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 on Twitter/X-like streams. The system continuously ingests short text messages (tweets) together with basic metadata and automatically assigns each incoming tweet to one of three sentiment classes: *positive*, *negative*, or *neutral*.

In this implementation, tweets are ingested from two sources:

- **Live stream (X API):** recent-search polling via the X API and publishing to Kafka.
- **High-volume simulation (Kaggle CSV):** a static dataset replayed as a stream and published to Kafka.

The goal is **near-real-time operational insight**: not merely producing sentiment labels, but doing so under a streaming architecture that can tolerate failures, scale horizontally, and support downstream storage and visualization.

Important: the project does not claim to build a state-of-the-art NLP model. Sentiment classification is intentionally implemented with a lightweight lexicon approach (VADER) to keep the spotlight on Big Data streaming design and integration rather than model optimization.

1.2 Scope and Limitations of the Project

1.2.1 Scope

The scope of this project is an end-to-end streaming pipeline (Kappa-style), covering ingestion, processing, storage, and downstream consumption:

- **Ingestion (Kafka):** both producers publish JSON messages into a single Kafka topic (e.g., `tweets_raw`).
- **Stream processing (Spark Structured Streaming):**
  - Parsing JSON messages into a structured schema, and extracting `event_time` from `created_at`.
  - Real-time sentiment labeling using a VADER-based UDF
  - **Complex aggregation:** watermarking for late data, sliding window aggregation.
- **Storage (MongoDB + distributed storage equivalent):**
  - Writing labeled raw tweets to MongoDB for query and dashboard use.
  - Writing aggregated window metrics to MongoDB (time window + region + source).

- Persisting labeled raw data to Parquet with partitioning (by source, region, and label) as a distributed-storage equivalent.
- Separating malformed or incomplete messages into a dedicated "bad records" Parquet sink for audit/debug.

1.2.2 Limitations

This implementation is a proof-of-concept, and it has clear limitations that should not be ignored:

- **API restrictions and coverage:** live ingestion depends on X API access limits and query constraints; it is not a guaranteed firehose.
- **Event-time quality:** the pipeline relies on `created_at` parsing. CSV datasets may contain legacy timestamp formats and time-zone abbreviations that are not consistently parsed across environments, which can reduce event-time correctness.
- **Distributed storage equivalence:** Parquet output is written in a Spark-compatible path . A real HDFS deployment would require cluster configuration and operational controls not covered here.

==========

2 Architecture and Design

2.1 Overall Architecture (Kappa Architecture)

The system is designed following the **Kappa Architecture**, which relies on a single unified streaming pipeline to process all incoming data. In this model, **Kafka serves as the immutable event log**, and **Apache Spark Structured Streaming** performs all transformations, aggregations, and analytics in real time.

Both live Twitter/X data and offline Kaggle CSV data are treated uniformly as streams and ingested into the same Kafka topic. This eliminates the need for a separate batch layer, avoids duplicated business logic, and simplifies maintenance. Historical reprocessing is achieved by replaying Kafka logs rather than executing separate batch jobs.

This architectural choice aligns well with real-time analytics use cases and reflects modern industry practice for event-driven Big Data systems.

2.2 System Components and Their Roles

2.2.1 Data Ingestion Layer

The ingestion layer consists of two independent Kafka producers, each targeting a different data source but producing a common JSON message format.

- **CSV Replay Producer:** Implemented in `producer_csv.py`, this component reads a Kaggle Twitter sentiment dataset line by line and publishes each record as a JSON message to Kafka. A configurable delay is introduced between messages to simulate real-time data arrival. This producer enables deterministic testing, debugging, and high-volume simulation without relying on external APIs.
- **Live Twitter/X Producer:** Implemented in `producer_x.py`, this producer connects to the Twitter/X API v2 using Tweepy. Tweets are retrieved via recent-search queries, normalized into a structured JSON format (tweet ID, text, creation time, language, source, and public metrics), and published to Kafka.

Both producers use the `kafka-python` client library and publish to a single Kafka topic (e.g., `tweets_raw`), ensuring a unified downstream processing path.

2.2.2 Streaming Layer (Apache Kafka)

Apache Kafka acts as the central backbone of the system.

- Kafka provides a durable, append-only commit log that buffers all incoming tweet events.
- Producers and consumers are fully decoupled, allowing ingestion and processing rates to scale independently.
- In the event of downstream failure, Spark can resume processing by replaying messages from Kafka using offsets stored in checkpoints.

Kafka therefore serves both as a messaging system and as a fault-tolerant data source for stream reprocessing.

2.2.3 Processing Layer (Apache Spark Structured Streaming)

The processing layer is implemented using Apache Spark Structured Streaming with a micro-batch execution model.

- Spark consumes streaming data directly from Kafka and deserializes JSON messages into a predefined schema.
- An **event-time column** is extracted from the `created_at` field, enabling time-based operations.
- Sentiment classification is performed using a VADER-based UDF with lazy initialization on executors to reduce overhead and serialization issues.
- A small reference dataset (e.g., language-to-region mapping) is joined using a **broadcast join** to enrich the stream.

In addition to per-record processing, the pipeline performs **stateful stream analytics**:

- Watermarking is applied to handle late-arriving events.
- Sliding window aggregations compute sentiment counts over time.
- Pivot operations transform sentiment labels into analytical columns (positive, negative, neutral).

Two independent streaming queries are executed:

- A raw stream that writes sentiment-labeled tweets to persistent storage.
- An aggregated stream that writes window-based sentiment statistics.

2.2.4 Storage Layer (MongoDB and Parquet)

The storage layer combines a NoSQL database with columnar files for analytical use.

- **MongoDB:**
  - Stores sentiment-labeled raw tweets for dashboard queries.
  - Stores aggregated sentiment metrics (window, region, source).
  - Schema flexibility supports heterogeneous tweet metadata.
- **Parquet (Distributed Storage Equivalent):**
  - Raw labeled tweets are written to Parquet files, partitioned by source, region, and sentiment label.
  - Malformed or invalid records are separated into a dedicated "bad records" sink for audit and debugging.
  - During development, Parquet is written to a Spark-compatible filesystem; the design is directly portable to HDFS or cloud object storage.

All connection parameters (Kafka brokers, MongoDB URI, database and collection names, output paths) are externalized using environment variables to ensure configurability and deployment flexibility.

2.3 Data Flow and Component Interaction

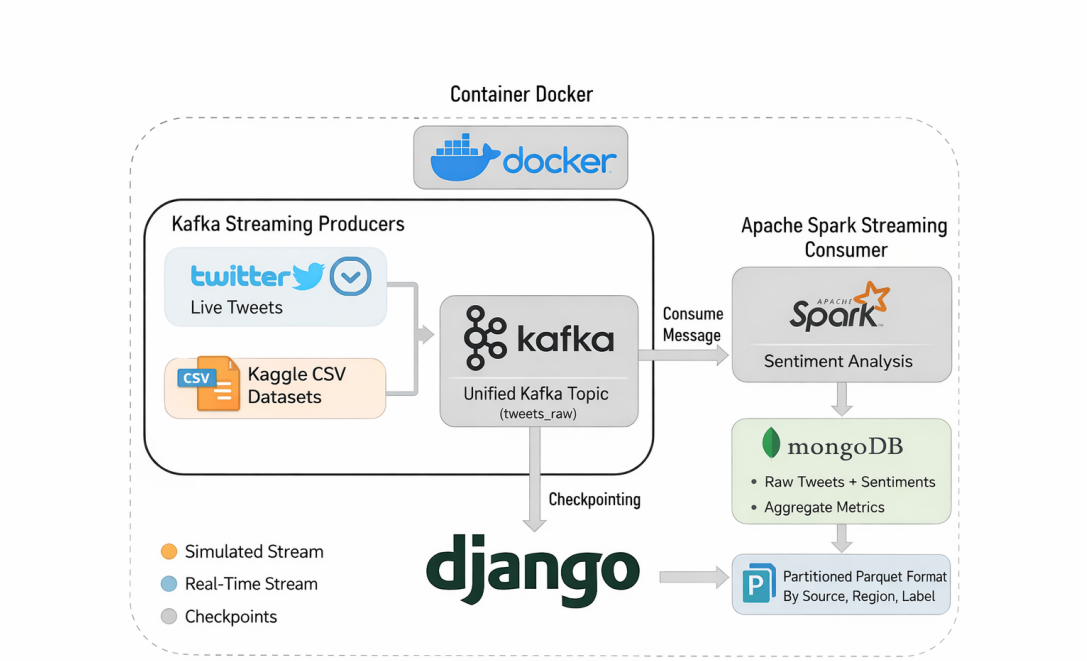

Figure 1: Data flow and component interaction

1. Tweets are collected from either the Twitter/X API or a Kaggle CSV dataset.
2. Producers serialize each tweet into a normalized JSON structure.
3. Messages are published to a Kafka topic acting as the system's event log.
4. Spark Structured Streaming consumes Kafka data in micro-batches.
5. Sentiment analysis, enrichment, and aggregation are applied in real time.
6. Processed results are written to MongoDB and Parquet for persistence and visualization.

3 Implementation Details

3.1 Source Code Organization and Documentation

The project source code is organized by functional responsibility, following a clear separation between data ingestion, stream processing, and configuration.

- **Kafka Producers**

  - `producer_csv.py`: Reads a Kaggle Twitter sentiment CSV dataset and replays it as a stream. Each row is converted into a normalized JSON message and published to Kafka with a configurable delay to simulate real-time arrival. This producer is primarily used for testing, debugging, and high-volume simulation.
  - `producer_x.py`: Connects to the Twitter/X API v2 using Tweepy. Tweets retrieved via recent-search queries are normalized into JSON objects containing text, timestamps, language, source, and public metrics before being sent to Kafka.

- **Spark Streaming Consumer**
  - `spark_streaming_to_mongo_hdfs.py`: Implements the main Apache Spark Structured Streaming pipeline. This script consumes Kafka data, performs sentiment analysis and enrichment, applies windowed aggregations, and writes results to MongoDB and Parquet sinks.

Each script contains inline comments explaining configuration parameters, transformation steps, and design decisions. The modular layout improves maintainability and allows individual components (producers or consumers) to be tested independently.

3.2 Environment-Specific Configuration

All environment-dependent parameters are externalized to avoid hardcoding sensitive or deployment-specific values.

Configuration is loaded from a `.env` file using the `python-dotenv` library and includes:

- Kafka bootstrap server and topic names
- MongoDB URI, database name, and collection names
- Output paths for Parquet and checkpoint directories
- (Optional) Twitter/X API bearer token

This approach improves security, enables quick reconfiguration across environments, and ensures the same codebase can be reused without modification.

3.3 Stream Processing Implementation

The Spark pipeline is implemented using Apache Spark Structured Streaming with a micro-batch execution model.

- Streaming data is read from Kafka and deserialized from JSON into a structured schema.
- An event-time column is derived from the `created_at` field and used as the basis for time-based operations.

- Sentiment analysis is performed using a VADER-based user-defined function (UDF) with lazy initialization on executors to reduce serialization overhead.
- Records are enriched by joining with a small reference dataset (e.g., language-to-region mapping) using a **broadcast join**.

Two independent streaming queries are executed:

- **Raw stream**: writes sentiment-labeled tweets to MongoDB and Parquet, partitioned by source, region, and sentiment label.
- **Aggregated stream**: applies watermarking and sliding windows to compute sentiment counts over time, followed by pivot operations to produce analytical columns.

Malformed or incomplete records are filtered and written to a separate Parquet sink, enabling auditability and easier debugging.

3.4 Deployment Strategy

The system is deployed locally in a step-by-step manner suitable for development and academic evaluation.

1. Start Kafka, Zookeeper, and MongoDB services.
2. Launch one or more Kafka producers (CSV replay or Twitter/X ingestion).
3. Submit the Spark Structured Streaming job using `spark-submit` with Kafka and MongoDB connector packages.
4. Monitor streaming progress via Spark logs and verify persistence in MongoDB and Parquet outputs.

Because the system follows Kappa Architecture, the same streaming job is responsible for both real-time processing and historical reprocessing by replaying Kafka offsets when needed.

3.5 Monitoring and Logging

Observability is provided through built-in logging and runtime metrics.

- Spark log level is set to `WARN` to balance signal and noise.
- Structured Streaming progress information (input rate, batch duration, processed rows) is available through Spark logs and the Spark UI.
- Producers log message publication status to the console.
- MongoDB collections and Parquet outputs can be inspected to validate successful writes.

Although no external monitoring stack (e.g., Prometheus or Grafana) is integrated, the current setup is sufficient to validate correctness, performance behavior, and fault tolerance for a proof-of-concept Big Data streaming system.

4 Lessons Learned

4.1 Lessons on Data Processing with Spark

Lesson 1: Trade-offs Between `foreachBatch` and Native Connectors

Problem Description Two different output strategies were explored for writing streaming results to MongoDB: manual insertion using `foreachBatch` and the MongoDB Spark Connector.

Approaches Tried

- `foreachBatch` with PyMongo for explicit control over writes.
- Native Structured Streaming sinks via the MongoDB Spark Connector.

While `foreachBatch` provided flexibility, it required more boilerplate code and careful handling of idempotency.

Final Solution The MongoDB Spark Connector was adopted in the final pipeline due to better integration with Spark's execution engine and simpler configuration.

Key Takeaways

- Native connectors are easier to reason about at scale.
- `foreachBatch` is powerful but increases operational risk if not carefully designed.
- Simpler pipelines are more robust under failure scenarios.

4.2 Lessons on Stream Processing Semantics

Lesson 2: Importance of Checkpointing and Event-Time Handling

Problem Description Early versions of the Spark streaming job lost progress after restarts, leading to message reprocessing and inconsistent results.

Approaches Tried

- Running streaming queries without checkpointing.
- Enabling checkpoint directories for all stateful queries.

Without checkpoints, Kafka offsets and aggregation state were not preserved.

Final Solution Checkpointing was enabled for all streaming queries, and event-time processing was combined with watermarking to handle late-arriving data.

Key Takeaways

- Checkpointing is mandatory for reliable Structured Streaming jobs.
- Exactly-once semantics depend on correct offset and state management.
- Event-time processing is essential for time-based analytics.

4.3 Lessons on Data Storage Design

Lesson 3: Schema Flexibility vs Consistency in MongoDB

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

Approaches Tried

- Enforcing a strict schema before insertion.
- Allowing flexible document structures with optional fields.

A strictly enforced schema resulted in unnecessary data loss and complex preprocessing logic.

Final Solution MongoDB's schema-less design was leveraged, while enforcing a minimal required contract (e.g., text, timestamp, sentiment, source).

Key Takeaways

- Schema flexibility is well-suited for semi-structured streaming data.
- A minimal contract schema is still required for reliable querying.
- Not all consistency should be enforced at ingestion time.

4.4 Lessons on Fault Tolerance

Lesson 4: Kafka as a Failure Isolation Buffer

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

Approaches Tried

- Direct ingestion into Spark without buffering.
- Kafka-mediated ingestion with retained messages.

Final Solution Kafka was used as a durable buffer, allowing Spark consumers to restart and replay data safely.

Key Takeaways

- Kafka decouples ingestion from processing lifecycles.
- Durable commit logs are essential for fault-tolerant streaming systems.
- Failure handling must be designed, not assumed.