2026

PROJET FINAL

SYSTEME DE RECOMMANDATION E-COMMERCE EN TEMPS REEL

Étudiants

DOUDOU Hissein

KADJO Aka Elishama

Professeure

KERKENI Imen

Introduction

Ce rapport présente la conception et l'implémentation d'un pipeline Big Data en streaming pour un site d'e-commerce. L'objectif principal de ce projet est de traiter en temps réel un flux massif d'interactions utilisateurs (clics) afin d'identifier instantanément des comportements anormaux, tels que des robots ou des requêtes abusives.

Pour répondre à ces exigences, nous avons mis en place une stack technologique robuste reposant sur **Apache Kafka** pour l'ingestion asynchrone et **Apache Flink (PyFlink)** pour le calcul stateful par fenêtre temporelle ("Event Time"). À travers ce rapport, nous détaillerons nos choix algorithmiques, justifierons l'architecture retenue et analyserons les limites d'un tel système face au monde de la production.

1. Qualité et temporalité des données

Pourquoi est-il essentiel de disposer d'un champ timestamp explicite ?

Dans un système e-commerce en temps réel, l'ordre d'arrivée des événements sur le réseau ne correspond presque jamais à l'ordre exact dans lequel l'utilisateur a effectué ses actions (latence réseau, déconnexions mobiles, retries). Un champ *timestamp* explicite inséré à la source (par le navigateur ou l'application) garantit que le moteur de streaming (Flink) traite les données dans leur véritable ordre chronologique métier, assurant ainsi l'exactitude des calculs (ex: agrégations par minutes précises, séquences d'actions).

Quelle est la différence entre Event Time et Processing Time ?

- **Event Time** : C'est l'heure à laquelle l'événement s'est réellement produit sur l'appareil de l'utilisateur (le champ *timestamp* dans notre payload JSON). C'est la seule métrique fiable pour des calculs métiers exacts.
- **Processing Time** : C'est l'heure à laquelle l'événement est reçu et traité par le serveur Flink. Il est soumis aux aléas de l'infrastructure (bouchons Kafka, redémarrages). L'utiliser fausserait les fenêtres temporelles en cas de pic de charge.

Quels problèmes peuvent apparaître si les timestamps sont absents ou incorrects ?

Si Flink devait se baser sur le Processing Time (en l'absence de timestamp métier), une panne réseau temporaire de 2 minutes suivie d'une reconnexion provoquerait un "raz-de-marée" d'événements. Flink les grouperait tous dans la même minute de Processing Time, déclenchant de fausses alertes d'anomalies (faux positifs). De même, un timestamp incorrect (ex: horloge client désynchronisée) pourrait placer l'événement dans une fenêtre déjà clôturée, causant sa perte (late data).

2. Détection d’Anomalies

Quelle définition d’anomalie avez-vous retenue ?

Nous avons défini une anomalie comme un volume anormalement élevé d'interactions (plus de 50 événements de type "click") provenant d'un même *user_id* au sein d'une fenêtre temporelle fixe et non-chevauchante (Tumbling Window) de 1 minute.

Pourquoi cette définition est-elle pertinente pour un système e-commerce ?

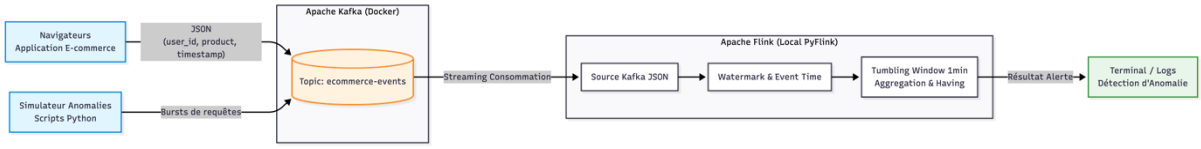
Un humain naviguant sur un site e-commerce clique à un rythme mesuré (lecture, scroll, réflexion). Un compte générant plus de 50 clics en 60 secondes (soit presque 1 clic par seconde de façon soutenue) témoigne d'un comportement non-humain. Cela permet d'identifier très tôt les robots (scrapers pillant le catalogue ou les prix), les scripts d'attaque (credential stuffing) ou des bugs côté client (boucle infinie dans le JavaScript).

Quel rôle jouent les fenêtres temporelles dans cette détection ?

Les fenêtres temporelles bornent l'analyse. Compter les clics sur la durée totale de vie du compte n'a pas de sens pour détecter un bot. La fenêtre Tumbling d'une minute permet d'isoler une "densité" d'action. De plus, elles libèrent l'état de Flink : une fois la fenêtre d'une minute fermée et le résultat (anomalie ou non) émis, Flink peut purger la mémoire associée au comptage de ce *user_id*, assurant la scalabilité infinie du système.

3. Architecture et tolérance aux pannes

Schéma d'Architecture



Quel est le rôle de Kafka dans cette architecture ?

Kafka agit comme un *buffer* (tampon) durable et scalable entre les producteurs (application web/mobile) et les consommateurs (Flink). Il découple l'ingestion du traitement. Il encaisse les pics de charge (bursts) que Flink ou la base de données ne pourraient pas ingérer instantanément, et stocke les données sur disque pour permettre le rejeu en cas de besoin.

Pourquoi Flink est-il adapté à ce type de traitement ?

Flink est conçu pour le streaming stateful (avec état) pur. Contrairement aux micro-batches, il traite les événements un par un avec une latence sub-milliseconde. Surtout, sa gestion native de l'Event Time et des Watermarks permet de clore des fenêtres de temps facilement même lorsque les données arrivent en retard (out-of-order logs), ce qui est indispensable dans ce cas d'usage e-commerce avec fenêtrage.

Que se passe-t-il si Kafka ou Flink devient temporairement indisponible ?

- **Si Flink tombe :** Les événements s'accumulent "en sécurité" dans les topics Kafka (grâce à la rétention sur disque de Kafka). Dès son redémarrage, Flink reprendra la lecture là où il s'était arrêté (grâce à ses checkpoints/savepoints stockant les offsets Kafka). Et grâce à l'Event Time, il recalculera les fenêtres de 1 minute d'hier avec la même exactitude algorithmique qu'en live.
- **Si Kafka tombe :** Les producteurs côté web/mobile ne pourront plus envoyer d'événements et devront soit implémenter un cache local (ex: LocalStorage) avec mécanisme de retry (ex: Exponential Backoff), soit accepter une perte de données temporaire.

4. Reproductibilité et Exécution

Quel est l'ordre correct d'exécution des scripts ?

1. Lancement de l'infrastructure Docker (*docker-compose up -d*) pour démarrer Zookeeper et Kafka.

2. Configuration de l'environnement Python local (*conda activate flink-env*) avec les dépendances PyFlink 2.2 et *pytest*.

3. Exécution des tests unitaires du Count-Min Sketch (*pytest tests/test_countmin.py -v*) pour valider la logique probabiliste.

4. Lancement du Producteur (*python scripts/producer.py*) pour générer le trafic de fond.

5. Lancement du Job Flink (*python scripts/pipeline.py*) pour consommer les événements, appliquer les watermarks et ouvrir les fenêtres.

6. Lancement du Simulateur d'Anomalie (*python scripts/anomaly_simulator.py*) pour injecter 10 000 événements groupés pour le *user_id=1*.

Comment vérifiez-vous que le pipeline fonctionne correctement ?

La vérification se fait sur trois plans :

1. **Tests unitaires :** La commande *pytest* valide à 100% les comportements du Count-Min Sketch (gestion du bruit, comptage, reset) avant même de lancer le pipeline.

2. **Console Flink :** Le terminal exécutant *pipeline.py* affiche en direct la table des anomalies dès qu'une fenêtre se ferme, prouvant le calcul exact.

3. **Dashboard Flink :** L'UI Web (accessible sur http://localhost:8082 pour le cluster local Python) montre le Job à l'état *RUNNING*, le graphe d'exécution (Source -> Window -> Sink), et les métriques de records processed confirmant que les données circulent sans backpressure.

Quelles erreurs courantes peuvent apparaître lors de l'exécution ?

- *Watermarks bloqués (Idle Partitions)* : Si le producteur n'écrit pas dans toutes les partitions Kafka, le watermark global de Flink n'avance plus, empêchant la fermeture de la fenêtre.

- *Incompatibilité Python/PyFlink* : PyFlink 2.2 nécessite strictement Python 3.10. Essayer de l'exécuter dans le conteneur Docker original (qui ne possède que Java) ou avec un Python > 3.10 provoque un crash instantané.

- *Port conflicts* : La tentative de lancer deux fois *pipeline.py* provoque une erreur de bind sur le port 8082 de l'interface REST Flink.

Captures d’écran d’exécution

```
(flink-env) (.venv) youssouf@MacBook-Pro-de-Hissein projet-final-ecommerce-examen % pytest tests/test_countmin.py -v
============================================================ test session starts ============================================================
platform darwin -- Python 3.11.11, pytest-9.0.2, pluggy-1.6.0 -- /Users/youssouf/Downloads/projet-final-ecommerce-examen/.venv/bin/python3
.11
cachedir: .pytest_cache
rootdir: /Users/youssouf/Downloads/projet-final-ecommerce-examen
collected 6 items

tests/test_countmin.py::test_ajout_et_requete PASSED                                                                                  [ 16%]
tests/test_countmin.py::test_comptage_multiple PASSED                                                                                 [ 33%]
tests/test_countmin.py::test_comptage_exact PASSED                                                                                    [ 50%]
tests/test_countmin.py::test_cle_inconnue PASSED                                                                                      [ 66%]
tests/test_countmin.py::test_jamais_sous_estime PASSED                                                                                [ 83%]
tests/test_countmin.py::test_reset PASSED                                                                                             [100%]

============================================================ 6 passed in 0.03s =============================================================
(flink-env) (.venv) youssouf@MacBook-Pro-de-Hissein projet-final-ecommerce-examen %
```

Figure 1 - Tests unitaires du Count-Min Sketch

```
(flink-env) youssouf@MacBook-Pro-de-Hissein projet-final-ecommerce-examen % python scripts/producer.py
Démarrage du producteur standard.
   Envoi continu de clics normaux vers Kafka (~10 évents/sec).
   (Pressez Ctrl+C pour arrêter)
```

Figure 2 – Production et envoi des clics dans Kafkta

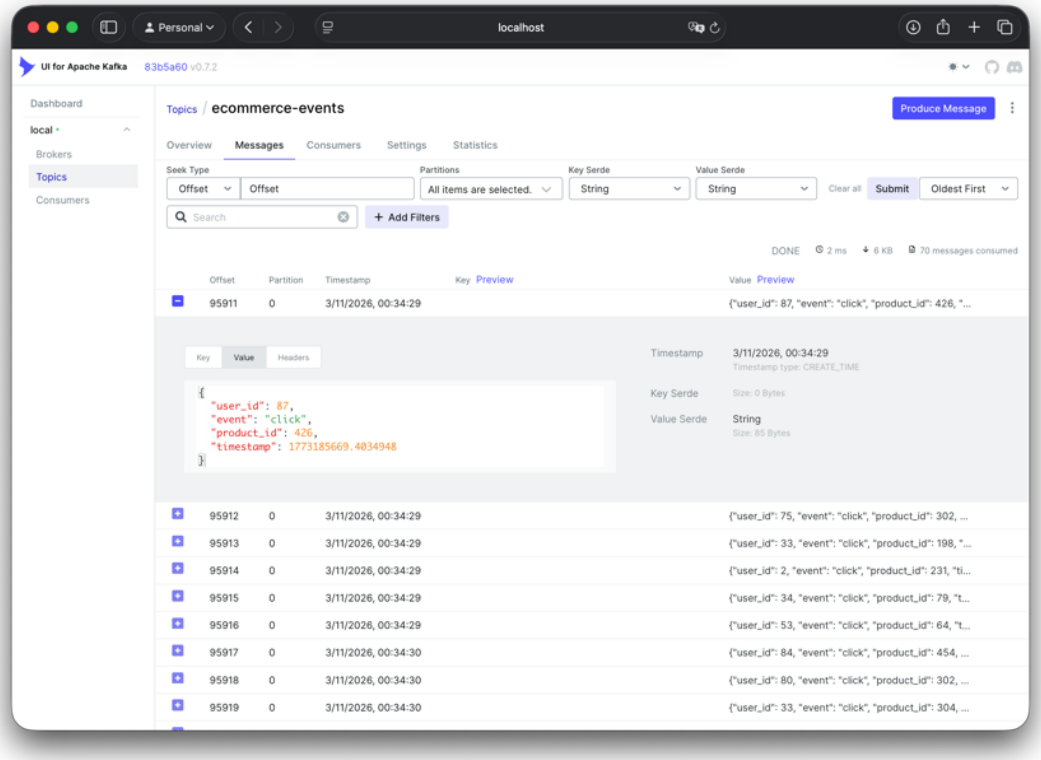


Figure 3 - Flux des clics dans Kafka UI

```
(flink-env) youssouf@MacBook-Pro-de-Hissein projet-final-ecommerce-examen % python scripts/anomaly_simulator.py
Lancement de la simulation d'anomalie...
   Envoi massif de 10 000 clics pour l'utilisateur 1 en cours...
```

Figure 4 – Production et envoi des anomalies

```
(flink-env) youssouf@MacBook-Pro-de-Hissein projet-final-ecommerce-examen % python scripts/pipeline.py
Pipeline démarré — seuil=50 clics / 1 min
Les anomalies détectées apparaîtront ci-dessous. En attente de la fermeture des fenêtres...
+----+-------------+-------------------------+-------------------------+---------------------+
| op |     user_id |            window_start |              window_end |          click_count |
+----+-------------+-------------------------+-------------------------+---------------------+
| +I |           1 | 2026-03-11 00:51:00.000 | 2026-03-11 00:52:00.000 |                1365 |
| +I |           1 | 2026-03-11 00:52:00.000 | 2026-03-11 00:53:00.000 |                5191 |
| +I |           1 | 2026-03-11 00:53:00.000 | 2026-03-11 00:54:00.000 |                2594 |
| +I |           1 | 2026-03-11 00:54:00.000 | 2026-03-11 00:55:00.000 |                5194 |
```

Figure 5 - Détection des anomalies par Flink

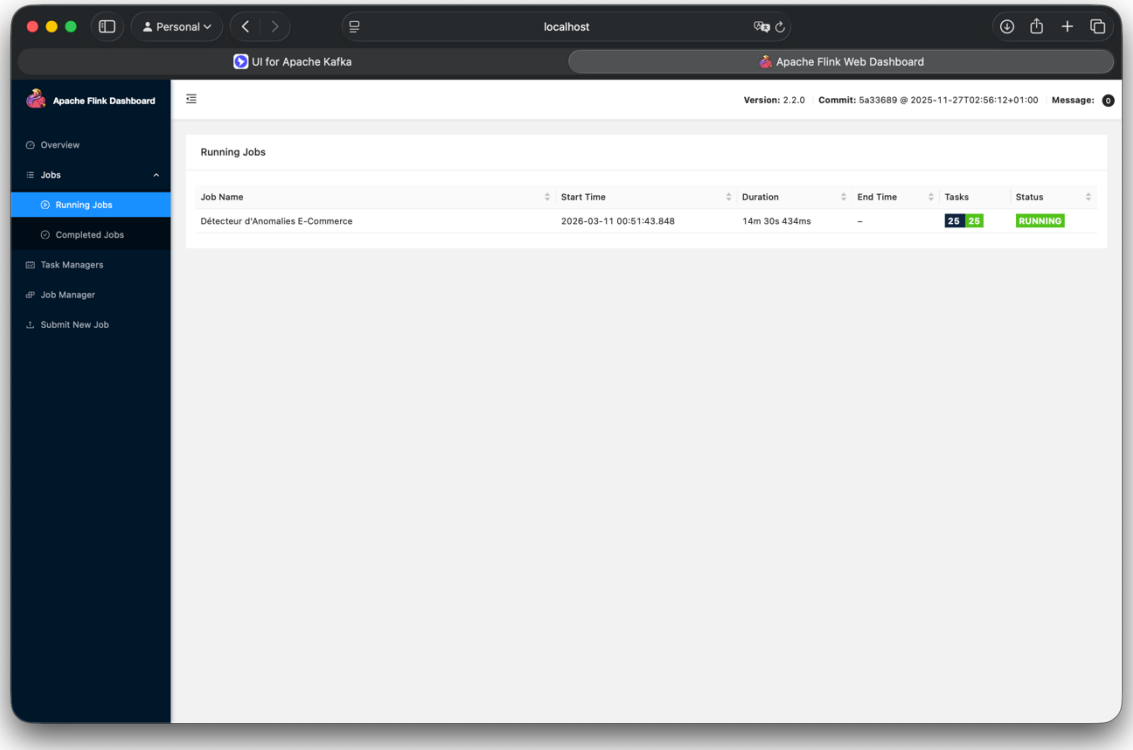


Figure 6 - Job Flink 1/2

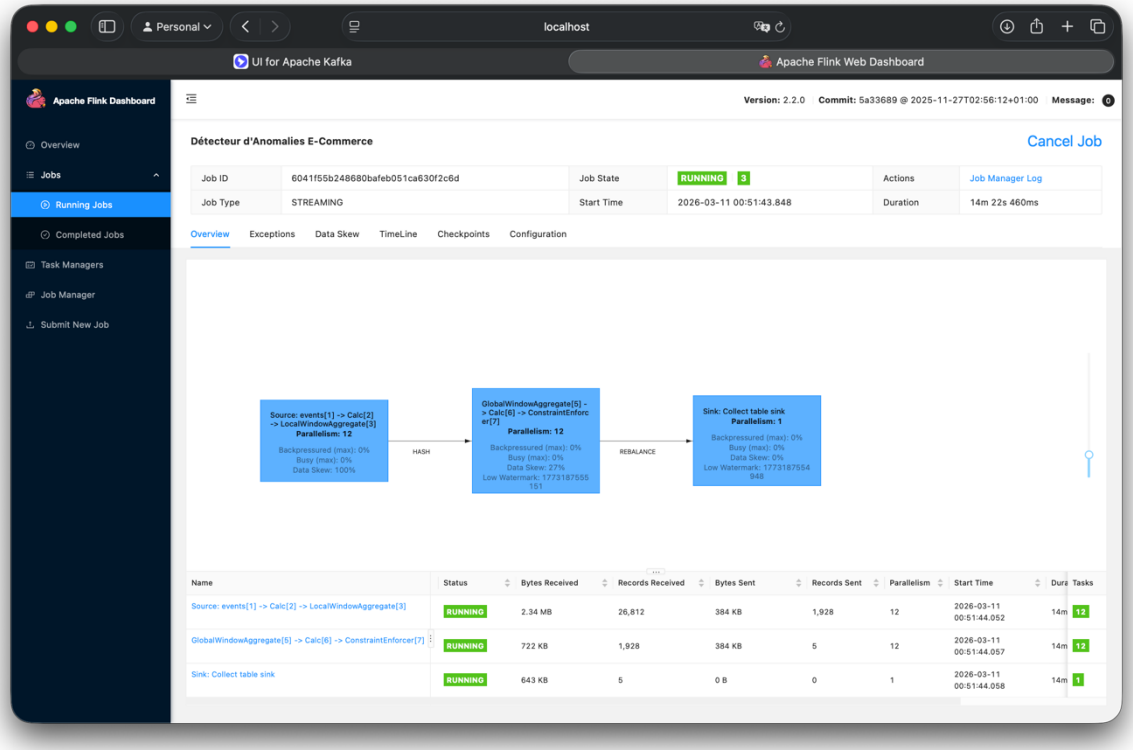


Figure 7 - Job Flink 2/2

5. Analyse Critique

Quelles sont les limites actuelles de votre pipeline ?

- **Comptage exact vs Mémoire** : L'utilisation de *COUNT(*)* avec une clause *GROUP BY* exige de Flink de maintenir l'état exact pour le croisement *user_id / window*. Si nous avions des centaines de millions d'utilisateurs simultanés, la State Backend de Flink saturerait la mémoire/disque.

- **Tolérance au retard (Lateness)** : Le watermark est paramétré à 5 secondes. Un événement arrivant avec 10 secondes de retard sur l'Event Time sera silencieusement éliminé (Late Data dropped) car la fenêtre de sa minute correspondante sera déjà calculée et fermée.

Quelles améliorations proposeriez-vous (performance, scalabilité, précision) ?

- **Performance/Scalabilité** : Remplacer l'agrégation SQL exacte par le **Count-Min Sketch** implémenté et testé dans *countmin.py* via une UDF (User Defined Function) Flink. Cela réduirait l'empreinte mémoire à une matrice de taille fixe, au prix d'une légère incertitude probabiliste (surestimation très acceptable pour interdire un bot).

- **Précision** : Utiliser des *Sliding Windows* (Fenêtres glissantes) d'une minute avançant toutes les 10 secondes, plutôt que des *Tumbling Windows* statiques. Actuellement, si le robot envoie 30 clics à 12h00m50s et 30 clics à 12h01m05s, il passe sous le radar (répartis sur deux fenêtres distinctes). Une fenêtre glissante l'aurait intercepté.

Que faudrait-il ajouter pour un déploiement en production ?

1. **Alerting Actionnable** : Écrire le Sink Flink vers un nouveau Topic Kafka *ecommerce-alerts* plutôt que vers le *print()*. Un micro-service se chargerait de lire ce topic pour envoyer les ordres de bloquage ou alerter un SOC via messagerie.

2. **State Backend** : Configurer un State Backend au lieu du Memory backend par défaut, et configurer un système de stockage persistant (S3 ou HDFS) pour les Checkpoints périodiques, garantissant le Exactly-Once delivery.

3. **Monitoring** : Connecter Prometheus/Grafana pour surveiller la santé de Flink (restarts, checkpoint duration, CPU du TaskManager) et l'application (Kafka lag sur ConsumerGroups).

Conclusion et Enseignements tirés

La réalisation de ce pipeline nous a permis de confronter la théorie du streaming aux défis réels de l'ingénierie des données. Nous avons particulièrement appris :

1. **La suprématie de l'Event Time**: Penser en *Event Time* demande un changement de paradigme. Gérer le retard (lateness) via les watermarks est complexe mais absolument vital pour garantir des résultats déterministes et exacts peu importe l'état du réseau.

2. **Les pièges des systèmes distribués** : Face au problème des fenêtres Flink qui ne se fermaient pas lors du développement, nous avons compris le mécanisme de blocage des watermarks sur les partitions Kafka inactives (Idle Partitions), corrigé grâce à l'option *idle-timeout*.

3. **Le compromis Mémoire / Précision** : L'implémentation parallèle d'un Count-Min Sketch (dans les scripts additionnels) nous a montré comment réduire drastiquement l'empreinte mémoire d'une application Flink au prix d'une perte algorithmique négligeable (la probabilité de collisions).

Ce projet constitue une excellente base qui valide la puissance d'Apache Flink et d'Apache Kafka pour répondre aux problématiques de détection de fraude et de bots dans un cadre e-commerce moderne.

Lien du projet sur GitHub.