What are the most challenging projects you worked on?

Revamp the entire payment system

1. Signing up for a wellness membership plan

2. And then cancelling plan

3. Failed, retry jobs that gets scheduled to run so that we will be running the job

4. How to clean up the job properly

5. Querying the terminal for the transcation

**Key Challenges:**

1. **Scalability Issues**: The existing system was not designed to handle high transaction volumes. As the business expanded and started processing millions of transactions a day, the old monolithic payment processor struggled to keep up with the demand, leading to slow response times and occasional outages.  
     
   Need to migrate to micro-service based
2. **Transaction Reliability**: Making sure no duplicate msg and no downpayment
3. **Integration with Multiple Payment Gateways**: The legacy system was hardcoded to work with a single payment gateway. As part of the revamp, we needed to support multiple payment providers (e.g., Stripe, PayPal, and local payment solutions). Each provider had different API structures, error handling, and rate limits, which added complexity.
4. **Security and Compliance**: Given the nature of the data (credit card information, user details), the system needed to be compliant with **PCI-DSS** standards and ensure encryption of sensitive information both **in transit** and **at rest**.

Challenge

1. Using a micrsoervice architecture

To address scalability and reliability, we transitioned the monolithic payment system to a **microservices architecture**. We split the payment system into different services, each responsible for a specific part of the payment workflow, such as transaction initiation, authorization, payment processing, and fraud detection.

* **How it worked**: This allowed us to scale each service independently. For example, the transaction authorization service could be scaled up when transaction volume was high, while the fraud detection service could operate at a different scale, based on the business needs.

2. Using kafka and the topics

Each step in the payment process (e.g., transaction initiation, authorization, success, failure) was treated as an event and placed into a Kafka topic.

Allow us to process large volume of transaction

**3. Idempotency and Exactly-Once Semantics:**

Ensuring **exactly-once delivery** was critical for transaction reliability. We used **idempotency keys** to guarantee that repeated requests (e.g., in case of network timeouts or retries) didn’t result in duplicate payments.

* **How it worked**: Each transaction request included a unique idempotency key. When a transaction was processed, the system stored the result associated with that key. If the same transaction request came in again with the same key, the system would return the previous response without reprocessing the payment.
* **Why it worked**: This allowed us to avoid duplicate payments and maintain the integrity of transactions even in the case of retries or network failures. We also leveraged Kafka’s **exactly-once semantics** for message processing to ensure messages weren’t lost or duplicated.

4. POS termianl payment design was a bit ocmplicated

Braintree POS does not send back a payment success/failed response, so we need to keep polling for the status of the transaction.

Can take a look at ackonlwdgement deadline

1. Finance-service sends a requestChargeFromInStoreReader request to the reader and receive RequestChargeInStoreContext.id
2. Store the transaction details in the DB.
3. Start polling job (or this can be achieved via pub/sub) to check the transaction status every 2 secs and check RequestChargeInStoreContext.status.
4. Update DB every time transactionstatus is changed and notify subscriber for UI to update.
5. Stop polling when status is COMPLETE, store complete transaction details in the DB and notify subscriber.

Due to the timeout of braintree, message queu configuration is really important

**Design walk-through**

Creating 3 separate topics as the configuration for each topic will differ   
Please see step 2 on how these steps are different

**Topic 1**: Messages in CREATED state. ACK deadline is 60 secs (Braintree API timeout)

**Topic 2:** Messages in PENDING state. ACK deadline is 180 secs (Reader timeout waiting on customer’s action)

**Topic 3:** Messages in COMPLETED, FAILED or CANCELLED state. Used to update UI. ACK deadline can be decided later.

Let’s review the message queue configuration before we dive deep into message processing

* **Delivery type:** There are 2 ways messages can be delivered to the subscriber. Pull or Push. Pull is recommended
  + **Pull (recommended):** Subscriber will pull the messages from the broker on-demand. Messages can be consumed in batches depending upon subscriber throughput. Ideal, if you need flexibility in processing rates and do not want to expose any public endpoint. There are multiple pull strategies:
    - **Continuous Pulling (recommended):** Continuously pull messages from Pub/Sub, with a short or no delay between pulls.
    - **Interval-Based Pulling:** Pull messages at fixed intervals.
    - **Event-Driven Pulling:** Trigger the pull operation based on specific events or conditions, such as reaching a message count threshold or resource availability.
* **Acknowledgement deadline:** How long Pub/Sub waits for the subscriber to acknowledge receipt before resending the message. Min/Default is 10 secs and Max is 600 secs.  
    
   **Braintree API timeout is 60 secs and Reader timeout is 180 secs**. Our subscriber should not timeout before Braintree API timeout.
  + **Topic 1**: Configure ACK deadline to be 10 secs with 3 retries and exponential back-off strategy.   
      
    If broker does not receives ACK after Tx with CREATED state message was sent to the subscriber within 10 secs, then message is put back to the queue and gets visible to the subscribers to consume.   
      
    After multiple retries, the message will move to DLQ which can be analyzed separately. The total\_timeout for a message to be processed should be >= 60 secs.
  + **Topic 2:** This is an interesting case when message in PENDING state is consumed and we need to call node query multiple times. There can be multiple approaches:
    - **ACK deadline is 180 secs**: Once the message is delivered to subscriber, it’s subscriber responsibility to keep calling node query every 2-3 secs until transaction gets COMPLETED or times out (180 secs). There is no point of setting retry policy as after 180 secs, transaction is marked as TIMEOUTand UI needs to send a new transaction.
      * **Cons** - If subscriber dies, the message will be lost as retry policy is 0.
    - **ACK deadline is 60 secs:** In this scenario, retry policy can be 3 retries with exponential back-off strategy. If broker does not receives ACK after Tx with PENDING state message was sent to the subscriber within 60 secs, then message is put back to the queue and gets visible to the subscribers to consume. After multiple retries, the message will move to DLQ which can be analyzed separately.
      * **Pros -** Even if subscriber dies while processing the message, the message will be visible again after 60 secs for subscribers.
    - **ACK deadline is 3 secs:** With such a short ACK deadline, we are relying on broker retry policy and making subscriber super light weight and not worrying about retry logic on the subscriber side. Messages in PENDING state will be delivered to the subscriber, subscriber makes a node query call with timeout of 2-3 secs and sends ACK if Tx is COMPLETED or FAILED else does nothing. Message becomes visible again after 3 secs for subscriber and another node query call will be made.

What are some good error handling logic?

**Retry Logic with Exponential Backoff**

* **Implement Retry Mechanism**: If a payment fails due to a temporary issue (e.g., network glitch, service downtime), implement an automatic retry logic with exponential backoff (e.g., retrying after 5 seconds, 15 seconds, 30 seconds, etc.).
* **Limit Retries**: Set a maximum number of retries to prevent infinite loops and unnecessary load on your payment gateway or system.
* **Notify User**: Notify users if retries are unsuccessful after several attempts, explaining the issue and suggesting next steps.

**. Grace Period for Pending Transactions**

* **Transaction Timeout**: Define a timeout period for transactions (e.g., 5 minutes). If the payment is not completed within this period, automatically cancel the transaction to prevent it from lingering in a pending state.
* **Inform the User**: Let users know if a transaction is pending or timed out and allow them to retry the payment process without losing their shopping cart or transaction data.

**Scenario: Handling Data Consistency in a Distributed System**

One time I was working with another Engineer, on a distributed system for an e-commerce platform that needs to handle inventory management across multiple geographic locations. The system is designed to allow real-time updates to stock levels whenever a purchase is made.

So 2 types of consistency models were introduced:

* **Sam** believes the system should use **eventual consistency** (with **CQRS** and **event-driven architecture**) to improve scalability and reduce latency in updating inventory.

* **Riley** argues for **strong consistency** in the system, insisting that the inventory should be updated in real-time across all locations to avoid issues like **overselling** products (selling more stock than is available), we don’t want customers to see inaccurate data. These 2 are very important here.

**How They Might Resolve the Disagreement:**

1. **Clarifying the Core Requirements:**
   * Sam and Riley first make sure they understand the **key requirements** for the system:
     + **Performance**: The system needs to handle a large volume of purchases efficiently, especially during peak sales periods.
     + **Accuracy**: Accurate stock levels are important to ensure that customers don’t purchase out-of-stock items.
     + **Scalability**: The system should scale across multiple geographic locations and handle significant load.
     + **User Experience**: Customers should see accurate availability information during checkout to avoid frustration.
2. **Exploring Each Engineer’s Position:**
   * **My perspective**
     + Sam argues that **eventual consistency** can offer **better performance** because it allows the system to prioritize availability and speed, even if the data is not immediately consistent across all nodes.
     + Sam proposes using an **event-driven architecture**, where inventory changes are captured as events (e.g., a new purchase), and updates are propagated asynchronously to different parts of the system.
   * **The other Engineer’s position:**
     + Riley emphasizes that **strong consistency** is critical when dealing with inventory management. Having accurate, synchronized stock levels across all locations is essential for avoiding overselling.
     + Riley also points out that in a distributed system, when the stock levels aren’t immediately updated across all nodes, it can lead to situations where a product shows as available, but is actually out of stock, leading to **overselling**.
     + And overselling is a very critical issue here
3. **Compromise and Hybrid Solution:**
   * Based on the test results, both engineers agree that neither approach is perfect on its own. They decide to implement a **hybrid solution**:
     + Use **strong consistency** for critical parts of the system where accurate stock levels are absolutely necessary (e.g., during checkout to prevent overselling).
     + For less critical operations, like updating inventory levels in the background or handling batch processing, use **eventual consistency** with an event-driven architecture to improve performance and scalability.
   * They also implement **conflict resolution mechanisms** (e.g., retries and compensating transactions) to handle occasional discrepancies in the event-driven model, ensuring that inconsistencies are resolved quickly and don’t cause lasting issues.
4. **Final Decision and Implementation:**
   * The hybrid solution is implemented, and the team ensures that **critical operations** (like checkout and inventory updates) are strongly consistent, while other non-critical updates benefit from the scalability and performance of eventual consistency.
   * They also add **monitoring** to track any inconsistencies in inventory levels and alert the team if conflicts arise that could lead to customer-facing issues.

**Outcome:**

* The hybrid solution works well, allowing the system to scale effectively while maintaining **accurate stock levels** where it matters most. Customer experience improves since they don’t face issues with overselling, and the backend system is more resilient and efficient.
* Sam and Riley are both satisfied with the solution because they were able to collaborate, test their ideas, and come up with a system that balances performance, accuracy, and scalability.

**We didn’t see a lot of issues here** so it was a good discussion in general very good. And this was quite important.