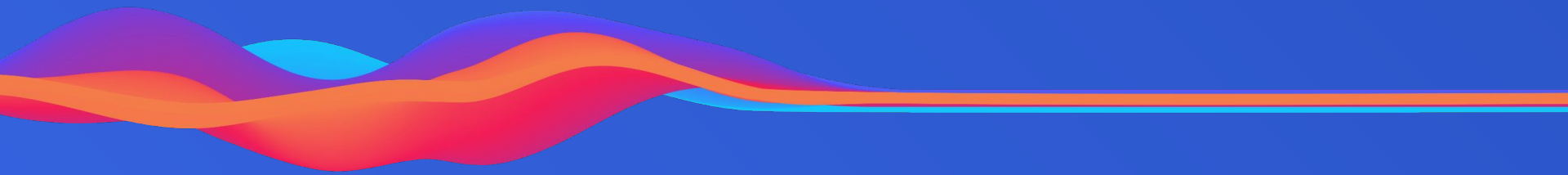
REVINATE

Using Kafka Streams to Build a Data Pipeline for the Hospitality Industry

September 17, 2024



REVINATE CDP

COLLECT

Multi source data connectivity & ingestion

- **PMS** — Property Management System
- **Folio** — (F&B, Golf, Spa, Ski)
- **Booking Engine** — Cart Abandonment / Lead Data
- **Satisfaction Surveys** — Guest Feedback
- **Restaurant, Spa, Ski, Golf** — Ancillary Data
- **Voice Channel** — Lead Management
- **Website** — Signups, Newsletters
- **Guest Preferences** — Interest data
- **API In** — Loyalty, etc

Systems & Data Integrations
Turnkey data activation and data flow management running on secure AWS Cloud

UNIFY / MERGE

Data processing engine

Advanced Profile Synthesis
Merge, dedupe, unify, organize and secure clean guest data

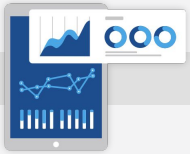
STORE

Store and search data

#1 Hospitality CRM
Storage and visualization of all hotel and brand Rich Guest Profiles

TARGET

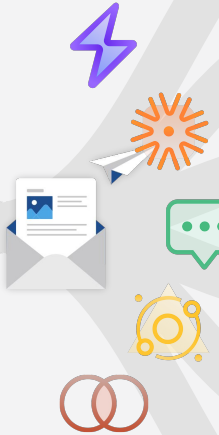
Define rules & create segments



Advanced Data Segmentation Set up segments and audiences for automated personalization

ACTIVATE

Connect & send data to external systems



- EMAIL
- VOICE
- MESSAGING
- SMS
- WEB
- API

Omnichannel Activation
Powering personalized guest lifecycle communications on all your direct channels natively

REVINATE

450+

REVINATORS WORLDWIDE (US, EMEA, APAC)

$150M+

INVESTED IN HOTELIER SUCCESS, BACKED BY TOP GLOBAL INVESTORS

#1 CDP/CRM

THE ULTIMATE OMNI-CHANNEL DIRECT BOOKING PLATFORM FOR HOTELS

800M+

RICH GUEST PROFILES UNDER MANAGEMENT

REVINATE

60 NPS

HOTELIERS LOVE OUR PURPOSE BUILT PLATFORM & EXPERT SERVICE

$12.8B+

LIFETIME DIRECT REVENUES GENERATED FOR REVINATE CUSTOMERS

6 Years Ago

- Monolith
- Java with Spring Boot
- Self-managed Kafka
- Spring Kafka
- Limited Kafka knowledge
- RabbitMQ

Today

- Microservices
- Vanilla Java
- Hosted Kafka
- Kafka Streams
- More Kafka knowledge

Data

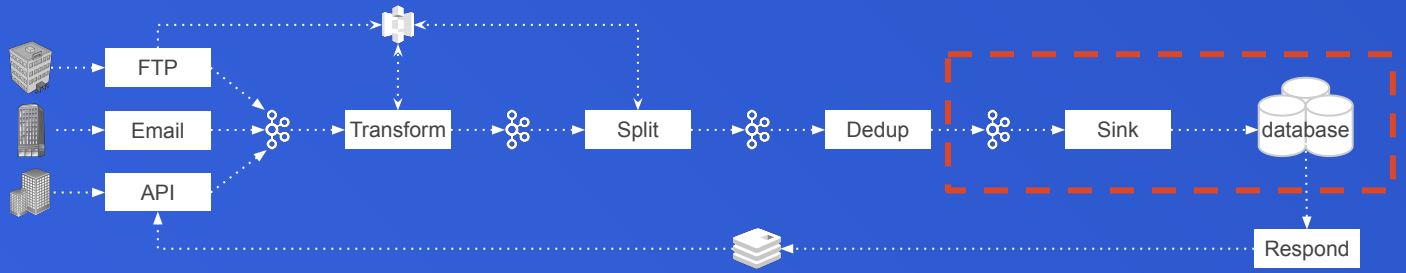
- Accounts (hotels)
- Reservations (room, spa, golf, restaurant)
- Guests (profiles, preferences, feedback)

Processing

- Multiple integration methods (webhooks, APIs, FTP, email)
- Realtime and batch workloads (duplicate source system messages are common)
- At-least-once
- Atomic messages, but with complex schema
- Order isn’t always reliable from source systems

REVINATE

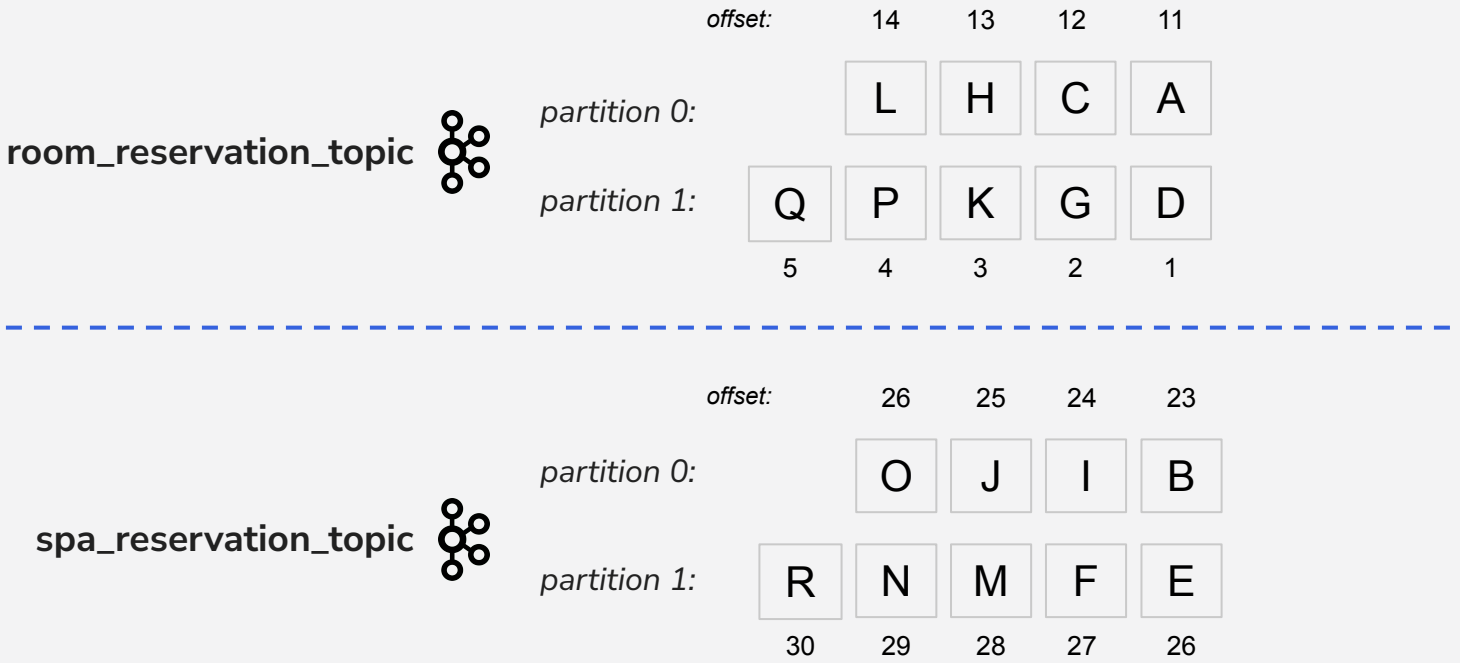
1. Supporting Diverse Message Types



"The common wisdom seems to be: put all events of the same type in the same topic, and use different topics for different event types" [1]

Source: [1] https://www.confluent.io/en-gb/blog/put-several-event-types-kafka-topic/

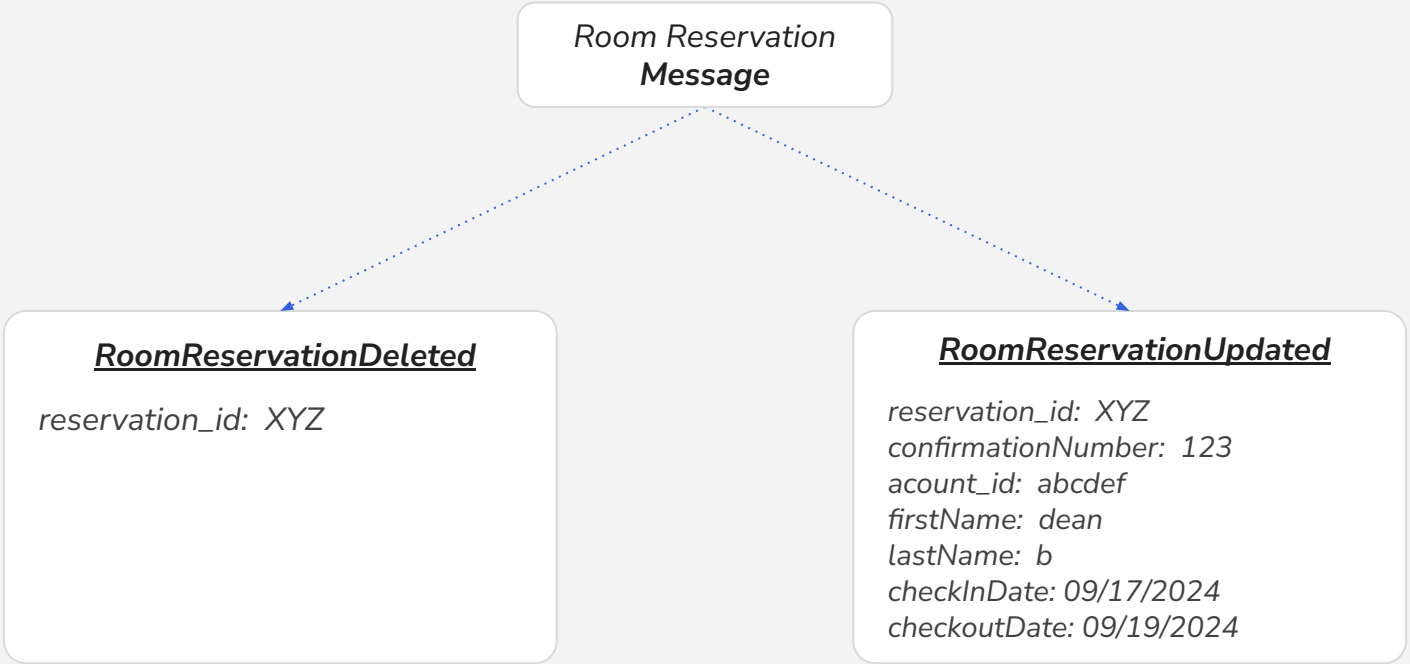
"Kafka only provides a total order over records <u>within</u> a partition, not between different partitions in a topic" [1]



Source: [1] https://kafka.apache.org/24/documentation.html#intro_consumers

Problem 1

Messages with different types can’t be ordered because they are on different topics

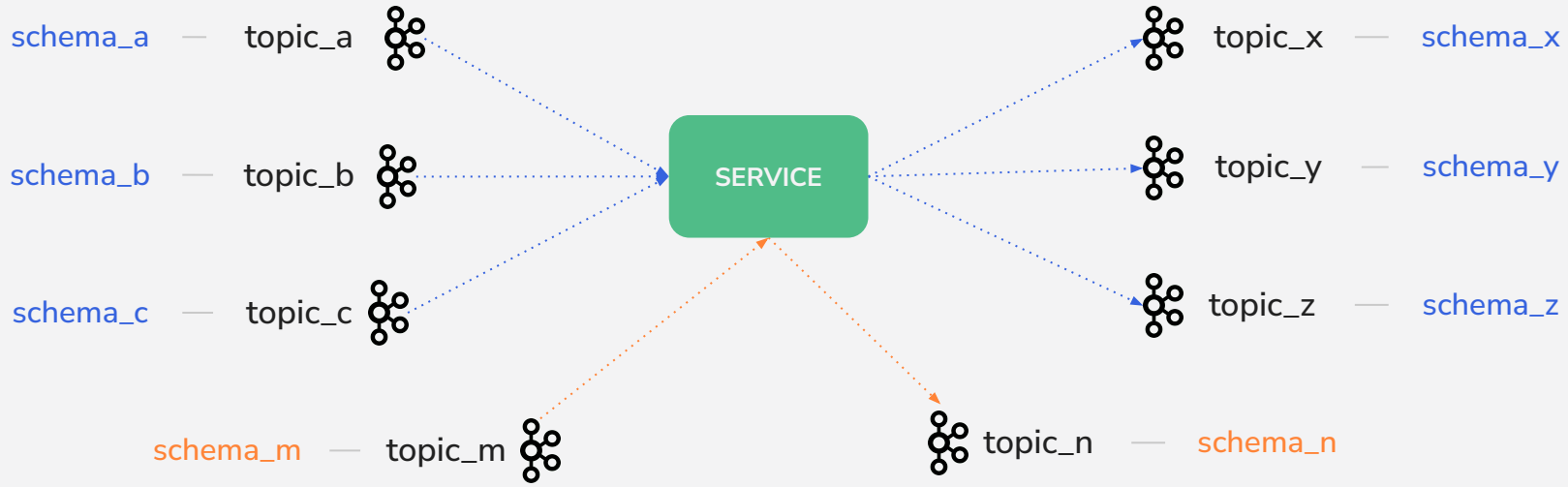


Problem 2

Multiple message types can be a challenge to maintain when they share common components

Problem 3

- Services may need to support multiple serdes for input and output topics
- Adding new schemas requires the addition of new topics

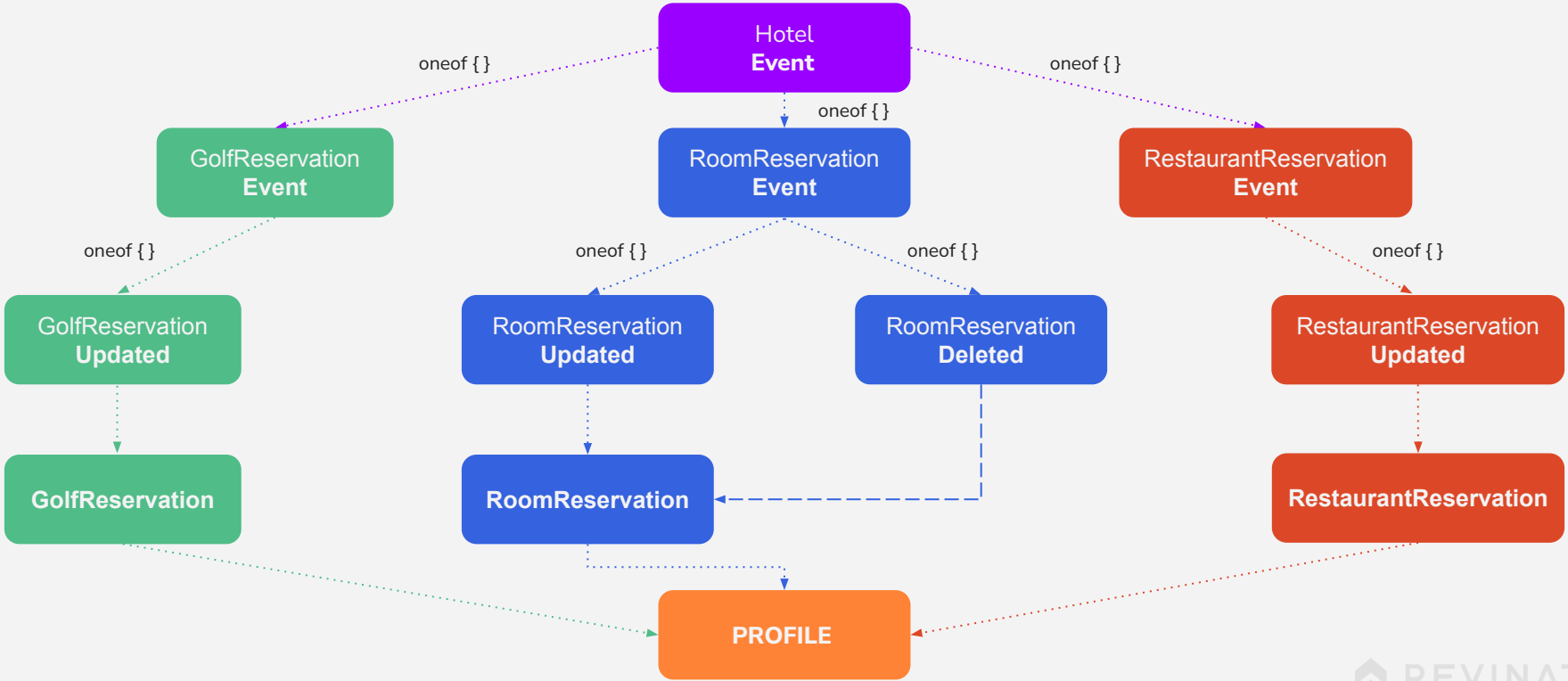


Goal

- New message types can be supported without creating new topics
- Message order can be guaranteed *across* message types
- Consistent processing of shared components
- One serde for all

Solution

Create a generalized schema that can be used for all topics

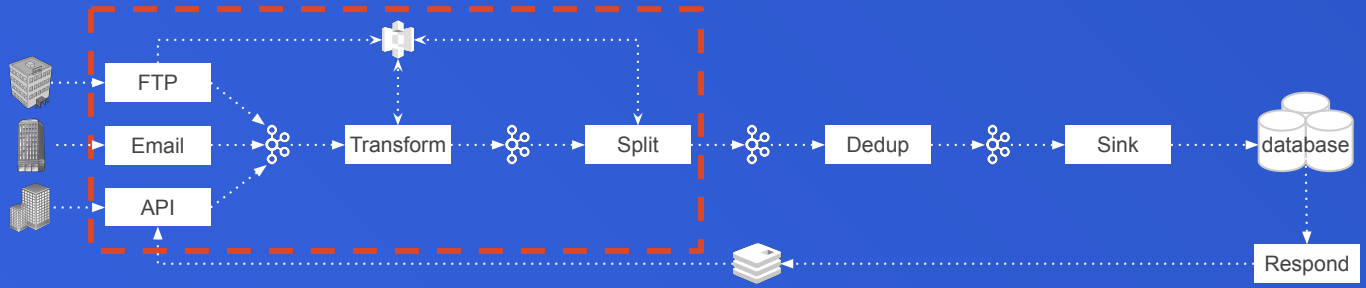


Disadvantage

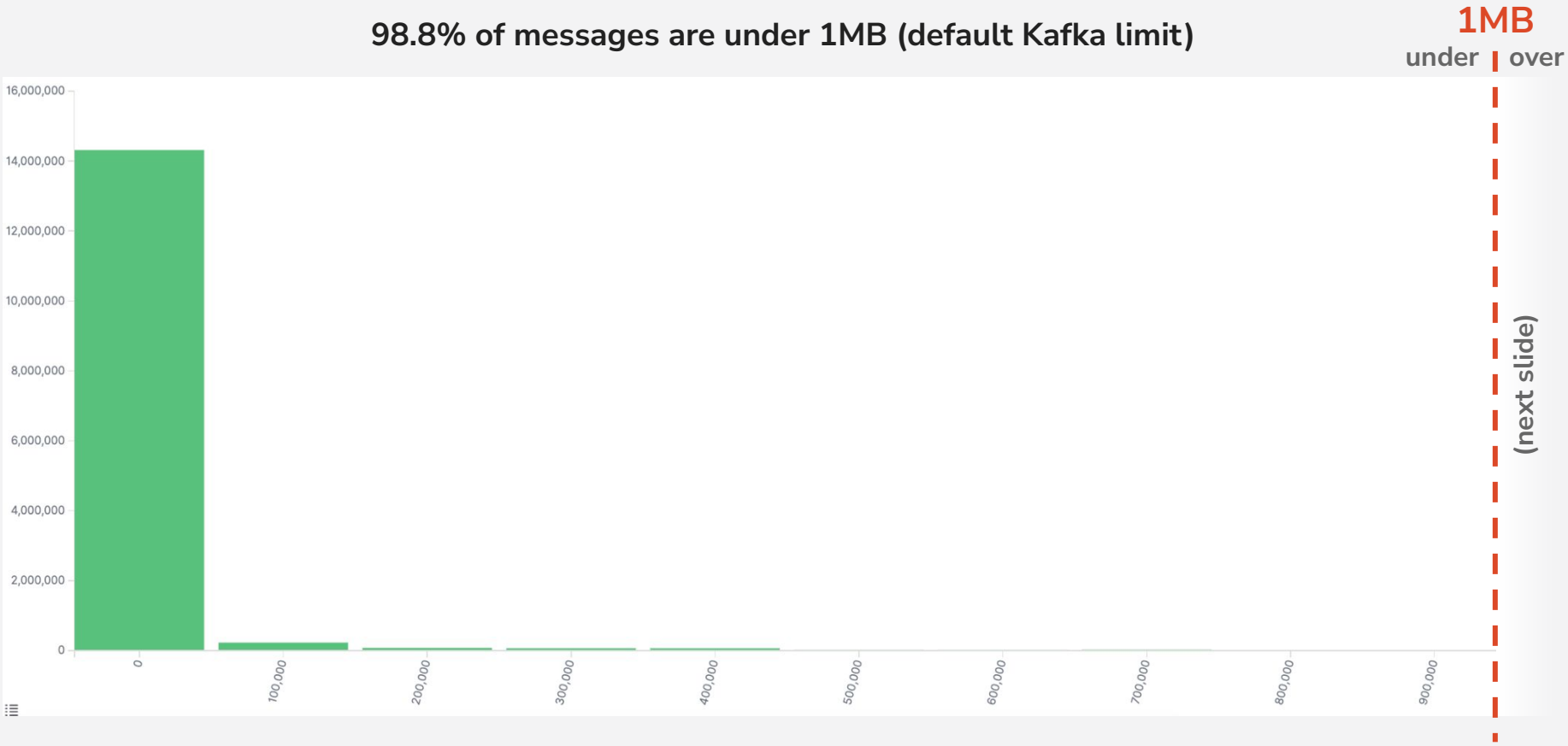
```
void process(HotelEvent hotelEvent) {
     switch(hotelEvent.eventType()) {
         case roomReservation -> {
             switch (roomRervation.getType()) {
                 case roomReservationUpdate -> {...}
                 case roomReservationDelete -> {...}
             }
         }
         case profile -> {...}
         case golfReservation -> {...}
         case spaReservation -> {...}
     }
}
```

REVINATE

2. Handling Large Messages



98.8% of messages are under 1MB (default Kafka limit)



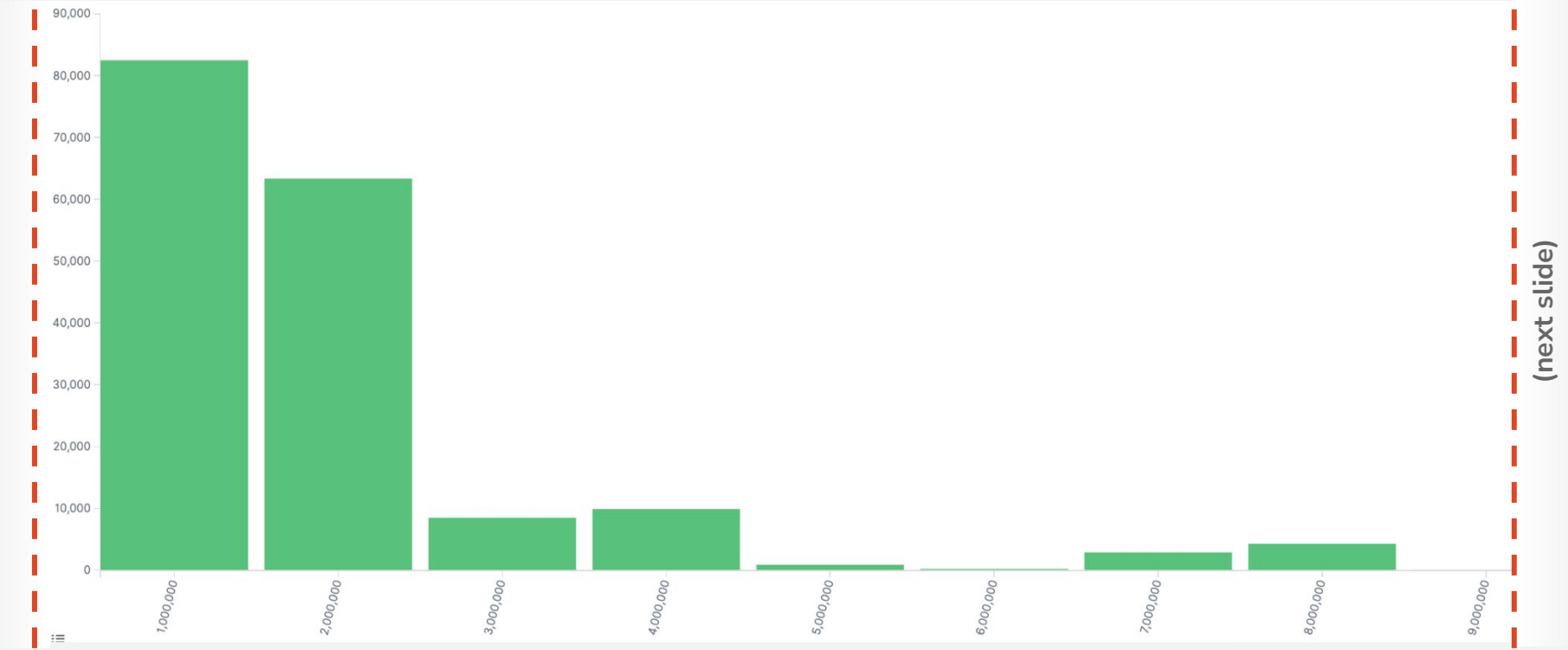
Source: *Revinate logging infrastructure (sample of 15 million messages)*

Only 1.2% of messages are between 1MB and 10MB...but we must support them

1MB

10MB

under | over

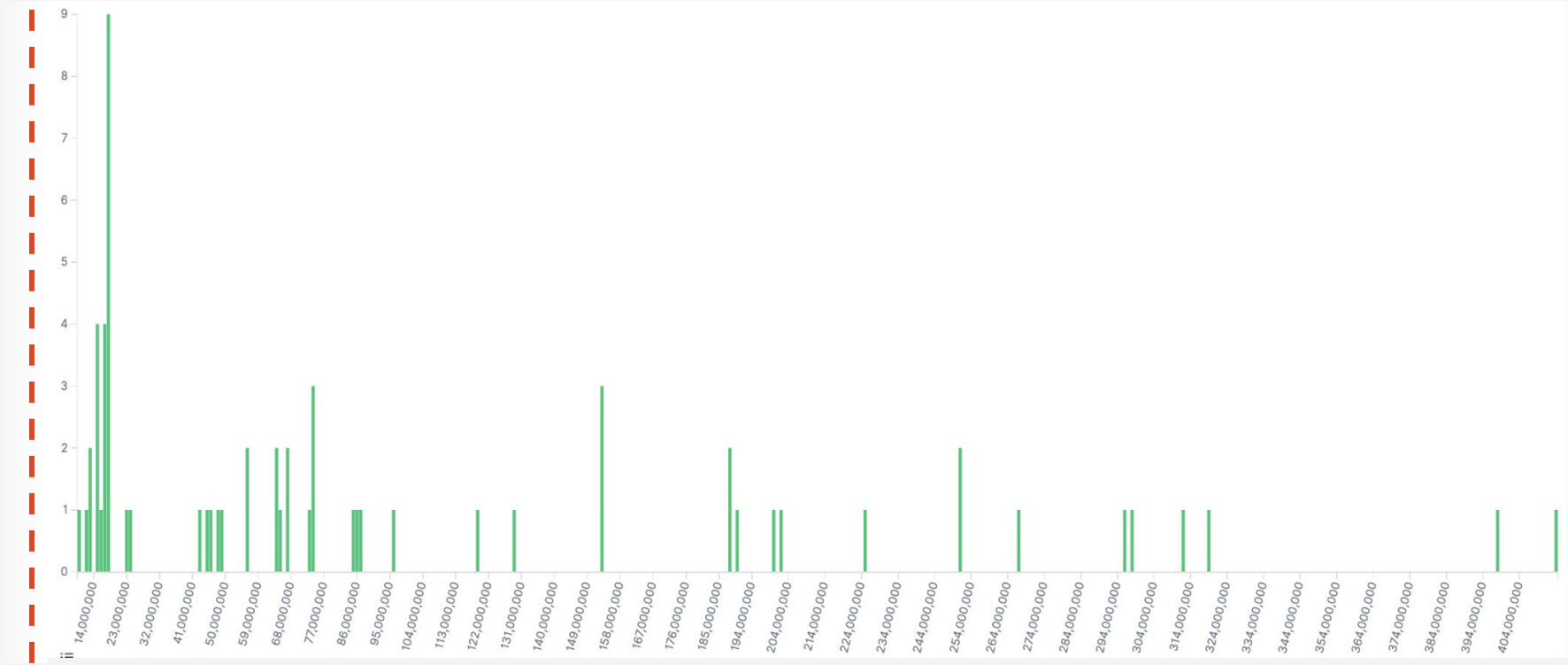


(next slide)

Source: *Revinate logging infrastructure (sample of 15 million messages)*

10MB

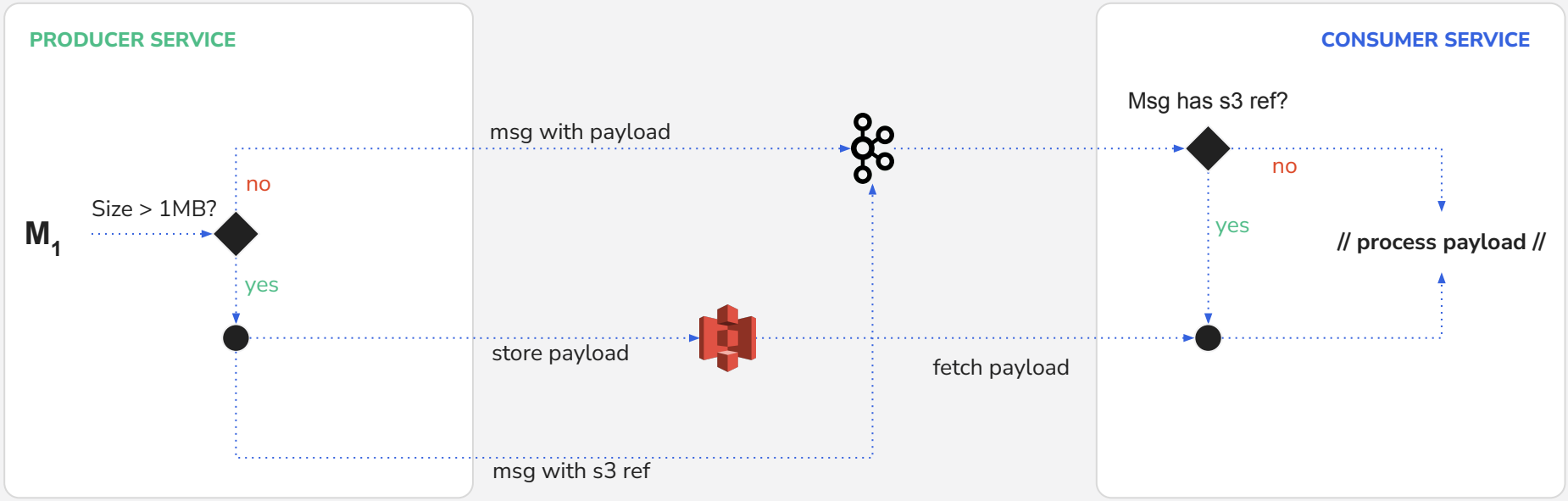
Barely any messages (0.005%) are over 10MB...but we must support them, too



Source: *Revinate logging infrastructure (sample of 15 million messages)*

Solution

Use the Claim-Check pattern [1]



Source: [1] https://learn.microsoft.com/en-us/azure/architecture/patterns/claim-check

Some things to consider

- Compression will affect message size (message.max.bytes)

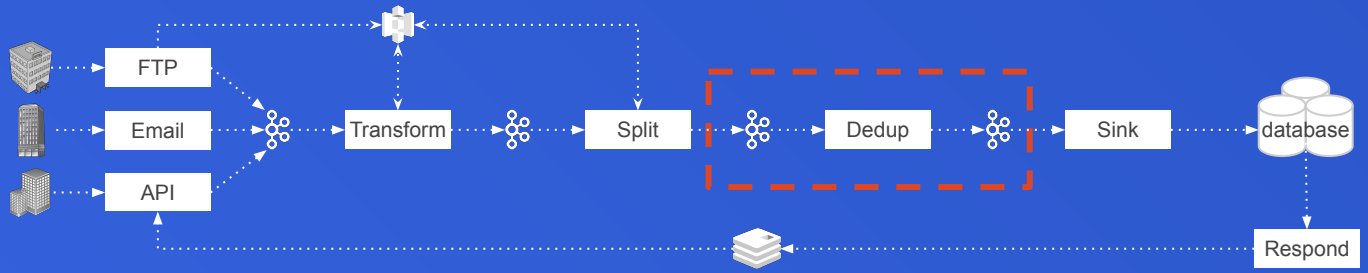
  "As with all byte sizes specified on the broker, this configuration deals with compressed message size, which means that producers can send messages that are much larger than this value uncompressed, provided they compress down to under the configured message.max.bytes size."[1]

- Message schema changes
- S3 message payload retention
- Alternative approaches
  - Custom consumer/producer
  - *Likafka* [2]

Source: [1] Kafka: Definitive Guide (by Neha Narkhede, Gwen Shapira ,and Todd Palino); *[2] https://github.com/becketgin/likafka-clients*

REVINATE

3. Deduplicating Message Payloads



Problem

- Message stream contains messages that are ***duplicates*** of messages that were previously processed
  - May result in erroneous duplicate processing errors
  - Unnecessary processing (waste of resources)

Cause

- 3rd party processing beyond our control
- Intentional data replay

Goal

- Identify duplicate messages
  - Redirect duplicates to a duplicate topic
  - Ignore duplicates

A message is a duplicate when (significant) fields are equivalent to fields of a previously processed message.

Msg_1

```
hotelEvent {
  received: 2024-08-21T23:57:29Z
  account_id: “rev-099987”
  roomReservation {
     update {
        name: “dean”
        checkinDate: 2024-09-17
        checkoutDate: 2024-09-21
        reservation_id: “123456”
     }
  }
}
```

Msg_2

```
hotelEvent {
  received: 2024-08-22T21:43:12Z
  account_id: “rev-099987”
  roomReservation {
     update {
        name: “dean”
        checkinDate: 2024-09-17
        checkoutDate: 2024-09-21
        reservation_id: “123456”
     }
  }
}
```

Remove non-significant fields, hash the result, and add to cache table

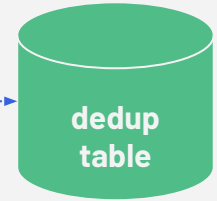
Msg_1

```
hotelEvent {
  received: 2024-08-21T23:57:29Z
  account_id: "rev-099987"
  roomReservation {
     update {
        firstName: "dean"
        lastName: "burson"
        checkinDate: 2024-09-17
        checkoutDate: 2024-09-21
        reservation_id: "123456"
     }
  }
}
```

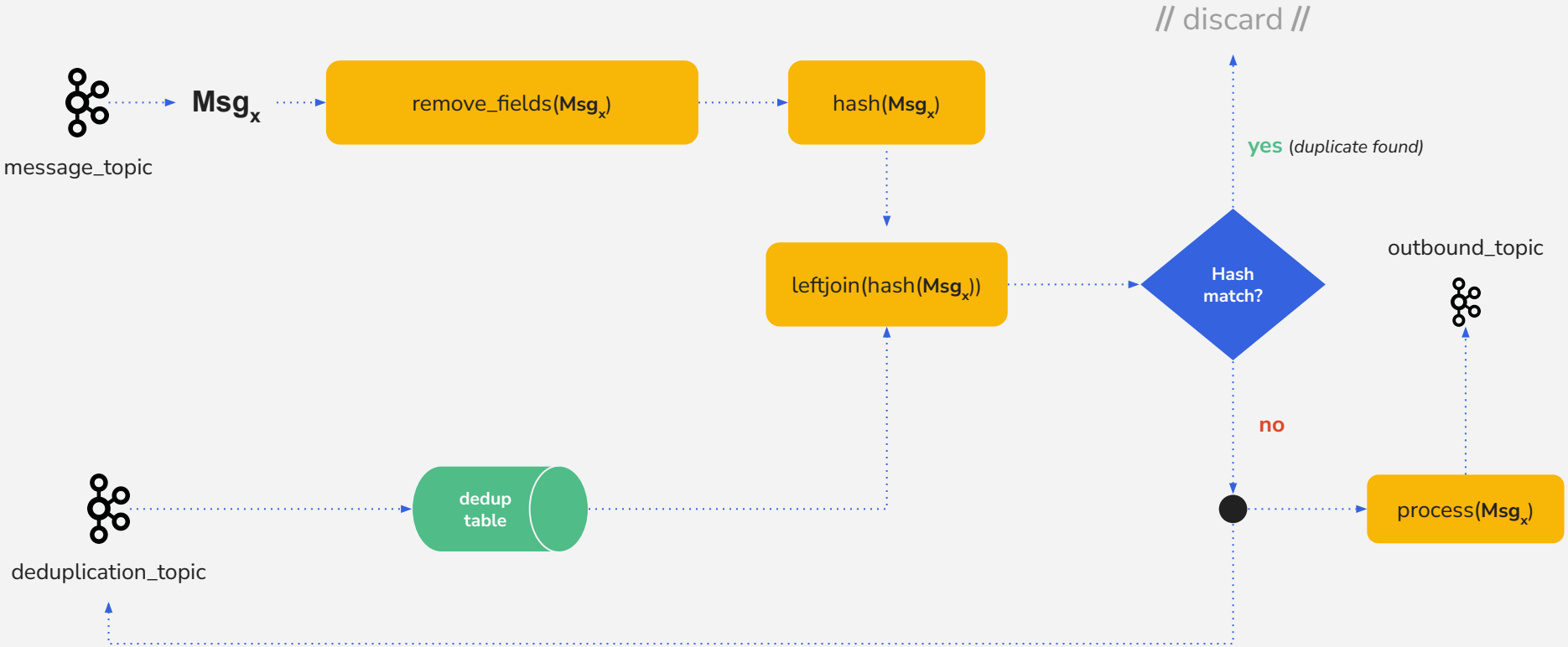
remove_fields(Msg_1)

```
hotelEvent {
  account_id: "rev-099987"
  roomReservation {
     update {
        firstName: "dean"
        lastName: "burson"
        checkinDate: 2024-09-17
        checkoutDate: 2024-09-21
        reservation_id: "123456"
     }
  }
}
```

hash(remove_fields(Msg_1))



Solution

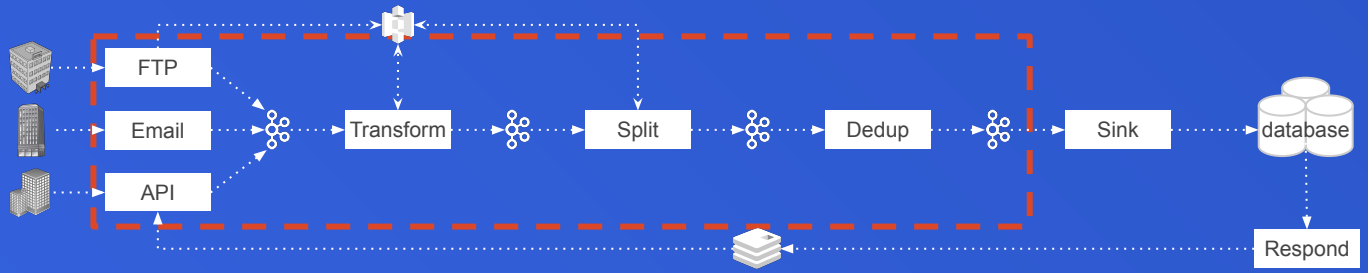

// discard //
message_topic
Msg$_x$
remove_fields(Msg$_x$)
hash(Msg$_x$)
yes (duplicate found)
outbound_topic
leftjoin(hash(Msg$_x$))
Hash match?
no
dedup table
deduplication_topic
process(Msg$_x$)

Some things to consider

- Dedup table structure
  - Message hash (Long as Key)
  - Current timestamp (for table maintenance)
- How long are messages retained in dedup table?
  - Hash cache may grow forever
  - How is time-to-live function implemented on dedup table
- Hash algorithm
- Will duplicate messages have same key?
  - If so, use partition KTable for dudup
  - If not, use GlobalKTable for dedup

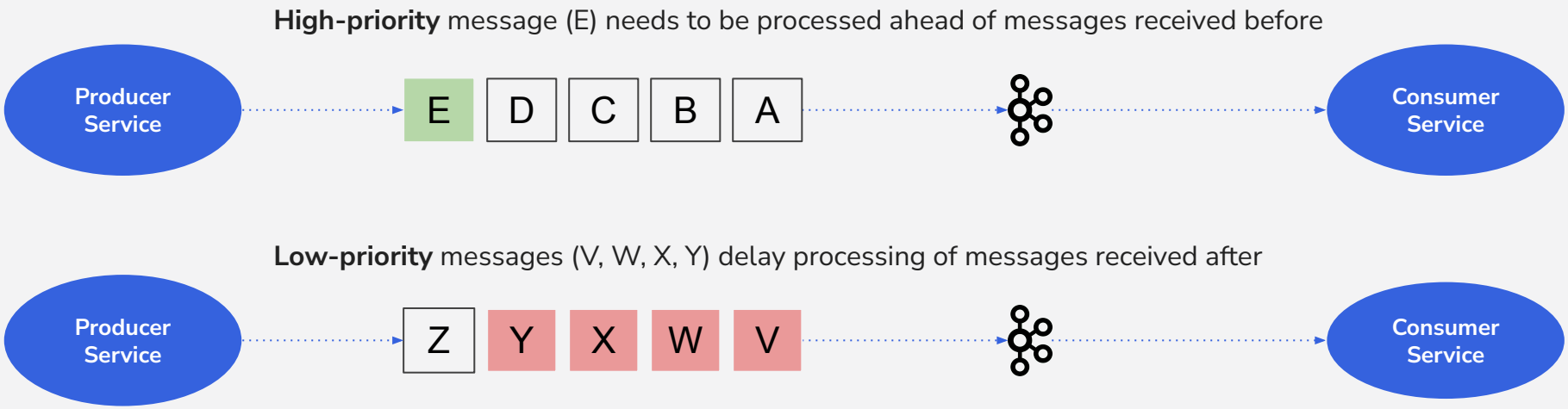
REVINATE

4. Prioritizing Messages



Problem

Topic contains messages whose processing priority is not reflected in partition order or by timestamp

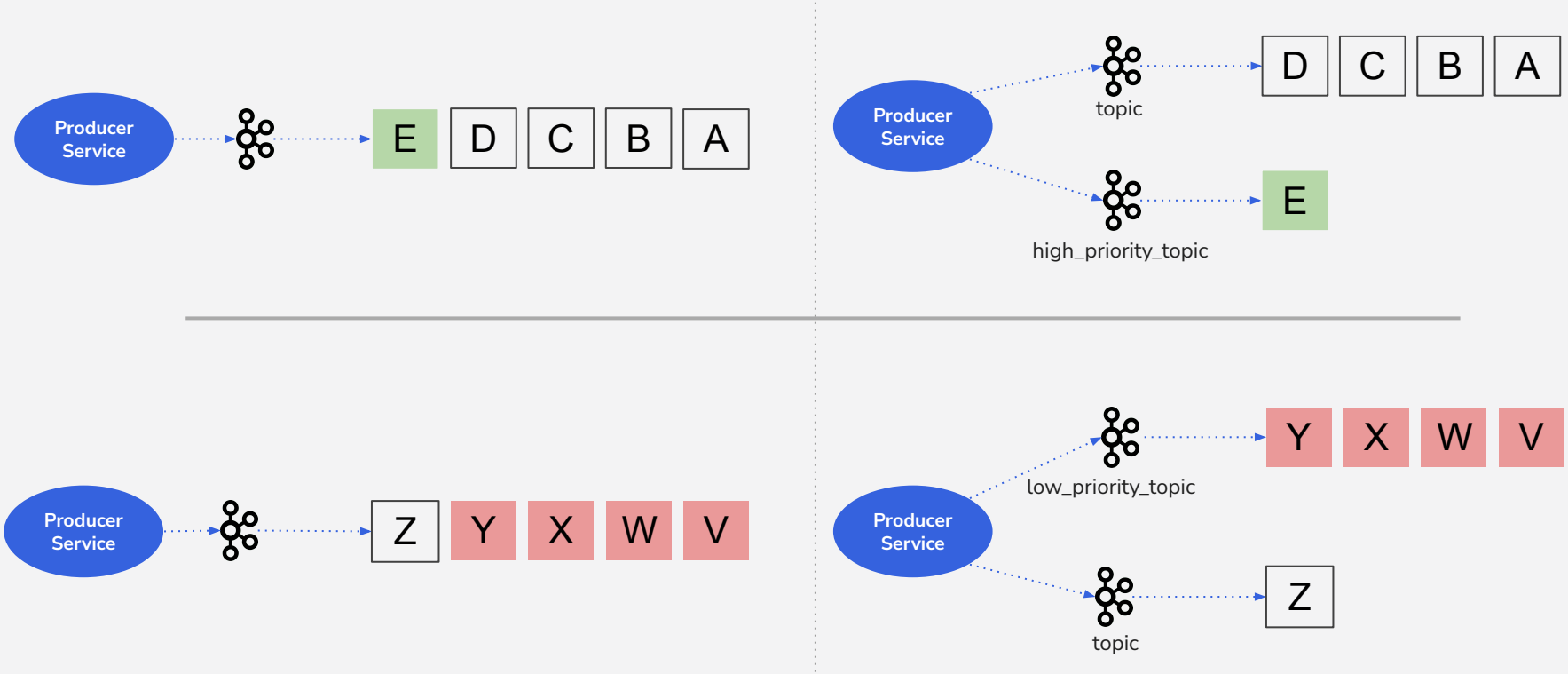


Goal

- Identify high-priority and low-priority messages
- Ensure that high-priority messages are processed as soon as possible
- Ensure that low-priority message do not block or impede processing of higher priority message

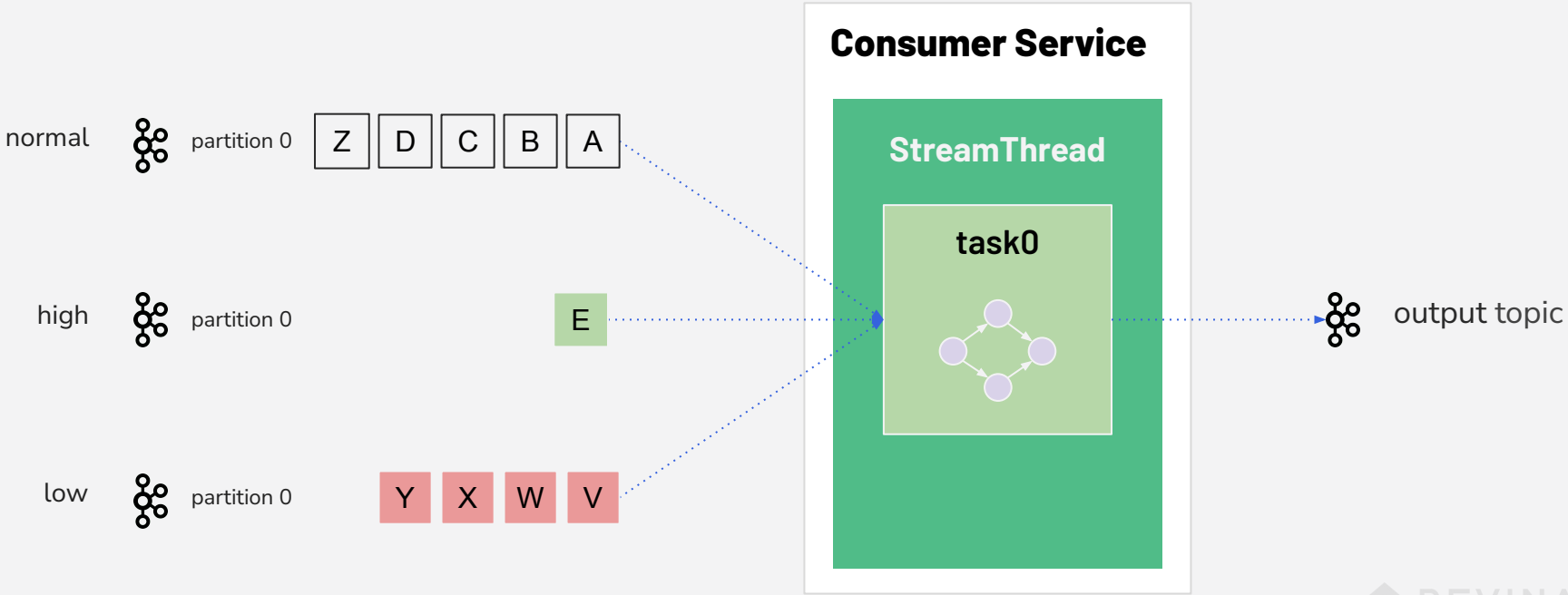
Steps

- Sort high/low priority messages into high/low priority topics
- Consume high/low priority topics using sub-topologies


Producer Service
E
D
C
B
A
Producer Service
topic
D
C
B
A
high_priority_topic
E
Producer Service
Z
Y
X
W
V
Producer Service
low_priority_topic
Y
X
W
V
topic
Z

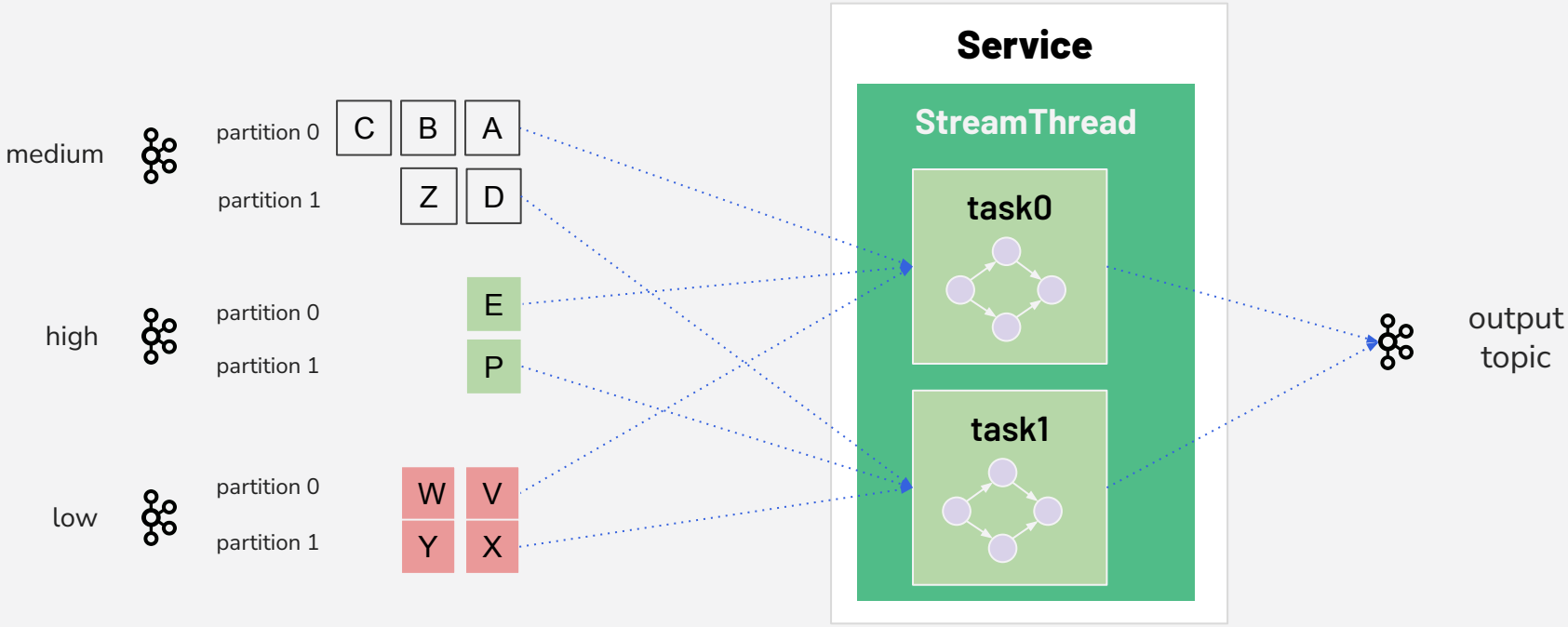
Sorting high/low priority messages ensures that messages are processed with minimal blocking (round-robin partition consumption), BUT...

"In Kafka Streams, the earliest timestamp across all partitions is chosen first for processing" [1]



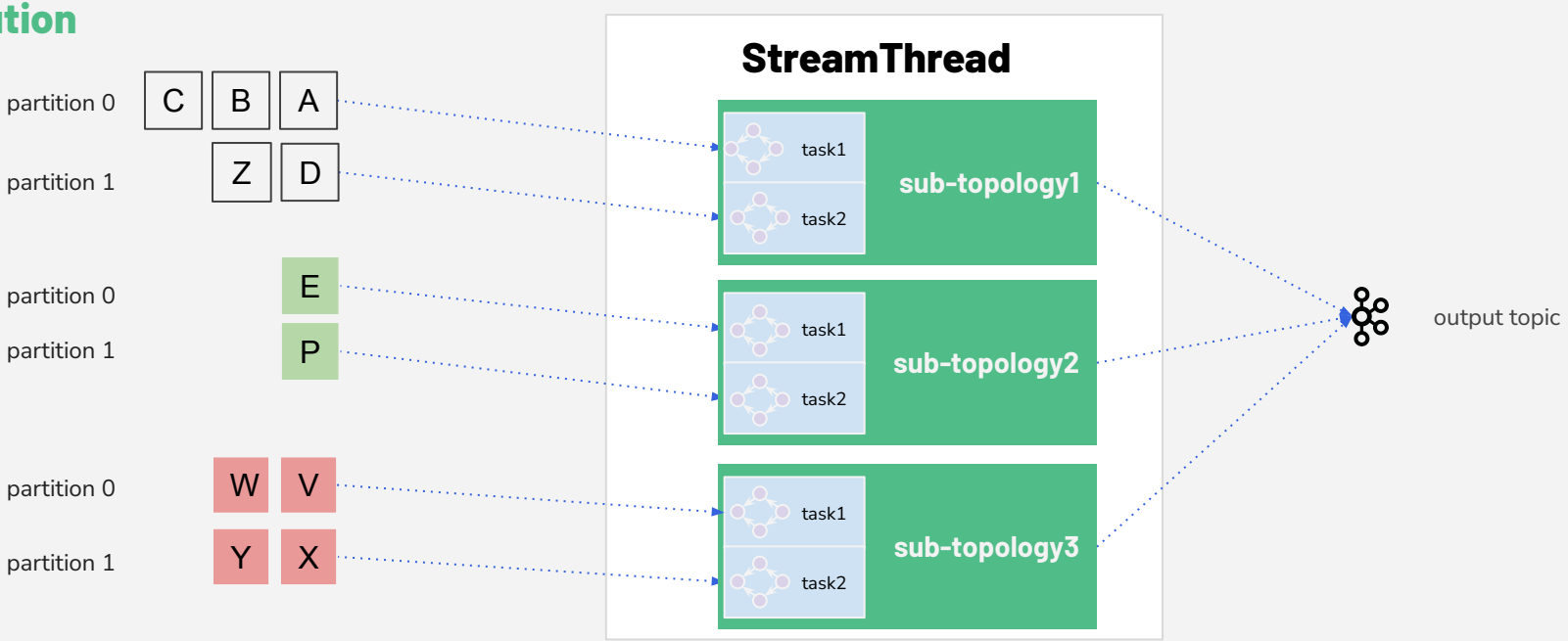
Source: [1] https://developer.confluent.io/courses/kafka-streams/time-concepts/

Additional partitions and tasks may improve throughput



"A sub-topology is a set of processors, that are all transitively connected as parent/child or via state stores in the topology, so different sub-topologies exchange data via topics and don't share any state stores. Each task may instantiate only one such sub-topology for processing. This further scales out the computational workload to multiple tasks." [1]

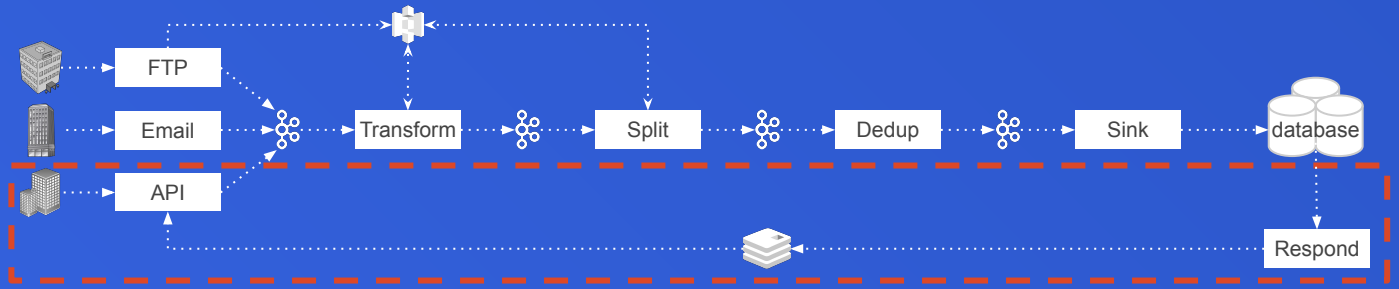
Solution



Source: [1] https://www.confluent.io/en-gb/blog/put-several-event-types-kafka-topic/

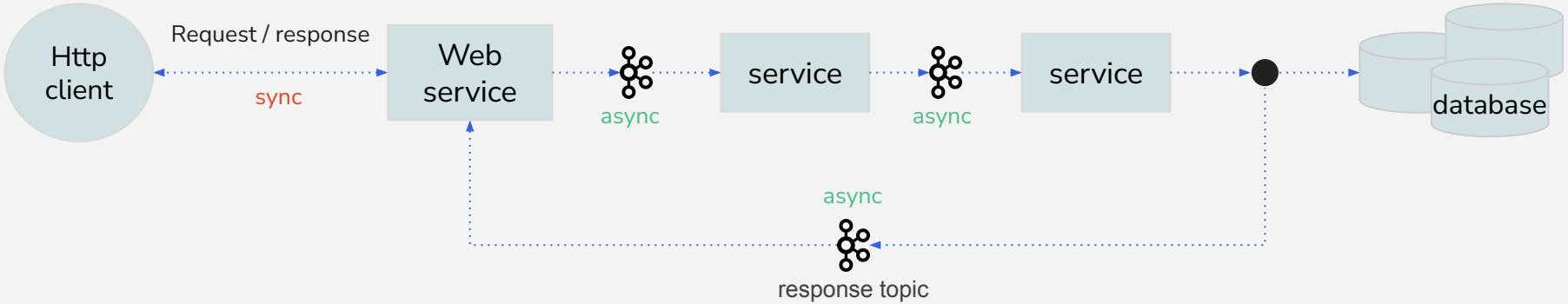
REVINATE

5. Broadcasting Events



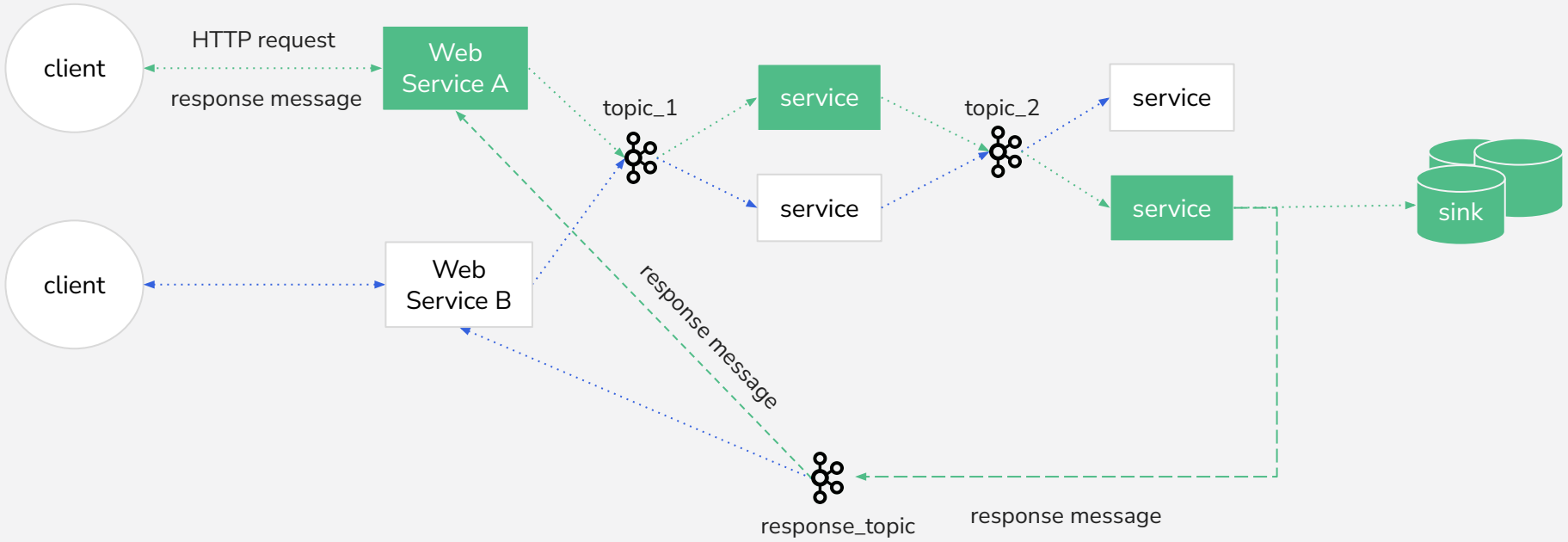
Problem 1

Kafka asynchronous message event stream is not suited to synchronous message handling (e.g. HTTP)



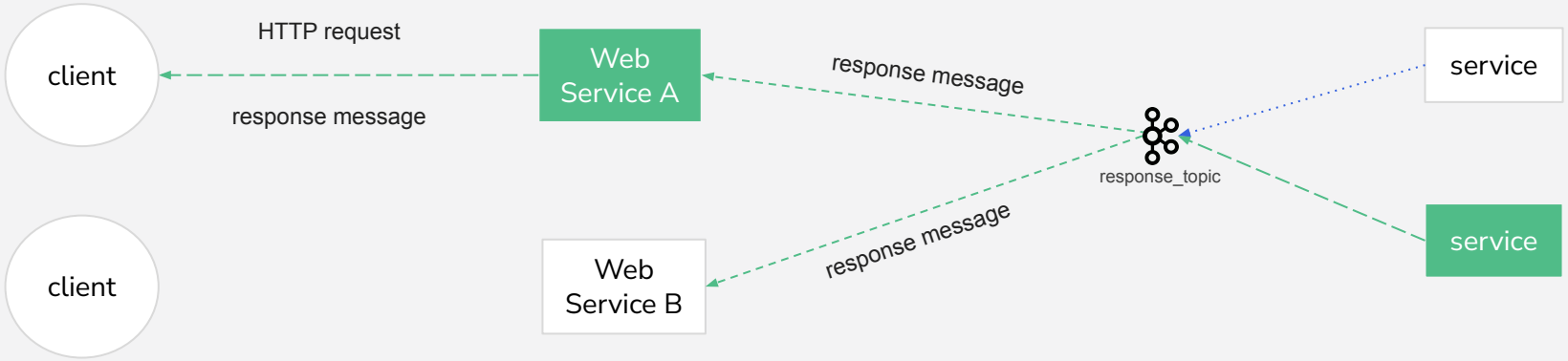
Problem 2

Request and response traverse an unpredictable path across service nodes



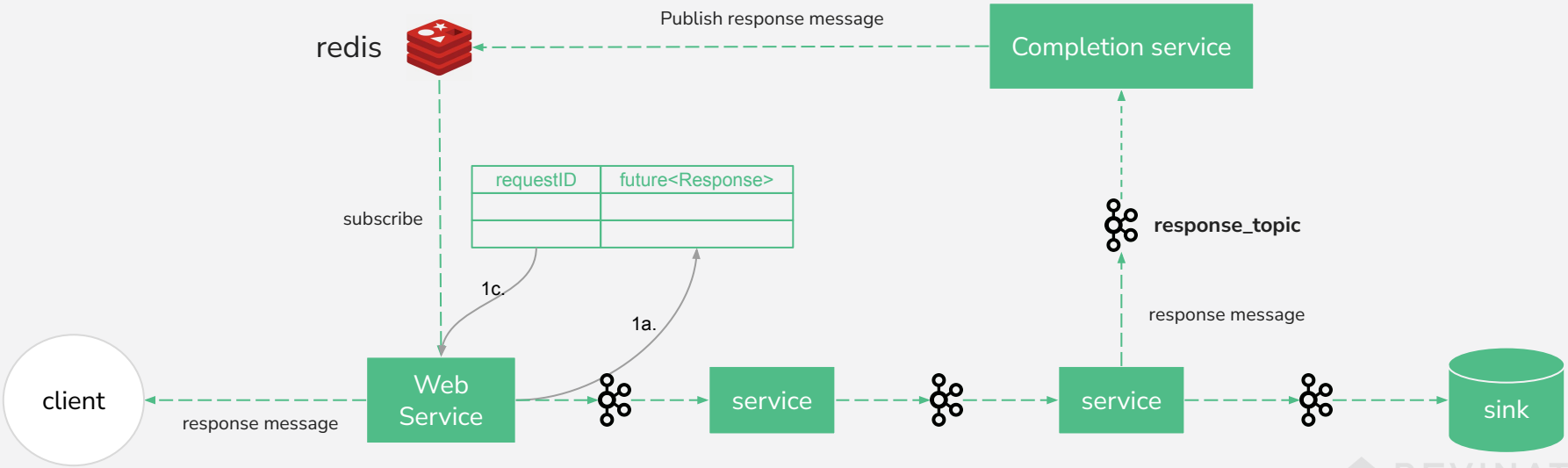
Goal

Ensure that the same (webservice) that received the client HTTP request message will receive the response generated in the asynchronous service pipeline



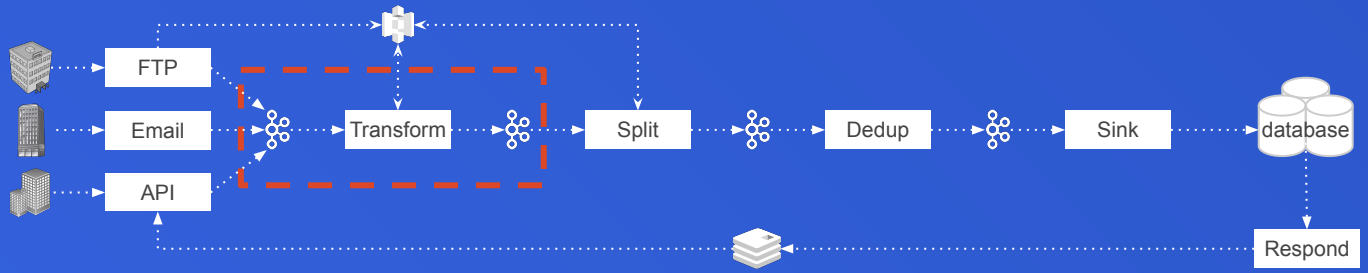
Solution

1. Webservice listener thread
   a. Enter `<requestId, CompeletableFuture<Response>>` in responseTable
   b. Post request to asynchronous pipeline
   c. Block on future
2. Webservice redis channel subscriber thread
   a. Listen for all responses
   b. Look up requestId in responseTable
   c. Complete future if found



REVINATE

6. Processing Messages in Parallel (Beyond Partitions)



"...the maximum parallelism at which your application may run is bounded by the maximum number of stream tasks, which itself is determined by maximum number of partitions of the input topic(s) the application is reading from" [1]

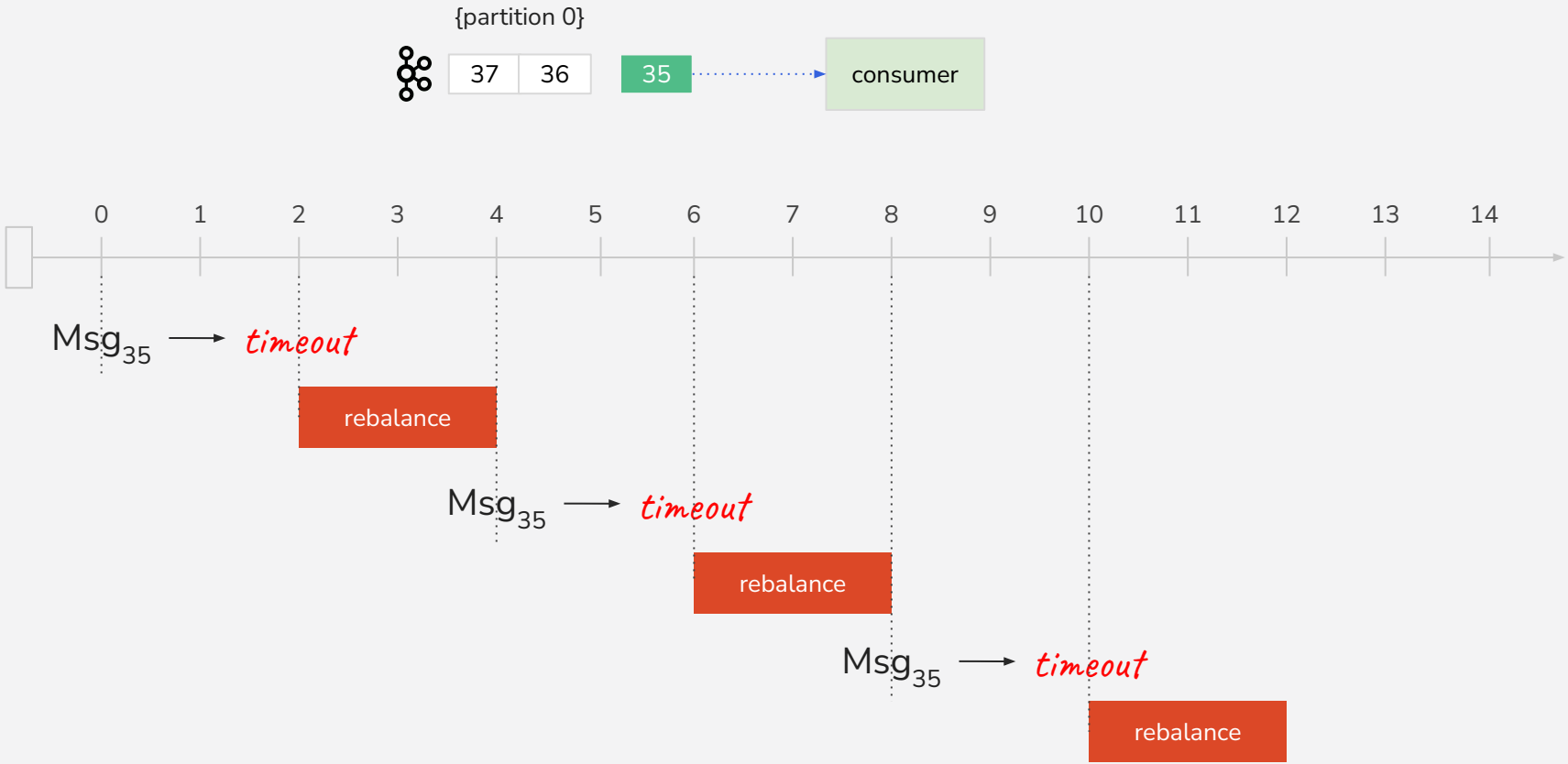
Latency vs. Throughput

- Latency = seconds per message = delay
- Throughput = messages per second (service overall) = volume

Problem

- Fixed number of topic partitions
- Message latency is high
  - Consumer lag
  - Polling timeouts resulting in rebalance events
    - "Stuck partition" (a.k.a. "slow consumer")

Source: [1] https://docs.confluent.io/platform/current/streams/architecture.html


{partition 0}
37
36
35
consumer
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
Msg35
timeout
rebalance
Msg35
timeout
rebalance
Msg35
timeout
rebalance

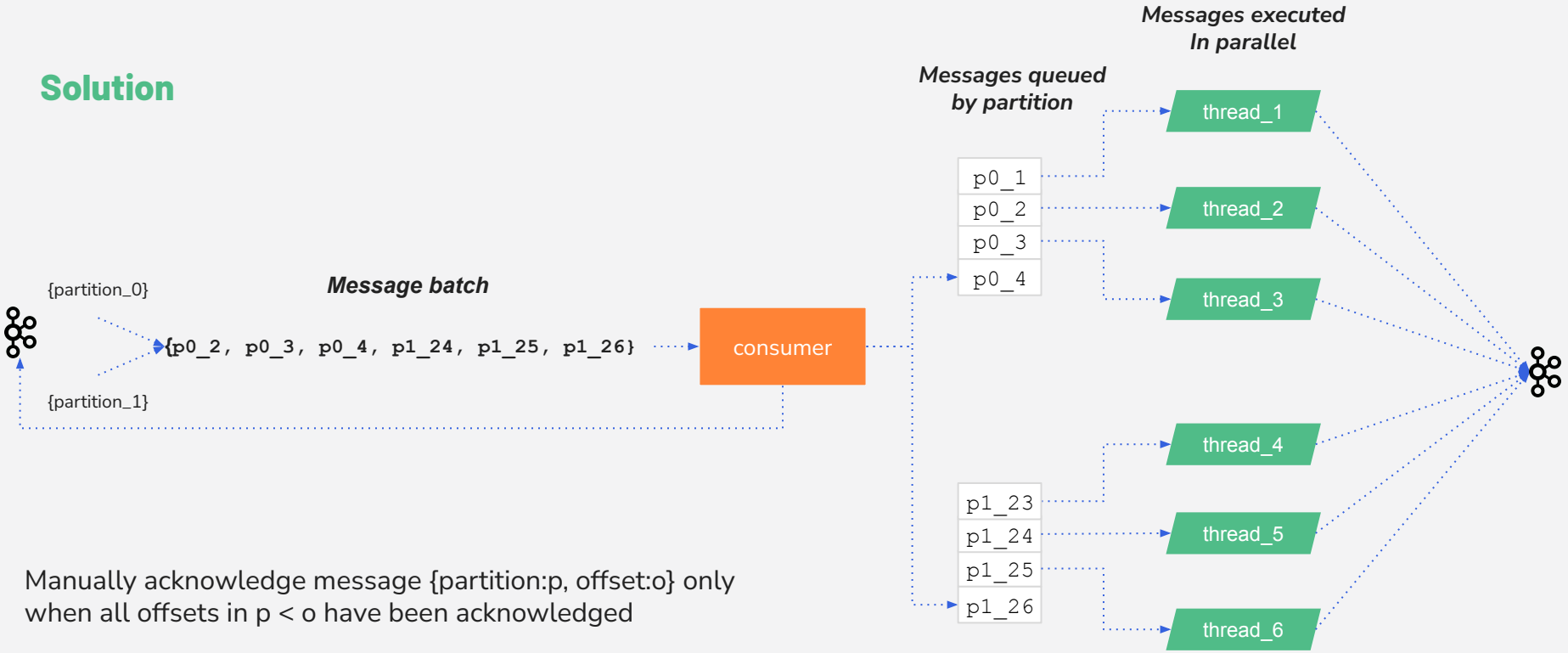
Goal

- Increase message throughput (msgs/sec)
  - Without increasing number of partitions
  - Without improving message latency
- Prevent stuck partitions
- All messages are processed
- Message scheduling order is preserved

Caveat

- Message processing order is not guaranteed

Solution

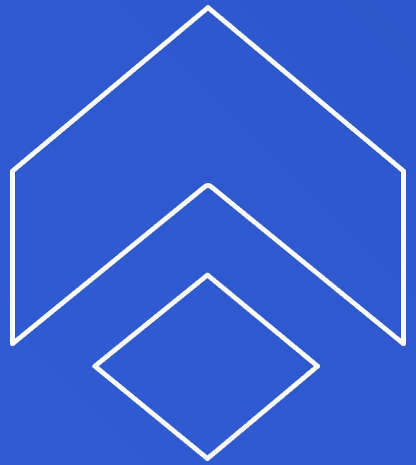


Manually acknowledge message {partition:p, offset:o} only when all offsets in p < o have been acknowledged

Some things to consider

- Kafka vs. Kafka Streams?
- Is process latency consistent for all messages?
- Are their (rare) outlier messages with much higher latency than average?
- What is cause of high message latency?
  - Processing complexity? (CPU bound)
  - Synchronous dependencies on other services? (blocking)
- Use the Confluent Parallel Consumer? [1]
- Wait for KIP-932 (Queues for Kafka)? [2]

Source: [1] https://github.com/confluentinc/parallel-consumer; [2] https://cwiki.apache.org/confluence/display/KAFKA/KIP-932%3A+Queues+for+Kafka



Questions?