BACKEND

Problem Statement

Your Challenge: Design and implement a resilient event processing system that can ingest, process, and analyze real-time data streams. Your solution should handle high throughput while maintaining data integrity and providing actionable insights.

• Ingest data from different sources (e.g., sensors, applications, APIs).

• Process the data reliably while handling errors and failures.

• Allow users to access the processed data through a simple API.

• Monitor system performance and detect issues.

• Handle high traffic and keep running smoothly even under stress

Solution approach:

High-Level Architecture

1. **Event Sources (Producers)**
   - IoT Sensors, Web Applications, APIs, Logs, etc.
   - Producers push events to a message queue (Kafka/Kinesis).
2. **Event Ingestion Layer**
   - **Kafka/Kinesis Streams** act as a buffer.
   - Supports high throughput & durability.
3. **Event Processing Layer**
   - **Apache Flink / Apache Spark Streaming**
   - Processes, transforms, enriches, and filters data.
   - Implements error handling (dead-letter queues, retries).
4. **Storage Layer**
   - **Elasticsearch, DynamoDB, PostgreSQL** (depends on use case).
   - Stores processed data for querying.
5. **API Layer**
   - **Spring Boot + REST API** for data retrieval.
   - Users can query the processed data.
6. **Monitoring & Alerting**
   - **Prometheus + Grafana / AWS CloudWatch**
   - Tracks performance, errors, and sends alerts.
7. **Resilience & Scalability**
   - Auto-scaling groups for microservices.
   - Load testing using **JMeter/K6**.

2. Detailed Breakdown of System Components

1. Event Ingestion Pipeline

- **Tools:** Kafka, AWS Kinesis, RabbitMQ (Kafka preferred).
- **Implementation Steps:**
  - Set up a **Kafka topic** with multiple partitions.
  - Implement **producers** (e.g., Python scripts, Java applications) to send real-time data.
  - Ensure **log retention** and **exactly-once semantics** for data integrity.
- **Key Considerations:**
  - Use **Kafka Connect** for integrating external sources.
  - Enable **idempotent producers** to prevent duplicate events.

2. Detailed Breakdown of System Components

1. Event Ingestion Pipeline

- **Tools:** Kafka.
- **Implementation Steps:**
  - Set up a **Kafka topic** with multiple partitions.
  - Implement **producers** to send real-time data.
  - Ensure **log retention** and **exactly-once semantics** for data integrity.
- **Key Considerations:**
  - Use **Kafka Connect** for integrating external sources.
  - Enable **idempotent producers** to prevent duplicate events.

2. Event Processing with Error Handling

- **Tools:** Apache Flink
- **Implementation Steps:**
  - Consumers **subscribe** to Kafka topics.
  - Process events **in real-time** (transform, enrich, aggregate).
  - Handle **failures** using:
    - **Dead Letter Queues (DLQ)**
    - **Retry mechanisms** (exponential backoff)
    - **Checkpointing & State Management** in Flink.
- **Error Handling Approach:**
  - **Retry up to 3 times**, then push failed events to **DLQ**.
  - Use **Kafka Streams state stores** to keep track of event status.

3. Queryable API for Processed Data

- **Tools:** Spring Boot (Java)
- **Database Options:**
  - **Elasticsearch** (for fast searches)
  - **PostgreSQL** (for structured data, preferred here)
  - **DynamoDB** (for scalable NoSQL storage)
- **API Endpoints:**

Method	Endpoint	Description
GET	/events	Get all processed events
GET	/events/{id}	Get event by ID
GET	/events/all	Get real-time event statistics

4. Monitoring & Alerts

- **Tools:** Prometheus, Grafana, other options can be: AWS CloudWatch, ELK Stack.
- **Key Metrics:**
  - **Event ingestion rate**
  - **Event processing latency**
  - **System errors & failures**
  - **Dead Letter Queue (DLQ) size**
- **Implementation Steps:**
  - Integrate **Prometheus exporters** in Kafka & Flink.
  - Configure **Grafana dashboards** for real-time monitoring.
  - Set up **alerts** for failures using AWS SNS or Slack notifications.

5. System Resilience & Load Testing

- **Resilience Strategies:**
  - **Horizontal Scaling:** Increase Kafka partitions, Flink job managers.
  - **Circuit Breakers:** Use **Resilience4j** to prevent cascading failures.
  - **Retries & Backpressure:** Kafka consumers with exponential backoff.
- **Load Testing:**
  - **Tools:** JMeter
  - **Scenarios:**
    - Simulate **100k+ events/sec**.
    - Test API with **high concurrent requests**.

3. Evaluation Criteria Breakdown

Criteria	How Our System Meets It
Scalability	Kafka partitions, auto-scaling, Flink parallelism
Resilience	DLQs, retries, circuit breakers, checkpointing
Performance	Low-latency processing, high throughput
Design	Modular, microservices-based, clean code
Innovation	Flink for real-time analytics, scalable Kafka-based ingestion

Technology Stack

- **Event Ingestion**: Kafka
- **Processing**: Apache Flink / Apache Spark
- **Storage**: PostgreSQL
- **API**: Spring Boot
- **Monitoring**: Prometheus + Grafana
- **Containerization & Deployment**: Docker

Benefits of DLQ

- **Prevents message loss** even if failures occur.
- **Easier debugging** by storing failed events separately.
- **Improves resilience** by isolating problematic data.

Setup Guide

Install Required Tools

Kafka Setup (Using Docker)

1. Install Docker from Docker Website
2. Start **Kafka & Zookeeper**: docker-compose up -d
3. Verify Kafka is running: docker ps

Set Up Apache Flink

Install Flink

1. Download Apache Flink:

wget https://dlcdn.apache.org/flink/flink-1.15.0-bin-scala_2.12.tgz

tar -xvzf flink-1.15.0-bin-scala_2.12.tgz

cd flink-1.15.0

1. Start flink

./bin/start-cluster.sh

Set Up Spring Boot API

🔹 Install Java & Maven

1. Install Java 17
2. Install maven

Restart Everything

docker-compose down

docker-compose up -d

How to Run the System

1. **Start dependencies**

docker-compose up -d

1. **Run the Spring Boot Application**

mvn spring-boot:run

1. **Test Event Processing**

Publish an event

curl -X POST "http://localhost:8080/events?eventType=Temperature&eventData=30C"

Retrieve events

curl http://localhost:8080/events

To check logs

docker-compose logs kafka

Demo images: