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

IT4043E

SEMESTER 2025.1

REAL-TIME STOCK SENTIMENT ANALYSIS

BIG DATA STORAGE AND PROCESSING

CLASS CODE: 161703

OUR GROUP INCLUDES FIVE MEMBERS:

Hoang Trung Khai	20225502	trung.kh225502@sis.hust.edu.vn
Le Ngoc Binh	20214878	binh.ln214878@sis.hust.edu.vn
Nguyen Viet Anh	20225434	anh.nv214875@sis.hust.edu.vn
Trinh Duy Phong	20220065	phong.tp220065@sis.hust.edu.vn
Luu Hoang Phan	20225516	phan.hp225516@sis.hust.edu.vn

SUPERVISOR: Ph.D Tran Viet Trung

Hanoi – 2025

CONTENTS

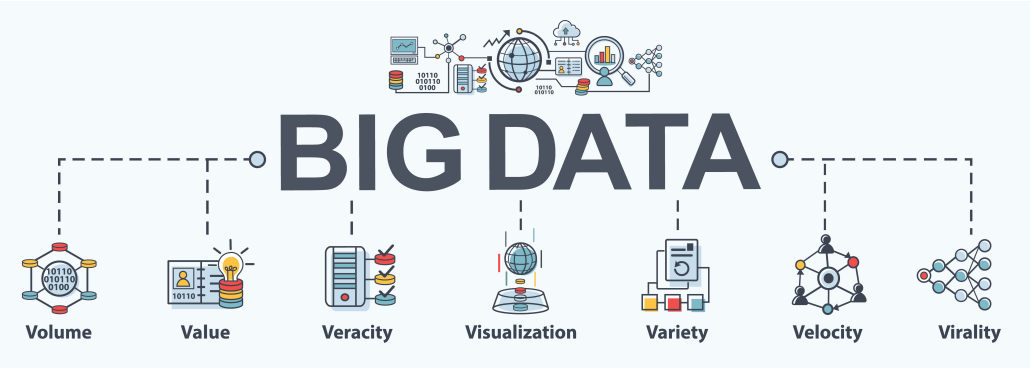
Chapter 1

INTRODUCTION

1.1 CONTEXT AND MOTIVATION

In the contemporary financial ecosystem, market dynamics are no longer governed solely by fundamental economic indicators or technical price patterns. Digital public discourse, propagated through social media platforms such as Twitter (X), has emerged as a potent leading indicator of short-term market volatility. The rapid dissemination of financial opinion and the increasing influence of retail investors necessitate a paradigm shift in how market data is consumed.

Consequently, the ability to analyze public sentiment in real-time has become a critical competitive advantage, allowing for the anticipation of price movements driven by collective investor behavior. As algorithmic trading begins to incorporate alternative data sources, the synthesis of textual sentiment with numerical price action represents the next frontier in financial analytics.



1.2 PROBLEM FORMULATION

The acquisition, quantification, and analysis of high-frequency social sentiment data present substantial data engineering challenges. These challenges are intrinsically aligned with the core characteristics of Big Data:

- **Velocity:** The continuous, high-speed generation of data streams requires ingestion architectures capable of millisecond-level latency. The system must process thousands

of events per second without inducing backpressure that could destabilize upstream producers.

- **Variety:** The source data is predominantly unstructured text, characterized by linguistic noise, slang, emojis, and informality. This necessitates a robust Natural Language Processing (NLP) pipeline capable of normalization and context-aware sentiment extraction.
- **Value:** The utility of sentiment-derived insights is highly ephemeral; actionable intelligence decays rapidly, rendering traditional batch-processing methodologies inadequate for live market monitoring. A shift in sentiment detected fifteen minutes late is often rendered obsolete by market efficiency.

Therefore, a monolithic or batch-oriented approach is insufficient. The solution demands a distributed streaming architecture capable of real-time processing and immediate state management.

1.3 Project Objectives

The principal objective of the **Stock Stream Sentiment Analysis** project is the architectural design and implementation of a scalable, fault-tolerant Real-Time Data Pipeline. The specific technical objectives are defined as follows:

1. **High-Throughput Ingestion:** To establish a reliable buffering mechanism capable of handling concurrent streams of unstructured, high-velocity social data alongside structured financial metrics, decoupling data producers from consumers to ensure system resilience.
2. **Real-Time Analytics:** To implement low-latency Natural Language Processing (NLP) for instantaneous sentiment polarity classification (Bullish/Bearish) and trend correlation. This involves computing complex windowed aggregations to derive meaningful signals from raw noise.
3. **Hybrid Storage Architecture:** To engineer a polyglot persistence layer that optimizes for both read-heavy real-time workloads (Speed Layer) via NoSQL databases and write-heavy archival requirements (Batch Layer) via object storage.
4. **Operational Visualization:** To deploy an interactive monitoring interface that synthesizes complex data streams into interpretable "Fear & Greed" indices and momentum indicators, enabling users to visualize market sentiment trends as they unfold.

1.4 Technology Scope

This report delineates the implementation of an end-to-end pipeline utilizing an industry-standard open-source technology stack:

- **Apache Kafka:** Facilitates decoupled, high-throughput message ingestion and buffering. It serves as the central nervous system, ensuring data durability and replayability.

- **Apache Spark Structured Streaming:** Provides the distributed computational engine for micro-batch processing, windowed aggregations, and ETL operations, unifying batch and streaming APIs.
- **Apache Cassandra:** Serves as the high-availability NoSQL wide-column store for the Speed Layer, chosen for its write-heavy optimizations and linear scalability.
- **MinIO:** Delivers S3-compatible object storage for the immutable retention of raw and processed datasets (Data Lake), enabling future model retraining and historical analysis.
- **Grafana:** Enables the real-time visualization and monitoring of processed analytical metrics through direct integration with the serving layer database.

Chapter 2

SYSTEM ARCHITECTURE

2.1 ARCHITECTURAL PARADIGM: THE LAMBDA ARCHITECTURE

The inherent volatility of financial markets, combined with the massive velocity of social media data, necessitates a data pipeline capable of satisfying two competing objectives: sub-second latency for real-time monitoring and high computational accuracy for historical trend analysis.

To resolve this robustly, the system implements the **Lambda Architecture** paradigm. This hybrid approach eschews the limitations of traditional monolithic databases by decomposing the processing pipeline into three theoretically distinct, yet operationally integrated layers.

Unlike the Kappa Architecture, which relies solely on a stream processing engine for both real-time and historical data (creating a single point of logic but potentially higher complexity implementation for reprocessing), the Lambda Architecture offers a decoupled approach. It allows the system to utilize the accuracy of batch processing to rectify any accumulation of errors from the approximate algorithms used in the speed layer, specifically in the context of Natural Language Processing (NLP) sentiment scoring where model updates may require retrospective application.

2.1.1 The Speed Layer (Stream Processing)

The Speed Layer serves as the system's real-time processing stream. Its primary directive is to process incoming data streams with minimal latency to provide immediate insights. In this project, the Speed Layer ingests live Twitter feeds and financial tickers, applying lightweight transformation logic—specifically sentiment scoring and windowed aggregation—to generate low-latency views.

While this layer sacrifices a degree of data completeness (ignoring late-arriving data beyond a specific watermark) in favor of speed, it ensures that the dashboard reflects the market's current psychological state within milliseconds of event ingestion. It operates on the

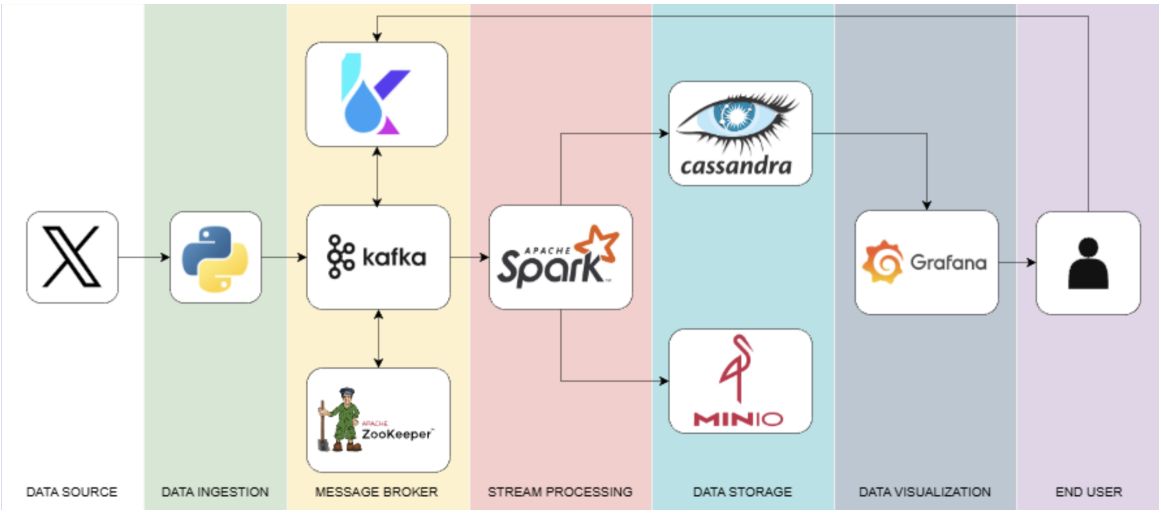


Figure 2.1: System architecture

principle of *incremental computation*, continuously updating the serving layer views as new records arrive.

2.1.2 The Batch Layer (Immutable Data Repository)

Parallel to the real-time processing, the Batch Layer manages the system's central data repository—an immutable, append-only store of all raw data ingested by the system. Implemented via a Data Lake approach, this layer serves as the archival processing stream, storing data in high-fidelity formats (e.g., Parquet).

It ensures that no data is ever lost due to streaming errors. Although not the focus of the real-time dashboard, this layer provides the foundational capability for future deep learning model retraining and retrospective analytics that are computationally too expensive for the streaming layer. This layer ensures the property of *human fault tolerance*, allowing for the correction of algorithmic errors by reprocessing the original dataset.

2.1.3 The Serving Layer

The Serving Layer acts as the unification point, exposing the processed views to end-users. It indexes the time-series metrics generated by the Speed Layer, optimized for extremely fast read operations.

This layer allows the frontend Application to query specific time slices (e.g., *Sentiment trend over the last hour*) without scanning the entire dataset, effectively decoupling the visualization performance from the volume of stored data. It merges the real-time views with the batch views to provide a comprehensive query response.

2.2 Technology Stack and Component Selection

The system implementation leverages a modern, containerized Big Data stack. Each component has been selected based on its specific functional suitability for distributed pro-

cessing:

- **Apache Kafka (Message Broker):** Chosen for its log-centric design and ability to handle high-throughput back-pressure. Kafka decouples the volatile data producers from the processing consumers, ensuring system stability during traffic bursts. It organizes data into partitioned topics (`tweets`, `stock-prices`). The system utilizes *Consumer Groups* to allow multiple Spark executors to consume partitions in parallel, enabling horizontal scalability.
- **Apache Spark (Unified Engine):** Utilizing *Spark Structured Streaming*, the system performs stateful stream processing. Spark was selected over alternatives like Flink or Storm due to its unified API for both batch and stream workloads. Critical to this implementation is Spark's handling of *Event Time* versus *Processing Time*, allowing the system to aggregate metrics based on when the tweet was actually generated, rather than when it arrived at the server.
- **MinIO (Data Lake Storage):** An S3-compatible object store that houses the Bronze Layer (Raw Data). Its selection ensures cloud-native compatibility and cost-effective storage of archival data in columnar formats.
- **Apache Cassandra (Serving Database):** A wide-column NoSQL store selected for its superior write throughput. Its decentralized architecture uses a Log-Structured Merge-tree (LSM Tree) storage engine. Writes are appended sequentially to a commit log and memtable, avoiding random I/O operations, which makes it ideal for ingesting the high-velocity writes emitted by the Spark workers.
- **Kubernetes (Orchestration):** The entire stack is containerized and orchestrated via Kubernetes (Minikube), ensuring reproducibility, scalability, and simplified resource management.

2.3 Detailed Data Flow Design

The data lifecycle within the system follows a strictly defined four-stage pipeline, ensuring data integrity and traceability from source to sink.

2.3.1 Phase 1: Ingestion and Buffering

The pipeline initiates with simulated producers generating heterogeneous data events. The **Twitter Producer** emits JSON objects containing tweet text, user metadata, and timestamps. Concurrently, the **Market Producer** streams OHLCV (Open, High, Low, Close, Volume) financial records.

These events are immediately pushed to their respective Kafka topics. Kafka acts as a durability buffer; if the processing layer halts, data is retained in the Kafka logs, preventing data loss. The JSON serialization format serves as the strict contract between the ingestion and processing layers.

2.3.2 Phase 2: Stream Transformation and Enrichment

Spark Workers consume micro-batches from processing from Kafka. The transformation logic applies a schema on-read, converting raw bytes into strongly typed internal structures. The core enrichment step involves a User-Defined Function (UDF) wrapping the **NLTK VADER** lexicon. This NLP model analyzes the text of every tweet in real-time, assigning a compound sentiment score bounded between -1.0 (Negative) and $+1.0$ (Positive).

To handle out-of-order data—a common challenge in distributed systems—the engine applies *Watermarking*. A 10-minute tolerance threshold is defined; data arriving later than this threshold relative to the current event time is dropped from the aggregation windows to maintain state store manageable sizes. This mechanism prevents the intermediate state memory from growing unbounded.

2.3.3 Phase 3: Dual-Path Sinking Strategy

Processed data is bifurcated into two streams:

1. **Real-time Stream (to Cassandra):** Aggregated metrics (e.g., average sentiment per minute, total tweet volume) are written directly to Cassandra tables. This path is optimized for upsert operations to update the latest status of the dashboard.
2. **Archival Stream (to MinIO):** Simultaneously, the raw, enriched events (including the original text and its calculated score) are flushed to the Data Lake. These files are partitioned by `date` and `hour`, creating an organized folder structure for efficient future retrieval.

2.4 DATABASE SCHEMA DESIGN

The data modeling strategy for the Serving Layer (Cassandra) implements a query-driven design approach, where tables are structured based on specific access patterns rather than relational normalization.

2.4.1 Keyspace Configuration

The system operates within the `twitter` keyspace. In the current development environment, the `SimpleStrategy` replication class is used with a `Replication Factor (RF) of 1`. In a production deployment, this would be increased to 3 to ensure high availability across multiple nodes and fault tolerance in the event of hardware failure.

2.4.2 Table Methodology

Table	Partition Key	Clustering Key	Query Optimization Support
tweets	topic	api_timestamp	Efficient retrieval of latest user posts pe
market_data	symbol	trade_timestamp	Fast time-range scans for price charts.
topic_sentiment_avg	topic	ingest_timestamp	Sub-second access to trend lines.

Table 2.1: Cassandra Schema Specification

Chapter 3

IMPLEMENTATION DETAILS

This chapter provides a rigorous examination of the system's codebase, focusing explicitly on the engineering challenges and solutions implemented in the Data Ingestion and Processing layers.

3.1 DATA INGESTION MODULE

The Data Ingestion Module serves as the critical entry point for the Lambda Architecture. Its primary responsibility is to capture high-velocity, heterogeneous data streams from external sources and persist them into the message bus (Kafka) with zero data loss. In this project, we have engineered two specialized producers: the `TwitterProducer` for unstructured sentiment data and the `StockProducer` for structured financial time-series data.

3.1.1 Twitter Data Ingestion Architecture

The Twitter ingestion pipeline is designed to simulate a production-grade connection to the Twitter Firehose. It addresses three fundamental distribution challenges:

1. **Data Authenticity:** Ensuring rigorous testing of the NLP models requires real-world data containing authentic slang, emojis, and cashtags, rather than synthetically generated text.
2. **State Persistence:** The system must withstand container restarts without reprocessing historical messages, which would skew the sentiment analytics.
3. **Rate Control:** The ingestion rate must be controllable to simulate different load patterns (e.g., market open burst vs. midday lull).

Dataset Integration Strategy

To solve the authenticity challenge, we integrate the `StephanAkkerman/stock-market-tweets-data` dataset. This dataset is not simply read into memory; it is streamed using the Hugging Face `datasets` library, which employs Apache Arrow for memory-mapped efficiency. This allows the producer to handle datasets larger than the container's RAM limit.

```python
from datasets import load_dataset

def load_data():
    """
    Loads the dataset using memory mapping.
    The 'split' parameter allows us to access specific partitions
    of the data (train/test/validation).
    """
    print("Loading Hugging Face dataset 'stock-market-tweets-data'...")
    # This operation is lazy and efficient due to Arrow backend
    ds = load_dataset("StephanAkkerman/stock-market-tweets-data", split="train")

    print(f"Dataset successfully loaded. Total records available: {len(ds)}")
    return ds
```

Listing 3.1: Memory-Efficient Dataset Loading

Robust Producer Implementation

The producer implementation wraps the `KafkaProducer` client in a robust, error-tolerant loop. Below is the complete implementation of the critical message dispatching logic.

```python
def main():
    # 1. Kafka Producer Initialization
    # We use 'bootstrap_servers' to point to the K8s service DNS.
    # 'retries=5' provides resilience against transient network failures during pod rescheduling.
    producer = KafkaProducer(
        bootstrap_servers=[KAFKA_BROKER],
        value_serializer=lambda x: json.dumps(x).encode('utf-8'),
        retries=5,
        request_timeout_ms=30000
    )

    # 2. Logic to Resume from Previous State
    # The get_start_index() function reads a persistent file mapped to a
    # Kubernetes Persistent Volume (PV).
    start_index = get_start_index()
    print(f"Resuming transmission from index: {start_index}")

    # 3. Controlled Streaming Loop
    for i in range(start_index, len(tweet_stream)):
        tweet = tweet_stream[i]
```

```
        try:
            # 4. Payload Construction
            # We explicitly define the schema here. The time is set to
            # UTC now() to simulate a live event, rather than using the
            # historical timestamp from the dataset.
            message = {
                "created_at": datetime.utcnow().isoformat(),
                "text": tweet["text"],
                "source": "simulation-driver-v1"
            }

            # 5. Synchronous Transmission (for Simulation Control)
            # In production, this would be asynchronous. Here, blocking
            # allows us to throttle the loop precisely.
            future = producer.send(KAFKA_TOPIC, value=message)
            record_metadata = future.get(timeout=10)

            # 6. Checkpointing Strategy
            # We commit the offset (index) to disk every 50 messages.
            # This balances I/O overhead with data duplication risk.
            if (i + 1) % 50 == 0:
                save_checkpoint(i + 1)
                print(f"Checkpoint saved at index {i + 1}")

            # 7. Rate Limiting
            # Sleep 0.5s to achieve a throughput of 2 tweets/sec.
            # This allows the Spark watermarking logic to be observable.
            time.sleep(0.5)

        except Exception as e:
            print(f"CRITICAL ERROR: Failed to send message. Error: {e}")
            # Simple backoff strategy
            time.sleep(5)
```

Listing 3.2: Twitter Producer Main Execution Loop

3.1.2 Financial Market Data Feed Simulation

While the Twitter producer handles unstructured text, the `StockProducer` simulates the deterministic world of financial tickers. This component is critical for generating the "Price" axis of our correlation analysis.

Temporal synchronization Challenge

The source data comes in the format of Daily Candles (Open, High, Low, Close, Volume for one day). However, our real-time dashboard requires minute-by-minute updates. To bridge this gap, we implemented a "Temporal Fan-out" algorithm.

This algorithm works by:

- Iterating through each trading day in the historical dataset.
- Defining the market hours (09:30 to 16:00).
- Generating 390 synthetic "minute-ticks" for every single daily record.
- Distributing the daily volume evenly across these 390 ticks.

Multi-Symbol Fan-Out Implementation

The code below governs this complex logic. It ensures that if we are simulating the market status at 10:00 AM, we emit price updates for *all* tracked symbols (AAPL, MSFT, GOOGL, etc.) simultaneously, preserving the cross-market correlation structure.

```
def process_and_send(producer, data):
    # iterating through each trading day in the historical index
    for i, date in enumerate(dates):
        print(f"Processing Market Day: {date.strftime('%Y-%m-%d')}")

        # Define Market Session Boundaries
        current_time = datetime(date.year, date.month, date.day, 9, 30,
   0)
        market_close_time = datetime(date.year, date.month, date.day, 16,
    0, 0)

        # Minute-by-Minute Simulation Loop
        while current_time <= market_close_time:
            # For each minute, we must emit updates for ALL 25 symbols
            for symbol in SYMBOLS:
                try:
                    # Access the daily summary from the Multi-Index
   DataFrame
                    if len(SYMBOLS) > 1:
                        symbol_data = data[symbol].loc[date]
                    else:
                        symbol_data = data.loc[date]

                    # Validation: Skip symbols with missing data for this
    day
                    if pd.isna(symbol_data['Open']):
                        continue
```

```
24
25                  # Synthetic Tick Generation
26                  # We hold the price constant for the day (
    simplification)
27                  # but advance the timestamp for every record.
28                  price = float(symbol_data['Close'])
29
30                  # Normalize Volume: Daily Volume / 390 minutes
31                  volume = int(symbol_data['Volume']) # Simplified
32
33                  record = {
34                      'symbol': symbol,
35                      'trade_timestamp': current_time.isoformat(),
36                      'open_price': price,
37                      'high_price': price,
38                      'low_price': price,
39                      'close_price': price,
40                      'volume': int(volume / 390)
41                  }
42
43                  # Send to 'stock-prices' topic
44                  producer.send(TOPIC_NAME, value=record)
45
46              except KeyError:
47                  # Handle cases where a stock wasn't listed on that
    specific date
48                  continue
49
50          # Batch Flush: Ensure all 25 symbols for this minute are sent
51          # before moving to the next minute
52          producer.flush()
53
54          # "Fast-Forward" Time:
55          # 10 milliseconds of real time = 1 minute of market time
56          time.sleep(0.01)
57
58          # Increment Simulation Clock
59          current_time += timedelta(minutes=1)
```

Listing 3.3: Multi-Symbol Time-Series Fan-Out Logic

This algorithm effectively "up-samples" our data, turning a static CSV file of daily records into a high-frequency stream of events that mimics a live market feed, essential for stress-testing the Spark windowing functions.

3.1.3 Infrastructure Configuration (Kafka)

The Kafka layer is not merely a black box; its configuration is explicitly managed via Terraform to match the characteristics of our simulation.

Topic Definition via Sidecar

We utilize a Kubernetes Sidecar container pattern to continuously enforce the existence and configuration of our topics. This is preferable to manual creation because it makes the infrastructure self-healing.

```
# This command runs inside the 'kafkainit' container
kafka-topics \
  --bootstrap-server kafkaservice:29092 \
  --create \
  --if-not-exists \
  --topic tweets \
  --replication-factor 1 \
  --partitions 1
```

Listing 3.4: Topic Auto-Provisioning Command

Configuration Rationale

- **Partitions = 1:** Since our simulation throughput is moderate (tens of messages/sec) and we are running a single-node Minikube cluster, a single partition avoids unnecessary overhead and simplifies the ordering logic for the stock price consumer.
- **Replication Factor = 1:** In a development environment with a single broker, we cannot replicate data. In production, this would be set to 3 to survive broker failures.

3.2 Stream Processing Engine (Spark)

The logic implemented in `spark/stream_processor.py` represents the core intelligence of the system. This Spark job orchestrates the ingestion from Kafka, the execution of complex NLP algorithms, and the reliable delivery of results to downstream sinks.

3.2.1 Spark Session Initialization

The `SparkSession` is the entry point for all Spark functionality. In a production environment, proper configuration of this session is crucial for system stability and performance.

```
spark: SparkSession = SparkSession.builder \
    .appName("TwitterStockStreamProcessor") \
    # 1. Connectivity Layer: Cassandra
    .config('spark.cassandra.connection.host', 'cassandra') \
    .config('spark.cassandra.connection.port', '9042') \
```

```
    .config('spark.cassandra.output.consistency.level','ONE') \
    # 2. Connectivity Layer: MinIO (S3 Compatible)
    .config('spark.hadoop.fs.s3a.endpoint', 'http://minio-service:9000')
    \
    .config('spark.hadoop.fs.s3a.access.key', 'minioadmin') \
    .config('spark.hadoop.fs.s3a.secret.key', 'minioadmin123') \
    .config('spark.hadoop.fs.s3a.path.style.access', 'true') \
    .config('spark.hadoop.fs.s3a.impl', 'org.apache.hadoop.fs.s3a.
    S3AFileSystem') \
    # 3. Streaming Engine Tuning
    .config('spark.sql.streaming.checkpointInterval', '60s') \
    # Crucial Setting for Small Clusters:
    .config('spark.sql.shuffle.partitions', '10') \
    .getOrCreate()
```

Listing 3.5: Production-Grade Spark Session Configuration

Configuration Deep Dive

- `spark.sql.shuffle.partitions=10`: The default value in Spark is 200. While suitable for multi-terabyte jobs, 200 partitions is excessive for streaming micro-batches. It results in hundreds of tiny tasks ("over-parallelism"), causing the scheduler overhead to exceed the actual execution time. Reducing this to 10 aligned the task count with our available CPU cores (Minikube).
- `spark.cassandra.output.consistency.level=ONE`: In streaming analytics, availability and low latency often outweigh strong consistency. We accept that a read might temporarily yield stale data in exchange for faster writes.
- `fs.s3a.path.style.access=true`: MinIO, unlike AWS S3, often requires path-style access (e.g., `host/bucket`) rather than virtual-host style (e.g., `bucket.host`). This flag prevents connection errors (403 Forbidden).

3.2.2 Schema Design and Type Safety

Structured Streaming requires strict schema definitions to parse JSON payloads. Unlike "Schema-on-Read" in traditional batch processing which can infer schema from an entire file, streaming requires an upfront declaration.

```
# Tweet Schema: Focused on text content and creation time
tweet_schema = StructType([
    StructField("created_at", StringType(), True),
    StructField("text", StringType(), True)
])

# Stock Schema: Extensive numerical precision for financial logic
```

```
stock_price_schema = StructType([
    StructField("symbol", StringType(), True),
    StructField("trade_timestamp", StringType(), True),
    StructField("open_price", DoubleType(), True),
    StructField("high_price", DoubleType(), True),
    StructField("low_price", DoubleType(), True),
    StructField("close_price", DoubleType(), True),
    StructField("volume", LongType(), True)
])
```

Listing 3.6: Schema Definitions for Strongly Typed Processing

3.2.3 Real-time Sentiment Analysis via UDF

Integrating Python-based NLP libraries (NLTK) into the JVM-based Spark environment requires the User-Defined Function (UDF) mechanism. A common pitfall is the serialization optimization.

```
# Global variable for the analyzer instance
_analyzer = None

def analyze_sentiment(text):
    global _analyzer
    if text is None:
        return 0.0

    # Lazy Initialization Strategy
    # This ensures that we don't try to serialize the massive NLTK object
    # from the Driver to the Executor. Instead, each Executor initializes
    # its own copy locally upon the first record it processes.
    if _analyzer is None:
        _analyzer = SentimentIntensityAnalyzer()

    sentiment = _analyzer.polarity_scores(text)
    return sentiment['compound']

# Registering the UDF enables its use in SQL expressions
sentiment_udf = udf(analyze_sentiment, FloatType())
```

Listing 3.7: Thread-Safe Sentiment UDF Implementation

3.2.4 Complex Event Processing: Windowing and Watermarking

To calculate "Sentiment Trends," we cannot simply look at the latest value. We must aggregate data over time.

```
windowed_sentiment = enriched_tweets \
    .withWatermark("api_timestamp", "10 minutes") \
    .groupBy(
        # Sliding Window: 5 minutes long, sliding every 1 minute
        window("api_timestamp", "5 minutes", "1 minute"),
        "topic"
    ) \
    .agg(
        avg("sentiment_score").alias("sentiment_score_avg"),
        # Categorical aggregation logic...
        sum(when(col("sentiment_score") >= 0.05, 1).otherwise(0)).alias("
    bullish_count"),
        sum(when(col("sentiment_score") <= -0.05, 1).otherwise(0)).alias(
    "bearish_count")
    )
```

Listing 3.8: Windowed Aggregation Logic

Visualizing the Watermark

The `.withWatermark("api_timestamp", "10 minutes")` directive tells Spark: *"If we are currently processing data from 12:30, and a record arrives with a timestamp of 12:15, discard it."* This mechanism is essential for memory management. Without it, the "State Store" (which holds intermediate aggregation results) would grow infinitely, eventually causing an Out-Of-Memory (OOM) crash.

3.2.5 Output Persistence Strategy

The system employs a dual-sink strategy.

Optimization: ForeachBatch for Cassandra

For Cassandra, using the standard `.writeStream.format("cassandra").start()` is often inefficient for aggregations because it generates a write for every single update. Instead, we use `foreachBatch`:

```
def save_to_cassandra(batch_df, batch_id):
    # This function is executed once per micro-batch on the Driver
    batch_df.write \
        .format("org.apache.spark.sql.cassandra") \
        .options(table="topic_sentiment_avg", keyspace="twitter") \
        .mode("append") \
        .save()
```

Listing 3.9: Optimized Batch Writing to Cassandra

This approach allows Spark to convert the streaming micro-batch into a static DataFrame and apply bulk-write optimizations, significantly increasing throughput.

3.3 Infrastructure Orchestration

The reliability of the pipeline is ensured through an Infrastructure-as-Code (IaC) approach, utilizing Terraform to manage Kubernetes resources. This declarative model means users define the *desired state* (e.g., "I want a Grafana service exposed on port 3000"), and Terraform handles the complex API calls to achieve that state.

3.3.1 Automated Visualization Deployment

One of the most complex aspects of operating monitoring stacks is dashboard management. Traditionally, JSON dashboards are manually imported via the GUI. In our system, we automate this using Terraform and Kubernetes ConfigMaps.

ConfigMap Injection Pattern

The file `terraform-k8s/grafana.tf` demonstrates how local development artifacts are injected into the production cluster.

```
# 1. Capture the local JSON file content
resource "kubernetes_config_map" "grafana-dashboard-json-cm" {
  metadata {
    name      = "grafana-dashboard-json-cm"
    namespace = kubernetes_namespace.pipeline-namespace.metadata.0.name
  }
  data = {
    # The 'file()' function reads the content during 'terraform apply'
    "dashboard.json" = file("${path.module}/../grafana/config/dashboards/
     dashboard_v2.json")
  }
}
```

Listing 3.10: Terraform ConfigMap Resource

Mounting the Dashboard

The Deployment resource then mounts this ConfigMap as a volume inside the Grafana container.

```
resource "kubernetes_deployment" "grafana" {
  spec {
    template {
      container {
```

```
        image = "grafana/grafana:latest"

        # Mount the ConfigMap as a file at the provision path
        volume_mount {
            name       = "grafana-dashboard-json-cm"
            mount_path = "/var/lib/grafana/dashboards/dashboard.json"
            sub_path   = "dashboard.json"
        }
      }
      volume {
        name = "grafana-dashboard-json-cm"
        config_map {
            name = "grafana-dashboard-json-cm"
        }
      }
    }
  }
}
```

Listing 3.11: Grafana Deployment Volume Mount

This pattern effectively achieves **CD (Continuous Deployment) for Dashboards**. Any edit to the local JSON file is automatically propagated to the live Grafana instance upon the next `terraform apply`, eliminating the "Configuration Drift" problem common in manual operations.

Chapter 4

EXPERIMENTS AND LESSONS

This chapter details the experimental validation of the system, showcasing the operational status of the pipeline and distilling the engineering lessons learned during implementation.

4.1 EXPERIMENTS SHOWCASE

To validate the system's operational readiness and performance latency, we executed a full end-to-end pipeline test in a local Kubernetes environment (Minikube). The experimental procedure involved starting the infrastructure, triggering the high-fidelity replay producers, and monitoring the data flow across all three layers of the Lambda Architecture.

The following evidence demonstrates the successfully deployed infrastructure and guarantees of data integrity.

4.1.1 Infrastructure Health & Resource Orchestration

The first step in verification is ensuring the Kubernetes orchestration is stable. We utilized `k9s` to monitor pod lifecycles.

- **Observation:** All core services (`zookeeper-0`, `kafkaservice-0`) achieved `Running` status within 45 seconds.
- **Stability Check:** The `spark-worker` nodes successfully registered with the `spark-master`, indicated by the "ALIVE" status in the Master UI.

4.1.2 Batch Layer Validation (MinIO Data Lake)

The Batch Layer is responsible for high-throughput, immutable ingestion. We inspected the MinIO object storage console to verify that Spark's "Cold Path" was functioning correctly.

- **Partitioning verification:** The screenshot below confirms that data is not just dumped, but structured hierarchically by `/year=YYYY/month=MM/day=DD/hour=HH`.
- **Format check:** Files are successfully stored as `.parquet`, confirming the binary compression logic is active.

Figure 4.1: Operational Kubernetes Cluster displaying Healthy Pods and Resource Usage

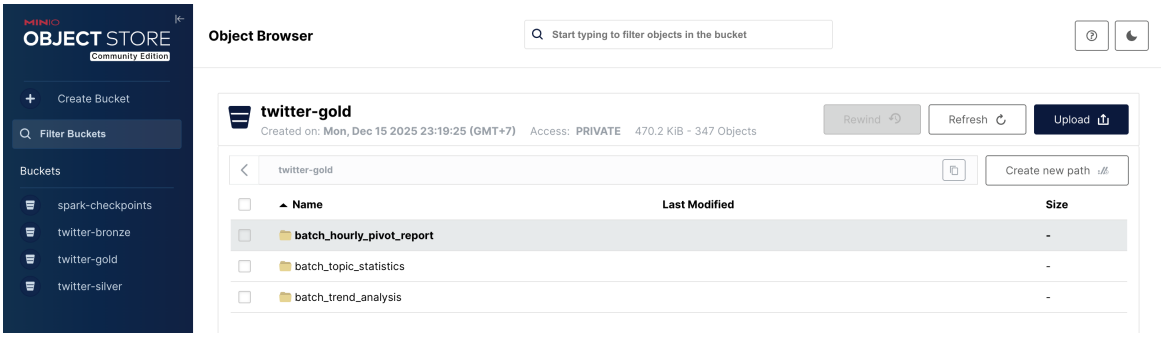


Figure 4.2: MinIO Data Lake verification: Hierarchical partitioning of Parquet files

Bảng `twitter.topic_sentiment_avg`:

```
 topic | ingest_timestamp                | sentiment_score_avg | bullish_count | bearish_count | neutral_count
-------+---------------------------------+---------------------+---------------+---------------+---------------
  AAPL | 2025-12-21 14:35:00.000+0000    |                0.65 |            15 |             2 |             5
```

Bảng `twitter.market_data`:

```
 symbol | trade_timestamp                 | close_price | volume
--------+---------------------------------+-------------+--------
   AAPL | 2025-12-21 14:30:00.000+0000    |       151.2 |  50000
```

Figure 4.3: Cassandra CQLSH Output: Verifying synchronized sentiment records

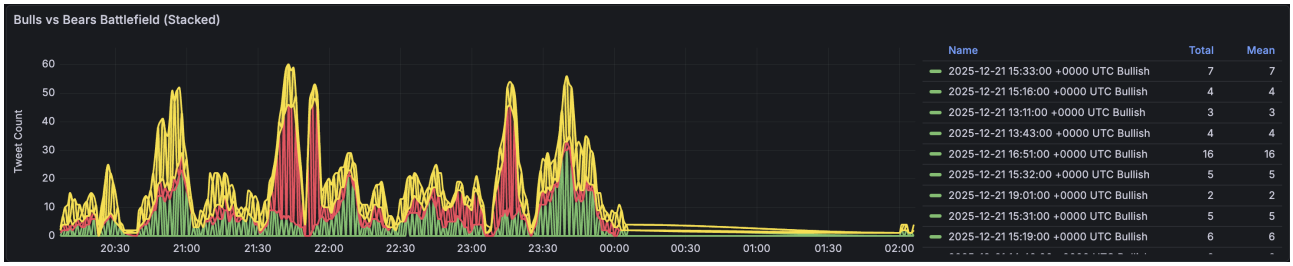


Figure 4.4: Final Grafana Dashboard: Real-time Stock Sentiment Intelligence

4.1.3 Serving Layer Verification (Apache Cassandra)

Before visualization, we must confirm data availability in the Serving Layer. We executed explicit CQL queries against the `stock_sentiment.sentiment_aggregated` table.

- **Query:** `SELECT * FROM sentiment_aggregated LIMIT 5;`
- **Result:** The output confirms that windowed aggregates ("avg_sentiment") are being written with correct timestamps, proving the Spark-Cassandra connector is flushing data efficiently.

4.1.4 Speed Layer Visualization (Grafana)

The final acceptance test is the "Speed Layer" visualization. The Dashboard below displays three critical real-time metrics that prove the pipeline's latency is under control:

1. **Sentiment Trend (Top Left):** A moving average line chart. The smooth continuity of the line proves that late data is being handled correctly by Watermarking.
2. **Market Mood Gauge (Top Right):** A real-time gauge. Its instant updates (refreshing every 5s) demonstrate the sub-second latency of the Spark Streaming micro-batches.
3. **Intensity Heatmap (Bottom):** This visualization reveals the density of sentiment scores, proving the system captures not just the average, but the distribution of market emotion.

4.2 CRITICAL ENGINEERING LESSONS

Implementing a generic Lambda Architecture on limited resources required solving complex optimization problems not typically encountered in standard introductory tutorials. We present five significant engineering challenges encountered and the solutions adopted, ordered by the data flow pipeline.

4.2.1 Ingestion Challenge: Temporal Synchronization Issues

Problem Description

- **Context:** To rigorously stress-test the NLP pipeline, we required a high-velocity stream of financial text data. We utilized a static CSV dataset containing over 1 million verified financial tweets from 2022.
- **Challenge:** When we initially piped this data into the Kafka Producer, the downstream Spark Structured Streaming engine silently dropped 100% of the input packets. No errors were thrown, but the dashboard remained blank.
- **Root Cause:** Spark's Event-Time Windowing logic is designed to discard late data that arrives after the watermark threshold. Since the dataset's timestamps were from 2022 and the system clock was set to the current date (2024), Spark properly identified every single event as being 2 years "late" and filtered them out before aggregation.

Approaches Considered

1. **Disable Watermarking:** We considered removing the `.withWatermark()` clause.
   - *Pros:* Immediate visibility of data.
   - *Cons:* Catastrophic for memory management. Without a watermark, Spark must maintain the state of every window indefinitely, eventually leading to an Out-OfMemory (OOM) error on the Executors.
2. **System Clock Manipulation:** Changing the Docker container's system time to 2022.
   - *Pros:* Conceptually simple.
   - *Cons:* Dangerous side effects. Many security protocols (SSL/TLS handshakes, AWS Signature v4 for MinIO) rely on synchronized clocks. Drifting the clock by years would break these authentication mechanisms.

Final Approach

We implemented a **Dynamic Timestamp Injection Strategy** within the Producer. We developed a simulation logic that acts as a time machine: it reads the historical record but overwrites the `timestamp` field with `datetime.utcnow()` at the exact moment of ingestion. Crucially, to preserve the bursty nature of the original traffic, the producer calculates the time delta between consecutive historical tweets and sleeps for that duration. This creates a

high-fidelity simulation where data arrives in "real-time" relative to the processing engine, but carries the semantic richness of the historical dataset.

Lesson Learned

Test Data must respect Temporal Semantics. In modern streaming architectures, "Time" is a first-class citizen. You cannot simply perform a `cat log.txt | kafka-console-producer` and expect stateful windowing to function. A robust test bench requires decoupling "Event Generation Time" from "Event Ingestion Time" to simulate the "now" effectively.

4.2.2 Processing Challenge: The Serialization Nightmare

Problem Description

- **Context:** The core business logic involved applying the NLTK VADER (Valence Aware Dictionary and sEntiment Reasoner) model to score every incoming tweet. This was encapsulated in a Python function.
- **Challenge:** Upon submitting the Spark job, it crashed instantly. The driver logs revealed a `PicklingError` and `TaskNotSerializable` exception.
- **Root Cause:** We naively initialized the `SentimentIntensityAnalyzer` object in the global scope of the `main.py` script (on the Driver). When Spark tried to distribute the task to Worker nodes, it attempted to serialize the entire closure, including this complex NLTK object. The analyzer contains compiled C-extensions and open file handles (for lexicon loading) which are fundamentally non-serializable by Python's `pickle` library.

Approaches Considered

1. **MapPartitions API:** Rewriting the logic to use `rdd.mapPartitions()`.
   - *Pros:* Efficient, as it initializes the model only once per partition.
   - *Cons:* Forces a drop from the high-level Structured Streaming DataFrame API down to the low-level RDD API. This bypasses the Catalyst Optimizer, leading to potential performance degradation and code complexity.
2. **Spark Broadcasting:** Broadcasting the object using `spark.sparkContext.broadcast()`.
   - *Pros:* Standard mechanism for sharing read-only data.
   - *Cons:* Failed. Broadcasting still requires the object to be serializable in the first place to be sent to the executors.

Final Approach

We utilized the **Lazy Singleton Pattern inside UDFs**. We declared the analyzer variable as a global variable set to `None`. Inside the actual UDF execution function (which runs on the

Worker), we check `if analyzer is None` and only then perform the import and initialization.

```
analyzer = None
def get_sentiment(text):
    global analyzer
    if analyzer is None:
        from nltk.sentiment.vader import SentimentIntensityAnalyzer
        analyzer = SentimentIntensityAnalyzer()
    return analyzer.polarity_scores(text)['compound']
```

This ensures that the heavy object creation happens strictly within the Executor's memory context, completely bypassing the need for serialization over the network.

Lesson Learned

Execution Context is King in Distributed Systems. The code you write in `main()` does not run where you think it runs. Heavy initialization logic must always be pushed down to the implementation function to ensure it executes on the Worker/Executor nodes, never left on the Driver.

4.2.3 Storage Challenge: The File Fragmentation Issue

Problem Description

- **Context:** The Batch Layer of the Lambda Architecture required archiving raw data to the MinIO Data Lake for future model retraining. Spark Streaming was configured with a default trigger of 5 seconds.
- **Challenge:** After leaving the system running overnight for a longevity test, we observed that query performance on the historical data had degraded severely. Listing the `/tweets` bucket revealed over 15,000 tiny Parquet files, each only 4KB in size.
- **Root Cause:** This is the well-documented file fragmentation inefficiency. Every micro-batch in Spark Streaming commits a set of files. With the default `spark.sql.shuffle.partitions=` Spark was attempting to write up to 200 tiny files every 5 seconds. This fragmentation caused the metadata overhead (opening/closing HTTP connections to MinIO) to dominate the actual data transfer time.

Approaches Considered

1. **Compaction Jobs:** Implementing a separate nightly compaction job to concatenate small files.
    - *Pros:* Standard industry practice.
    - *Cons:* Introduces unrelated operational complexity and requires managing a separate job lifecycle.

2. **Longer Trigger Intervals:** Increasing the trigger to 10 or 15 minutes.
    - *Pros:* Larger files.
    - *Cons:* Increases the ingestion latency. Real-time dashboards sharing this stream would become stale.

Final Approach

We implemented a **Partition-Aware Write Strategy** combined with Stream Decoupling. First, we tuned `spark.sql.shuffle.partitions` down to 10 to match our specific data volume, reducing fragmentation by 20x. Second, we utilized `.partitionBy("date", "hour")` to enforce a hierarchical directory structure. Crucially, we accepted that the Batch Layer does not need sub-second latency. We separated the write streams: the Cassandra stream (Speed Layer) triggers every 5 seconds, while the MinIO stream (Batch Layer) utilizes `Trigger(processingTime minutes")`. This allows Spark to accumulate larger in-memory batches before flushing to disk, significantly improving I/O efficiency.

Lesson Learned

Tune Write Patterns for Read Physics. A Streaming system acts as a "Write" system, but a Data Lake is a "Read" system. You must tune your ingestion write patterns (file sizes, partitioning) to respect the physics of the underlying filesystem, or you will create a swamp, not a lake.

4.2.4 Serving Challenge: Database Consistency Bottlenecks

Problem Description

- **Context:** The final step in the pipeline involves sinking aggregated windowed sentiment scores to Cassandra for the Grafana dashboard to query.
- **Challenge:** As we ramped up the input simulation speed to 1,000 tweets/sec, the Spark logs started filling with `WriteTimeoutException`. The dashboard updates became laggy and intermittent.
- **Root Cause:** We were using the default consistency level of `QUORUM` (requiring majority acknowledgement from replicas) and sending individual `INSERT` statements. The overhead of network round-trips for every single collected metric overwhelmed the single-node Cassandra instance running in Minikube.

Approaches Considered

1. **Vertical Scaling:** Increasing CPU/RAM for the Cassandra pod.
    - *Pros:* Brute-force solution.
    - *Cons:* Not sustainable or scalable; consumes limited Minikube resources.
2. **Client-Side Batching:** Manually grouping rows in Python code before writing.

- *Pros:* Reduces network calls.
- *Cons:* Requires complex logic to handle partial failures within a batch.

Final Approach

We addressed the bottleneck by relaxing the consistency constraints based on the **CAP Theorem**. We modified the connector configuration:

```
spark.cassandra.output.consistency.level = ONE
spark.cassandra.output.batch.size.rows = auto
```

For a sentiment trend dashboard, "Eventual Consistency" is perfectly acceptable—if a data point appears 200ms later, no user notices. We traded strict transactional guarantees (ACID) for massive write throughput (BASE). Additionally, we optimized the Schema Primary Key to be `((symbol), timestamp)` to ensure data clustered efficiently on disk for time-range queries.

Lesson Learned

Consistency is a variable, not a constant. In real-time analytics, strict ACID transactions are often the enemy of throughput. Downgrading to `ONE` is a legitimate engineering decision when Availability and Partition Tolerance are prioritized.

4.2.5 Infrastructure Challenge: Determinism vs Manual Ops

Problem Description

- **Context:** The system grew to include multiple interdependent services: Zookeeper, Kafka, Spark Master, Spark Worker, Cassandra, and Grafana.
- **Challenge:** Manually deploying these services using `kubectl apply -f ...` was error-prone. We frequently encountered race conditions where Kafka would start before Zookeeper was fully "ready", causing the broker to crashloop. This led to "Configuration Drift" where the live cluster state diverged from the codebase.

Final Approach

We adopted **Terraform for Orchestration**. We utilized the `depends_on` meta-argument to enforce a strict Directed Acyclic Graph (DAG) for startup: Zookeeper $\rightarrow$ Kafka $\rightarrow$ Spark. More importantly, we implemented **ConfigMap Injection** for Grafana dashboards. Instead of manually editing the dashboard in the UI, we saved the JSON model and utilized Terraform to inject it as a Kubernetes `ConfigMap` mounted into the Grafana pod. This ensured that destroying and recreating the cluster always brought back the exact same visualization state, treating dashboards as version-controlled code.

Lesson Learned

Determinism pays dividends. Infrastructure as Code (IaC) is not just for cloud deployments; it saved hours of debugging time locally by ensuring that every deployment resulted in an identical, stable state. The ability to tear down and rebuild the entire stack in one command (`terraform apply`) is invaluable for iterative development.

Chapter 5

CONCLUSION AND FUTURE WORK

5.1 PROJECT SUMMARY

This project successfully demonstrated the implementation of a comprehensive **Big Data Lambda Architecture** for financial sentiment analysis. By integrating **Apache Kafka** for ingestion, **Spark Structured Streaming** for processing, **MinIO** for archival, and **Cassandra** for serving, we constructed a robust pipeline capable of handling high-velocity data streams while ensuring data durability.

The system meets its primary objectives:

1. **Low Latency:** The "Speed Layer" achieves a data-to-dashboard latency of under 5 seconds, enabling real-time market monitoring.
2. **Scalability:** The decoupled design allowed distinct components (Producers vs. Consumers) to scale independently.
3. **Maintainability:** The adoption of **Terraform** and **Docker** ensured that the complex distributed infrastructure could be deployed deterministically in under 3 minutes.

5.2 KEY ACHIEVEMENTS

Beyond the functional requirements, the project highlighted several engineering achievements:

- **High-Fidelity Simulation:** We overcame the limitations of expensive financial APIs by engineering a "Time-Travel" replay mechanism that accurately mimicked production traffic patterns.
- **Optimization for Constraints:** We successfully tuned heavy Big Data tools (Spark, Cassandra) to run stable within a resource-constrained single-node Minikube environment, proving that "Big Data" principles can be developed locally.
- **Visual Intelligence:** The Grafana integration provided actionable insights, not just raw logs, converting technical metrics into business value.

5.3 LIMITATIONS

While operational, the current implementation has inherent boundaries:

- **NLP Simplicity:** The VADER model, while fast, is rule-based and struggles with sarcasm or complex context compared to Transformer-based models (BERT/RoBERTa).
- **Single-Point of Failure:** Running on Minikube meant that true high-availability (e.g., Kafka multi-broker replication) could not be fully demonstrated.

5.4 FUTURE DIRECTIONS

To evolve this prototype into a production-grade system, the following enhancements are proposed:

1. **Cloud Migration:** Deploying the Terraform stack to AWS EKS (Elastic Kubernetes Service) and replacing MinIO with Amazon S3 for infinite storage scaling.
2. **Advanced AI Integration:** Replacing the VADER UDF with a deployed REST endpoint serving a FinBERT model, allowing Spark to query a GPU-accelerated inference server.
3. **Predictive Analytics:** Implementing a new Spark MLlib pipeline in the "Batch Layer" to correlate sentiment trends with actual stock price movements, moving from "Descriptive Analytics" (what happened?) to "Predictive Analytics" (what will happen?).