Webhook Notifier Event Processing Solution

1. Introduction

The Webhook Notifier Event Processing Solution is designed to handle high-throughput event processing (up to 1 billion events per month) for a webhook notification system. This solution ensures three core principles: **Scalability**, **Reliability**, and **Fairness**. It leverages **Apache Kafka** as the message broker, **Redis** for rate limiting and deduplication, and incorporates retry mechanisms and circuit breakers for robust webhook invocation. The implementation uses **Java Spring** as the primary programming language, with **PostgreSQL** as the database. The entire system is deployed on a **Kubernetes (k8s)** cluster using Helm charts, with comprehensive monitoring via Loki, Grafana, Prometheus and AlertManager.

This document outlines the architecture, processing flow, infrastructure, monitoring strategy, testing approach, and future improvements.

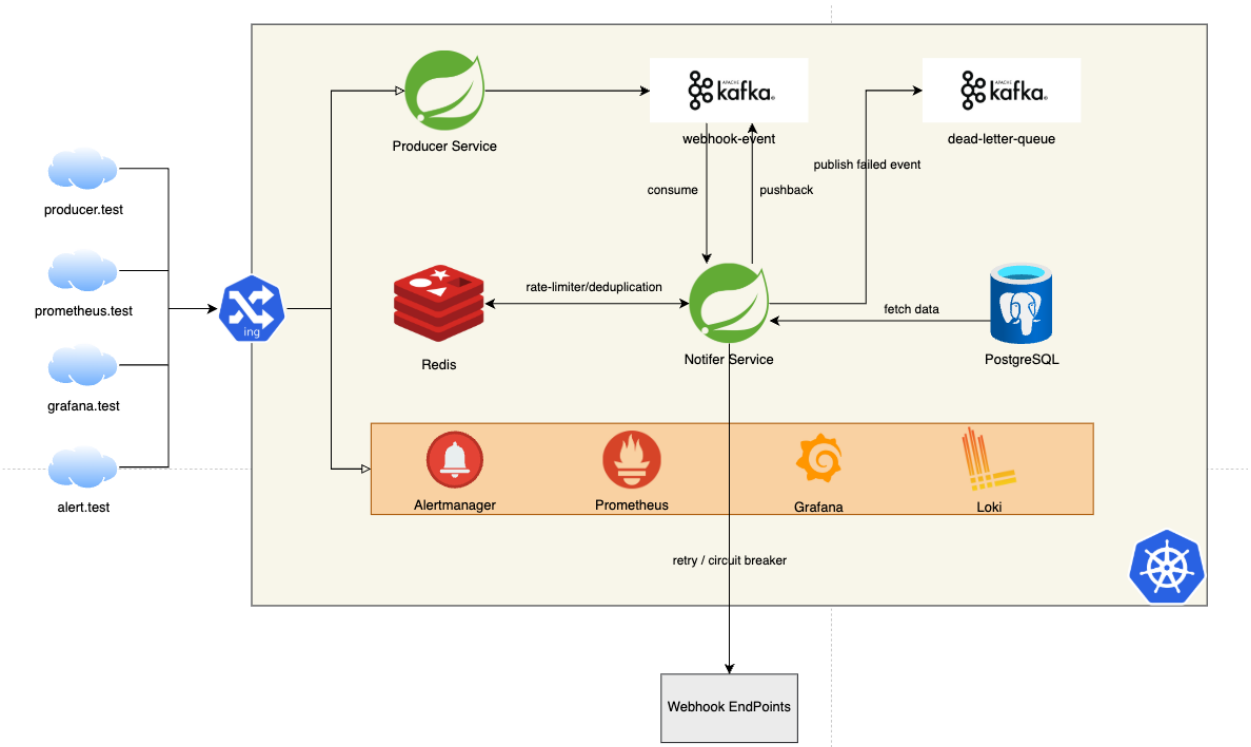
2. System Architecture

2.1 Components

- **Producer**: Pre-existing component within the webhook registration system, responsible for publishing events to a Kafka topic.
- **Consumer (Notifier)**: Processes events from Kafka in batches, ensuring scalability, reliability, and fairness.
- **Message Broker**: Apache Kafka, handling event queuing and distribution.
- **Rate Limiter & Deduplication**: Redis, used for rate limiting webhook calls and preventing duplicate event processing.
- **Database**: PostgreSQL, storing detailed event data for retrieval during processing.
- **Dead Letter Queue (DLQ)**: A Kafka topic for storing events that fail processing after retries or circuit breaker triggers.
- **Monitoring Stack**: Loki (log aggregation), Promtail (log collection), Grafana (visualization), Prometheus (metrics collection) and AlertManager.

2.2 Architecture Diagram

The system architecture illustrates the interaction between components:



3. Event Processing Flow

3.1 Overview

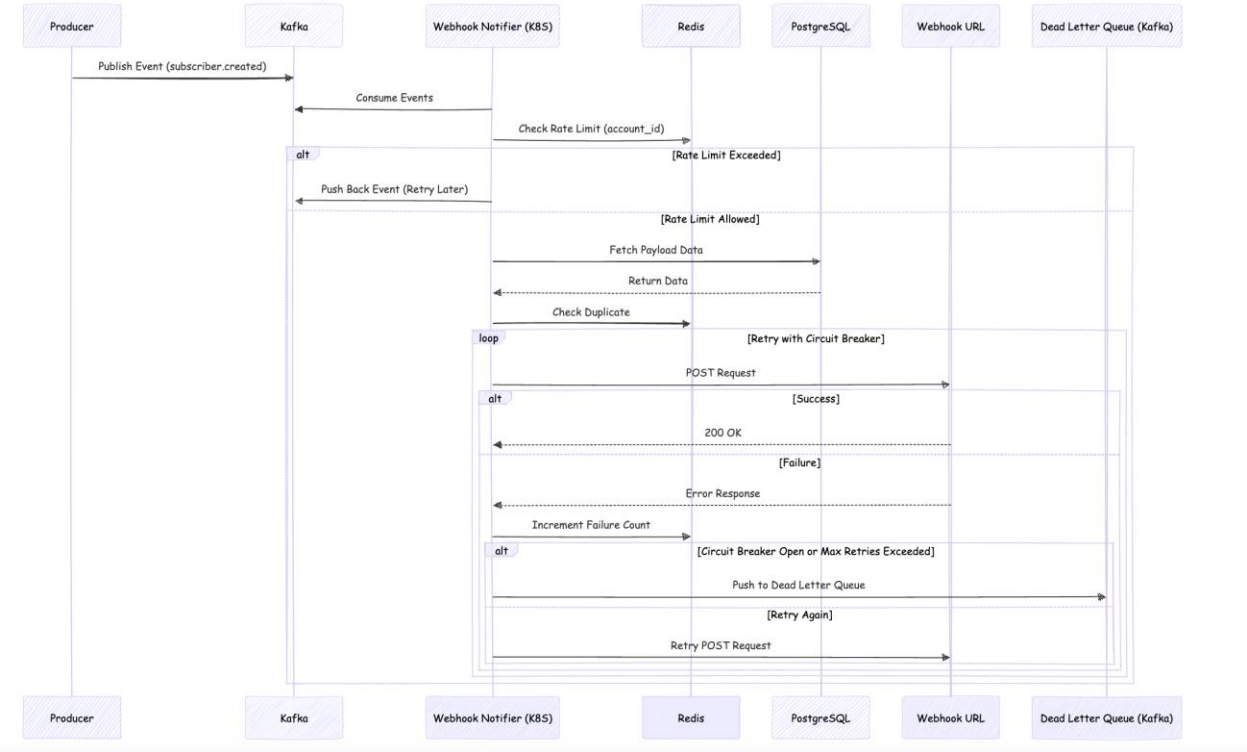
The processing flow ensures efficient handling of events from production to webhook invocation, with safeguards for reliability and fairness.

3.2 Detailed Flow

- **Event Production**:
    - The Producer publishes events to a designated Kafka topic.
- **Event Consumption**:
    - The Consumer retrieves events in batches (default: 100 events per batch) from Kafka.
    - Group event by event type.
    - **Rate Limiting Check**: Verifies against Redis rate limits by account id; if exceeded, the event is re-queued to Kafka for later processing.
    - Event data is fetched from PostgreSQL in bulk, supporting multiple event types.
    - Batch processing is parallelized using multi-threading for scalability.
- **Per-Event Processing**:
    - **Deduplication Check**: Queries Redis to identify duplicates; skips processing if detected.
    - **Webhook Invocation**:
        - Executes HTTP calls to webhook endpoints with a retry mechanism (e.g., exponential backoff).
        - Implements a circuit breaker per webhook URL to prevent cascading failures.
        - On success: Mark processed event in Redis, proceeds to the next event.
        - On failure (after retries or circuit breaker activation): Pushes the event to the Dead Letter Queue (DLQ).
- **Batch Completion**:
    - If all events in the batch are processed successfully (no exceptions), the Consumer sends an acknowledgment (ACK) to Kafka, committing the offset.

3.3 Sequence Diagram

The sequence diagram below outlines the event processing logic:



4. Infrastructure

4.1 Deployment

The system is deployed on a Kubernetes cluster using Helm charts for the following components:

- **Producer**: Publishes events to Kafka (Mock service for test).
- **Consumer (Notifier)**: Processes events and invokes webhooks.
- Webhook Endpoints: Receives POST requests from Notifier (Mock service for test).
- **Kafka**: Message broker with configurable topics and partitions.
- **Redis**: In-memory store for rate limiting and deduplication.
- **PostgreSQL**: Persistent storage for event data.
- **Monitoring Stack**: Loki, Promtail, Grafana, Prometheus and AlertManager.

4.2 Configuration

- **Kafka**: Configured for high throughput and durability (e.g., multiple partitions, replication factor).
- **Redis**: Optimized for low-latency key-value operations.
- **k8s**: Auto-scaling enabled for Consumer pods based on workload

5. Monitoring and Observability

5.1 Logging

- **Tools**: Loki for log aggregation, Promtail for log collection.
- **Visualization**: Grafana dashboards displaying log data.

5.2 Metrics

- **Collector**: Prometheus.
- **Defined Metrics**:
    - kafka.batch.processing.time: Duration to process a Kafka batch (ms).
    - kafka.event.count: Total events processed.
    - webhook.execution.count: Total webhook invocation attempts.
    - webhook.success.count: Successful webhook calls.
    - webhook.failure: Failed webhook calls.
    - webhook.circuit.open: Instances of circuit breaker activation.

5.3 Alerts

Alerts are configured in Prometheus using Alertmanager to notify teams of critical conditions. Below are example configurations for key metrics:

5.3.1 High Batch Processing Time

- **Metric**: kafka.batch.processing.time
- **Condition**: Batch processing time exceeds 500ms for 5 minutes.
- **Action**: Notify the operations team to investigate Consumer performance or Kafka lag.

5.3.2 High Webhook Failure Rate

- **Metric**: webhook.failure and webhook.execution.count
- **Condition**: Webhook failure rate exceeds 10% over 10 minutes.
- **Action**: Escalate to the team to check webhook endpoints or circuit breaker states.

5.3.3 Circuit Breaker Activation

- **Metric**: webhook.circuit.open
- **Condition**: Any circuit breaker opens for more than 5 minutes.
- **Action**: Investigate specific webhook URLs and coordinate with vendors if necessary.

5.3.4 Low Event Processing Throughput

- **Metric**: kafka.event.count
- **Condition**: Event processing rate drops below 100 events/second for 15 minutes.
- **Action**: Check Consumer pod scaling and resource utilization.

5.4 Notification Channels

- Alerts are routed via Alertmanager to channels such as Slack, email, or PagerDuty, based on severity (warning or critical).

6. Testing

6.1 Unit Testing

- **Coverage**: Partial unit tests implemented for key components (e.g., Consumer logic, webhook retry).
- **Limitation**: Not 100% coverage due to effort constraints.

6.2 Performance Testing

- **Tool**: JMeter.
- **Scenario**: Simulated bunch of API requests to the Producer, publishing events to Kafka.

- **Results**: We will determine how many events it can process in a minute for an instance -> adjust config for batch size, thread pool size, rate limit size... to tuning performance.

6.3 Test Data Generation

- **Tools**: Custom Python scripts to generate test data and CSV files for JMeter.

7. Benchmark Results

- Test Setup: 10,000 API requests sent via JMeter to simulate event production.
- Results: Throughput of approximately 70 events/second per instance (0.25 vCPU, 500 MB RAM). Ran with 2 instances, configured to limit 2,000 events per minute per account ID.
- Scalability Target: Validation for 1 billion events per month is pending.

8. Future Improvements (To-Do)

- **Optimize Batch Processing**:
  - Fine-tune batch size and thread pool configurations to maximize throughput and minimize latency.
- **Advanced DLQ Handling**:
  - Implement retry policies for DLQ events and a manual intervention workflow for unresolved failures.
- **Deploy on AWS Cloud**:
  - Migrate the Kubernetes cluster to AWS EKS (Elastic Kubernetes Service) for managed scalability and resilience.
  - Utilize AWS-managed services (e.g., MSK for Kafka, ElastiCache for Redis, RDS for PostgreSQL) to reduce operational overhead.
- **Setup CI/CD Pipeline**:
  - Implement a CI/CD pipeline using tools like GitHub Actions, CodePipeline or Jenkins and ArgoCD.
  - Automate build, test, and deployment processes for Producer, Consumer to AWS EKS.
- **Handle Webhook Calls with Vendor Rate Limits**:

    - **Solution**: Extend the existing rate limiter in Redis to respect vendor-specific limits (e.g., API quotas per minute). Configure dynamic throttling per webhook URL based on vendor documentation, and queue excess events in Kafka with a delay for retry.
- **Optimize Event Data Fetching**:
    - **Option 1**: Send event payloads directly in Kafka messages to eliminate DB fetches, reducing latency and DB load. Requires Producer to embed all necessary data in the event.
    - **Option 2**: Use a PostgreSQL read replica for Consumer queries to offload the master DB, ensuring high availability and scalability.
- **Add Delay in Producer for DB Sync**:
    - Introduce a configurable delay (e.g., 1-5 seconds) in the Producer before publishing events to Kafka, allowing time for master-to-replica DB synchronization. Ensure consistency when using read replicas in the Consumer.

9. Conclusion

The Webhook Notifier Event Processing Solution effectively addresses high-volume event processing with a focus on scalability (parallel, multi-threading, batching), reliability (retries, circuit breakers, DLQ), and fairness (rate limiting). Deployed on Kubernetes with robust monitoring, it provides a solid foundation for webhook notifications. Ongoing improvements will further enhance its resilience, performance, and operational efficiency, particularly with planned AWS deployment and CI/CD integration.