Streaming data analytics for web application

Group 4

October 01, 2025

Contents

Streaming data analytics for web application

Group Information

Name	Bits ID
Balaji O M	2024mt03025
Balasubramaniyan	2024mt03053

Name	Bits ID
Deep Pokala	2024mt03042
Jagriti Sharma	2024mt03116

Abstract

This project implements a comprehensive real-time data streaming pipeline using Apache Flume, Apache Kafka, and Python-based producer and consumer applications. The system demonstrates end-to-end event processing capabilities where a Python producer generates synthetic JSON events, writes them to a log file, which is then tailed by Apache Flume and forwarded to Apache Kafka. A Python consumer subscribes to the Kafka topic and performs real-time analytics on the streaming data.

The architecture showcases modern streaming data processing patterns, containerization with Docker, and resilient service integration with proper error handling and retry mechanisms. The implementation serves as a foundation for understanding distributed streaming systems and can be extended for production-scale data processing scenarios.

Architecture Diagram

Figure 1: End-to-end streaming data pipeline architecture showing data flow from producer through Flume to Kafka and consumer processing.

Implementation Details

1. System Components

1.1 Python Producer Service

- **Technology**: Python 3.11 with Faker library
- **Functionality**: Generates synthetic e-commerce events with product information, prices, and timestamps
- **Output**: JSON-formatted events written to `/data/logs/input.log`
- **Configuration**: Configurable production rate via `PRODUCE_RATE_PER_SEC` environment variable

1.2 Apache Flume Agent

- **Version**: Apache Flume 1.9.0
- **Source**: Exec source with `tail -F` command to monitor log file
- **Channel**: Memory channel for high-throughput processing
- **Sink**: Kafka sink configured to publish to `events` topic
- **Configuration**: Custom Docker image built from OpenJDK 8 base

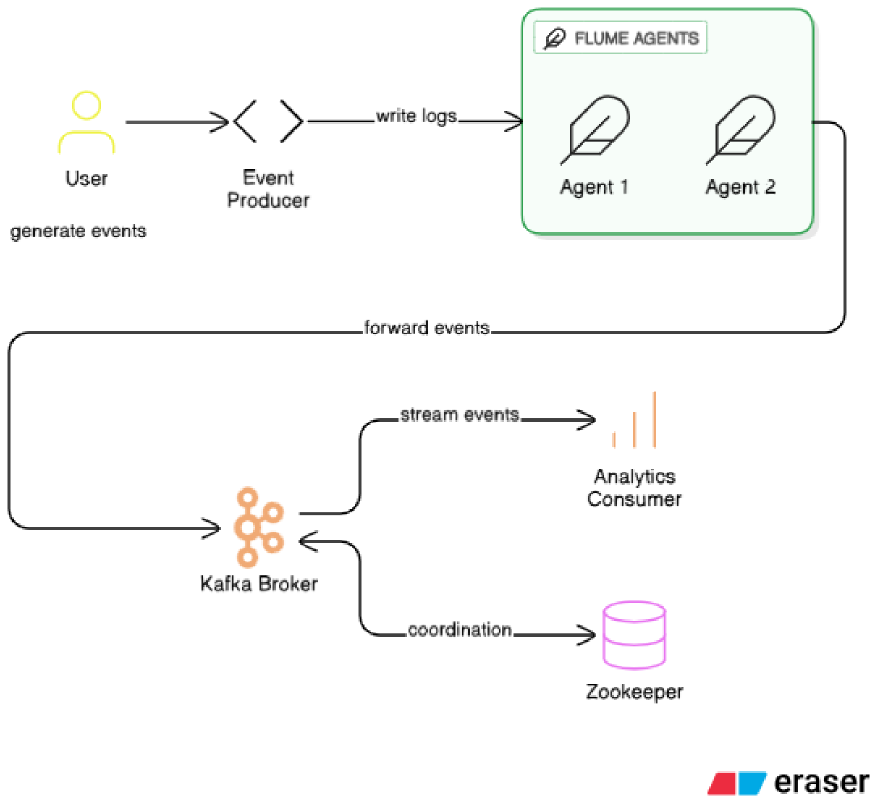


Figure 1: Streaming Architecture

1.3 Apache Kafka Cluster

- **Version**: Apache Kafka 3.6 with Zookeeper 3.9
- **Topic**: `events` (auto-created with single partition)
- **Configuration**: PLAINTEXT protocol for internal communication
- **Listeners**: Configured for both internal container and external access

1.4 Python Consumer Service

- **Technology**: Python 3.11 with kafka-python library
- **Functionality**: Subscribes to Kafka topic and performs real-time analytics
- **Processing**: Price categorization (low/medium/high buckets)
- **Resilience**: Connection retry logic with configurable backoff

2. Containerization Strategy

2.1 Docker Compose Orchestration

- **Services**: 5 containerized services (Zookeeper, Kafka, Flume, Producer, Consumer)
- **Networking**: Custom Docker network for service discovery
- **Volumes**: Shared volume for log file access between producer and Flume
- **Dependencies**: Proper service startup ordering with `depends_on`

2.2 Custom Flume Image

- **Base**: OpenJDK 8 JRE slim
- **Installation**: Automated Flume 1.9.0 download and setup
- **Security**: Non-root user execution with proper permissions
- **Configuration**: Embedded flume.conf for Kafka integration

3. Data Flow Architecture

1. **Event Generation**: Producer creates JSON events with product data
2. **File Persistence**: Events written to shared log file via bind mount
3. **Event Ingestion**: Flume tails the file and buffers events in memory channel
4. **Message Publishing**: Flume publishes events to Kafka `events` topic
5. **Event Consumption**: Consumer subscribes and processes events in real-time
6. **Analytics**: Price-based categorization and event metadata extraction

3.1 Sequence diagram *Figure 2: Diagram showing data flow from producer through Flume to Kafka and consumer processing.*

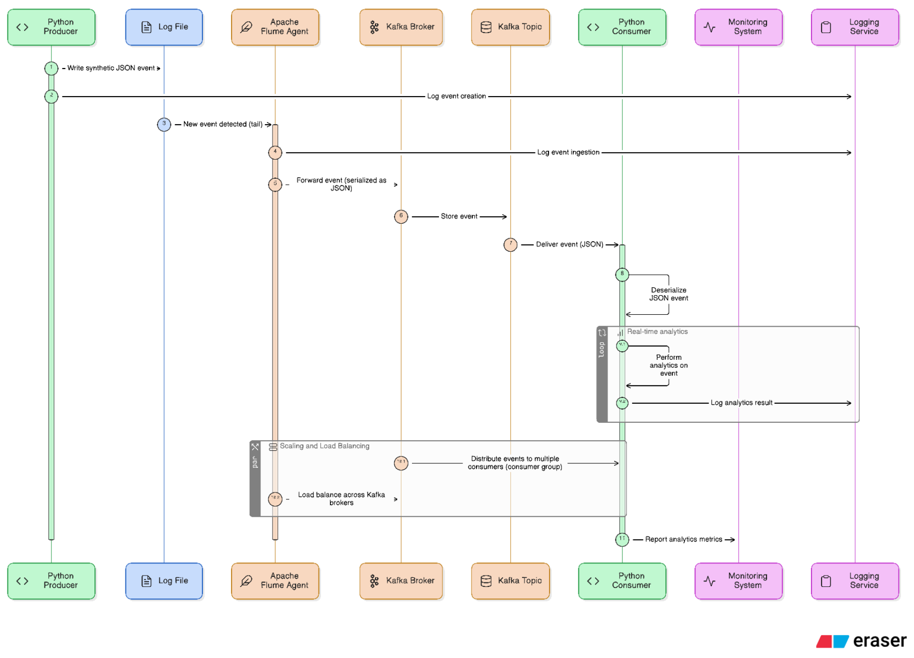


Figure 2: Data Flow

4. Error Handling and Resilience

- **Consumer Retry Logic**: Configurable connection attempts with exponential backoff
- **Service Dependencies**: Proper startup ordering to prevent connection failures
- **Health Monitoring**: Container health checks and logging integration
- **Graceful Shutdown**: Proper resource cleanup on service termination

Scripts/Commands for Streaming Integration

1. Environment Setup

```bash
# Clone repository and navigate to project directory
git clone <repository-url>
cd stream-processing-assignment-1

# Create necessary directories
mkdir -p data/logs
touch data/logs/input.log
```

2. Service Management

```bash
# Start all services
make up
# or
docker compose up -d --build

# Stop all services
make down
# or
docker compose down -v

# View service logs
make logs
# or
docker compose logs -f --tail=200

# Check service status
make ps
# or
docker compose ps
```

3. Data Management

```bash
# Reset log file (clean slate)
make clean
```

```
# Create Kafka topic manually (if needed)
make topic
# or
docker exec -it kafka kafka-topics.sh --bootstrap-server localhost:9092 \
  --create --if-not-exists --topic events --replication-factor 1 --partitions 1
```

4. Monitoring and Debugging

```
# List Kafka topics
docker exec -it kafka kafka-topics.sh --bootstrap-server localhost:9092 --list

# Consume messages directly from Kafka
docker exec -it kafka kafka-console-consumer.sh \
  --bootstrap-server localhost:9092 --topic events --from-beginning

# Check log file contents
tail -f data/logs/input.log

# Monitor specific service logs
docker compose logs -f producer
docker compose logs -f flume
docker compose logs -f consumer
```

5. Integration with External Platforms

5.1 Kafka Connect Integration

```
# Example: Connect to external database
# Create connector configuration
cat > kafka-connect-jdbc.json << EOF
{
  "name": "jdbc-sink-connector",
  "config": {
    "connector.class": "io.confluent.connect.jdbc.JdbcSinkConnector",
    "connection.url": "jdbc:postgresql://postgres:5432/streaming_db",
    "topics": "events",
    "auto.create": "true",
    "key.converter": "org.apache.kafka.connect.json.JsonConverter",
    "value.converter": "org.apache.kafka.connect.json.JsonConverter"
  }
}
EOF

# Deploy connector
curl -X POST -H "Content-Type: application/json" \
  --data @kafka-connect-jdbc.json \
  http://localhost:8083/connectors
```

5.2 Elasticsearch Integration

```
# Add Elasticsearch to docker-compose.yml
# Example configuration for logstash pipeline
input {
  kafka {
    bootstrap_servers => "kafka:9092"
    topics => ["events"]
    codec => "json"
  }
}

output {
  elasticsearch {
    hosts => ["elasticsearch:9200"]
    index => "streaming-events-%{+YYYY.MM.dd}"
  }
}
```

5.3 Real-time Analytics with Apache Spark

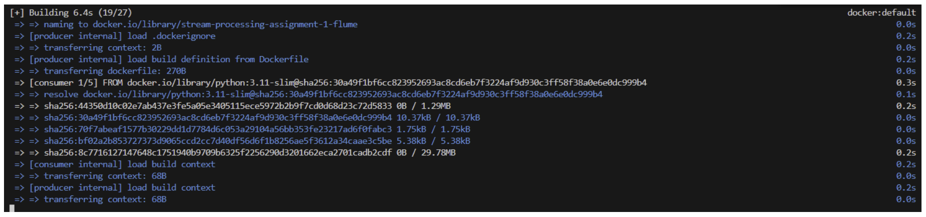
```
# Spark Streaming job example
from pyspark.sql import SparkSession
from pyspark.sql.functions import *

spark = SparkSession.builder \
    .appName("StreamingAnalytics") \
    .getOrCreate()

df = spark \
    .readStream \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "localhost:9092") \
    .option("subscribe", "events") \
    .load()

# Process streaming data
processed_df = df.select(
    from_json(col("value").cast("string"), schema).alias("data")
).select("data.*")

# Write to output sink
query = processed_df.writeStream \
    .outputMode("append") \
    .format("console") \
    .start()
```

6. Production Deployment Commands

```
# Scale consumer instances
docker compose up -d --scale consumer=3

# Deploy with external Kafka cluster
export KAFKA_BOOTSTRAP_SERVERS=external-kafka:9092
docker compose up -d producer consumer

# Monitor resource usage
docker stats

# Backup Kafka data
docker exec kafka tar -czf /tmp/kafka-backup.tar.gz /bitnami/kafka/data
```

7. Application screenshots

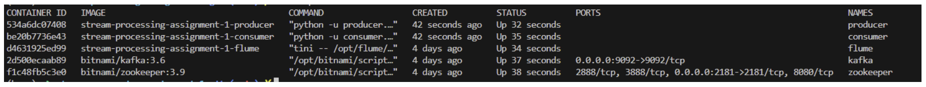
Start the application:



```
[+] Building 6.4s (19/27)                                                                                                                docker:default
 => => naming to docker.io/library/stream-processing-assignment-1-flume                                                                             0.0s
 => [producer internal] load .dockerignore                                                                                                          0.2s
 => => transferring context: 2B                                                                                                                     0.0s
 => [producer internal] load build definition from Dockerfile                                                                                       0.2s
 => => transferring dockerfile: 270B                                                                                                                0.0s
 => [consumer 1/5] FROM docker.io/library/python:3.11-slim@sha256:30a49f1bf6cc823952693ac8cd6eb7f3224af9d930c3ff58f38a0e6e0dc999b4                   0.3s
 => => resolve docker.io/library/python:3.11-slim@sha256:30a49f1bf6cc823952693ac8cd6eb7f3224af9d930c3ff58f38a0e6e0dc999b4                           0.1s
 => => sha256:44350d10c02e7ab437e3fe5a05e3405115ece5972b2b9f7cd0d68d23c72d5833 0B / 1.29MB                                                          0.2s
 => => sha256:30a49f1bf6cc823952693ac8cd6eb7f3224af9d930c3ff58f38a0e6e0dc999b4 10.37kB / 10.37kB                                                    0.0s
 => => sha256:70f7abeaf1577b30229dd1d7784d6c053a29104a56bb353fe23217ad6f0fabc3 1.75kB / 1.75kB                                                      0.0s
 => => sha256:bf02a2b853727373d9065ccd2cc7d40df56d6f1b8256ae5f3612a34caae3c5be 5.38kB / 5.38kB                                                      0.0s
 => => sha256:8c7716127147648c1751940b9709b6325f2256290d3201662eca2701cadb2cdf 0B / 29.78MB                                                         0.2s
 => [consumer internal] load build context                                                                                                          0.2s
 => => transferring context: 68B                                                                                                                    0.0s
 => [producer internal] load build context                                                                                                          0.2s
 => => transferring context: 68B                                                                                                                    0.0s
```

Figure 3: Make up

Check if the services are running using docker ps:



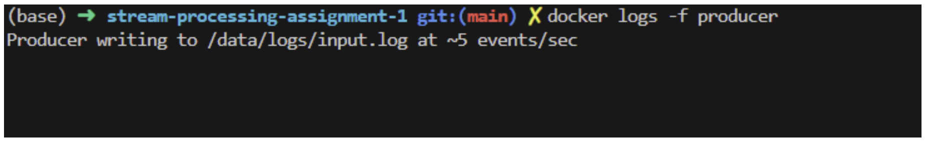
```
CONTAINER ID   IMAGE                                  COMMAND                  CREATED          STATUS          PORTS                                                  NAMES
534a6dc07408   stream-processing-assignment-1-producer   "python -u producer.…"   42 seconds ago   Up 32 seconds                                                          producer
be20b7736e43   stream-processing-assignment-1-consumer   "python -u consumer.…"   42 seconds ago   Up 35 seconds                                                          consumer
d4631925ed99   stream-processing-assignment-1-flume      "tini -- /opt/flume/…"   4 days ago       Up 34 seconds                                                          flume
2d500ecaab89   bitnami/kafka:3.6                         "/opt/bitnami/script…"   4 days ago       Up 37 seconds   0.0.0.0:9092->9092/tcp                                 kafka
f1c48fb5c3e0   bitnami/zookeeper:3.9                     "/opt/bitnami/script…"   4 days ago       Up 38 seconds   2888/tcp, 3888/tcp, 0.0.0.0:2181->2181/tcp, 8080/tcp   zookeeper
```

Figure 4: Docker ps

Check producer service:

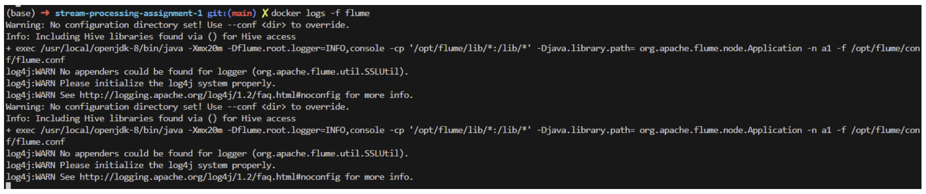
Check flume service:

Check consumer service:



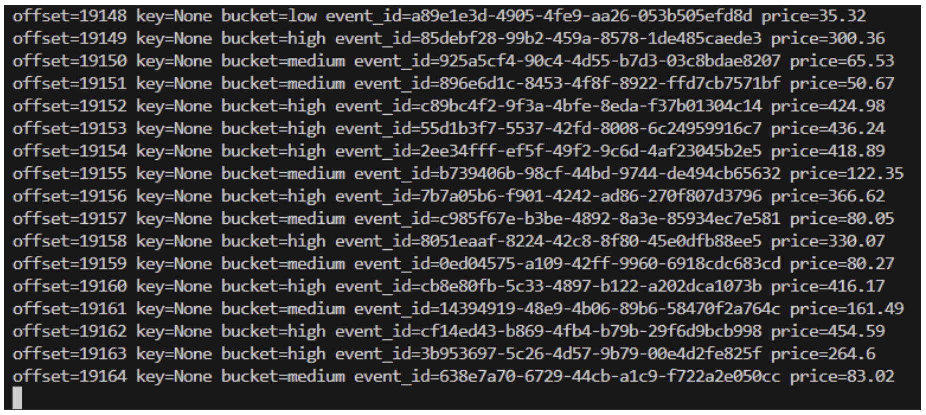
```
(base) ➜ stream-processing-assignment-1 git:(main) ✗ docker logs -f producer
Producer writing to /data/logs/input.log at ~5 events/sec
```

Figure 5: Producer service



```
(base) ➜ stream-processing-assignment-1 git:(main) ✗ docker logs -f flume
Warning: No configuration directory set! Use --conf <dir> to override.
Info: Including Hive libraries found via () for Hive access
+ exec /usr/local/openjdk-8/bin/java -Xmx20m -Dflume.root.logger=INFO,console -cp '/opt/flume/lib/*:/lib/*' -Djava.library.path= org.apache.flume.node.Application -n a1 -f /opt/flume/con
f/flume.conf
log4j:WARN No appenders could be found for logger (org.apache.flume.util.SSLUtil).
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.
Warning: No configuration directory set! Use --conf <dir> to override.
Info: Including Hive libraries found via () for Hive access
+ exec /usr/local/openjdk-8/bin/java -Xmx20m -Dflume.root.logger=INFO,console -cp '/opt/flume/lib/*:/lib/*' -Djava.library.path= org.apache.flume.node.Application -n a1 -f /opt/flume/con
f/flume.conf
log4j:WARN No appenders could be found for logger (org.apache.flume.util.SSLUtil).
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.
```

Figure 6: Flume service



```
offset=19148 key=None bucket=low event_id=a89e1e3d-4905-4fe9-aa26-053b505efd8d price=35.32
offset=19149 key=None bucket=high event_id=85debf28-99b2-459a-8578-1de485caede3 price=300.36
offset=19150 key=None bucket=medium event_id=925a5cf4-90c4-4d55-b7d3-03c8bdae8207 price=65.53
offset=19151 key=None bucket=medium event_id=896e6d1c-8453-4f8f-8922-ffd7cb7571bf price=50.67
offset=19152 key=None bucket=high event_id=c89bc4f2-9f3a-4bfe-8eda-f37b01304c14 price=424.98
offset=19153 key=None bucket=high event_id=55d1b3f7-5537-42fd-8008-6c24959916c7 price=436.24
offset=19154 key=None bucket=high event_id=2ee34fff-ef5f-49f2-9c6d-4af23045b2e5 price=418.89
offset=19155 key=None bucket=medium event_id=b739406b-98cf-44bd-9744-de494cb65632 price=122.35
offset=19156 key=None bucket=high event_id=7b7a05b6-f901-4242-ad86-270f807d3796 price=366.62
offset=19157 key=None bucket=medium event_id=c985f67e-b3be-4892-8a3e-85934ec7e581 price=80.05
offset=19158 key=None bucket=high event_id=8051eaaf-8224-42c8-8f80-45e0dfb88ee5 price=330.07
offset=19159 key=None bucket=medium event_id=0ed04575-a109-42ff-9960-6918cdc683cd price=80.27
offset=19160 key=None bucket=high event_id=cb8e80fb-5c33-4897-b122-a202dca1073b price=416.17
offset=19161 key=None bucket=medium event_id=14394919-48e9-4b06-89b6-58470f2a764c price=161.49
offset=19162 key=None bucket=high event_id=cf14ed43-b869-4fb4-b79b-29f6d9bcb998 price=454.59
offset=19163 key=None bucket=high event_id=3b953697-5c26-4d57-9b79-00e4d2fe825f price=264.6
offset=19164 key=None bucket=medium event_id=638e7a70-6729-44cb-a1c9-f722a2e050cc price=83.02
```

Figure 7: Consumer service

Conclusion

This streaming data processing assignment successfully demonstrates the implementation of a modern, containerized data pipeline using industry-standard technologies. The project showcases several key concepts:

Key Achievements

1. **End-to-End Pipeline**: Successfully implemented a complete data flow from event generation to real-time processing
2. **Containerization**: Leveraged Docker and Docker Compose for consistent, reproducible deployments
3. **Service Integration**: Demonstrated proper service orchestration with dependency management
4. **Resilience**: Implemented retry logic and error handling for production-ready applications
5. **Scalability**: Architecture supports horizontal scaling of consumer instances

Technical Insights

- **Flume Integration**: Successfully configured Flume as a reliable data ingestion layer with Kafka sink
- **Kafka Configuration**: Proper listener configuration for both internal and external access
- **Python Ecosystem**: Leveraged kafka-python for robust consumer implementation
- **Monitoring**: Comprehensive logging and debugging capabilities

Future Enhancements

1. **Schema Registry**: Implement Avro schemas for data validation and evolution
2. **Stream Processing**: Integrate Apache Flink or Kafka Streams for complex event processing
3. **Metrics**: Add Prometheus/Grafana monitoring for operational visibility
4. **Security**: Implement SASL/SSL authentication and authorization
5. **Data Quality**: Add validation and transformation layers

Learning Outcomes

This project provided hands-on experience with: - Distributed streaming architectures - Container orchestration and service discovery - Event-driven system design patterns - Real-time data processing concepts - Production deployment considerations

The implementation serves as a solid foundation for understanding modern data streaming platforms and can be extended for various use cases including IoT data processing, real-time analytics, and event-driven microservices architectures.