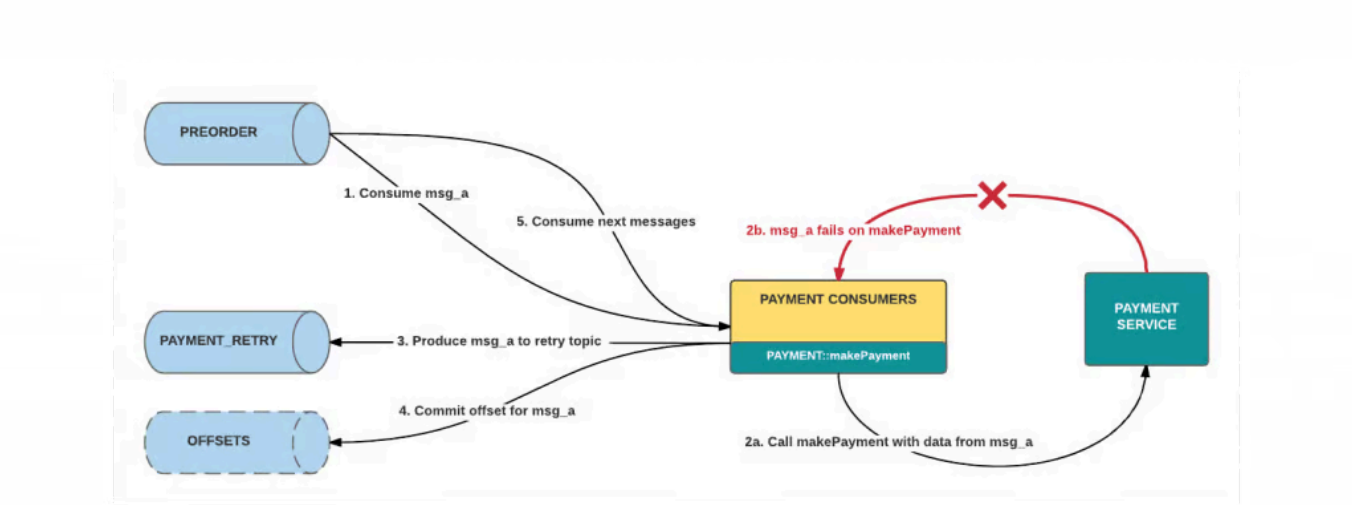
Uber Building Reliable Reprocessing and Dead Letter Queues with Apache Kafka

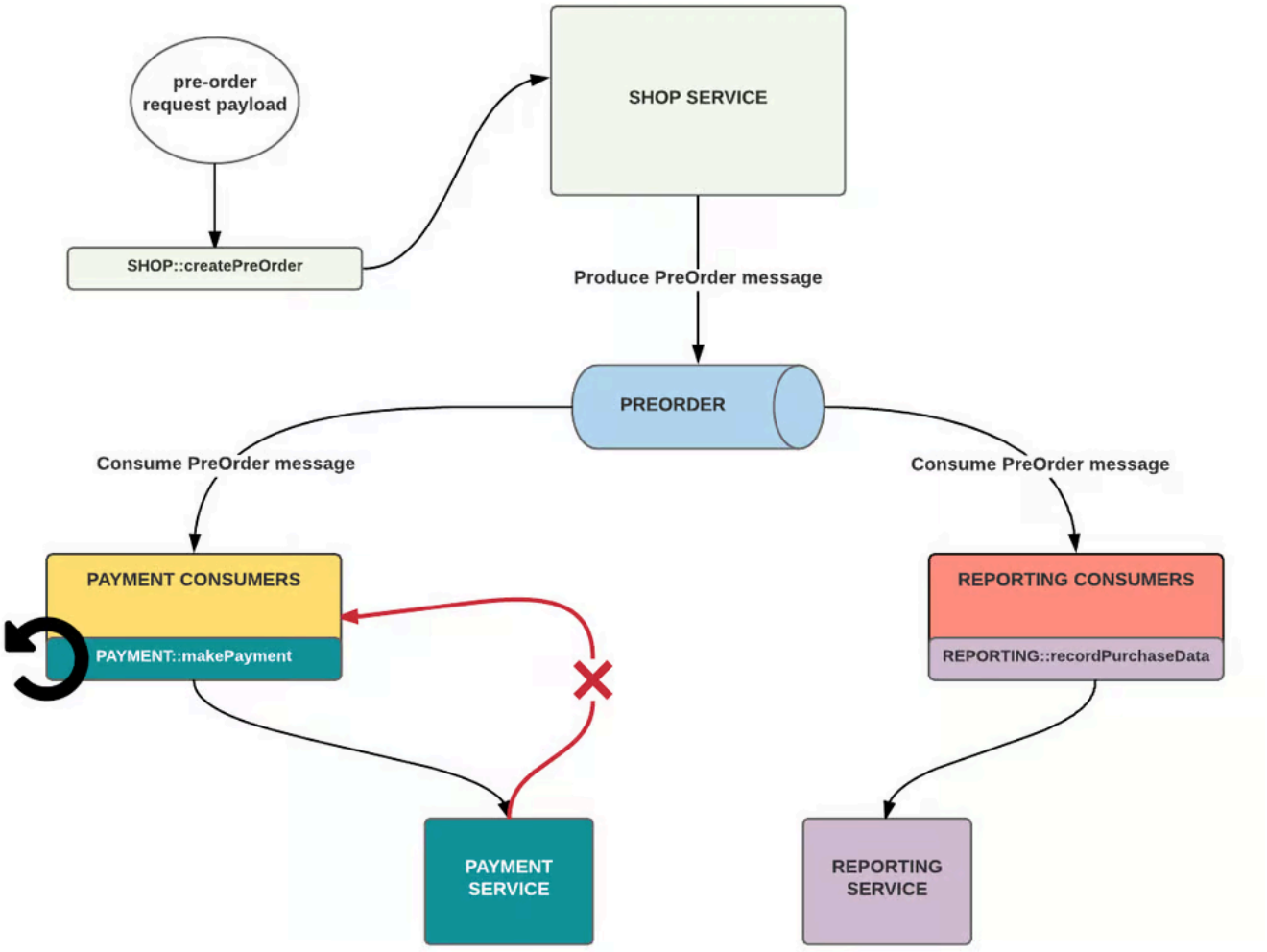


- in distributed systems, retries are inevitable.
- From network error to replication issue and even outage in downstream services
- At Uber, our system must be fault-talerant and uncompromising when it comes to failing intelligently.
- Uber extended Kafka's role in existing event-driven architecture by non-blocking request reprocessing and dead letter queues to achieve decoupled, observable error-handling without disrupting real-time traffic.

Working in an event-drien architecture:

- The backend of driver injury protection sits in a kafka msging architcture that runs through a java service hooked into multiple dependencies within Uber's system.
- In this model, we want to both
- 1. make a payment,
- 2. create a separate record capturing data for each product pre-order per user to generate real-time product analytics.
- This is analogous to how a single driver injury protection trip premium processed by our program's backend architecture has both an actual charge component and a separate record created for reporting purposes.
- In our example, each function is made available via API of its respective service.

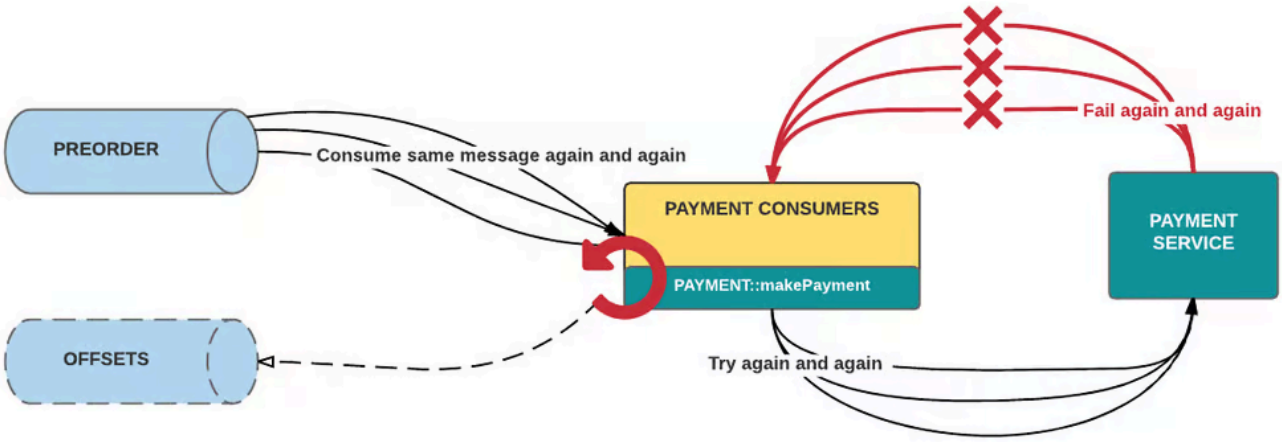
- below Figure, models them within 2 corresponding consumer groups, both subscribed to the same channel of pre-order events (kafka topic, PreOrder)



The problem with simple retries:

- While retrying at client level with a feedback cycle can be useful, retries in large-scale systems may still need to:
- **Clogged batch processing:**
  - when we required to process a large no. of msgs in real-time, repeatedly failed msgs can clog batch processing.
  - worst offenders consistently exceed retry limit, which also means that they take longest and use most resources.
  - without a success response, kafka consumer will not commit a new offset and batches with these bad msgs would be blocked, as they are reconsumed again and again.
- **Difficulty Retrieving metadata:**

- It can be cumbersome to obtain metadata on retries, such as timestamps and nth retry.

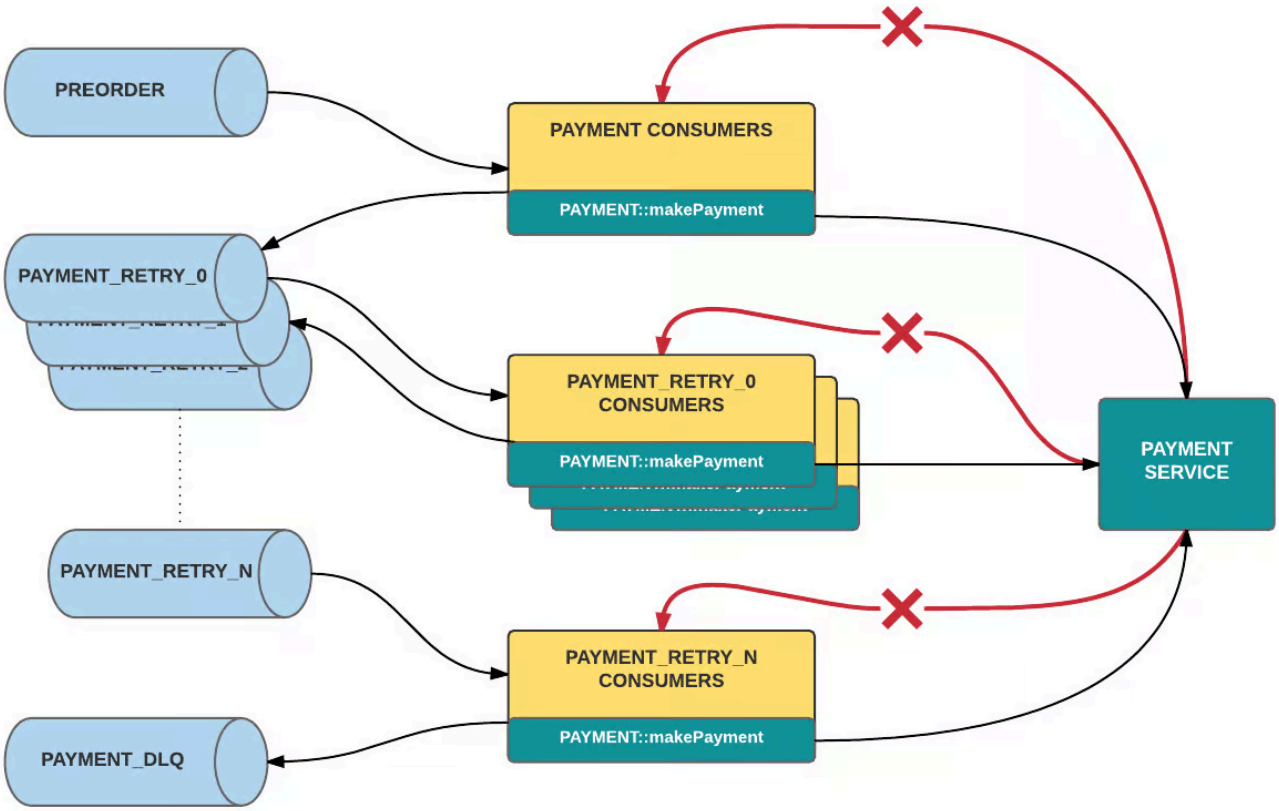


- If there is a breaking change in the downstream Payment Service,
- for instance, unexpected charge denial for previously valid pre-orders, then these messages would fail all retries.
- The consumer that received that specific message does not commit the message's offset, meaning that this message would be consumed again and again at the expense of new messages that are arriving in the channel and now must wait to be read.
- if requests continue to fail retry after retry, we want to collect these failures in a DLQ for visibility and diagnosis.
- A DLQ should allow listing for viewing contents of queue, purging for clearing those contents, and merging for reprocessing dead-lettered msgs.
- we needed a retry strategy that would reliably and scalably afford us these capabilities.

Processing in separate queues:

- To address the problem of blocked batches, we set up a distinct retry queue using a separately defined kafka topic.
- Under this paradigm, when a consumer handler returns a failed response for a given msg after a certain no. of retries, consumer publishes that msg to its corresponding retry topic.
- the handler then returns true to original consumer, which commits its offset.
- Consumer success is redefined from a successful handler response, meaning zero failure, to establishment of a conclusive result for consumed msg, which is either expected response or its placement elsewhere to be separately handled.
- retrying requests in this type of system is very straightforward.
- As with the main processing flow, a separate group of retry consumers will read off their corresponding retry queue.

- These consumers behave like those in the original architecture, except that they consume from a different Kafka topic.
- When the handler of a particular topic returns an error response for a given message, it will publish that message to the next retry topic below it.



- it is imp to simply re-attempt failed requests immediately one after other.
- doing so will amplify no. of calls, essentially spamming bad requests.
- rather each subsequent level of retry consumers can enforce a processing delay,
- this mechanism follows a leaky bucket pattern where flow rate is expressed by blocking nature of delayed msg consumption within retry queues.
- consequently, our queues are not so much retry queues as they're delayed processing queues, where re-execution of error cases is our best-effort delivery: handler invocation will occur at least after the configured timeout but possibly later.

What we gain with queue-based reprocessing:

- **Unblocked batch processing:**
  - Failed msgs enter their own designated channels, enabling successes in the same batch to proceed instead of requiring them to be reprocessed along with failures.
  - Thus, consumption of incoming requests moves forward unblocked, achieving higher real-time throughput.

- **Decoupling:**
  - independent work streams that operate on same event each have their own consumer flows, with separate reprocessing and DLQ.
  - Failure in one dependency doesn't require retrying that particular msg for others that had succeeded.
  - For example, in Figure 1, if reporting had errored out, but payment had succeeded, only the former would need to be re-attempted and potentially dead-lettered.
- **Configurability:**
  - creating new topics incurs practically no overhead, and msg produced to these topics can abide by same schema.
  - The original processing, along with each of the retry channels, can be managed under an easily-written, higher-level consumer class, governed by config when it comes to which topic name to read to publish to (in event of failure) as well as length of enforced delay before executing an instance's handler.
  - We can also differentiate treatment of different types of errors, allowing cases such as network flakiness to be re-attempted, whill NPE and other code bugs should go straight into DLQ bcz retries wouldn't fix them.
- **Observability**:
  - Segmentation of msg processing into different topics facilitates easy tracing of an errored msg's path, when and how many times msg has been retried, and exact properties of its payload.
  - Monitoring the rate of production into original processing topic compared to those of reprocessing topic and DLQ can inform threshold for automated alerts and tracking real-service uptime.
- **Flexbility:**
  - Though Kafka itself is written in Scala & Java, Kafka supports client libraries in several langs.
  - Kafka msg formatting with a serialization framework like Avro supports evolvable schemas, in the event of that our data model needs to be updated, minimal rejiggering is required to reflect this change.