Rapport — Projet Big Data (Architecture Lambda) : Lichess

# 1. Résumé exécutif

Ce projet met en place un pipeline de données basé sur l’architecture Lambda pour analyser des parties de jeu d’échecs Lichess en temps réel et en batch. La couche Speed ingère un flux d’événements via l’API publique Lichess et les publie dans Kafka (topic 'lichess\_moves'). En parallèle, une couche Batch lit/agrège des données depuis HDFS avec Spark et écrit des résultats en Parquet/JSON. Les résultats temps réel et batch sont exposés à un outil BI (Power BI) et à un panneau de filtrage (scripts Python). L’ensemble s’exécute dans un environnement isolé Docker Compose (Kafka, Zookeeper, Spark, HDFS, Airflow).

# 2. Architecture globale (Lambda)

• Domaine : Jeux d’échecs — flux de parties TV Lichess.

• Source : API publique Lichess (TV channels + stream de partie).

• Speed layer : Producer Python → Kafka; HDFS incrémental; Spark Streaming via Airflow (extensible); Consumers → Excel/PowerBI .

• Batch layer : Spark job (PySpark) lisant HDFS et écrivant des agrégations dans HDFS.

• Serving : fichiers sur HDFS (batch), fichier Excel + éventuelle base/visualisation (temps réel).

• Orchestration: Airflow (DAG 'lichess\_streaming\_dag').

• Infra: Docker Compose (Hadoop/HDFS, Spark Master/Worker, Kafka, Zookeeper, Jupyter, Airflow).

## 2.1 Composants & versions

|  |  |  |  |
| --- | --- | --- | --- |
| Service | Image / Version | Rôle | Ports principaux |
| Kafka | wurstmeister/kafka | Bus d’événements (topic 'lichess\_moves') | 9092 |
| Zookeeper | wurstmeister/zookeeper | Coordination Kafka | 2181 |
| Hadoop NameNode | bde2020/hadoop-namenode:2.0.0-hadoop3.2.1-java8 | HDFS NN + UI | 9870, 9000 |
| Hadoop DataNode | bde2020/hadoop-datanode:2.0.0-hadoop3.2.1-java8 | Stockage blocs HDFS | 9864 (interne) |
| YARN RM | bde2020/hadoop-resourcemanager:2.0.0-hadoop3.2.1-java8 | Orch. jobs YARN | 8088 (interne) |
| YARN NM | bde2020/hadoop-nodemanager:2.0.0-hadoop3.2.1-java8 | Exécution conteneurs | 8042 (interne) |
| HistoryServer | bde2020/hadoop-historyserver:2.0.0-hadoop3.2.1-java8 | Historique jobs | 8188 (interne) |
| Spark Master | bde2020/spark-master:3.0.1-hadoop3.2 | Planification Spark | 8080, 7077 |
| Spark Worker | bde2020/spark-worker:3.0.1-hadoop3.2 | Exécution executors | — |
| Jupyter (PySpark) | jupyter/pyspark-notebook | Dev/Tests PySpark | 8888 |
| Airflow stack | apache/airflow:3.0.6 (+Redis, Postgres) | Orchestration DAGs | 8082 API, Web 8080 (interne) |

# 3. Couche Speed (Streaming)

Topic Kafka : 'lichess\_moves' (1 partition / 1 réplica). Broker : kafka:9092.

Producer (Python):

• API: https://lichess.org/api/tv/channels → récupération d’un gameId TV courant.

• Stream NDJSON: /api/stream/game/{gameId} (entêtes Accept: application/x-ndjson).

• Publication vers Kafka (JSON). Périodicité de polling ~60s (reconnexion tolérante aux erreurs).

Consumers (Python):

• Consumer-Excel: consolide les événements et écrit/agrège dans lichess\_powerbi.xlsx (feuilles raw\_data, player\_stats, game\_stats). Rafraîchissement batch toutes les 60s.

• Consumer-HDFS: agrège par id de partie et met à jour /user/jovyan/lichess/lichess\_<gameId>.json sur HDFS via WebHDFS (9870).

## 3.1 Schéma – Événements

Événement début de partie (exemples de champs) : id, players{white{user{name},rating}, black{…}}, wc, bc, status{name}, winner.

Événement mouvement (exemples) : fen, lm (last move), wc, bc, status?, winner?

Schéma interne consolidé pour Excel/HDFS : timestamp, game\_id, player, rating, last\_move, color, wc, bc, status, winner.

# 4. Couche Batch (Spark)

Script : work/model/batch\_panel.py (ex.).

Entrée HDFS : hdfs://namenode:9000/lichess/streaming/lichess.json

Sorties HDFS : hdfs://namenode:9000/lichess/batch\_results/games\_by\_status et avg\_moves (JSON).

Transformations clés :

• groupBy(status) → count nb\_games.

• Estimation de nb\_moves à partir de la chaîne 'moves' (si disponible) et calcul de avg\_moves.

• Affichage/échantillon (df.show) pour contrôle.

# 5. Orchestration (Airflow)

DAG: lichess\_streaming\_dag (schedule manuel).

Tâche SparkSubmitOperator: application=/opt/airflow/dags/scripts/spark\_streaming.py, driver/executor mémoire=2g, 2 cœurs.

Airflow déployé via docker-compose dédié (Postgres, Redis, Webserver/API, Scheduler, Worker, Triggerer).

# 6. Frontends

Dashboard temps réel : Power BI connecté au fichier Excel 'lichess\_powerbi.xlsx' (rafraîchi par le consumer).

Panneau de filtrage : notebooks/scripts Python (work/model) permettant requêtes additionnelles et écriture HDFS; possibilité d’exposer via Flask/FastAPI.

Alerting simple : log/print côté consumers + possibilité d’extension (email/webhook).

## Frontend Flask — Panneau de visualisation et filtrage

Un **frontend web léger** a été développé avec **Flask** afin d’exposer les résultats produits par le pipeline (fichier Excel généré par le consumer Kafka → Excel).

### Structure du projet

* app.py : application Flask (logique backend).
* templates/index.html : page HTML principale (Jinja2).
* static/styles.css : feuille de style.
* static/bg.jpg : image de fond (optionnelle).
* requirements.txt : dépendances Python.
* lichess\_powerbi.xlsx : fichier Excel source (alimenté en continu).

### Fonctionnement

* À chaque requête sur /, l’application lit le fichier **Excel** contenant les données Lichess.
* Les onglets raw\_data et game\_stats sont détectés automatiquement (avec variantes raw-data, gamestats, etc.).
* Le script reconstruit les colonnes clés :
  + joueurs blanc/noir,
  + statut de la partie,
  + nombre de coups,
  + vainqueur (⚪ blanc, ⚫ noir, ½–½ nulle),
  + durée approximative de la partie (calculée à partir des horloges wc et bc).
* Le résultat est rendu dans un tableau HTML interactif (tri au clic, filtrage en texte libre côté client).

### Variables et configuration

* Variable d’environnement : LICHESS\_XLSX\_PATH (chemin vers le fichier Excel).
* Valeur par défaut dans app.py : /Users/.../lichess\_powerbi.xlsx.
* Peut être surchargée dans un déploiement Docker ou local.

### Dépendances

* Flask, pandas, numpy, openpyxl, gunicorn.
* Installées via requirements.txt.

### Lancement local

python3 -m venv .venv && source .venv/bin/activate

pip install -r requirements.txt

export LICHESS\_XLSX\_PATH=/chemin/vers/lichess\_powerbi.xlsx

python app.py # accès http://localhost:5000

### Intégration au pipeline

Le frontend repose sur le **consumer Kafka → Excel** :

* Le fichier lichess\_powerbi.xlsx est continuellement mis à jour.
* Le frontend lit ce fichier à chaque chargement de page, garantissant un rafraîchissement automatique des données.

Ce composant offre une **visualisation simple et interactive** complémentaire à Power BI : idéal pour tester le pipeline et filtrer rapidement les résultats.

# 7. Isolation & Déploiement (Docker / Compose)

Stacks :

• Infra data (HDFS/Spark/Kafka/Zoo/Jupyter) — fichier docker-compose.yml ; réseau 'bigdata'.

• Orchestration Airflow — docker-compose séparé (profil CeleryExecutor).

Volumes persistants : hadoop\_namenode, hadoop\_datanode, hadoop\_historyserver, postgres-db-volume.

Commandes typiques :

docker network create bigdata # une fois  
 docker compose -f docker-compose.yml up -d # infra data  
 docker compose -f docker-compose.yaml up -d # stack Airflow

Vérifications : NameNode UI http://localhost:9870, Spark Master http://localhost:8080, Jupyter http://localhost:8888, Kafka 9092.

# 8. Sécurité & Conformité

Gestion basique des secrets via .env; réseau Docker isolé; pas de données personnelles (domaine jeu).

# 9. Tests & Validation

Tests de connectivité : producteur → Kafka; consumers → HDFS/Excel; Spark → HDFS; Airflow → SparkSubmit.

Tests volumétrie : augmentation fréquence/période du producteur; validation du lag Kafka et de la latence bout-en-bout.

Qualité : schéma attendu, champs non-nuls essentiels (id/lm), cohérence des timestamps.

# 10. Mode opératoire

1) Lancer l’infra Docker (réseau + compose).  
2) D démarrer le producer Lichess (work/model/Extract.py ou script équivalent).  
3) Démarrer un consumer (Excel ou HDFS) selon besoin.  
4) Vérifier le fichier Excel puis créer un rapport/graph dans Power BI.  
5) Lancer le job batch Spark (batch\_panel.py) manuellement ou via Airflow.  
6) Consulter HDFS pour les sorties (batch\_results/\*).