GitHub PR Big Data Pipeline

A Real-Time and Batch Data Processing System

Prepared by:
Alaeddine ACHACH
Idris SADDI
Mohamed Yassine ELLINI
Hatem GHARSALLAH

National Institute of Applied Sciences and Technologies (INSAT)
University of Carthage, Tunisia

June 2025

GitHub repo :
https://github.com/Ala-Eddine-Achach/Big-Data-project

Abstract

This report presents the design and implementation of a scalable Big Data pipeline for ingesting, processing, and visualizing GitHub Pull Request (PR) data in near real-time. The architecture combines **streaming** and **batching** paradigms to support both live monitoring and historical analytics, providing comprehensive visibility into collaborative software development activity.

The pipeline integrates **Apache Kafka** for event streaming, **MongoDB** as a **data lake** for storing both raw and aggregated data, **Apache Spark** for batch analytics, and **Flask-SocketIO** for real-time delivery of insights to a web-based **dashboard**.

By combining decoupled **stream processing**, efficient **batch data transformations**, and interactive visualization, the system enables actionable insights into software collaboration patterns at scale. This report details the architecture, component design, testing approach, and key lessons from implementing an end-to-end Big Data solution.

Contents

1 Introduction

In modern data engineering, the ability to combine **streaming** and **batching** data processing is essential for building scalable analytics pipelines. In this project, we apply these techniques to the domain of collaborative software development by analyzing GitHub Pull Requests (PRs).

PRs encapsulate valuable signals about contributor behavior, team dynamics, and project evolution. Understanding these signals requires both **real-time streaming** of current activity and **batch analysis** of historical trends.

Our goal is to design a robust data pipeline that:

- Ingests GitHub PR data continuously via **streaming**.
- Stores raw and processed data in a central **data lake** (MongoDB).
- Performs **batch analytics** to extract time-based metrics.
- Delivers insights through a real-time, interactive **dashboard**.
- Scales to support large and evolving datasets.

To achieve this, we implement a hybrid architecture:

- **Apache Kafka** provides a resilient and scalable **event streaming** backbone.
- **MongoDB** serves as both a **data lake** and a real-time data source for the dashboard.
- **Apache Spark** performs periodic **batch processing** of PR data.
- **Flask-SocketIO** enables low-latency **streaming** of updates to the frontend **dashboard**.

The resulting architecture demonstrates core data engineering principles and provides a practical example of combining **streaming** and **batching** techniques to deliver real-time analytics.

2 Architecture Overview

The system is designed as a hybrid data pipeline that integrates **stream processing**, **batch analytics**, and a centralized **data lake** to support scalable and low-latency analysis of GitHub Pull Request (PR) data.

A combination of **streaming** and **batching** components ensures that the system provides both immediate insights and long-term historical trends. The architecture also includes an interactive **dashboard** for visualizing key metrics.

2.1 Data Flow

1. **Streaming Ingestion**: A Kafka producer continuously fetches PR data from the GitHub API and streams it into a Kafka topic.
2. **Stream Processing + Data Lake Ingestion**: A Kafka consumer writes PR messages into the **raw_prs** collection of the **data lake** (MongoDB).
3. **Batch Analytics**: An Apache Spark job performs periodic **batch processing** of PR data and stores aggregated results in the **analytics** collection.
4. **Real-time Streaming to Dashboard**: A Flask-SocketIO backend continuously streams updates from MongoDB Change Streams and Kafka messages to the frontend **dashboard**.
5. **Visualization**: The React-based **dashboard** displays both historical analytics and live streaming data.

2.2 Technology Stack

- **Apache Kafka**: Distributed **streaming** platform for real-time data ingestion.
- **MongoDB Data Lake**: NoSQL database serving as a **data lake** for raw and processed PR data.
- **Apache Spark**: Engine for large-scale **batch analytics**.
- **Flask-SocketIO**: WebSocket-based server for **real-time streaming** to the **dashboard**.
- **React Dashboard**: Interactive frontend for visualizing both **streaming** updates and **batch analytics**.
- **Docker Compose**: Container orchestration for reproducible pipeline deployment.

High-Level Architecture Diagram

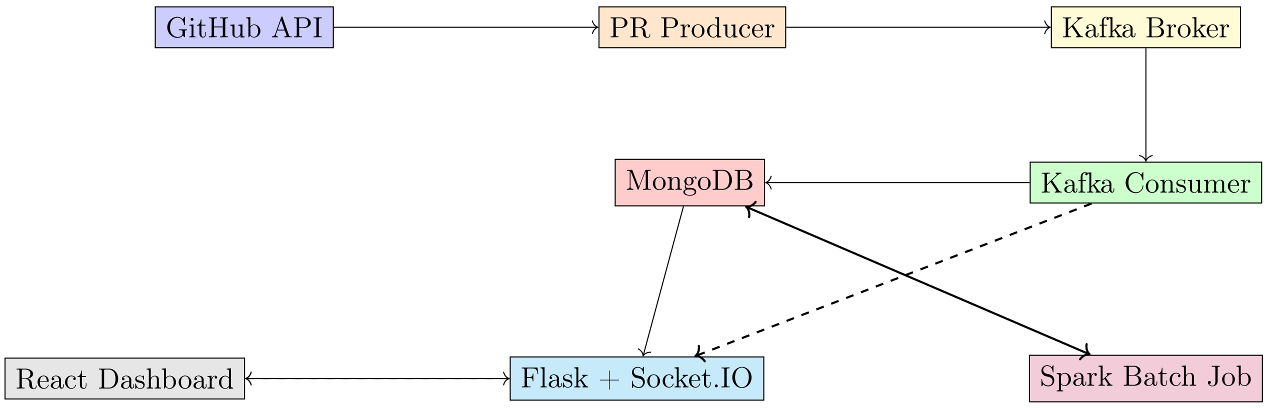

Figure 1: System Architecture of the GitHub PR Big Data Pipeline

3 Component Breakdown

3.1 GitHub PR Producer

The PR Producer is a Python-based **streaming data ingestion** component. It fetches pull request data from the GitHub API and streams it into the Kafka messaging layer.

- Periodically fetches PR data in batches from GitHub.
- Streams PR records into the Kafka topic `github-prs-topic`.
- Implements retry and backoff strategies for API and Kafka operations.

3.2 Apache Kafka Broker

Kafka acts as the core **stream processing** backbone of the architecture. It decouples data ingestion from downstream processing and enables scalable, fault-tolerant delivery of PR records.

- Provides a **streaming** pipeline from the PR Producer to downstream consumers.
- Supports high-throughput, distributed processing.
- Enables multiple consumers to read data independently.

3.3 Kafka Consumer to Data Lake (MongoDB)

A dedicated Kafka consumer writes PR records into the **data lake** (MongoDB), serving as the raw data store for both real-time and batch processing.

- Consumes PR records from Kafka.
- Writes records into the `raw_prs` collection of the MongoDB **data lake**.
- Uses `upsert` logic to maintain an accurate, deduplicated raw data store.

3.4 MongoDB Data Lake

MongoDB functions as the pipeline's **data lake**, supporting both operational storage and batch analytics.

- `raw_prs` stores unprocessed PR records ingested via the Kafka Consumer.
- `analytics` stores results of **batch analytics** from Spark.
- Replica set configuration enables **streaming change events** via MongoDB Change Streams.
- Acts as the unified **data lake** for both streaming updates and historical batch analysis.

3.5 Apache Spark Batch Job

The Spark Batch Job performs periodic **batch processing** of PR data to compute time-based aggregations and advanced metrics.

- Reads raw PR data from the **raw_prs** collection in the **data lake**.
- Filters and transforms merged PRs.
- Aggregates data by time slot and weekday to produce actionable insights.
- Writes batch results into the `analytics` collection in the **data lake**.

3.6 Flask + Socket.IO Backend

The Flask-SocketIO backend provides a low-latency **streaming** layer that connects the **data lake** and Kafka to the interactive dashboard.

- Emits `initial_analytics` and `initial_prs` data on client connection.
- Subscribes to MongoDB Change Streams to emit `data_update` and `data_delete` events.
- Consumes Kafka messages in parallel and streams them via `raw_pr_update` events.
- Serves as the core **real-time streaming API** for the dashboard.

3.7 React Dashboard

The React-based dashboard provides a user-friendly interface to visualize both **streaming** and **batch** analytics results.

- Connects to the Flask-SocketIO backend via WebSockets.
- Renders real-time PR updates and batch-aggregated metrics.
- Displays dynamic visualizations of PR activity and trends using charts and heatmaps.

4 Real-time Data Flow

4.1 Streaming Architecture

The backend architecture was refactored to support a fully **streaming-first** design, replacing traditional REST-based APIs. Flask-SocketIO provides a persistent WebSocket connection to stream live updates to clients.

4.2 Streaming Components

- **Kafka Consumer Thread**: Runs inside the Flask backend, continuously consuming PR updates and emitting `raw_pr_update` events.
- **MongoDB Change Streams**: Captures insertions, updates, and deletions in the **analytics** collection, triggering `data_update` and `data_delete` events.
- **WebSocket API**: Provides a single persistent connection for streaming both initial state and incremental updates to the React dashboard.

4.3 Benefits of Streaming Architecture

- **Low Latency**: Clients receive PR updates and analytics changes in near real-time.
- **Unified Pipeline**: Both **streaming** data (Kafka) and **batch** data (MongoDB) are streamed via the same WebSocket API.
- **Simplicity**: The dashboard requires no polling or periodic refresh — all data flows reactively via WebSocket events.

5 Data Processing and Analytics

While real-time updates provide immediate visibility into GitHub PR activity, historical trends and aggregated metrics require batch processing.

The Spark batch job performs the following analytics:

- Filters merged PRs to focus on completed contributions.
- Calculates merge time in hours for each PR.
- Aggregates PR merges by time slots (4-hour intervals) and weekdays.
- Computes average merge times and PR counts per time slot and day.

These analytics enable the dashboard to display:

- Average merge time heatmaps.
- Temporal patterns of PR activity.
- Historical trends in repository contribution velocity.

This hybrid approach (streaming + batch) provides both immediate and strategic insights for engineering teams.

6 Testing and Validation

6.1 Kafka Pipeline Validation

We validated the Kafka ingestion pipeline by:

- Logging PR records sent from the `GitHub PR Producer`.
- Verifying that the `kafka_to_mongo_consumer` correctly receives messages and inserts them into MongoDB.
- Ensuring that duplicate PR records are not inserted thanks to `upsert` logic.

Kafka was found to provide reliable and consistent message delivery throughout the testing process.

6.2 MongoDB Validation

MongoDB was tested using the following approaches:

- Manual querying of `raw_prs` and `analytics` collections.
- Integration tests using `test_mongo_connection.py` script.
- Verifying that Spark jobs correctly update analytics data.
- Monitoring change streams to ensure real-time update events are emitted.

6.3 WebSocket Validation

We tested the WebSocket pipeline by:

- Connecting the dashboard and monitoring received events.
- Verifying that `initial_analytics` and `initial_prs` are received on connect.
- Verifying that `data_update`, `raw_pr_update`, and `data_delete` events are emitted in response to database changes.
- Validating connection stability over long-running dashboard sessions.

The WebSocket connection was found to be stable and low-latency during all tests.

6.4 End-to-End Test

Finally, we tested the full pipeline:

1. Starting from GitHub PR fetching.
2. Through Kafka ingestion and MongoDB storage.
3. Spark batch job aggregation.
4. WebSocket-based real-time updates on the dashboard.

The system operated as intended with all components working seamlessly together.

7 Challenges and Lessons Learned

7.1 Handling Real-Time Updates

One of the key challenges was implementing a reliable real-time pipeline for updates. We initially considered REST APIs but moved to WebSocket-based architecture for better scalability and lower latency.

7.2 MongoDB Change Streams

MongoDB Change Streams required configuring the database as a replica set. This added complexity but enabled robust detection of data changes and allowed for real-time push updates to the dashboard.

7.3 Kafka Consumer and Offset Management

Managing Kafka consumer offsets and ensuring idempotent processing in the MongoDB consumer required careful testing. Using `update_one(..., upsert=True)` helped simplify the pipeline.

7.4 Spark Batch Job Design

Designing the Spark job to compute meaningful analytics was an iterative process. We optimized the job to filter merged PRs, calculate merge times, and group data in time slots and weekdays to produce useful visualizations.

7.5 Deployment and Monitoring

Coordinating multiple services via Docker Compose highlighted the importance of robust health checks and service dependencies. We also identified future improvements related to monitoring Kafka consumer lag and MongoDB status.

8 Conclusion

This project demonstrates a practical implementation of a hybrid **streaming** and **batching** data pipeline for real-time analytics of GitHub pull request activity.

By combining **Apache Kafka**, a **MongoDB data lake**, **Apache Spark**, and a **real-time dashboard**, the system delivers both immediate visibility and long-term insights into software collaboration.

Key outcomes include:

- Continuous **streaming** ingestion of PR data.
- Scalable **batch processing** for historical trends.
- Low-latency delivery of updates to an interactive **dashboard**.

The architecture is modular and can be extended to larger data volumes, additional GitHub events, or advanced analytics. It serves as a strong example of applying modern **data engineering** patterns to real-world use cases.