CY Tech

ING3 / IA Groupe B

Année 2025–2026

Big Data

Rapport de projet

Sujet : Projet Big Data 2026
Encadrant : Rakib SHEIKH

Auteurs :
Younes BENABDELLAH
Simon CHANTHRABOUTH-LIEBBE
David DEL CAMPO
Enzo EDMOND

Date : 07 février 2026

Table des matières

1 Introduction

L'objectif de ce projet est de déployer une architecture Big Data complète, similaire à celles que l'on retrouve dans les entreprises du CAC 40, afin de se familiariser avec les différentes briques d'une infrastructure de données moderne. Le cas d'usage porte sur les données des taxis jaunes de la ville de New York, publiées mensuellement par la *Taxi and Limousine Commission* (TLC).

Le pipeline mis en place couvre l'ensemble du cycle de vie de la donnée :

1. **Collecte et stockage** dans un Data Lake (MinIO) ;
2. **Nettoyage et ingestion** multi-branche via Apache Spark ;
3. **Modélisation dimensionnelle** dans un Data Warehouse (PostgreSQL) ;
4. **Visualisation** des données via un tableau de bord Streamlit ;
5. **Prédiction par Machine Learning** du tarif total d'une course via LightGBM ;
6. **Orchestration** de l'ensemble du pipeline via Apache Airflow.

Ce rapport décrit les actions menées pour chacun de ces exercices, les choix techniques et architecturaux effectués, ainsi que les résultats obtenus.

2 Architecture globale

L'ensemble de l'infrastructure est orchestrée par **Docker Compose** et comprend les services suivants :

Table 1 – Services déployés via Docker Compose

Service	Port	Rôle
Spark Master	8081, 7077	Coordinateur du cluster Spark
Spark Worker 1	–	Nœud de calcul (2 Go, 2 cœurs)
Spark Worker 2	–	Nœud de calcul (2 Go, 2 cœurs)
MinIO	9000, 9002	Stockage objet S3-compatible (Data Lake)
PostgreSQL 16	5432	Base de données (Data Warehouse)
pgAdmin 4	5050	Interface d'administration PostgreSQL
Airflow 2.10	8085	Orchestrateur de pipelines (DAG)

La pile technologique utilisée est la suivante :

— **Scala 2.13 + Apache Spark 3.5.5** pour le traitement distribué (exercices 1 et 2) ;
— **SQL (PostgreSQL)** pour la modélisation dimensionnelle (exercice 3) ;
— **Python + Streamlit** pour la visualisation (exercice 4) ;
— **Python + LightGBM** pour le Machine Learning (exercice 5) ;
— **Apache Airflow** pour l'orchestration du pipeline complet (exercice 6) ;
— **uv** comme gestionnaire d'environnements virtuels Python (imposé par le sujet).

3 Exercice 1 : Collecte des données et intégration

3.1 Objectif

Récupérer le fichier Parquet mensuel des yellow taxis de NYC depuis le site de la TLC et le stocker dans le Data Lake MinIO (bucket `nyc-raw`).

3.2 Démarche

Le programme Scala (`Main.scala`) propose deux approches :

1. **Upload direct vers MinIO** (`uploadDirectToMinio`) : le fichier Parquet est téléchargé depuis l'URL officielle [1] et streamé directement vers MinIO via le client Java MinIO. Cette méthode conserve le fichier original tel quel dans le bucket.
2. **Upload via Spark** (`uploadViaSparkLocal`) : le fichier est d'abord téléchargé en local dans `data/raw/`, puis lu et réécrit par Spark vers MinIO au format dataset Parquet (répertoire avec des fichiers `part-*.parquet`).

La méthode 1 a été retenue comme méthode principale car elle est plus directe et préserve le fichier original. La fonction `ensureBucket` crée automatiquement le bucket s'il n'existe pas.

Les paramètres de connexion MinIO sont configurables par **variables d'environnement** (`MINIO_ENDPOINT`, `MINIO_ACCESS_KEY`, `MINIO_SECRET_KEY`) avec des valeurs par défaut pour l'exécution locale. Cette approche permet au même code de fonctionner en local et dans le conteneur Airflow sans modification.

3.3 Configuration MinIO

L'accès à MinIO est configuré avec les identifiants par défaut du `docker-compose.yml` :

```
endpoint  = http://localhost:9000
accessKey = minio
secretKey = minio123
bucket    = nyc-raw
```

Un script shell alternatif utilisant le client `mc` (MinIO Client) est également disponible :

```
mc alias set local http://localhost:9000 minio minio123
mc mb local/nyc-raw
mc cp "data/raw/yellow_tripdata_2025-05.parquet" local/nyc-raw/
```

1. https://d37ci6vzurychx.cloudfront.net/trip-data/yellow_tripdata_2025-05.parquet

4 Exercice 2 : Nettoyage et ingestion multi-branche

4.1 Objectif

Nettoyer les données brutes et les ingérer selon deux branches :

— **Branche 1** : produire un Parquet nettoyé pour le ML (bucket `nyc-clean`) ;

— **Branche 2** : charger les données dans le schéma dimensionnel PostgreSQL.

4.2 Branche 1 : Nettoyage et production du Parquet clean

Le DataFrame brut est chargé depuis MinIO puis filtré selon un contrat de validation strict basé sur le *data dictionary* de la TLC :

TABLE 2 – Règles de validation appliquées

Colonne	Règle
VendorID	$\in \{1, 2, 6, 7\}$
Datetimes	Non null, dropoff $\geq$ pickup
passenger_count	≥ 1
trip_distance	> 0
RatecodeID	$\in \{1, 2, 3, 4, 5, 6\}$
store_and_fwd_flag	$\in \{\text{Y}, \text{N}\}$
PU/DOLocationID	Non null, > 0
payment_type	$\in \{0, 1, 2, 3, 4, 5, 6\}$
total_amount	≥ 0

Résultat : sur les **4 591 845** lignes du fichier brut, **3 211 616** lignes passent la validation, soit un taux de conservation de 69,9 %.

Le bucket `nyc-clean` est créé automatiquement par la fonction `ensureBucket` si nécessaire, puis le Parquet nettoyé y est écrit. Comme pour l'exercice 1, toute la configuration (MinIO, PostgreSQL, chemins) est pilotée par variables d'environnement avec des valeurs par défaut locales, assurant la portabilité vers Airflow.

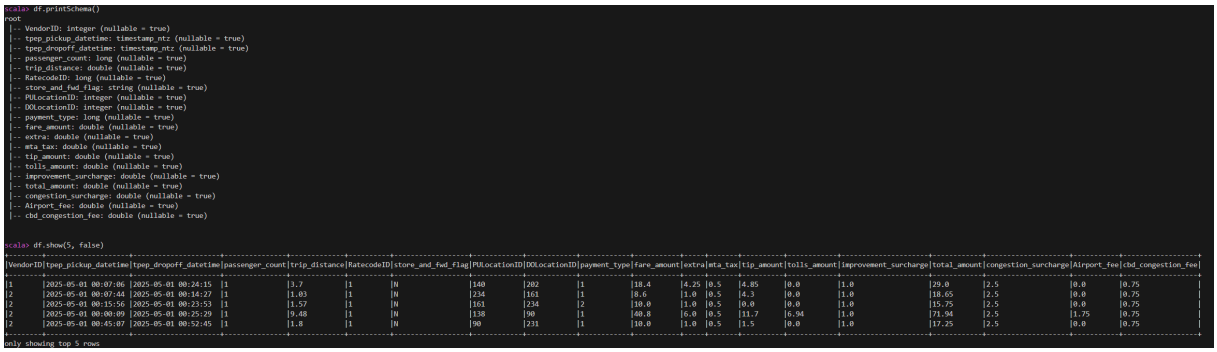
4.3 Branche 2 : Chargement dans PostgreSQL

Une fois le Parquet nettoyé, le même job Spark charge les dimensions manquantes et la table de faits dans PostgreSQL via JDBC :

— **dim_date** : extraite de `tpep_pickup_datetime` et `tpep_dropoff_datetime` au format `yyyyMMdd` avec extraction de l'année, du mois, du jour et du jour de la semaine.

— **dim_location** : construite à partir des `PULocationID`/`DOLocationID` et enrichie avec le fichier `taxi_zone_lookup.csv` de la TLC (borough, zone, service_zone).

— **fact_trip** : table de faits contenant les clés étrangères vers les dimensions et les mesures (montants, distances, nombre de passagers).



```
scala> df.printSchema()
root
 |-- VendorID: integer (nullable = true)
 |-- tpep_pickup_datetime: timestamp_ntz (nullable = true)
 |-- tpep_dropoff_datetime: timestamp_ntz (nullable = true)
 |-- passenger_count: long (nullable = true)
 |-- trip_distance: double (nullable = true)
 |-- RatecodeID: long (nullable = true)
 |-- store_and_fwd_flag: string (nullable = true)
 |-- PULocationID: integer (nullable = true)
 |-- DOLocationID: integer (nullable = true)
 |-- payment_type: long (nullable = true)
 |-- fare_amount: double (nullable = true)
 |-- extra: double (nullable = true)
 |-- mta_tax: double (nullable = true)
 |-- tip_amount: double (nullable = true)
 |-- tolls_amount: double (nullable = true)
 |-- improvement_surcharge: double (nullable = true)
 |-- total_amount: double (nullable = true)
 |-- congestion_surcharge: double (nullable = true)
 |-- Airport_fee: double (nullable = true)
 |-- cbd_congestion_fee: double (nullable = true)


scala> df.show(5, false)
+--------+--------------------+---------------------+---------------+-------------+----------+------------------+------------+------------+------------+-----------+-----+-------+----------+------------+---------------------+------------+--------------------+-----------+--------------------+
|VendorID|tpep_pickup_datetime|tpep_dropoff_datetime|passenger_count|trip_distance|RatecodeID|store_and_fwd_flag|PULocationID|DOLocationID|payment_type|fare_amount|extra|mta_tax|tip_amount|tolls_amount|improvement_surcharge|total_amount|congestion_surcharge|Airport_fee|cbd_congestion_fee|
+--------+--------------------+---------------------+---------------+-------------+----------+------------------+------------+------------+------------+-----------+-----+-------+----------+------------+---------------------+------------+--------------------+-----------+--------------------+
|1       |2025-05-01 00:07:06 |2025-05-01 00:24:15  |1              |3.7          |1         |N                 |140         |202         |1           |18.4       |4.25 |0.5    |4.85      |0.0         |1.0                  |29.0        |2.5                 |0.0        |0.75                |
|2       |2025-05-01 00:07:44 |2025-05-01 00:14:27  |1              |1.03         |1         |N                 |234         |161         |1           |8.6        |1.0  |0.5    |4.3       |0.0         |1.0                  |18.65       |2.5                 |0.0        |0.75                |
|2       |2025-05-01 00:15:56 |2025-05-01 00:23:53  |1              |1.57         |1         |N                 |161         |234         |2           |10.0       |1.0  |0.5    |0.0       |0.0         |1.0                  |15.75       |2.5                 |0.0        |0.75                |
|2       |2025-05-01 00:00:09 |2025-05-01 00:25:29  |1              |9.48         |1         |N                 |138         |90          |1           |40.8       |6.0  |0.5    |11.7      |6.94        |1.0                  |71.94       |2.5                 |1.75       |0.75                |
|2       |2025-05-01 00:45:07 |2025-05-01 00:52:45  |1              |1.8          |1         |N                 |90          |231         |1           |10.0       |1.0  |0.5    |1.5       |0.0         |1.0                  |17.25       |2.5                 |0.0        |0.75                |
+--------+--------------------+---------------------+---------------+-------------+----------+------------------+------------+------------+------------+-----------+-----+-------+----------+------------+---------------------+------------+--------------------+-----------+--------------------+
only showing top 5 rows
```

FIGURE 1 – Schéma du fichier Parquet brut – `df.printSchema()` et `df.show(5)` dans le spark-shell

5 Exercice 3 : Data Warehouse et modèle dimensionnel

5.1 Contexte

Le schéma initial est de type OLTP, optimisé pour l'insertion rapide de transactions. Pour permettre des analyses métier (agrégation, drill-down par période, zone, type de paiement, etc.), une transformation vers un modèle OLAP est nécessaire.

5.2 Choix du modèle en étoile

Le **modèle en étoile** a été retenu pour les raisons suivantes :

— **Simplicité** : structure facile à comprendre et à requêter ;

— **Performance** : les jointures sont directes (fait ↔ dimension) ;

— **Adéquation** : les dimensions identifiées sont déjà plates (pas de hiérarchies complexes nécessitant un modèle en flocon).

5.3 Définition du grain

1 ligne = 1 course de taxi. Chaque enregistrement dans `fact_trip` représente une course individuelle avec ses mesures associées.

5.4 Schéma du modèle

Le modèle comprend **6 tables de dimensions** et **1 table de faits** :

Table 3 – Tables du modèle en étoile

Table	Type	Description
dim_vendor	Dimension	Fournisseurs TPEP (Creative Mobile, Curb, etc.)
dim_ratecode	Dimension	Types de tarification (standard, JFK, Newark, etc.)
dim_payment	Dimension	Modes de paiement (CB, cash, etc.)
dim_store_and_fwd	Dimension	Indicateur store-and-forward
dim_date	Dimension	Dates avec année, mois, jour, jour de semaine
dim_location	Dimension	Zones géographiques (borough, zone, service_zone)
fact_trip	Fait	Mesures : montants, distance, nb passagers + FK

5.5 Scripts SQL

Deux scripts SQL sont fournis :

— **creation.sql** : suppression puis création de toutes les tables avec contraintes de clés primaires et étrangères ;

— **insertion.sql** : insertion des données de référence (vendors, ratecodes, payment types, store_and_fwd flags) fournies par le métier.

Les dimensions `dim_date` et `dim_location` ainsi que la table de faits `fact_trip` sont chargées dynamiquement par le job Spark de l'exercice 2 (branche 2).

6 Exercice 4 : Visualisation des données

6.1 Objectif

Connecter un outil de visualisation à la base PostgreSQL et produire un tableau de bord présentant les indicateurs clés et les tendances des courses de taxi.

6.2 Approche

L'exercice a été réalisé en deux temps :

1. Un **notebook Jupyter** (`ex04_eda.ipynb`) pour l'analyse exploratoire des données (EDA), avec des requêtes SQL via SQLAlchemy et des visualisations Matplotlib/Seaborn.
2. Un **tableau de bord Streamlit** (`app.py`) pour une présentation interactive.

La connexion à PostgreSQL est réalisée via SQLAlchemy :

```
from sqlalchemy import create_engine
engine = create_engine(
    "postgresql://bigdata:bigdata123@localhost:5432/bigdata_dwh"
)
```

6.3 Indicateurs clés (KPI)

Le dashboard présente les KPI suivants, calculés sur les données de mai 2025 :

TABLE 4 – KPI du tableau de bord

Indicateur	Valeur
Nombre total de courses	3 211 616
Revenu total	96 543 422,61 $
Tarif moyen	30,06 $
Distance moyenne	3,49 miles
Taux de pourboire	12,6 %

6.4 Visualisations

Le dashboard est structuré en trois sections :

— **Volume et revenu** : nombre de courses par jour de la semaine (bar chart) + revenu par heure (line chart). On observe un pic d'activité le jeudi et une courbe de revenu qui suit le rythme de la journée avec un creux entre 2h et 5h du matin.

— **Paiement et pourboires** : tarif moyen par type de paiement + taux de pourboire par vendor.

— **Top 10 des zones de pickup** : tableau et bar chart des zones les plus fréquentées. Manhattan domine largement avec Upper East Side South, Midtown Center et Upper East Side North dans le top.

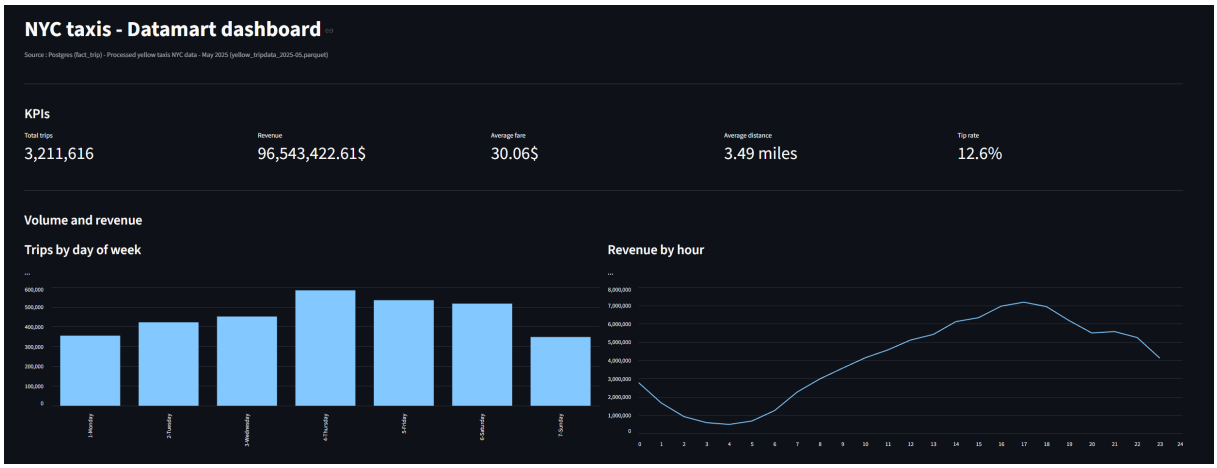


FIGURE 2 – Dashboard Streamlit – KPI, courses par jour, revenu par heure

FIGURE 3 – Dashboard Streamlit – Tarif par type de paiement, pourboire par vendor, top 10 zones

7 Exercice 5 : Prédiction par Machine Learning

7.1 Objectif

Concevoir un modèle de Machine Learning capable de prédire le `total_amount` (prix total) d'une course de taxi à partir des features disponibles, avec un RMSE inférieur à 10.

7.2 Architecture du code

Le code est organisé en modules Python conformes aux bonnes pratiques (PEP 8, documentation NumpyDoc, tests unitaires) :

Table 5 – Organisation des modules ML

Module	Rôle
`config.py`	Constantes (MinIO, features, target, hyperparamètres)
`data_loader.py`	Chargement des données Parquet depuis MinIO via boto3
`feature_engineering.py`	Transformation des features + validation anti-leakage
`train.py`	Pipeline d'entraînement complet
`predict.py`	Inférence batch ou single

7.3 Sélection des features

Les features sélectionnées sont divisées en deux catégories :

Features brutes issues du Parquet nettoyé :

— `trip_distance` : distance de la course
— `PULocationID`, `DOLocationID` : zones de départ et d'arrivée
— `passenger_count` : nombre de passagers
— `VendorID`, `RatecodeID`, `payment_type` : variables catégorielles
— `store_and_fwd_flag` : encodé en 0/1

Features dérivées extraites de `tpep_pickup_datetime` :

— `pickup_hour` : heure du pickup (0–23)
— `pickup_dayofweek` : jour de la semaine (0=lundi, 6=dimanche)
— `pickup_is_weekend` : indicateur weekend (0/1)

7.4 Prévention du data leakage

Les colonnes composantes du `total_amount` (`fare_amount`, `extra`, `mta_tax`, `tip_amount`, `tolls_amount`, `improvement_surcharge`, `congestion_surcharge`, `Airport_fee`, `cbd_congestion_fee`) sont explicitement interdites comme features. Un mécanisme de validation (`validate_no_leakage`) lève une erreur si l'une de ces colonnes est détectée dans la matrice de features.

7.5 Modèle

Le modèle retenu est un **LightGBM Regressor** avec les hyperparamètres suivants :

Table 6 – Hyperparamètres du modèle LightGBM

Paramètre	**Valeur**
n_estimators	500
learning_rate	0,05
max_depth	8
num_leaves	63
min_child_samples	50
subsample	0,8
colsample_bytree	0,8
reg_alpha	0,1
reg_lambda	0,1

Le split train/test est de 80/20 (`random_state=42`), soit **2 569 292** lignes d'entraînement et **642 324** lignes de test.

7.6 Résultats

Table 7 – Métriques d'évaluation du modèle

Métrique	**Valeur**	**Interprétation**
RMSE	**5,42**	Bien en dessous du seuil de 10
MAE	**2,69**	Erreur moyenne de 2,69 $ par course
R^2	**0,949**	Le modèle explique 94,9 % de la variance

Le modèle respecte largement la contrainte du sujet (RMSE < 10). Le choix du RMSE comme métrique principale est justifié par le fait qu'il pénalise davantage les erreurs importantes, ce qui est pertinent pour des prédictions de tarif où des écarts importants pourraient avoir un impact métier significatif.

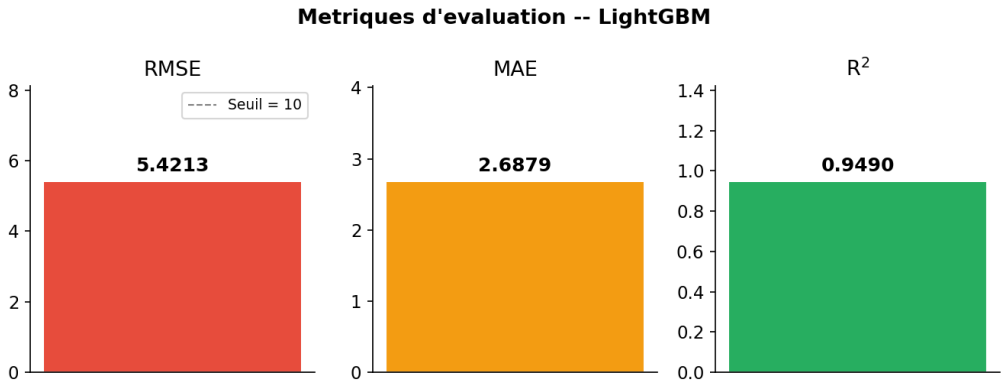


FIGURE 4 – Métriques d'évaluation du modèle LightGBM

7.7 Importance des features

L'analyse de l'importance des features (nombre de splits) révèle que les variables géographiques (`DOLocationID`, `PULocationID`) et la distance dominent, ce qui est cohérent : le tarif dépend principalement de la zone et de la distance parcourue. L'heure de pickup est également significative, reflétant les surcharges horaires (heures de pointe, nuit).

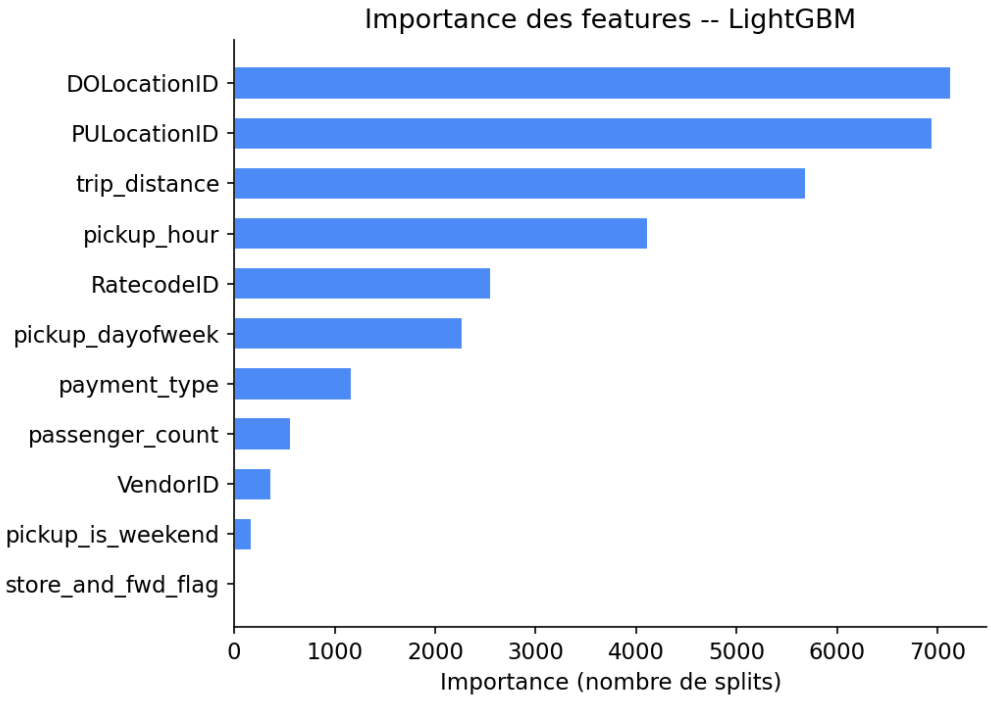


FIGURE 5 – Importance des features dans le modèle LightGBM

7.8 Analyse des prédictions

Le graphique de prédictions vs. valeurs réelles montre que le modèle suit bien la diagonale $y = x$, avec une dispersion modérée pour les tarifs élevés. La majorité des courses (tarifs entre 10 \$ et 40 \$) sont prédites avec précision.

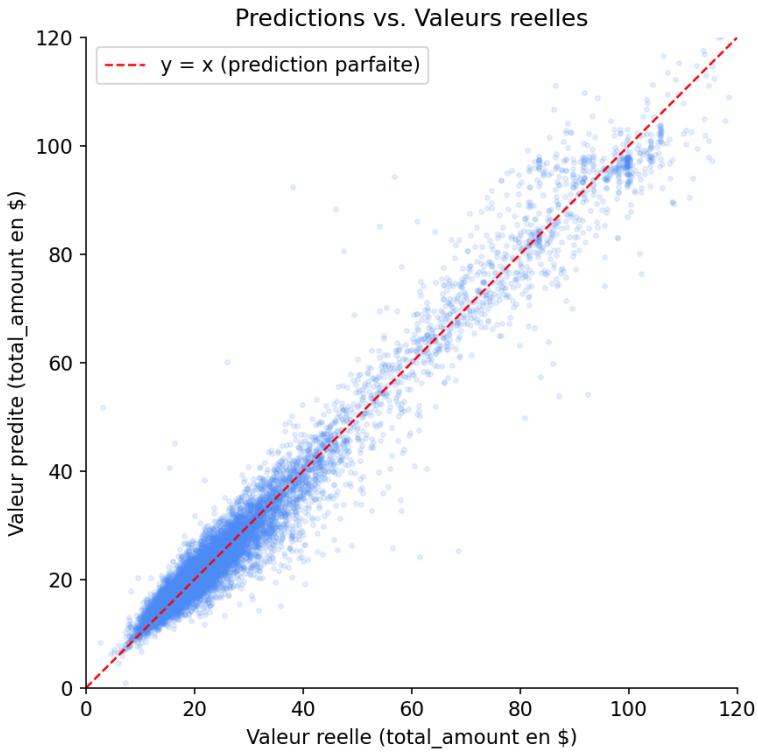


FIGURE 6 – Prédictions vs. valeurs réelles

La distribution des résidus est centrée autour de zéro (moyenne $\approx 0,00\,\$$) avec une légère asymétrie droite, indiquant que le modèle sous-estime légèrement les tarifs élevés. Cela s'explique par la présence de courses atypiques (courses longues vers les aéroports, tarifs négociés) qui représentent des cas rares dans le jeu de données.

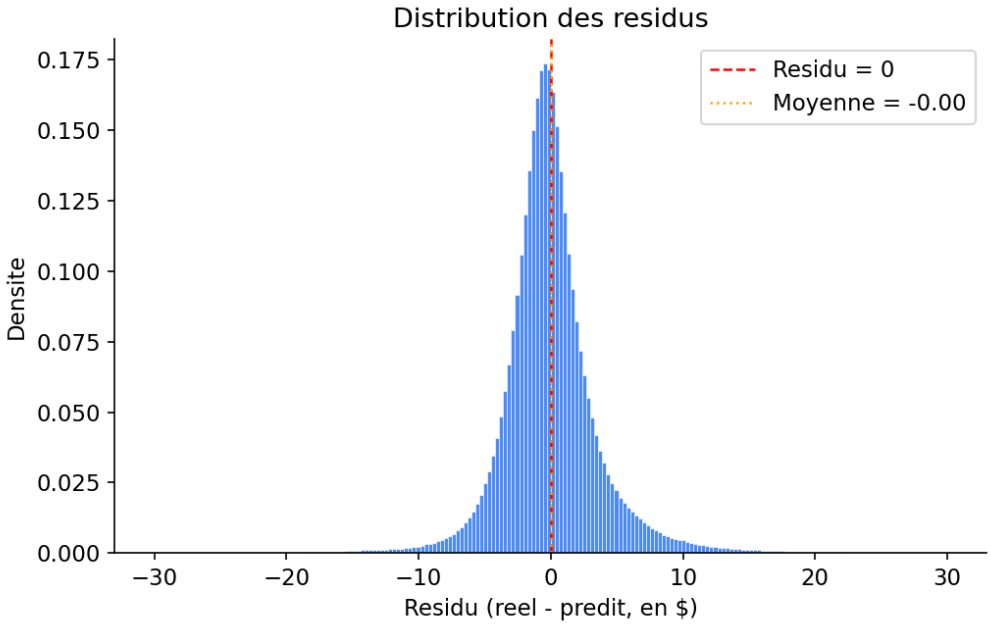


FIGURE 7 – Distribution des résidus

7.9 Tests unitaires

Des tests unitaires sont implémentés pour valider le pipeline :

— **Validation des données** (`test_data_validation.py`) : vérifie la présence des colonnes requises, l'absence de valeurs nulles, les domaines de validité (VendorID, RatecodeID, distance positive, etc.).

— **Validation des features** (`test_features.py`) : vérifie que les features temporelles sont dans les bornes attendues, que l'encodage est correct, et surtout que les colonnes de leakage ne sont jamais utilisées.

8 Exercice 6 : Orchestration avec Apache Airflow

8.1 Objectif

Automatiser l'ensemble du pipeline Big Data (exercices 1 à 5) sous la forme d'un **DAG Airflow** (*Directed Acyclic Graph*), afin d'exécuter toute la chaîne depuis l'interface web d'Airflow.

8.2 Infrastructure

Un service Airflow a été ajouté au `docker-compose.yml` :

— **Image** : basée sur `apache/airflow:2.10.2`, étendue via un Dockerfile personnalisé (`docker/airflow.Doc`

— **Executor** : `SequentialExecutor` avec une base SQLite intégrée (suffisant pour un pipeline séquentiel).

— **Port** : 8085 (interface web).

— **Volumes** : le workspace complet est monté dans `/workspace` pour accéder aux JARs, scripts SQL et code Python.

Le Dockerfile personnalisé installe les dépendances nécessaires à l'exécution de toutes les étapes :

Listing 1 – Contenu du airflow.Dockerfile

```
FROM apache/airflow:2.10.2
USER root
RUN apt-get update && apt-get install -y \
    openjdk-17-jdk postgresql-client libgomp1 sbt ...
USER airflow
RUN pip install pandas numpy pyarrow s3fs \
    scikit-learn lightgbm joblib
```

Cette image contient à la fois **Java 17** (exécution des JARs Spark), **psql** (scripts SQL) et les **bibliothèques Python ML** (entraînement LightGBM).

8.3 Portabilité du code : variables d'environnement et JARs

Pour que le même code fonctionne en local (développement) et dans le conteneur Airflow (orchestration), deux adaptations ont été nécessaires :

1. **Variables d'environnement** : les paramètres de connexion (MinIO, PostgreSQL, chemins de fichiers) utilisent `sys.env.getOrElse("VAR", "default")` en Scala (et `os.getenv` côté Python). En local, les valeurs par défaut (`localhost`) s'appliquent. Dans Airflow, le DAG injecte les valeurs Docker (`minio`, `postgres`).
2. **Fat JARs** : les exercices 1 et 2 sont compilés en JARs autonomes via `sbt assembly`, incluant leurs dépendances. Airflow les exécute ensuite directement avec `java -jar`.

Listing 2 – Génération des fat JARs

```
cd ex01_data_retrieval && sbt clean assembly
cd ex02_data_ingestion && sbt clean assembly
```

Les JARs produits sont accessibles par Airflow via le bind mount du workspace :

— `ex01_data_retrieval/target/scala-2.13/ex01-retrieval-assembly.jar`

— `ex02_data_ingestion/target/scala-2.13/ex02-ingestion-assembly.jar`

8.4 Structure du DAG

Le DAG `tp_bigdata_pipeline` enchaîne **6 tâches** séquentielles via des `BashOperator` :

TABLE 8 – Tâches du DAG Airflow

Ordre	Task ID	Action
1	`wait_services`	Attend que MinIO et PostgreSQL soient prêts (health checks)
2	`ex03_init_schema`	Exécute `creation.sql` via `psql`
3	`ex03_seed_dimensions`	Exécute `insertion.sql` via `psql`
4	`ex01_retrieval`	Télécharge le Parquet NYC et l'envoie dans MinIO (JAR)
5	`ex02_ingestion`	Nettoyage + ingestion multi-branche (JAR)
6	`ex05_training`	Entraîne le modèle LightGBM (Python)

L'ordre d'exécution respecte les dépendances entre exercices :

`wait_services` → `ex03_init_schema` → `ex03_seed_dimensions`
→ `ex01_retrieval` → `ex02_ingestion` → `ex05_training`

Le schéma PostgreSQL est initialisé en amont pour garantir que l'ingestion (ex02) puisse insérer les données dans les tables du datamart.

8.5 Injection des variables d'environnement

Chaque tâche du DAG injecte les variables d'environnement adaptées au réseau Docker :

Listing 3 – Variables d'environnement dans le DAG

```
MINIO_ENV = {
    "MINIO_ENDPOINT": "http://minio:9000",
    "MINIO_ACCESS_KEY": "minio",
    "MINIO_SECRET_KEY": "minio123",
}
PG_ENV = {
    "PG_HOST": "postgres",
    "PG_PORT": "5432",
    "PG_DATABASE": "bigdata_dwh",
    "PG_USER": "bigdata",
    "PG_PASSWORD": "bigdata123",
}
```

Les hôtes `minio` et `postgres` correspondent aux noms des services Docker, résolus automatiquement sur le réseau `spark-network`.

8.6 Difficultés rencontrées

— **Permissions fichiers** : le conteneur Airflow s'exécute avec un utilisateur dédié. Des ajustements de permissions (`chown`, `chmod`) ont été nécessaires sur les répertoires montés (`dags/`, `logs/`, `plugins/`) ainsi que sur les répertoires de sortie des modèles.

— **Accès Java** : l'exécution des JARs Spark avec Java 17 a nécessité les options `--add-exports=java.base/` (et `sun.util.calendar` pour l'ingestion), conformément à la configuration attendue.

— **Adressage réseau en conteneur** : dans Airflow, `localhost` ne pointe pas vers MinIO. Les paramètres ont donc été externalisés via variables d'environnement (ex. `MINIO_ENDPOINT=http://min`

— **Ressources** : l'exécution complète du pipeline dans Docker a nécessité d'augmenter la RAM et le swap alloués à WSL/Docker pour éviter les erreurs de type `exit code 137`.

— **Approche initiale vs finale** : une tentative initiale de compilation `sbt` dans le conteneur Airflow a été abandonnée (problèmes de permissions et de stabilité). La solution retenue est l'assemblage local des fat JARs (script `scripts/build_jars.sh`), puis leur exécution dans le DAG.

8.7 Résultat de l'exécution

La figure 8 présente l'interface web d'Airflow après une exécution complète du DAG `tp_bigdata_pipeline`. Les six tâches du pipeline (`wait_services`, `ex03_init_schema`, `ex03_seed_dimensions`, `ex01_retrieval`, `ex02_ingestion`, `ex05_training`) apparaissent en vert dans la vue *Graph*, ce qui confirme l'exécution correcte de bout en bout. Sur l'exécution présentée, `ex01_retrieval` (téléchargement du Parquet TLC et envoi vers MinIO) est l'une des étapes les plus coûteuses, suivie de `ex02_ingestion` (traitements Spark et chargement du datamart PostgreSQL), tandis que les tâches SQL de l'exercice 3 restent très rapides.

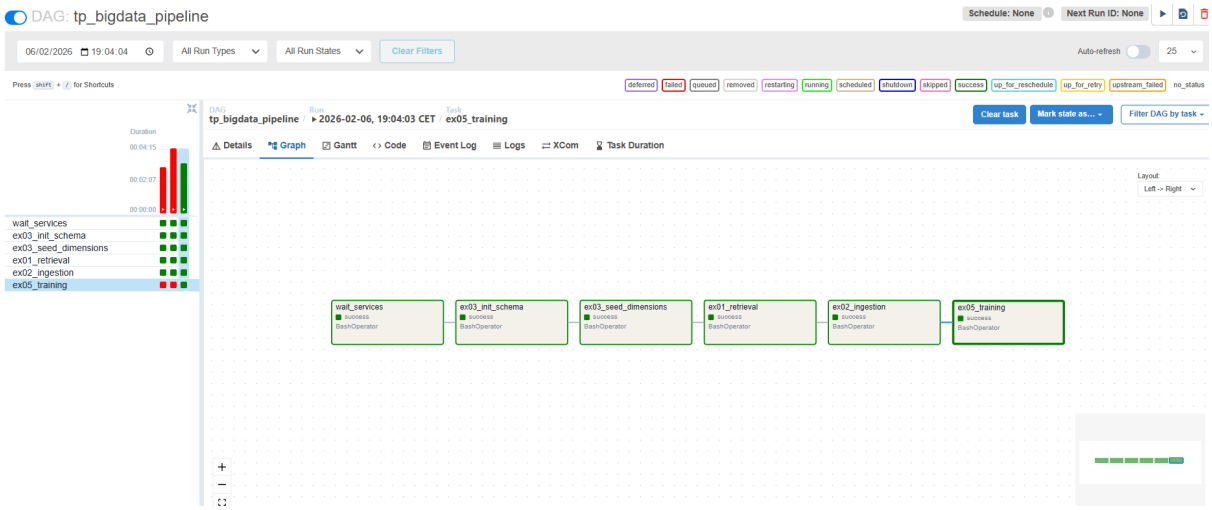


FIGURE 8 – Interface Airflow – Vue *Graph* du DAG `tp_bigdata_pipeline` après exécution réussie

9 Conclusion

Ce projet a permis de déployer une architecture Big Data complète couvrant l'ensemble du cycle de vie des données, depuis la collecte brute jusqu'à la prédiction par Machine Learning.

Synthèse des réalisations :

- **Exercice 1** : collecte automatisée des données Parquet et stockage dans le Data Lake MinIO via un programme Scala.
- **Exercice 2** : pipeline d'ingestion multi-branche avec nettoyage (de 4,6M à 3,2M de lignes) et chargement vers MinIO (branche ML) et PostgreSQL (branche analytique).
- **Exercice 3** : modélisation en étoile avec 6 dimensions et 1 table de faits, implémentée en SQL avec contraintes référentielles.
- **Exercice 4** : tableau de bord Streamlit connecté à PostgreSQL, affichant KPI, tendances temporelles et zones les plus actives.
- **Exercice 5** : modèle LightGBM atteignant un RMSE de 5,42 (objectif < 10) et un R^2 de 0,949, avec prévention du data leakage et tests unitaires.
- **Exercice 6** : orchestration du pipeline via un DAG Airflow, avec portabilité assurée par les variables d'environnement et l'assemblage local des fat JARs (`scripts/build_jars.sh`).

L'architecture Docker Compose (Spark, MinIO, PostgreSQL, pgAdmin, Airflow) fournit un environnement reproductible et portable, permettant à chaque membre de l'équipe de travailler sur une base commune. L'ajout d'Airflow permet de rejouer l'intégralité du pipeline en un seul lancement depuis l'interface web.

Références

[1] NYC Taxi and Limousine Commission, "TLC Trip Record Data," 2025. Disponible : https://www.nyc.gov/site/tlc/about/tlc-trip-record-data.page.

[2] Apache Software Foundation, "Apache Spark," 2025. Disponible : https://spark.apache.org/.

[3] MinIO Inc., "MinIO : High Performance Object Storage," 2025. Disponible : https://min.io/.

[4] G. Ke et al., "LightGBM : A Highly Efficient Gradient Boosting Decision Tree," *Advances in Neural Information Processing Systems*, 2017.

[5] Streamlit Inc., "Streamlit : The fastest way to build data apps," 2025. Disponible : https://streamlit.io/.