HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

REAL-TIME X (TWITTER) SENTIMENT ANALYSIS

Project Report

Supervisor: Ph.D. Tran Viet Trung

Presented by: Group 21
Nguyen Duc Anh – 20225468
Nguyen Hai Duong – 20214887
Bui Thi Thu Uyen – 20225537
Nguyen Quoc Thai – 20225456
Tran Thi Hien – 20214896

Hanoi – 2025

Contents

1 Problem Definition

1.1 Selected Problem

The selected problem of this project is **real-time sentiment analysis of Twitter (X) data**. Twitter is a social media platform where users continuously publish short text messages expressing opinions on products, political events, and social issues. These messages form a high-velocity data stream whose analytical value is strongly time-dependent.

The objective of the system is to automatically classify incoming tweets into sentiment categories (positive, negative, or neutral) as they arrive, enabling near-real-time insights into public opinion trends.

It is important to note that the primary focus of this project is not the development of an advanced natural language processing model, but rather the design and implementation of a scalable, fault-tolerant Big Data streaming pipeline capable of handling continuous data ingestion and processing.

1.2 Suitability of the Problem for Big Data

This problem is inherently suitable for a Big Data solution due to the following characteristics:

- **Volume:** Twitter generates a massive number of tweets every day, making traditional single-node storage and processing solutions impractical.
- **Velocity:** Tweets are produced continuously and must be processed with low latency to preserve their analytical relevance.
- **Variety:** The data consists of unstructured text combined with semi-structured metadata such as timestamps, user identifiers, and engagement statistics.

Traditional batch-processing approaches are insufficient for this scenario because they introduce high latency and delay actionable insights. Likewise, tightly coupled systems that directly connect data sources to processing engines lack resilience under failure conditions. Therefore, a distributed, streaming-based Big Data architecture using systems such as Apache Kafka and Apache Spark Streaming is required to ensure scalability, reliability, and real-time processing.

1.3 Scope and Limitations of the Project

1.3.1 Scope

The scope of this project focuses on the design and implementation of an end-to-end real-time Big Data pipeline rather than on advanced sentiment model optimization. Specifically, the project includes:

- Real-time ingestion of tweet data from the Twitter/X API as well as simulated high-volume data from a static dataset.

- Stream processing and sentiment classification using Apache Spark Streaming.
- Persistent storage of processed sentiment results in MongoDB to support downstream analysis and visualization.

1.3.2 Limitations

Despite demonstrating a complete streaming pipeline, the project has several limitations:

- **API Restrictions:** Live data ingestion is constrained by Twitter/X API rate limits, especially under free or academic access tiers.
- **Sentiment Model Accuracy:** The sentiment analysis relies on the VADER lexicon, which is computationally efficient but limited in handling sarcasm, domain-specific language, and complex contextual cues.
- **Production Readiness:** The system is implemented as a proof-of-concept and does not include advanced security hardening, large-scale cluster deployment, or automated scaling mechanisms.

2 Architecture and Design

2.1 Overall Architecture (Kappa Architecture)

The system follows the **Kappa Architecture** model, which is centered around a single, continuous streaming pipeline. All data—regardless of origin—is processed in real time without a separate batch layer.

In this project, both live Twitter/X data and offline Kaggle CSV datasets are ingested as event streams into Apache Kafka. Apache Spark Structured Streaming consumes these events for real-time sentiment analysis, and MongoDB is used as the final persistent storage. This design avoids duplication of logic between batch and streaming jobs, reduces operational complexity, and aligns well with real-time analytics requirements.

2.2 System Components and Their Roles

2.2.1 Data Ingestion Layer

The data ingestion layer is implemented using two Kafka producers, each serving a distinct purpose:

- **Simulated CSV Producer:** Implemented in `producer-validation-tweets.py`, this producer reads a Kaggle CSV file line by line and publishes each record as a message to a Kafka topic. A fixed delay between messages is introduced to simulate real-time data arrival. This producer is primarily used for controlled testing and development.

- **Live Twitter/X Producer:** Implemented in `producer_x.py`, this producer connects to the Twitter/X API v2 using Tweepy and fetches recent tweets based on a query. Each tweet is converted into a structured JSON message containing metadata such as tweet ID, text, timestamp, language, and engagement metrics, then sent to Kafka.

Both producers publish messages to Kafka using the `kafka-python` client library.

2.2.2 Streaming Layer (Apache Kafka)

Apache Kafka acts as the central event log and message broker for the entire system.

- All tweet data from different producers is published to Kafka topics.
- Kafka decouples data ingestion from processing, allowing producers and consumers to operate independently.
- Message durability and fault tolerance are guaranteed through Kafka's log-based storage mechanism.

2.2.3 Processing Layer (Apache Spark Streaming)

The processing layer is implemented using Apache Spark Structured Streaming.

- Spark reads streaming data directly from Kafka topics.
- Incoming messages are deserialized from JSON format and mapped to a predefined schema.
- Sentiment analysis is performed using the NLTK VADER sentiment analyzer, either inside a Spark UDF or during micro-batch processing.
- Each tweet is classified into one of three categories: *positive*, *negative*, or *neutral.*

Two processing approaches are demonstrated in the project:

- A `foreachBatch` approach that writes results to MongoDB using PyMongo.
- A native MongoDB Spark Connector approach that writes streaming DataFrames directly to MongoDB.

2.2.4 Storage Layer (MongoDB)

MongoDB is used as the persistent storage backend for processed streaming data.

- Each processed tweet is stored as a document containing the original text, sentiment label, source, and metadata.

- The schema-less design allows flexibility in storing heterogeneous tweet attributes.
- MongoDB supports high write throughput, which is suitable for streaming workloads.

Connection parameters such as MongoDB URI, database name, and collection name are externalized using environment variables defined in a `.env` file.

2.3 Data Flow and Component Interaction

The end-to-end data flow of the system is summarized as follows:

1. Tweet data is collected either from a Kaggle CSV file or the Twitter/X API.
2. Producers serialize each tweet into a JSON message.
3. Messages are published to Apache Kafka topics.
4. Spark Structured Streaming consumes data from Kafka in micro-batches.
5. Sentiment analysis is applied to tweet text using the VADER model.
6. The sentiment-labeled results are written to MongoDB for persistence and downstream visualization.

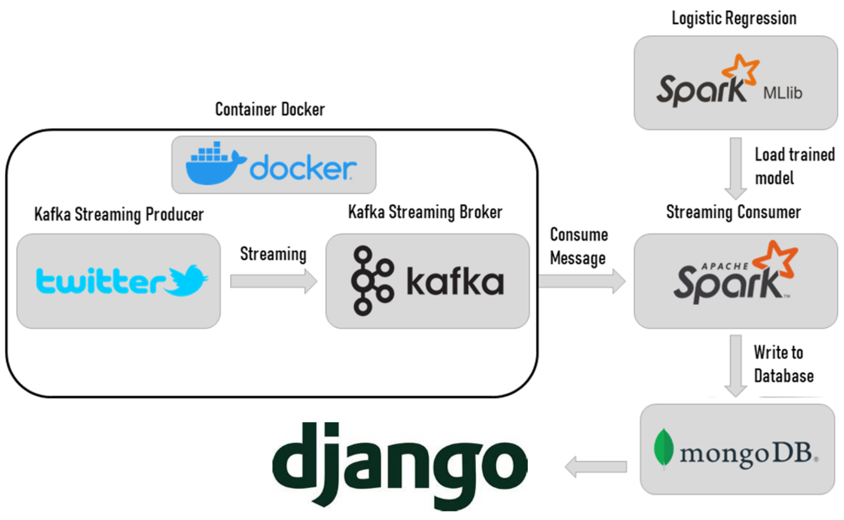

Figure 1: Data Flow and Component Interaction in the Kappa Architecture

3 Implementation Details

3.1 Source Code Organization and Documentation

The project source code is organized into clearly separated modules based on system responsibilities.

- **Kafka Producers:**
  - `producer-validation-tweets.py` handles simulated streaming by reading a Kaggle CSV file and publishing each record to Kafka with a controlled delay.
  - `producer_x.py` connects to the Twitter/X API v2 using Tweepy and streams live tweets into Kafka in JSON format.
- **Spark Consumers:**
  - `consumer_spark.py` demonstrates micro-batch processing using `foreachBatch`, performing sentiment analysis and writing results to MongoDB via PyMongo.
  - `sparkconsumer_x.py` implements a fully structured streaming pipeline using Spark UDFs and the MongoDB Spark Connector.

Each script contains inline comments explaining key logic, configuration parameters, and processing steps. This modular structure improves readability, debugging, and maintainability.

3.2 Environment-Specific Configuration

To avoid hardcoding sensitive information, all environment-dependent parameters are externalized using a `.env` configuration file.

The configuration includes:

- Twitter/X API Bearer Token
- Kafka bootstrap server address
- Kafka topic names
- MongoDB connection URI, database, and collection names

The `python-dotenv` library is used to load these variables at runtime. This approach ensures better security, simplifies deployment across environments, and allows configuration changes without modifying source code.

3.3 Deployment Strategy

The system is designed to be deployed locally in a modular and incremental manner.

1. Start infrastructure services (Kafka, Zookeeper, MongoDB) using Docker Compose.
2. Launch Kafka producers (CSV-based or Twitter/X-based) to ingest data.
3. Submit Spark Streaming jobs using `spark-submit` with required Kafka and MongoDB connector packages.
4. Verify data persistence by checking MongoDB collections.

Because the system follows Kappa Architecture, there is no separate batch deployment. All data is processed through the same streaming pipeline, simplifying operational management.

3.4 Monitoring and Logging

Basic monitoring and observability are implemented using built-in logging mechanisms.

- Spark log level is configured to `WARN` to reduce verbosity while preserving error visibility.
- Producers log message counts and fetch status to the console.
- Spark Structured Streaming provides runtime metrics such as batch duration and processing progress.
- MongoDB insertion can be verified directly via database inspection.

While no dedicated monitoring stack (e.g., Prometheus or Grafana) is deployed, the current setup is sufficient for academic evaluation and functional verification of the streaming pipeline.

4 Lessons Learned

4.1 Lessons on Data Ingestion

Lesson 1: Handling Multiple Data Sources in a Unified Pipeline

Problem Description The system ingests data from two heterogeneous sources: live Twitter/X API data and offline Kaggle CSV datasets. These sources differ in schema, arrival rate, and data completeness, which complicates unified processing.

Approaches Tried

- Separate Kafka topics and separate Spark jobs for each source.
- A unified Kafka topic with standardized JSON messages.

The first approach increased system complexity and duplicated processing logic.

Final Solution All producers serialize data into a unified JSON schema before publishing to Kafka. Spark consumes a single stream and applies one processing pipeline.

Key Takeaways

- Early schema standardization simplifies downstream processing.
- Kappa Architecture benefits from treating all data as event streams.

4.2 Lessons on Data Processing with Spark

Lesson 2: Trade-offs Between foreachBatch and Native Connectors

Problem Description Two Spark writing strategies were explored: manual insertion using `foreachBatch` and native MongoDB Spark Connector.

Approaches Tried

- `foreachBatch` with PyMongo.
- Structured Streaming output using MongoDB Spark Connector.

Final Solution The MongoDB Spark Connector was chosen for the final pipeline due to better performance and cleaner integration.

Key Takeaways

- Native connectors reduce boilerplate code.
- `foreachBatch` is flexible but harder to optimize at scale.

4.3 Lessons on Stream Processing

Lesson 3: Importance of Checkpointing and Recovery

Problem Description Without checkpointing, Spark Streaming jobs lost progress after restarts, leading to data reprocessing.

Approaches Tried

- Running Spark jobs without checkpoints.
- Enabling checkpoint directories for streaming queries.

Final Solution Checkpointing was enabled to store offsets and streaming state, ensuring recovery from failures.

Key Takeaways

- Checkpointing is mandatory for reliable stream processing.
- Exactly-once semantics depend on proper offset management.

4.4 Lessons on Data Storage

Lesson 4: Schema Flexibility vs Consistency in MongoDB

Problem Description Tweet metadata fields vary across sources and API responses, leading to inconsistent document structures.

Approaches Tried

- Strict schema enforcement before insertion.
- Flexible document-based storage with optional fields.

Final Solution MongoDB's schema-less design was leveraged, while enforcing a minimal required field set.

Key Takeaways

- Schema flexibility is valuable for semi-structured streaming data.
- A minimal contract schema is still necessary.

4.5 Lessons on Monitoring and Debugging

Lesson 5: Logging as the Primary Debugging Tool

Problem Description Lack of centralized monitoring made it difficult to trace failures across Kafka, Spark, and MongoDB.

Approaches Tried

- Default verbose Spark logging.
- Reduced log level with targeted print-based diagnostics.

Final Solution Spark log level was set to `WARN`, and key pipeline stages were logged explicitly.

Key Takeaways

- Clear, minimal logging is more effective than excessive verbosity.
- Most streaming bugs are detected via logs, not dashboards.

4.6 Lessons on Scaling and Performance

Lesson 6: Partitioning and Throughput Balance

Problem Description Initial Spark jobs suffered from uneven load and slow processing due to default partition settings.

Approaches Tried

- Default Spark partition configuration.
- Manual tuning of shuffle partitions.

Final Solution Partition counts were tuned based on workload size, improving throughput and stability.

Key Takeaways

- Default Spark settings are rarely optimal.
- Partition tuning has a direct impact on streaming latency.

4.7 Lessons on Fault Tolerance

Lesson 7: Kafka as a Buffer for Failure Isolation

Problem Description Temporary Spark failures risked data loss during continuous ingestion.

Approaches Tried

- Direct ingestion into Spark.
- Kafka-mediated ingestion with retained messages.

Final Solution Kafka was used as a durable buffer, allowing Spark consumers to restart without losing data.

Key Takeaways

- Kafka decouples ingestion from processing.
- Durable logs are essential for fault-tolerant streaming systems.