪 **Métiers :**

- ML Engineer
- DevOps Engineer
- Backend Engineer (si ça touche au backend de ton produit)

📦 **livrables :**

- Modèle déployé en production
- **Endpoint** ou **job** opérationnel

Étape 8: Monitoring & maintenance

🎯 **Objectif : Garantir la performance du modèle dans le temps.**

⚒️ **How To :**

- Monitoring :
  - performance modèle
  - data drift / concept drift
    - data drift : la distribution des données a changé
    - concept drift : la relation entre les features et le target a changé
  - latence, erreurs
- Alertes
- Retraining manuel ou automatique

👪 **Métiers :**

- ML Engineer
- Data Scientist
- DevOps Engineer

📦 **livrables :**

- Dashboards de monitoring
- Stratégie de retraining

Étape 9: Amélioration continue

🎯 **Objectif : Faire évoluer le modèle avec le produit et les usages.**

⚒️ **How To :**

- Feedback utilisateurs
- A/B tests
- Nouvelles features
- Nouvelles versions du modèle

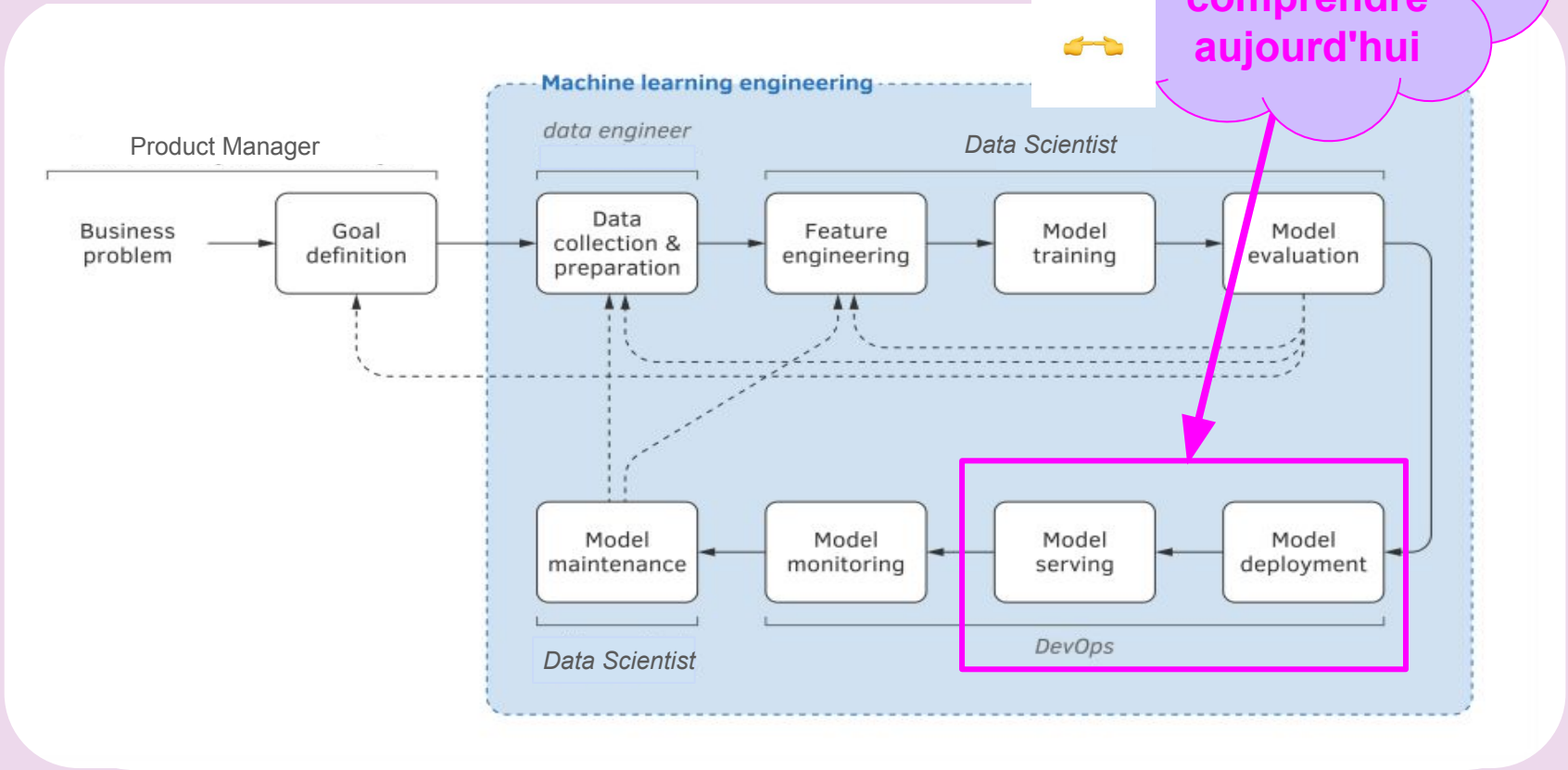
👪 **Métiers :**

- Product Manager
- Data Scientist
- ML Engineer

📦 **livrables :**

- Nouvelles versions du modèle
- Gain de performance mesuré

Ce qu'on essayera de comprendre aujourd'hui



source: Machine Learning Engineering book by Andriy Burkov

“Deployment” vs “Serving”

→ Les deux termes sont souvent utilisés de manière **interchangeable pour dire « mettre un modèle en prod »,** mais, pour être rigoureux :

Deployment

=

Rendre un modèle entraîné **utilisable en production**

Serving

=

Exposer un modèle entraîné via une interface (souvent une **API**)

CM 2: Model Serving et APIs

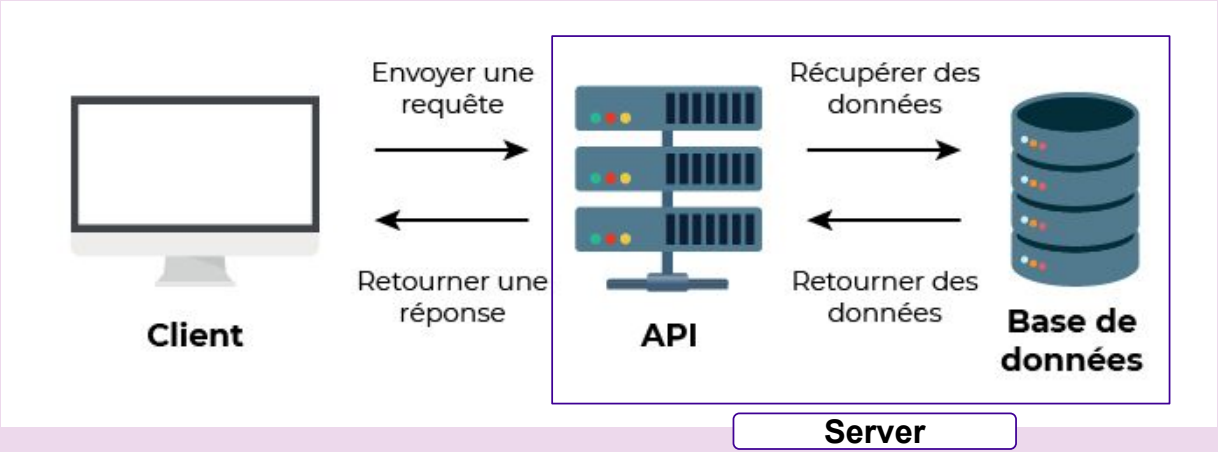
Part 2 of 10

Model Serving : qu'est-ce qu'une API ?

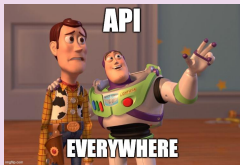
Une **API** (Application Programming Interface) est une **interface** standardisée qui permet à des applications de communiquer entre elles en **échangeant** des **données** ou des services, sans exposer leur implémentation interne.

En pratique:

- Un **serveur** expose des **endpoints** (des paths pour chercher, mettre, les données)
- Un **client** envoie des **requêtes** au serveur (**HTTP**, **websocket**)
- Les données (**requête** et **response**) sont échangées dans un format défini (**json**, xml, html, etc)



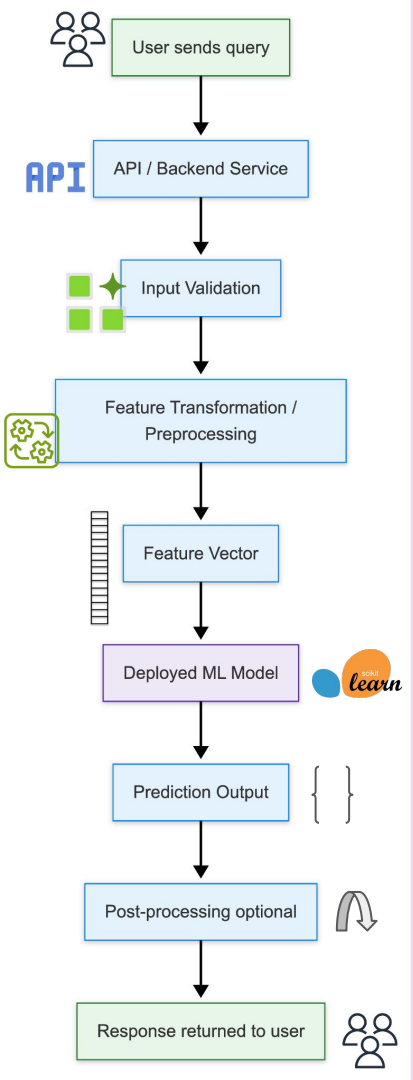
Chaque site web, application mobile ou logiciel utilise des APIs pour échanger des données et des services.



Model Serving : à quoi sert une API dans le contexte ML ?

Une **API** qui “**serve**” un modèle ML va :

1. Récupérer les données des utilisateurs (input features),
2. Effectuer du preprocessing
3. Faire une prédiction avec le modèle `(model.predict(input_features)`
4. Postprocess la prédiction
5. Envoyer la prédiction finale à l'utilisateur.

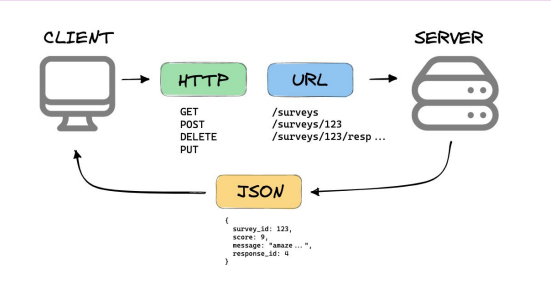


Différents types d'API

Types d'API

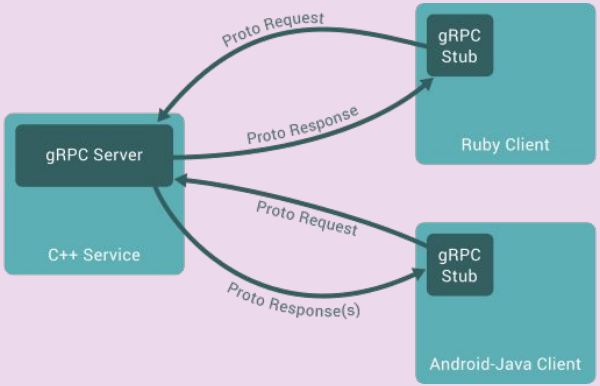
REST API

REST : expose des services via des endpoints HTTP et échange des données en JSON.



gRPC

gRPC : communication rapide entre services via HTTP/2 et messages binaires fortement typés.



GraphQL

GraphQL : permet au client de demander exactement les données dont il a besoin via une requête unique.

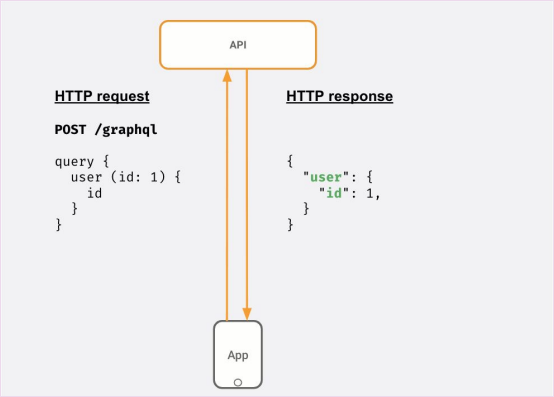


Tableau comparatif des types d'API

Fonctionnalité	REST	gRPC	GraphQL
Data format	JSON	Protobuf (binary)	JSON
Protocole	HTTP	HTTP/2	HTTP
Performance	Medium	⭐ très bonne	Medium, très bonne si deployer pour le bon usecase
Typing des données	Weak	⭐ très bonne	Strong
Debugging	⭐ Facile	Difficile	Medium
Usecases	Public APIs, ML, backend web & mobile	Internal backend-to-backend, API demandant haut débit	Frontend complexe, différents types de requêtes
Forces	simple, passe partout, facile à coder	très rapide et efficace, typing fort, streaming bidirectionnel	pas de over-fetchng, flexible, un seul endpoint
Faiblesses	over-fetching, pas de strict contract	complexe, difficile à coder	backend complexe, caching difficile

Anatomie d'une URL

origin

hostname

site

query

search

pathname

`https://carole-ibrahim.github.io/blogs/1/?user=carole&level=expert`

scheme

protocol

TLD

eTLD

registrable domain
(eTLD+1)

endpoint

début de
query params

1re query
param

séparateur de
query params

2e query
param

Pour analyser une URL, aller sur :
`https://web.dev/articles/url-parts#url_analyzer`

URL analyzer

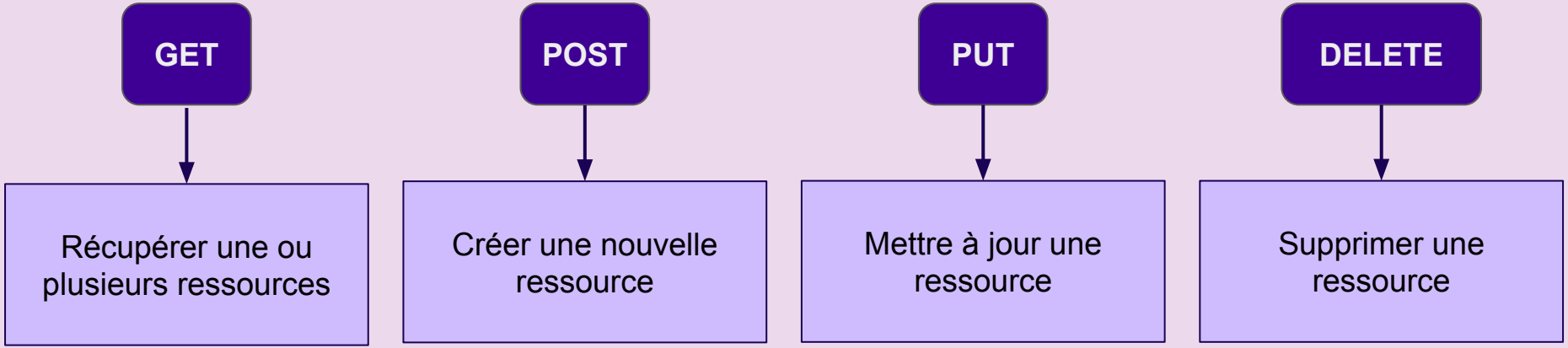
💡 TLD = top-level domain
eTLD = effective top level domain

What the hell is HTTP(s)?

https://

→ **HTTP** (HyperText Transfer Protocol) est un **protocole** de **communication** utilisés par les clients et les serveurs pour s'**échanger** des **données** sur le web.

→ **HTTP** admet plusieurs **méthodes** qui définissent plusieurs **actions**:



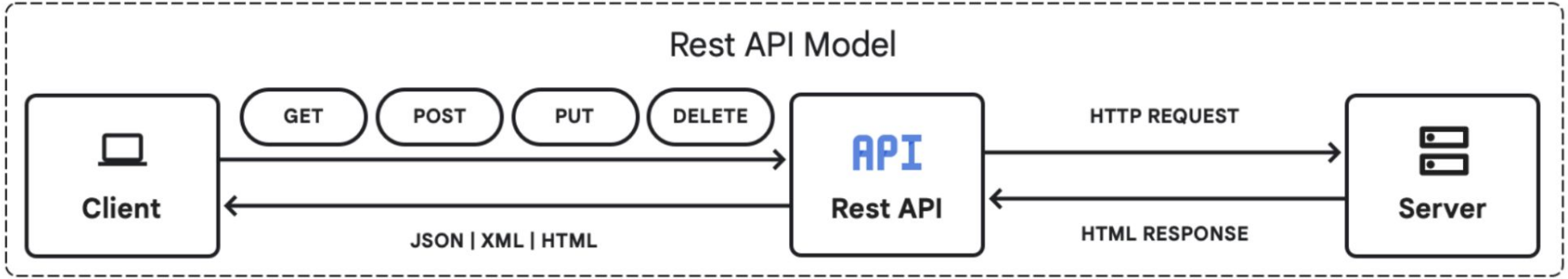
Qu’est ce qu’une API “REST” ?

C’est simple !!

REST = REpresentational State Transfer

Voilà !

Sérieusement: la REST API



REST = REpresentational State Transfer

- **Reprensentational** car elle représente les données sous une représentation (JSON, HTML, xml)
- **State** ou plutôt **statelessness**: l'API est sans état. Chaque requête du client au serveur doit **contenir toutes les infos** nécessaires pour comprendre et répondre. Le serveur n**e sauvegarde aucune info** du client. Par conséquent, **n'importe quel serveur** contenant la REST API peut répondre à n'importe quelle requête. Donc **Scalable**.
- **Transfer**: bah c'est une API donc elle transfère des données

Comment utiliser une API concrètement ?

Prenons l'exemple d'une API ML qui estime les notes des élèves de TRIED:

domain_name: https://tried_notes_estimation

Un endpoint POST: https://tried_notes_estimation/predict

Cet endpoint prend en entrée un JSON :

```
{
  "nombre d'heures de travail": 37,
  "concentration": "moyenne"
}
```

L'API répond :

```
{
  "prediction": "passable",
  "score": 14
}
```

Comment utiliser une API concrètement ?

Outils pour envoyer des requêtes à cette API:

1. **cURL**(client URL) = command line tool pour transférer des données du et au serveur.

```
curl -X POST https://tried notes estimation/predict
  -H "Content-Type: application/json" \
  -d '{"nombre d'heures de travail": 37,"concentration": "moyenne"}'
```

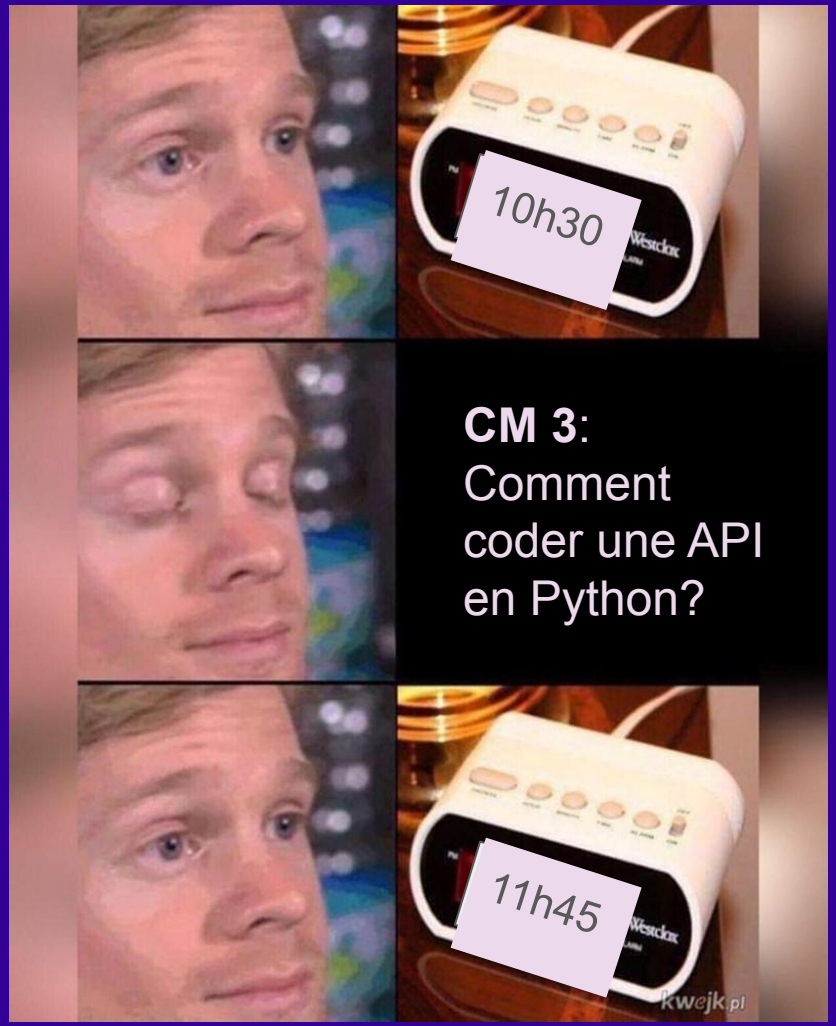
2. Lib **requests** de Python

```
import requests
response = requests.post(
    "https://tried notes estimation/predict",
    json="nombre d'heures de travail": 37,"concentration": "moyenne"}
)
```

3. Avec POSTMAN, friendly UI pour une API

Coffee Time ☕☕☕☕☕☕☕☕☕☕☕☕☕☕☕

Ceci n'est pas un coffeeshop



Part 3 of 10

Comment coder une API REST en Python?

Python admet deux standards qui définissent comment les apps Python interagissent avec le web.

WSGI

- Synchrone i.e une seule requête par thread
- HTTP
- serveurs: gunicorn

ASGI

- **Asynchrone** (concurence de requête)
- HTTP et websocket
- serveurs: uvicorn, hypercorn

Comment coder une API REST en Python?

Les frameworks/libs les plus populaires:

Caractéristique	django	Flask	FastAPI
Philosophie	"Batteries incluses" (Complet)	"Micro" (Minimaliste)	"Performance & Type-Safe"
Interfaces	WSGI	WSGI	ASGI
Cas d'usage Idéal	SaaS complexe, E-commerce, CMS	Micro-apps, prototypes rapides	APIs haute performance, IA/ML
Performance	Standard (Synchrone)	Rapide (car léger)	**Exceptionnelle** (Asynchrone)
Sécurité	Native	À configurer soi-même	Haute (via Pydantic/Type Hints)
Vitesse de dév.	Très rapide (pour le complexe)	Très rapide (pour le simple)	Maximale (documentation auto)
Position Marché	Standard Entreprise / Stable	Favori des Startups / Outils internes	Leader Moderne / Standard Microservices

On va utiliser FastAPI ajd !!

Caractéristique	Flask	FastAPI
		"Performance & Type-Safe"
		APIs haute performance, IA/ML
		Exceptionnelle (Asynchrone)
		Haute (via Pydantic/Type Hints)
		Maximale (Documentation auto)
		Leader Moderne / Standard Microservices

Architecture FastAPI

- Construit sur **Starlette** (un web framework pour Python) en **ASGI**
- Les endpoints d'une API sont appelés **route**

```
# main.py
from fastapi import FastAPI
app = FastAPI()

@app.get("/my_route") ## This is an endpoint
async def read_root():
    return {"message": "Hello, World from FastAPI!"}
```

- Ensuite, dans le terminal:

```
> fastapi dev main.py
INFO:     Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
```

- Interactive Docs generated automatically : go to: http://127.0.0.1:8000/docs.
- Alternative docs: `http://127.0.0.1:8000/redoc`.

Architecture FastAPI : path et query parameters

```
@app.get("/users/{user_id}")
def get_user(user_id: int, active: bool = True):
    return {"id": user_id, "active": active}
```

ici:

- `user_id` : **path parameter,** donc à mettre dans le path du endpoint
  → Un path parameter est utilisé pour identifier la ressource
- `active` : **query parameter,** donc paramètre passé à la fonction du endpoint
  → Un query parameter est utilisé pour filtrer cette ressource

Architecture FastAPI : Pydantic models

Pydantic est une librairie Python qui permet de :

- définir des structures de données typées
- valider automatiquement les données

Pourquoi utiliser Pydantic ?

Prenons cette fonction comme exemple:

```
def create_user(user_data):
    name = user_data["name"]
    age = int(user_data["age"])
    if age < 0:
        raise ValueError("Invalid age")
```

Éventuel risques lors de l'utilisation de cette fonction:

- clé (dict key) **manquante** (name, age)
- **mauvais type** (age passé en str au lieu de int)

Architecture FastAPI : Pydantic models

Pydantic permet de valider les args de la fonction en amont en créant des modèles de données avec leurs types et des règles pour les données (validators).

Exemple:

```
from pydantic import BaseModel
class User(BaseModel):
    name: str
    age: int
```

Usage:

```
data = {"name": "Alice", "age": "25"}
user = User(**data)
print(user.name)  # Alice
print(user.age)   # 25 (cast automatique)
```

Error Handling:

```
User(name="Alice", age="abc")
ValidationError:
age
    Input should be a valid integer
```

Architecture FastAPI : Sync vs Async

Synchrone (sync)

- Les tâches s'exécutent **les unes après les autres**
- Le programme **attend** qu'une tâche se termine avant de continuer : **programme bloquant.**

```
import time
def tache():
    time.sleep(3)
    return "terminé"
tache()
tache()
```

⏱️ **Temps total** : 3 s + 3 s = 6 secondes

👉 **Chaque tâche bloque l'exécution**

Asynchrone (async)

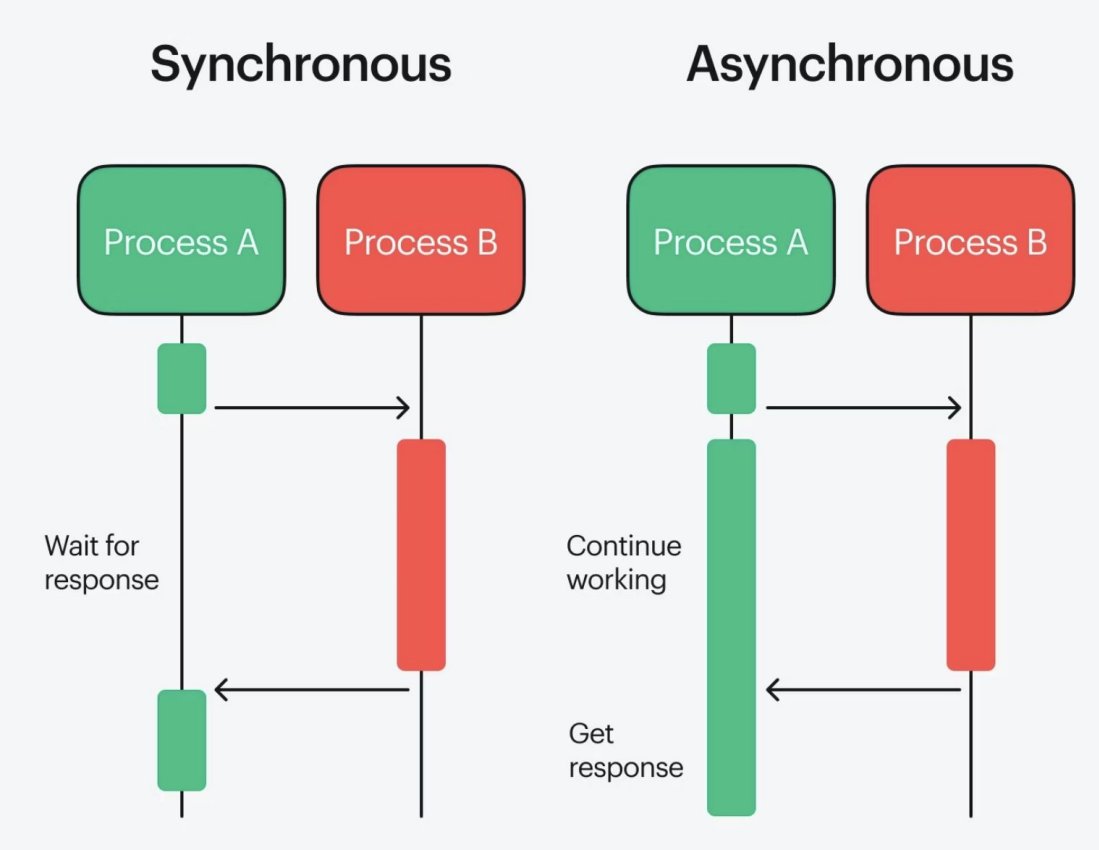
- Les tâches peuvent se **chevaucher** dans le temps
- Pendant l'attente, le programme peut faire autre chose : **programme non bloquant**

```
import asyncio
async def tache():
        await asyncio.sleep(3)
        return "terminé"
async def main():
      await asyncio.gather(tache(), tache())
asyncio.run(main())
```

⏱️ Temps total : **3 secondes**

👉 Les tâches attendent en même temps

Architecture FastAPI : Sync vs Async



Architecture FastAPI : Quand utiliser l’asynchrone ?

L’asynchrone est idéal pour des tâches **I/O bound** :

- Appels API externes
- Requêtes base de données
- Lecture / écriture réseau
- Accès disque

❌ Pas utile pour :

- Calculs lourds (CPU bound)
- Entraînement de modèles ML

👉 Si tu fais de l’I/O → async
👉 Si tu fais du calcul → sync

Exemples :

- Appel à une API → async ✅
- Requête DB → async ✅
- Prédiction ML lourde (predict d’un modèle ML) → sync

Demo Time

Part 4 of 10

12h15
Westclox
TP 2 :
Entraîner un
modèle ML
“dummy”
12h30
Westclox
kwejk.pl

Part 5 of 10

Luch time

Ceci n'est pas un destructeur de buffet

Part 6 of 10

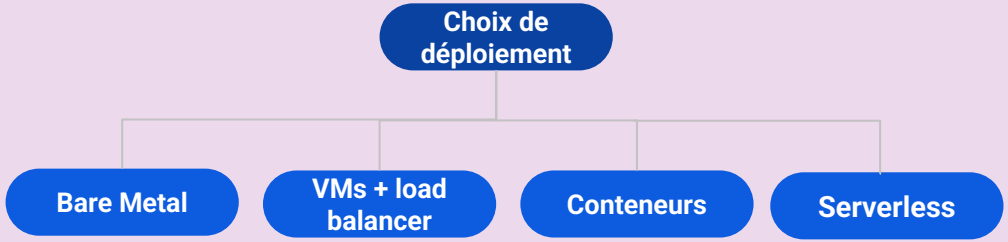
15h
TP 4 : coder une FastAPI pour servir le modèle dummy
15h45
kwejk.pl

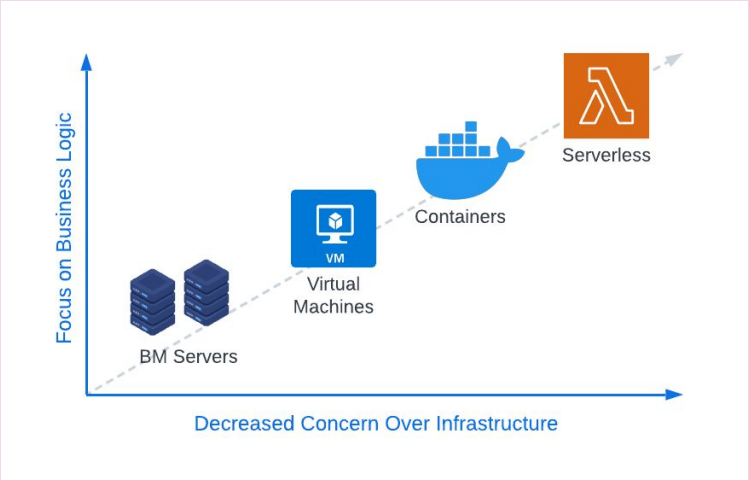
Part 7 of 10

Part 8 of 10

Model Deployment

Rappel: **Model deployment** c'est mettre le modèle dans un **environnement de prod**, donc si on a créé une **API** qui sert le modèle, ça revient à déployer cette **API + modèle** sur une infrastructure de prod.





ATTENTION : slides suivantes chargées

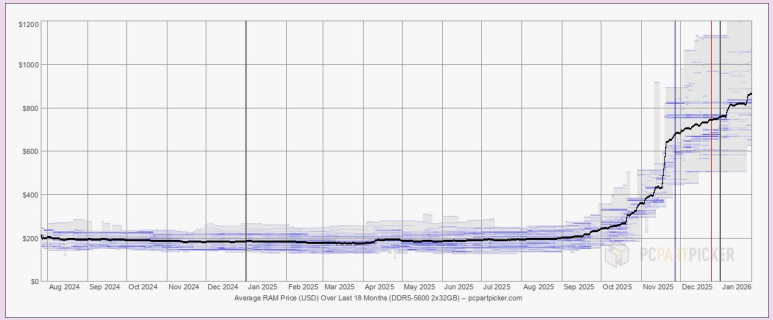
Model Deployment : Bare Metal

Le déploiement **bare metal** consiste à exécuter le modèle et l'API **directement sur des serveurs physiques**, sans couche de virtualisation (pas de VM).

✅ Avantages

- Performances maximales (pas de surcoût de virtualisation).
- Accès direct au matériel (GPU, CPU, réseau)
- Latence minimale et performances prévisibles
- **Sécurité maximale** si on achète nos propres serveurs.

⚠️ Inconvénients



- Gestion de l'infrastructure plus complexe
- Scalabilité plus lente (le temps de louer une nouvelle BM)

Model Deployment: Virtual Machine + Load Balancer

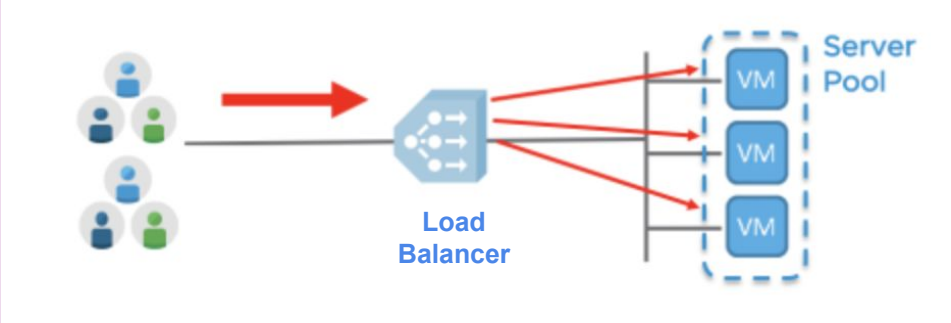
On loue plusieurs **machines virtuelles** (dans le cloud ou sur des serveurs physiques) pour héberger le modèle et l'API de prédiction.
Un **load balancer** en amont assure la répartition du trafic, la tolérance aux pannes et la montée en charge.

✅ Avantages

- **Bonne isolation**
- **Facilité de gestion par rapport au bare metal**
- **Compatible avec la plupart des stacks**

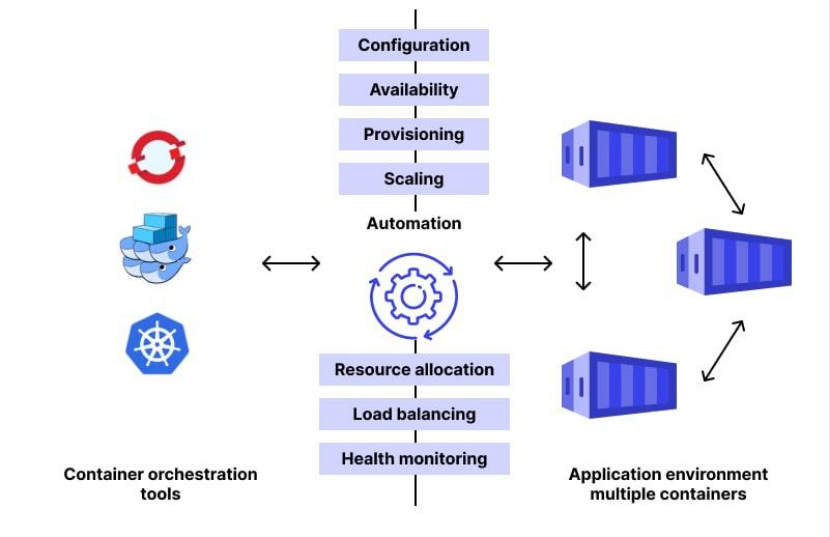
⚠️ Inconvénients

- **Démarrage lent**
- **Moins flexible que les conteneurs**



Model Deployment: conteneurs + orchestrateur

→ Le code de l'API et du modèle est encapsulé dans un **conteneur Docker,** garantissant un environnement **reproductible**.
→ Le conteneur est ensuite déployé sur une **VM** ou un **serveur**. Un même serveur peut héberger plusieurs conteneurs.
→ Les conteneurs sont **orchestrés** (ex. **Kubernetes**) et automatiquement répartis sur plusieurs VMs en fonction des ressources disponibles.



✅ Avantages

- **Léger et rapide à démarrer**
- **Très bonne portabilité**
- **Excellente utilisation des ressources**
- **Standard en production**

⚠️ Inconvénients

- **Isolation plus faible que les VMs**

Model Deployment: Serverless

→ En serverless, le développeur **ne gère pas les serveurs** (VMs, conteneurs, scaling).
→ Le code (ex. une API ou une fonction de prédiction) est déployé sous forme de fonctions exécutées à la demande par le **cloud provider.**
→ Le provider se charge automatiquement de :

- l'allocation des ressources,
- l'autoscaling,
- haute disponibilité

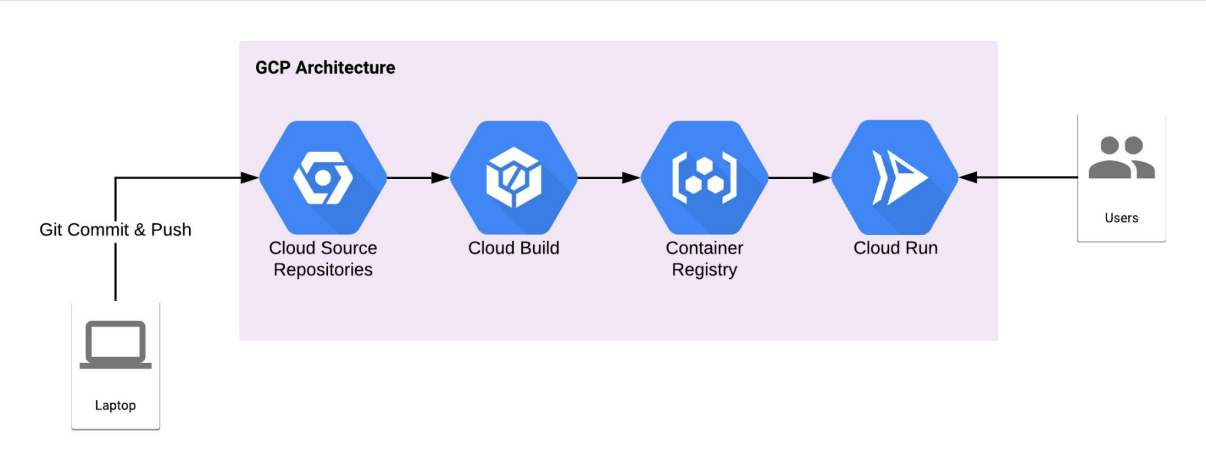
👉 On paie uniquement à l'usage **"pay-as-used basis".**

✅ Avantages

- **Pas d'infrastructure à gérer**
- **Scaling automatique**
- **Paiement à l'usage**

⚠️ Inconvénients

- **Peu adapté aux modèles lourds**



Model Deployment Strategies

Strategies

Single

Échanger l'ancien modèle avec le nouveau (brute force)

Silent

Préparer toute une nouvelle infra (p. ex. API + modèle), la mettre en prod, recevoir les requêtes, mais ne pas renvoyer tant qu'on n'a pas 100 %validé. Ensuite, on switch.
C'est une stratégie qui demande beaucoup plus de ressources

Canary

On déploie la nouvelle API et on switch les users progressivement de l'ancienne:

- 1er jour: ancienne 95%, nouvelle 5%
- 2e jour: 80%, 20%
- etc

Multi-arm Bandits

Les deux versions en prod, on a une phase d'exploration (on envoie les requêtes aux deux) et, en fonction des métriques, le MAB va switcher le traffic vers le modèle le plus performant.
Ce qui résout autant le déploiement que le online testing.

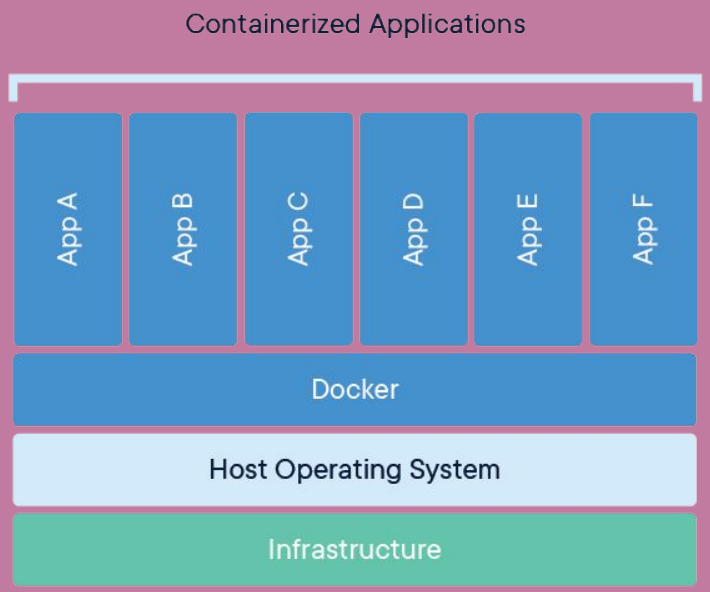
Qu’est ce que Docker ?

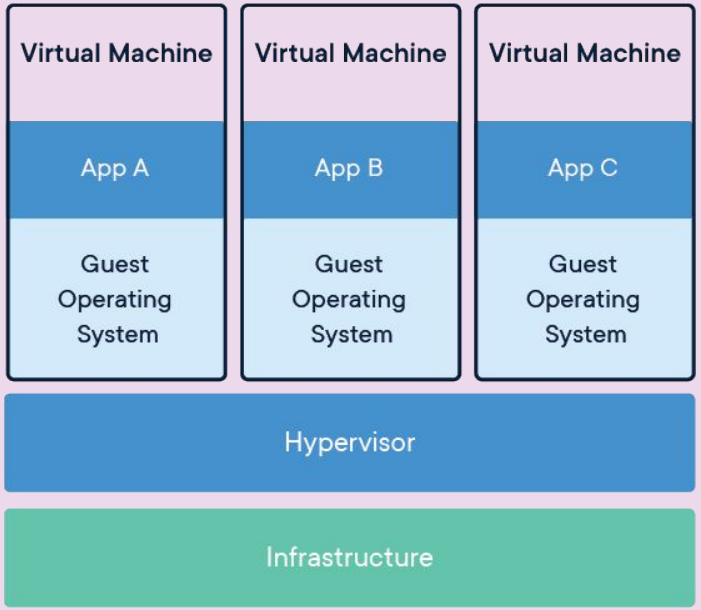
Docker est un service de conteneurisation des applications et des dépendances dans une entité “container” pour les rendre **reproductibles** sur n’importe quelle machine ou environnement.

Concepts clés :

- 🧱**image Docker** : est la recette du conteneur (l'OS, le code de l’appli, les deps, les fichiers configs, les commandes pour run l'app.).
- 📦 **Container**: instance vivante qui run l’image. On peut **run** plusieurs conteneurs d’une même image.
- 📄**Dockerfile :** fichier pour construire l’image docker
- 🗂️ **Registry :** endroit pour sauvegarder les images Docker.

Container vs VM





Conteneur	VM
Partage le kernel	OS complet
Léger	Lourd
Démarrage instantané	Lent à démarrer

Part 9 of 10

Part 10 of

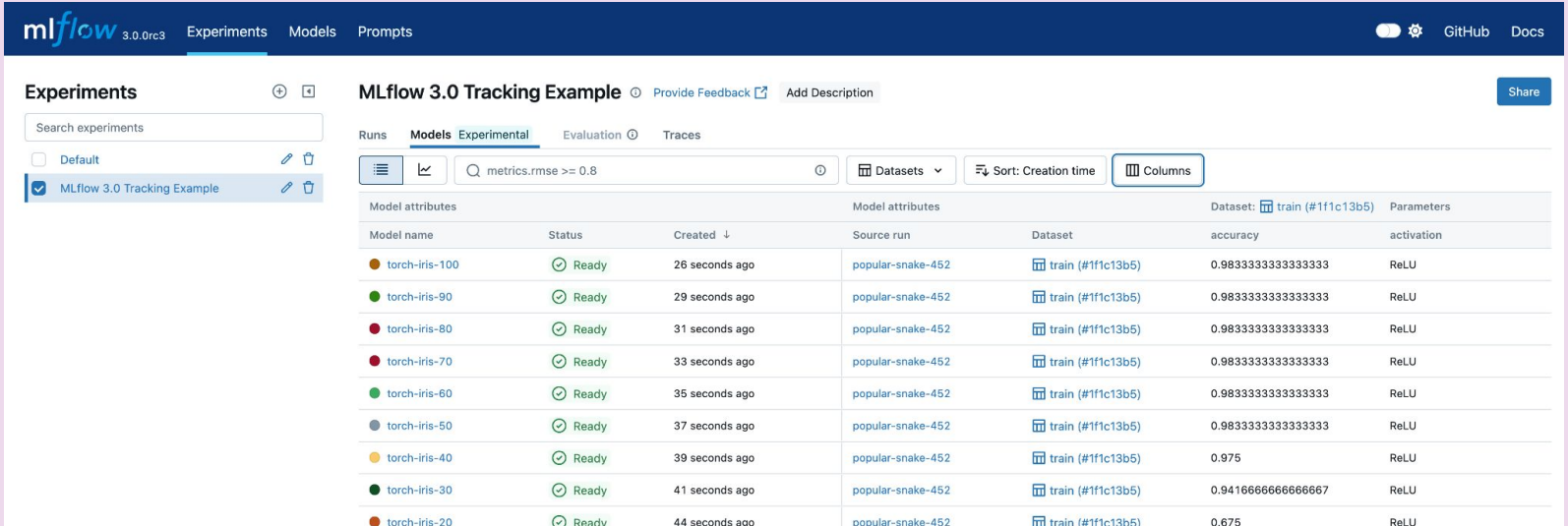
Introduction à MLflow

Overview:

MLflow est une plateforme open-source qui sert à sauvegarder des informations clés sur les modèles ML. Elle peut être utilisée pour :

- **Tracking** des modèles
- **Registry** de modèles (bibliothèque de modèles disponibles)
- Outil de **déploiement** de modèles

1 projet = 1 experiment **1 modèle = 1 run**



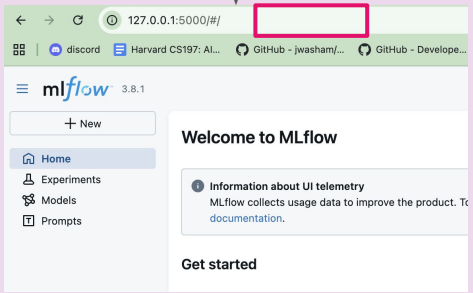
Comment setup MLflow?

- Self Hosting
  - SQLITE DATABASE
  - File System
  - Remote Tracking Server
- Managed hosting with Databricks

```python
import mlflow

mlflow.set_tracking_uri("sqlite:///mlflow.db")
mlflow.set_experiment("my-first-experiment")
```

```python
import mlflow

# Creates local mlruns directory for experiments
mlflow.set_experiment("my-first-experiment")
```

```python
import mlflow

# Connect to remote MLflow server
mlflow.set_tracking_uri("http://localhost:5000")
mlflow.set_experiment("my-first-experiment")
```



💡 **En entreprise, MLflow est déployé comme un service centralisé.**
On provisionne une **VM** chez un cloud provider, sur laquelle tourne le **serveur MLflow** en continu.

1. MLflow as a Tracking Tool

- **C’est l’usage le plus classique de MLflow**
- **Que logger dans les runs?**
  - **l'artéfact** du modèle (i.e. le pickle du modèle avec les weights)
  - les **métriques** du modèle (accuracy, précision, RMSE, etc)
  - Les **paramètres** du modèle
  - le **dataset** d’entrainement
  - (des **figures**, des fichiers, ..)

- On log ces éléments depuis le code soit de façon **manuelle** soit en mode **autologging** si le framework est compatible avec MLflow (Supported libraries: Scikit-learn, XGBoost, LightGBM, PyTorch, Keras/TensorFlow, Spark, and more)

1. MLflow as a Tracking Tool (demo)

mlflow 3.8.1 GitHub Docs

iris_classification_experiment › Runs ›

RandomForest_Iris

Overview | Model metrics | System metrics | Traces | Artifacts

Description

Mon modele ML pour TRIED

Metrics (4)

Search metrics

Metric	Value	Models
accuracy	0.9	model
precision	0.9023569023569024	model
recall	0.9	model
f1_score	0.8997493734335839	model

Parameters (6)

Search parameters

Parameter	Value
model_type	RandomForestClassifier
n_estimators	100
max_depth	None

About this run

Created at	01/23/2026, 06:40:20 PM
Created by	carole
Experiment ID	1
Status	Finished
Run ID	36d9ce7e86ca46ddb388cd2743dd1588
Duration	3.7s
Source	ipykernel_launcher.py
Registered prompts	—

Datasets

None

Tags

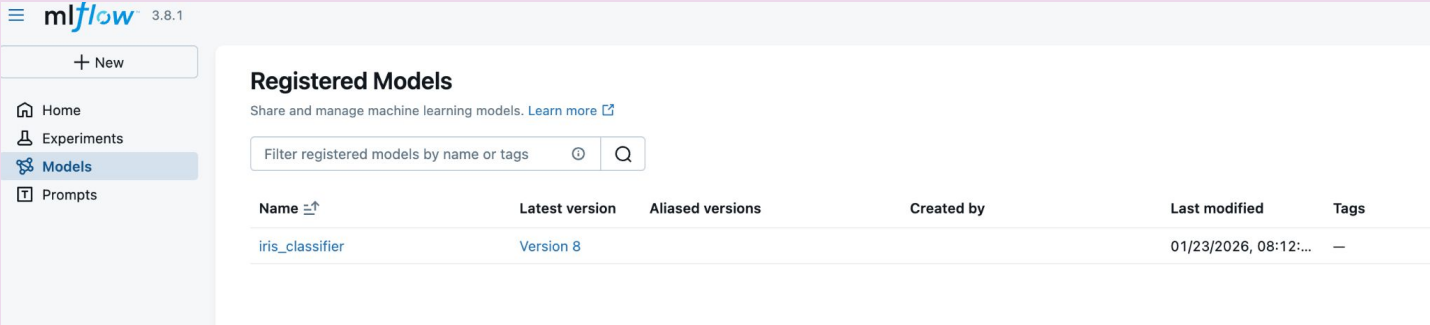
model_family: random_forest dataset: iris

Registered models

iris_classifier v6 +1

2. Mlflow as a Model Registry Tool

→ **MLflow** permet de **sauvegarder** les modèles ML sur le serveur pour les **reloader** quand on veut dans **l'onglet Models.**



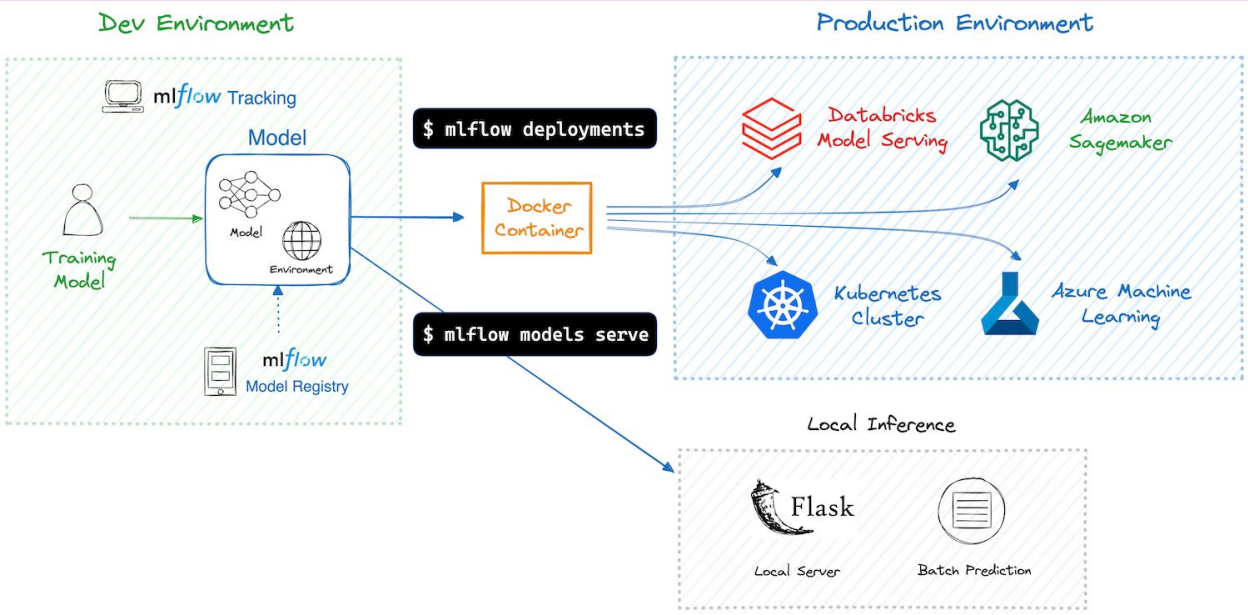

```python
model_name = "iris_classifier"
model_version = "latest"

# Load the model from the Model Registry
model_uri = f"models:/{model_name}/{model_version}"
model = mlflow.sklearn.load_model(model_uri)

# Generate a new dataset for prediction and predict
X_new, _ = make_regression(n_features=4, n_informative=2, random_state=0, shuffle=False)
y_pred_new = model.predict(X_new)
```

3. MLflow as a deployment tool:

MLflow a des intégrations qui permettent de déployer/serve les modèles dans son registry directement sans coder une API.

Plusieurs options:

- **Déployer en local**
- **Déployer sur Kubernetes**
- **Déployer sur Amazon SageMaker**



3. MLflow as a deployment tool : déployer en local

```
> mlflow models serve -m runs:/<run_id>/model -p 5000

> curl http://127.0.0.1:5000/invocations -H "Content-Type:application/json"

--data '{"inputs": [[1, 2], [3, 4], [5, 6]]}'
```

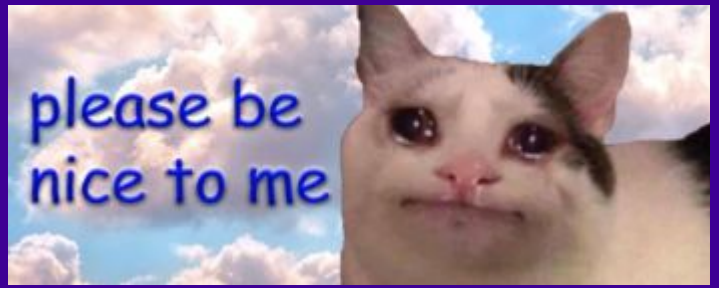
Qu'est ce qui se passe sous le capot ?

- MLflow crée une FastAPI avec une route /invocations qui est /predict
- D'autres endpoints sont également créer
  - **/invocations:** POST requests with input data and returns predictions.
  - **/ping:** Used for health checks.
  - **/health:** Same as /ping
  - **/version:** Returns the MLflow version.

Défauts de déployer via MLflow

- Que en pip ou conda
- Les endpoints sont définis pour nous, donc moins de fonctionnalités
- Compatible pour des APIs simples

Feedback sur l'intervention sur ce Google Form :



SCAN ME

form

Ressources