System Design - Billing Engine

Description

This system architecture proposes a design option for an insurance Policy Management, Billing and Payments system and integration with third party payment providers. While a real world application would be complex, this design only focuses on key areas.

Assumptions

As per available information AAA Life has roughly 1.7 million active policies (https://www.aaalife.com/about-us). Let us make assumptions based on this scale

Active policies : 2 million
Active system users: 2 million (users using the application to request quotes, make payments, etc.)
Requests to make payments : Assume 80% people will pay towards the end of the month in the last week. Of these 80% would have setup automatic recurring payments..
So assume 2 Mil. payments per week at peak load in combination of recurring and via website.

Requests to make payments at peak load
Per week : 2 Mil.
Per Day: 2 Mil / 7 = 285, 714
Per sec: 2 Mil / (7 x 24 * 60 * 60) = 2Mil/ 604,800 = 3.3

Non-functional requirements

Availability: System should be highly available to process payments in timely manner

Security: System should adhere to PCI standards and handle all financial data with care. Use of secure protocols like https and strong encryption and authorization is required. All personal data stored in the database should be encrypted.

Performance: System should perform without excessive delays. Especially when making payment requests which go through third party systems.

Scalability: System should scale horizontally for peak loads towards end of month processing.

Overview

Given the above, it is desirable to host the system in Cloud for security. Also horizontal scaling required during peak load is better managed in Cloud.

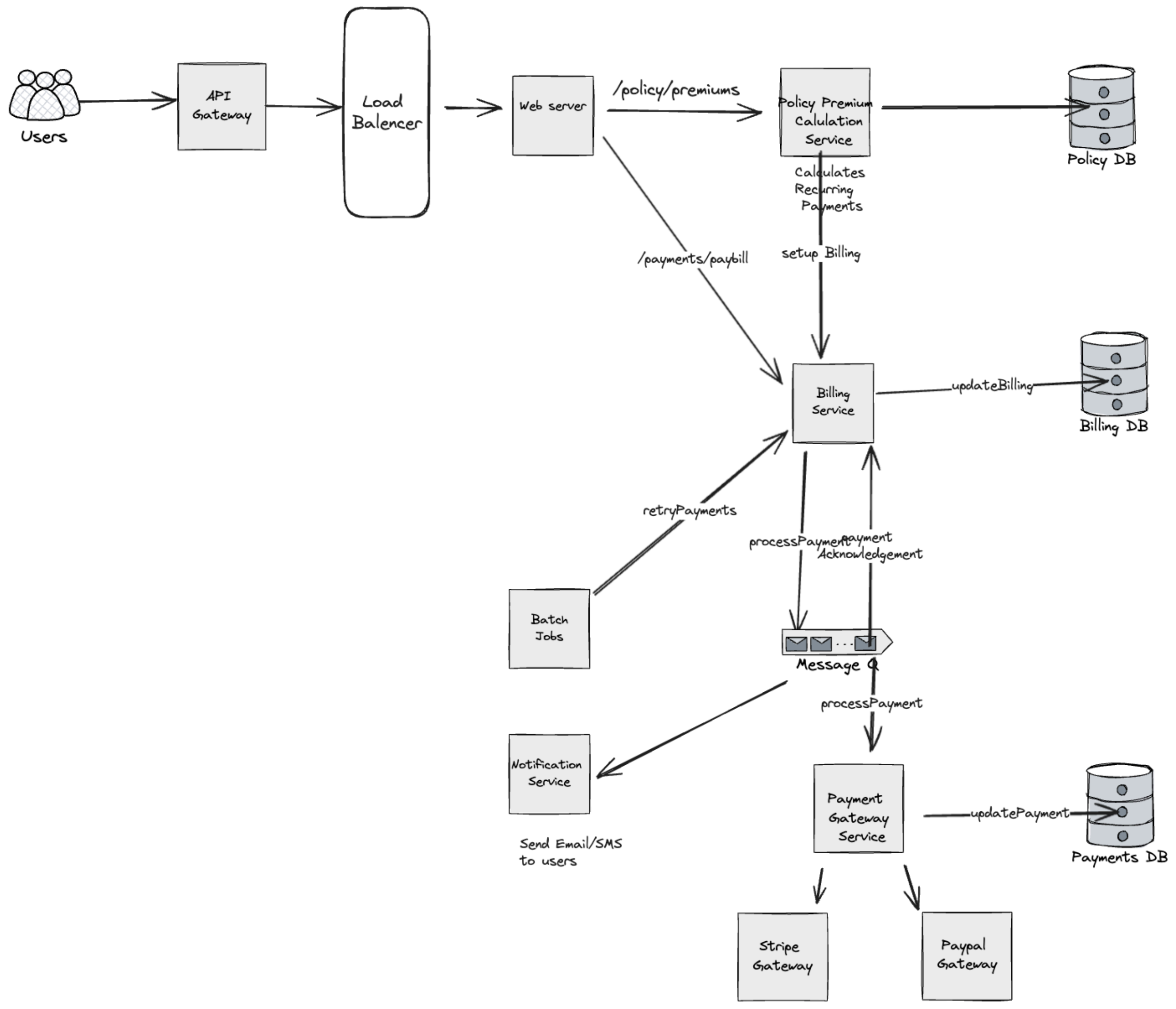
It would also be desirable to use microservice architecture as different parts of the system have different scaling and availability requirements.

The data should be stored in relational database giving the transactional nature of the application (MySQL, Postgres)

To address performance of long workflows we will use messaging (pub/sub) system like Kafka or RabbitMQ.

Scaling can be achieved by deploying microservices to a K8 cluster.

High level design



Data Model

POLICY

ID
HolderName
stateDate
endDate
isDelinquent
paymentStatus(GRACE/CURRENT)
totalPremiumAmount

PAYMENT

ID
amount
policyId
transactionReference
retryCount
startDateTime
endDateTime
status (SUCCESS/FAILED)

BILLING_SCHEDULE

ID
policyId
paymentDueDate
amount
status (DUE/COMPLETE/NONE/PARTIAL)

Design Workflows

Calculate recurring premiums based on policy metadata

- User posts a request to Premium Calculation Service. This will calculate the premium and the schedule.
- Premium Calculation Service will notify Billing Service for the updated schedule

Trigger payment reminders and retry logic

- These will be handled via batch jobs.
- There will be three jobs. Payment Reminder Job, Retry Failed Payments, Process Recurring Payments
- Payment Reminder Job will send notifications via Notifications service
- Process Recurring Payments and Retry Failed Payments will call Billing Service which will send payment requests to gateway

Record and track payments and delinquency status

- Payment Gateway will forward requests to payment providers services which in turn will make external calls to third party systems
- The processing results will be notified back to the Billing Service
- Billing Service will update schedule and policy status

Integrate with third-party payment providers

- Payment Gateway will maintain details of each provider
- Based on payment type, a specific provider will be chosen
- The API for the provider will be invoked
- The transaction will be recorded in PaymentsDB

Focus areas

Event-driven or pub/sub architecture for notifications

Message Brokers like RabbitMQ will be used to deliver messages between services. RabbitMQ can handle 20,000 messages per sec. So it should scale well for the application. Async. messaging will decouple inter service communication in workflows which are performance sensitive and immediate response or action is not required.

- Sending of emails can be done asynchronous
- Processing of Payments. Given the high number payment requests, we could face rate limiting by third party provider APIs. Hence we may need to retry after a short while. Hence requests for payments by Billing Service should be asynchronous.
- Notifying Billing Service of a new Policy Schedule. The billing setup can be done after policy is issued and accepted by user. Hence this workflow is also asynchronous

Grace period handling and retry schedule logic

Grace Period Handling

- Billing service will mark a policy as under grace period when the batch job for checking delinquent policies runs.
- It will use the information present in database to check if policy has entered grace period or is delinquent

Payment Retries

- There are two scenarios. First is rate limiting or error on third party providers.Other is bad payment information.
- Payment Gateway Service will record when the payment request was sent and the corresponding response in Payment DB
- In case of API error, it will retry upto 3 times. The retryCount and next attempt timestamp will be stored in the database
- Payment Gateway Service will run a scheduled job to retry. If retryCount reaches 3 it will stop further retries and notify billing service of failed payment
-

  If payment is rejected, then Billing service will be notified.
- Billing service will send email notification to Policy Holder
- Once payment details are corrected, the Retry Job will send payment for processing

Extensibility for adding multiple payment channels

- Payment Gateway Service acts as a facade between the Message Queue and integration services.
- The payment request message will contain payment details. Based on the details appropriate service will be called.
- For recurring payments, prior authorization tokens will be stored and used

Notes on scaling and security
Scaling is achieved by horizontal scaling of K8 container instances per microservice. Decoupling via messaging services allows each component to process messages according to its performance capabilities. E.g. a burst of payment requests won't overwhelm the Payment Gateway. However with this, you will also have a problem of duplicate messages and dead letter queue which needs to be handled by the application

Data should be stored and transmitted securely and comply with PCI-DSS standards. For recurring payments, the application should get authorization tokens per user. These tokens should be used for processing payments. It will avoid the need to store credit card information on file.