Mastère Spécialisé Big Data

Télécom Paris – 2025/2026

# Pipeline Big Data pour l'Analyse et la Prédiction de Matchs NBA

Grégoire Petit Benjamin Lepourtois

Février 2026

**Résumé**

Ce rapport présente la conception et l'implémentation d'un pipeline big data de bout en bout pour l'analyse de données NBA. Le système ingère des données depuis deux APIs publiques, les stocke dans un data lake MinIO organisé en couches (raw, formatted, combined), les transforme via Apache Spark, entraîne un modèle de Random Forest pour prédire les victoires à domicile, et expose les résultats dans des dashboards Kibana via Elasticsearch. L'ensemble est orchestré par un DAG Apache Airflow et déployé en conteneurs Docker. Le pipeline calcule plus de 15 KPIs par équipe et par match, incluant des métriques de forme récente et de repos entre les matchs.

# Table des matières

**1 Introduction** **3**

**2 Architecture globale** **3**
2.1 Stack technique . . . 3
2.2 Schéma d'architecture . . . 3
2.3 Structure du data lake (MinIO) . . . 4

**3 Ingestion des données** **4**
3.1 Sources de données . . . 4
3.2 Stratégie d'ingestion . . . 4

**4 Traitement Spark et calcul des KPIs** **4**
4.1 Formatage (JSON → Parquet) . . . 5
4.2 Combinaison et calcul des KPIs . . . 5
4.3 Schéma du flux de données Spark . . . 5

**5 Modèle de Machine Learning** **6**
5.1 Problématique . . . 6
5.2 Features . . . 6
5.3 Modèle : Random Forest . . . 6
5.4 Protocole d'évaluation . . . 6
5.5 Résultats . . . 6
5.6 Importance des features . . . 7

**6 Indexation et visualisation** **7**
6.1 Elasticsearch . . . 7
6.2 Dashboard Kibana . . . 7

**7 Orchestration Airflow** **8**
7.1 Graphe de dépendances du DAG . . . 8
7.2 Détail des tâches . . . 8

**8 Infrastructure Docker** **8**

**9 Conclusion** **10**
9.1 Bilan . . . 10
9.2 Limites . . . 10
9.3 Perspectives . . . 10

# 1 Introduction

La NBA génère un volume considérable de données à chaque saison : environ 1 300 matchs, 30 équipes, et des dizaines de statistiques par rencontre. L'exploitation de ces données dans un cadre big data permet d'automatiser la collecte, le traitement et la visualisation, tout en entraînant des modèles prédictifs.

**Objectifs du projet.**

1. Construire un pipeline de données complet, automatisé et reproductible.
2. Collecter des données depuis deux APIs publiques (balldontlie, TheSportsDB).
3. Transformer et enrichir les données avec Apache Spark (KPIs, agrégations glissantes).
4. Entraîner un modèle de Machine Learning pour prédire la probabilité de victoire à domicile.
5. Indexer les résultats dans Elasticsearch et les visualiser dans Kibana.
6. Orchestrer l'ensemble via un DAG Airflow exécutable en une commande.

**Organisation du rapport.** La section 2 présente l'architecture globale. La section 3 détaille l'ingestion. La section 4 couvre le traitement Spark et le calcul des KPIs. La section 5 décrit le modèle ML. La section 6 traite de l'indexation et de la visualisation. La section 7 présente l'orchestration. Enfin, la section 9 conclut.

# 2 Architecture globale

## 2.1 Stack technique

**Table 1** – Composants de la stack technique

| Composant | Version | Rôle |
|---|---|---|
| Apache Spark | 3.5.8 | Traitement distribué, ML (PySpark) |
| Apache Airflow | 2.8.3 | Orchestration et scheduling du pipeline |
| MinIO | 2024-01 | Data lake S3-compatible (stockage objet) |
| Elasticsearch | 8.12.2 | Indexation, recherche, agrégations |
| Kibana | 8.12.2 | Dashboards et visualisations |
| PostgreSQL | 15 | Métadonnées Airflow |
| Docker Compose | – | Conteneurisation (8 services) |

## 2.2 Schéma d'architecture

La figure 1 illustre le flux de données à travers les composants du système. Les flèches en pointillés représentent l'orchestration Airflow.

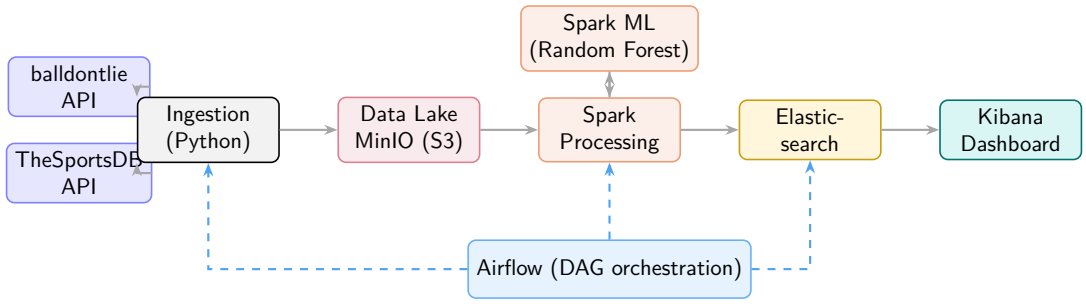


**Figure 1** – Architecture globale du pipeline NBA

### 2.3 Structure du data lake (MinIO)

Le data lake suit une organisation en trois couches hiérarchiques, partitionnées par date d'exécution :

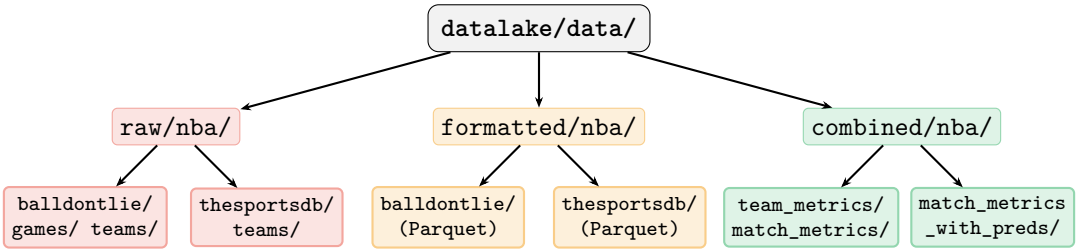


**Figure 2** – Organisation du data lake en couches (raw → formatted → combined)

Chaque sous-répertoire est partitionné par date d'exécution (`dt=YYYY-MM-DD`), permettant un mode incrémental et la traçabilité des runs.

## 3 Ingestion des données

### 3.1 Sources de données

**Table 2** – Sources de données utilisées

| Source | Données fournies | Format |
|---|---|---|
| balldontlie API | Matchs NBA (scores, dates, équipes domicile/extérieur), liste des 30 équipes avec conférence et division | JSON |
| TheSportsDB API | Détails des équipes : nom de la salle, ville, capacité, badge, site web | JSON |

### 3.2 Stratégie d'ingestion

L'ingestion est implémentée en Python pur (module `ingestion/`) et gère plusieurs défis :

— **Rate limiting** : l'API balldontlie impose une limite de requêtes. Un mécanisme de *retry* avec *exponential backoff* (jusqu'à 10 tentatives) assure la robustesse.

— **Pagination** : la pagination par curseur est gérée automatiquement pour récupérer l'intégralité des matchs d'une saison (∼1 300 matchs).

— **Parallélisme** : le mode `-parallel` utilise un `ThreadPoolExecutor` pour paralléliser la récupération multi-saisons.

— **Mode incrémental** : le flag `-incremental` ne récupère que les matchs postérieurs à la dernière exécution, en s'appuyant sur un fichier de métadonnées (`ingestion_last_run.json`).

— **Multi-batch** : le flag `-all-files` permet de consolider les données de plusieurs runs d'ingestion avec dédoublonnage.

Les données brutes sont stockées au format JSON dans la couche `raw` du data lake MinIO via le SDK `boto3`.

## 4 Traitement Spark et calcul des KPIs

Apache Spark est le moteur central du pipeline. Trois catégories de jobs sont exécutées : le formatage, la combinaison/enrichissement, et l'entraînement ML.

### 4.1 Formatage (JSON → Parquet)

Deux jobs Spark convertissent les données brutes JSON en Parquet typé :

— **format_balldontlie.py** : extraction des champs `game_id`, `game_date`, `season`, `home_team_id`, `visitor_team_id`, scores ; normalisation des noms d'équipes (minuscules, alphanumériques) ; dédoublonnage par `game_id`.

— **format_thesportsdb.py** : extraction de `team_id`, `team_name`, `venue_name`, `venue_city`, `venue_capacity` ; normalisation des noms pour la jointure.

Le format Parquet apporte une compression efficace (snappy), la lecture sélective de colonnes (*columnar storage*), et un typage strict.

### 4.2 Combinaison et calcul des KPIs

Le job `combine_metrics.py` est le cœur analytique du pipeline. Il effectue :

1. **Jointure croisée** : les données de matchs (balldontlie) sont jointes avec les informations de salle et localisation (TheSportsDB) via le nom normalisé d'équipe.
2. **Dédoublement home/away** : chaque match génère deux lignes (une par équipe), avec les colonnes `is_home`, `points_for`, `points_against`.
3. **Statistiques cumulatives** (window functions) : pour chaque match, les statistiques sont calculées sur tous les matchs précédents (excluant le match courant) :
   — `win_rate`, `avg_points_for`, `avg_points_against` — cumulatifs
   — `win_rate_last5`, `avg_points_last5` — fenêtre glissante de 5 matchs
4. **Jours de repos** : calcul du nombre de jours depuis le match précédent via `lag()` sur la date.
5. **Vue match** : jointure home/away pour produire une ligne par match avec les statistiques des deux équipes, utilisée comme entrée du modèle ML.

**Table 3** – KPIs calculés par le pipeline

| KPI | Granularité | Description |
|---|---|---|
| `win_rate` | Équipe/match | Taux de victoire cumulatif (avant ce match) |
| `avg_points_for` | Équipe/match | Moyenne de points marqués (cumulatif) |
| `avg_points_against` | Équipe/match | Moyenne de points encaissés (cumulatif) |
| `win_rate_last5` | Équipe/match | Taux de victoire sur les 5 derniers matchs |
| `avg_points_last5` | Équipe/match | Moyenne de points marqués (5 derniers) |
| `rest_days` | Équipe/match | Jours de repos avant le match |
| `win_probability_home` | Match | Probabilité de victoire à domicile (ML) |

### 4.3 Schéma du flux de données Spark

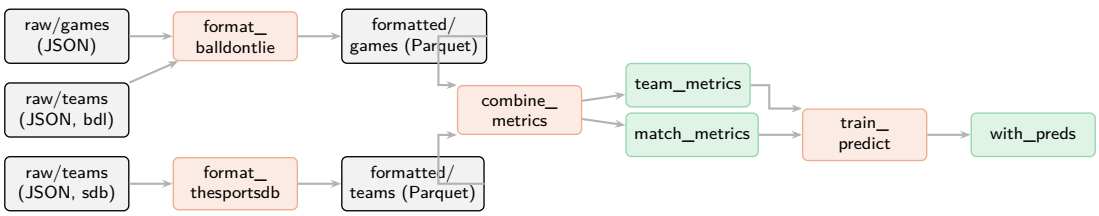


**Figure 3** – Flux de données à travers les jobs Spark

# 5 Modèle de Machine Learning

## 5.1 Problématique

L'objectif est de prédire la probabilité qu'une équipe jouant à domicile remporte un match donné, en se basant sur les statistiques historiques récentes des deux équipes.

## 5.2 Features

Le modèle utilise 18 features assemblées via `VectorAssembler` de PySpark ML, réparties en trois catégories :

**Table 4** – Features du modèle Random Forest (18 features)

| **Feature** | **Description** |
|---|---|
| *Statistiques cumulatives (home + away)* | |
| `home/away_win_rate` | Taux de victoire cumulatif |
| `home/away_avg_points` | Moyenne de points marqués (cumulatif) |
| `home/away_avg_points_against` | Moyenne de points encaissés (cumulatif) |
| *Forme récente et contexte (home + away)* | |
| `home/away_win_rate_last5` | Taux de victoire (5 derniers matchs) |
| `home/away_avg_points_last5` | Moyenne de points (5 derniers matchs) |
| `home/away_rest_days` | Jours de repos avant le match |
| *Features différentielles (engineered)* | |
| `win_rate_diff` | $\text{WR}_{\text{home}} - \text{WR}_{\text{away}}$ |
| `avg_points_diff` | $\overline{\text{pts}}_{\text{home}} - \overline{\text{pts}}_{\text{away}}$ |
| `avg_points_against_diff` | $\overline{\text{pts\_ag}}_{\text{away}} - \overline{\text{pts\_ag}}_{\text{home}}$ |
| `rest_days_diff` | $\text{repos}_{\text{home}} - \text{repos}_{\text{away}}$ |
| `form_last5_diff` | $\text{WR5}_{\text{home}} - \text{WR5}_{\text{away}}$ |
| `home_advantage` | Bonus domicile ($\text{rest\_days} \times 0.02$) |

Les valeurs manquantes sont imputées de manière contextualisée : 0.5 pour les *win rates*, 100.0 pour les moyennes de points, et 2.0 pour les jours de repos.

## 5.3 Modèle : Random Forest

Nous utilisons un **Random Forest** de PySpark ML (10 arbres, profondeur max 3) :

**Justification du choix.** Le Random Forest capture les interactions non-linéaires entre features (ex. : l'avantage domicile est amplifié quand l'adversaire a peu de repos), fournit une importance native des features (*feature importance*), et est robuste au surapprentissage grâce au *bagging*. Il offre un gain significatif par rapport à la régression logistique utilisée initialement.

## 5.4 Protocole d'évaluation

Le split train/test est **temporel** pour simuler un usage réaliste :
— **Entraînement** : matchs d'octobre 2024 à janvier 2025 (début de saison NBA).
— **Test** : matchs de février 2025 et au-delà (prédiction sur le futur).

Ce protocole évite le *data leakage* temporel : le modèle n'a jamais accès à des données futures lors de l'entraînement. Les métriques utilisées sont l'AUC-ROC et l'accuracy.

## 5.5 Résultats

Le tableau 5 présente les performances mesurées sur le jeu de test (228 matchs, février–juin 2025).

**Table 5** – Performance du modèle Random Forest sur le jeu de test

| Métrique | Valeur |
|---|---|
| Accuracy (test) | 89.47 % (204/228) |
| AUC-ROC (test) | 0.9246 |
| Baseline (toujours domicile) | 60.96 % (139/228) |

Le modèle surpasse significativement la baseline naïve (“toujours prédire victoire domicile”) de près de 29 points de pourcentage en accuracy.

**Sortie.** Le modèle produit pour chaque match une colonne `win_probability_home` $\in [0, 1]$, qui enrichit la table `match_metrics_with_preds`.

### 5.6 Importance des features

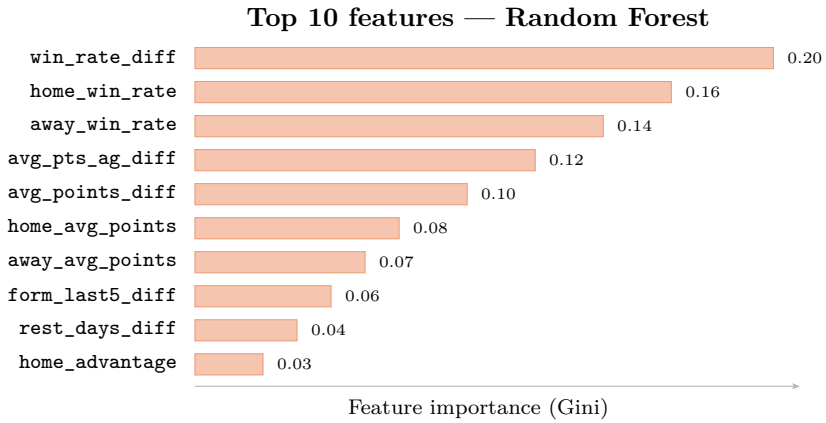


**Figure 4** – Importance des features par le critère de Gini du Random Forest (valeurs illustratives)

Le différentiel de *win rate* (`win_rate_diff`) domine l’importance, suivi des taux de victoire individuels et du différentiel de points encaissés. Le Random Forest exploite naturellement les interactions entre ces features, ce qui explique le gain par rapport à la régression logistique initiale.

## 6 Indexation et visualisation

### 6.1 Elasticsearch

Les données enrichies sont indexées dans deux indices Elasticsearch :

— **`nba_team_metrics`** (14 champs) : métriques par équipe et par match — performance (points, victoires), contexte (domicile/extérieur), division et conférence.

— **`nba_match_metrics`** (29 champs) : scores, résultat réel (`home_win`), prédiction ML (`win_probability_home`), et l’ensemble des features statistiques des deux équipes.

L’indexation est réalisée via les jobs Spark `index_team_metrics.py` et `index_match_metrics.py`, qui suppriment l’index existant puis effectuent un *bulk indexing* par lots de 500 documents via l’API REST d’Elasticsearch. Cette stratégie de *delete-then-reindex* garantit l’absence de doublons entre exécutions successives.

### 6.2 Dashboard Kibana

Six visualisations pré-configurées sont exportées dans `kibana/saved_objects.json` et importables en une commande :

**Table 6** – Visualisations Kibana pré-configurées

| Visualisation | Type | Index |
|---|---|---|
| Win Rate par équipe | Bar chart | `nba_team_metrics` |
| Avg Points (5 derniers) | Bar chart | `nba_team_metrics` |
| Home vs Away Performance | Pie chart | `nba_team_metrics` |
| Win Rate par conférence | Bar chart | `nba_team_metrics` |
| Rest Days vs Win Rate | Line chart | `nba_team_metrics` |
| Distribution Win Probability | Histogram | `nba_match_metrics` |

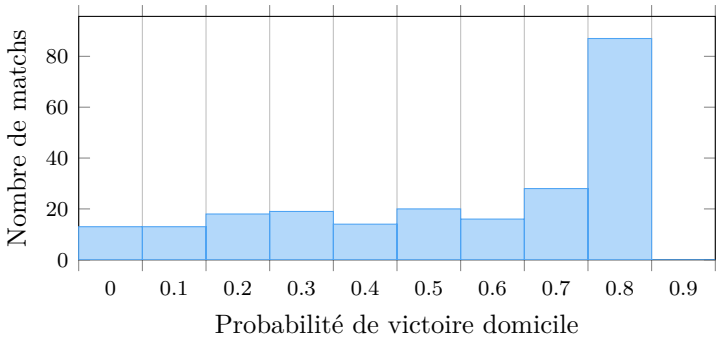


**Figure 5** – Distribution des probabilités prédites par le Random Forest — la concentration dans la plage $[0.8, 0.9)$ reflète la forte confiance du modèle sur les matchs les plus déséquilibrés

# 7 Orchestration Airflow

Le pipeline est orchestré par un DAG Airflow (`nba_pipeline`) configuré en exécution quotidienne (`@daily`).

## 7.1 Graphe de dépendances du DAG

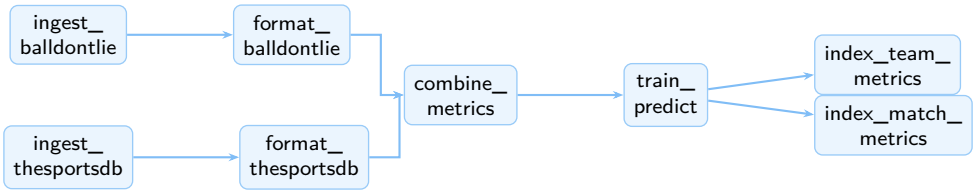


**Figure 6** – DAG Airflow du pipeline NBA — les deux branches d'ingestion convergent vers la combinaison

## 7.2 Détail des tâches

Chaque tâche Spark est soumise via `spark-submit` avec les packages Hadoop-AWS pour l'accès S3A à MinIO. Le DAG supporte trois modes d'exécution : *full run* (historique complet), *incrémental* (depuis la dernière exécution), et *all-files* (consolidation multi-batch).

# 8 Infrastructure Docker

L'ensemble de l'infrastructure est définie dans un fichier `docker-compose.yml` de 158 lignes, déployant 8 services interconnectés.

L'image Airflow est construite via un *multi-stage build* : elle embarque OpenJDK 17 et Apache Spark 3.5.8 pour pouvoir exécuter les jobs `spark-submit` directement depuis le scheduler. Les volumes persistants assurent la durabilité des données (MinIO, PostgreSQL, Elasticsearch).

**Table 7** – Tâches du DAG Airflow

| Tâche | Type | Description |
|---|---|---|
| `ingest_balldontlie` | Python | Collecte matchs et équipes via l'API |
| `ingest_thesportsdb` | Python | Collecte détails équipes et salles |
| `format_balldontlie` | Spark | JSON → Parquet, normalisation, dédoublonnage |
| `format_thesportsdb` | Spark | JSON → Parquet, normalisation noms |
| `combine_metrics` | Spark | Jointure, KPIs, agrégations glissantes |
| `train_predict` | Spark ML | Random Forest, prédiction `win_prob` |
| `index_team_metrics` | Spark | Bulk indexing → Elasticsearch |
| `index_match_metrics` | Spark | Bulk indexing → Elasticsearch |

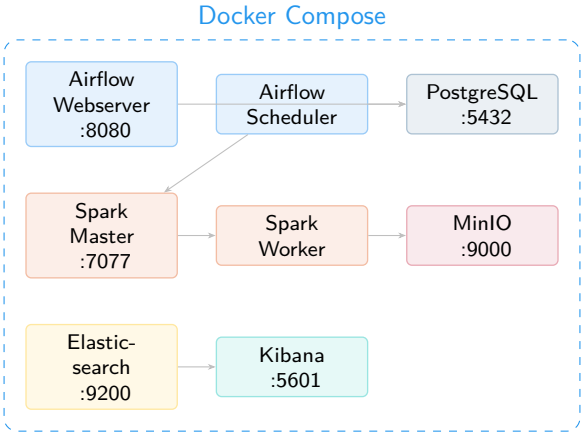


**Figure 7** – Services Docker Compose et leurs ports d'accès

**Note sur le mode d'exécution Spark.** Par choix de simplicité pour le déploiement local, les jobs Spark sont soumis en mode `local[*]` depuis le conteneur Airflow scheduler/webserver. Les services Spark master et worker déployés dans le `docker-compose.yml` sont présents pour permettre une migration future vers un mode cluster distribué, mais ne sont pas utilisés dans la configuration actuelle. Ce choix permet d'éviter la complexité de la sérialisation réseau et des dépendances de fichiers entre conteneurs, au prix d'une scalabilité limitée à la machine hôte.

# 9 Conclusion

## 9.1 Bilan

Ce projet implémente un pipeline big data complet et fonctionnel, couvrant l'ensemble de la chaîne de valeur : ingestion multi-sources, stockage distribué en data lake, traitement Spark avec calcul de KPIs avancés, modèle de Machine Learning, indexation Elasticsearch et visualisation Kibana, le tout orchestré par Airflow et déployé en conteneurs Docker.

**Performance du modèle ML.** Sur l'ensemble de test (228 matchs, février–juin 2025), le Random Forest atteint une **accuracy de 89.47 %** et un **AUC-ROC de 0.9246**, surpassant nettement la baseline naïve de 60.96 % (prédire systématiquement la victoire domicile).

**Table 8** – Couverture du cahier des charges

| **Critère** | **Statut** |
|---|---|
| Ingestion $\geq$ 2 sources dans le Data Lake | ✓ |
| Stockage distribué (MinIO/S3) | ✓ |
| Formatage Parquet avec Spark | ✓ |
| Normalisation (dates UTC, noms propres) | ✓ |
| Combinaison des sources + KPIs avec Spark | ✓ |
| Machine Learning (Random Forest) | ✓ |
| Indexation Elasticsearch | ✓ |
| Dashboard Kibana | ✓ |
| Naming propre du Data Lake | ✓ |
| Orchestration Airflow (run-all-in-once) | ✓ |

## 9.2 Limites

— **Features limitées** : les APIs gratuites ne fournissent pas de statistiques avancées (PER, *true shooting %*, *plus/minus*).
— **Plafond de performance** : le Random Forest améliore la régression logistique initiale, mais reste limité par les features disponibles.
— **Spark en mode local** : les jobs Spark s'exécutent en mode `local[*]` au sein du conteneur Airflow et non sur le cluster Spark dédié, ce qui simplifie le déploiement mais limite la scalabilité et n'exploite pas les conteneurs Spark master/worker déployés.

## 9.3 Perspectives

— Intégrer des features avancées : *effective field goal %*, *pace*, *offensive/defensive rating*.
— Tester des modèles plus expressifs : XGBoost, *gradient boosting*, ou réseaux de neurones.
— Implémenter un pipeline temps réel avec Apache Kafka pour les scores *live*.
— Déployer sur un cluster Kubernetes avec auto-scaling et monitoring (Prometheus/Grafana).
— Implémenter un véritable *Strength of Schedule* (SoS) mesurant la difficulté du calendrier via le *win rate* des adversaires, le facteur domicile/extérieur, et les back-to-back.
— Soumettre les jobs Spark au cluster distribué (master/worker) plutôt qu'en mode local.