Ecole Polytechnique de Thiès

Département **Génie Informatique et Télécoms**

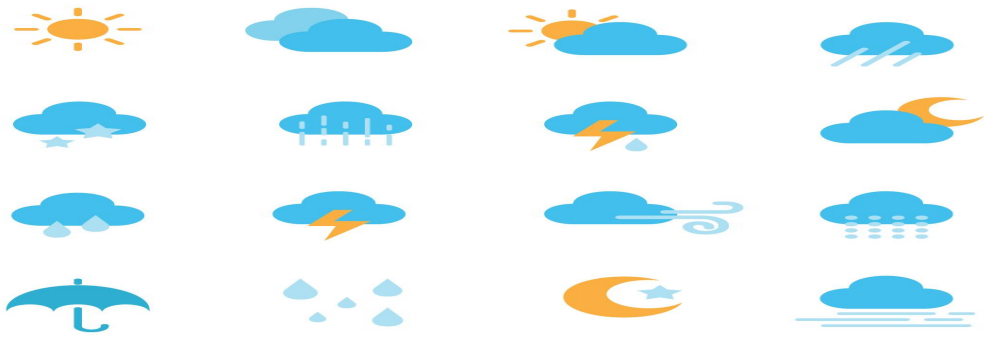
DIC 2 GIT - 2025

Cours de **Big Data & Applications**

Professeur : **Mr Guèye**

Projet Matière

Sujet : Données météo en temps réel et en prédiction



Présenté par :

- Ameth BA
- Thierno Saydou Talla

Table des matières

I. INTRODUCTION **3**
II. Objectifs **3**
III. ARCHITECTURE GÉNÉRALE **3**
IV. INGESTION DES DONNÉES **5**
1) Sources de données 5
2) Configuration Kafka 6
V. TRAITEMENT AVEC SPARK **6**
VI. Modèle de prédiction **8**
VII. Stockage des Données **9**
VIII. Visualisation & monitoring **10**
1. Onglet "Vue d'ensemble" 10
2. Onglet "Analyses" 11
3. Onglet "Monitoring" 12
IX. Difficultés rencontrées et solutions mises en œuvre **13**
X. Conclusion **13**
XI. Sources et références **14**

I. INTRODUCTION

Les données météorologiques évoluent en permanence et nécessitent un traitement en temps réel pour être utiles. Que ce soit pour l'agriculture, les transports ou la planification urbaine, disposer d'informations actualisées est devenu essentiel.

Dans un monde où les capteurs IoT et les APIs produisent des volumes croissants de données, les architectures Big Data offrent des solutions pour traiter ces flux continus. Ces technologies permettent de passer d'une simple collecte à une analyse intelligente capable de générer des prédictions.

Ce rapport présente un pipeline complet de traitement de données météo. Nous aborderons l'architecture générale, puis chaque étape : ingestion, traitement Spark, prédictions, stockage et visualisation.

II. Objectifs

Ce projet vise à mettre en pratique les concepts du Big Data à travers un cas concret. L'objectif principal est de concevoir une architecture capable de traiter des flux de données météo en temps réel.

Le système doit collecter des données depuis une ou plusieurs sources, les nettoyer et les fusionner . Il doit réaliser des agrégations temporelles et générer des prédictions à court terme pour les températures,basées sur l'historique récent .

Un dashboard interactif doit permettre de visualiser les données(météo temps réel , courbes d'évolutions ,etc) et surveiller le système (journalisation, détection de problèmes,etc). Cette dimension de monitoring est assez intéressante pour assurer la fiabilité du pipeline.

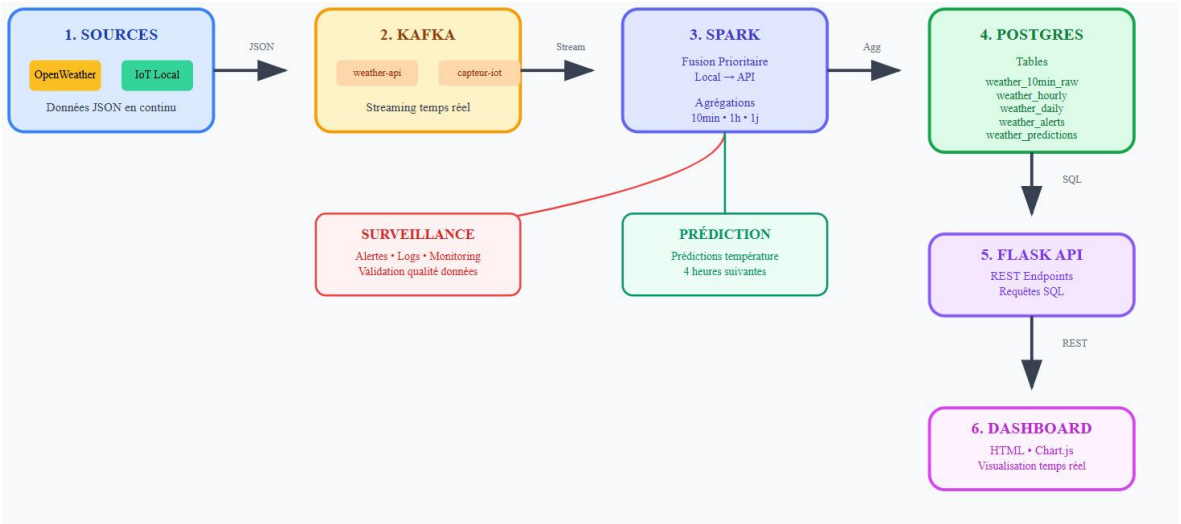
III. ARCHITECTURE GÉNÉRALE

Nous avons développé un pipeline Big Data complet qui collecte des données météo depuis deux sources différentes (capteur Iot simulé et l'api OpenWeatherMap). Le système traite ces données en temps réel avec Spark, génère des prédictions automatiques et les visualise dans un dashboard web. Il ya aussi un système de monitoring qui gère la détection d'anomalies et surveille l'ensemble des composants .

Le fonctionnement global suit un **pipeline en 6 étapes** qui seront détaillées par la suite**:**

- **Deux producteurs Python** collectent les données : l'un interroge l'API OpenWeather, l'autre simule un capteur IoT local. Ces producteurs envoient leurs données au format JSON vers Apache Kafka, un broker de messages qui garantit la livraison fiable et permet le découplage entre sources et traitement.
- **Kafka** organise les données en topics séparés (weather-api et capteur-iot) et les stream vers Spark Structured Streaming. Ce moteur de traitement distribué lit les flux en temps réel, parse les messages JSON selon leurs schémas respectifs, puis nettoie et fusionne les données en privilégiant le capteur local quand disponible.
- **Spark Structured Streaming** (moteur de traitement distribué) lit simultanément les deux flux , parse les messages JSON selon leurs schémas respectifs, puis nettoie et fusionne les données en **privilégiant le capteur local quand disponible.**

  Spark réalise ensuite trois niveaux d'agrégations temporelles :
  - ✓ Toutes les 10 minutes : fusion intelligente avec moyennes (priorité capteur local > API)
  - ✓ Toutes les heures : statistiques complètes (min, max, moyenne) calculées sur les données 10 minutes
  - ✓ Journalières : mêmes statistiques (min, max, moyenne) mais sur fenêtres d'une journée
- Les résultats sont automatiquement **stockés dans PostgreSQL**(base de données relationnelle) qui centralise toutes les données : données 'brutes' ,agrégations, prédictions et alertes .
- En parallèle, un **modèle de prédiction** simple analyse les tendances récentes et génère des prévisions à 4 heures.
- Enfin, un **dashboard web** interagit avec une API REST exposée par une application Flask. L'interface, construite avec Chart.js, affiche les données dans des graphiques répartis en trois onglets : vue d'ensemble temps réel, analyses statistiques et monitoring système.



IV. INGESTION DES DONNÉES

L'ingestion des données repose sur Apache Kafka, qui joue un rôle central dans notre architecture.
Apache Kafka est une plateforme distribuée de streaming d'événements. Elle permet de publier, stocker, transmettre et traiter des flux de données en temps réel. Elle repose sur un modèle publish/subscribe, dans lequel des producteurs envoient des messages dans des topics, et des consommateurs les lisent à leur rythme.

1) Sources de données

Pour simuler un environnement réaliste, nous utilisons deux sources complémentaires.
- Le premier producteur interroge l'API OpenWeather toutes les 10 minutes et collecte des données complètes : température,temperature ressentie, humidité, vent(vitesse&direction), pression et conditions météo. Ces informations sont formatées en JSON et envoyées au topic Kafka weather-api.

```
payload = {
    "timestamp": current.get("dt"),
    "datetime": datetime.datetime.fromtimestamp(current.get("dt")).isoformat(),
    "temperature": current.get("temp"),
    "humidity": current.get("humidity"),
    "wind_speed": current.get("wind_speed"),
    "wind_deg": current.get("wind_deg"),
    "pressure": current.get("pressure"),
    "feels_like": current.get("feels_like"),
    "uvi": current.get("uvi"), #Indice UV
   "weather_main": current.get("weather")[0].get("main") if current.get("weather") else None,
    "weather_description": current.get("weather")[0].get("description") if current.get("weather") else None,

}
```

- Le second producteur simule un capteur IoT local qui génère des données partielles toutes les 10 minutes aussi. Il produit uniquement température, humidité et pression,

avec un format JSON différent. Ces données sont envoyées au topic capteur-iot.

```
def simulate_sensor():
    return {
        "temperature": round(random.uniform(15.0, 30.0), 2),
        "humidity": round(random.uniform(40.0, 80.0), 2),
        "pressure": round(random.uniform(0.0, 10.0), 2),
        "timestamp": datetime.utcnow().isoformat()
    }
```

Cette approche multi-sources permet de tester la fusion de données avec des priorités différentes et de simuler la redondance nécessaire en environnement de production. Elle reproduit également les défis réels du Big Data où il faut intégrer des sources hétérogènes avec des formats variables.

2) Configuration Kafka

Kafka est déployé via **Docker Compose** avec **Zookeeper** pour la coordination du cluster et un broker principal pour gérer les topics. L'interface Kafka-UI permet de superviser les messages en temps réel.
Les producteurs sont configurés pour envoyer les messages avec une fréquence de 10 minutes. Chaque topic maintient un historique des messages pour permettre la reprise en cas d'interruption (config par défaut de kafka).
La sérialisation JSON facilite l'interopérabilité entre les composants.

V. TRAITEMENT AVEC SPARK

Le cœur du pipeline repose sur S**park Structured Streaming** qui lit simultanément les deux topics Kafka.

- **Configuration de Spark**

La configuration de Spark Structured Streaming a nécessité plusieurs ajustements, en particulier sous Windows(OS que nous avons utilisé) :

Packages requis :
- ✓ spark-sql-kafka-0-10 pour connecter Spark à Kafka
- ✓ Le driver postgresql pour l'écriture dans PostgreSQL

Ressources allouées :
- ✓ 4 Go de mémoire pour le driver
- ✓ 4 Go pour l'executor

Compatibilité Windows :
- ✓ Chemins Python explicitement définis pour éviter les conflits d'environnement (notamment avec Anaconda ou Python système)

Paramétrage du streaming Kafka :
- ✓ Option startingOffsets mis à **latest**, afin d'éviter le retraitement complet

de l’historique Kafka .

- **Lecture ,nettoyage et fusion des données**.

Kafka lit de façon simultanée les deux topics Kafka (weather-api et capteur-iot) en mode streaming (ReadStream).
Les données JSON sont d'abord parsées selon leurs schémas respectifs, puis converties dans une structure standardisée.
Une logique de validation élimine les valeurs aberrantes : températures en dehors de la plage -40°C à +50°C, humidité hors de 0-100%.
La fusion privilégie les données du capteur local quand elles sont disponibles, car elles représentent les conditions réelles du site. La fonction **coalesce** de Spark permet cette logique de priorité. Les doublons sont éliminés sur la base du timestamp et de la source(api ou local).

- **Agrégations multi-temporelles**

Le système génère automatiquement trois niveaux d'agrégation grâce aux fenêtres glissantes de Spark.

- Les agrégations 10 minutes combinent les mesures reçues dans chaque fenêtre en privilégiant le capteur local. Les données sont ensuite moyennées avec validation de cohérence.
- Les agrégations horaires calculent les statistiques min, max et moyenne sur les données 10 minutes. Ces données servent de base aux prédictions.
- Les agrégations journalières offrent une vue d'ensemble des tendances pour les analyses à plus long terme.

Les **watermarks** gèrent les données en retard avec une tolérance de 5 minutes pour les données brutes et 1 heure pour les agrégations journalières. Cette configuration c’est pour équilibrer précision et performance.

- **Système de surveillance**

La classe **AlertManager** surveille le pipeline en continu et génère des alertes selon trois catégories.

- DATA_QUALITY: Les alertes de qualité détectent les valeurs aberrantes et les batches vides.
- SYSTEM_HEALTH : Les alertes système surveillent l'état des composants Spark.
- PROCESSING_ERROR : Les alertes de traitement capturent les erreurs d'exécution.

Chaque alerte est associée à un niveau de sévérité de LOW à CRITICAL.
Le système génère deux types de traces. Tous les événements du pipeline (traitement des batches, erreurs, prédictions générées, etc.) sont enregistrés dans le fichier *./logs/weather_pipeline.log* avec horodatage et niveau de sévérité. En parallèle, les alertes spécifiques détectées par le système de surveillance sont sauvegardées dans la table *weather_alerts* en base avec leurs détails complets pour être utilisé après pour le dashboard de monitoring.

La validation des batches vérifie automatiquement la cohérence des données avant sauvegarde. Elle compte les enregistrements, détecte les valeurs manquantes et identifie les anomalies statistiques. Cette approche préventive évite la corruption des données stockées.

VI. Modèle de prédiction

Nous avons utilisé un modèle simple reposant sur des principes physique , suffisant pour les besoins du projet. L'objectif n'était pas de construire un modèle avancé, mais d'intégrer une étape de prédiction dans le pipeline. Des améliorations sont possibles en utilisant des modèles de machine learning ou de deep learning, mais cela sortait du cadre de ce travail.

- **Approche algorithmique**

  Le modèle de prédiction utilise une approche empirique basée sur l'historique des 6 dernières heures.

  L'algorithme combine plusieurs facteurs physiques pour estimer l'évolution de la température.

  - ✓ moyenne pondérée:

    La base du calcul repose sur une moyenne pondérée des températures récentes, où les valeurs les plus récentes ont un **poids plus important**.

    Cette approche reflète l'inertie thermique naturelle tout en restant sensible aux changements récents.

  - ✓ régression linéaire:

    Une régression linéaire sur les 6 derniers points détermine la tendance thermique. Cette tendance est ensuite pondérée par un facteur décroissant pour les prédictions H+1 à H+4, reflétant l'augmentation de l'incertitude avec l'horizon temporel.

- Corrections climatiques

  Le modèle intègre des corrections basées sur les conditions météorologiques actuelles.

  - ✓ L'humidité relative:

    Elle influence la température ressentie et les échanges thermiques. Une humidité élevée (>80%) produit un effet refroidissant, tandis qu'une humidité faible (<30%) favorise le réchauffement.

  - ✓ Le Vent:

    Le vent génère un refroidissement par **convection forcée.** L'algorithme applique une correction progressive selon la vitesse : refroidissement modéré entre 10 et 20 km/h, fort au-delà.

    Cette approche empirique reproduit les phénomènes physiques observés.

- **Implémentation dans le pipeline**

  Les prédictions sont générées automatiquement à chaque nouvelle donnée horaire. La fonction generate_predictions est appelée par Spark via le

mécanisme foreachBatch. Elle récupère les 8 dernières heures depuis PostgreSQL, applique le modèle et sauvegarde les résultats.

Un **score de confiance** accompagne chaque prédiction. Il se base sur la variance des températures récentes : *plus les conditions sont stables, plus la confiance est élevée*. Ce score aide les utilisateurs à interpréter la fiabilité des prédictions.

Le modèle stocke ses résultats dans la table weather_predictions avec

- timestamp de génération
- prédictions H+1 à H+4
- score de confiance.

Ces données sont ensuite utilisées pour alimenter le dashboard .

VII. Stockage des Données

PostgreSQL centralise toutes les données du pipeline dans un schéma relationnel optimisé. Cette approche facilite les requêtes du dashboard. La base contient cinq tables principales correspondant aux différents niveaux d'agrégation et fonctionnalités.

- **weather_raw_10min :**
  stocke les données fusionnées toutes les 10 minutes et sert de référence pour toutes les analyses suivantes :
  - ✓ timestamp : horodatage de la fenêtre d'agrégation
  - ✓ temperature, humidity, pressure : valeurs «moyennes» des mesures reçues
  - ✓ wind_speed, wind_deg : vitesse et direction du vent (API uniquement)
  - ✓ feels_like : température ressentie
  - ✓ weather_main, weather_description : conditions météo textuelles

- **Les tables weather_hourly et weather_daily** contiennent les statistiques calculées par Spark et alimentent les graphiques d'analyse :
  - ✓ timestamp : heure ou date de la période
  - ✓ temperature_avg, temperature_min, temperature_max : statistiques thermiques
  - ✓ humidity_avg, pressure_avg, wind_speed_avg, feels_like_avg : moyennes des autres paramètres
  - ✓ weather_main : condition météo dominante
- **La table weather_predictions** archive chaque série de prédictions
  - ✓ prediction_time : moment où la prédiction a été calculée
  - ✓ h_plus_1, h_plus_2, h_plus_3, h_plus_4 : températures prédites pour les 4 prochaines heures
  - ✓ confidence : score de fiabilité de la prédiction (0 à 1)
- **La table weather_alerts** enregistre tous les incidents détectés pour faciliter le monitoring:
  - ✓ alert_type : catégorie (data_quality, system_health, processing_error)

- ✓ severity : niveau (low, medium, high, critical)
- ✓ message : description de l'incident
- ✓ component : élément du pipeline concerné (Spakr,Kafka,etc)
- ✓ metadata : détails techniques en JSON
- ✓ created_at : timestamp de détection
- ✓ resolved : statut de résolution (pas exploité)

Des index sur les colonnes **timestamp** accélèrent les requêtes temporelles fréquentes du dashboard.

VIII. Visualisation & monitoring

Le dashboard est construit avec une architecture web en deux parties distinctes. Le backend utilise Flask, qui sert d'intermédiaire entre la base PostgreSQL et l'interface utilisateur.

Flask expose trois endpoints principaux :

- ◆ **/api/current** qui retourne les dernières données météo avec timestamp, température, humidité, vent et conditions actuelles
- ◆ **/api/predictions** qui fournit les quatre prédictions horaires avec leurs scores de confiance
- ◆ **/api/alerts** qui liste les alertes avec leurs détails

Le frontend est développé en *HTML, CSS et JavaScript* pour l'interactivité.
Chart.js, une bibliothèque de graphiques JavaScript, génère tous les éléments visuels : graphiques linéaires, barres, radar charts. L'interface communique avec le backend via des requêtes AJAX qui récupèrent les données JSON depuis les endpoints Flask.

Le dashboard met à jour automatiquement les données toutes les deux minutes(surtout pour les alertes ,parce que pour les autres données c'est minimum 10 minutes pour en avoir de nouvelles) en interrogeant les endpoints de l'API.

Les graphiques sont interactifs(fonctionnalités natives de chatjs), avec des fonctions de zoom,et des infobulles détaillées.
En cas de problème de connexion, le système affiche des messages d'erreur clairs et propose une tentative de reconnexion.

Le dashbord est composé de 3 onglets:

1. Onglet "Vue d'ensemble"

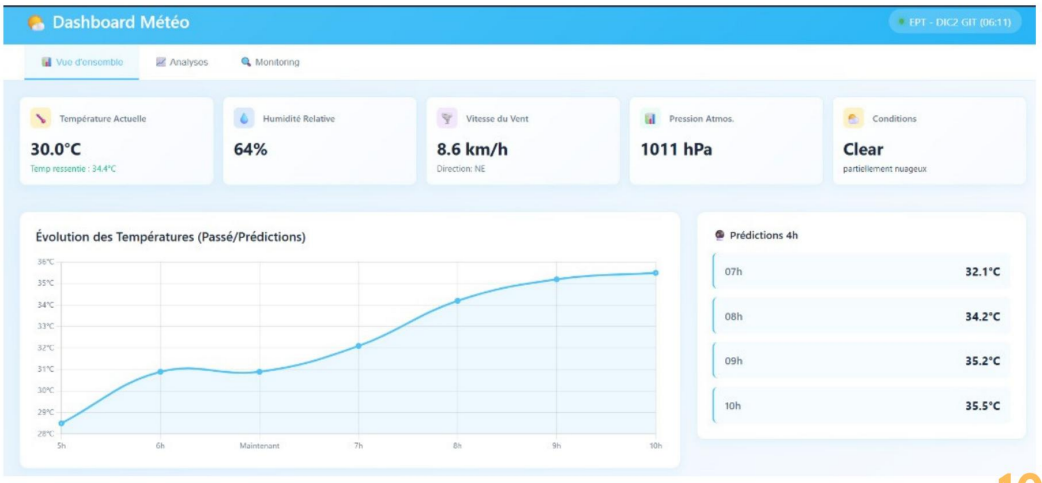
Cet onglet centralise les informations essentielles pour un aperçu rapide de la

situation météo. On y retrouve cinq cartes d'information :

- ✓ la température actuelle avec la température ressentie
- ✓ l'humidité relative en pourcentage
- ✓ la vitesse du vent en km/h avec sa direction (N, NE, E, etc.)
- ✓ la pression atmosphérique en hPa
- ✓ enfin les conditions météo textuelles (Clear, Cloudy, Rain, etc.) tirés de openweathermap.

Un graphique(courbe) montre l'évolution des températures sur 7 heures(2 h avant , l'heure actuelle et les 4 prédictions futures).
Pour plus de lisibilité un panneau latéral liste les 4 prédictions (heure - temp moyenne prédite).

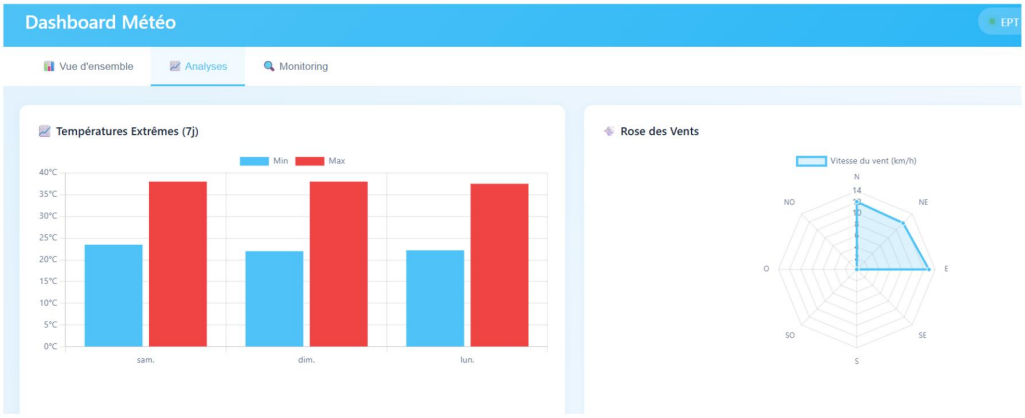
Dashboard de visualisation **Vue d'ensemble**



2. *Onglet "Analyses"*

Cet onglet contient deux graphiques pour étudier les tendances météo :

- Un histogramme des températures min/max sur les 7 derniers jours, permettant de visualiser les amplitudes thermiques.
- Une rose des vents en radar chart, qui montre les directions dominantes et la force moyenne du vent.



3. *Onglet "Monitoring"*

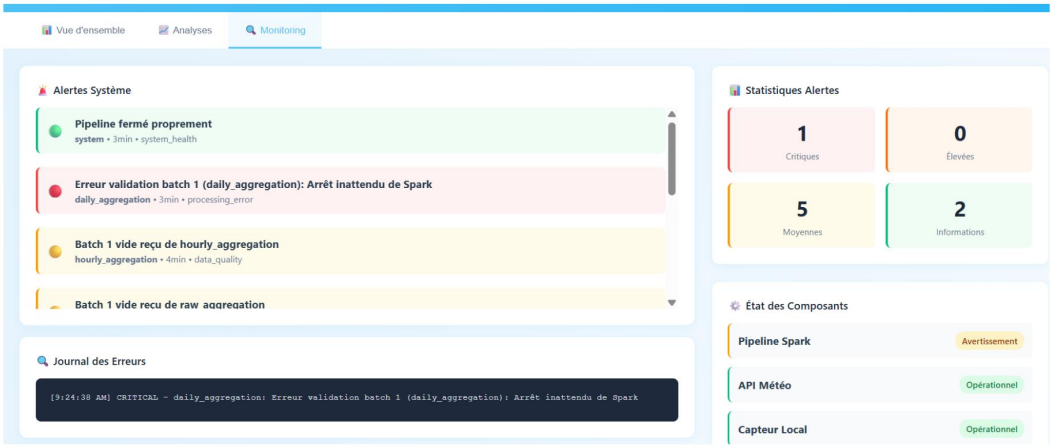
L'onglet "Monitoring" assure le suivi technique du système en temps réel.

Cet onglet ne devrait ,normalement ,pas être sur la même interface que les autres,vu que c'est plus une fonctionnalité admin .

Il comprend :

- Une section Alertes sous forme de tableau listant les alertes actives avec leur horodatage, gravité (codée par couleur), message, et composant concerné.
- Un journal des erreurs affiché dans un terminal simulé, contenant les derniers logs techniques (timestamp, niveau, message).
- Un état des composants via trois cartes : statut du pipeline Spark, connectivité avec l'API météo, et réception des données du capteur local.
- Des statistiques d'alertes récapitulant le nombre d'alertes par niveau (CRITICAL, HIGH, MEDIUM, INFO) avec des compteurs colorés.

Dashboard de visualisation Monitoring



IX. Difficultés rencontrées et solutions mises en œuvre

- **Problèmes de mémoire avec Spark**

  L'installation locale a été compliquée par des erreurs de type OutOfMemoryError. Il a fallu ajuster les paramètres executor.memory et driver.memory, finalement fixés à 4G chacun après plusieurs essais.

- **Connexion Spark–Kafka sous Windows**

  Des erreurs de classpath et des conflits entre versions JAR sont apparus.
  La solution : déclarer les variables d'environnement Python avant tout import Spark, utiliser spark.jars.packages pour que Spark télécharge automatiquement les connecteurs adaptés (Kafka et PostgreSQL).

- **Tests du pipeline trop lents**

  Les agrégations horaires et journalières nécessitaient d'attendre des heures pour obtenir des données, ce qui ralentissait les validations et surtout causait des problèmes de mémoire.
  Pour pouvoir avoir assez de données :
  une fois le schéma validé, des scripts SQL ont été utilisés pour peupler manuellement les tables weather_hourly et weather_daily avec des données réalistes, ce qui a permis de tester le dashboard immédiatement, sans attendre l'accumulation naturelle des flux .

X. Conclusion

Ce projet a abouti à la mise en place d'un pipeline Big Data complet et opérationnel, couvrant toutes les étapes : de l'ingestion des données météo en temps réel à leur visualisation, en passant par le traitement et les prédictions automatiques.
L'utilisation de technologies variées (Kafka, Spark, PostgreSQL, Flask) a permis de mesurer la richesse mais aussi la complexité des architectures Big Data modernes. Le système de monitoring renforce la fiabilité de l'ensemble, ce qui le rend adaptable à des contextes proches de la production.
Ce projet a permis de consolider des compétences concrètes sur toute la chaîne Big Data : conception d'une architecture distribuée, gestion de flux en temps réel, stockage structuré, et création d'interfaces interactives.
Des pistes d'amélioration restent possibles, notamment :

- ✓ l'ajout de modèles prédictifs plus avancés (machine learning, deep learning),
- ✓ l'intégration de nouvelles sources météo,
- ✓ ou le déploiement dans un environnement cloud avec mise à l'échelle automatique.

Ces prolongements offriraient un cadre idéal pour approfondir les compétences acquises et explorer de nouvelles dimensions du Big Data.

XI. Sources et références

https://kafka.apache.org/documentation/
https://medium.com/@kiranvutukuri/spark-structured-streaming-workflow-pa-b20cf45ac99c
https://nimblewasps.medium.com/mastering-memory-management-in-apache-spark-understanding-the-bridge-between-spark-and-jvm-memory-99d5c95ee774
https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html
https://www.youtube.com/watch?v=uD_q4Rm4i2Q&t=1555s
https://claude.ai/
https://chatgpt.com/
https://openclassrooms.com/fr/courses/8493836-realisez-des-calculs-distribues-avec-spark