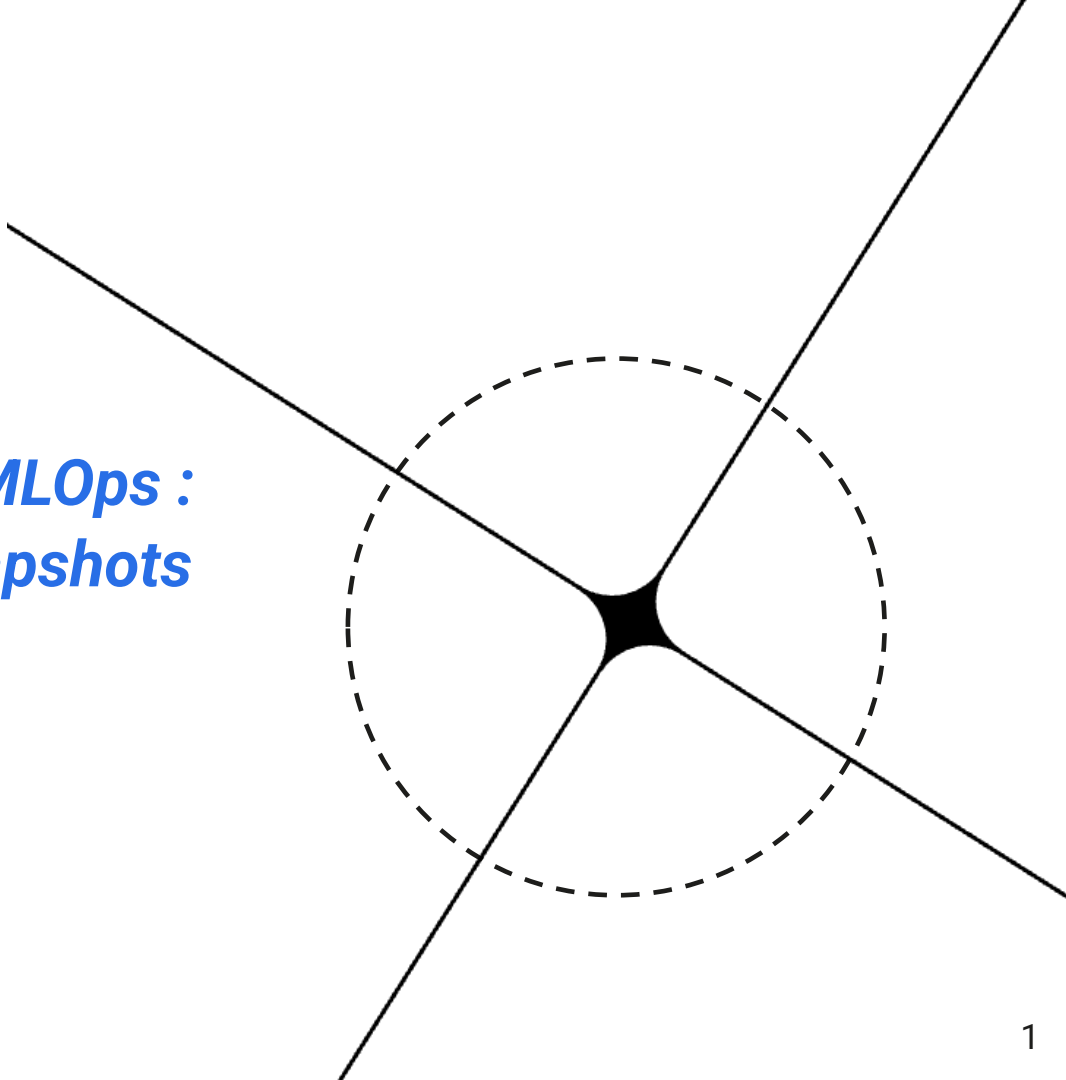
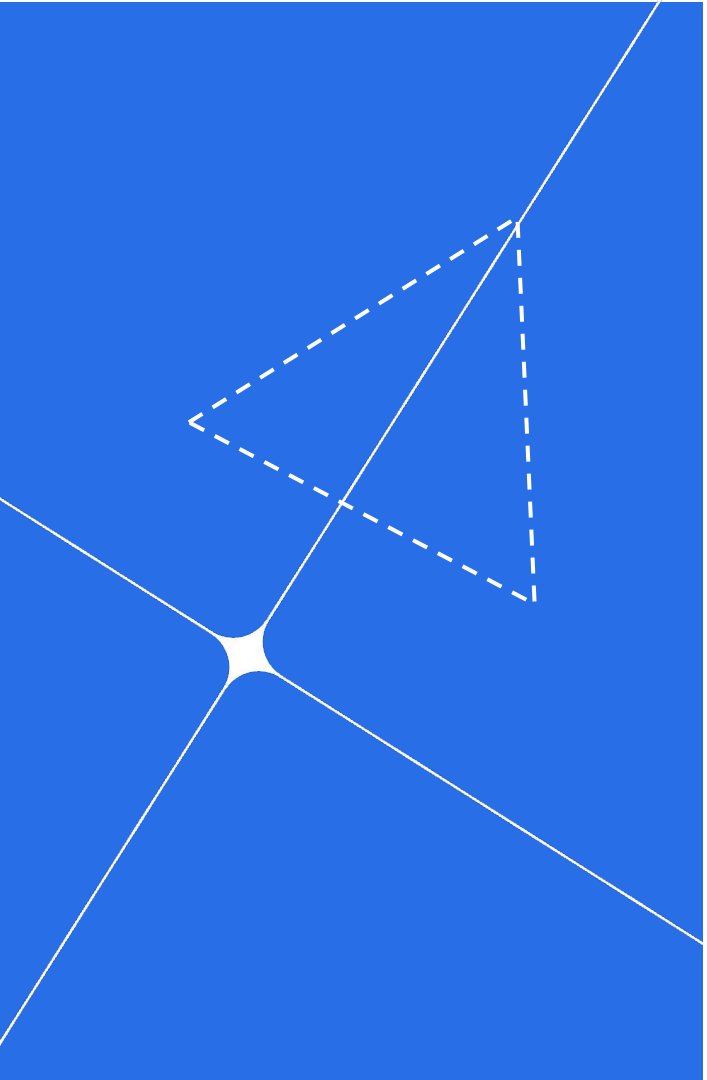
INSTITUT POLYTECHNIQUE DE PARIS

TELECOM SudParis

Data engineering pour le MLOps : Ingestion, validation & snapshots temporels

Julien Romero





De l’importance de l’ingestion de données

Exemple du TP : StreamFlow

- Vous êtes un ingénieur ML dans une entreprise appelée StreamFlow vendant un produit avec un abonnement. On vous demande de concevoir un modèle de prédiction de l'attrition (churn) des clients, c'est-à-dire de prédire si un client va quitter votre service ou non.
  - Votre modèle doit être déployable en production
- Tous les mois :
  - Vous recevez les données sur les activités des utilisateurs sur votre site
  - Vous avez accès aux informations de paiement et les interactions avec le support
  - Vous pouvez gérer des labels churn/no churn plus tard
- Nous allons commencer par traiter l'ingestion des données, qui est cruciale pour entraîner un modèle

Pourquoi les systèmes de ML dépendent de pipelines de données robustes

- Un modèle n'est jamais meilleur que les données utilisées pour l'entraîner.
- Un pipeline d'ingestion doit garantir :
  - La **cohérence des données** (schéma, types, distribution).
  - La **stabilité temporelle** (ce qui est vu à un instant t doit être reconstruit plus tard).
  - La **complétude** (tous les fichiers, toutes les lignes, toutes les clés étrangères).
  - La **traçabilité** (on doit comprendre l'état des données lors de l'entraînement).
- Sans un pipeline solide, même un excellent modèle produit des prédictions incohérentes.

Exemple : corruption silencieuse qui casse le modèle de churn

- Cas réel typique :
  - À t0, le modèle de churn fonctionne correctement.
  - À t1, un nouvel export mensuel a introduit un changement discret :
    - valeurs manquantes dans *nb_sessions*,
    - nouvelle colonne non gérée,
    - type string au lieu de float pour une métrique d'usage.
  - Le pipeline ignore ces anomalies et les données "polluées" entrent dans le training set.
  - Le modèle dérive, la performance s'effondre, mais aucune erreur n'a été détectée.
- Ce type de bug coûte très cher et met parfois des semaines à diagnostiquer.

Training-serving skew et ingestion inconsistente

- Le **training-serving skew** survient lorsque :
  - Les données utilisées pendant l'entraînement ne correspondent pas à celles utilisées en production.
  - Les features sont calculées différemment entre les environnements.
  - L'ingestion mensuelle réécrit accidentellement des valeurs historiques.
  - Certaines transformations sont appliquées uniquement lors de l'inférence ou uniquement lors de l'entraînement.
- Une ingestion correcte garantit que **les données d'entraînement représentent fidèlement le contexte réel de production**.

Pourquoi l'ingestion est la première étape de la reproductibilité

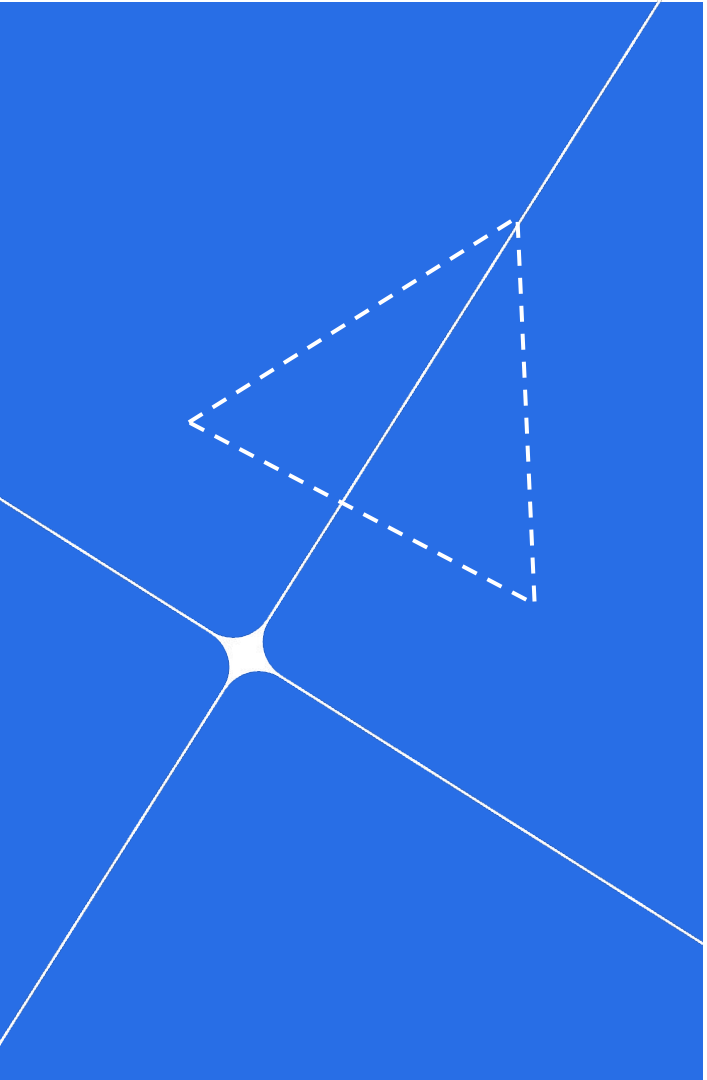
- La reproductibilité en MLOps repose sur l'idée suivante :
  - À partir des mêmes données sources, tout run doit reconstruire exactement le même état.
- Cela implique :
  - un schéma stable,
  - une logique d'ingestion idempotente (faire deux ingestions donne le même résultat qu'une seule),
  - aucune dépendance cachée,
  - des transformations déterministes,
  - la capacité à rejouer un mois passé (ex: janvier 2024) et obtenir le même résultat.
- Sans une ingestion reproductible, impossible de garantir la reproductibilité des modèles.

Principe MLOps : des pipelines déterministes

- Pour un pipeline déterministe :
  - Même input = même output.
  - Pas d'effets de bord (pas de réécriture involontaire, pas de comportements aléatoires).
  - Les scripts produisent un état identique, que ce soit sur votre machine, dans Docker ou sur un serveur CI/CD.
  - Possibilité de reconstruire une version antérieure des données avec un timestamp donné.
- La déterminisme est une condition nécessaire pour obtenir des **artefacts de modèles traçables et fiables**.

Objectifs pédagogiques du cours d'aujourd'hui

- À la fin du cours et du TP, vous serez capables de :
  1. Comprendre la structure d'un pipeline d'ingestion moderne.
  2. Implémenter une logique d'upsert fiable dans PostgreSQL.
  3. Appliquer une validation systématique avec Great Expectations.
  4. Produire des snapshots temporels pour garantir la cohérence historique.
  5. Orchestrer l'ensemble avec Prefect pour obtenir un pipeline réexécutable et observable.
- Ces fondations préparent l'introduction au Feature Store (Feast) lors du prochain cours.



Des données brutes à une pipeline de données structurées

Définition : pipeline d'ingestion

- Dans un système MLOps, un pipeline d'ingestion est un ensemble d'étapes qui :
  - Reçoit des données brutes (CSV, extracts externes).
  - Les nettoie, valide et structure dans une base relationnelle.
  - Prépare les données pour le calcul de features et l'entraînement.
  - Garantit un comportement répétable, auditables et déterministe.
- L'ingestion est la première brique d'un pipeline ML fiable.

Données brutes, données traitées, données “curées”

- **Raw data (brutes)**
  - Issues directement de la source.
  - Peu fiables : incohérences, types variés, colonnes manquantes, bruit.
- **Processed data (traitées)**
  - Normalisées, typées, stockées dans PostgreSQL.
  - Base de travail pour les features et la validation.
- **Curated data (curées)**
  - Données stabilisées, enrichies et versionnées (snapshots temporels).
  - Utilisées pour l’entraînement, le test, et le Feature Store.
- Cette hiérarchie protège le pipeline des erreurs amont.

Modes d’ingestion : full, incremental, upsert

- **Full reload**
  - Réécrit entièrement les tables.
  - Simple mais coûteux et risqué pour des volumes croissants.
- **Incremental ingestion**
  - Charge uniquement les nouvelles lignes.
  - Nécessite une logique de delta (timestamps, marqueurs).
- **Upsert ingestion**
  - Insère ou met à jour selon la clé primaire.
  - Évite duplication et perte de données.
  - Recommandé pour les pipelines ML récurrents.

Pourquoi l'upsert est la norme en production ML

- L'upsert 'update ou insert si n'existe pas déjà) répond directement aux contraintes du ML en production :
  - Les fichiers mensuels peuvent contenir des mises à jour d'utilisateurs existants.
  - La même ingestion doit pouvoir être rejouée sans créer de doublons.
  - Des valeurs tardives ou corrigées doivent remplacer l'existant.
  - Le modèle se base sur un historique fiable et stable.
  - L'upsert permet une ingestion idempotente, essentielle pour CI/CD et orchestration automatique.
- C'est aujourd'hui le standard des pipelines de données continus.

Exemple SQL générique : INSERT … ON CONFLICT

Modèle générique d'upsert PostgreSQL :

```
INSERT INTO subscriptions (user_id, plan_type, renewal_date, status)
VALUES ($1, $2, $3, $4)
ON CONFLICT (user_id)
DO UPDATE SET
    plan_type = EXCLUDED.plan_type,
    renewal_date = EXCLUDED.renewal_date,
    status = EXCLUDED.status;
```

Points clés :

- user_id doit être une clé unique ou primaire.
- EXCLUDED représente les valeurs nouvellement proposées.
- Le comportement est déterministe, même si la même ligne est réinsérée plusieurs fois.

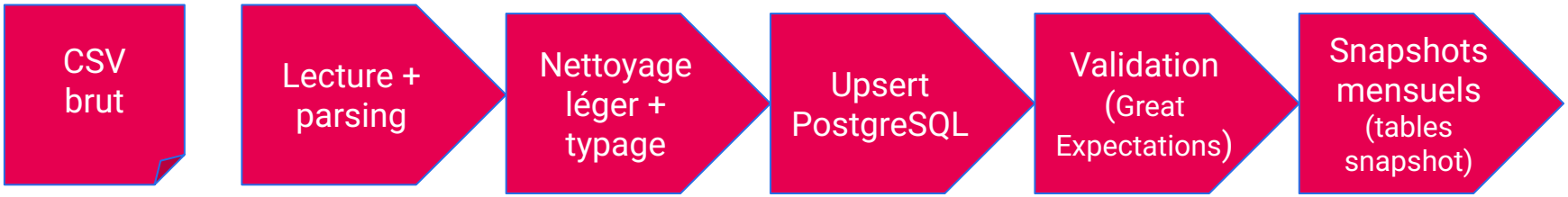
L’importance de l’idempotence

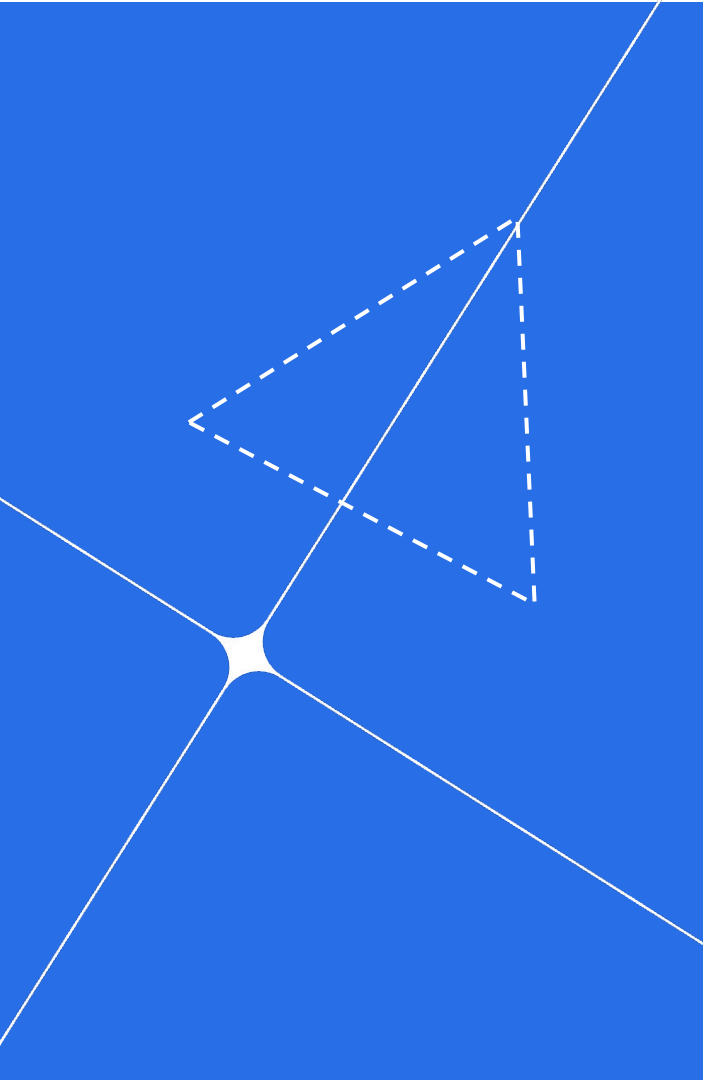
- L’idempotence signifie :
  - Exécuter deux fois le pipeline produit le même état final.
- Exemples :
  - Un étudiant relance l’ingestion après une erreur => la base reste correcte.
  - Une tâche Prefect échoue au milieu, puis redémarre => pas de doublons.
  - Un mois passé doit pouvoir être reconstruit à l’identique pour une analyse ou un audit.
  - Une ingestion nocturne automatisée doit résister aux interruptions réseau.
- Sans idempotence, la reproductibilité du système ML s’effondre.

Outils de support : PostgreSQL déjà configuré

- Dans notre architecture :
  - Toutes les tables relationnelles sont déjà définies dans PostgreSQL.
  - Chaque table possède une clé primaire stable (ex: user_id).
  - Les schémas sont adaptés à un usage ML (tables agrégées, labels, profils).
  - L'ingestion se fait depuis Docker via des scripts Python orchestrés par Prefect.
- PostgreSQL sert de backbone structuré au pipeline d'ingestion.

Pipeline d'ingestion étape par étape





Logique temporelle et les snapshots

Pourquoi le machine learning exige une cohérence temporelle

- Un modèle de ML doit apprendre sur des données reflétant l'état réel du monde à un instant donné.
- Cela implique :
  - Des features calculées avec les informations disponibles à ce moment-là, pas après.
  - Une séparation stricte entre passé et futur lors de l'entraînement.
  - La capacité de reconstruire l'état des données à n'importe quel mois historique.
- Sans cohérence temporelle, les performances du modèle sont artificiellement élevées et trompeuses.

Prévenir la data leakage pendant l'entraînement

- Le **data leakage** survient lorsque l'entraînement utilise des informations qui n'existaient pas encore lors de la prédiction réelle.
- Exemples typiques :
  - Utiliser un agrégat calculé sur une période postérieure au label.
  - Incorporer une correction de données faite 3 mois après les faits.
  - Utiliser une version "live" qui a été mise à jour après la période d'étude.
- Les snapshots permettent d'éviter toute fuite de données en figeant une version strictement datée.

Qu’est-ce qu’un “snapshot” en data engineering ?

- Un **snapshot** est une copie figée, immuable, d’un ensemble de données à un moment précis.
- Propriétés essentielles :
  - Versionnée par une date (ex: 2024-01-31).
  - Représente l’état exact des tables à cet instant.
  - Sert de base pour l’entraînement du modèle.
  - Ne doit jamais être modifiée a posteriori.
- Les snapshots sont le fondement d’un pipeline ML fiable et auditable.

Exemple réel : snapshots mensuels pour abonnements ou finance

- Dans les entreprises :
  - Les banques conservent des positions financières mensuelles pour calculer risque, conformité, reporting.
  - Les services d'abonnement (SaaS, télécoms, streaming) stockent un état mensuel du client pour analyser churn ou lifetime value.
  - Les systèmes de scoring nécessitent des vues mensuelles constantes pour garantir l'équité et la comparabilité.
- Les snapshots permettent d'expliquer pourquoi un score ou une décision a été prise à une époque donnée.

Le rôle du timestamp as_of

- Le champ *as_of* indique la date de référence du snapshot.
- Fonctions :
  - Identifie de manière unique un état figé.
  - Permet de reconstruire un dataset temporel complet.
  - Sert d'index logique pour l'entraînement (sélection par mois).
  - Sépare nettement données historiques et données mises à jour.
- *as_of* devient un élément central de la traçabilité du pipeline ML.

Tables snapshot vs tables “live”

- **Tables live**
  - Contiennent l’état actuel des entités (abonnements, usage, paiements).
  - Se mettent à jour avec chaque ingestion.
- **Tables snapshot**
  - Contiennent une version figée, immuable, datée.
  - Permettent de reconstruire un dataset historique complet.
  - Sont indexées par (user_id, as_of).
- Les modèles devraient toujours s’entraîner sur les tables snapshot, jamais sur les tables live.

Comment Feast utilisera les snapshots plus tard

- Feast, notre Feature Store, exploitera ces snapshots pour :
  - Faire des Historical Feature Retrievals : récupérer les features telles qu'elles existaient au moment du label.
  - Garantir la correspondance entre features d'entraînement et features d'inférence.
  - Éviter automatiquement la data leakage grâce à l'alignement temporel des données.
  - Reproduire des jeux de données à partir d'un timestamp ou d'un range de dates.
- Les snapshots sont donc une dépendance directe du Feature Store.

Pattern SQL pour créer un snapshot

```
INSERT INTO subscriptions_snapshots
SELECT
    user_id,
    plan_type,
    status,
    months_since_signup,
    '{AS_OF}' AS as_of  # AS_OF est un paramètre donné en entrée
FROM subscriptions;
```

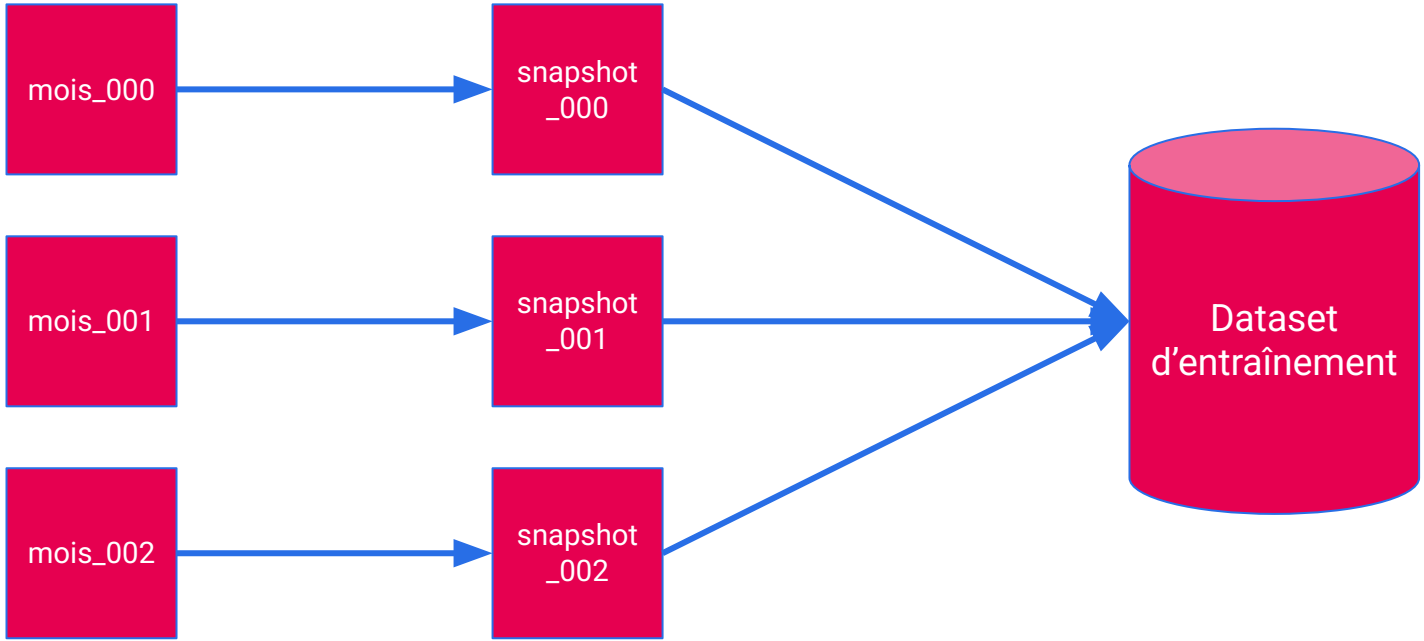
Principes clés :

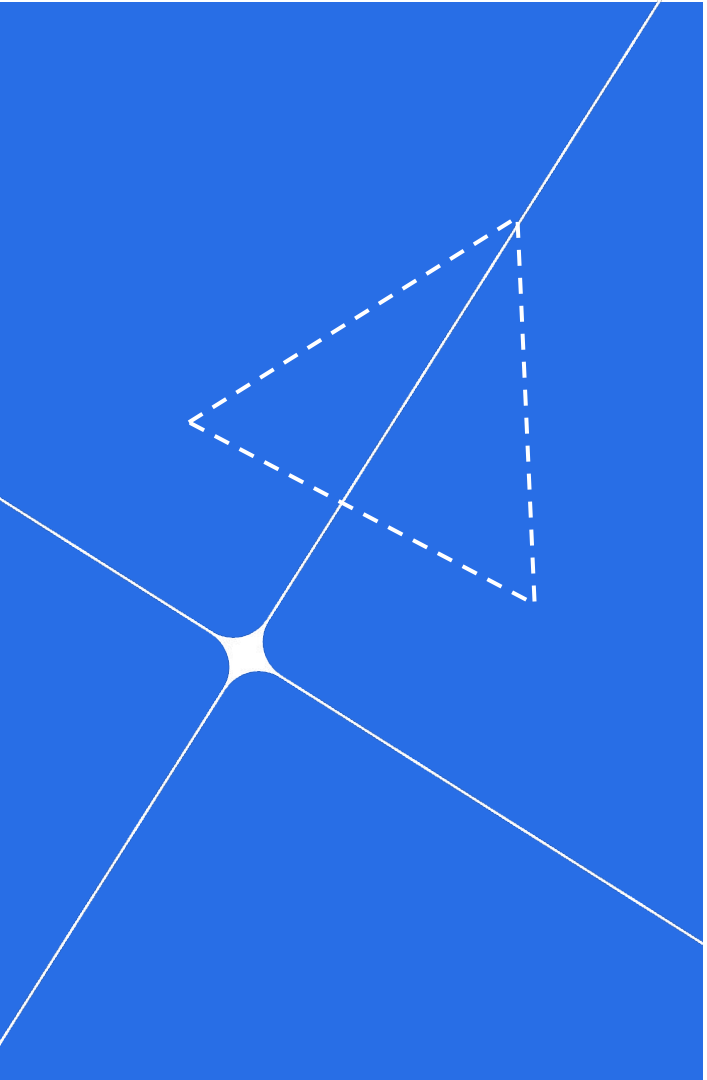
- Le snapshot n'écrase pas l'existant : il ajoute une nouvelle partition temporelle.
- Il doit être idempotent : relancer l'ingestion crée le même snapshot.
- Aucune transformation imprévisible ne doit intervenir après insertion.

Patterns de validation des snapshots

- Avant d’accepter un snapshot, on valide :
  - **Aucune ligne manquante** par rapport aux tables live.
  - **Unicité** : (user_id, as_of) est unique.
  - **Homogénéité** : distributions cohérentes d’un mois à l’autre.
  - **Cohérence** : même schéma, mêmes types, mêmes colonnes.
  - **Intégrité temporelle** : pas de *as_of* futur, pas de données hors période.
- Les snapshots doivent être stables et comparables entre mois.

Timeline des snapshots pour l'entraînement





Validation des données avec Great Expectations

Pourquoi la validation est essentielle (failures rapides > erreurs silencieuses)

- Dans les systèmes ML, la majorité des pannes provient de données incorrectes et non détectées.
- La validation permet :
  - De détecter immédiatement les anomalies.
  - D'éviter que des données corrompues entrent dans le training set.
  - De rendre le pipeline **prévisible et contrôlable**.
  - D'offrir une garantie de qualité avant les snapshots.
- Une erreur détectée tôt vaut mieux qu'un modèle entraîné sur des données fausses.

Notions de base : le schema drift

- Le **schema drift** survient quand le schéma des données change de façon non prévue :
  - Colonnes ajoutées ou supprimées.
  - Types qui changent (string → int, int → float).
  - Valeurs manquantes nouvellement introduites.
  - Ordre des colonnes modifié.
- Le ML est très sensible à ces dérives : même un petit changement peut invalider la pipeline.

Introduction à Great Expectations

- **Great Expectations (GE)** est un framework pour :
  - Définir des règles explicites sur les données (“expectations”).
  - Exécuter ces règles automatiquement lors de l’ingestion.
  - Produire des rapports lisibles par humains et machines.
  - Arrêter ou alerter quand une violation se produit.
- GE formalise la qualité des données directement dans le pipeline.

Expectations : définition et exemples

- Une expectation = une règle déclarative appliquée à une colonne ou une table.
- Exemples :
  - “Cette colonne ne doit pas contenir de valeurs nulles.”
  - “Cette colonne doit être comprise entre 0 et 2000.”
  - “Le nombre de lignes doit être au moins 10 000.”
  - “Les user_id doivent être uniques.”
- Collections d’expectations = expectation suite.

Exemple d'expectation : aucune valeur NULL sur user_id

```
expectation_suite.expect_column_values_to_not_be_null("user_id")
```

Pourquoi cette règle est essentielle :

- Le user_id est la clé primaire.
- Toute valeur NULL rend l'upsert ou le snapshot incohérent.
- Un NULL peut cacher une corruption amont (erreur d'export).
- Cette expectation protège directement l'intégrité du pipeline.

Exemple d'expectation : bornes numériques réalistes

```
expectation_suite.expect_column_values_to_be_between(
    "nb_sessions",
    min_value=0,
    max_value=5000
)
```

Objectifs :

- Détecter les valeurs impossible (ex: -3 sessions).
- Prévenir les ruptures de distribution.
- Repérer les erreurs d'intégration ou de format (ex: texte à la place d'un nombre).

Ces contrôles stabilisent les features.

Échec strict (“hard fail”) vs échec toléré (“soft fail”)

- **Hard fail**
  - Arrête immédiatement la pipeline.
  - Utilisé pour les étapes critiques : ingestion => validation => snapshots.
  - Garantit que rien de corrompu n’entre dans la base historique.
- **Soft fail**
  - Génère un avertissement mais la pipeline continue.
  - Utilisé pour le monitoring ou les analyses non bloquantes.
- Dans notre cours, l’ingestion utilise exclusivement des hard fails.

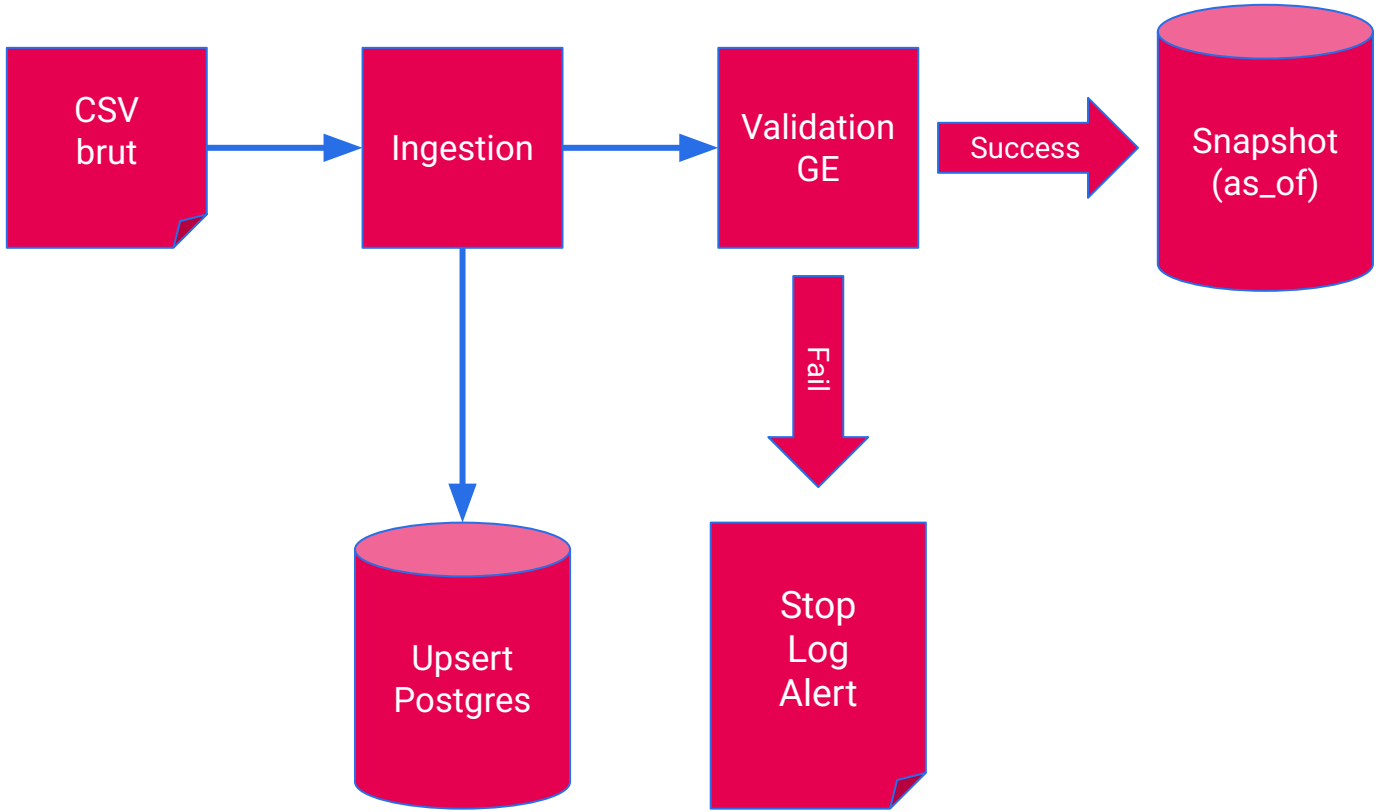
Output GE : le rapport de validation

- GE produit un objet JSON et/ou un rapport HTML contenant :
  - La liste des expectations testées.
  - Leur statut : success / failure.
  - Les valeurs problématiques éventuelles.
  - Les statistiques de la table (distributions, types).
  - Un résumé lisible pour l'utilisateur.
- Ce rapport permet d'auditer chaque ingestion mensuelle.

Emplacement de la validation dans la pipeline d'ingestion

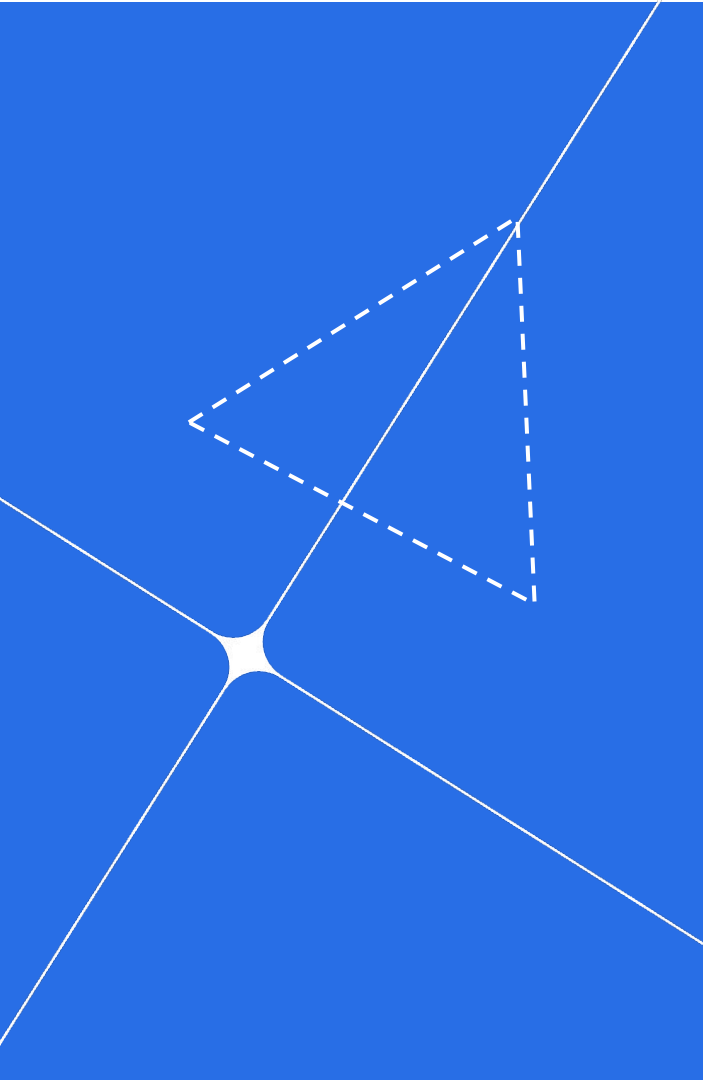
- Ordre strict :
  1. Upsert des tables live.
  2. Validation via GE.
  3. Snapshots uniquement si la validation réussit.
- La validation agit comme un garde-fou :
  - Si quelque chose ne va pas, la pipeline s'arrête avant d'altérer l'historique.

La validation comme porte d'entrée



Exemple : Valeur négative injectée volontairement

- Imaginez qu'un export mensuel contient :
  - *nb_sessions = -12* pour un utilisateur.
- Sans validation :
  - L'erreur passe silencieusement.
  - Les features agrégées deviennent incohérentes.
  - Le modèle interprète cela comme un comportement atypique => dérive.
  - Impossible de comprendre la cause plus tard.
- Avec GE :
  - L'expectation "nb_sessions >= 0" échoue immédiatement.
  - La pipeline s'arrête.
  - Aucune pollution n'entre dans les snapshots.
  - Le bug est corrigé avant qu'il ne devienne coûteux.



Fondamentaux de l'orchestration

Introduction à Prefect

Pourquoi a-t-on besoin d'un orchestrateur ?

- Un pipeline de ML moderne implique de nombreuses étapes dépendantes :
  - chargement de données, validations, snapshots, extraction de features, entraînement, déploiement.
- Sans orchestrateur :
  - Ordonnancement manuel fragile.
  - Pas de mécanisme de reprise après échec.
  - Pas de logs centralisés.
  - Paramétrisation et réplication difficiles.
  - Aucun moyen de planifier (batch mensuel, quotidien, CI/CD).
- Un orchestrateur garantit un **pipeline fiable, traçable et automatisable**.

Qu’est-ce que Prefect ?

- **Prefect** est un orchestrateur orienté Python permettant de :
  - Définir des unités de travail (“tasks”) et des pipelines (“flows”).
  - Exécuter ces flows localement ou dans des environnements distribués.
  - Gérer logs, erreurs, états, dépendances et paramètres.
  - Superviser l’exécution via une interface ou des logs intégrés.
- Il remplace des scripts ad hoc par une **infrastructure de workflow robuste.**

Notre pipeline d'ingestion

- Nous allons partir des données brutes que nous insérerons dans une base de données relationnelle. Ensuite, nous validerons les tables et créerons des snapshots
- En détails :
    1. Chargement de CSV(s)
    2. Upsert (update + insert) dans les tables relationnelles
    3. Validation des tables avec Great Expectations
    4. Matérialisation des snapshots mensuels
    5. Log et orchestration avec Prefect

Tasks vs Flows

- **Task**
  - Fonction unitaire, atomique, réutilisable.
  - Exemples : lire un CSV, exécuter un upsert, valider une table.
- **Flow**
  - Structure globale orchestrant plusieurs tasks.
  - Définit l'ordre d'exécution, les dépendances, la logique métier.
- Un flow assemble plusieurs tasks pour produire un pipeline déterministe.

Logging et gestion des échecs dans Prefect

- Prefect enregistre automatiquement :
  - L'état de chaque task (success, failed, retry, cancelled).
  - Les logs détaillés (print, exceptions, stack trace).
  - Le temps d'exécution et les paramètres utilisés.
- En cas d'erreur :
  - Le flow s'arrête, sauf si des retries sont configurés.
  - L'erreur est reportée proprement dans les logs.
- Cela rend le debug beaucoup plus rapide que dans un script classique.

Exemple générique de task Prefect

```
from prefect import task

@task
def load_csv(path):
    import pandas as pd
    df = pd.read_csv(path)
    return df
```

Caractéristiques :

- Décorateur @task : instrumentation automatique.
- Retour du dataframe : passage possible entre tasks.
- Fonction pure recommandée pour la reproductibilité (pas d’appels extérieurs).

Exemple générique de flow Prefect

```python
from prefect import flow

@flow
def ingest_month(seed_dir, as_of):
    df = load_csv(f"{seed_dir}/users.csv")
    upsert_users(df)
    validate_users()
    create_snapshot(as_of)
```

Le flow orchestre les tasks et permet de rejouer intégralement une ingestion mensuelle.

Passage de paramètres à un flow

Les flows acceptent des paramètres dynamiques :

```
ingest_month(
    seed_dir="/data/seeds/month_000",
    as_of="2024-01-31"
)
```

Avantages :

- Rejouer n'importe quel mois passé.
- Automatiser une ingestion planifiée.
- Faciliter le debug (exécuter uniquement une date spécifique).

Les paramètres rendent la pipeline flexible et réutilisable.

Variables d’environnement et configuration

- Prefect s’intègre naturellement avec :
  - Les variables d’environnement (ex : DB_URL, SEED_DIR).
  - Les fichiers .env chargés par Docker.
  - Les secrets ou configurations externes (non utilisés dans ce cours mais standard en production).
- Objectif : **séparer le code de la configuration**, pour plus de robustesse.

Logique de retry

Les tasks peuvent être configurées pour réessayer automatiquement :

```
@task(retries=3, retry_delay_seconds=10)
```

Utile lorsque :

- La base de données met du temps à démarrer.
- Un réseau ou un disque est saturé temporairement.
- Une dépendance externe est momentanément indisponible.

Cela rend la pipeline plus résiliente.

Place de Prefect dans l'architecture complète

Dans notre système :

Ingestion → Validation → Snapshots → Feature Store → Training → Serving
(orchestration Prefect)

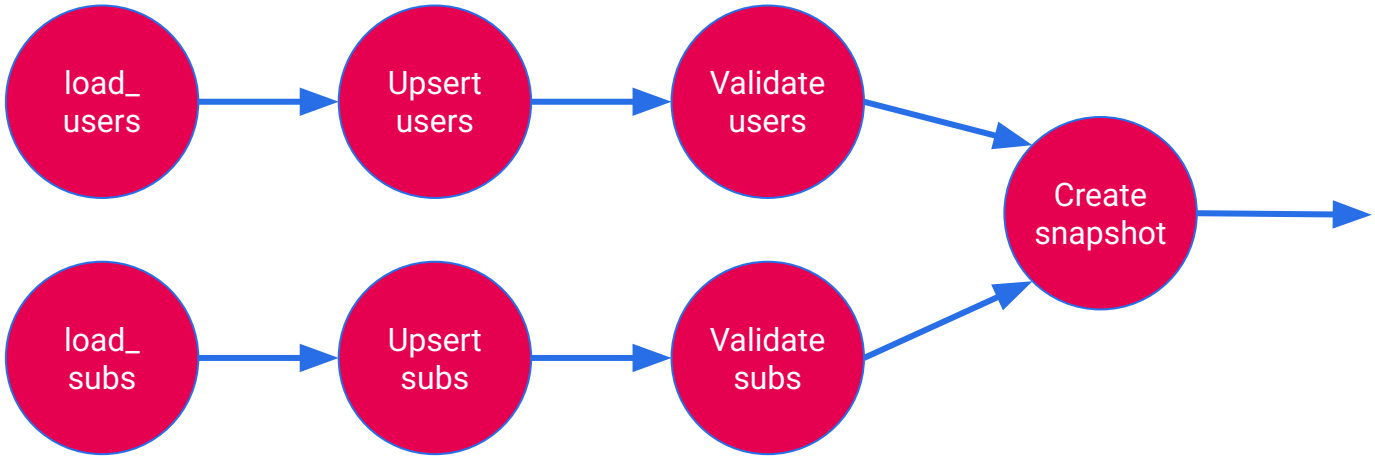
Prefect orchestre :

- Les flows d'ingestion (aujourd'hui).
- Les flows d'entraînement (cours suivants).
- Les éventuels flows de réentraînement / CI/CD.

Il est la "colonne vertébrale" du pipeline ML.

DAG de la pipeline d'ingestion

Les pipelines sont souvent représentées par des DAG (Directed Acyclic Graph), où chaque nœud représente un tâche et chaque arête une dépendance (un tâche ne peut s'exécuter tant que tout ses dépendances ne sont pas terminées)



Utilisation locale de Prefect dans Docker Compose

- Dans notre environnement :
  - Prefect s'exécute dans son propre conteneur.
  - Les flows Python sont présents dans ce conteneur.
  - Vous interagissez via :
    - `docker compose exec prefect python /opt/prefect/flows/ingest_flow.py`
- Pas besoin d'interface web : tout se passe dans les logs.

Comment vous lancerez la pipeline dans le TP

Dans le TP 2, vous exécuterez :

```
docker compose exec \
    -e SEED_DIR=/data/seeds/month_000 \
    -e AS_OF=2024-01-31 \
    prefect python /opt/prefect/flows/ingest_flow.py
```

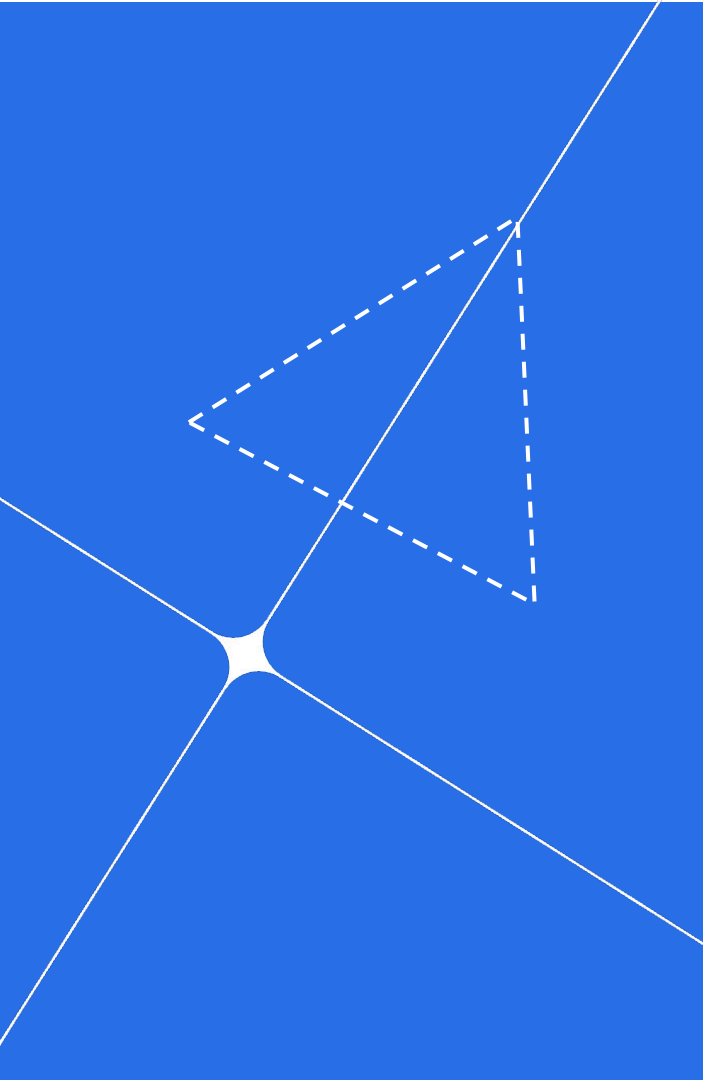
Le flow :

- Charge les CSV.
- Upsert dans PostgreSQL.
- Valide chaque table.
- Crée les snapshots.

Succès visible via les logs Prefect.

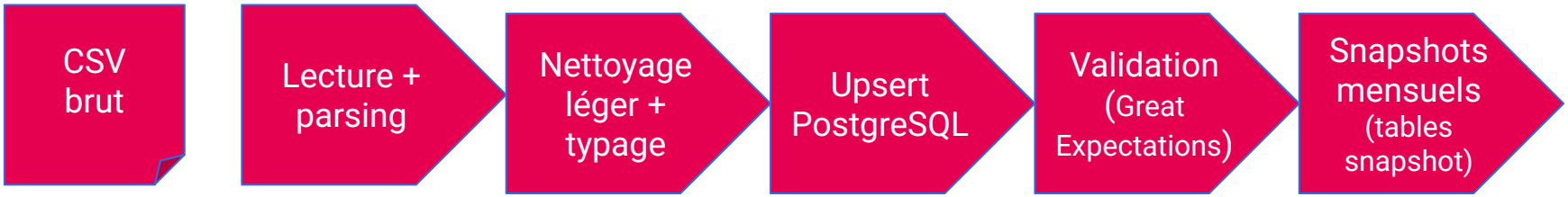
Déboguer les tasks Prefect

- Méthodes principales :
  - Lire les logs avec *docker compose logs -f prefect*.
  - Ajouter des logger.info() dans les tasks.
  - Interroger PostgreSQL pour vérifier l'état intermédiaire.
  - Exécuter uniquement certaines tasks dans un notebook/local pour reproduction.
  - Vérifier les erreurs liées aux fichiers montés dans Docker.
- Prefect fournit une visibilité très fine sur chaque étape.



Architecture d’ingestion de bout en bout

Vue d'ensemble : tout assembler



Séquence complète

1. **Lecture CSV** : collecte des données brutes pour le mois courant.
2. **Upsert** : insertion ou mise à jour dans les tables live via ON CONFLICT.
3. **Validation** : application des expectations définies (GE).
4. **Snapshots** : création de vues figées, indexées par *as_of*.

Chaque étape dépend de la précédente. Aucun snapshot n’est créé tant que la validation n’a pas réussi.

Ingestion mensuelle déterministe

- Pour garantir la reproductibilité :
  - Le même mois (*AS_OF*) produit toujours exactement le même snapshot.
  - Les réexécutions ne changent rien : upsert + validation = **idempotence**.
  - Les transformations sont explicites, versionnées dans le code.
  - Le pipeline ne dépend pas d'un état implicite (notebook, cache, variable globale).
- La déterminisme est la clé pour auditer les modèles et rejouer l'historique.

Ce qui change lorsque l'on ingère month_001, month_002…

- À chaque nouveau mois :
  - Les fichiers bruts contiennent de nouveaux utilisateurs, ou des mises à jour d'utilisateurs existants.
  - Les upserts modifient uniquement les lignes nécessaires dans les tables live.
  - Une nouvelle entrée snapshot est ajoutée, ex :
    - as_of = '2024-02-29' pour month_001
    - as_of = '2024-03-31' pour month_002
  - La taille de l'historique augmente progressivement.
- Le système construit ainsi une timeline complète des états successifs.

Le rôle des tables de métadonnées

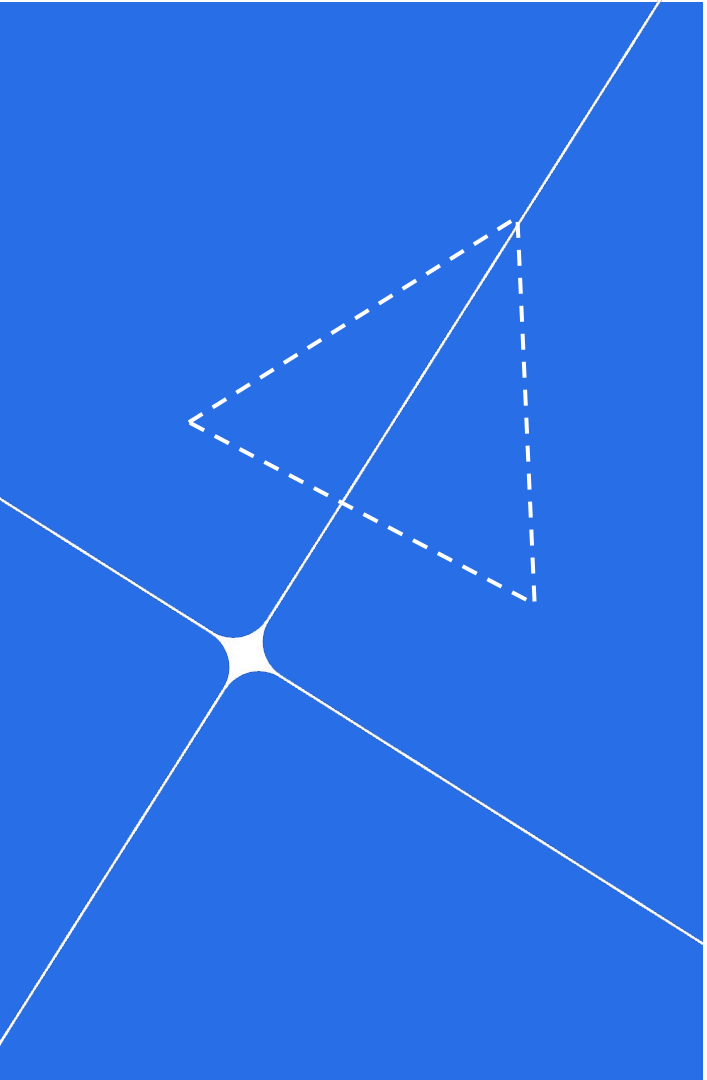
- Les pipelines ML industriels utilisent souvent des tables de métadonnées pour :
  - Enregistrer les dates d'ingestion (ingested_at).
  - Suivre l'état de validation : success / failure / warnings.
  - Documenter les volumes ingérés, les anomalies, les statistiques.
  - Garantir la traçabilité complète de chaque run.
- Dans notre cours, cet aspect reste simplifié, mais constitue un pilier des systèmes de production.

Comment cette étape prépare Feast (Cours 3)

- Feast dépend de snapshots cohérents pour :
  - Construire des tables de features alignées temporellement.
  - Garantir l'absence de data leakage entre features et labels.
  - Permettre le Historical Feature Retrieval, fondamental pour entraîner un modèle fiable.
  - Donner une définition unique et versionnée des features utilisées par l'API et par le training script.
- Sans ingestion + snapshots, un Feature Store est impossible à utiliser correctement.

Résumé du comportement du système d'ingestion

- Le système mis en place garantit :
  - Une ingestion fiable et idempotente des données mensuelles.
  - Une validation systématique de la qualité et du schéma.
  - La création de snapshots immuables, reconstruisibles et alignés temporellement.
  - Un pipeline orchestré, traçable, réexécutable à volonté (Prefect).
  - Une base solide pour la suite du cours : Feature Store, training, service API, monitoring.
- Cette architecture représente la fondation d'un système ML reproductible et industrialisable.



TP 2

Résumé du TP1

- À la fin du TP 1, vous disposez de :
  - Un environnement Docker Compose fonctionnel (PostgreSQL, API minimaliste).
  - La capacité à démarrer/arrêter les services et lire les logs.
  - Une compréhension des concepts essentiels : images, conteneurs, volumes, réseaux.
- Vous êtes maintenant prêts à construire un pipeline d’ingestion réel.

Ce que vous allez implémenter dans le TP 2

Dans le TP 2, vous allez coder les composantes clés suivantes (conceptuellement) :

1. Lecture et chargement des fichiers CSV d'un mois.
2. Upsert dans les tables PostgreSQL (idempotence).
3. Application des expectations (Great Expectations).
4. Création des snapshots mensuels avec as_of.
5. Orchestration Prefect d'un flow complet ingest_month.
6. Vérification des résultats via requêtes SQL + logs.

Ce TP assemble toutes les briques de l'ingestion MLOps.

Vérifier l'état de PostgreSQL avec des requêtes

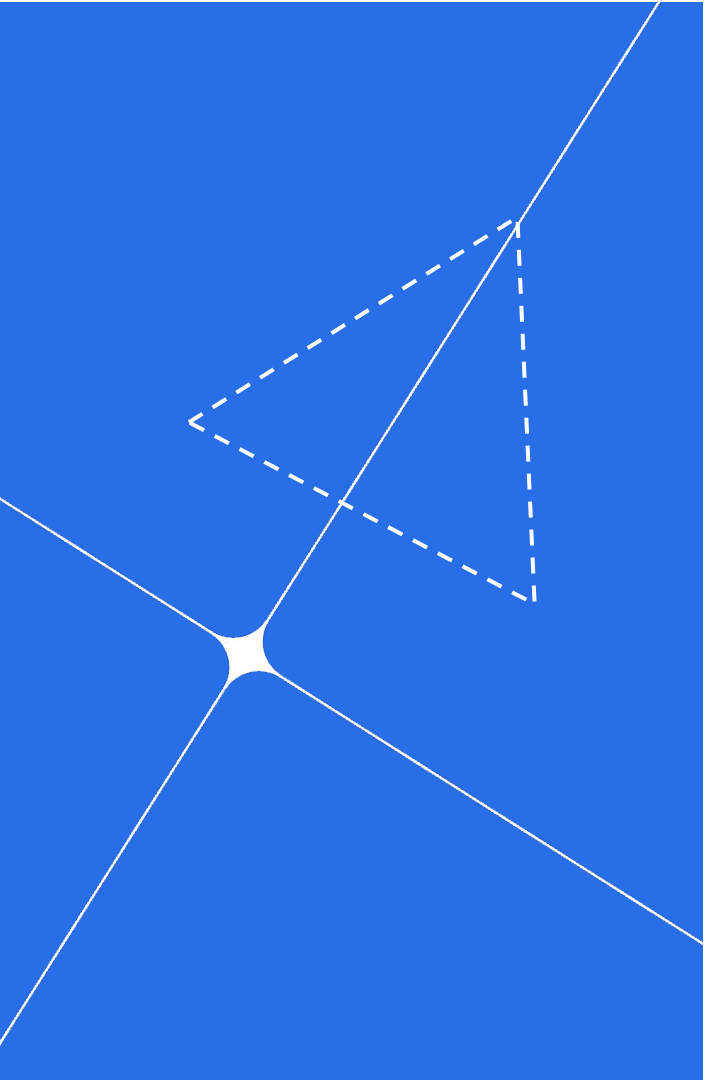
- Pour valider l'ingestion, utilisez :
  - docker compose exec db psql -U demo -d demo
- Requêtes utiles :
  - Compter les lignes ingérées :
    - SELECT COUNT(*) FROM subscriptions;
  - Vérifier un utilisateur particulier :
    - SELECT * FROM usage_agg_30d LIMIT 5;
  - Vérifier un snapshot :
    - SELECT * FROM subscriptions_snapshots WHERE as_of = '2024-01-31';
  - Comparer live vs snapshot :
    - EXCEPT, INTERSECT, ou simple inspection visuelle.
- La validation SQL fait partie intégrante du débogage.

Déboguer validation et création de snapshots

- Méthodes recommandées :
  - Inspecter les logs Prefect :
    - docker compose logs -f prefect
  - Exécuter le flow avec des print / logger.info() dans les tasks.
  - Vérifier les échecs GE (messages d'erreurs explicites).
  - Vérifier les paths montés dans Docker (erreur fréquente).
  - S'assurer que AS_OF est passé correctement à la task snapshot.
  - Rejouer le flow avec un sous-ensemble de tables pour isoler le problème.
- Le debug doit être systématique et basé sur des observations précises.

Résultat attendu à la fin du TP 2

- À la fin du TP, vous devez obtenir :
  - Toutes les tables live correctement alimentées pour month_000.
  - Un flow Prefect complet capable de :
    - charger les CSV,
    - upsert dans PostgreSQL,
    - valider les données (GE),
    - créer les snapshots.
  - Des snapshots cohérents contenant un champ as_of = '2024-01-31'.
  - Des logs Prefect indiquant un run sans échec.
- C’est le premier pipeline ML complet du cours.



En route pour le TP