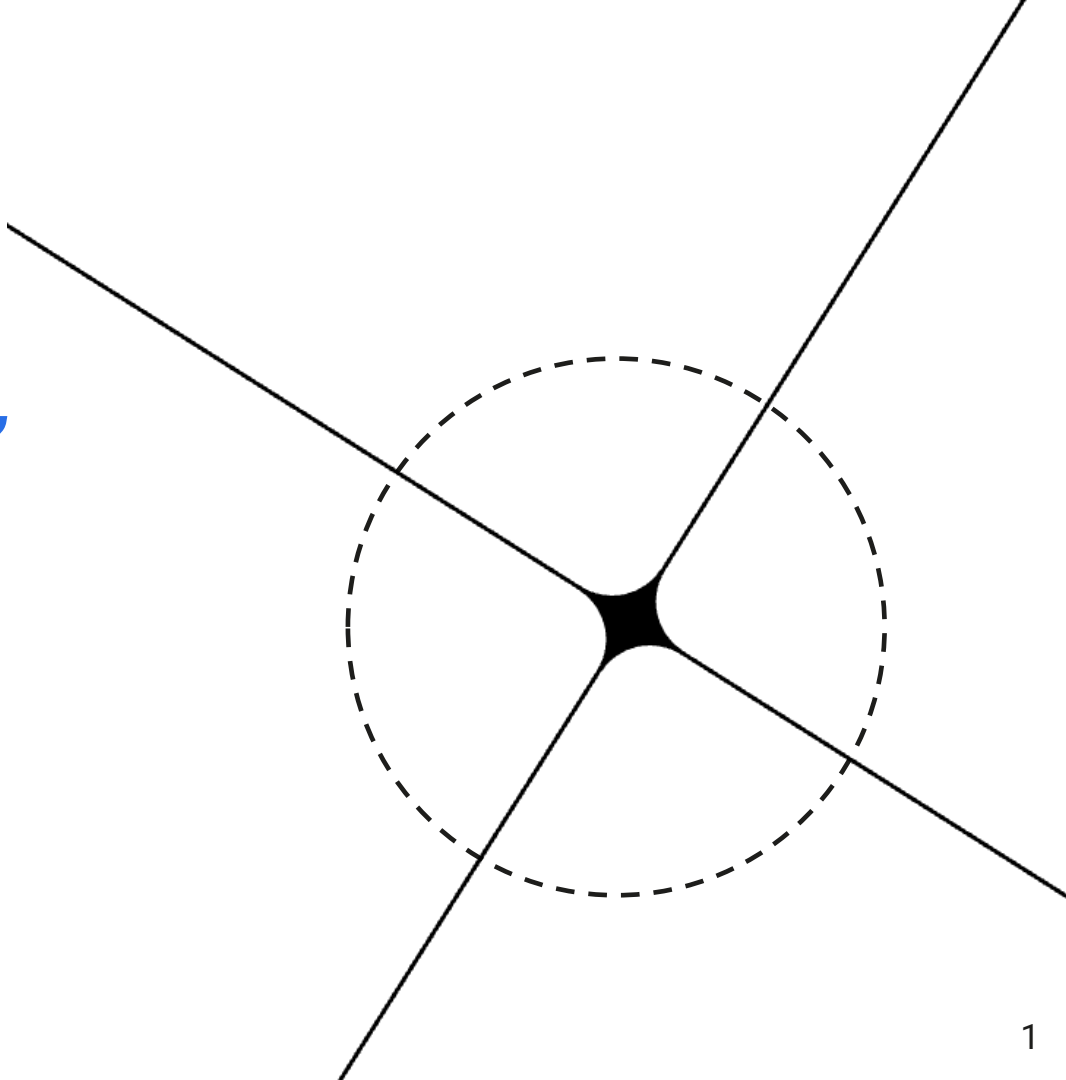
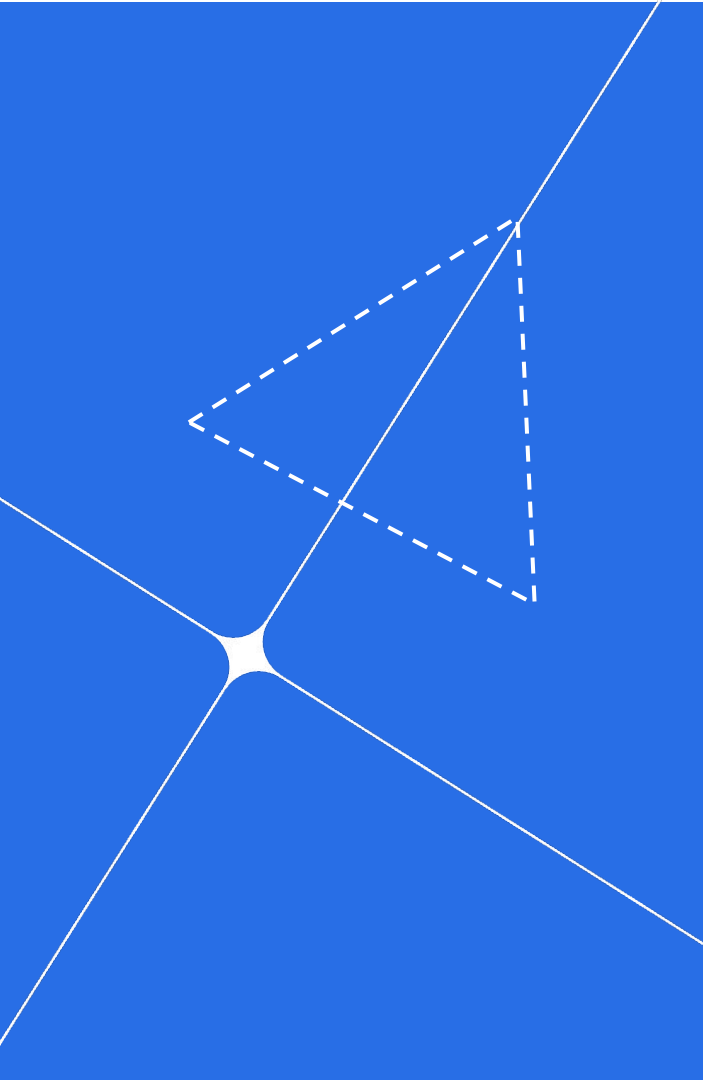
INSTITUT POLYTECHNIQUE DE PARIS

TELECOM SudParis

Entraînement d'un modèle, Tracking avec MLflox, Model Registry et déploiement avec une API

Julien Romero





Introduction et contexte

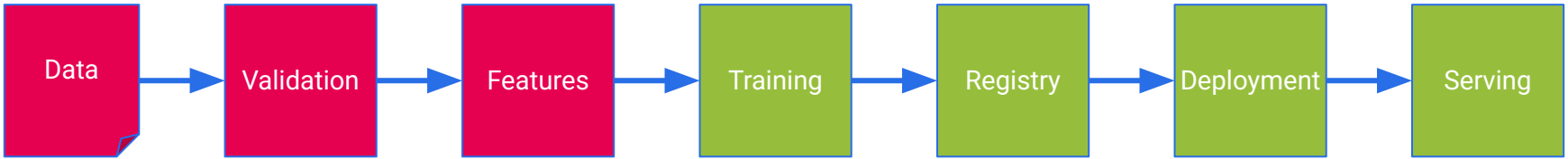
Objectifs du cours

- Comprendre le rôle du training dans un système ML en production
- Introduire MLflow comme colonne vertébrale du cycle modèle
- Relier :
  - Les features à l'entraînement
  - L'entraînement au registry
  - Le registry à l'API
- Comprendre la logique staging / production
- Poser les bases du serving industriel (API + features en ligne)

Pourquoi ça casse ici, en pratique

- Zone de fragilité majeure des systèmes ML
  - Beaucoup de modèles fonctionnent offline
  - Peu fonctionnent correctement en production
- Raisons fréquentes d'échec
  - Training différent du serving (skew)
  - Modèle non traçable :
    - impossible de savoir ce qui est en production
  - Évaluations non comparables (différents jeux de tests, différentes métriques)
  - Entraînement non reproductible
  - Déploiement manuel, non contrôlé
  - Absence de rollback
- **Le problème n'est pas l'algorithme : le problème est le système autour du modèle**

Rappel : pipeline MLOps complet



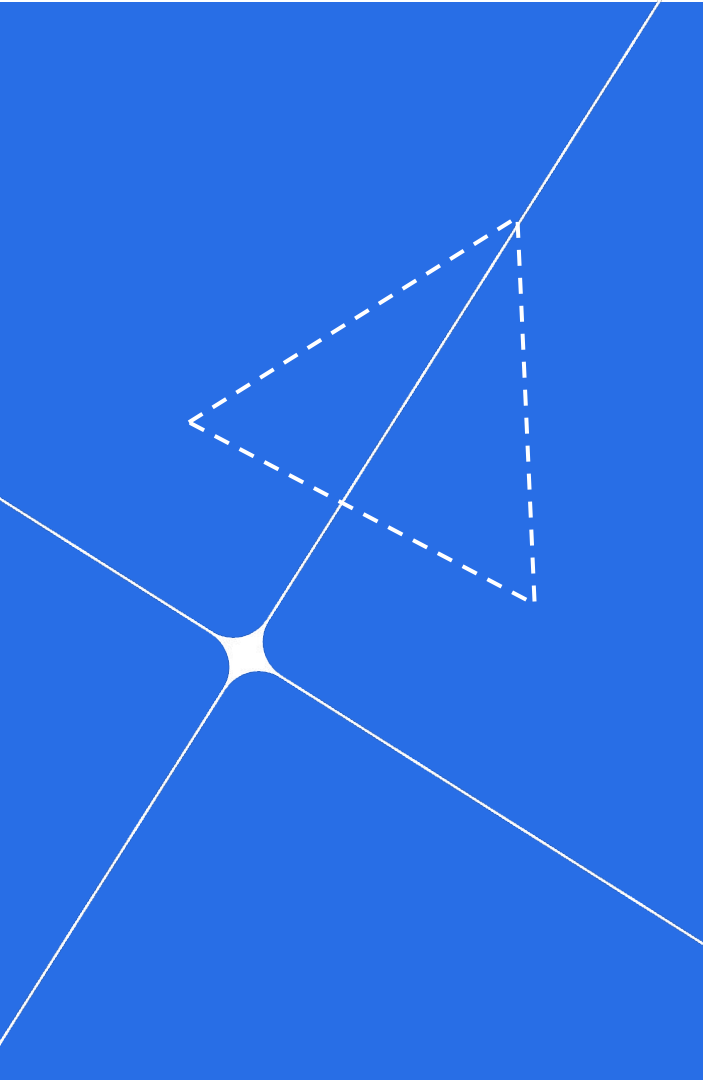
- Ingestion & validation : vues précédentes
- Feature Store : cohérence training / inference
- **Training** :
  - point de jonction data/modèle
- **Registry** :
  - mémoire centrale des modèles
- **Serving** :
  - exposition contrôlée en production
- **Focus du cours** : Du dataset de features au modèle en production

Où on en est, où on va

- Ce que vous avez déjà construit
  - Pipelines de données déterministes
  - Snapshots temporels
  - Feature Store (offline / online)
  - Alignement training / inference
- Ce cours
  - Entraîner un modèle comme un composant système
  - Suivre et comparer les expériences
  - Gérer des versions de modèles
  - Servir un modèle via une API
- Ce qui arrive ensuite (Cours 5 et 6)
  - Monitoring en production
  - Drift (données & performance)
  - Réentraînement automatisé
  - Promotion automatique des modèles

“Production-ready training”, ça veut dire quoi ?

- **Ce que ce n’est pas**
  - model.fit(X, y)
  - Un notebook isolé
  - Un fichier .pkl sur disque
  - Une métrique affichée une fois
- **Ce que c’est**
  - Pipeline déterministe
  - Données versionnées implicitement
  - Évaluations comparables dans le temps
  - Modèle :
    - traçable
    - versionné
    - déployable automatiquement
  - Séparation claire :
    - entraînement
    - validation
    - serving
- Entraîner un modèle = **produire un artefact industriel**



Des features aux modèles : L'entraînement comme une composante du système

Training ≠ Notebook

- **Idée clé**
  - En production, l'entraînement n'est pas :
    - un notebook exploratoire
    - une cellule exécutée une fois
  - C'est un composant système, au même titre que :
    - ingestion
    - API
    - monitoring
- **Conséquences**
  - Code exécutable sans interaction humaine
  - Entrées et sorties clairement définies
  - Exécution répétable
  - Observable (logs, métriques)
- **Le training est une pipeline**, pas une expérience ponctuelle

Entrées d'une pipeline d'entraînement

- Inputs explicites
  - **Features**
    - issues du Feature Store
    - offline, temporellement correctes
  - **Labels**
    - définis pour une période donnée
    - alignés avec les features
  - **Configuration**
    - hyperparamètres
    - métriques
    - seuils
  - **Code**
    - logique de training
    - preprocessing éventuel
    - Logique d'évaluation
- **Principe MLOps** : Tout ce qui influence le modèle doit être une entrée traçable

Pipelines déterministes : définition

- **Définition**
  - Même inputs donne même outputs
  - À l'identique, aujourd'hui ou dans 6 mois
- **Outputs concernés**
  - Métriques
  - Modèle entraîné
  - Artefacts (courbes, coefficients, signatures)
- **Pourquoi c'est critique**
  - Comparer des modèles dans le temps
  - Debugger une régression
  - Reproduire un modèle en production
  - Autoriser l'automatisation (CI/CD)
- Sans déterminisme, pas de gouvernance possible

Ce qui casse le déterminisme

- Sources classiques
  - **Aléatoire**
    - seeds non fixées
    - split aléatoire non contrôlé
  - **Temps**
    - now(), today()
    - dépendance à l'horloge système
  - **Data leakage**
    - features calculées après le label
    - Corrections a posteriori
  - **Code drift**
    - modification silencieuse du code
    - Dépendances non figées
- Symptôme typique : "Je ne retrouve pas les résultats d'hier"

Rappel crucial : alignement temporel

- **Principe :** Le modèle doit apprendre avec uniquement l'information disponible au moment de la prédiction réelle
- **Alignement requis**
  - Label à date t
  - Features calculées avant ou à t
  - Pas d'accès au futur
- **Pourquoi c'est vital**
  - Éviter des performances artificiellement élevées
  - Garantir une évaluation réaliste
  - Assurer cohérence training / inference

Stratégies de split train / validation

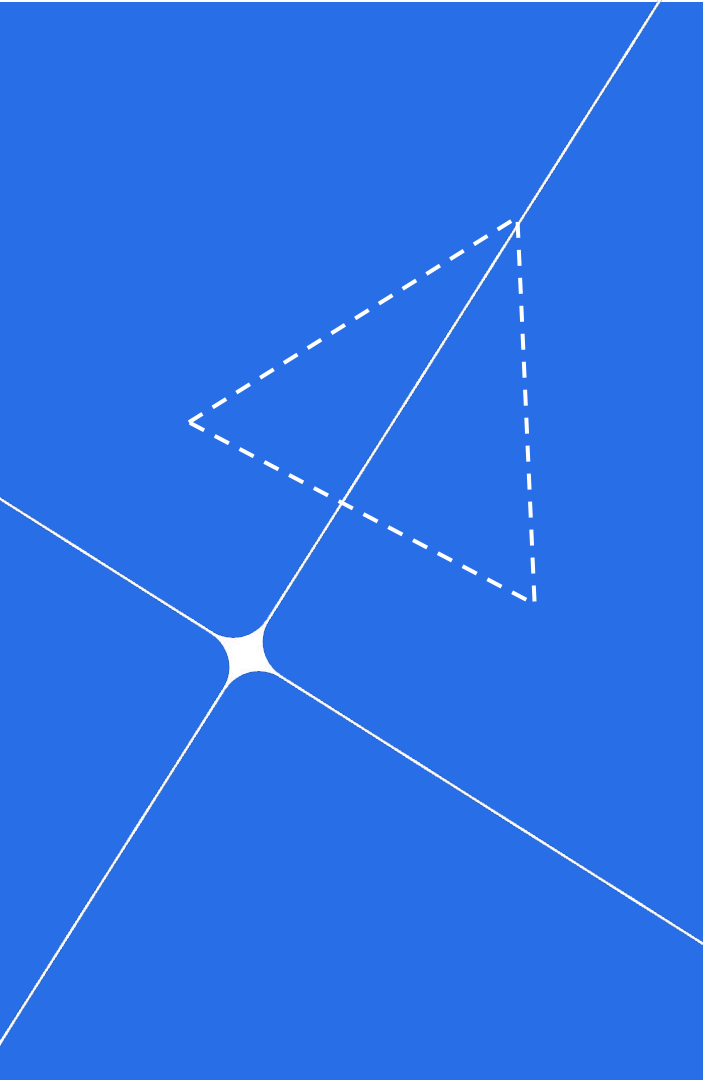
- **Split par entité (ex: user_id)**
  - Séparation par individus
  - Utile si :
    - données indépendantes
    - pas de dépendance temporelle forte
- **Split temporel**
  - Train sur le passé
  - Validation sur le futur
  - Recommandé pour :
    - churn
    - finance
    - séries temporelles
- **Règle pratique**
  - Si le temps a du sens, split temporel
  - Sinon, split par entité

Pipeline d'entraînement minimal

- Étapes haut niveau
  1. Charger labels pour une période donnée
  2. Construire un dataframe ids + timestamps + labels
  3. Récupérer features historiques dans le feature store
  4. Vérifier le dataset (taille, valeurs)
  5. Split train / validation
  6. Entraîner un modèle
  7. Évaluer avec métriques choisies
  8. Produire artefacts et métriques
- Pas d'optimisation prématurée : Pipeline clair > modèle complexe

Checkpoint #1

- À ce stade, vous devez pouvoir :
  - Expliquer pourquoi le training est une pipeline
  - Identifier toutes les entrées qui influencent un modèle
  - Définir ce qu’est un pipeline déterministe
  - Lister les causes principales de non-reproductibilité
  - Justifier une stratégie de split adaptée au problème
- Si une étape n’est pas claire ici, elle cassera tout le système ensuite



Pourquoi le tracking d’expériences est obligatoire en production ?

Question centrale en production

- Motivation
  - “Quel modèle est en production… et pourquoi celui-là ?”
  - “Quelle version du code ? Quelles features ? Quel dataset ?”
  - “Quelle performance attendue ? Sur quelles métriques ?”
  - “Si ça casse : on rollback vers quoi ?”
- Sans tracking :
  - impossible de répondre factuellement
  - débats “à l’intuition”
  - pertes de temps énormes en incident
- Le tracking = mémoire et preuve du système ML

Ce qu'il faut traquer (minimum viable)

- **Paramètres (inputs du training)**
  - hyperparamètres
  - paramètres de pipeline (fenêtres temporelles, split)
  - seeds
  - version de la config
- **Métriques (outputs mesurables)**
  - AUC / logloss / F1 / calibration…
  - métriques train vs validation
  - éventuellement par segment (plan_type, pays…)
- **Artefacts**
  - modèle sérialisé
  - courbe ROC, matrice de confusion
  - feature importance / coefficients
  - échantillon de prédictions
- **Contexte de code**
  - version du code (commit hash)
  - environnement (versions libs)

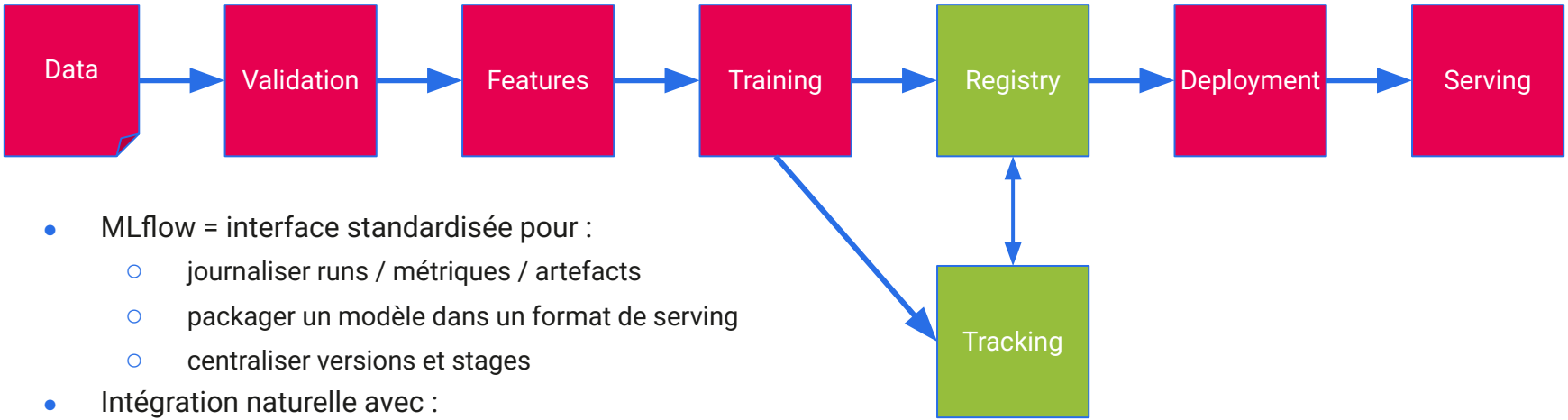
Pourquoi “des logs + des fichiers” ne suffisent pas

- **Approche naïve**
  - results_run_12.json
  - model_final_v3.pkl
  - logs console + screenshots
- **Problèmes**
  - pas de structure standard
  - impossible de comparer automatiquement des runs
  - perte d’info (qui a lancé, quand, avec quoi)
  - impossible de faire des recherches
  - pas de lien natif :
    - run → modèle → métriques → artefacts → code
- Ce n’est pas “audit-ready”
- Ce n’est pas “CI/CD-ready”

Experiment tracking vs Model registry

- **Experiment tracking**
  - historise des runs
  - but : exploration contrôlée, comparaison, reproductibilité
  - granularité : un entraînement = un run
  - objets : params, metrics, artifacts, tags
- **Model registry**
  - gère des versions de modèles
  - but : gouvernance + déploiement
  - granularité : un modèle = plusieurs versions
  - objets : version, stage, approbation, rollback
- Tracking = “comment ce modèle a été produit”
- Registry = “quel modèle doit être utilisé en production”

Position de MLflow dans la stack MLOps



- MLflow = interface standardisée pour :
  - journaliser runs / métriques / artefacts
  - packager un modèle dans un format de serving
  - centraliser versions et stages
- Intégration naturelle avec :
  - pipelines d'entraînement (scripts, orchestrateurs)
  - CI/CD (promotion conditionnelle)
- serving (charger "Production")

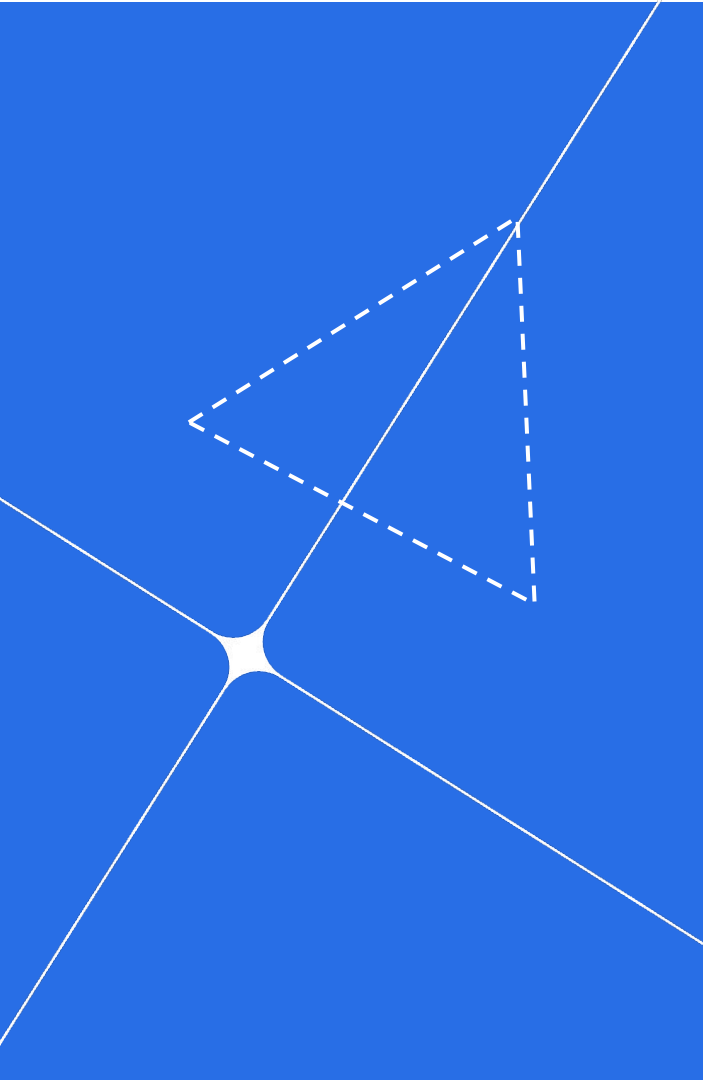
Ce que MLflow ne résout pas

- **MLflow n'est pas :**
  - un Feature Store (ne calcule pas les features)
  - un orchestrateur (ne planifie pas les jobs)
  - un outil de validation data (pas Great Expectations)
  - un système de monitoring de production (drift, latence, logs API)
  - une solution de data versioning complète
- En bref
  - MLflow gère le cycle du modèle
  - pas le cycle complet des données / infra
- Il faut une stack, pas un outil unique

Checkpoint #2

Vous devez être capables de :

- expliquer pourquoi la question “quel modèle et pourquoi” est critique
- lister ce qu’il faut tracer (paramètres / métriques / artéfacts / contexte)
- expliquer pourquoi fichiers + logs = insuffisant
- distinguer clairement :
  - tracking (runs)
  - registry (versions + stages)
- situer MLflow dans une architecture MLOps complète
- identifier les limites de MLflow



Concepts clés de MLflow

MLflow : vue d'ensemble

- MLflow repose sur trois piliers
  1. **Experiment Tracking**
     - runs
     - métriques
     - paramètres
     - artefacts
  2. **Model Packaging**
     - format standardisé
     - interface de prédiction
     - environnement associé
  3. **Model Registry**
     - versions
     - stages (Staging / Production)
     - gouvernance
- Objectif global : **structurer le cycle de vie du modèle**

Runs et Experiments

- **Experiment**
  - regroupement logique de runs
  - ex : “churn_model_v1”
  - permet comparaison et itération contrôlée
- Run
  - une exécution complète d’un training pipeline
  - correspond à :
    - un dataset
    - une config
    - une version de code
- **Propriétés clés d’un run**
  - horodaté
  - immuable après exécution
  - comparable aux autres runs
- Un run = une tentative mesurable et traçable

Paramètres, métriques, artefacts

- **Paramètres**
  - entrées du training
  - ex : hyperparamètres, fenêtres temporelles, seeds
  - valeurs scalaires ou catégorielles
- **Métriques**
  - sorties mesurés
  - ex : AUC, loss, F1
  - souvent suivies dans le temps
- **Artefacts**
  - fichiers associés au run
  - ex :
    - modèle
    - plots
    - feature importance
    - samples de prédictions
- Séparation claire : inputs / outputs / preuves

Métriques dans le temps

- Pourquoi l'historique est essentiel
  - comparer des modèles entre eux
  - détecter des régressions
  - suivre la progression d'un pipeline
- Comparaisons typiques
  - modèle A vs modèle B
  - nouvelle feature vs baseline
  - nouvelle période de données
- Signal faible mais critique
  - baisse progressive de métrique
  - variance inhabituelle
  - incohérence train / validation
- Tracking = observabilité du training

Pourquoi le modèle seul ne suffit pas

- Anti-pattern
  - “Voici le modèle, il a un AUC de 0.82”
- Ce qui manque
  - sur quelles données ?
  - avec quelles features ?
  - avec quels paramètres ?
  - comparé à quoi ?
  - stable dans le temps ?
- Bon principe
  - Un modèle sans contexte est inexploitable
  - Les artefacts racontent l’histoire du modèle
- Le modèle est une conséquence, pas l’objet principal

Format de modèle MLflow (pyfunc)

- Idée clé
  - Un modèle est exposé via une interface standard
  - Indépendante de :
    - la librairie (sklearn, xgboost, torch…)
    - le langage interne
- Interface conceptuelle
  - predict(input) => output
- Avantages
  - même logique de chargement en training et en API
  - découplage modèle/serving
  - interchangeabilité des implémentations
- Le modèle devient un composant logiciel

Signature de modèle : pourquoi c’est crucial

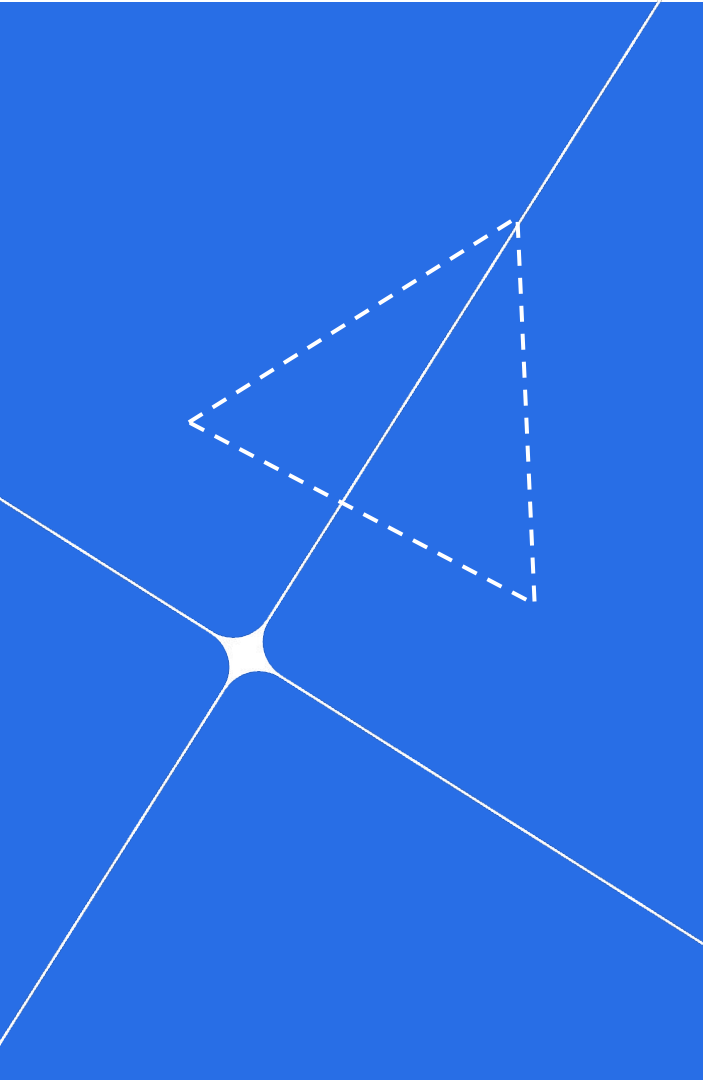
- Signature
  - schéma des entrées
  - schéma des sorties
- Rôles
  - validation automatique des inputs
  - détection d’incohérences au serving
  - documentation implicite du contrat modèle
- Sans signature
  - erreurs silencieuses
  - mismatch colonnes / types
  - bugs difficiles à diagnostiquer
- Une signature = un contrat d’API du modèle

Capture de l'environnement

- Problème classique
  - même modèle, comportement différent
  - dépendances non alignées
  - versions de librairies divergentes
- MLflow capture
  - dépendances Python
  - versions des librairies ML
  - parfois OS-level (partiel)
- Objectif
  - pouvoir relancer une prédiction :
    - aujourd'hui
    - ailleurs
    - Plus tard
- Reproductibilité n'est pas uniquement données + code

Versionnement de modèle vs tracking

- Tracking
  - compare des runs
  - répond à :
    - “qu’est-ce qui a marché ?”
    - “comment on en est arrivé là ?”
- Versionnement (Registry)
  - gère des modèles “officiels”
  - répond à :
    - “quel modèle est autorisé ?”
    - “lequel est en production ?”
- Relation
  - plusieurs runs peuvent donner un modèle versionné
  - tous les runs ne deviennent pas des versions
- Exploration n’est pas la production



Évaluation de modèle : ce qui compte en production

Checkpoint #3

- MLflow vous garantit :
  - traçabilité des runs
  - comparabilité des expériences
  - packaging standard des modèles
  - gestion des versions et des stages
- MLflow ne garantit pas :
  - qualité des données
  - absence de data leakage
  - pertinence des métriques
  - monitoring en production
  - décisions business correctes
- MLflow est un outil structurant, pas une solution magique

L'évaluation n'est pas "juste l'accuracy"

- Problème classique
  - Accuracy souvent utilisée par défaut
  - Facile à expliquer
  - Souvent trompeuse en production
- Pourquoi
  - Déséquilibre des classes
  - Décisions asymétriques (faux positifs et faux négatifs n'ont pas la même valeur)
  - L'accuracy ne reflète pas l'usage réel du score
- Une bonne métrique dépend du contexte d'utilisation

AUC : métrique centrale pour le churn

- Pourquoi l'AUC (Area Under the Curve, voir cours ML/DL) est pertinente
  - Indépendante du seuil
  - Mesure la capacité de ranking
  - Stable face aux déséquilibres modérés
- Interprétation
  - Probabilité qu'un utilisateur churn soit mieux scoré qu'un utilisateur non churn
  - Comparaison robuste entre modèles
- Usage typique
  - Classer les utilisateurs du plus à risque au moins à risque
  - Définir des actions sur le top-k%
- AUC = bonne métrique de sélection de modèle

Pièges de l’accuracy (cas déséquilibré)

- Exemple churn
  - 95% des utilisateurs ne churnent pas
  - Modèle trivial : “jamais churn”
  - Accuracy = 95%
  - Valeur business = 0
- Ce que l’accuracy cache
  - incapacité à détecter les cas rares
  - absence de signal utile pour l’action
- Accuracy élevée n’implique pas modèle utile

Métriques complémentaires

- Precision
  - Parmi les alertes, combien sont correctes
  - Coût des faux positifs
- Recall
  - Parmi les churns réels, combien sont détectés
  - Coût des faux négatifs
- F1
  - compromis precision / recall
- Calibration de modèle
  - fiabilité des probabilités : Vérifier que quand un modèle prédit qu'un événement à X% de chance de se passer, il se passe vraiment X% du temps
    - Ex : Si on prédit qu'une équipe A bat une équipe B 80% du temps, après 100 matches, on devrait avoir 80 victoires pour A
    - Ex : Dans un système de recommandation, si un utilisateur regarde 80% de comédies romantiques et 20% de thrillers, le système de recommandation devrait suivre ces valeurs
- Plusieurs métriques = vision complète

Métriques offline vs impact business

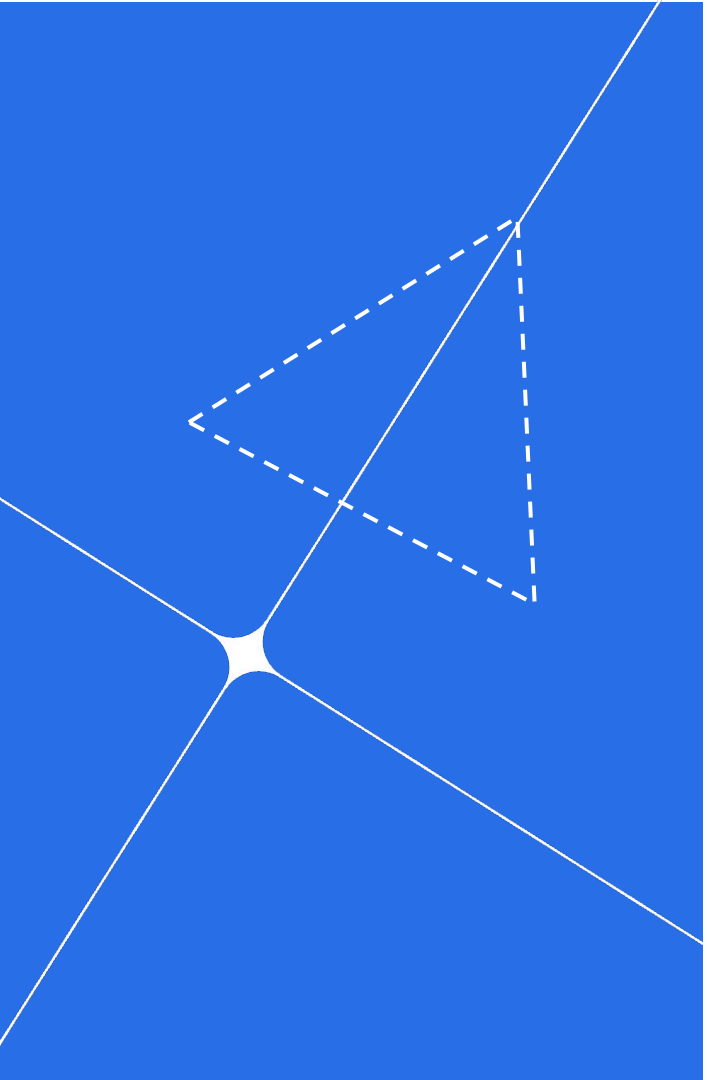
- Offline
  - AUC
  - F1
  - logloss
  - calibration
- Business (non mesurable à l'entraînement en général)
  - taux de rétention
  - coût des actions
  - ROI des campagnes
  - satisfaction utilisateur
- Lien clé
  - métriques ML entraînent des décisions
  - décisions entraînent coûts / gains
- Une métrique n'a de valeur que reliée à une action

Spoiler prochain cours : dérive des métriques

- Constat
  - Un modèle ne reste jamais optimal
  - Les données changent
  - Les comportements évoluent
- Symptômes
  - baisse progressive de l'AUC
  - dégradation de calibration
  - instabilité par segment
- Conséquence
  - nécessité de surveiller les métriques dans le temps
  - préparation au monitoring et au retraining

Checkpoint #4

- Vous devez être capables de :
  - expliquer pourquoi l’accuracy est souvent insuffisante
  - justifier l’usage de l’AUC pour le churn
  - identifier les limites des métriques seuil-dépendantes
  - relier métriques ML et décisions business
  - anticiper pourquoi les métriques évoluent dans le temps
- Choisir une métrique = choisir une stratégie de décision



Model Registry et Lifecycle Management

Pourquoi un model registry existe

- Le registry permet de stocker des modèles de façon organisée
- Problème sans registry
  - modèles dispersés (fichiers, dossiers, buckets)
  - aucune source de vérité
  - décisions de déploiement implicites
  - rollback artisanal
- Rôle du registry
  - point central des modèles utilisables
  - historique des versions
  - contrôle de ce qui peut aller en production
- Le registry est à la production ML ce que Git est au code

Un modèle n’est pas un fichier

- Vision naïve
  - modèle = .pkl, .joblib, .bin
- Vision production
  - modèle = artefact versionné comprenant :
    - binaire du modèle
    - signature (schéma entrée/sortie)
    - environnement
    - métriques associées
    - lien vers le run d’origine
- Conséquence
  - on ne “copie” pas un modèle
  - on référence une version
- Le fichier est un détail d’implémentation

Versions et stages de modèle

- Version
  - incrément automatique
  - liée à un run précis
  - immuable
- Stage (état logique)
  - None : expérimental
  - Staging : candidat à la production
  - Production : modèle officiel
  - (Archived : obsolète)
- Principe clé
  - une seule version en Production
  - plusieurs versions en Staging possibles
- Le stage exprime une intention opérationnelle

Logique de promotion

- Questions fondamentales
  - Qui peut promouvoir un modèle ?
  - Quand un modèle devient-il “meilleur” ?
  - Sur quels critères ?
- Critères typiques
  - métriques > modèle courant
  - validation humaine
  - tests de non-régression
  - conformité métier / légale
- Point important
  - promotion n’est pas l’entraînement
  - décision explicite, traçable
- Gouvernance minimale mais explicite

Le rollback : opération critique

- Le rollback est une opération permettant de revenir à une version précédente
- Pourquoi le rollback est indispensable
  - bugs en production
  - données inattendues
  - dérive rapide
- Bon rollback
  - instantané
  - déterministe
  - sans réentraînement
- Registry bien conçu
  - permet de revenir à :
    - version N-1
    - version stable connue
  - sans modifier le code
- Si le rollback est difficile, le système est fragile

Lineage : de la donnée au déploiement

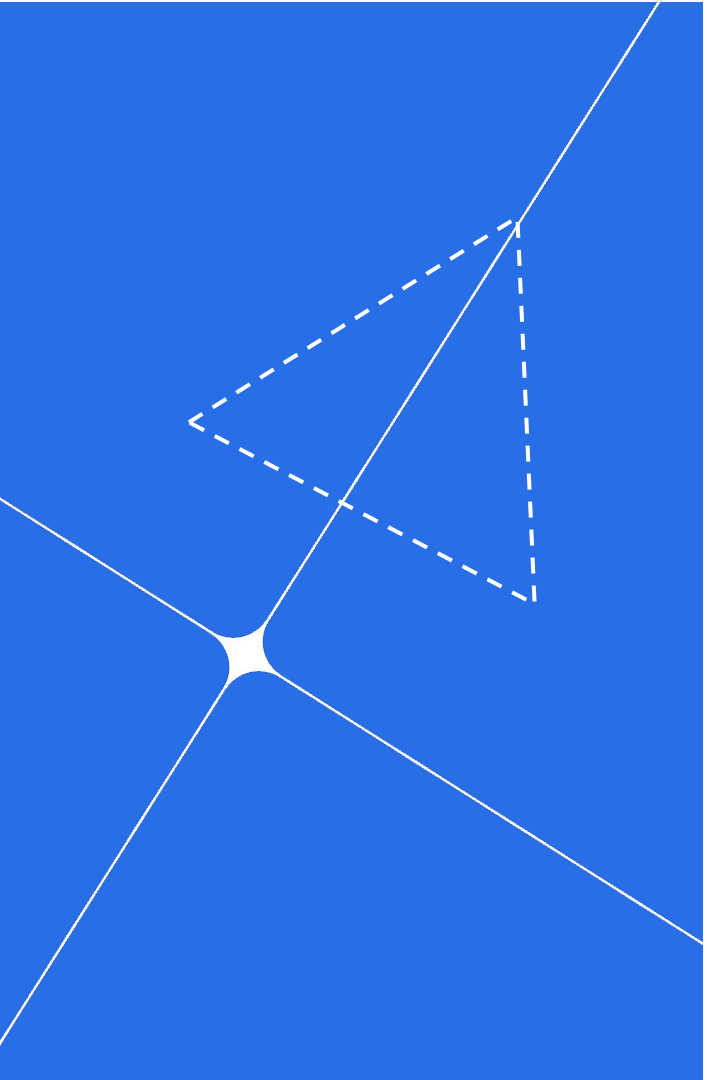
- Lineage = description de toutes les composantes donnant lieu à un résultat final
- Chaîne de traçabilité
- Pourquoi c'est critique
  - auditabilité
  - explication des décisions
  - analyse post-mortem
  - conformité (finance, santé, RGPD…)
- Sans lineage, pas de confiance

Human-in-the-loop aujourd'hui

- État réaliste des systèmes ML
  - décisions souvent manuelles
  - validation humaine avant production
  - promotion via UI / revue
- Évolution naturelle
  - règles automatiques
  - promotion conditionnelle
  - CI/CD des modèles
- Message clé
  - commencer simple
  - structurer dès maintenant
- L'automatisation vient après la traçabilité

Checkpoint #5

- Vous devez être capables de :
  - expliquer le rôle d'un model registry
  - distinguer fichier vs modèle versionné
  - décrire versions et stages
  - justifier une logique de promotion
  - expliquer pourquoi le rollback est central
  - raisonner en termes de lineage end-to-end
- Gérer des modèles = gérer un cycle de vie, pas des fichiers



Architecture de serving : Du Registry à l'API

Sans serving, le training est inutile

- Réalité opérationnelle
  - Un modèle non servi :
    - ne génère aucune valeur
    - ne peut pas être testé en conditions réelles
    - ne peut pas être monitoré
- Serving = moment de vérité
  - contraintes temps réel
  - données partielles
  - erreurs utilisateurs
  - charge concurrente
- Le serving est souvent là où les hypothèses du training cassent

Architecture de serving

- Composants
  - API : orchestration + contrat sur les entrées/sorties
  - Feature Store : fourniture des features
  - Registry : sélection du bon modèle

Séparation stricte des responsabilités

- API
  - validation des entrées
  - gestion des erreurs
  - exposition HTTP
  - observabilité (logs, métriques)
- Feature Store
  - calcul historique (offline)
  - lookup rapide (online)
  - cohérence training / inference
- Modèle
  - transformation features et prédiction
  - aucune logique d’accès aux données
- Mélanger ces rôles = dette technique assurée

Pourquoi l'API ne doit pas calculer les features

- Anti-pattern courant
  - recalcul des features dans l'API
  - duplication de logique
  - versions divergentes
- Conséquences
  - training-serving skew
  - bugs silencieux
  - performances imprévisibles
- Bon principe
  - l'API consomme des features
  - elle ne les définit pas
- La logique feature appartient au Feature Store

Contraintes du online feature retrieval

- **Latence**
  - objectif : quelques millisecondes
  - pas de jointures lourdes
  - pas de calcul dynamique
- **Fraîcheur**
  - dépend de la matérialisation
  - compromis fraîcheur / coût
- **Complétude**
  - features parfois manquantes
  - gestion des valeurs par défaut
- Le serving impose des compromis absents du training

Charger le modèle “Production”

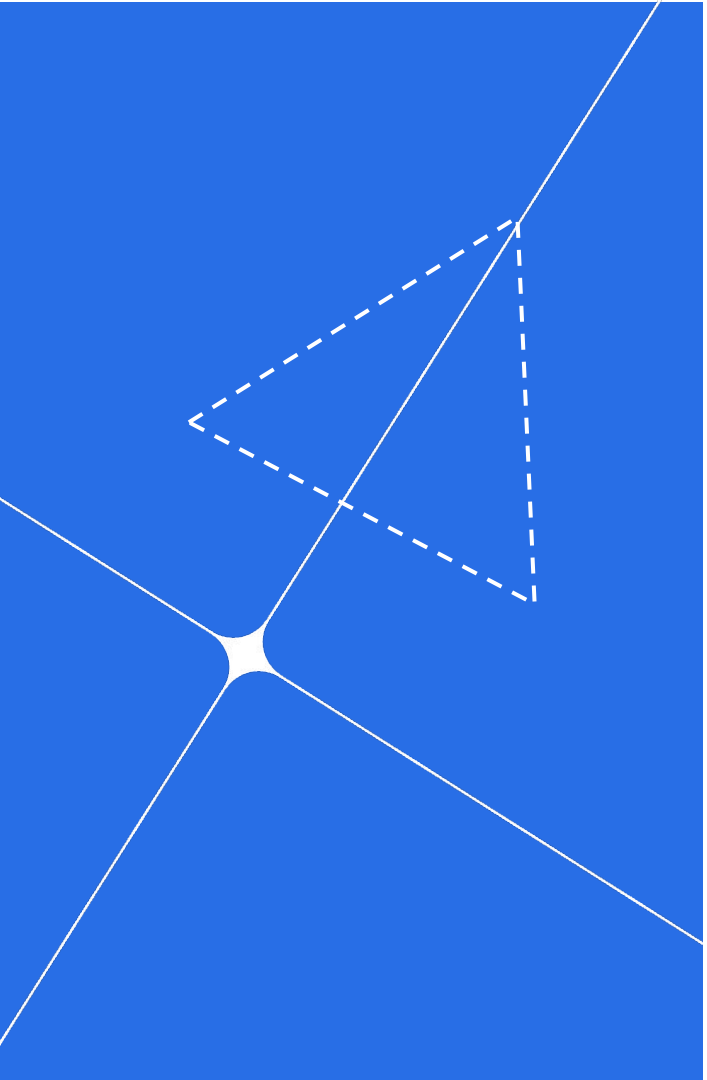
- Pourquoi le stage est critique
  - l’API ne doit pas connaître :
    - les numéros de version
    - les expériences
  - elle charge :
    - le modèle validé
- Avantages
  - promotion sans redéployer le code
  - rollback immédiat
  - séparation décisions ML / infra
- Le registry devient le point de contrôle

Piège critique : chargement du modèle

- Mauvaise pratique
  - charger le modèle à chaque requête
  - latence élevée
  - consommation mémoire
  - instabilité sous charge
- Bonne pratique
  - chargement au démarrage du service
  - modèle en mémoire
  - prédictions rapides
- Règle simple
  - Le modèle est un état du service, pas une dépendance par requête

Checkpoint #6

- Invariants d’une architecture de serving saine
  - API = orchestration, pas feature engineering
  - Feature Store = source unique des features
  - Modèle = composant pur de prédiction
  - Le modèle chargé est celui en Production
  - Le modèle est chargé une seule fois
  - Le rollback ne nécessite pas de redéploiement
- Toute violation crée un système fragile



Design d'API pour les systèmes ML

Ce qui rend une API ML différente

- API classique
  - CRUD (create, read, update, et delete/GET, POST, PUT et DELETE)
  - logique métier déterministe
  - réponses exactes attendues
- API de prédiction
  - comportement probabiliste
  - dépend fortement des données
  - qualité variable dans le temps
  - couplée à un modèle versionné
- Conséquence
  - l'API ML est un composant expérimental contrôlé
  - Nécessite plus de garde-fous
- Une API ML est un système statistique exposé

Validation des entrées

- Pourquoi c’est critique
  - un input invalide = prédiction absurde
  - erreurs souvent silencieuses
- Bonnes pratiques
  - schémas stricts (types, champs obligatoires)
  - validation des IDs d’entités
  - refus explicite des requêtes ambiguës
- Exemples d’erreurs à bloquer
  - entité inconnue
  - champs manquants
  - mauvais types
  - payloads partiels incohérents
- Mieux vaut refuser que prédire n’importe quoi

Design de la sortie

- Sortie minimale utile
  - score / probabilité
  - identifiant de la requête
- Sorties recommandées
  - probabilité plutôt que classe brute
  - version du modèle utilisée
  - timestamp
  - éventuellement :
    - seuil appliqué
    - décision dérivée
- Principe
  - séparer score et décision
  - laisser la décision au système aval
- Une probabilité est plus flexible qu’un label

Health checks & readiness

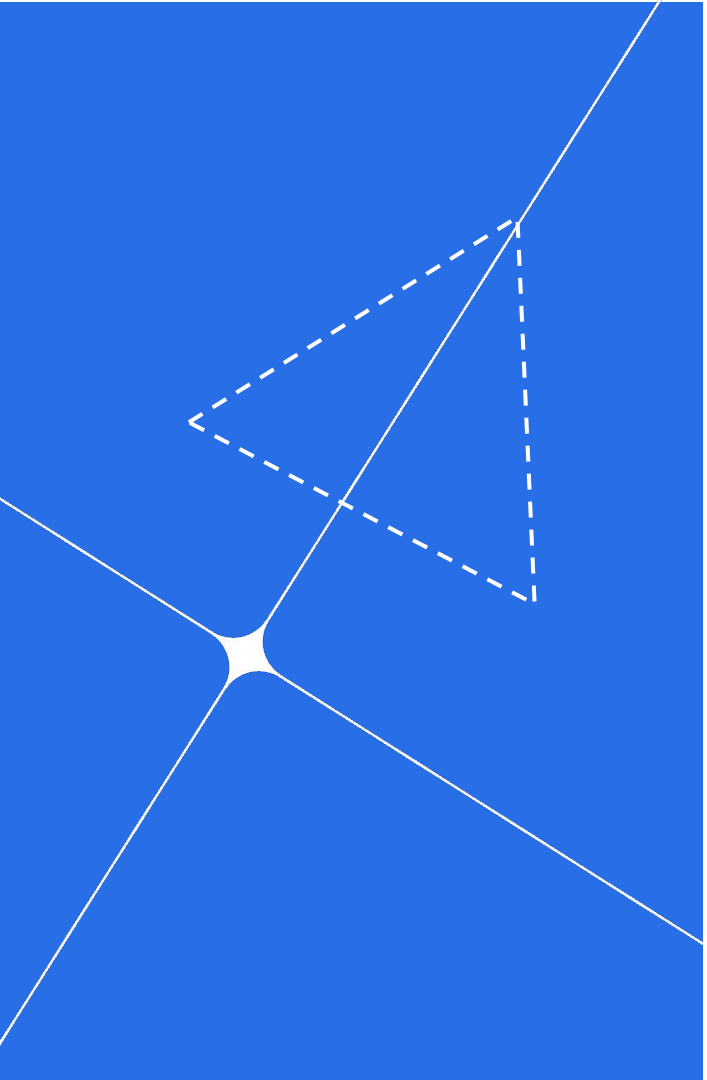
- Health check (/health)
  - le service répond
  - dépendances accessibles
- Readiness
  - modèle chargé en mémoire
  - feature store accessible
  - registry atteignable
- Pourquoi séparer
  - un service “up” peut être inutilisable
  - nécessaire pour orchestration / scaling
- Un service ML doit dire s’il est prêt à prédire

Modes d’erreur spécifiques aux API ML

- Erreurs liées aux données
  - features manquantes
  - valeurs hors distribution
  - entités absentes
- Erreurs liées au modèle
  - modèle non chargé
  - signature incompatible
  - version supprimée
- Erreurs systémiques
  - latence excessive
  - dépendance externe lente (feature store)
- Principe
  - erreurs explicites
  - logs exploitables
  - pas de fallback silencieux
- Les erreurs ML doivent être observables

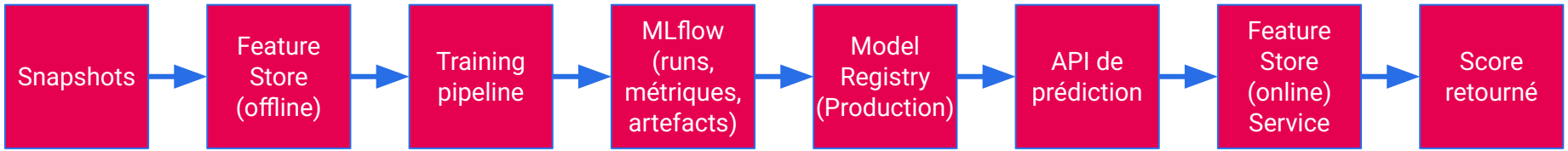
Checkpoint #7

- Vous devez être capables de :
  - expliquer pourquoi une API ML est différente d’une API classique
  - définir une validation d’entrée robuste
  - concevoir une sortie exploitable et traçable
  - distinguer health vs readiness
  - anticiper les modes d’erreur spécifiques au ML
- Une API ML sûre protège le modèle et le business



Vue de bout-en-bout et transition vers le TP

Vue end-to-end : du training à la prédiction



- Message clé
  - Chaque brique a un rôle unique
  - Le modèle n’est jamais isolé
- La valeur vient de l’intégration, pas du modèle seul

Ce qui change quand un nouveau modèle est entraîné

- Ce qui change
  - métriques
  - version du modèle
  - artefacts associés
  - éventuellement le stage (Staging vers Production)
- Ce qui ne change pas
  - API
  - logique de serving
  - contrat d’entrée / sortie
  - consumers aval
- Un bon système absorbe le changement de modèle sans casser le reste

Stabilité vs variabilité dans un système ML

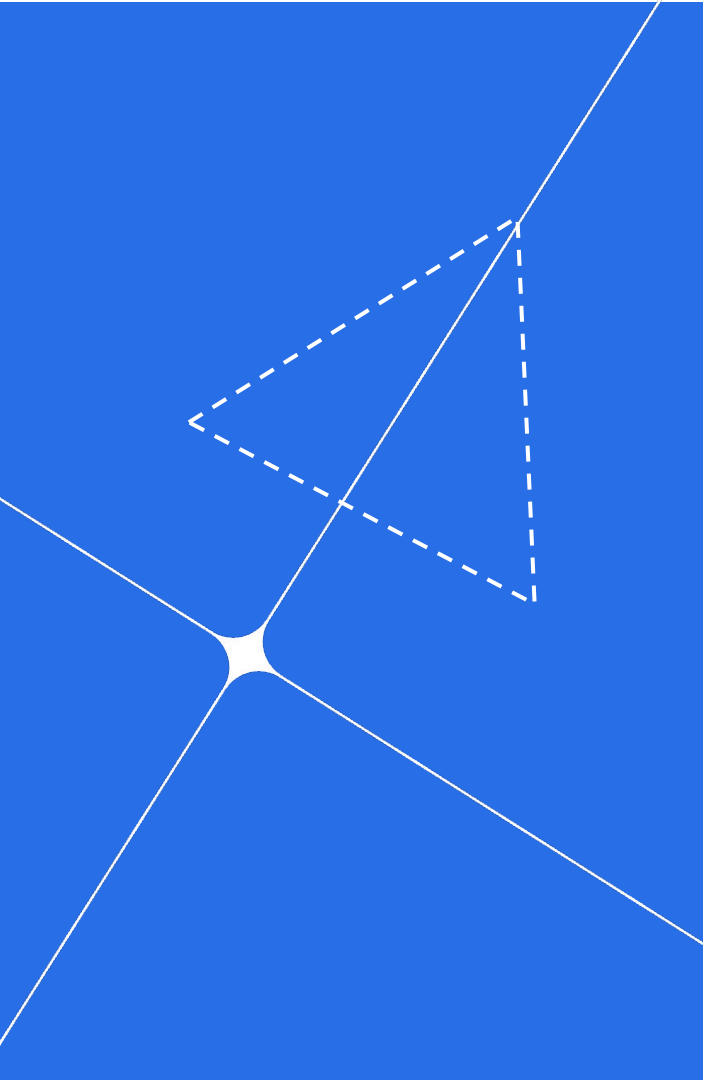
- Doit être stable
  - schéma des features
  - signature du modèle
  - API contract
  - pipeline de déploiement
- Peut évoluer
  - modèle
  - hyperparamètres
  - métriques
  - seuils de décision
- La stabilité est une condition de l’itération rapide

Ce que vous allez implémenter dans le TP

- Concrètement, dans le lab
  - pipeline d'entraînement déterministe
  - récupération de features via le Feature Store
  - log des runs et métriques dans MLflow
  - enregistrement du modèle
  - promotion en Production
  - chargement du modèle Production dans l'API
  - première prédiction end-to-end
- Premier vrai système ML "production-like"

Checkpoint final

- À l’issue de ce cours, vous devez pouvoir :
  - expliquer pourquoi le training est une pipeline
  - justifier l’usage d’un experiment tracking
  - distinguer tracking vs registry
  - raisonner sur le cycle de vie d’un modèle
  - expliquer comment une API charge le bon modèle
  - décrire un flux complet, du training au serving
- À partir d’ici, on n’entraîne plus des modèles, on opère des systèmes ML



En route pour le TP