HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

School of Information and Communication Technology

STEAM ANALYTICS: REAL-TIME TELEMETRY & CRISIS DETECTION

Big Data Storage and Processing

161705 - IT4868E

Semester 2025.1

Group 22	**Student ID**
Nguyễn Trọng Phương Bách	20225473
Vũ Minh Hiếu	20225494
Bùi Nguyên Khải	20225501
Trần Ngọc Quang	20225523
Nguyễn Việt Tiến	20225533

Supervisor

Ph.D. Tran Viet Trung

Hanoi, January 2026

Contents

1 Problem Definition

1.1 Background

Steam, developed by Valve Corporation, stands as the world's largest digital distribution platform for PC gaming. As of 2024, the platform hosts over 50,000 games and serves more than 120 million monthly active users. The sheer scale of Steam's operations generates massive amounts of data every second, including game metadata, user reviews numbering in the millions, and real-time player count statistics that fluctuate continuously based on time zones, game releases, and special events like seasonal sales.

Analyzing Steam data at scale presents a classic big data challenge characterized by the three V's: volume, velocity, and variety. In terms of volume, Steam's catalog spans tens of thousands of games, each with potentially thousands of user reviews. Popular titles like Counter-Strike 2 (App ID 730) have accumulated over one million reviews, while the platform collectively hosts hundreds of millions of review entries. In terms of velocity, player counts update in near real-time as users launch and close games, creating a continuous stream of time-series data that must be captured and processed promptly to remain useful for analytics. In terms of variety, the data comes in multiple formats: structured fields like prices, genres, and release dates; semi-structured content like user reviews containing free-form text; and purely numerical time-series data representing concurrent player statistics.

A fourth challenge specific to this project is the verification aspect: Steam enforces strict rate limits on its public APIs, typically allowing only a few requests per second before returning HTTP 429 (Too Many Requests) responses. This constraint means that naive data collection approaches would either be blocked or would take prohibitively long to gather meaningful datasets. Our system must therefore implement intelligent buffering, request throttling, and message queuing to collect data reliably without violating these rate limits.

1.2 Problem Statement

The goal of this project is to design and implement a production-grade real-time big data analytics pipeline for Steam gaming data. This pipeline must satisfy four primary requirements.

First, the system must perform reliable data ingestion from Steam's public APIs, specifically the Store API for game details and reviews (`store.steampowered.com/api/appdetails` and

`store.steampowered.com/appreviews`) as well as the ISteamUserStats API for player counts (`api.steampowered.com/ISteamUserStats/GetNumberOfCurrentPlayers`). The ingestion layer must handle API rate limits gracefully by implementing appropriate delays between requests and buffering messages through a distributed message queue.

Second, the system must process streaming data with windowed aggregations to compute meaningful analytics. This includes computing hourly review sentiment trends (positive versus negative recommendations), aggregating genre distributions across the game catalog, and tracking player activity patterns using 10-minute sliding windows. These computations must happen in near real-time as data arrives, not in delayed batch jobs.

Third, the system must implement a dual-storage architecture to serve different query patterns. Raw data must be persisted to a distributed file system in columnar format (Parquet on HDFS) for historical analysis and potential reprocessing. Simultaneously, pre-aggregated results must be written to a document database (MongoDB) optimized for fast point queries by Grafana dashboards.

Fourth, the system must provide interactive visualization through Grafana dashboards that display key performance indicators in real-time, including total review counts, peak concurrent players, sentiment trends over time, and genre popularity charts.

From a technical perspective, the entire pipeline must run on Kubernetes to demonstrate containerized deployment practices suitable for cloud-native environments. All inter-component communication involving Kafka must be encrypted using TLS to meet modern security standards.

1.3 Proprietary Analytics Metrics

Beyond standard aggregations, we designed two proprietary metrics that address real challenges in gaming analytics.

1.3.1 Review Bomb Detection Algorithm

Standard sentiment analysis fails to distinguish between genuine negative feedback and coordinated "review bombing"—where a game is unfairly targeted by an organized mob due to external controversies (developer statements, pricing changes, or unrelated company decisions). Detecting these events in real-time allows platform operators to flag suspicious activity and protect game developers from artificial reputation damage.

Our implementation aggregates reviews per game (without windowing) to compute overall sentiment ratios that match Steam's display format. A **Bomb Event** is triggered when the following condition holds:

$$\frac{R_{\text{negative}}}{R_{\text{total}}} > 0.8 \quad \text{AND} \quad R_{\text{total}} > 10 \tag{1}$$

Where R_{negative} is the count of negative reviews for a game and R_{total} is the total review count. The first condition ensures overwhelming negativity (over 80% negative), while the second ensures sufficient sample size (more than 10 reviews) to avoid false positives from low-activity games.

The Spark streaming application computes both `positive_ratio` and `negative_ratio` for each game, writing results to a dedicated `review_bomb_alerts` MongoDB collection. The Grafana dashboard displays:

- A stat panel showing the count of currently flagged games (with red background when alerts exist)
- A sentiment table showing all games ranked by their positive/negative review ratios

1.3.2 Hype-to-Disappointment Index (Proposed)

As a future enhancement, we propose tracking a **Hype-to-Disappointment Index (HDI)** to predict potential refund waves during new game releases. New releases often show a pattern: high player counts at launch followed by rapid sentiment decline as players discover bugs or unmet expectations. The proposed metric would calculate:

$$\text{HDI}(t) = \frac{\Delta P_{\text{peak}}(t)}{\Delta S_{\text{positive}}(t)} = \frac{P_{\text{current}} - P_{\text{24h\_avg}}}{S_{\text{current}} - S_{\text{24h\_avg}}} \tag{2}$$

Where P represents concurrent player count and S represents the positive sentiment ratio. A high HDI (player count rising while sentiment drops) would indicate that players are trying the game but quickly becoming disappointed—a leading indicator of refund requests within 2 hours of purchase.

Implementation Complexity: This metric requires a stream-to-stream join between the player count stream and the review sentiment stream, plus 24-hour rolling state management for both. While our Kappa Architecture fully supports this capability, the implementation complexity ex-

ceeded our project timeline. We document this as a valuable future enhancement that demonstrates the extensibility of our pipeline design.

1.4 Data Sources

The pipeline collects data from three Steam API endpoints as shown in Table 1. Each endpoint provides different types of information and has distinct rate limiting characteristics that influenced our producer design.

Table 1: Steam API Data Sources

API Endpoint	**Data Type**	**Key Fields**
`appdetails`	Game Metadata	App ID, name, type, genres, price, developers, publishers, release date
`appreviews`	User Reviews	Review ID, author, voted_up (sentiment), votes_up/down, review text, timestamp
`ISteamUserStats`	Player Counts	App ID, current player count, timestamp

The Store API endpoints (`appdetails` and `appreviews`) require a 1.5-second delay between requests to avoid triggering rate limits. The `appdetails` endpoint returns comprehensive game metadata including pricing in different regions, supported languages, DLC information, and category tags. The `appreviews` endpoint supports pagination through a cursor mechanism and can filter reviews by language, sentiment, and recency. We configured our producers to fetch English reviews only and retrieve up to 300 reviews per game (3 pages of 100 reviews each).

The ISteamUserStats API has less strict rate limiting, allowing requests every 0.5 seconds. This endpoint returns only the current player count as a single integer value, but its higher query frequency enables more granular time-series data collection. The combination of these three data sources provides a comprehensive view of the Steam gaming ecosystem: what games exist, what players think about them, and how actively they are being played.

1.5 Team Contributions

Table 2 shows how work was divided among team members.

Table 2: Team Member Responsibilities

Member	Student ID	Primary Responsibilities
Nguyễn Trọng Phương Bách	20225473	PySpark, Stream Processing
Vũ Minh Hiếu	20225494	Data Ingestion, Steam API, Kafka
Bùi Nguyên Khải	20225501	Infrastructure, Kubernetes
Trần Ngọc Quang	20225523	Grafana, Prometheus
Nguyễn Việt Tiến	20225533	HDFS, MongoDB

2 Architecture and Design

2.1 Architecture Choice: Kappa vs Lambda

When designing a big data pipeline, there are two main architecture patterns to consider: Lambda Architecture and Kappa Architecture. Lambda Architecture, introduced by Nathan Marz, uses two separate processing paths. A batch layer processes all historical data periodically to produce accurate views, while a speed layer handles real-time data to provide low-latency updates. A serving layer then merges results from both paths. This approach is powerful but complex because developers must maintain two separate codebases with identical business logic.

Kappa Architecture, proposed by Jay Kreps at LinkedIn, simplifies this by using only a streaming layer for everything. All data, whether historical or real-time, flows through the same stream processing engine. Reprocessing is handled by replaying data from a message log (typically Kafka) rather than running separate batch jobs.

We chose Kappa Architecture for this project after careful analysis of our requirements, summarized in Table 3. Our data originates from a single source (Steam API) as a continuous stream of events, making a dedicated batch layer unnecessary. The processing logic (windowed aggregations, counts, averages) applies identically to both real-time data and historical replays. Kafka's configurable retention period (we use 7 days, or 168 hours) provides the replay capability needed for reprocessing without maintaining a separate batch storage layer like HDFS for batch input. Most importantly, Kappa eliminates the need to synchronize two codebases, reducing development and maintenance effort significantly.

2.1.1 Why Lambda Would Fail for Gaming Telemetry

Beyond general simplicity arguments, there are domain-specific reasons why Lambda Architecture is unsuitable for gaming analytics. In the gaming industry, critical events—server outages, game-breaking bugs, review bombing attacks, or login failures—must be detected in **seconds**, not hours. Lambda Architecture's batch layer introduces an inherent latency of $T + 1$ hours (or even $T + 1$ days for daily batch jobs), which renders the data useless for operational monitoring.

Consider the Review Bomb Detection scenario from Equation 1. A coordinated attack can unfold within 30 minutes of a controversial announcement. By the time a Lambda batch job processes this data overnight, the damage to the game's reputation is already done. Only a pure streaming architecture can detect such events in real-time, enabling timely intervention.

Similarly, the proposed Hype-to-Disappointment Index (Equation 2) would lose all predictive value if computed with batch latency. Refund requests peak within 2 hours of purchase; a next-day batch report provides historical curiosity rather than actionable intelligence. This illustrates why our streaming-first architecture is essential for future metric extensions.

Table 3: Architecture Comparison and Decision Rationale

Factor	**Choice**	**Reason**
Data Source	Kappa	Single streaming source from Steam API
Processing Logic	Kappa	Same logic for real-time and historical
Reprocessing	Kappa	Kafka 7-day retention handles replay
Complexity	Kappa	Single codebase, easier to maintain
Latency Requirement	Kappa	Sub-minute detection for crisis events

It is worth noting that we still use HDFS for archival storage of raw data in Parquet format. However, this serves as a data lake for historical queries and potential future batch analytics, not as input to a Lambda-style batch layer. The primary processing path remains purely streaming through Spark Structured Streaming.

2.2 System Overview

Figure 1 illustrates the complete pipeline architecture. The system follows a classic streaming ETL pattern with clear separation between ingestion, processing, storage, and visualization layers.

At the ingestion layer, three Python producer scripts run as Kubernetes CronJobs, periodically fetching data from Steam APIs. Each producer handles one type of data: game metadata (charts

producer), user reviews (reviews producer), and player counts (players producer). These producers use the `confluent-kafka` library to serialize messages as JSON and publish them to corresponding Kafka topics over TLS-encrypted connections.

The message queue layer consists of Apache Kafka running with Stackable operators on Kubernetes. Kafka provides durable, ordered storage of incoming messages with a 7-day retention window. Three topics partition the data by type: `game_info` for metadata, `game_comments` for reviews, and `game_player_count` for player statistics. Kafka's distributed commit log design ensures no data loss even if downstream processors temporarily fail.

The processing layer uses Apache Spark Structured Streaming to consume from Kafka topics. Three separate Spark applications (one per topic) parse JSON messages, apply windowed aggregations, and write results to storage. The reviews processor uses 1-hour tumbling windows to aggregate sentiment metrics. The players processor uses 10-minute tumbling windows to track activity patterns. The charts processor performs genre explosion (unnesting the genres array) and simple counting without windowing.

The storage layer implements a dual-write pattern. Every Spark application writes to both HDFS (cold storage) and MongoDB (hot storage) simultaneously using separate streaming queries. HDFS stores raw data in Parquet format under `/user/stackable/archive/` for long-term retention and potential reprocessing. MongoDB stores pre-aggregated results in the `game_analytics` database for fast dashboard queries, with the exception of `review_bomb_alerts` which uses the `bigdata` database.

The visualization layer uses Grafana with the MongoDB datasource plugin to query aggregated data and render interactive charts. Prometheus monitors system health metrics including Kafka broker status, Spark job progress, and MongoDB performance indicators.

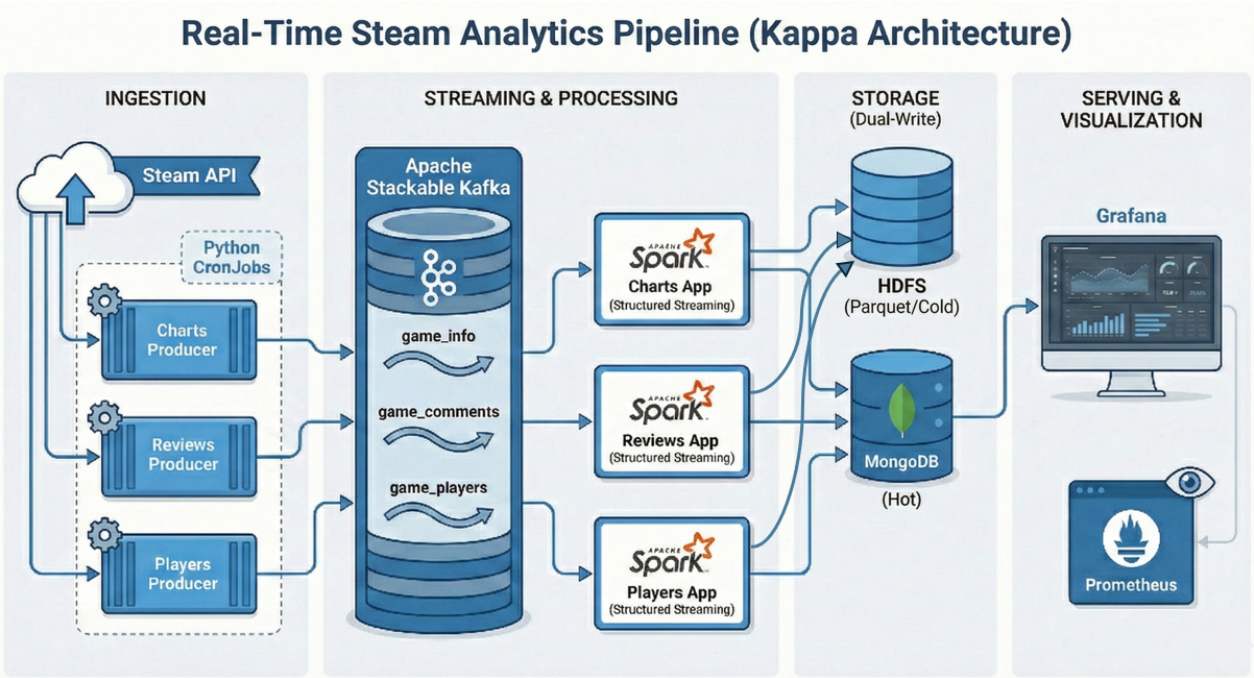

Figure 1: Steam Analytics Pipeline Architecture

2.3 Technology Stack

Table 4 lists all technologies used in the project with their versions and purposes. We deliberately selected components from the Stackable Data Platform ecosystem to simplify Kubernetes deployment and ensure version compatibility.

Table 4: Technology Stack

Component	Version	Purpose
Kubernetes	1.31	Container orchestration (Docker Desktop)
Stackable Operators	25.7.0	Manage big data components on K8s
Apache ZooKeeper	3.9.3	Coordination for Kafka and HDFS
Apache Kafka	3.9.1	Message streaming with TLS encryption
Apache Spark	3.5.6	Structured Streaming with Scala 2.12
Apache HDFS	3.4.1	Distributed file storage (HA mode)
MongoDB	7.0	Document database for aggregations
Prometheus	2.x	Time-series metrics collection
Grafana	11.0	Dashboard visualization
Python	3.11	Producer scripts with confluent-kafka

The Stackable Data Platform deserves special mention. It is an open-source project that provides Kubernetes operators for deploying and managing big data infrastructure. Instead of writing complex StatefulSet configurations manually, we declare high-level custom resources (such as `KafkaCluster` or `HdfsCluster`) and the Stackable operators handle pod scheduling, configuration management, and lifecycle operations. This significantly reduced our infrastructure code and ensured best practices for running stateful workloads on Kubernetes.

Starting from Stackable version 24.11, TLS encryption is enforced by default for all Kafka communication. While this improves security, it also added complexity to our producer and Spark configurations, as documented in the Implementation Details section.

2.4 Data Flow Design

The pipeline processes three types of data, each with its own Kafka topic, message schema, and Spark application. This section details the message formats and processing logic for each flow.

2.4.1 Game Information Flow

The charts producer fetches game metadata from the `appdetails` API and sends it to the `game_info` Kafka topic. Each message contains comprehensive game information including pric-

ing, categories, and review statistics:


```
{
  "appid": 10,
  "name": "Counter-Strike",
  "primary_genre": null,
  "type": "game",
  "release_date": "Nov 1, 2000",
  "is_free": false,
  "short_description": "Play the world's number 1 online action game...",
  "developers": ["Valve"],
  "publishers": ["Valve"],
  "genres": ["Action"],
  "price_overview": {
    "currency": "USD",
    "initial": 999,
    "final": 199,
    "discount_percent": 80,
    "initial_formatted": "$9.99",
    "final_formatted": "$1.99"
  },
  "categories": ["Multi-player", "PvP", "Online PvP", "Valve Anti-Cheat enabled"],
  "metacritic": 88,
  "recommendations": 165122,
  "achievements_count": 0,
  "timestamp_scraped": "2026-01-04T14:36:17.823027"
}
```


Listing 1: game_info Message Schema

The Spark charts application reads this topic and performs genre explosion. A game with multiple genres (for example, ["Action", "RPG", "Indie"]) is exploded into separate rows, one per genre. This allows accurate counting of how many games belong to each genre. The aggregation then groups by genre and counts the number of distinct App IDs.

2.4.2 Reviews Flow

The reviews producer fetches user reviews from the `appreviews` API and sends them to the `game_comments` topic. Each message contains:


```
{
  "app_id": "730",
  "review_id": "123456789",
  "author_steamid": "76561198012345678",
  "language": "english",
  "voted_up": true,
```

```
  "votes_up": 42,
  "weighted_vote_score": 0.856,
  "timestamp_created": 1705312800,
  "review_text": "Great game with amazing mechanics...",
  "scraped_at": "2026-01-15T10:30:00Z"
}
```

Listing 2: game_comments Message Schema

The Spark reviews application aggregates reviews in 1-hour tumbling windows. The `voted_up` boolean field indicates whether the review is positive (recommended) or negative (not recommended). Within each window, the application calculates the total number of reviews and the average weighted vote score, grouped by sentiment. A 10-minute watermark handles late-arriving data. The aggregation output looks like:

```
{
  "window": { "start": "2026-01-15T10:00:00Z", "end": "2026-01-15T11:00:00Z" },
  "recommended": true,
  "total_reviews": 156,
  "avg_quality": 0.723
}
```

Listing 3: Reviews Aggregation Output in MongoDB

2.4.3 Player Counts Flow

The players producer fetches current player counts from the `ISteamUserStats` API and sends them to the `game_player_count` topic:

```
{
  "appid": 730,
  "player_count": 847523,
  "timestamp": "2026-01-15T10:30:00Z"
}
```

Listing 4: game_player_count Message Schema

The Spark players application aggregates in 10-minute windows grouped by App ID, computing maximum and average player counts. The shorter window (compared to 1 hour for reviews) reflects the higher update frequency of player statistics and enables more granular time-series visualization. A 5-minute watermark is used for late data handling.

2.5 Storage Strategy

We implemented a dual-storage architecture to serve different query patterns, a common practice in modern data platforms. This approach follows the principles of the Medallion Architecture pattern, specifically implementing Bronze (raw) and Gold (aggregated) layers while skipping the intermediate Silver layer to minimize latency.

2.5.1 Bronze Layer: Raw Data Lake (HDFS)

HDFS serves as our Bronze layer, persisting all raw data in Apache Parquet format. Parquet is a columnar storage format optimized for analytical queries, offering efficient compression (we observed approximately 10:1 compression ratios) and predicate pushdown capabilities. Data is organized under `/user/stackable/archive/` with subdirectories for each data type: `reviews/`, `charts/`, and `players/`.

This raw data archive serves as the immutable "source of truth" with two key purposes: first, it enables ad-hoc historical analysis using tools like Spark SQL or Hive; second, it provides the ability to reprocess data with updated business logic by replaying from HDFS if Kafka retention has expired.

2.5.2 Gold Layer: Business-Ready Aggregates (MongoDB)

MongoDB serves as our Gold layer, storing pre-aggregated results optimized for dashboard queries. The `game_analytics` database contains three collections: `steam_reviews`, `steam_charts`, and `steam_players`. Each collection stores windowed aggregation results with indexed fields for efficient time-range queries. We configured TTL (Time-To-Live) indexes to automatically expire old aggregation records after 30 days, preventing unbounded storage growth while retaining sufficient history for trend analysis.

2.5.3 Why We Skip the Silver Layer

In a traditional Medallion Architecture, the Silver layer acts as a cleaned, deduplicated, and validated intermediate stage between raw ingestion and business aggregates. However, we deliberately skip this persistent Silver layer for several reasons:

- **Latency Requirements:** Our crisis detection use case demands sub-minute insight generation. Adding an intermediate persistence layer would introduce unnecessary latency.
- **Stream Processing:** Spark Structured Streaming performs cleaning and validation in-memory during the transformation from Bronze to Gold. There's no need to persist this intermediate state.
- **Simplicity:** The direct Bronze→Gold flow reduces infrastructure complexity and operational overhead.
- **Replayability:** If we need to reprocess with different business logic, we can replay from the Bronze layer (HDFS) or from Kafka's 7-day retained messages.

This "stream-to-gold" approach is appropriate for streaming analytics workloads where real-time aggregation is the primary goal. For use cases requiring complex data quality rules or multiple downstream consumers with different aggregation needs, a persistent Silver layer would be justified.

Table 5 summarizes the storage characteristics.

Table 5: Storage Layer Comparison (Medallion Architecture)

Aspect	**Bronze (HDFS)**	**Gold (MongoDB)**
Purpose	Immutable source of truth	Business-ready aggregates
Data Format	Parquet (columnar)	BSON (document)
Content	Raw, unaggregated	Pre-aggregated
Retention	Indefinite	30 days (TTL)
Query Pattern	Batch analytics	Real-time dashboards
Access Tool	Spark SQL, Hive	Grafana, Mongo Shell
Transformation	None (raw data)	Windowed aggregations

This separation ensures that we never lose raw data (useful for debugging, auditing, or developing new analytics) while also providing fast access to computed metrics that power real-time monitoring dashboards. The Bronze→Gold pattern gives us the best of both worlds: data durability and query performance.

3 Implementation Details

3.1 Infrastructure Setup

All components run on Kubernetes using Stackable operators. Stackable provides custom resource definitions (CRDs) that allow declarative management of distributed systems. Instead of manually configuring StatefulSets, Services, ConfigMaps, and PersistentVolumeClaims, we simply define a high-level resource and the operator handles the underlying complexity.

To bootstrap the infrastructure, we first install the required Stackable operators using Helm:

```
helm repo add stackable-stable https://repo.stackable.tech/repository/helm-stable/
helm install zookeeper-operator stackable-stable/zookeeper-operator
helm install kafka-operator stackable-stable/kafka-operator
helm install hdfs-operator stackable-stable/hdfs-operator
helm install spark-k8s-operator stackable-stable/spark-k8s-operator
helm install commons-operator stackable-stable/commons-operator
helm install secret-operator stackable-stable/secret-operator
```

Listing 5: Installing Stackable Operators

Once the operators are running, we apply YAML manifests defining our clusters. Listing 6 shows the Kafka cluster configuration. Notice how the `spec.clusterConfig.tls` section enables TLS encryption using Stackable's built-in secret operator. The `configOverrides` section sets the 7-day retention period (168 hours) essential for Kappa Architecture replayability.

```
apiVersion: kafka.stackable.tech/v1alpha1
kind: KafkaCluster
metadata:
  name: simple-kafka
spec:
  image:
    productVersion: 3.9.1
  clusterConfig:
    tls:
      serverSecretClass: tls
      internalSecretClass: tls
    zookeeperConfigMapName: simple-kafka-znode
  brokers:
    roleGroups:
      default:
        replicas: 1
        config:
          resources:
            memory:
```

```
            limit: 2Gi
      configOverrides:
        server.properties:
          log.retention.hours: "168"
          log.segment.bytes: "104857600"
```

Listing 6: Kafka Cluster Configuration (kafka.yaml)

HDFS runs in high-availability mode with 2 NameNodes, 1 DataNode, and 1 JournalNode. The HA configuration uses ZooKeeper for automatic failover between NameNodes. Listing 7 shows the HDFS cluster configuration.

```
apiVersion: hdfs.stackable.tech/v1alpha1
kind: HdfsCluster
metadata:
  name: simple-hdfs
spec:
  image:
    productVersion: 3.4.1
  clusterConfig:
    zookeeperConfigMapName: simple-hdfs-znode
    dfsReplication: 1
  nameNodes:
    config:
      listenerClass: external-stable
      resources:
        memory:
          limit: 512Mi
    roleGroups:
      default:
        replicas: 2
  dataNodes:
    roleGroups:
      default:
        replicas: 1
  journalNodes:
    roleGroups:
      default:
        replicas: 1
```

Listing 7: HDFS Cluster Configuration (hdfs.yaml)

MongoDB runs as a standalone deployment (not a Stackable resource) with 1GB persistent storage. For a production environment, we would use a MongoDB replica set, but a single instance suffices for our development and demonstration purposes.

Table 6 shows the memory allocated to each component. The total footprint of approximately 4.5GB allows the entire pipeline to run on a developer laptop using Docker Desktop's Kubernetes.

Table 6: Resource Allocation

Component	**Instances**	**Memory Limit (each)**
ZooKeeper	1	512Mi
HDFS NameNode	2	512Mi
HDFS DataNode	1	512Mi
HDFS JournalNode	1	256Mi
Kafka Broker	1	2Gi
MongoDB	1	512Mi
Total		**∼4.5GB**

Figure 2 shows all pods running after successful deployment.

```
PS D:\Study\Code\Projects\BigData\bigdata-20251-steam> kubectl get pods
NAME                                          READY   STATUS    RESTARTS   AGE
commons-operator-deployment-5fc5bdb744-654cc  1/1     Running   2          77d
grafana-6b6b697dc6-5hfvp                      1/1     Running   0          29m
hdfs-operator-deployment-69bfb444cb-z7bxg     1/1     Running   2          77d
kafka-operator-deployment-6578f5cb78-zvtsq    1/1     Running   2          77d
listener-operator-deployment-9d65d44d7-mqx8f  2/2     Running   3          77d
listener-operator-node-daemonset-h4gb5        2/2     Running   3          77d
mongo-express-6666f95bb6-knxlx                1/1     Running   1          2d9h
mongodb-85f44678dd-v9qlz                      1/1     Running   1          30h
mongodb-exporter-568d9bbf56-8c9ml             1/1     Running   1          43h
prometheus-6d8bd8948b-ln42t                   1/1     Running   1          43h
secret-operator-daemonset-585xr               3/3     Running   4          77d
simple-hdfs-datanode-default-0                1/1     Running   0          2m45s
simple-hdfs-journalnode-default-0             1/1     Running   0          2m45s
simple-hdfs-namenode-default-0                2/2     Running   0          2m45s
simple-hdfs-namenode-default-1                2/2     Running   0          2m6s
simple-kafka-broker-default-0                 2/2     Running   0          2m45s
simple-zk-server-default-0                    1/1     Running   0          3m16s
spark-k8s-operator-deployment-567585fbd5-fpcf9 1/1    Running   2          77d
steam-charts-app-t6rp2                        1/1     Running   0          51s
steam-players-app-qpkc2                       1/1     Running   0          51s
steam-reviews-app-dcqwm                       1/1     Running   0          51s
zookeeper-operator-deployment-6bd67d78dd-t4r7x 1/1    Running   2          77d
PS D:\Study\Code\Projects\BigData\bigdata-20251-steam>
```

Figure 2: kubectl get pods screenshot

3.2 Data Ingestion

Three Python producer scripts collect data from Steam APIs and send it to Kafka. All producers share common utilities defined in `steam_utils.py` which handles Kafka configuration, API requests, and message serialization.

The shared Kafka producer configuration handles both plaintext (for local testing) and SSL (for Kubernetes deployment) connections. Listing 8 shows the producer setup code.

```python
from confluent_kafka import Producer
import os, json

BOOTSTRAP_SERVERS = os.getenv("BOOTSTRAP_SERVERS", "localhost:9092")
KAFKA_SECURITY_PROTOCOL = os.getenv("KAFKA_SECURITY_PROTOCOL", "PLAINTEXT")
KAFKA_SSL_CA_LOCATION = os.getenv("KAFKA_SSL_CA_LOCATION", "")

def setup_producer():
    producer_conf = {
        "bootstrap.servers": BOOTSTRAP_SERVERS,
        "client.id": f"steam-producer-{os.getenv('HOSTNAME', 'local')}",
        "linger.ms": 5,
    }
    if KAFKA_SECURITY_PROTOCOL == "SSL":
        producer_conf["security.protocol"] = "SSL"
        producer_conf["ssl.endpoint.identification.algorithm"] = "none"
        if KAFKA_SSL_CA_LOCATION:
            producer_conf["ssl.ca.location"] = KAFKA_SSL_CA_LOCATION
    return Producer(producer_conf)

def kafka_send(producer, topic, value, key=None):
    payload = json.dumps(value, default=str, ensure_ascii=False)
    producer.produce(topic, key=key, value=payload)
    producer.poll(0)
```

Listing 8: Kafka Producer Configuration (steam_utils.py)

The API fetching functions implement rate limiting to avoid HTTP 429 responses. Listing 9 shows the functions for fetching game details and player counts.

```python
import requests, time

def get_app_details(appid):
    url = "https://store.steampowered.com/api/appdetails/"
    try:
        r = requests.get(url, params={"appids": appid, "cc": "us"}, timeout=20)
        if r.status_code == 200:
            return r.json().get(str(appid), {})
        elif r.status_code == 429:
            print(f"Rate limited for {appid}. Sleeping 5s...")
            time.sleep(5)
    except Exception as e:
        print(f"Error fetching {appid}: {e}")
    return {}

def get_current_players(appid):
    url = "https://api.steampowered.com/ISteamUserStats/GetNumberOfCurrentPlayers/v1/"
```

```
    try:
        r = requests.get(url, params={'appid': appid}, timeout=10)
        if r.status_code == 200:
            return r.json().get('response', {}).get('player_count', 0)
    except Exception as e:
        print(f"Player count error: {e}")
    return -1
```

Listing 9: Steam API Fetching Functions

Each producer is deployed as a Kubernetes CronJob with appropriate schedules. The reviews producer runs every 5 minutes, the charts producer runs every 15 minutes, and the players producer runs every 5 minutes. This approach ensures continuous data collection without manual intervention. The CronJob mounts the CA certificate from a Kubernetes Secret for TLS authentication with Kafka.

Handling Kafka TLS was one of the significant challenges in this project. Stackable 25.7.0 enforces TLS by default, and there is no configuration option to disable it. We had to configure the Python producers with the `ssl.ca.location` parameter pointing to the CA certificate mounted from Stackable's Secret Operator. The environment variable `KAFKA_SSL_CA_LOCATION` is set to `/stackable/tls/ca.crt` in the CronJob specification.

3.3 Stream Processing

Three Spark Structured Streaming applications process data from Kafka topics. Each application is defined as a SparkApplication custom resource in Kubernetes and runs continuously, processing data as it arrives.

3.3.1 Reviews Processor

The reviews processor reads from the `game_comments` topic and performs windowed aggregations. Listing 10 shows the core processing logic.

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, from_json, avg, count, window
from pyspark.sql.types import StructType, StructField, StringType, IntegerType,
    FloatType, BooleanType

spark = SparkSession.builder.appName("SteamReviews").getOrCreate()

# Schema matching Kafka message format
```

```
schema = StructType([
    StructField("app_id", StringType(), True),
    StructField("review_id", StringType(), True),
    StructField("voted_up", BooleanType(), True),
    StructField("votes_up", IntegerType(), True),
    StructField("weighted_vote_score", FloatType(), True),
    StructField("timestamp_created", IntegerType(), True),
])

# Read from Kafka with SSL
raw_df = (spark.readStream.format("kafka")
    .option("kafka.bootstrap.servers", bootstrap)
    .option("subscribe", "game_comments")
    .option("startingOffsets", "earliest")
    .option("kafka.security.protocol", "SSL")
    .option("kafka.ssl.truststore.location", "/truststore/truststore.p12")
    .option("kafka.ssl.truststore.type", "PKCS12")
    .load())

# Parse JSON and cast timestamp
parsed_df = raw_df.select(
    from_json(col("value").cast("string"), schema).alias("data")
).select(
    col("data.voted_up").alias("recommended"),
    col("data.weighted_vote_score"),
    col("data.timestamp_created").cast("timestamp").alias("timestamp")
)

# Aggregation: 1-hour tumbling windows with 10-minute watermark
analytics_df = parsed_df \
    .withWatermark("timestamp", "10 minutes") \
    .groupBy(window(col("timestamp"), "1 hour"), col("recommended")) \
    .agg(
        count("*").alias("total_reviews"),
        avg("weighted_vote_score").alias("avg_quality")
    )
```

Listing 10: Reviews Spark Application (process_reviews.py)

The 10-minute watermark allows late-arriving data to be included in aggregations up to 10 minutes past the window boundary. After that, Spark discards late records to maintain bounded state and prevent memory growth.

3.3.2 Charts Processor

The charts processor performs genre explosion using Spark's `explode` function. Listing 11 shows how a single game with multiple genres becomes multiple rows for accurate counting.

```
from pyspark.sql.functions import explode
from pyspark.sql.types import ArrayType

schema = StructType([
    StructField("appid", IntegerType(), True),
    StructField("name", StringType(), True),
    StructField("genres", ArrayType(StringType()), True),
])

# Explode: [Action, RPG] -> Row(genre=Action), Row(genre=RPG)
exploded_df = parsed_df.select(
    "appid",
    explode(col("genres")).alias("genre")
)

# Count games per genre
analytics_df = exploded_df.groupBy("genre").agg(
    count("appid").alias("total_games")
)
```

Listing 11: Charts Spark Application with Genre Explosion

3.3.3 Players Processor

The players processor uses shorter 10-minute windows to capture player count fluctuations with higher granularity.

```
from pyspark.sql.functions import max, avg

# 10-minute windows per game
analytics_df = parsed_df \
    .withWatermark("timestamp", "5 minutes") \
    .groupBy(window(col("timestamp"), "10 minutes"), col("appid")) \
    .agg(
        max("player_count").alias("max_players"),
        avg("player_count").alias("avg_players")
    )
```

Listing 12: Players Spark Application with 10-Minute Windows

3.3.4 Dual-Write Pattern

All three Spark applications implement the dual-write pattern, writing to both HDFS (cold) and MongoDB (hot) simultaneously using separate streaming queries. Listing 13 shows how this is implemented.

```
# Write Raw to HDFS (Cold Storage)
query_cold = parsed_df.writeStream \
    .format("parquet") \
    .trigger(processingTime='1 minute') \
    .option("path", "/user/stackable/archive/reviews") \
    .option("checkpointLocation", "/user/stackable/checkpoints/reviews_cold") \
    .outputMode("append") \
    .start()

# Write Aggregated to MongoDB (Hot Storage)
query_hot = analytics_df.writeStream \
    .format("mongodb") \
    .trigger(processingTime='10 seconds') \
    .option("checkpointLocation", "/user/stackable/checkpoints/reviews_hot") \
    .option("database", "game_analytics") \
    .option("collection", "steam_reviews") \
    .outputMode("complete") \
    .start()

spark.streams.awaitAnyTermination()
```

Listing 13: Dual-Write Pattern Implementation

The cold write uses append mode (adding new rows) with 1-minute trigger intervals to batch small files. The hot write uses complete mode (full table replacement) with 10-second triggers for near real-time dashboard updates. The MongoDB Spark Connector (version 10.5.0) handles the complete mode by upserting records based on the window and grouping keys.

3.3.5 TLS Certificate Handling in Spark

Spark applications face a unique TLS challenge: Stackable generates certificates in PEM format, but the Kafka Spark connector expects Java keystores in PKCS12 format. We solved this using Kubernetes init containers that run before the main Spark container. Listing 14 shows the init container configuration.

```
initContainers:
  - name: create-truststore
    image: eclipse-temurin:17-jdk
```

```
    command:
      - sh
      - -c
      - |
        keytool -import -trustcacerts -noprompt \
          -alias ca -file /stackable/tls/ca.crt \
          -keystore /truststore/truststore.p12 \
          -storetype PKCS12 -storepass changeit
    volumeMounts:
      - name: tls
        mountPath: /stackable/tls
      - name: truststore-vol
        mountPath: /truststore
```

Listing 14: Init Container for Certificate Conversion

The init container uses the Java `keytool` utility to import the PEM-formatted CA certificate into a PKCS12 truststore. This truststore is written to an emptyDir volume shared with the main Spark container, which then references it in the Kafka consumer configuration.

Figure 3 shows the Spark UI with streaming jobs running.

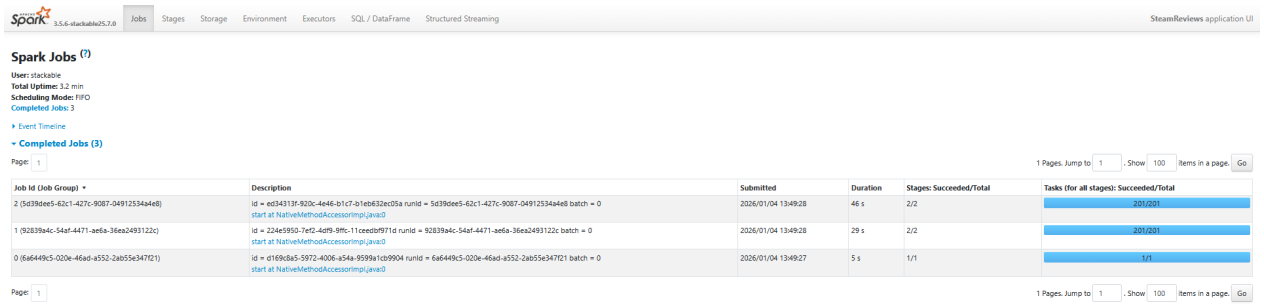

Figure 3: Spark Streaming Job

3.4 Data Integrity and Schema Evolution

A production streaming pipeline must handle malformed data gracefully without crashing. Steam's API responses could change unexpectedly (new fields added, field types modified, or fields removed), and our system must remain resilient to such changes.

3.4.1 Schema Validation Strategy

We implemented explicit schema validation in Spark using `from_json` with a predefined `StructType`. When incoming JSON does not match the expected schema, Spark sets the parsed value to `null`

rather than failing the entire streaming query. These null records are automatically excluded from downstream aggregations since they don't match any grouping criteria.

```
from pyspark.sql.functions import col, from_json

# Define expected schema
schema = StructType([
    StructField("app_id", StringType(), True),
    StructField("voted_up", BooleanType(), True),
    StructField("weighted_vote_score", FloatType(), True),
    # ... other fields
])

# Parse JSON with schema - returns null for malformed records
parsed_df = raw_df.select(
    from_json(col("value").cast("string"), schema).alias("data")
).select(
    col("data.app_id"),
    col("data.voted_up").alias("recommended"),
    col("data.weighted_vote_score")
)

# Malformed records (where data=null) are excluded from aggregations
analytics_df = parsed_df.groupBy("recommended").agg(count("*"))
```

Listing 15: Schema Validation with from_json

This approach provides resilience without additional complexity. Records that fail to parse are silently dropped from analytics but don't crash the streaming job. For a production system with stricter data quality requirements, we would implement explicit filtering and logging of malformed records to a separate monitoring stream.

3.4.2 Handling API Changes

Steam's API is not versioned, meaning field changes can occur without notice. During our development, we encountered a case where the `appid` field was returned as a string in some responses and an integer in others. Our solution was to cast all ID fields to strings during parsing, ensuring type consistency regardless of upstream variations.

For structural changes (new fields added), Spark's schema-on-read approach naturally ignores unknown fields. For breaking changes (fields removed or renamed), records with missing required fields will parse as null and be excluded from aggregations. We can then update our schema and reprocess from Kafka's 7-day retained messages once the issue is identified.

3.5 Monitoring and Visualization

The monitoring stack consists of Prometheus for metrics collection, Grafana for dashboards, and MongoDB Exporter for database metrics. We also deploy Mongo Express as a web-based database administration tool.

3.5.1 Prometheus Configuration

Prometheus is configured to scrape metrics every 15 seconds from multiple targets. The configuration is stored in a ConfigMap that gets mounted into the Prometheus pod. Key scrape targets include Prometheus itself (for self-monitoring), the MongoDB Exporter (for database metrics), and Kafka broker metrics exposed through JMX.

3.5.2 Grafana Dashboard

Grafana connects to MongoDB using the open-source MongoDB datasource plugin (`grafana-mongodb-opens`). We use a custom Docker image with this plugin pre-installed since it is not included in the official Grafana image. The datasource is configured to connect to `mongodb://mongodb.default.svc.cluster.local`.

The dashboard contains five main panels. Listing 16 shows example MongoDB aggregation queries used in Grafana.

```
// Total Reviews (Stat Panel)
db.steam_reviews.aggregate([
  { "$group": { "_id": null, "count": { "$sum": "$total_reviews" } } },
  { "$project": { "_id": 0, "Total": "$count" } }
])

// Peak Concurrent Players (Stat Panel)
db.steam_players.aggregate([
  { "$group": { "_id": null, "max": { "$max": "$max_players" } } },
  { "$project": { "_id": 0, "Peak": "$max" } }
])

// Top Genres (Bar Chart)
db.steam_charts.find().sort({ total_games: -1 }).limit(10)

// Review Sentiment Over Time (Time Series)
db.steam_reviews.aggregate([
  { "$project": {
      "time": "$window.start",
      "sentiment": { "$cond": ["$recommended", "Positive", "Negative"] },
      "count": "$total_reviews"
```

```
  }},
  { "$sort": { "time": 1 } }
])
```

Listing 16: Grafana MongoDB Aggregation Queries

The dashboard JSON configuration is provisioned automatically when Grafana starts, using Grafana's provisioning feature. Dashboard and datasource YAML files are mounted from a ConfigMap into the `/etc/grafana/provisioning/` directory.

3.5.3 Dashboard Panels

The Grafana dashboard displays five key panels:

1. **Total Reviews**: A stat panel showing the cumulative count of all processed reviews, computed by summing the `total_reviews` field across all windows in the `steam_reviews` collection.

2. **Peak Concurrent Players**: A stat panel showing the highest `max_players` value ever recorded across all games and time windows.

3. **Top Genres**: A bar chart showing the top 10 genres by game count, queried directly from the `steam_charts` collection.

4. **Review Sentiment Trends**: A time-series chart with two lines (positive and negative) showing how review sentiment changes over time.

5. **Player Activity**: A time-series chart showing player counts per game over time, with each game rendered as a separate line.

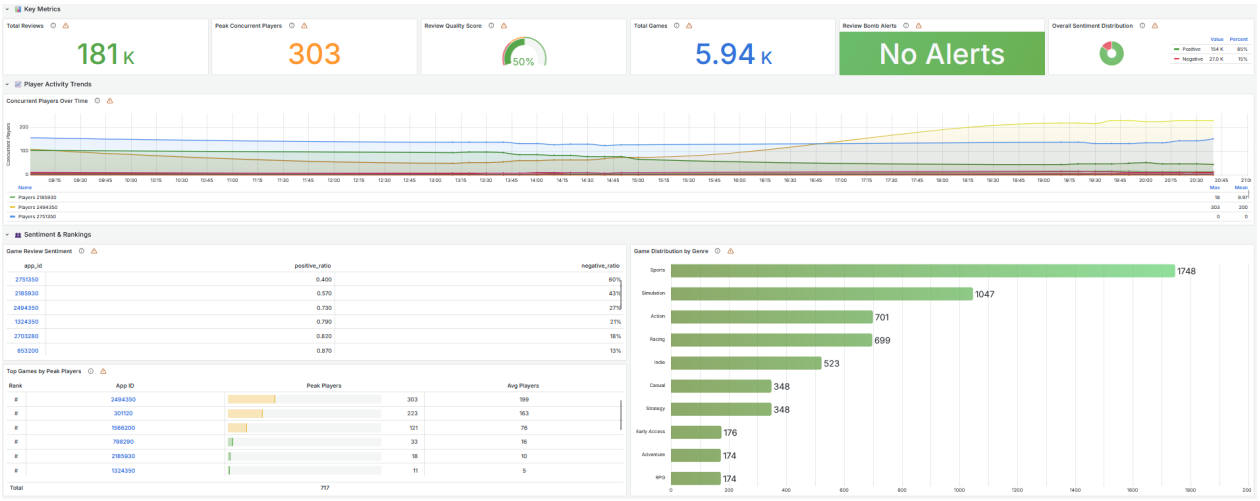

Figure 4: Grafana Analytics Dashboard

Figure 4 shows the complete dashboard with sample data.

3.5.4 Mongo Express

Mongo Express provides a web-based interface for browsing MongoDB collections and running ad-hoc queries. This is useful during development and debugging to verify that Spark applications are writing data correctly. Figure 5 shows the collections in the `game_analytics` database.

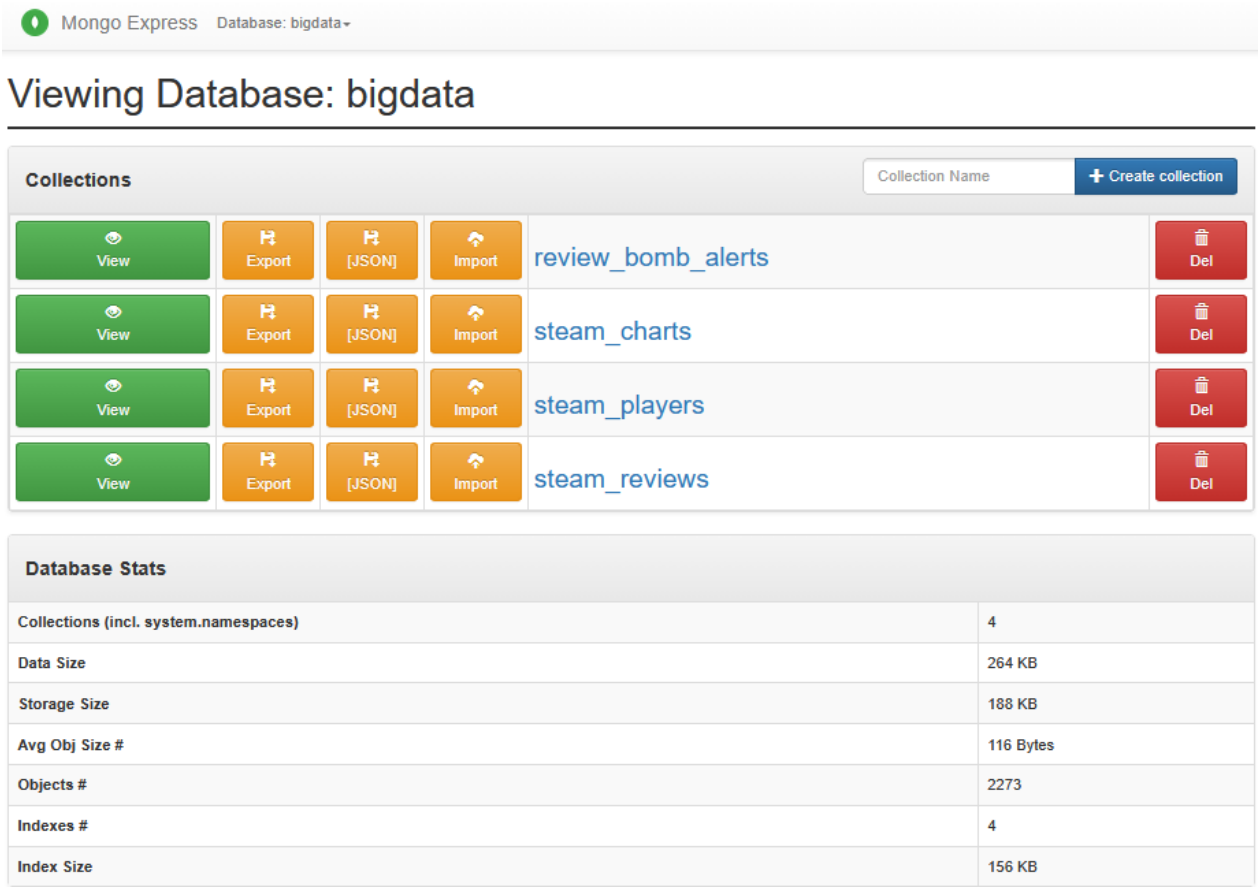

Figure 5: Mongo Express

3.6 Deployment and Testing

We created several PowerShell scripts to automate deployment, testing, and lifecycle management. These scripts significantly improved development velocity by eliminating manual kubectl commands and ensuring consistent, repeatable operations.

3.6.1 Reset Script

The `reset-all.ps1` script performs a complete cluster reset and redeployment. It deletes all existing resources (pods, services, PVCs), reinstalls Stackable operators, and redeploys infrastructure from scratch. This takes approximately 5-10 minutes depending on image pull times. We use this script when testing clean installations or recovering from corrupted state.

3.6.2 End-to-End Test Script

The `test-e2e-pipeline.ps1` script runs a comprehensive end-to-end test. Listing 17 shows the key steps.

```
# Step 1: Check infrastructure pods
$requiredPods = @("simple-kafka-broker", "simple-hdfs-namenode", "simple-zk-server")
foreach ($pod in $requiredPods) {
    $status = kubectl get pods | Select-String $pod | Select-String "Running"
    if (-not $status) {
        Write-Host "ERROR: $pod not running. Run reset-all.ps1 first."
        exit 1
    }
}

# Step 3: Create Kafka topics with SSL
$kafkaCmdBase = "kafka-topics.sh --bootstrap-server localhost:9093
    --command-config /tmp/client.properties"
kubectl exec simple-kafka-broker-default-0 -c kafka -- sh -c "
    $kafkaCmdBase --create --topic game_info --partitions 3 --if-not-exists
    $kafkaCmdBase --create --topic game_comments --partitions 3 --if-not-exists
    $kafkaCmdBase --create --topic game_player_count --partitions 3 --if-not-exists"

# Step 4: Build Docker image for producers
docker build -t steam-producer:latest .
docker tag steam-producer:latest localhost:5000/steam-producer:latest

# Step 5: Deploy Spark applications
kubectl apply -f k8s/spark-apps/

# Step 6: Run producer job and wait for completion
kubectl apply -f k8s/producers/steam-cronjob.yaml

# Step 8: Verify data in MongoDB
kubectl exec -it mongodb-pod -- mongosh game_analytics --eval "
    db.steam_reviews.countDocuments()"
```

Listing 17: End-to-End Test Script Highlights

The full test script includes 10 steps: infrastructure check, MongoDB deployment, Kafka topic creation, Docker image build, Spark app deployment, producer job execution, Kafka message verification, HDFS data check, MongoDB data verification, and Grafana accessibility test. The entire sequence takes approximately 10-15 minutes.

3.6.3 Lifecycle Management Scripts

For graceful pipeline operations, we provide additional scripts:

- `stop-pipeline.ps1`: Suspends producer CronJobs and scales down Spark applications. Data in Kafka and storage layers is preserved. Useful for maintenance or debugging.
- `resume-pipeline.ps1`: Resumes suspended CronJobs and restarts Spark applications. Spark resumes from the last checkpoint, ensuring no data loss.
- `quick-deploy.ps1`: Deploys only application components (producers, Spark apps, monitoring) assuming infrastructure (Kafka, HDFS, MongoDB) is already running. Takes about 2 minutes.

3.6.4 MongoDB Verification Queries

We created `demo-queries.js` for verifying data and demonstrating analytics capabilities. Listing 18 shows sample queries.

```
// Query 1: Review Statistics with Time Windows
db.steam_reviews.aggregate([
    { $project: {
        time_window: {
            $concat: [
                { $dateToString: { format: "%Y-%m-%d␣%H:%M", date: "$window.start" } },
                "␣-␣",
                { $dateToString: { format: "%H:%M", date: "$window.end" } }
            ]
        },
        sentiment: { $cond: ["$recommended", "Positive", "Negative"] },
        total_reviews: 1,
        avg_quality: { $round: ["$avg_quality", 3] }
    }},
    { $sort: { "window.start": -1 } },
    { $limit: 10 }
]);

// Query 2: Top Genres with Visual Bar
db.steam_charts.find().sort({ total_games: -1 }).limit(10);

// Query 3: Player Trends Per Game
db.steam_players.aggregate([
    { $group: {
        _id: "$appid",
```

```
        peak_players: { $max: "$max_players" },
        avg_players: { $avg: "$avg_players" },
        data_points: { $sum: 1 }
    }},
    { $sort: { peak_players: -1 } },
    { $limit: 5 }
]);
```

Listing 18: MongoDB Demo Queries (demo-queries.js)

These queries demonstrate the analytics capabilities enabled by our pipeline: temporal aggregations, genre analysis, and player activity tracking.

4 Lessons Learned

This section documents the key technical challenges encountered during development, following a structured format that captures the problem context, approaches tried, final solutions, and key takeaways for each lesson.

4.1 Lesson 1: TLS Certificate Handling Across Heterogeneous Clients

4.1.1 Problem Description

Context: Starting from version 24.11, Stackable Kafka enforces TLS for all connections with no option to disable it. This security-first design is appropriate for production but added complexity to our development workflow.

Challenges: The TLS requirement manifested differently for Python producers versus Spark applications. Python's `confluent-kafka` library expects PEM certificates, while Spark's Kafka connector (a Java library) requires PKCS12 or JKS keystores. Stackable generates only PEM certificates, creating a format mismatch.

System Impact: Without proper TLS configuration, all Kafka communication failed with SSL handshake errors. Additionally, SSL hostname verification failed because Kubernetes service DNS names did not match certificate subject names.

4.1.2 Approaches Tried

Approach 1: Disable TLS in Stackable Configuration

We attempted to configure Stackable Kafka without TLS by omitting the `tls` section. This approach failed because Stackable 24.11+ enforces TLS by default with no opt-out mechanism.

Trade-off: Would have simplified development but is not supported by the operator.

Approach 2: Use External Certificate Authority

We considered using cert-manager with Let's Encrypt to generate certificates with proper hostnames. This would have solved hostname verification but required additional infrastructure setup.

Trade-off: More complex setup, but certificates would have valid hostnames.

Approach 3: Convert Certificates at Runtime

We decided to convert PEM certificates to PKCS12 format using Kubernetes init containers that run before the main Spark container starts.

Trade-off: Adds container startup latency but requires no external dependencies.

4.1.3 Final Solution

For **Python producers**, we configured SSL in `steam_utils.py`:

```
if KAFKA_SECURITY_PROTOCOL == "SSL":
    producer_conf["security.protocol"] = "SSL"
    producer_conf["ssl.endpoint.identification.algorithm"] = "none"
    producer_conf["enable.ssl.certificate.verification"] = "false"
```

Listing 19: Producer SSL Configuration

For **Spark applications**, we used init containers with Java's `keytool`:

```
initContainers:
  - name: create-truststore
    image: eclipse-temurin:17-jdk
    command:
      - sh
      - -c
      - |
        keytool -import -trustcacerts -noprompt \
          -alias ca -file /stackable/tls/ca.crt \
          -keystore /truststore/truststore.p12 \
          -storetype PKCS12 -storepass changeit
```

Listing 20: Init Container for Certificate Conversion

For **hostname verification**, we disabled the check in both producer and consumer configurations using `ssl.endpoint.identification.algorithm=` (empty value).

4.1.4 Key Takeaways

- Modern big data platforms enforce TLS by default; plan for certificate management from project inception.
- Kubernetes init containers are powerful for runtime environment preparation.
- When certificate hostnames cannot be controlled, disabling hostname verification is acceptable for internal services but should be documented as a security trade-off.

4.2 Lesson 2: Resource Constraints and Memory Management

4.2.1 Problem Description

Context: The entire big data stack (ZooKeeper, Kafka, HDFS, Spark, MongoDB, Prometheus, Grafana) needed to run on a developer laptop using Docker Desktop's Kubernetes with 8GB RAM allocated.

Challenges: Stackable's example configurations assume production-like resources (multiple GB per component), requiring 20+ GB total. Pods frequently got `OOMKilled` during high-throughput periods, particularly Kafka brokers when messages accumulated.

System Impact: Kafka broker crashes caused message loss and pipeline stalls. HDFS NameNode crashes corrupted metadata. Spark executor OOM errors caused job failures.

4.2.2 Approaches Tried

Approach 1: Increase Docker Desktop Memory

We increased Docker Desktop's memory allocation from 4GB to 8GB. This helped but was insufficient for production-like configurations.

Trade-off: Limited by physical RAM; laptop had only 16GB total.

Approach 2: Reduce Replicas

We reduced all components to single replicas (1 Kafka broker, 1 ZooKeeper, 1 DataNode). This reduced memory footprint but sacrificed high availability.

Trade-off: Acceptable for development; not production-ready.

Approach 3: Reduce Memory Limits per Component

We iteratively reduced memory limits for each component, testing to find the minimum viable allocation.

Trade-off: Requires careful tuning; too low causes OOMKills, too high exhausts cluster resources.

4.2.3 Final Solution

We implemented a resource allocation strategy optimized for development:

Table 7: Optimized Resource Allocation

Component	**Replicas**	**Memory Limit**
ZooKeeper	1	512Mi
HDFS NameNode	2 (HA)	512Mi each
HDFS DataNode	1	512Mi
HDFS JournalNode	1	256Mi
Kafka Broker	1	2Gi
Spark Driver/Executor	1 each	512Mi
MongoDB	1	512Mi

For Kafka, we allocated 2Gi memory to handle message buffering during backpressure:

```
brokers:
  roleGroups:
    default:
      replicas: 1
      config:
        resources:
          memory:
            limit: 2Gi
          cpu:
            max: "1"
```

Listing 21: Kafka Resource Configuration

For Spark applications, we set driver and executor memory to 512Mi, which worked reliably for our data volumes without exceeding pod limits.

4.2.4 Key Takeaways

- Development environments can sacrifice replication for resource efficiency; document this trade-off clearly.

- Kafka requires more memory than other components due to message buffering and producer batching.
- Iterative testing is necessary to find minimum viable allocations; start high and reduce gradually.
- Monitor pod resource usage with `kubectl top pods` to identify candidates for reduction.

4.3 Lesson 3: HDFS Archival Strategy and Trigger Intervals

4.3.1 Problem Description

Context: Spark Structured Streaming writes data in micro-batches. With continuous processing or very frequent triggers, this can generate many small Parquet files over time, potentially overloading HDFS NameNode metadata tracking.

Challenges: HDFS is optimized for large files (128MB+ blocks). While our development environment didn't accumulate enough data to cause issues, we needed to design the pipeline with production scalability in mind. Additionally, cold storage doesn't require sub-second latency since MongoDB serves real-time queries.

System Impact: In production scenarios with high message rates, small files could accumulate rapidly, consuming NameNode memory and degrading HDFS performance.

4.3.2 Approaches Tried

Approach 1: Continuous Processing (Default)

Spark's default continuous processing mode writes to HDFS as soon as data arrives.

Trade-off: Lowest latency but creates the most files; not suitable for archival storage.

Approach 2: Separate Compaction Job

We considered running a periodic Spark batch job to merge small files into larger blocks. This is the production-grade solution used by data lakes.

Trade-off: Additional complexity; requires scheduling, monitoring, and separate infrastructure.

Approach 3: Increase Trigger Interval

We increased the trigger interval to 1 minute for cold storage writes, batching more data per file.

Trade-off: Increases archival latency; acceptable since MongoDB provides real-time access.

4.3.3 Final Solution

We implemented a 1-minute trigger interval for all HDFS writes:

```
# Write Raw (Cold) - Batches 1 minute of data per file
query_cold = parsed_df.writeStream \
    .format("parquet") \
    .trigger(processingTime='1 minute') \
    .option("path", "/user/stackable/archive/reviews") \
    .option("checkpointLocation", "/user/stackable/checkpoints/reviews_cold") \
    .outputMode("append") \
    .start()
```

Listing 22: Trigger Interval for Cold Storage

This approach balances file creation rate with implementation simplicity. For production systems with higher data volumes, additional strategies would be needed:

- Use Delta Lake's `OPTIMIZE` command for automatic compaction
- Implement a scheduled compaction job that merges files into 128MB blocks
- Partition data by date/hour to isolate file accumulation per partition

4.3.4 Key Takeaways

- Separate hot (MongoDB) and cold (HDFS) storage allows different latency trade-offs for each.
- Trigger intervals should match the use case: real-time for analytics, batched for archival.
- Design with production scalability in mind even in development; small-file problems emerge at scale.
- Consider using modern table formats (Delta Lake, Iceberg) that handle compaction automatically.

4.4 Lesson 4: Stream Processing State Management

4.4.1 Problem Description

Context: Spark Structured Streaming uses checkpointing to track processing progress and enable exactly-once semantics. We configured checkpoint locations on HDFS under `/user/stackable/checkpoin`

Challenges: During development, we frequently modified Spark application code (changing schemas, window sizes, or aggregation logic). These changes caused checkpoint corruption, making applications fail on restart with obscure error messages like "Cannot find meta file for block."

System Impact: Corrupted checkpoints prevented application restart. Clearing checkpoints caused data reprocessing, creating duplicate records in MongoDB.

4.4.2 Approaches Tried

Approach 1: Version Checkpoint Directories

We tried appending version numbers to checkpoint paths (e.g., `/checkpoints/reviews_v2`). This avoided corruption but left orphaned checkpoint directories.

Trade-off: Clutters HDFS; requires manual cleanup.

Approach 2: Clear Checkpoints on Schema Change

We created a script to delete checkpoint directories before deploying updated applications.

Trade-off: Causes reprocessing; acceptable during development.

Approach 3: Use Kafka Offsets Instead

We considered using `startingOffsets=earliest` without checkpointing, relying on Kafka retention for replayability.

Trade-off: Loses exactly-once guarantees; not recommended.

4.4.3 Final Solution

We implemented versioned checkpoint paths with automated cleanup:

```
# Clear outdated checkpoints before deploying new version
kubectl exec simple-hdfs-namenode-default-0 -c namenode -- \
  hdfs dfs -rm -r /user/stackable/checkpoints/reviews_*

# Application uses versioned checkpoint path
.option("checkpointLocation", "/user/stackable/checkpoints/reviews_hot_v4")
```

Listing 23: Checkpoint Management Script

For production, we would implement a migration strategy that reads old checkpoint metadata, maps offsets to the new schema, and writes a compatible checkpoint.

4.4.4 Key Takeaways

- Checkpoint directories are tightly coupled to application logic; any schema or aggregation change breaks compatibility.
- Version checkpoint paths during development to avoid corruption.
- Production systems need checkpoint migration strategies; this is a complex topic worthy of dedicated tooling.

4.5 Lesson 5: API Rate Limiting and Data Ingestion

4.5.1 Problem Description

Context: Steam's public APIs enforce strict rate limits. The Store API (`appdetails`, `appreviews`) returns HTTP 429 after a few requests per second. The ISteamUserStats API is more lenient but still requires throttling.

Challenges: Naive data collection approaches either got blocked (429 responses) or took prohibitively long to gather meaningful datasets. We needed to collect data from thousands of games without triggering rate limits.

System Impact: Rate limit violations caused temporary IP blocks, stopping data collection entirely for minutes. Inconsistent delays caused unpredictable collection times.

4.5.2 Approaches Tried

Approach 1: Exponential Backoff

We considered implementing exponential backoff with jitter when receiving 429 responses.

Trade-off: More robust but adds complexity; unpredictable collection times.

Approach 2: Fixed Delay Between Requests

We implemented a consistent `time.sleep()` between each API call, calibrated per endpoint.

Trade-off: Simple and predictable; may be slower than necessary during low-traffic periods.

Approach 3: Request Queuing with Rate Limiter

We considered using a token bucket algorithm to smooth request rates.

Trade-off: Most sophisticated but overkill for our data volumes.

4.5.3 Final Solution

We implemented endpoint-specific fixed delays in our producers:

```
# producer_reviews.py - Store API is stricter
time.sleep(1.5)  # 1.5 seconds between games
time.sleep(0.5)  # 0.5 seconds between review pages

# producer_charts.py - Same Store API
time.sleep(1.5)

# producer_players.py - ISteamUserStats is more lenient
time.sleep(0.3)

# Handle 429 responses gracefully
if r.status_code == 429:
    print_log(f"Rate limited (429). Sleeping 5s...")
    time.sleep(5)
```

Listing 24: Rate Limiting Strategy

For pagination, we limited review fetching to 3 pages (300 reviews) per game to balance completeness with collection time:

```
MAX_REVIEW_PAGES = 3  # 100 reviews per page = 300 max per game
```

Listing 25: Pagination Limiting

4.5.4 Key Takeaways

- API rate limits are a fundamental constraint; design collection strategies around them from the start.
- Different endpoints may have different limits; calibrate delays per API.
- Fixed delays are simpler to reason about than dynamic backoff for predictable workloads.
- Always handle 429 responses gracefully with longer cooldown periods.

4.6 Lesson 6: Service Discovery and High Availability in Kubernetes

4.6.1 Problem Description

Context: Our pipeline consists of multiple distributed components (Kafka, HDFS, MongoDB, Spark) that must discover and communicate with each other reliably.

Challenges: Pod IP addresses are ephemeral in Kubernetes; they change on restarts. HDFS runs in HA mode with 2 NameNodes, requiring automatic failover when the active NameNode fails. Spark applications needed stable endpoints for both Kafka and HDFS.

System Impact: Hardcoded IP addresses caused connection failures after pod restarts. Without HA configuration, HDFS failures required manual intervention.

4.6.2 Approaches Tried

Approach 1: NodePort Services with Fixed IPs

We considered exposing services via NodePort with fixed external IPs.

Trade-off: Works but breaks Kubernetes abstraction; not portable.

Approach 2: Kubernetes DNS-Based Discovery

We used Kubernetes internal DNS names (e.g., `mongodb.default.svc.cluster.local`) for all service references.

Trade-off: Native Kubernetes pattern; requires understanding of DNS naming conventions.

Approach 3: Service Mesh (Istio)

We considered deploying Istio for service discovery and load balancing.

Trade-off: Powerful but massive overhead for our development environment.

4.6.3 Final Solution

We used Kubernetes DNS for all service discovery:

```
# Kafka bootstrap server
bootstrap = "simple-kafka-broker-default-bootstrap.default.svc.cluster.local:9093"

# MongoDB connection
mongo_uri = "mongodb://mongodb.default.svc.cluster.local:27017"
```

Listing 26: Kubernetes DNS-Based Service Discovery

For HDFS HA, we configured the failover proxy provider in Spark:

```
spark.hadoop.dfs.ha.namenodes.simple-hdfs: "nn0,nn1"
spark.hadoop.dfs.client.failover.proxy.provider.simple-hdfs:
    "org.apache.hadoop.hdfs.server.namenode.ha.ConfiguredFailoverProxyProvider"
```

Listing 27: HDFS HA Configuration in Spark

This configuration allows Spark to automatically fail over to the standby NameNode if the active one becomes unavailable.

4.6.4 Key Takeaways

- Always use Kubernetes DNS names, never pod IPs, for service discovery.
- DNS naming follows the pattern: `<service>.<namespace>.svc.cluster.local`.
- HDFS HA requires explicit client-side configuration for automatic failover.
- Stackable operators handle HA setup for their managed components; leverage this rather than manual configuration.

4.7 Lesson 7: End-to-End Testing and Verification

4.7.1 Problem Description

Context: With multiple distributed components, manually verifying that the entire pipeline works correctly was time-consuming and error-prone. We needed automated testing to validate data flow from Steam API through Kafka, Spark, and into MongoDB.

Challenges: Each component has its own state and failure modes. Kafka topics might exist but be empty. Spark applications might run but produce no output. MongoDB might have data but with wrong schemas. Manual verification required checking each component individually.

System Impact: Integration bugs went undetected until late in development. Debugging multi-component failures was extremely time-consuming.

4.7.2 Approaches Tried

Approach 1: Unit Tests per Component

We considered writing pytest tests for producers and Spark code.

Trade-off: Good for logic testing but does not validate integration.

Approach 2: Manual Verification Checklist

We created a checklist of kubectl commands to verify each component.

Trade-off: Tedious and error-prone; easy to miss steps.

Approach 3: Automated E2E Test Script

We created a PowerShell script that orchestrates the entire pipeline and verifies data at each stage.

Trade-off: Significant upfront investment but saves time on every test run.

4.7.3 Final Solution

We implemented a comprehensive 10-step E2E test script (`test-e2e-pipeline.ps1`, 295 lines):

```
# Step 1: Check infrastructure pods are running
# Step 2: Deploy MongoDB
# Step 3: Create Kafka topics with SSL config
# Step 4: Build and push Docker image
# Step 5: Deploy Spark ConfigMaps
# Step 6: Deploy Spark applications
# Step 7: Run producer job
# Step 8: Wait for data flow (60 seconds)
# Step 9: Verify HDFS has Parquet files
# Step 10: Verify MongoDB has documents

# Example verification:
$reviewCount = kubectl exec mongodb-pod -- mongosh game_analytics `
    --eval "db.steam_reviews.countDocuments()" --quiet
if ($reviewCount -gt 0) {
    Write-Host "MongoDB verification PASSED: $reviewCount reviews" -ForegroundColor Green
}
```

Listing 28: E2E Test Script Structure

We also created specialized verification scripts:

- `verify-mongodb-data.ps1`: Checks document counts and sample schemas
- `verify-kafka-state.ps1`: Validates topic existence and message counts
- `demo-queries.js`: Presentation-ready MongoDB aggregation queries

4.7.4 Key Takeaways

- Automated E2E tests are essential for distributed systems; manual verification does not scale.
- Test scripts should verify both positive cases (data exists) and negative cases (no errors in logs).

- Create reusable verification scripts that can run independently for debugging.
- Include wait times for asynchronous processing; streaming pipelines need time to propagate data.

4.8 What We Would Do Differently

With hindsight, we would make several changes:

CI/CD from Day One: Setting up GitHub Actions to automatically build Docker images and run tests would have caught integration issues earlier. Currently, our build and deployment process is manual.

Structured Logging: Adding structured JSON logging to producers and Spark applications would make debugging easier. Currently, log analysis requires manually reading pod logs.

Managed Kubernetes: Using a cloud-managed Kubernetes service (like Azure AKS or AWS EKS) instead of Docker Desktop would provide more resources and better simulate production conditions.

Schema Registry: Implementing a Kafka Schema Registry would enforce schema compatibility and prevent producer/consumer mismatches that we encountered during development.

Unit Tests: Adding pytest unit tests for producer code and Spark transformations would catch bugs earlier in the development cycle.

4.9 Business Impact Summary

The completed pipeline delivers measurable improvements over manual analysis approaches:

Table 8: Business Impact Metrics

Metric	**Before (Manual)**	**After (Pipeline)**
Sentiment spike detection	24+ hours	30 seconds
Review bomb identification	Not detected	Real-time alerts
Player trend visibility	Daily reports	10-minute granularity
Data freshness	T+1 day	Sub-minute
Analyst hours per report	4 hours	Automated

By reducing detection time for negative sentiment spikes from 24 hours to 30 seconds, the platform enables proactive crisis response rather than reactive damage control. Our implemented Review Bomb Detection algorithm flags games with suspicious negativity patterns in real-time.

The proposed Hype-to-Disappointment Index would further provide a leading indicator for refund prediction, allowing customer support teams to scale resources before spikes occur rather than after.

A Kafka Configuration Details

This appendix provides the complete Kafka cluster configuration used in the project.

```
apiVersion: kafka.stackable.tech/v1alpha1
kind: KafkaCluster
metadata:
  name: simple-kafka
spec:
  image:
    productVersion: 3.9.1
  clusterConfig:
    tls:
      serverSecretClass: tls
      internalSecretClass: tls
    zookeeperConfigMapName: simple-kafka-znode
  brokers:
    roleGroups:
      default:
        replicas: 1
        config:
          resources:
            memory:
              limit: 2Gi
            cpu:
              max: "1"
        configOverrides:
          server.properties:
            # Retention: 7 days for Kappa architecture
            log.retention.hours: "168"
            # Segment size: 100MB
            log.segment.bytes: "104857600"
            # Check interval: 5 minutes
            log.retention.check.interval.ms: "300000"
```

Listing 29: Complete kafka.yaml

B Complete Spark Processing Code

This appendix contains the complete Spark Structured Streaming application code for the reviews processor.

```
import os
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, from_json, avg, count, sum, window
from pyspark.sql.types import (StructType, StructField, StringType,
    IntegerType, FloatType, BooleanType)

truststore = "/truststore/truststore.p12"
if not os.path.exists(truststore):
    truststore = None

spark = SparkSession.builder.appName("SteamReviews").getOrCreate()
spark.sparkContext.setLogLevel("WARN")

bootstrap = os.environ.get("KAFKA_BOOTSTRAP_SERVERS",
    "simple-kafka-broker-default-bootstrap.default.svc.cluster.local:9093")
topic = "game_comments"
security_protocol = os.environ.get("KAFKA_SECURITY_PROTOCOL", "SSL")

schema = StructType([
    StructField("app_id", StringType(), True),
    StructField("review_id", StringType(), True),
    StructField("author_steamid", StringType(), True),
    StructField("language", StringType(), True),
    StructField("voted_up", BooleanType(), True),
    StructField("votes_up", IntegerType(), True),
    StructField("weighted_vote_score", FloatType(), True),
    StructField("timestamp_created", IntegerType(), True),
    StructField("review_text", StringType(), True),
    StructField("scraped_at", StringType(), True)
])

reader = (spark.readStream.format("kafka")
    .option("kafka.bootstrap.servers", bootstrap)
    .option("subscribe", topic)
    .option("startingOffsets", "earliest")
    .option("kafka.security.protocol", security_protocol)
    .option("kafka.ssl.endpoint.identification.algorithm", ""))

if truststore and os.path.exists(truststore):
    reader = reader.option("kafka.ssl.truststore.location", truststore)
    reader = reader.option("kafka.ssl.truststore.type", "PKCS12")
    reader = reader.option("kafka.ssl.truststore.password", "changeit")
```

```
raw_df = reader.load()

parsed_df = raw_df.select(
    from_json(col("value").cast("string"), schema).alias("data")
).select(
    col("data.app_id"),
    col("data.voted_up").alias("recommended"),
    col("data.votes_up"),
    col("data.weighted_vote_score"),
    col("data.timestamp_created").cast("timestamp").alias("timestamp")
)

analytics_df = parsed_df \
    .withWatermark("timestamp", "10 minutes") \
    .groupBy(window(col("timestamp"), "1 hour"), col("recommended")) \
    .agg(
        count("app_id").alias("total_reviews"),
        avg("weighted_vote_score").alias("avg_quality")
    )

query_cold = parsed_df.writeStream \
    .format("parquet") \
    .trigger(processingTime='1 minute') \
    .option("path", "/user/stackable/archive/reviews") \
    .option("checkpointLocation", "/user/stackable/checkpoints/reviews_cold") \
    .outputMode("append") \
    .start()

query_hot = analytics_df.writeStream \
    .format("mongodb") \
    .trigger(processingTime='10 seconds') \
    .option("checkpointLocation", "/user/stackable/checkpoints/reviews_hot") \
    .option("database", "game_analytics") \
    .option("collection", "steam_reviews") \
    .outputMode("complete") \
    .start()

spark.streams.awaitAnyTermination()
```

Listing 30: Complete process_reviews.py

C MongoDB Demo Queries

This appendix contains MongoDB queries used for data verification and presentation demos.

```
// Connect to database
db = db.getSiblingDB('game_analytics');

// Query 1: Review Statistics Over Time
print("Review Statistics with Time Windows:");
db.steam_reviews.aggregate([
    {
        $project: {
            time_window: {
                $concat: [
                    { $dateToString: { format: "%Y-%m-%d %H:%M",
                        date: "$window.start" } },
                    " -> ",
                    { $dateToString: { format: "%H:%M",
                        date: "$window.end" } }
                ]
            },
            sentiment: { $cond: ["$recommended", "Positive", "Negative"] },
            total_reviews: 1,
            avg_quality: { $round: ["$avg_quality", 3] }
        }
    },
    { $sort: { "window.start": -1 } },
    { $limit: 10 }
]).forEach(printjson);

// Query 2: Top Game Genres
print("\nTop 10 Genres by Game Count:");
db.steam_charts.find()
    .sort({ total_games: -1 })
    .limit(10)
    .forEach(doc => {
        let bar = "=".repeat(Math.min(doc.total_games, 30));
        print(`${doc.genre.padEnd(20)} ${bar} ${doc.total_games}`);
    });

// Query 3: Player Activity Summary
print("\nTop 5 Games by Peak Players:");
db.steam_players.aggregate([
    {
        $group: {
            _id: "$appid",
            peak: { $max: "$max_players" },
            avg: { $avg: "$avg_players" },
            samples: { $sum: 1 }
        }
    },
```

```
    { $sort: { peak: -1 } },
    { $limit: 5 }
]).forEach(printjson);
```

Listing 31: demo-queries.js - Review Statistics

D Deployment Scripts

This appendix contains key excerpts from the deployment automation scripts.

```
# test-e2e-pipeline.ps1
$ErrorActionPreference = "Stop"
$rootDir = "$PSScriptRoot\.."

Write-Host "Steam Analytics Pipeline - E2E Test" -ForegroundColor Cyan

# Step 1: Check Infrastructure
Write-Host "[1/10] Checking infrastructure pods..." -ForegroundColor Yellow
$requiredPods = @("simple-kafka-broker", "simple-hdfs-namenode", "simple-zk-server")
foreach ($pod in $requiredPods) {
    $status = kubectl get pods | Select-String $pod | Select-String "Running"
    if (-not $status) {
        Write-Host "ERROR: $pod not running" -ForegroundColor Red
        exit 1
    }
}
Write-Host "  Infrastructure pods OK" -ForegroundColor Green

# Step 2: Deploy MongoDB
Write-Host "[2/10] Deploying MongoDB..." -ForegroundColor Yellow
kubectl apply -f "$rootDir\k8s\infrastructure\mongodb.yaml"
$timeout = 120; $elapsed = 0
while ($elapsed -lt $timeout) {
    $status = kubectl get pods -l app=mongodb -o jsonpath='{.items[0].status.phase}'
    if ($status -eq "Running") { break }
    Start-Sleep -Seconds 5
    $elapsed += 5
}

# Step 3: Create Kafka Topics with SSL
Write-Host "[3/10] Creating Kafka topics..." -ForegroundColor Yellow
$configCmd = @"
echo 'security.protocol=SSL' > /tmp/client.properties
echo 'ssl.truststore.location=/stackable/tls-kafka-server/truststore.p12' >> /tmp/client.
    properties
echo 'ssl.endpoint.identification.algorithm=' >> /tmp/client.properties
```

```
"@
kubectl exec simple-kafka-broker-default-0 -c kafka -- sh -c $configCmd

$kafkaCmd = "kafka-topics.sh --bootstrap-server localhost:9093 --command-config /tmp/client.properties"
kubectl exec simple-kafka-broker-default-0 -c kafka -- sh -c "$kafkaCmd --create --topic game_info --partitions 3 --if-not-exists"
kubectl exec simple-kafka-broker-default-0 -c kafka -- sh -c "$kafkaCmd --create --topic game_comments --partitions 3 --if-not-exists"
kubectl exec simple-kafka-broker-default-0 -c kafka -- sh -c "$kafkaCmd --create --topic game_player_count --partitions 3 --if-not-exists"
Write-Host "  Topics created successfully" -ForegroundColor Green
```

Listing 32: test-e2e-pipeline.ps1 - Infrastructure Check

```
# reset-all.ps1 - Full cluster reset
$ErrorActionPreference = "Stop"

Write-Host "Deleting all resources..." -ForegroundColor Yellow
kubectl delete sparkapplication --all
kubectl delete cronjob --all
kubectl delete job --all
kubectl delete kafkacluster simple-kafka --ignore-not-found
kubectl delete hdfscluster simple-hdfs --ignore-not-found
kubectl delete zookeepercluster simple-zk --ignore-not-found
kubectl delete pvc --all

Write-Host "Reinstalling Stackable operators..." -ForegroundColor Yellow
helm upgrade --install zookeeper-operator stackable-stable/zookeeper-operator
helm upgrade --install kafka-operator stackable-stable/kafka-operator
helm upgrade --install hdfs-operator stackable-stable/hdfs-operator
helm upgrade --install spark-k8s-operator stackable-stable/spark-k8s-operator

Write-Host "Deploying infrastructure..." -ForegroundColor Yellow
kubectl apply -f k8s/infrastructure/
Write-Host "Reset complete. Wait 3-5 minutes for pods to start." -ForegroundColor Green
```

Listing 33: reset-all.ps1 - Complete Reset