1 Overview

The event notification service utilizes Change Data Capture (CDC) for monitoring the MySQL database and implementing message queuing for event processing. The service processes success and failure events, sending email notifications to subscribed users in near real-time, based on their preferences and relevant event attributes.

2 Goals

1. **Notify users:** Notification service should send email notifications to users about event processing results.
2. **Near real-time:** Poll the events from the database and notify customers in near real-time. By receiving timely notifications, users can take appropriate actions, resolve issues, and make data-driven decisions to optimize their processes.
3. **Scalability:** Notification service should be able to handle a large number of events and users.

3 Architecture Design

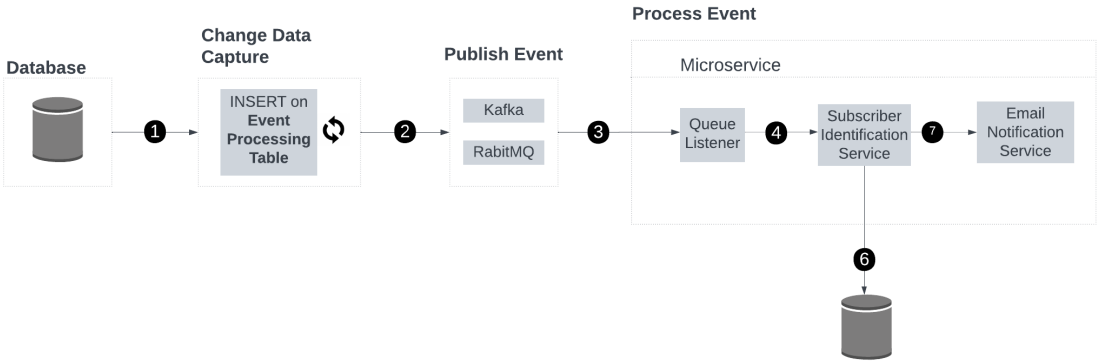

Figure 1: Architecture design

4 Proposed Solution

To achieve the above goals, the following key factors are considered:

1. Leverage Change Data Capture (CDC) technology to capture changes from the MySQL database in near real-time without impacting its performance
2. Utilize an asynchronous message queue to decouple event producers(CDC) and consumers, ensuring efficient and scalable event processing

3. Event consumers are designed to categorize events as success or failure based on the queue output.
4. The notification microservice functions as the recipient of the message and dispatches event status emails to the subscribed customers.

5 Scalability in design

1. The CDC technology is considered to eliminate basic polling in design so that the database is decoupled from new business logic. If the number of transactions increases in the database, CDC can manage to provide powerful performance by reading transaction log files rather than hitting the database directly.
2. The async queue architecture should be designed to scale horizontally, enabling it to accommodate increased data streams and consumer demand without sacrificing performance
3. Enable parallel processing of messages within consumers, enabling higher throughput and load balancing.
4. Architectures like Kafka should be used to implement data replication, ensuring fault tolerance and availability..

6 Fault tolerance

1. CDC connectors should be distributed across multiple machines so that, if anything goes wrong, other connectors can be used till faulty one can be restarted
2. Each poll/hit to database can be logged such that even when the connector goes down the next change capture can resume from where it left rather than starting new or capturing only from the point of failure
3. We can implement "Dead letter queue (DLQ)" such that if an event cannot be processed after a certain number of retries, move it to a DLQ topic to avoid processing bottlenecks. This will help in later analysis and debugging.
4. Email retries should be set to a fixed amount with helps overcome network issues at the same time keeping the performance bottleneck intact.

7 Appendix

7.1 Success mail template

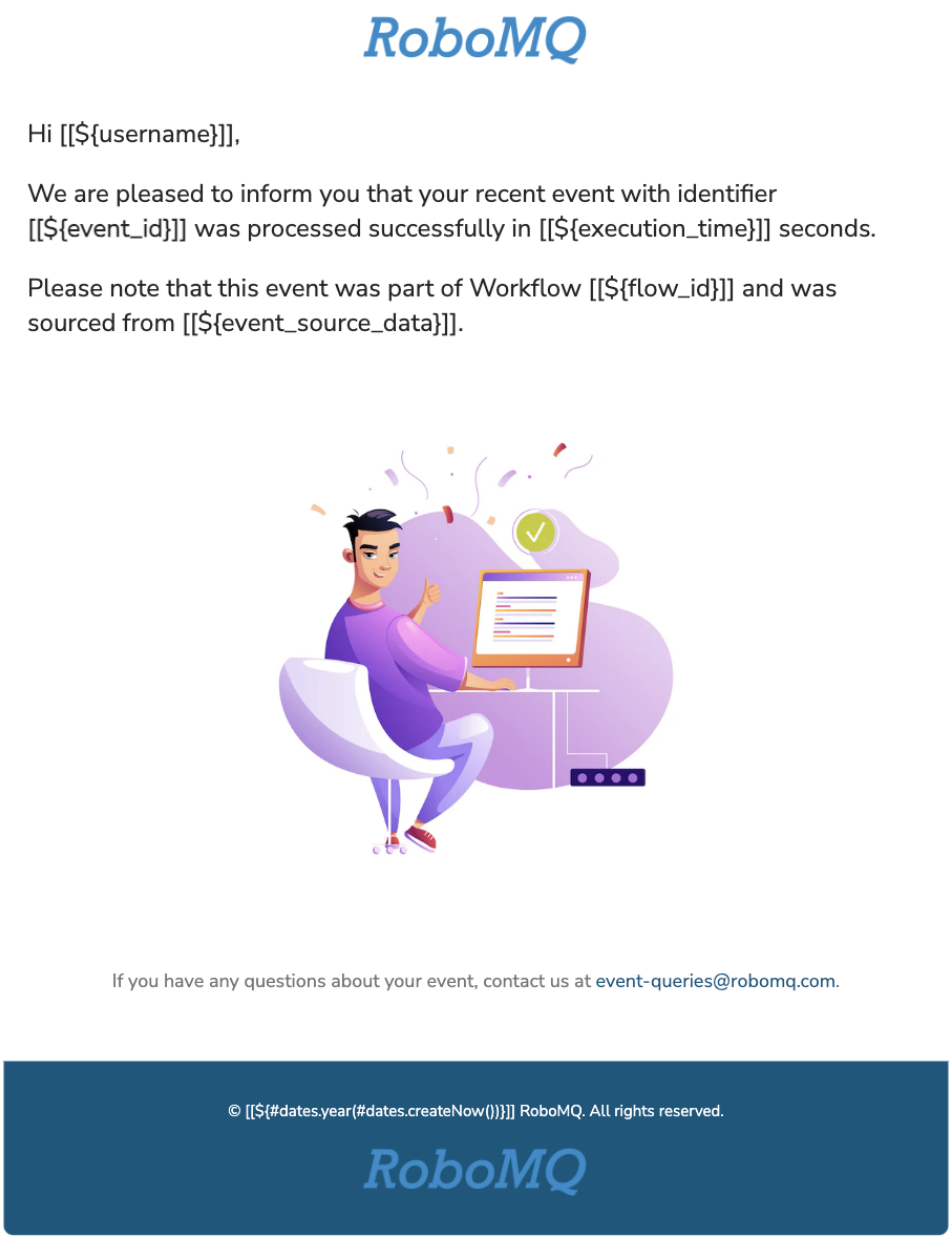

Figure 2: Success mail template

7.2 Sample Success mail

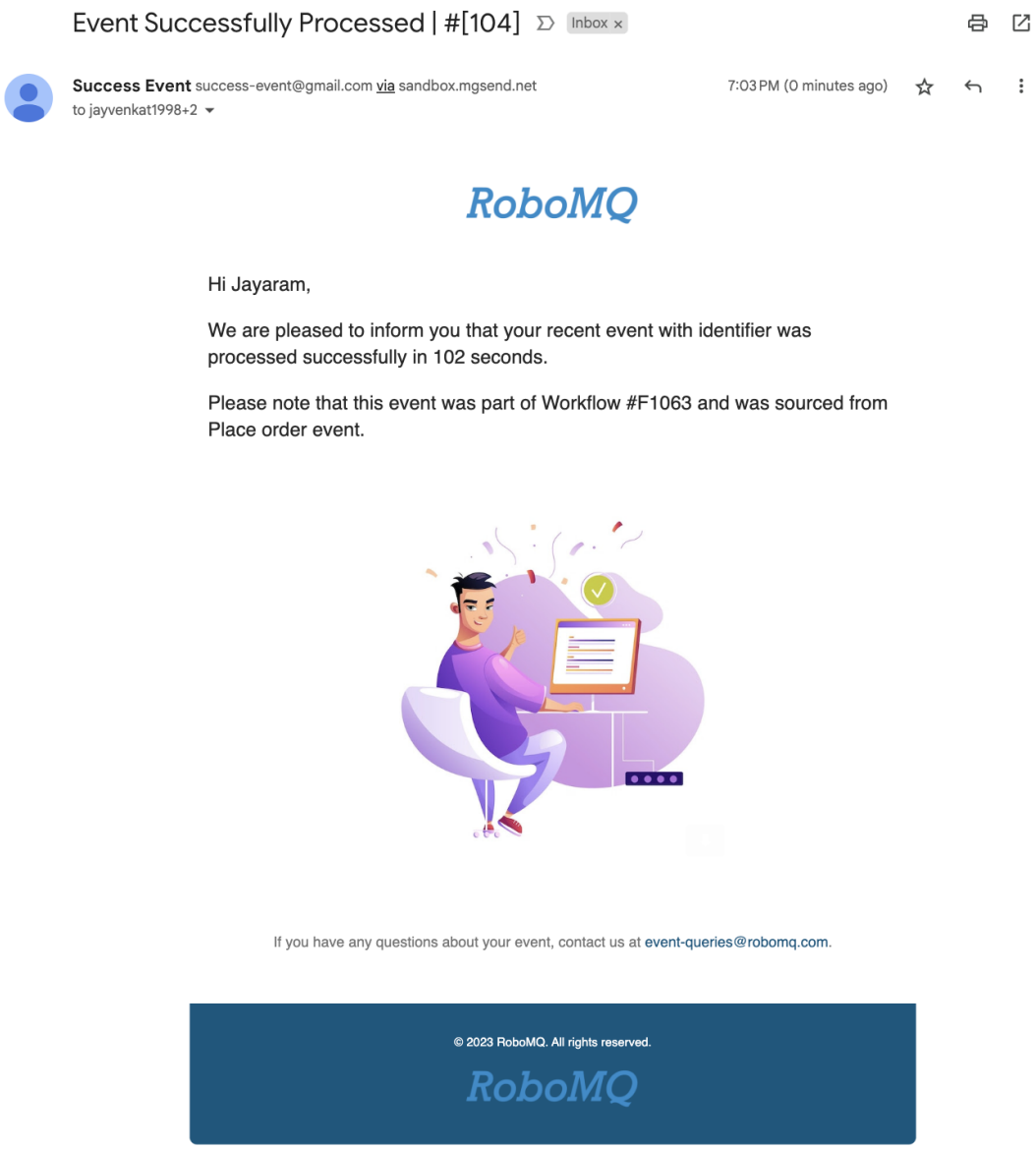

Figure 3: Sample Success mail

7.3 Failure mail template

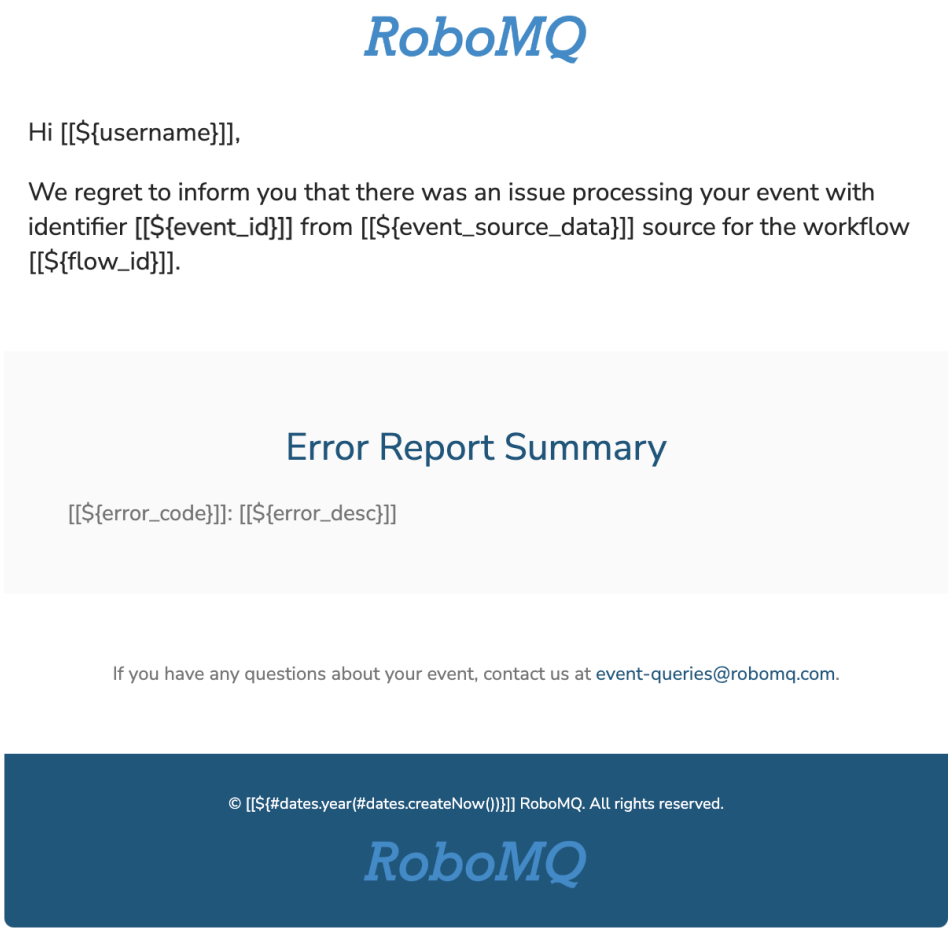

Figure 4: Failure mail template

7.4 Sample Failure mail

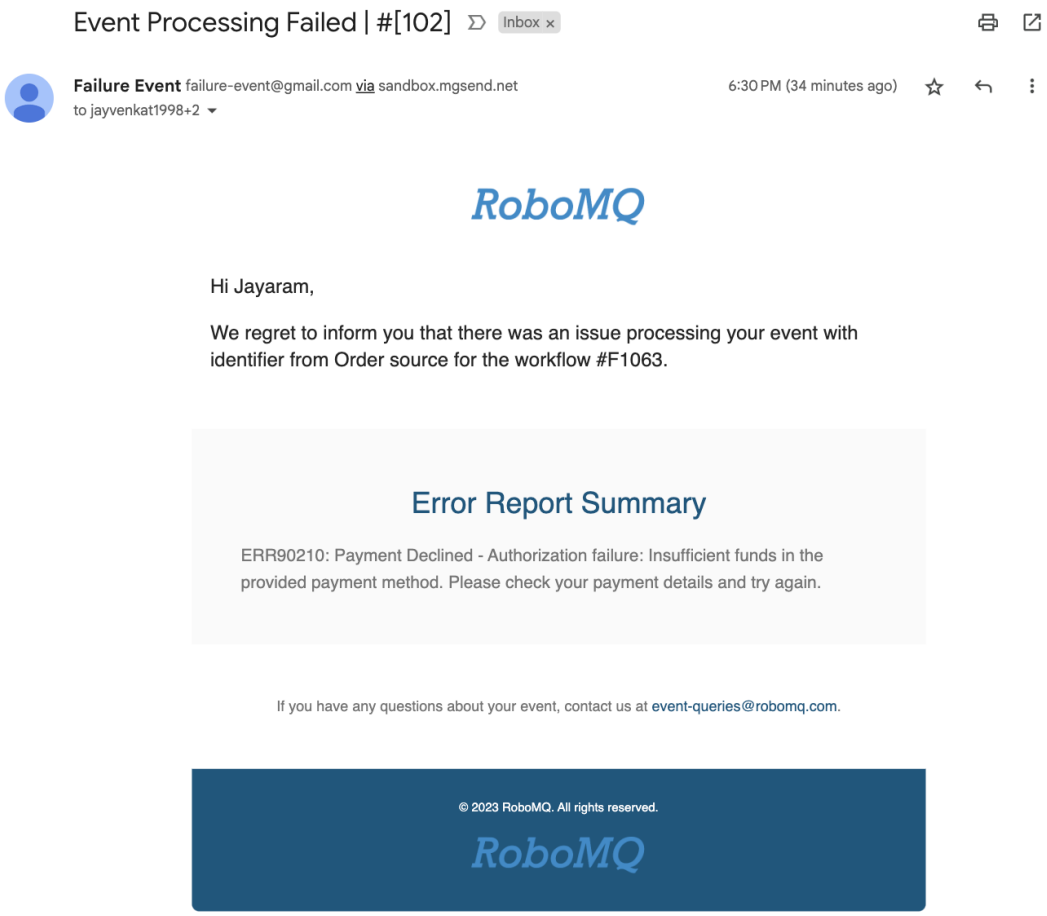

Figure 5: Sample Failure mail

7.5 Prototype Architecture

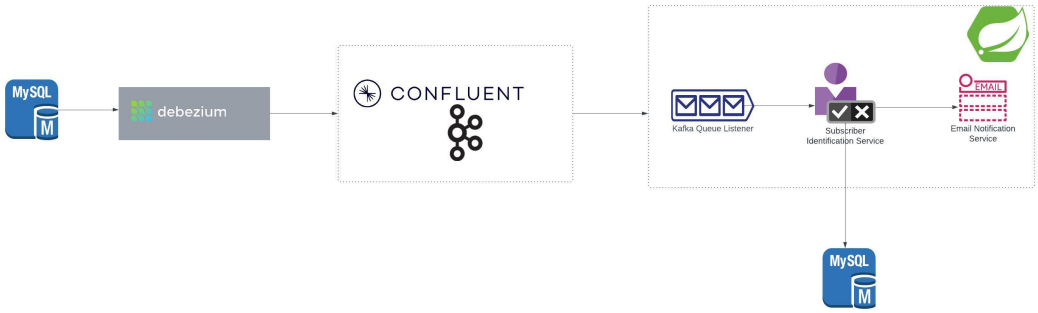

Figure 6: Prototype Architecture