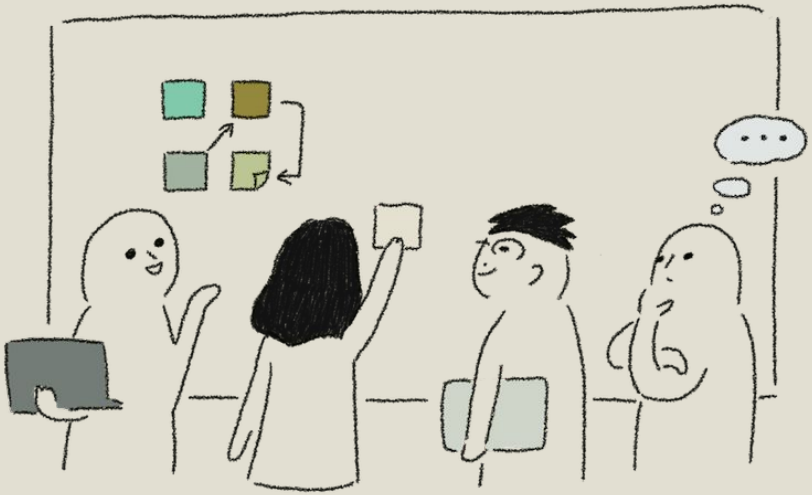
DATE 20250331

Session 5



backend

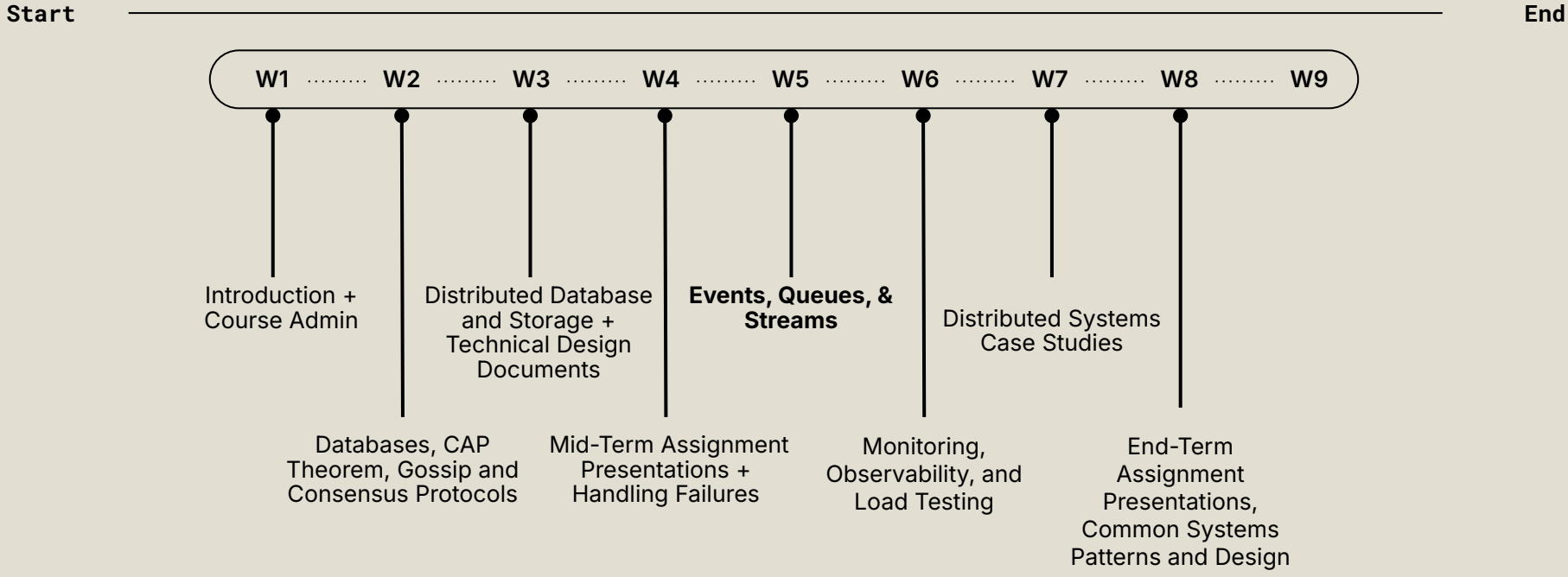
cohort #0 by open camp

#5's agenda

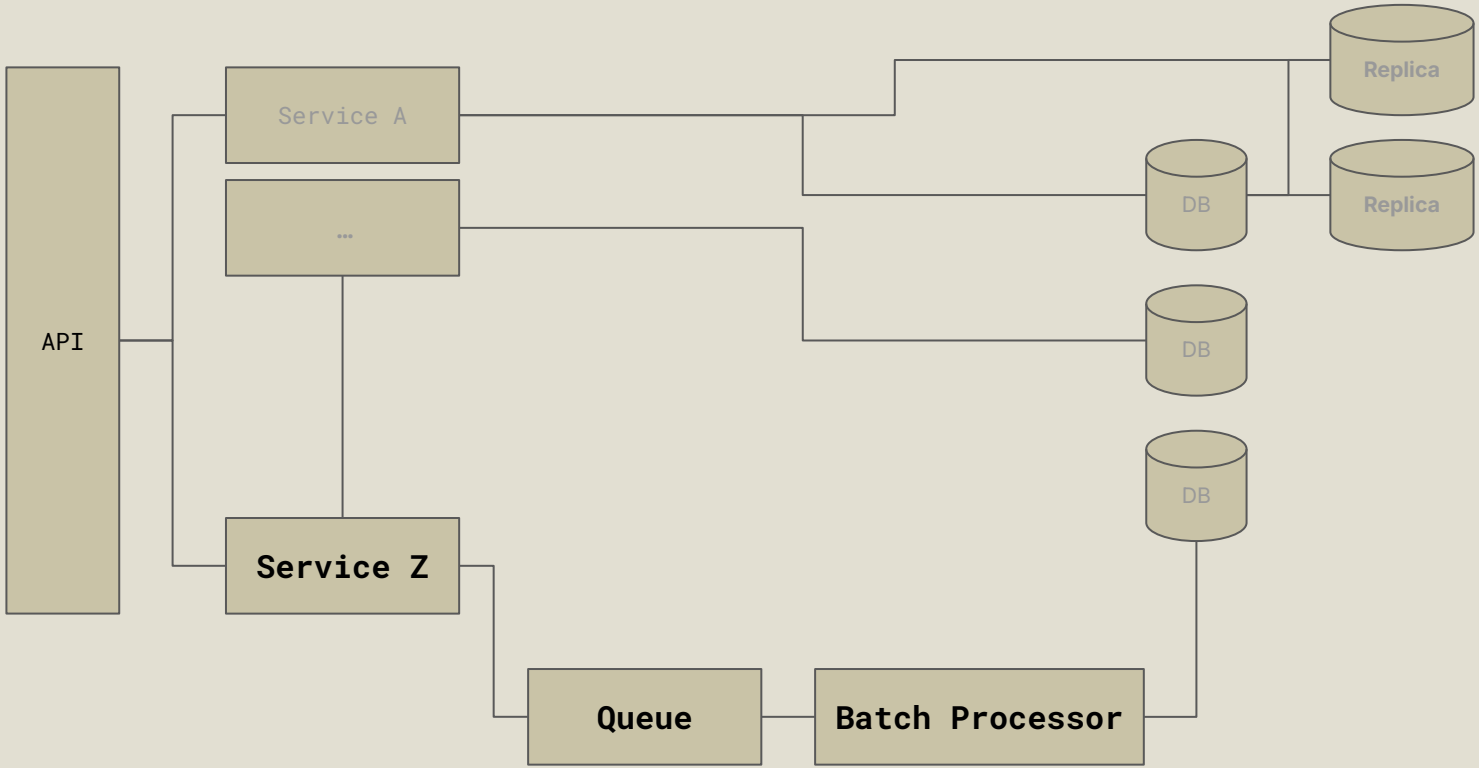
1 ..... admin matters (if any)
2 ..... all about events
3 ..... rabbitmq and kafka
4 ..... w5 assignment

admin matters

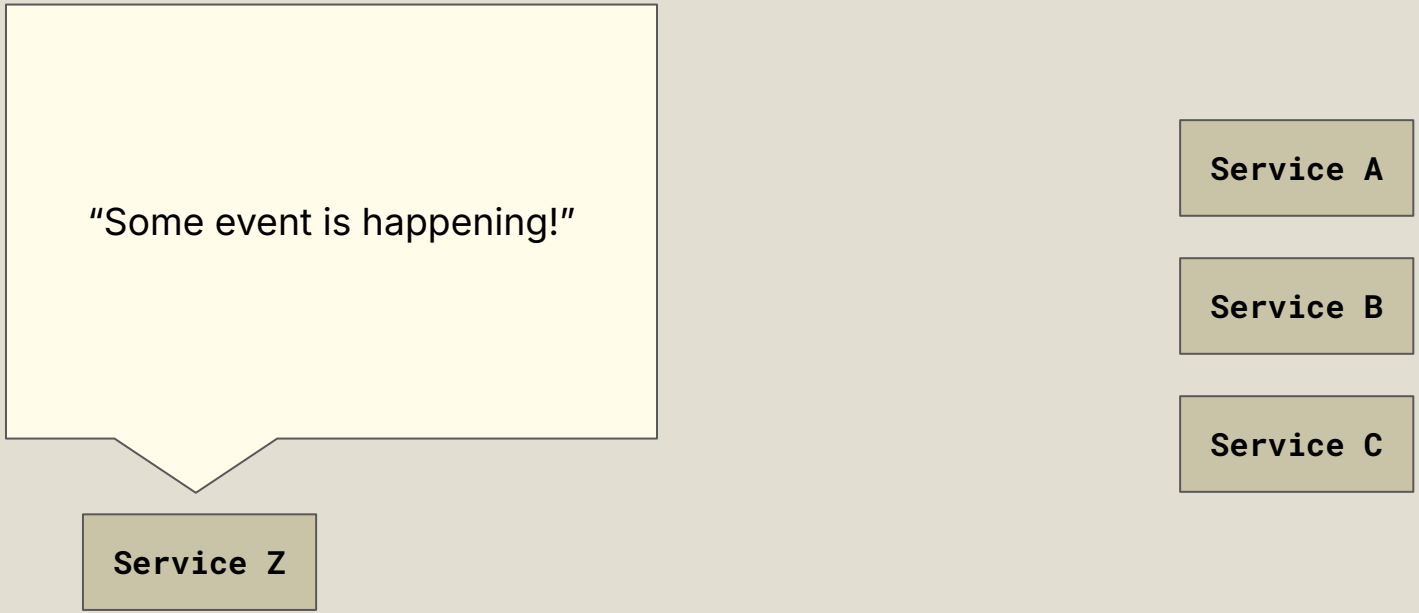
Curriculum: https://opencamp-cc.github.io/backend-curriculum/



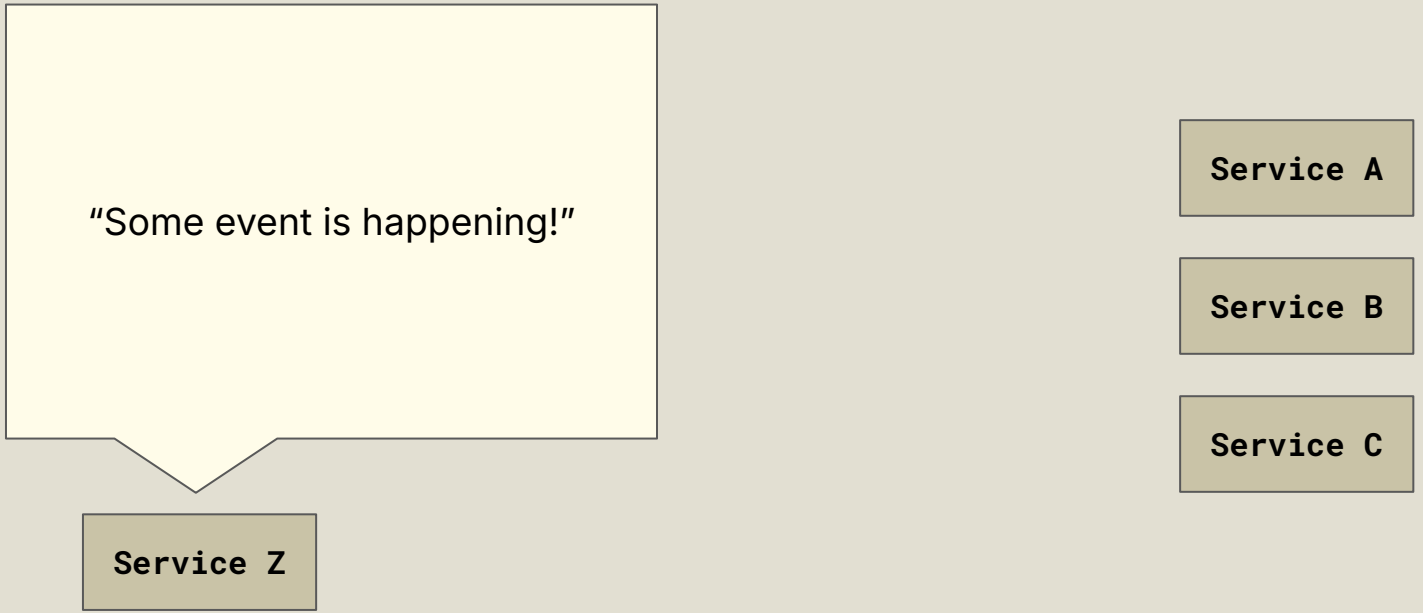
5.1 all about events



Queues: Async Processing and Retrieval

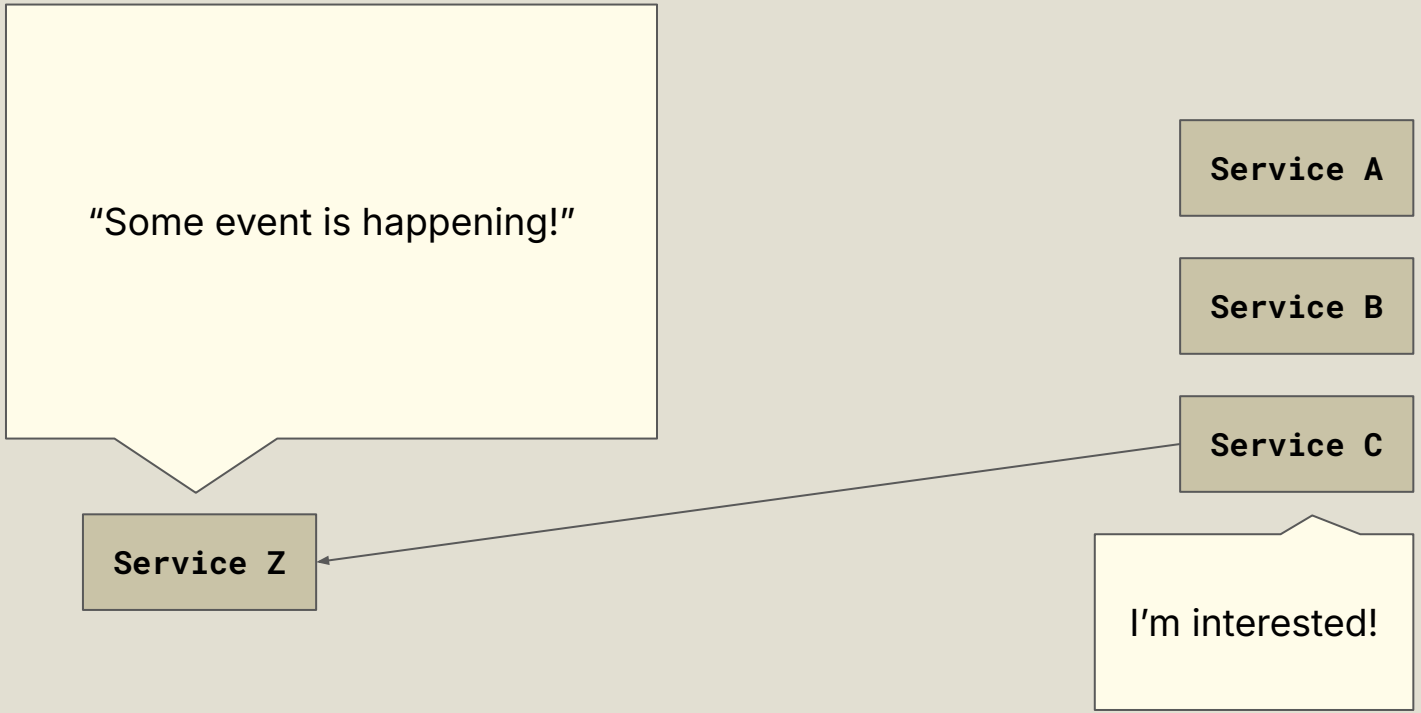


Fires an event…

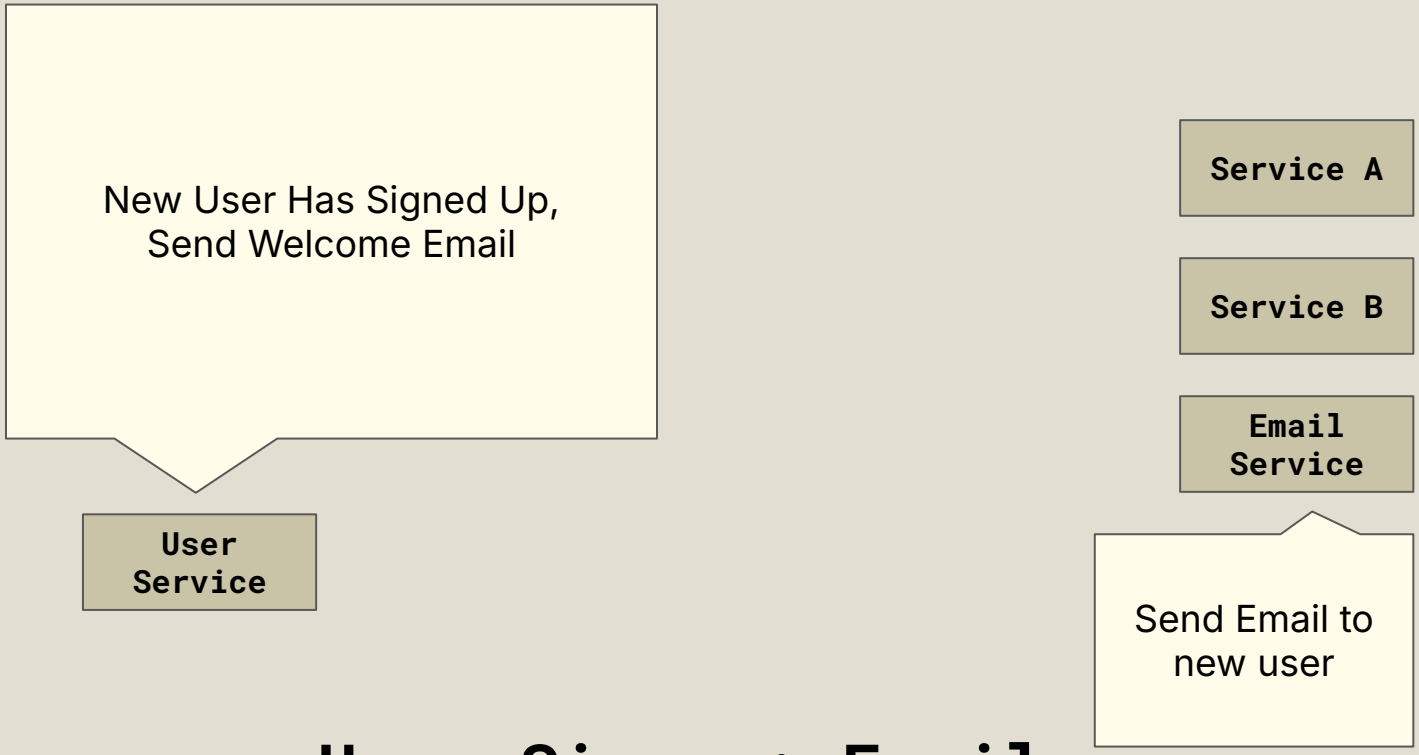


… but doesn’t care if others receive it

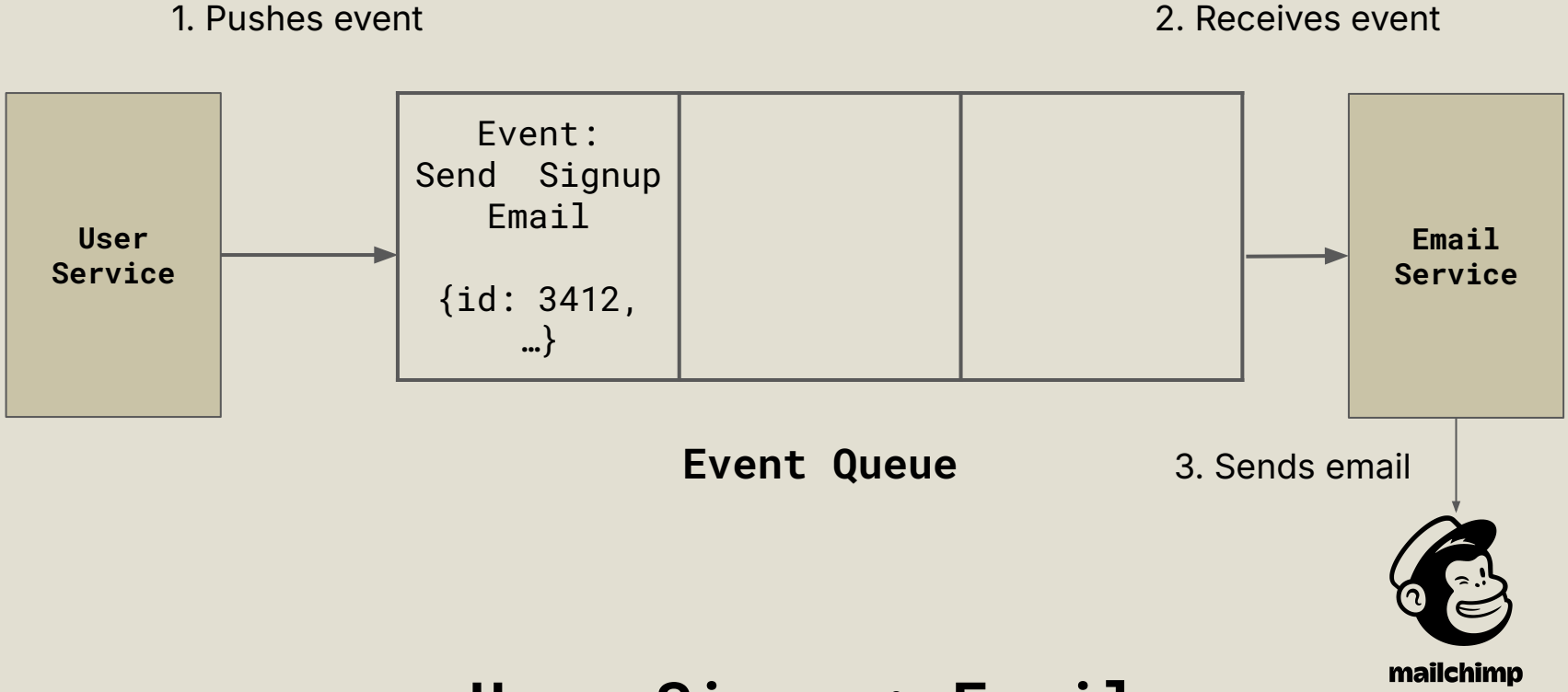
Fires an event…



… and others follow up



User Signup: Email

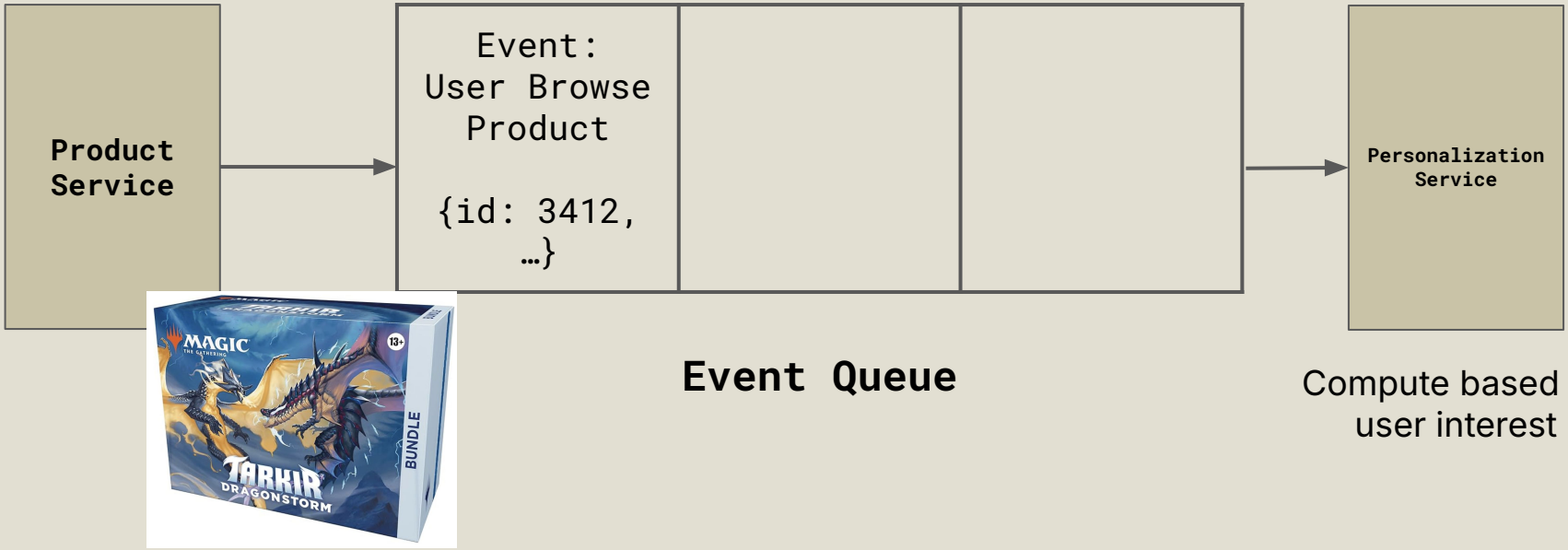


User Signup: Email

Personalization?

1. Pushes event

2. Receives event



Personalization: Background Processing

Events can be:

- **Lossy**: it is possible that they get “lost” or “ignored”
- **One-way**: you are not guaranteed to get a response
- **One-to-Many**: an event can be received by one or multiple listeners (subscribers)
- **Asynchronous**: events may not be processed immediately

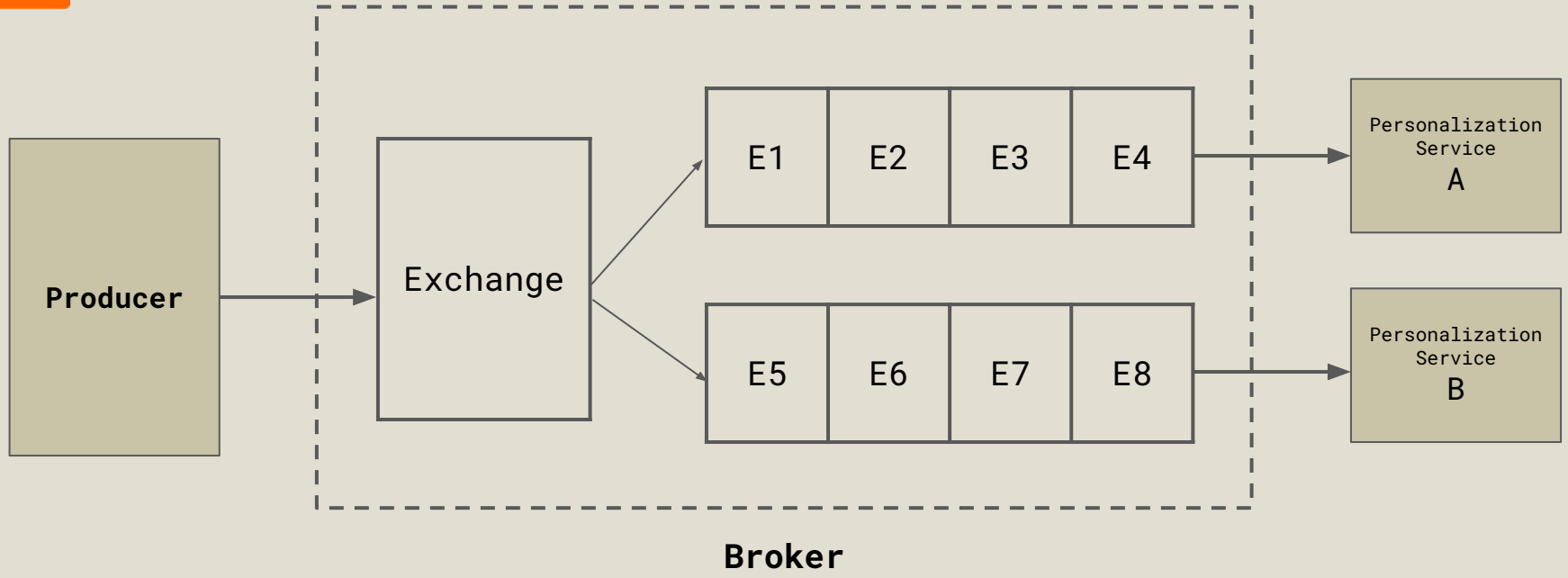
Distributed Message Broker

Event Streaming Platform

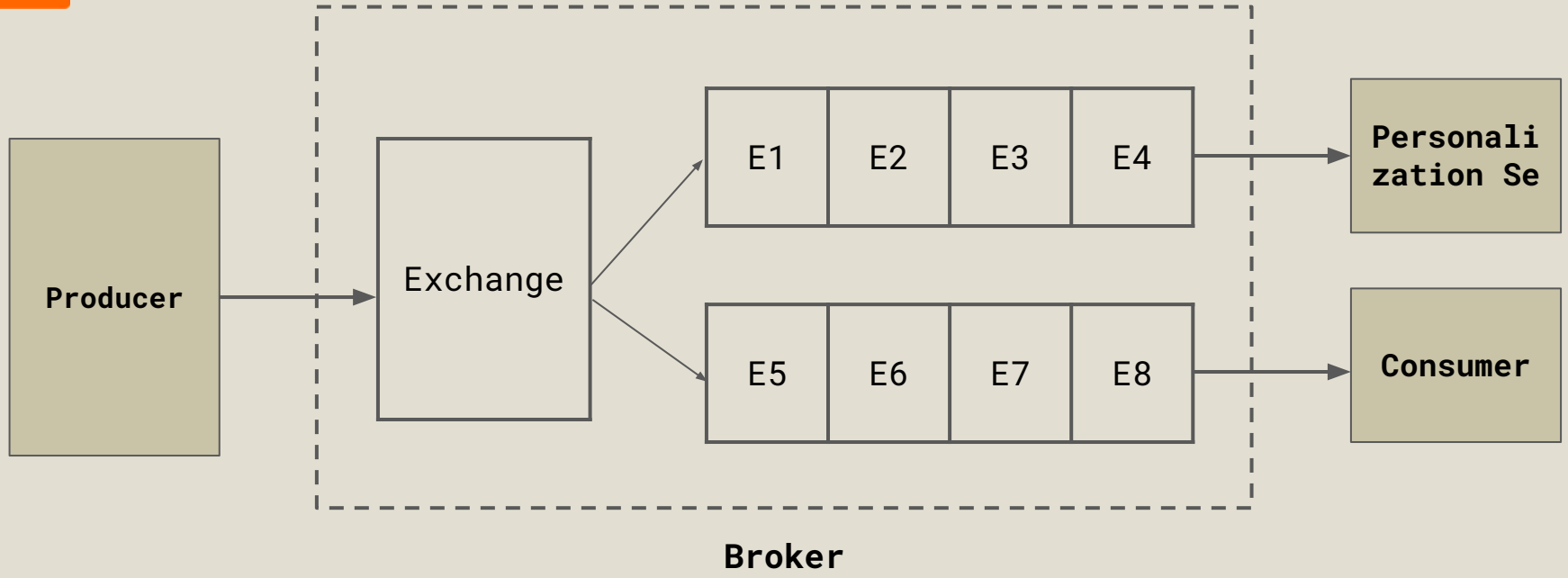
Two Message Queue Systems

5.1.1 rabbitmq

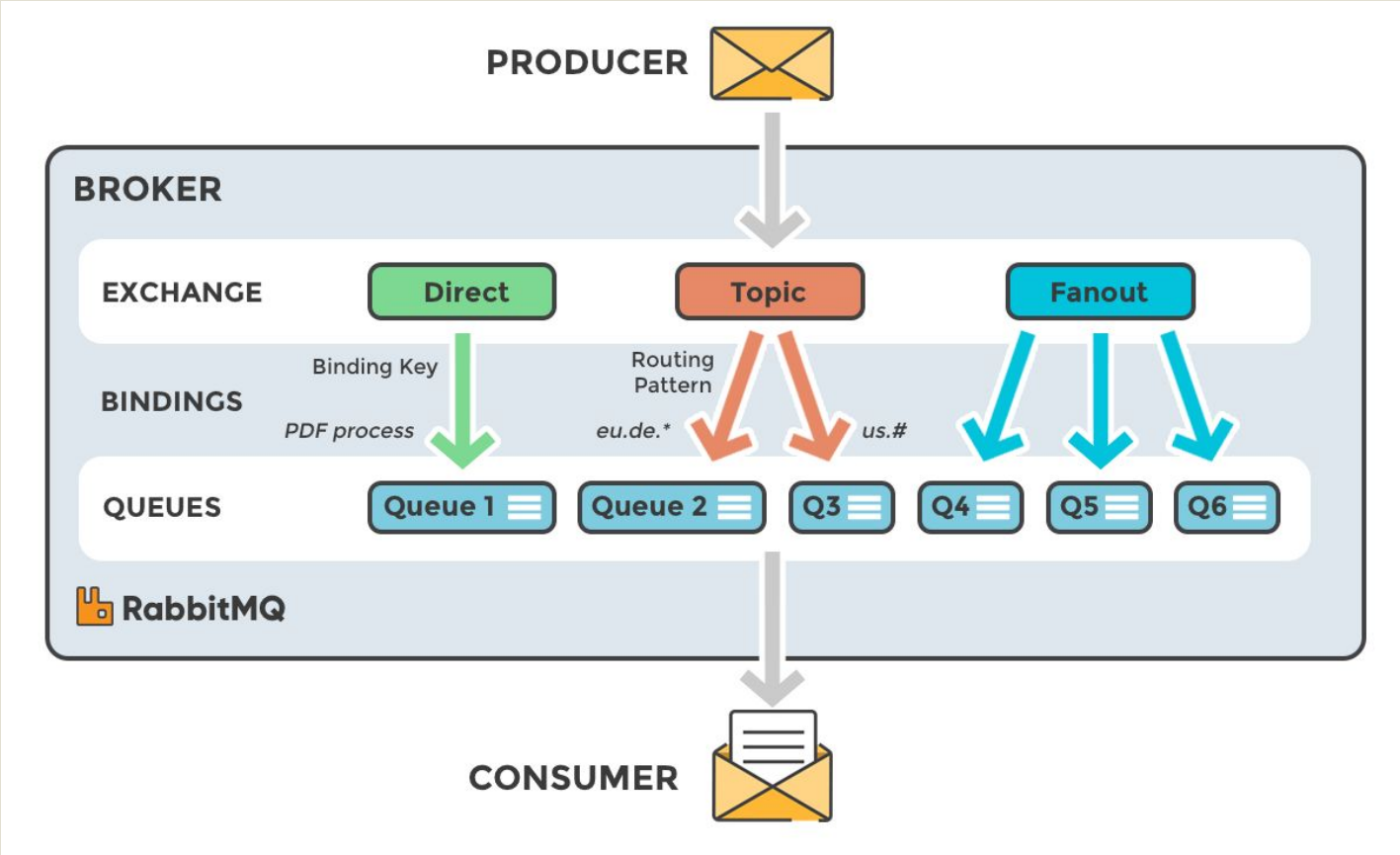
RabbitMQ: Exchange Broker



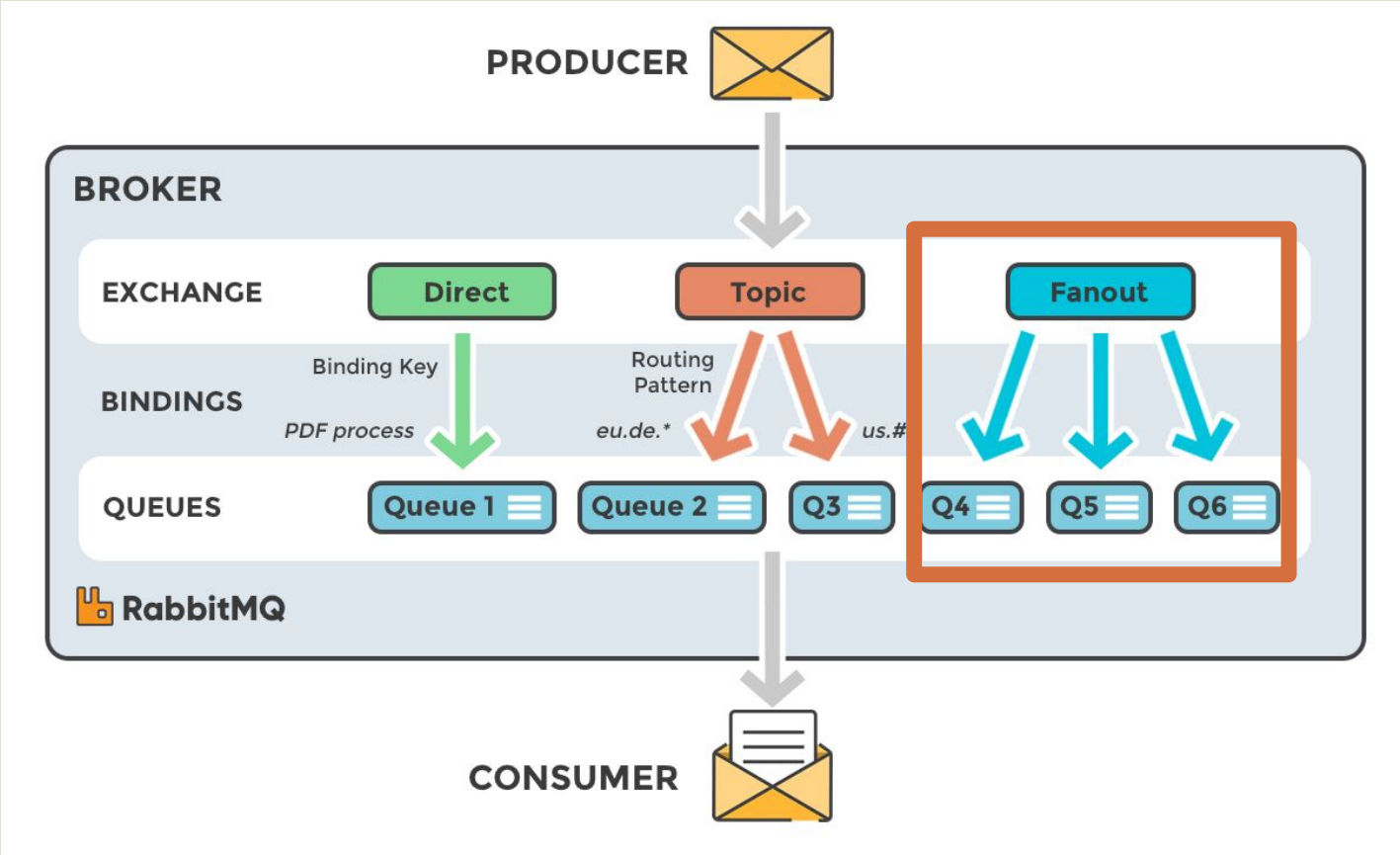
Events are “sorted” into queues at the Exchange



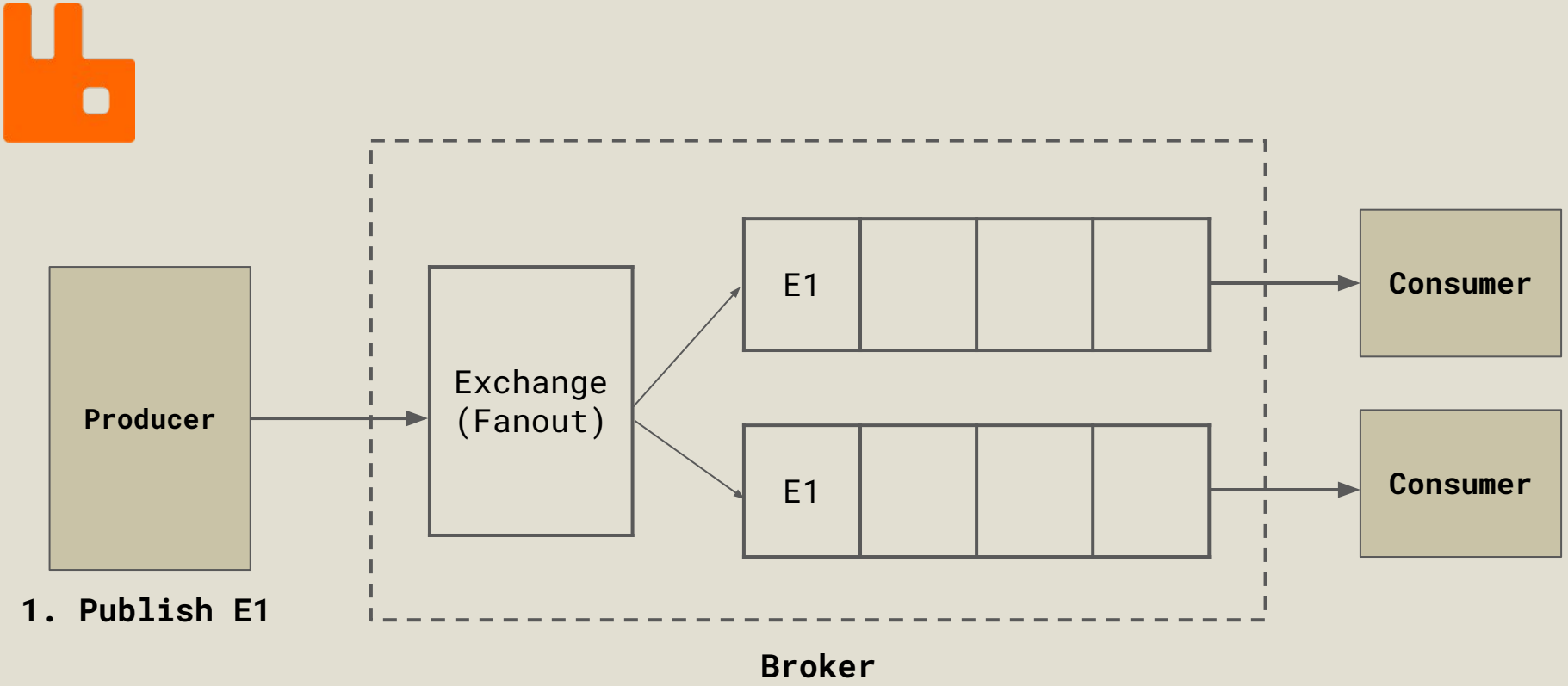
Events are “sorted” into queues at the Exchange



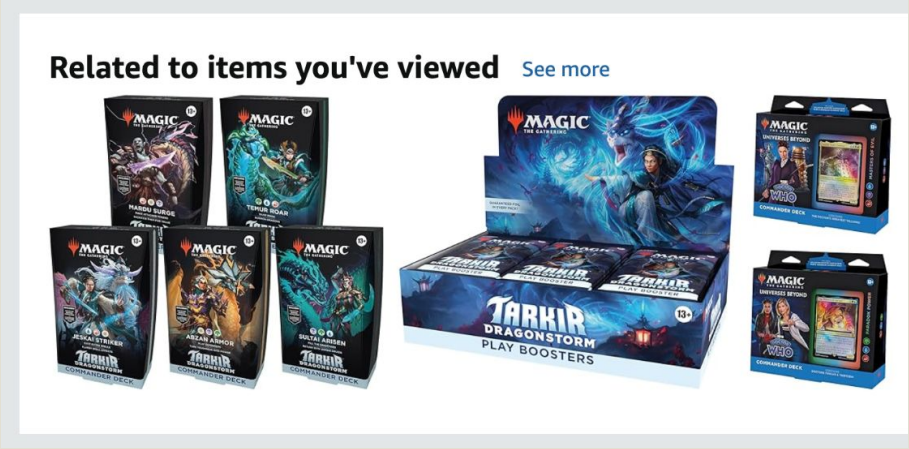
Direct, Topic, and Fanout Exchanges



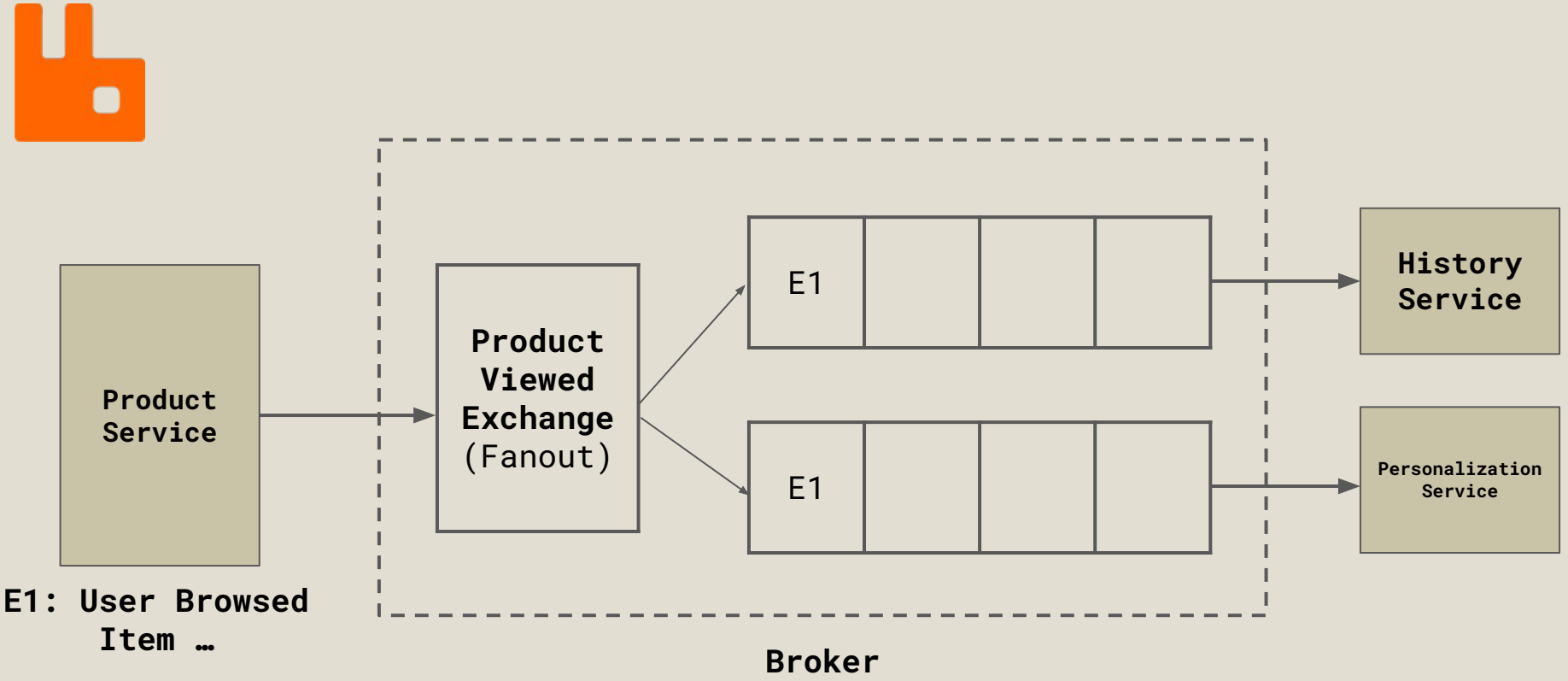
Direct, Topic, and Fanout Exchanges



Fanout: Event is sent to all queues



Personalization + History



Using Fanout for all interested services


```
channel.exchange_declare(exchange='productviews',
                         exchange_type='fanout')
```

Declaring (creating) an Exchange on Producer

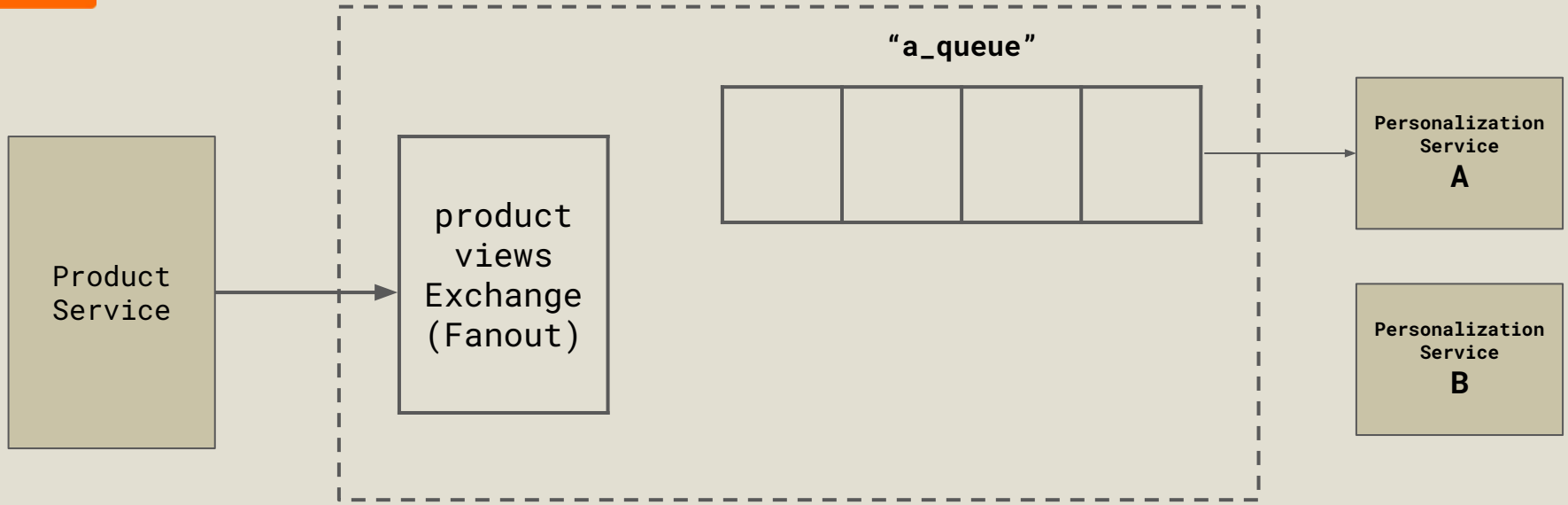
productviews Exchange Created

Personalization
Service
A

```
result = channel.queue_declare(queue='a_queue',
                               exclusive=True)
```

Declaring (creating) an Queue on Consumer

Bind the queues to the Exchange via Routing Key

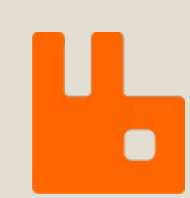


Queue is declared (created)

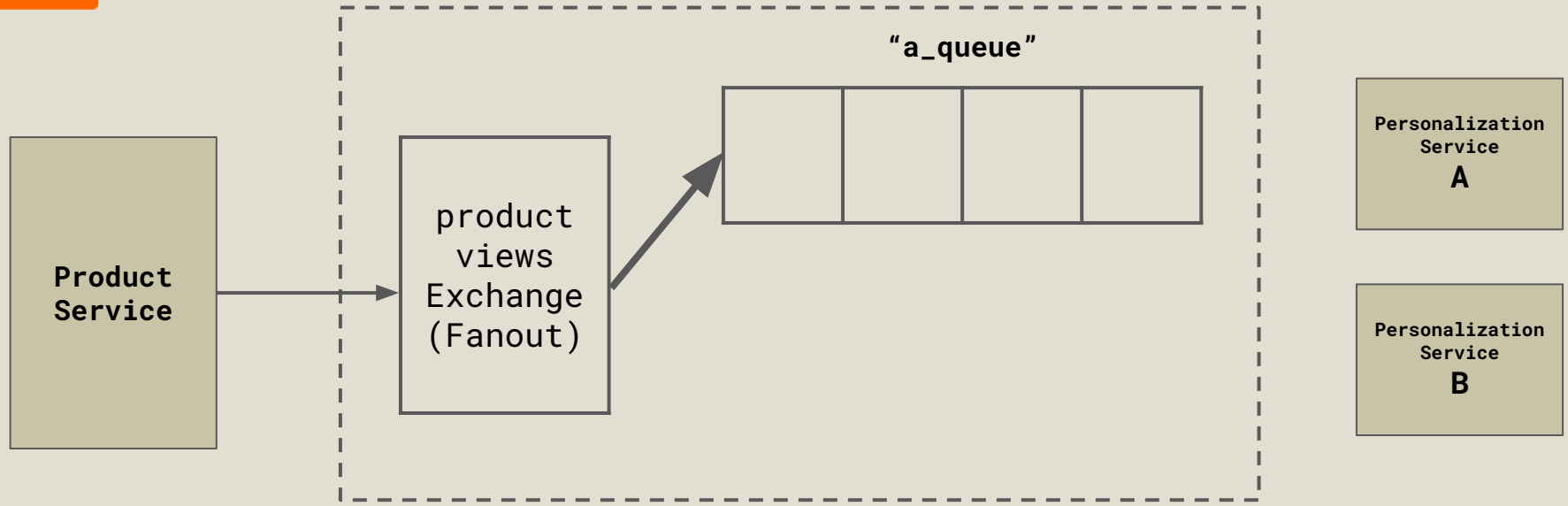
Personalization Service
A

```
channel.queue_bind(exchange='productviews',
                   queue='a_queue')
```

Binding a Queue to an Exchange (Service A)



Bind the queues to the Exchange via Routing Key



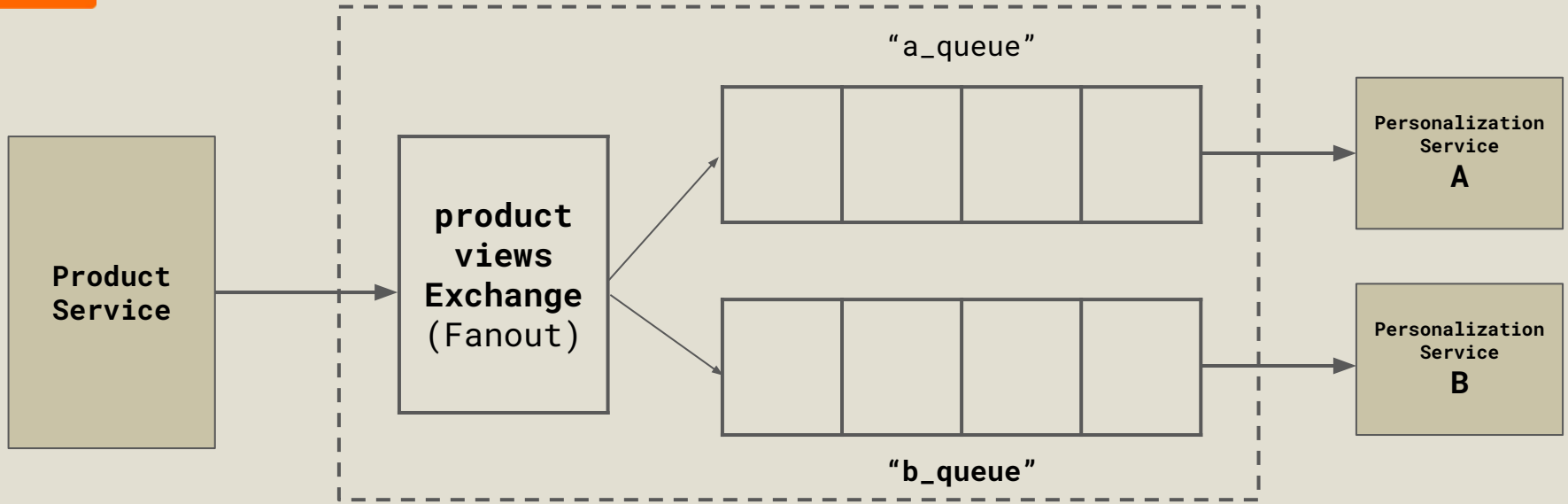
Binding ‘a_queue to ‘productviews’

Personalization Service
B

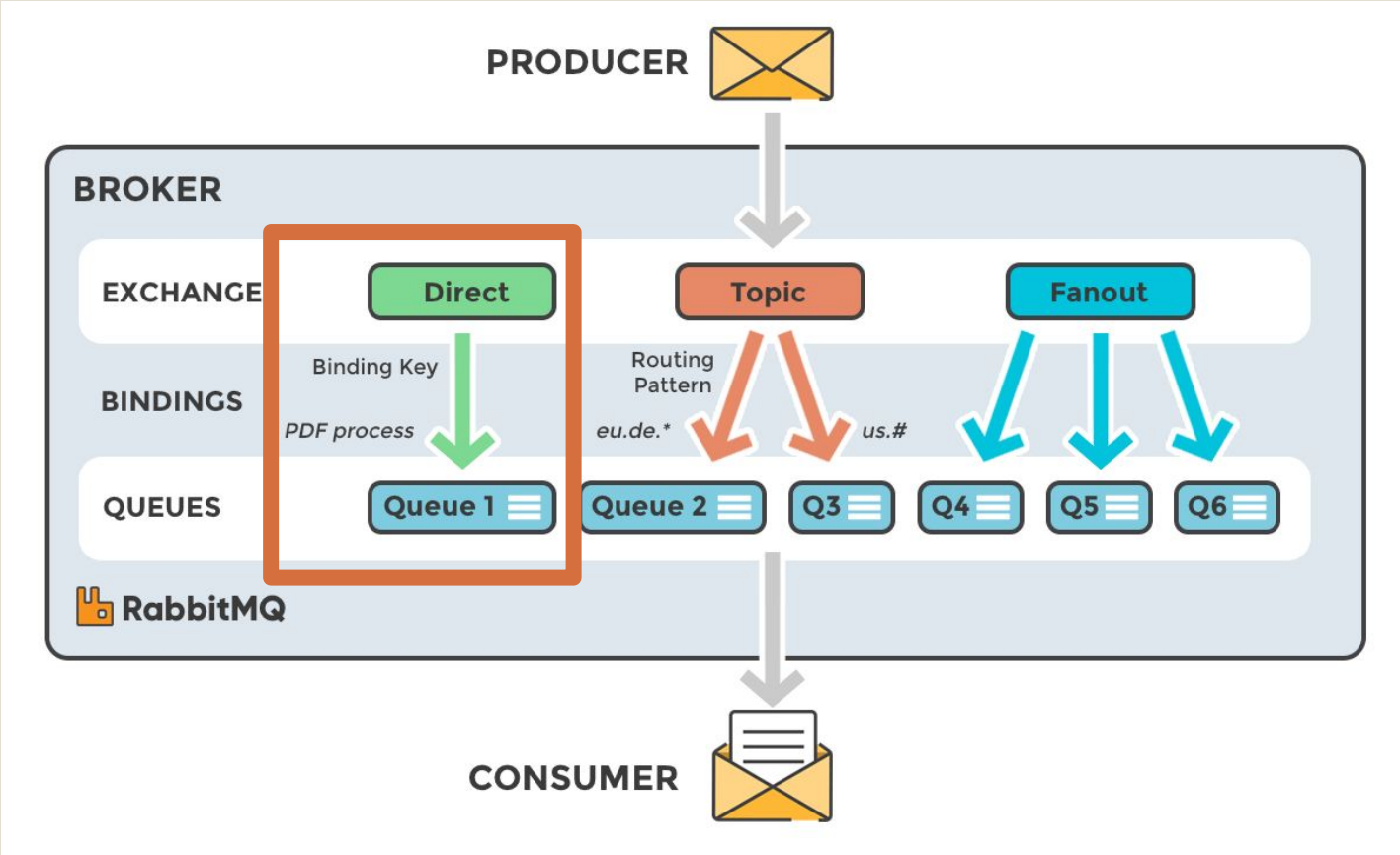
```
result = channel.queue_declare(queue='b_queue',
                               exclusive=True)


channel.queue_bind(exchange='productviews',
                   queue='b_queue')
```

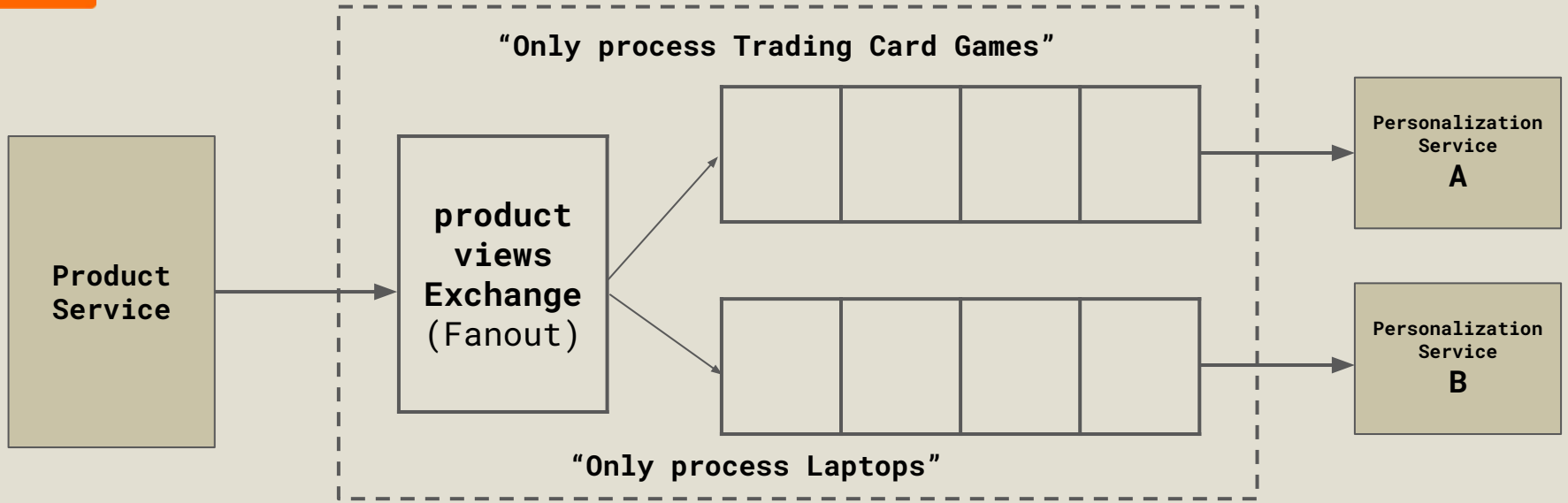
Binding a Queue to an Exchange (Service B)



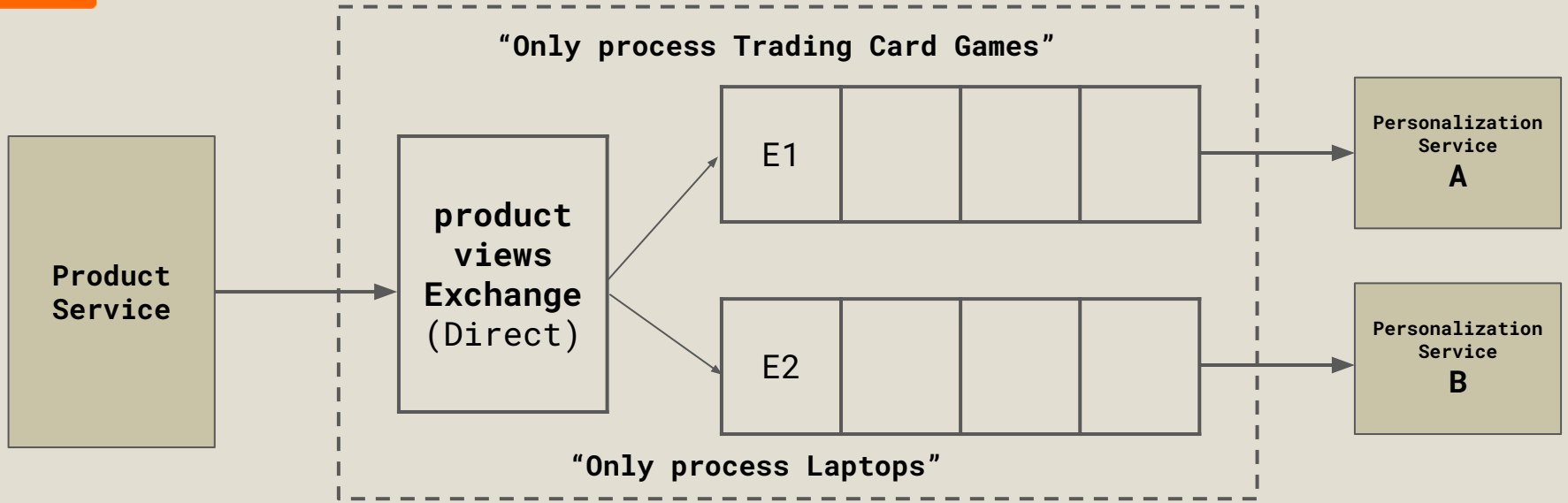
Fanout Exchange: Bound to Two Queues



Direct, Topic, and Fanout Exchanges



Can we filter for only certain events?



E1: Browsed Magic Cards
E2: Browsed Macbook

Use Direct Exchange to Filter by Routing Key


```
channel.exchange_declare(exchange='productviews',
                         exchange_type='direct')
```

Declaring (creating) an Exchange on Producer

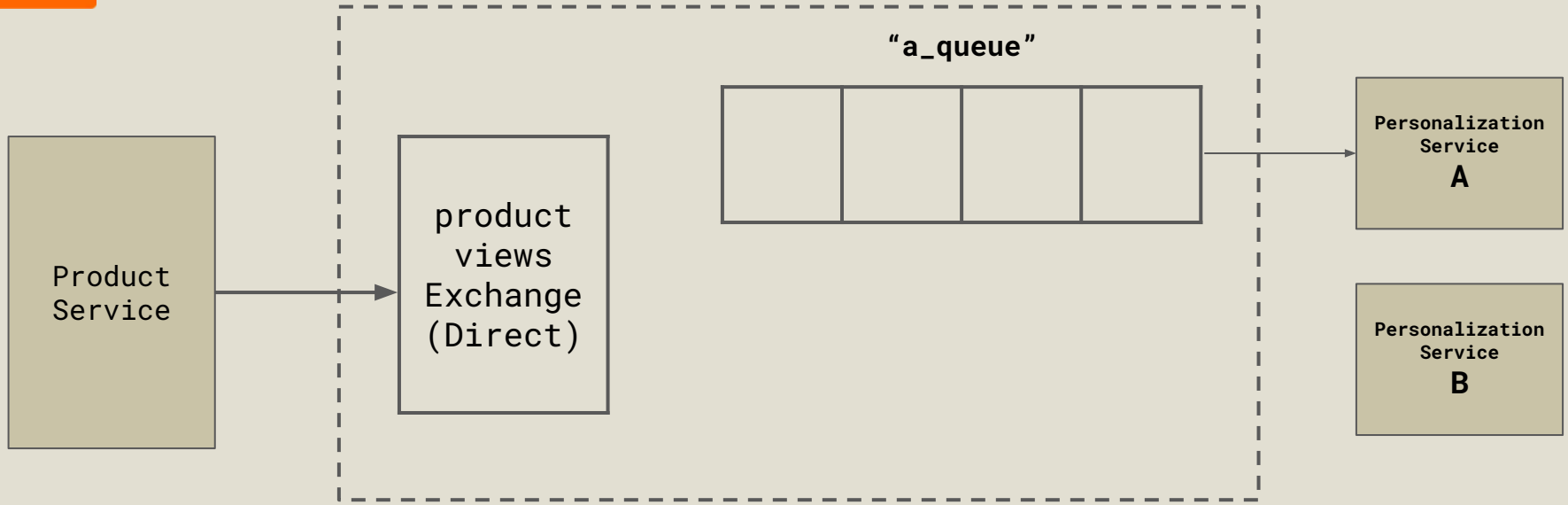
productviews Exchange Created

Personalization
Service
A

```
result = channel.queue_declare(queue='a_queue',
                               exclusive=True)
```

Declaring (creating) an Queue on Consumer

Bind the queues to the Exchange via Routing Key

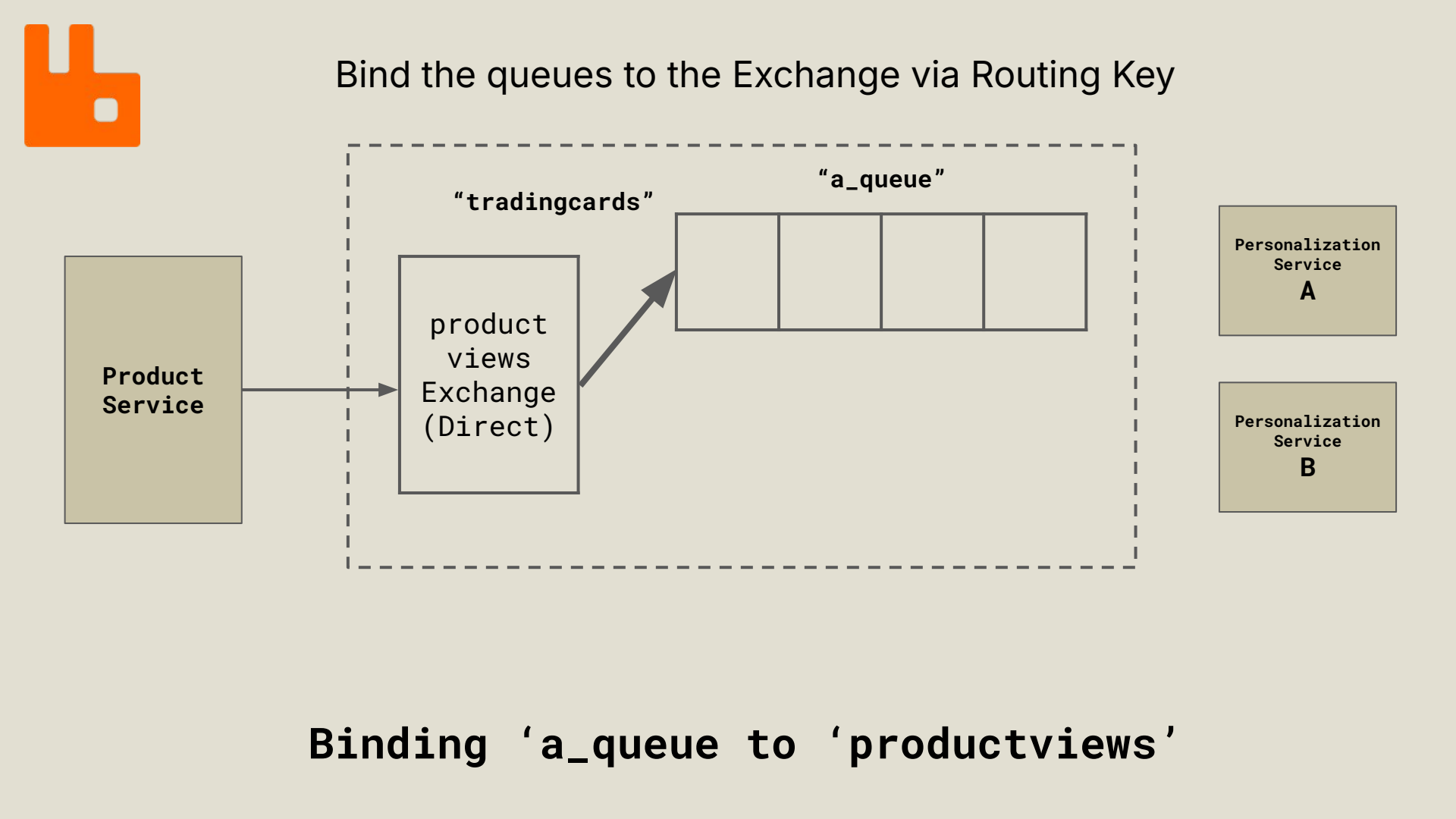


Queue is declared (created)

Personalization Service
A

```
channel.queue_bind(exchange='productviews',
                   queue='a_queue',
                   routing_key='tradingcards')
```

Binding a Queue to an Exchange (Service A)

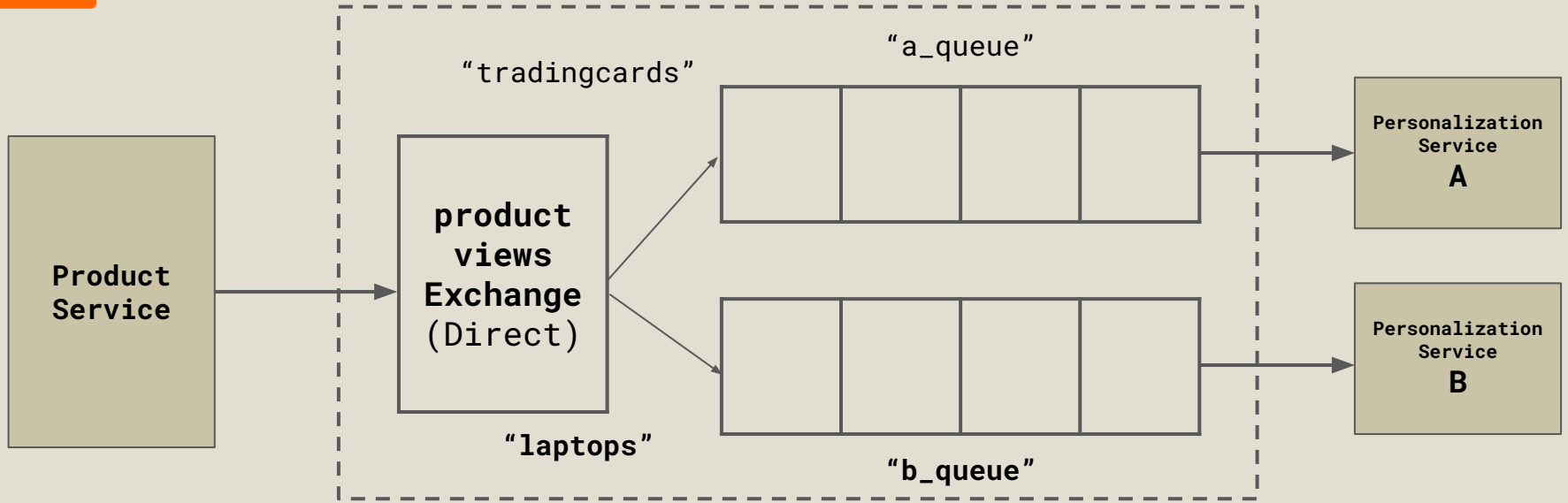

Bind the queues to the Exchange via Routing Key
“tradingcards”
“a_queue”
Product Service
product views Exchange (Direct)
Personalization Service A
Personalization Service B
Binding ‘a_queue to ‘productviews’

Personalization Service
B

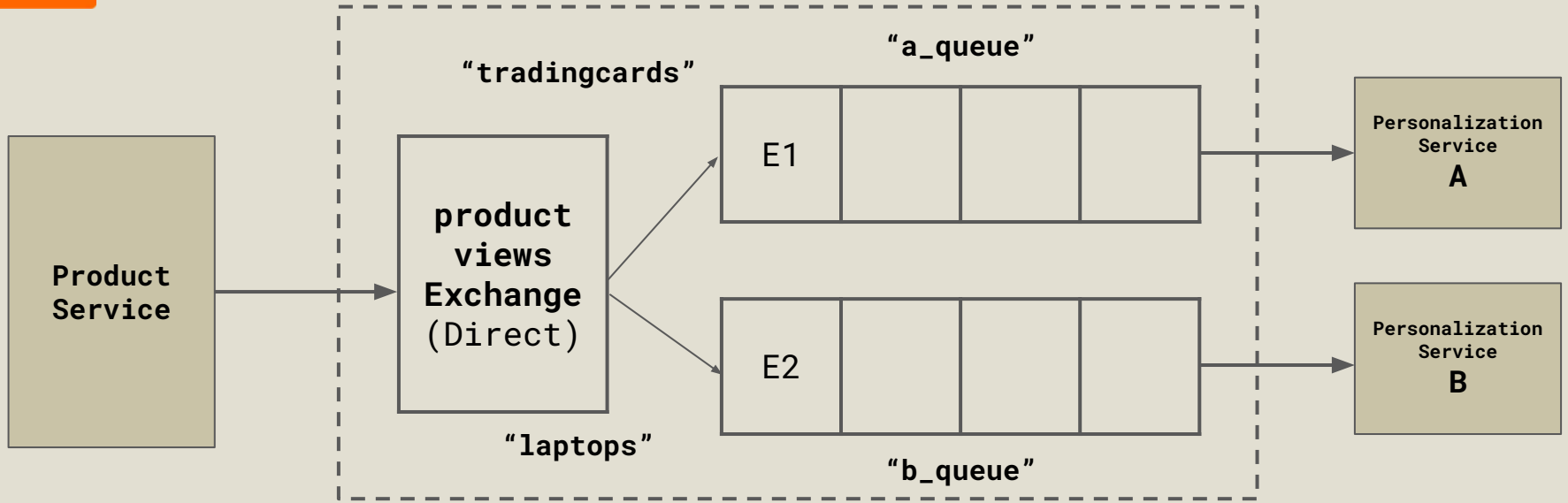
```
result = channel.queue_declare(queue='b_queue',
                               exclusive=True)


channel.queue_bind(exchange='productviews',
                   queue='b_queue',
                   routing_key='laptops')
```

Binding a Queue to an Exchange (Service B)

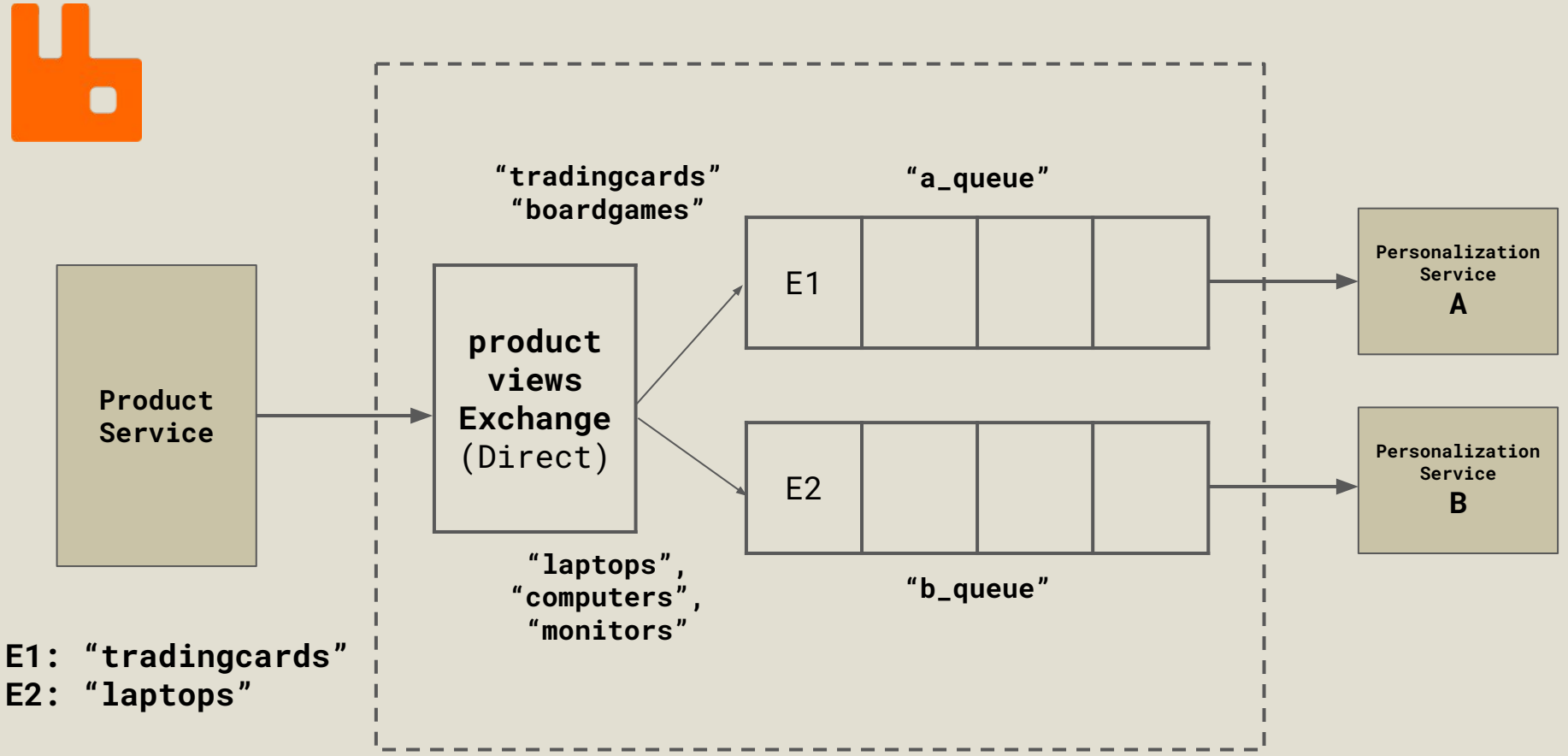


Direct Exchange: Bind Each Category to a Queue



E1: “tradingcards”
E2: “laptops”

Direct Exchange: Bind Each Category to a Queue

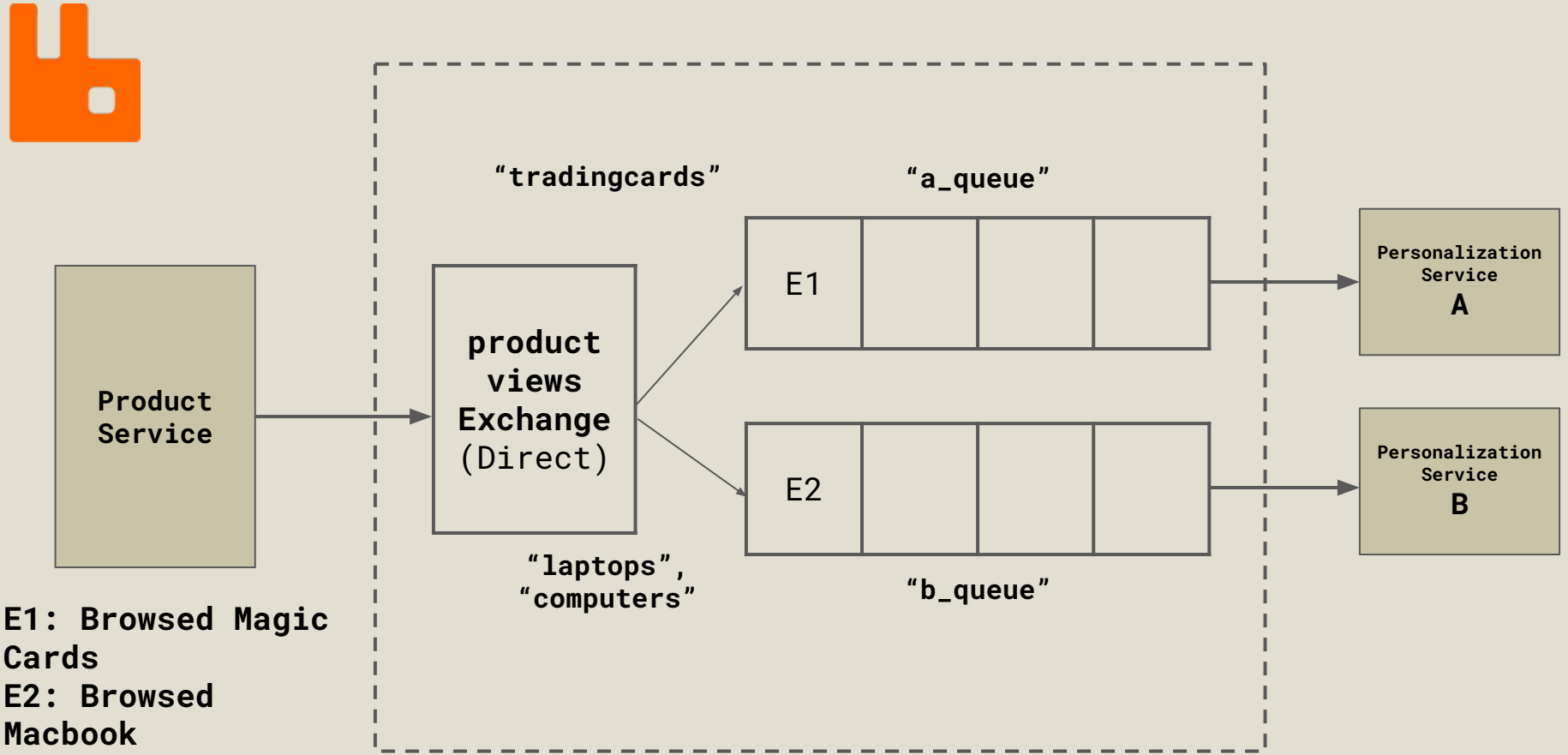


What if we want to handle more categories?

Personalization Service
B

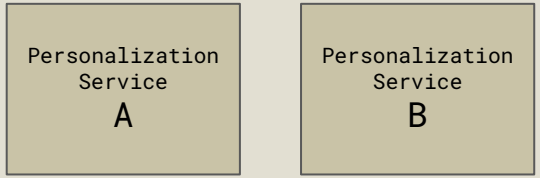
```
channel.queue_bind(exchange='productviews',
                   queue='b_queue',
                   routing_key='computers')
```

Binding the same Queue to an Exchange



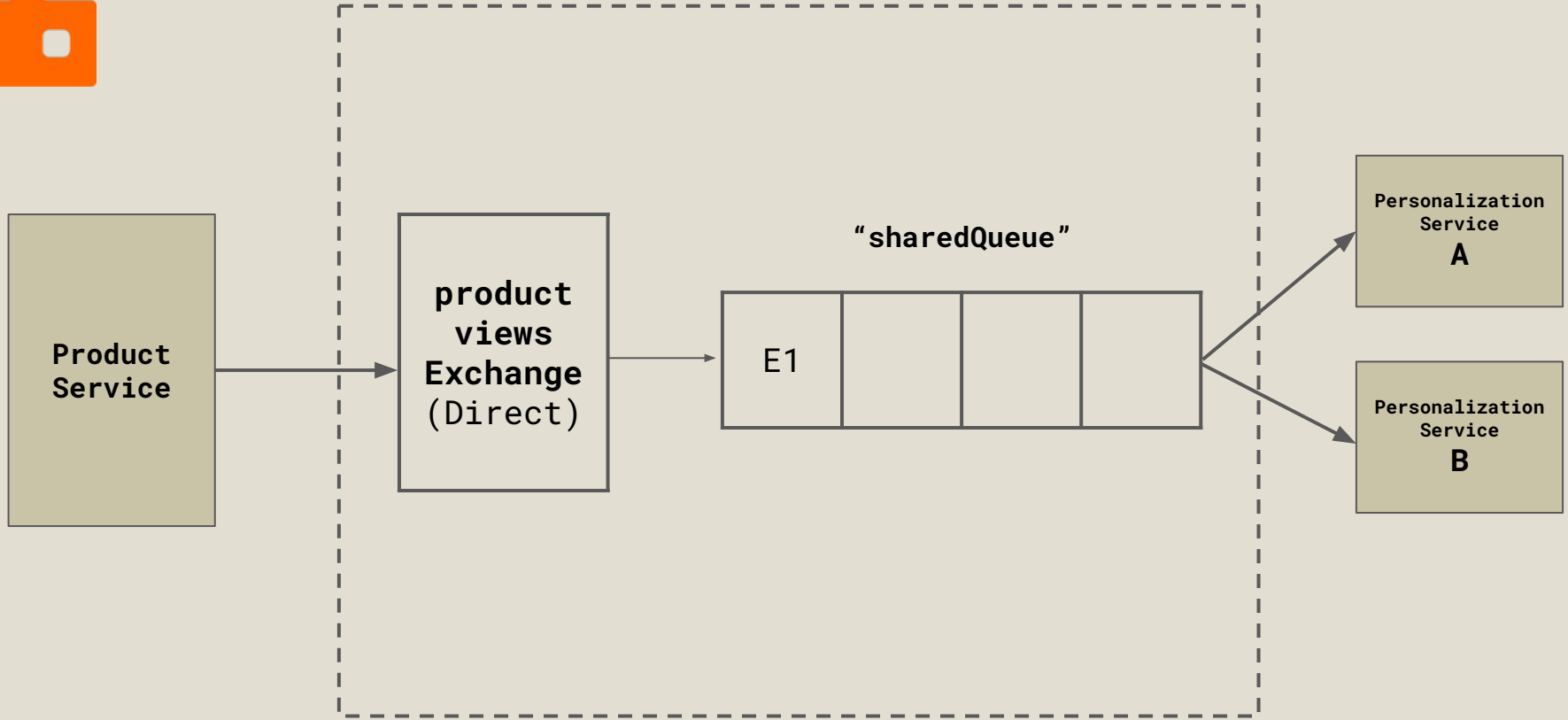
E1: Browsed Magic Cards
E2: Browsed Macbook

"b_queue" will now receive "computers" as well

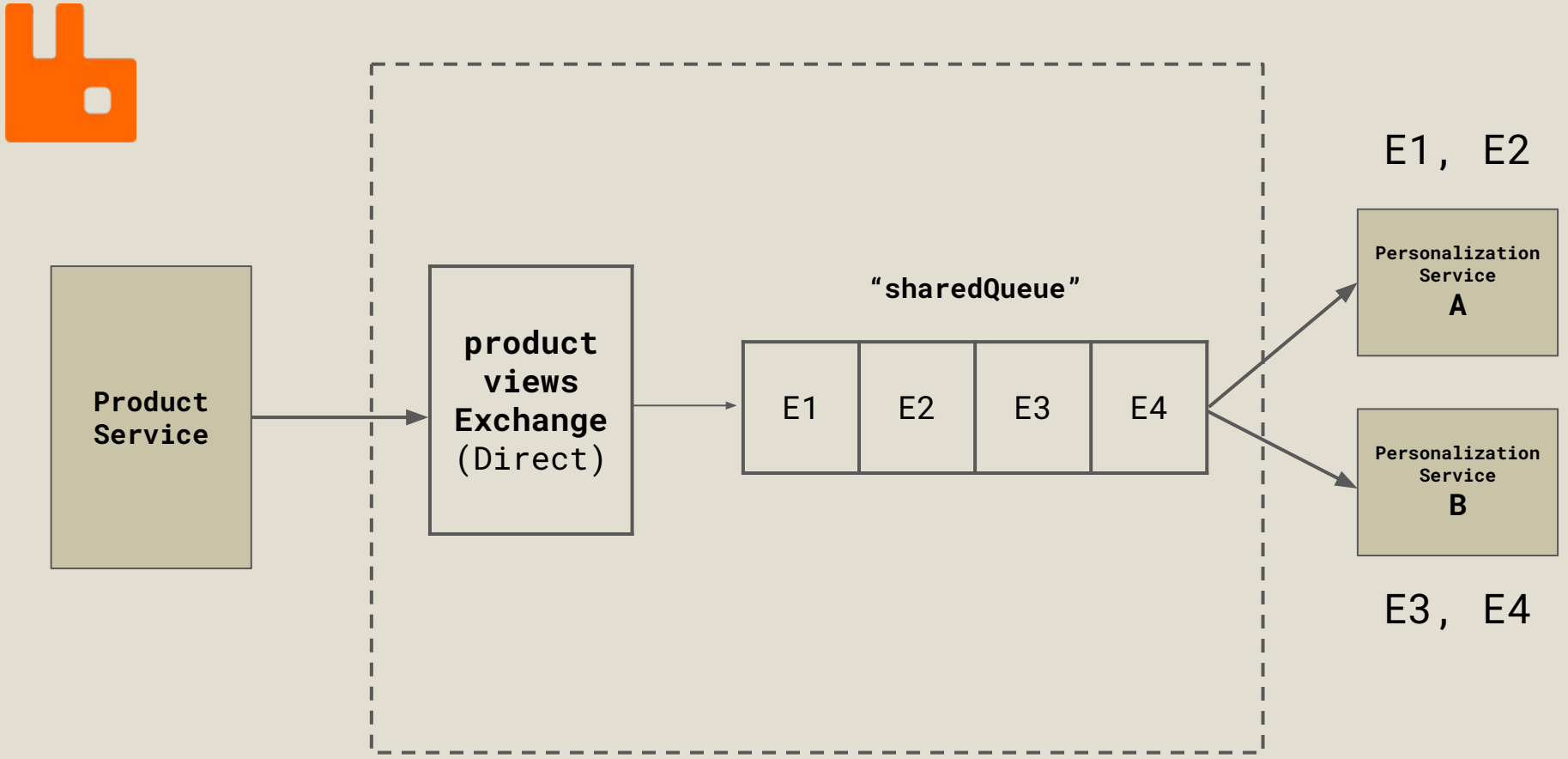



```
result = channel.queue_declare(queue='sharedQueue',
                               exclusive=True)
```

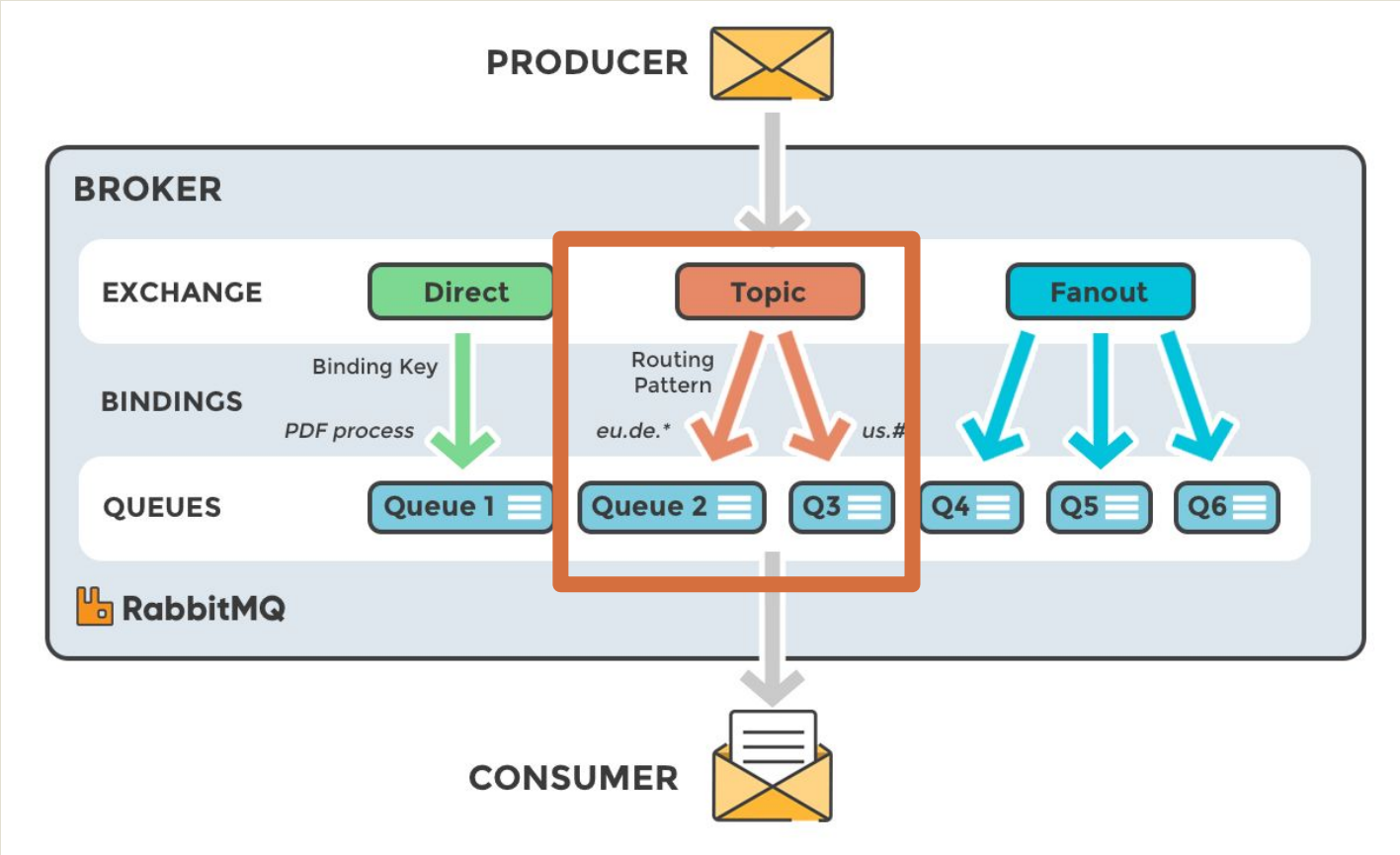
What if Services A and B bind the same queue name?



Services can share a queue, but…



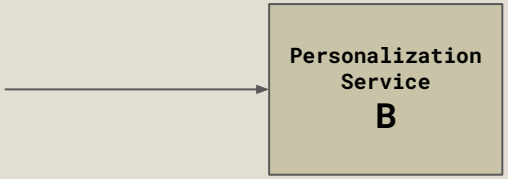
Events are load balanced (round-robin)



Direct, Topic, and Fanout Exchanges

E1: Any type of Magic Cards

E2: Apple Laptops
E3: Any types of PC

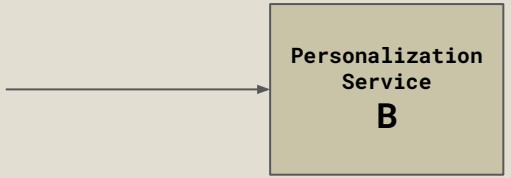


Can we support more without many routing keys?

E1: category.cards.magic.tarkir
category.cards.magic.*

E2: category.computers.laptops.apple
E3: category.computers.pc.dell

category.computers.#



Topic Exchange: Use Wildcards

“The binding key must also be in the same form. The logic behind the topic exchange is similar to a direct one - a message sent with a particular routing key will be delivered to all the queues that are bound with a matching binding key.

However there are two important special cases for binding keys:

*** (star) can substitute for exactly one word.**

 # (hash) can substitute for zero or more words.”

RabbitMQ: Topics

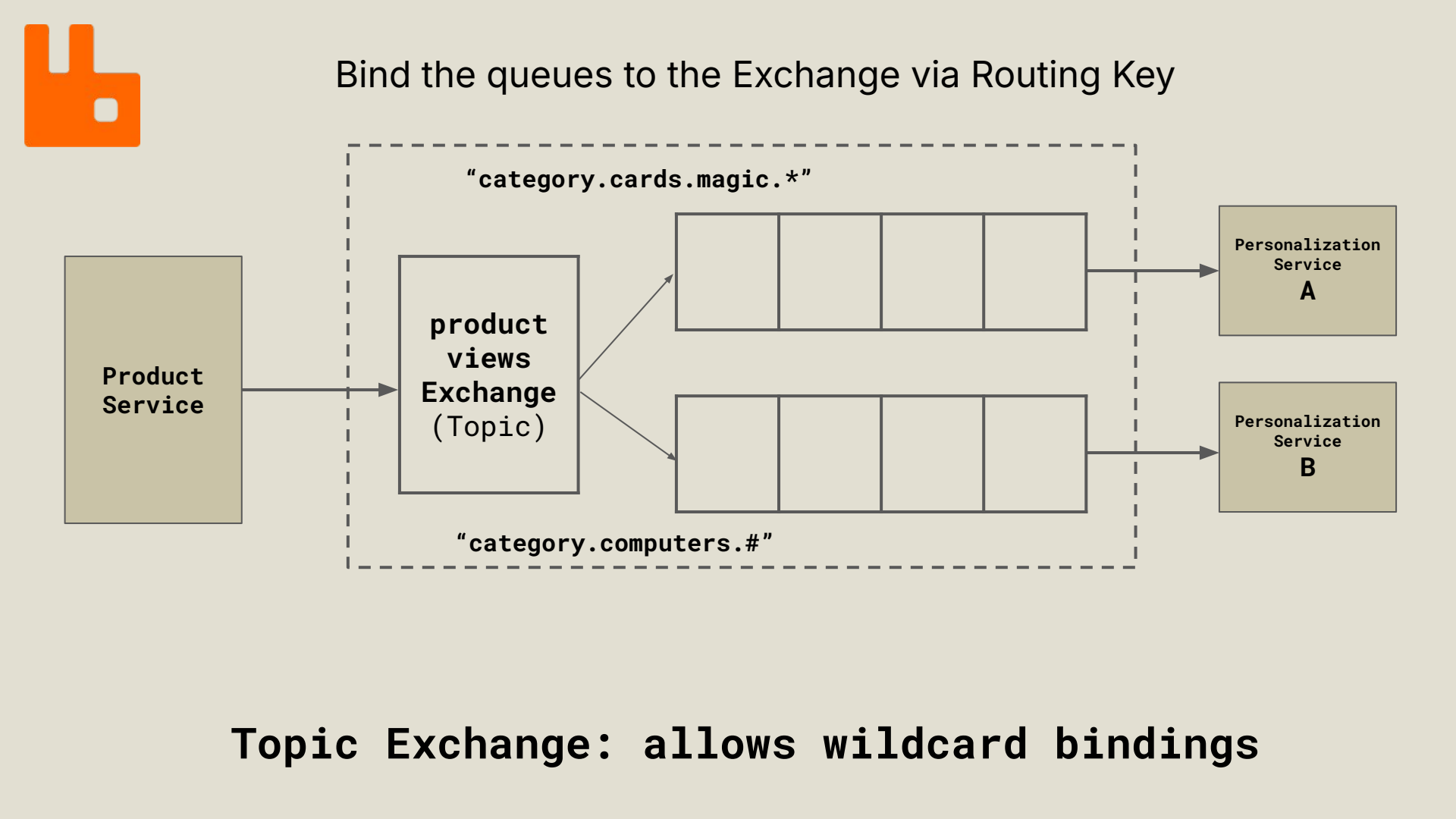
https://www.rabbitmq.com/tutorials/tutorial-five-python

*** (star) can substitute for exactly one word.**

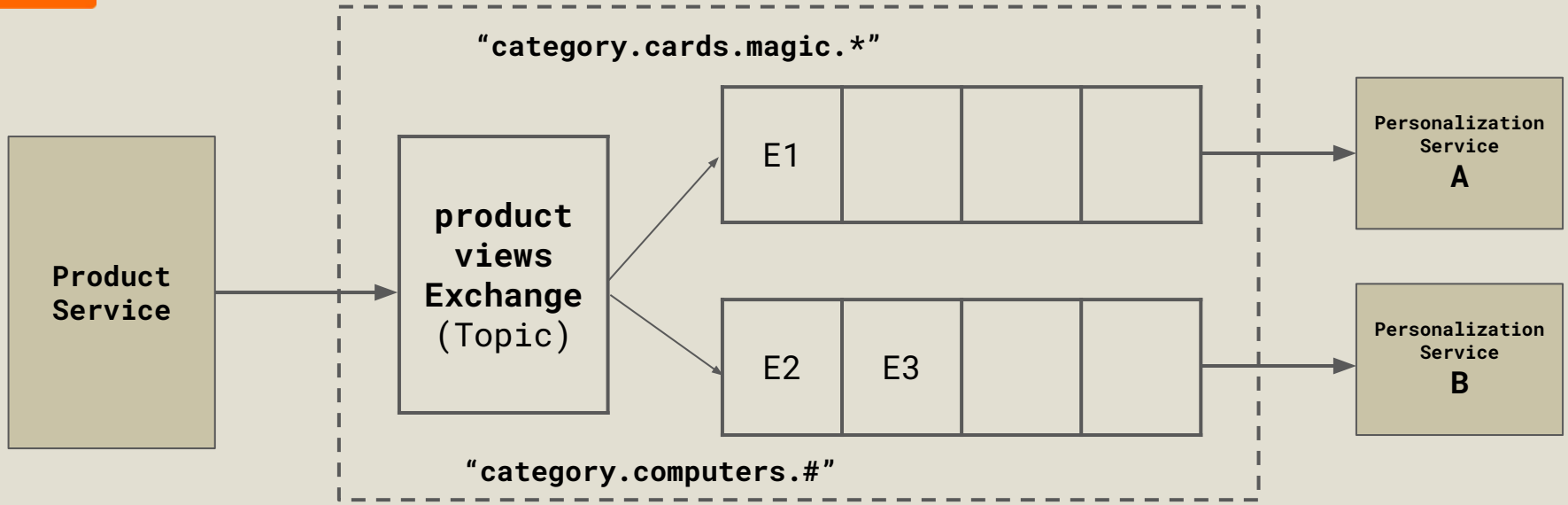
`category.cards.magic.*`
`category.cards.magic.tarkir` ✅
`category.cards.magic.innistrad` ✅
`category.cards.magic.innistrad.booster` ❌

 # (hash) can substitute for zero or more words.

`category.computers.#`
`category.computers` ✅
`category.computers.macbook` ✅
`category.computers.macbook.pro` ✅


Bind the queues to the Exchange via Routing Key
“category.cards.magic.*”
Product Service
product views Exchange (Topic)
Personalization Service A
Personalization Service B
“category.computers.#”
Topic Exchange: allows wildcard bindings

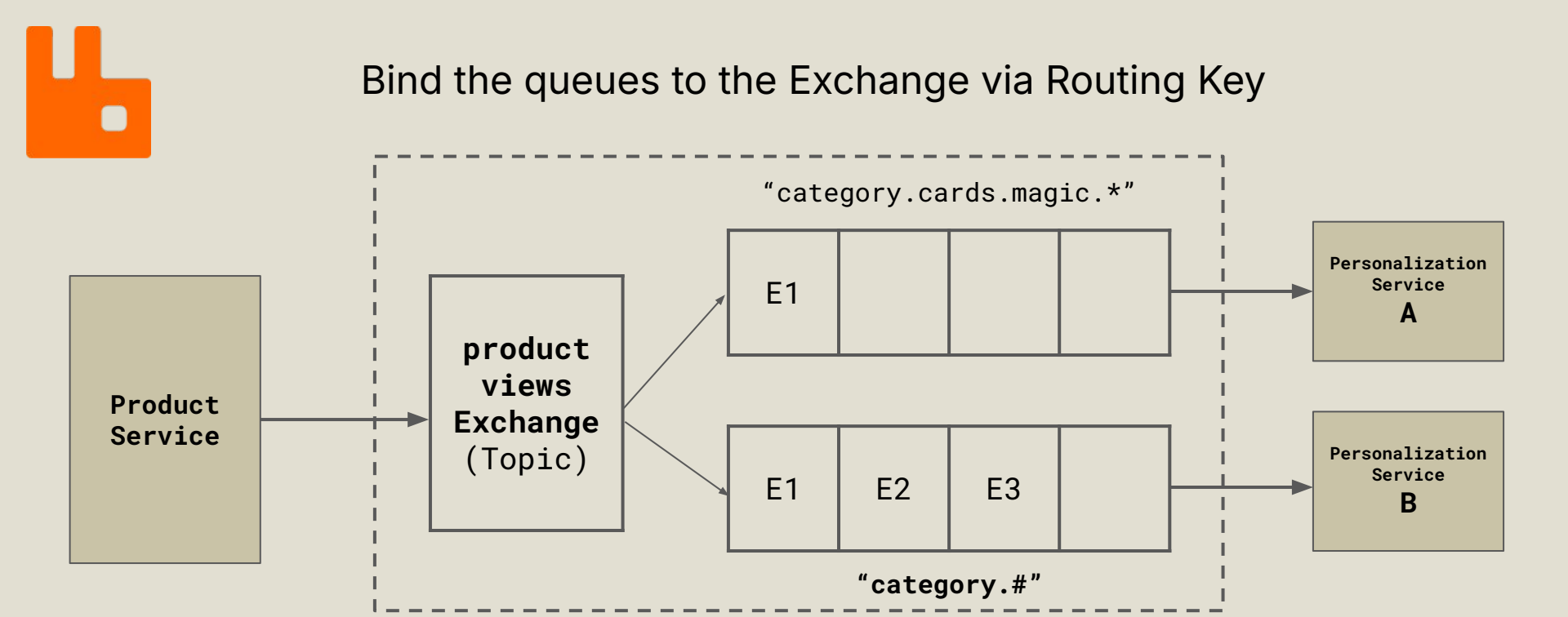
Bind the queues to the Exchange via Routing Key



E1: category.cards.magic.tarkir
E2: category.computers.laptop.apple
E3: category.computers.pc.dell

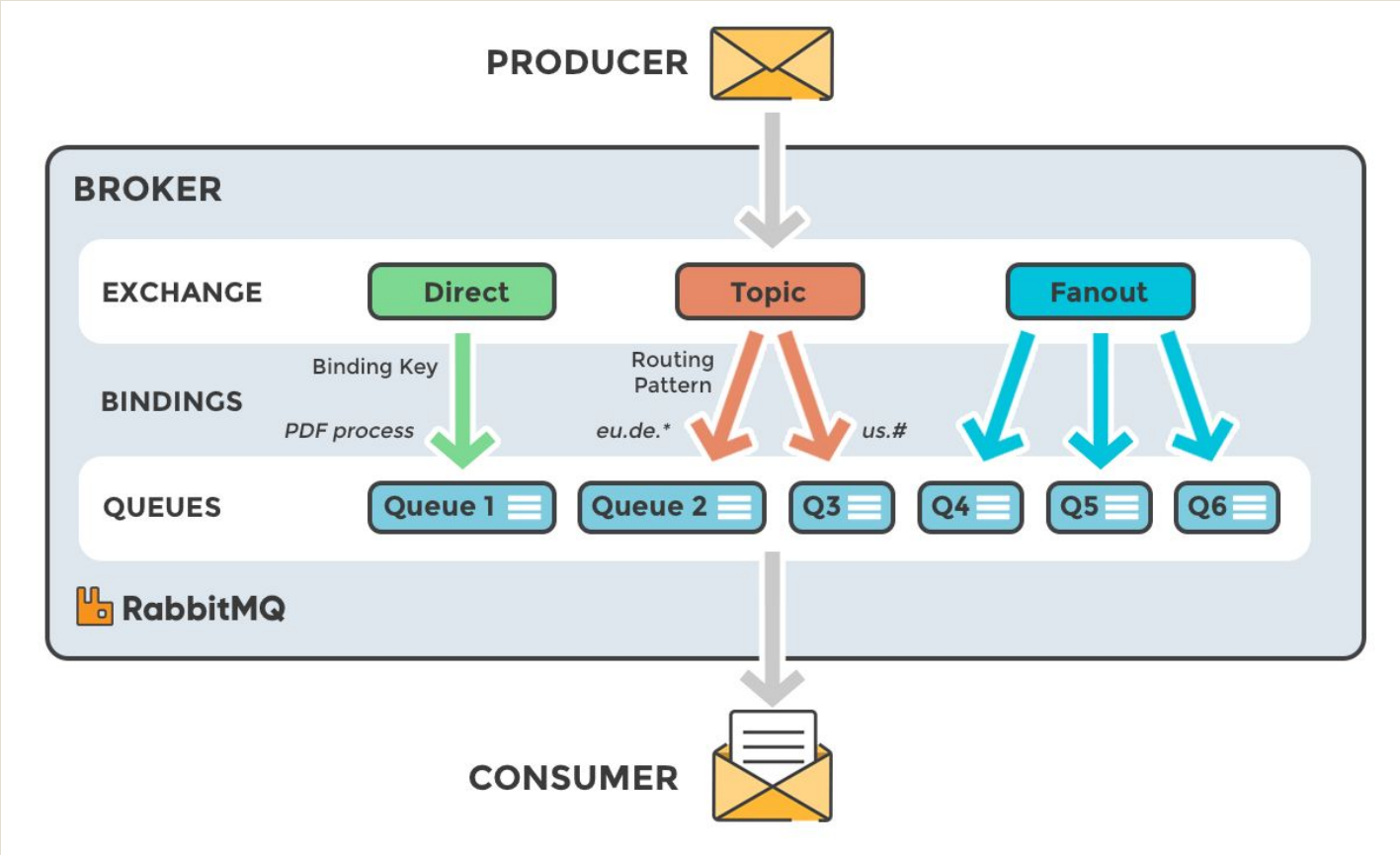
Topic Exchange: allows wildcard bindings

What about “`category.#`”?



E1: Browsed Magic Cards
E2: Browsed Macbook
E3: Browsed Dell PC

Topic Exchange: Wildcard bindings

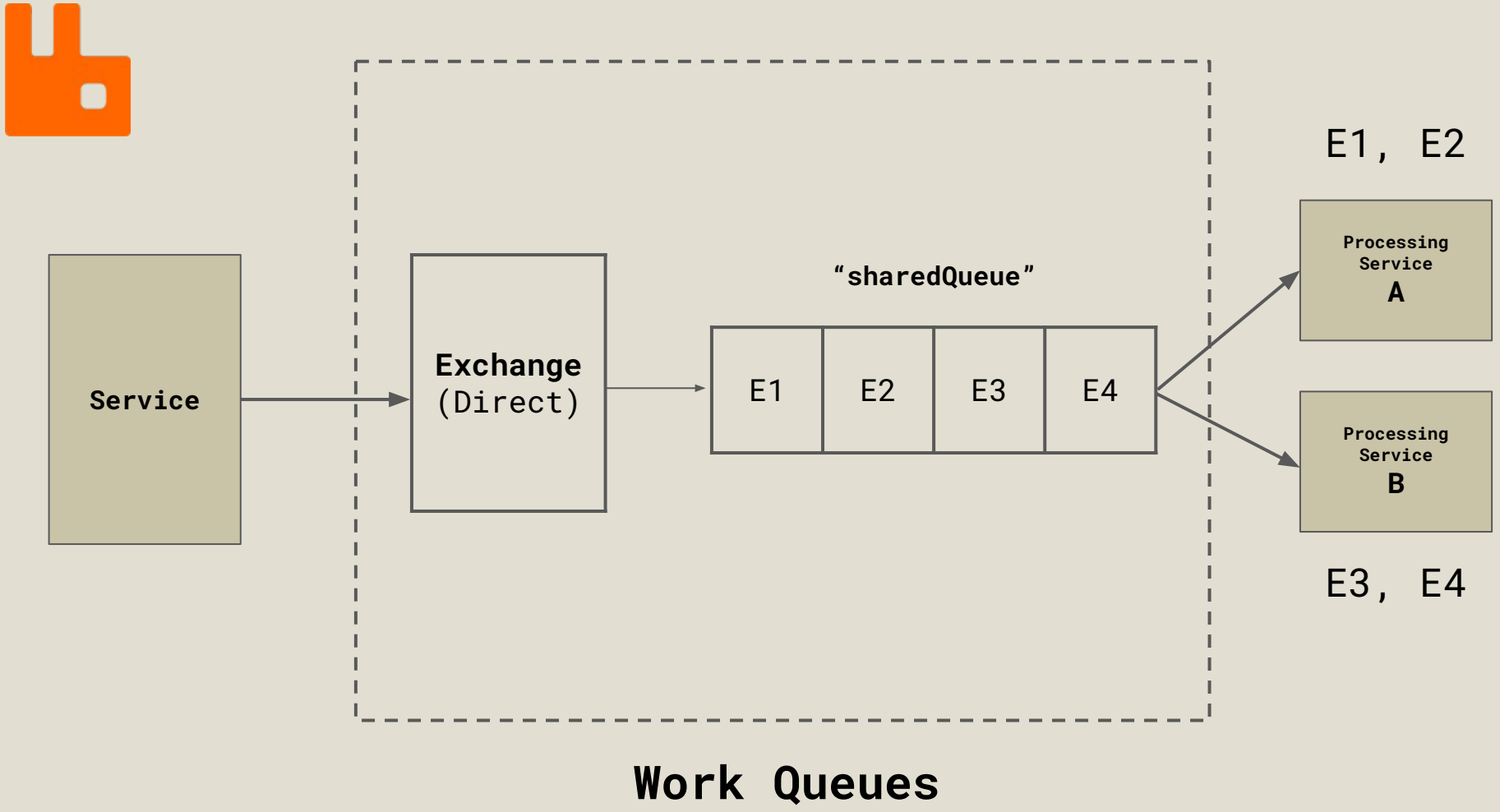


Direct, Topic, and Fanout Exchanges

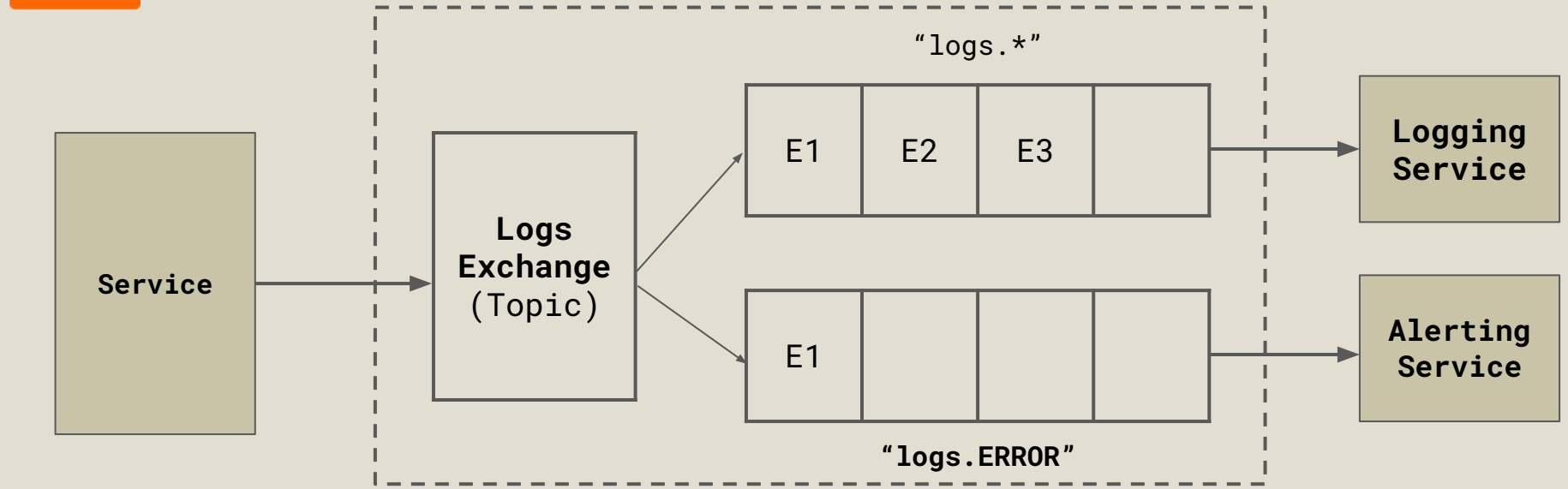
Exchange Type	Routing Key Supported	Use Cases
Fanout	N.A.	Receive everything from this exchange
Direct	“Routing Key” (no wildcards)	Receive events from selected keys without more fine-grained filters
Topic	“Routing Key” with wildcards, (* and #)	Allows fine-grained filters to only receive events of a subset of a routing key

RabbitMQ: Exchange Binding Summary

5.1.2 other common patterns


Service
Exchange
(Direct)
“sharedQueue”
E1
E2
E3
E4
E1, E2
Processing
Service
A
Processing
Service
B
E3, E4

Work Queues



E1: logs.ERROR
E2: logs.INFO
E3: logs.INFO

Error Logging: Alert and Logging

Peer Discussion

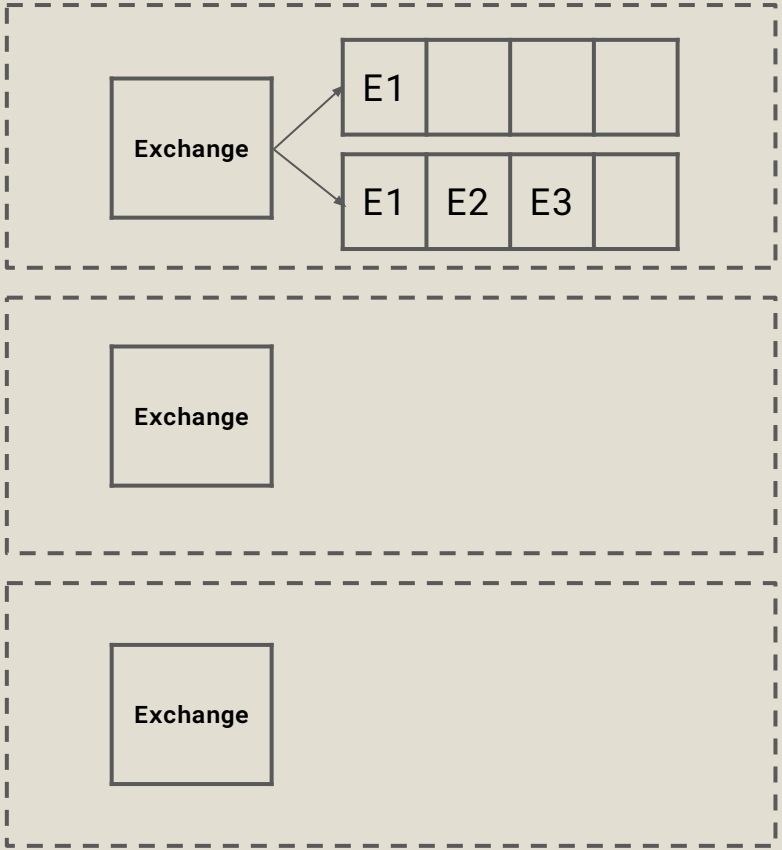
Identify at least 3 more possible use cases for RabbitMQ queues in pairs.

Which exchange pattern should I use? (Direct, Fanout, or Topic)

~10 mins, pick 1 person to share after discussing.

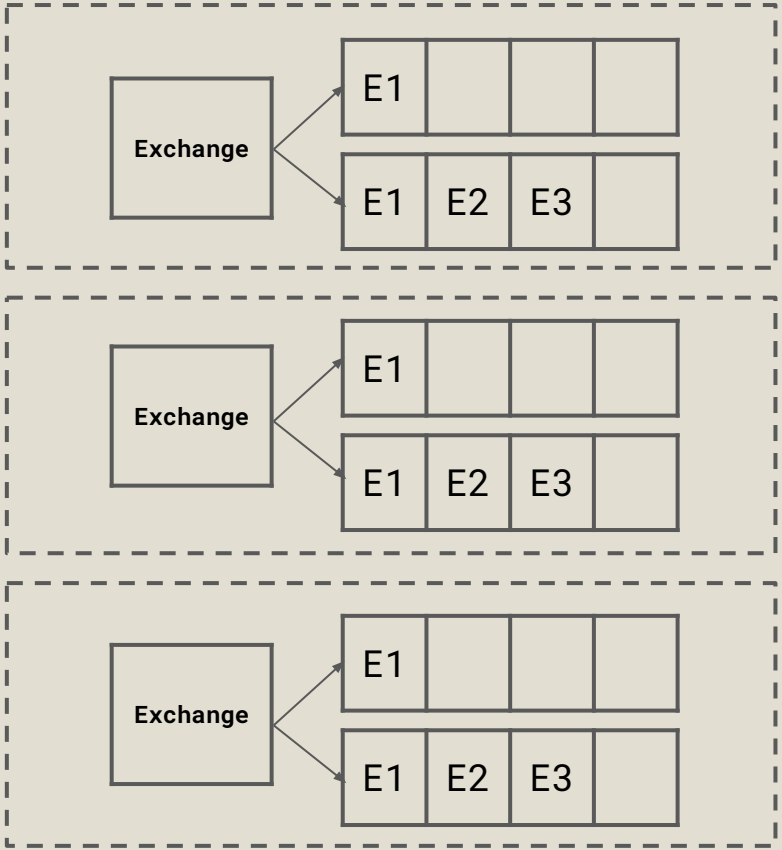
5.1.3 reliability options

Service

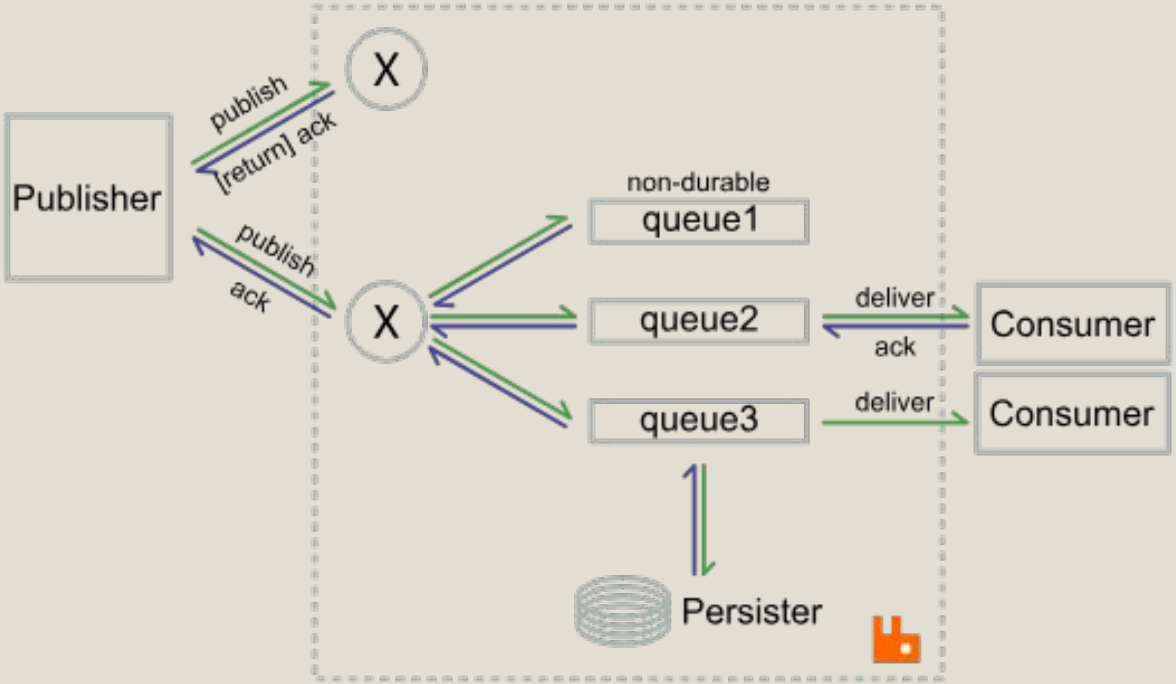


RabbitMQ Clustering
Queues not replicated by default

Service



RabbitMQ Clustering: Quorum Queues (Needs majority)



Publisher Confirms

https://www.rabbitmq.com/blog/2011/02/10/introducing-publisher-confirms

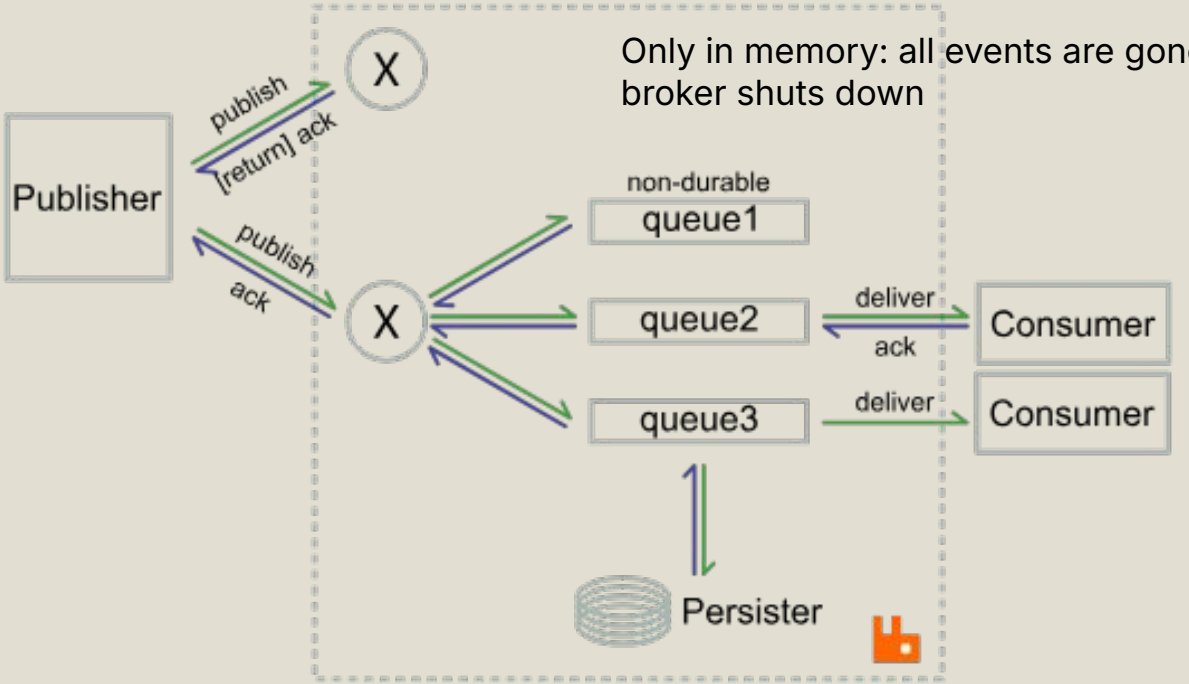
When the broker acknowledges a message, it assumes responsibility for it and informs the publisher that it has been handled successfully; what "handled successfully" means is context-dependent.

The basic rules are as follows:

- **an un-routable mandatory or immediate message is confirmed right after the *basic.return*;**
- **otherwise, a transient message is confirmed the moment it is enqueued; and,**
- **a persistent message is confirmed when it is persisted to disk or when it is consumed on every queue.**

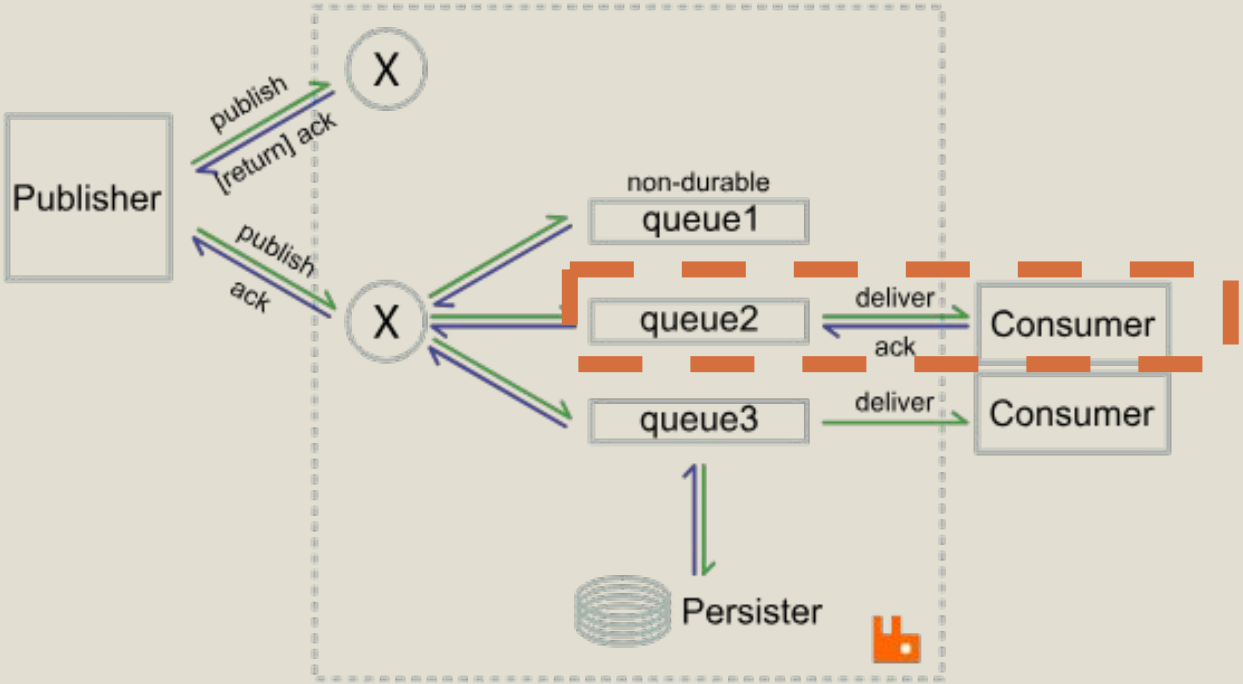
Publisher Confirms

https://www.rabbitmq.com/blog/2011/02/10/introducing-publisher-confirms



Publisher Confirms

https://www.rabbitmq.com/blog/2011/02/10/introducing-publisher-confirms



Consumer Acknowledgements

https://www.rabbitmq.com/docs/confirms#consumer-acknowledgements

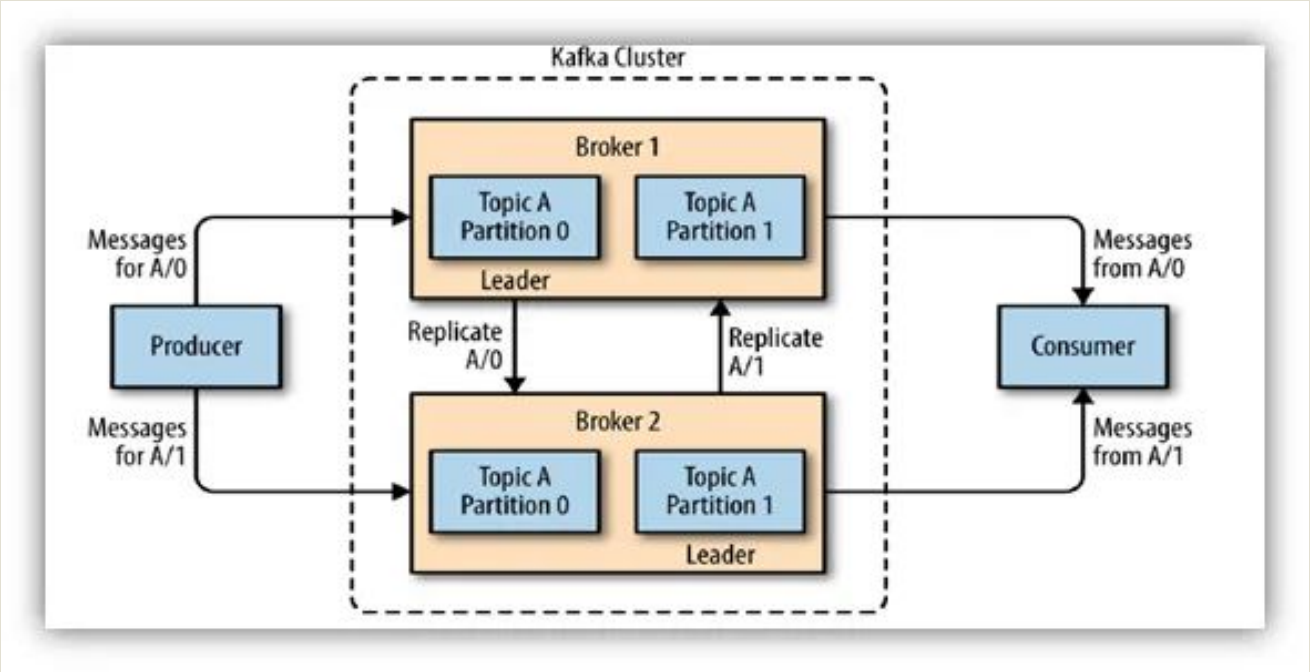
RabbitMQ: Summary

1. **Queues form the asynchronous data flow of distributed systems.** Data flows can be synchronous (eg. HTTP calls) or asynchronous.

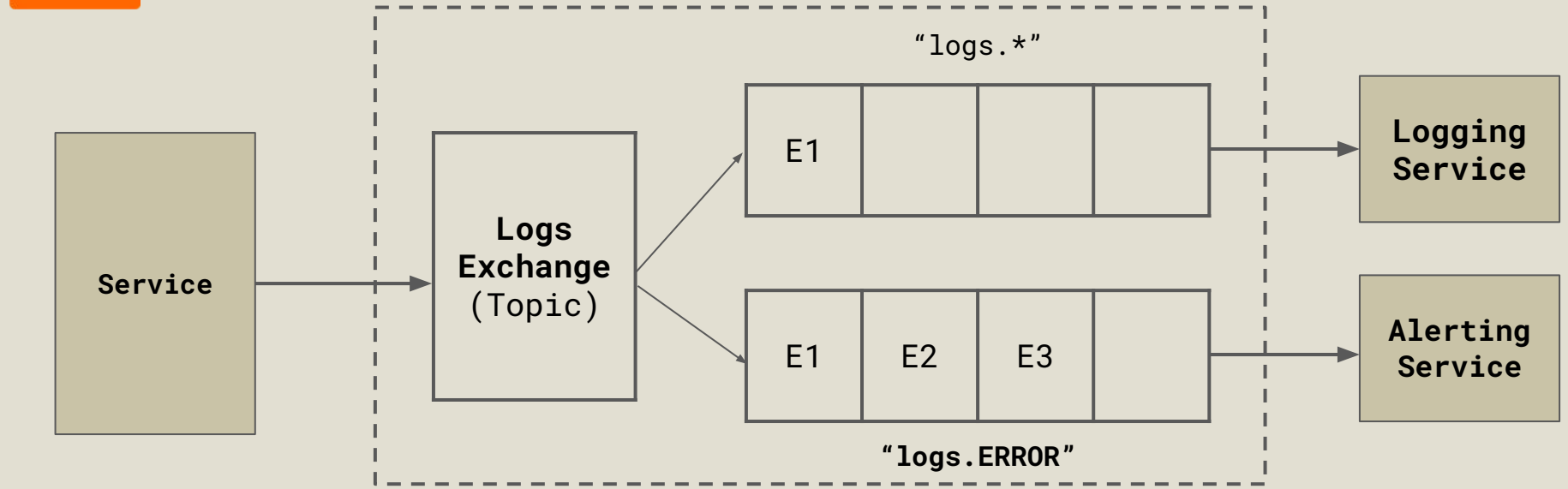
2. **Flexible Implementation:** Choose between Topic, Fanout, or Direct exchanges to determine how you wish to publish and consume events to implement patterns you want (eg. Work Queues, filtered events, wildcards)

3. **Cluster or don't**. Clustering is optional for RabbitMQ, and can be used as a lightweight solution, or a full-featured reliable event system with clustering, quorum queues.

5.2 kafka

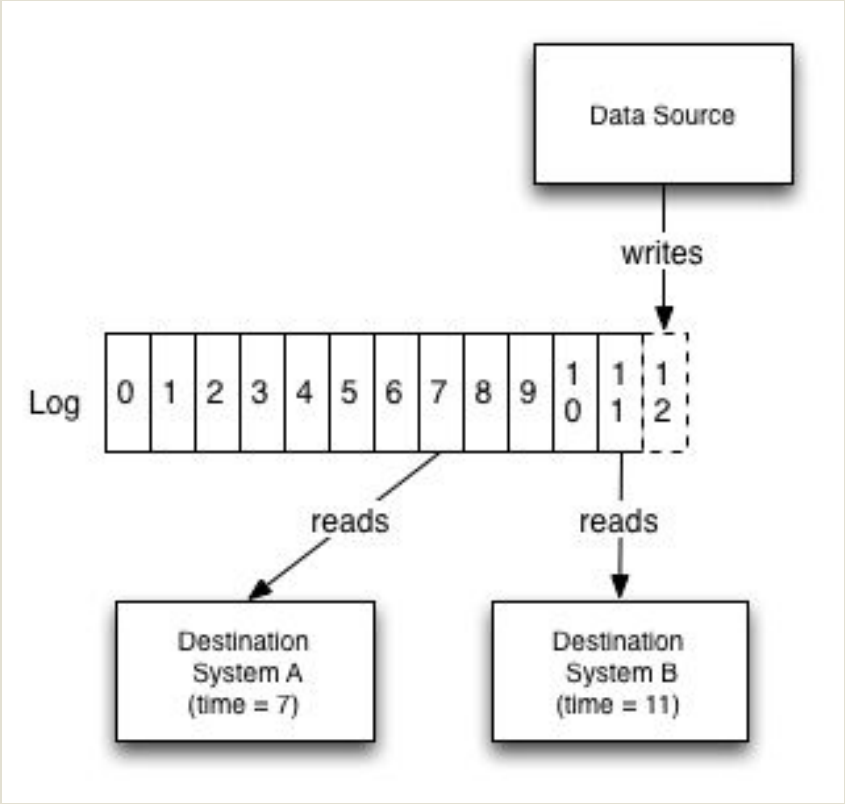


Kafka: Architecture



E1: logs.ERROR
E2: logs.INFO
E3: logs.INFO

RabbitMQ: Events Deleted After They are Consumed



Heart of Kafka: Append-Only Log

Append-only log in Raft

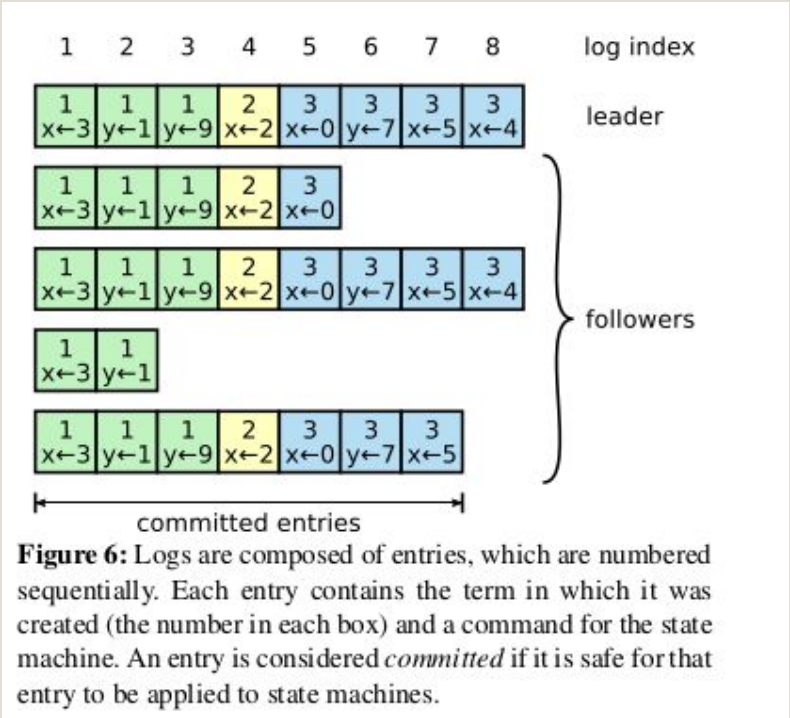
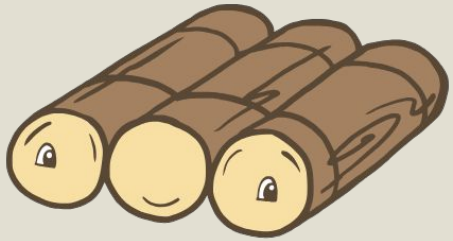


Figure 6: Logs are composed of entries, which are numbered sequentially. Each entry contains the term in which it was created (the number in each box) and a command for the state machine. An entry is considered *committed* if it is safe for that entry to be applied to state machines.



Raft

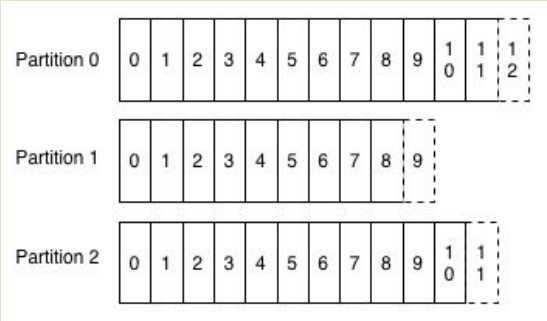
Raft Algorithm: Quick Summary

https://raft.github.io/raft.pdf

“We used a few tricks in Kafka to support this kind of scale:

- Partitioning the log
- Optimizing throughput by batching reads and writes
- Avoiding needless data copies

In order to allow horizontal scaling we chop up our log into partitions:”



Kafka: Architecture

https://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying

Distributes partitions amongst consumers in the same group

Topic, eg. “**product views**”

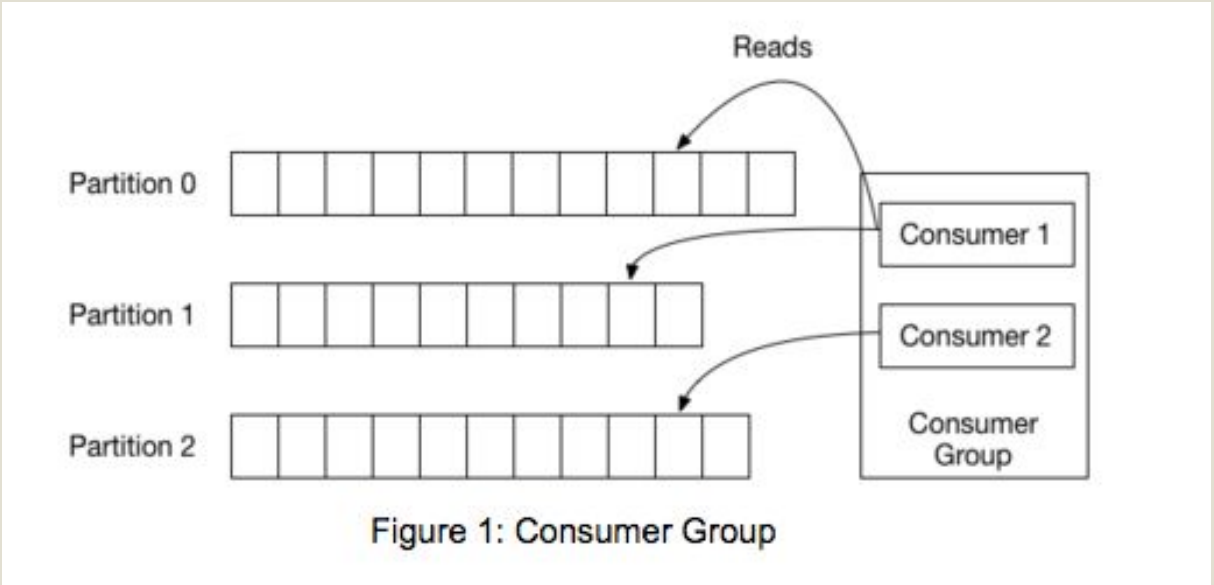


Figure 1: Consumer Group

Kafka: Topics and Consumer Group

https://www.confluent.io/en-gb/blog/tutorial-getting-started-with-the-new-apache-kafka-0-9-consumer-client/

Last “saved” checkpoint that Kafka knows this consumer has processed

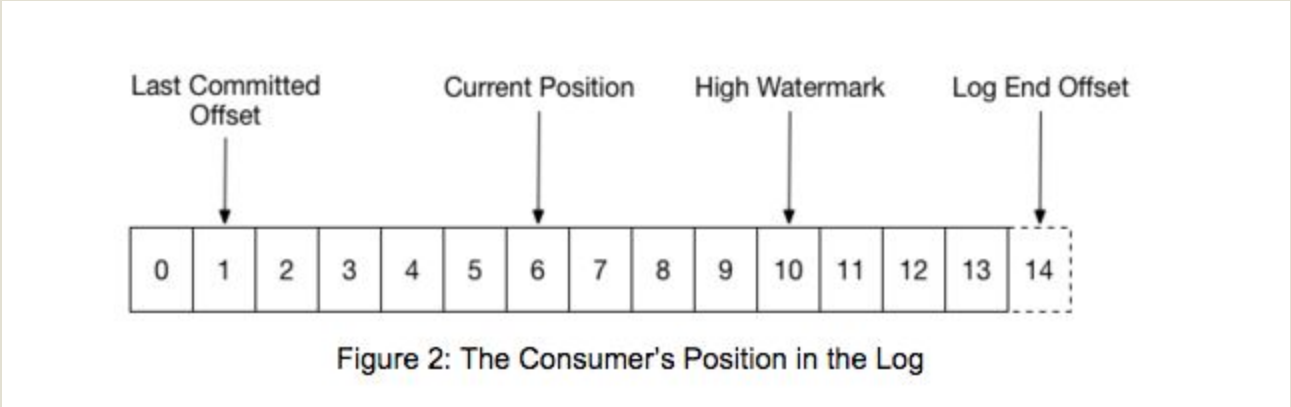


Figure 2: The Consumer's Position in the Log

Kafka: Topics and Consumer Group

https://www.confluent.io/en-gb/blog/tutorial-getting-started-with-the-new-apache-kafka-0-9-consumer-client/

"The high watermark is the offset of the last message that was successfully copied to all of the log's replicas."

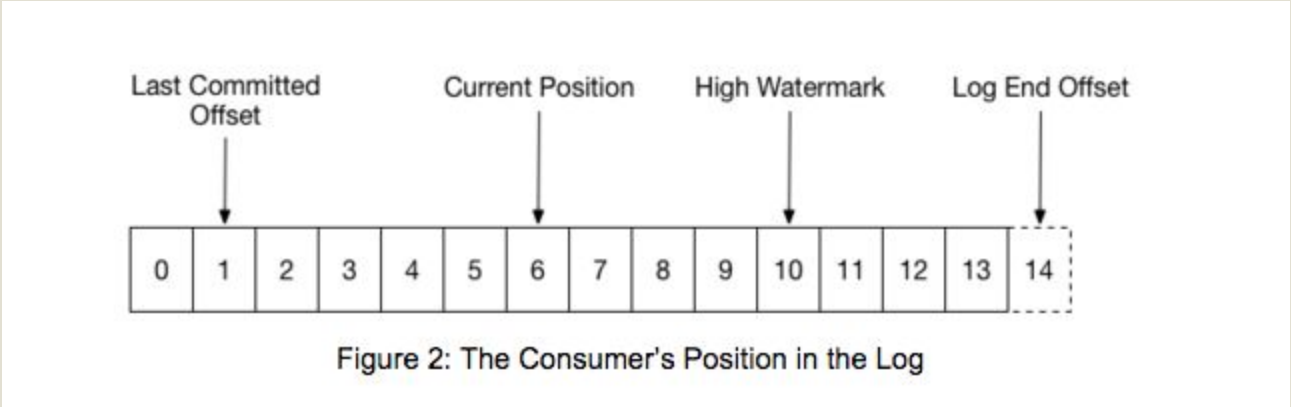


Figure 2: The Consumer's Position in the Log

Kafka: Topics and Consumer Group

https://www.confluent.io/en-gb/blog/tutorial-getting-started-with-the-new-apache-kafka-0-9-consumer-client/

"The log end offset is the offset of the last message written to the log."

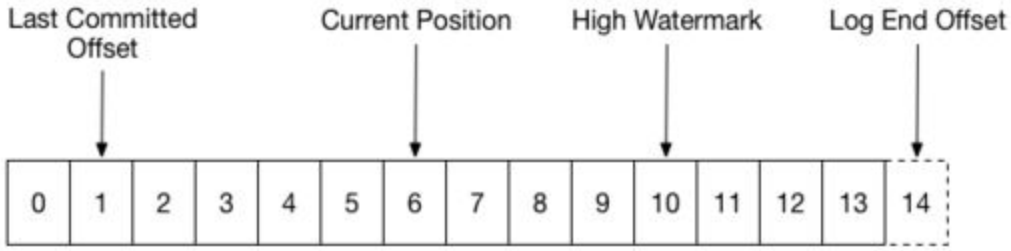


Figure 2: The Consumer's Position in the Log

Kafka: Topics and Consumer Group

https://www.confluent.io/en-gb/blog/tutorial-getting-started-with-the-new-apache-kafka-0-9-consumer-client/

Exchange Type	Routing Key Supported	Use Cases
Fanout	N.A.	Receive everything from this exchange
Direct	“Routing Key” (no wildcards)	Receive events from selected keys without more fine-grained filters
Topic	“Routing Key” with wildcards, (* and #)	Allows fine-grained filters to only receive events of a subset of a routing key

Can we do the same as RabbitMQ? Hint: No

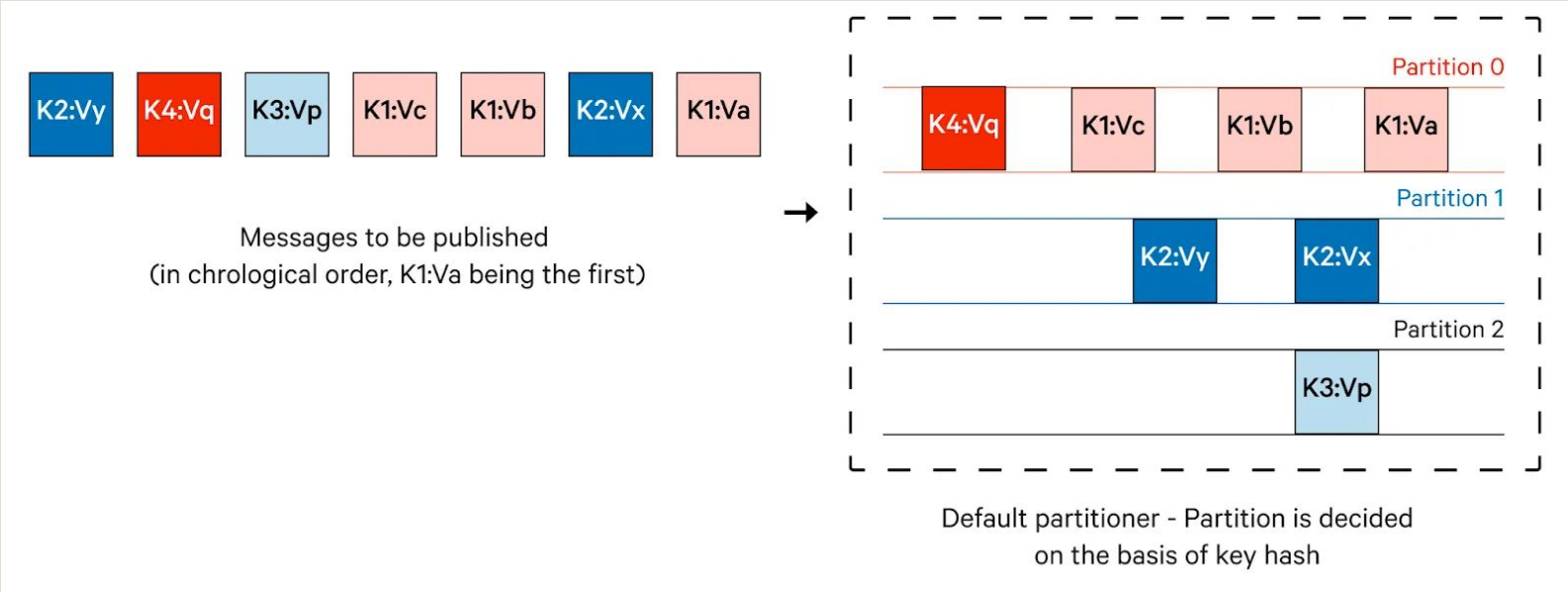
“If the producer specifies a key, Kafka applies a hash function to the key, which results in a numerical value.

This hash value is then used to determine which partition the message will be sent to.

In simpler terms, Kafka uses the key to ensure that all messages with the same key are sent to the same partition, allowing for grouping of related messages.”

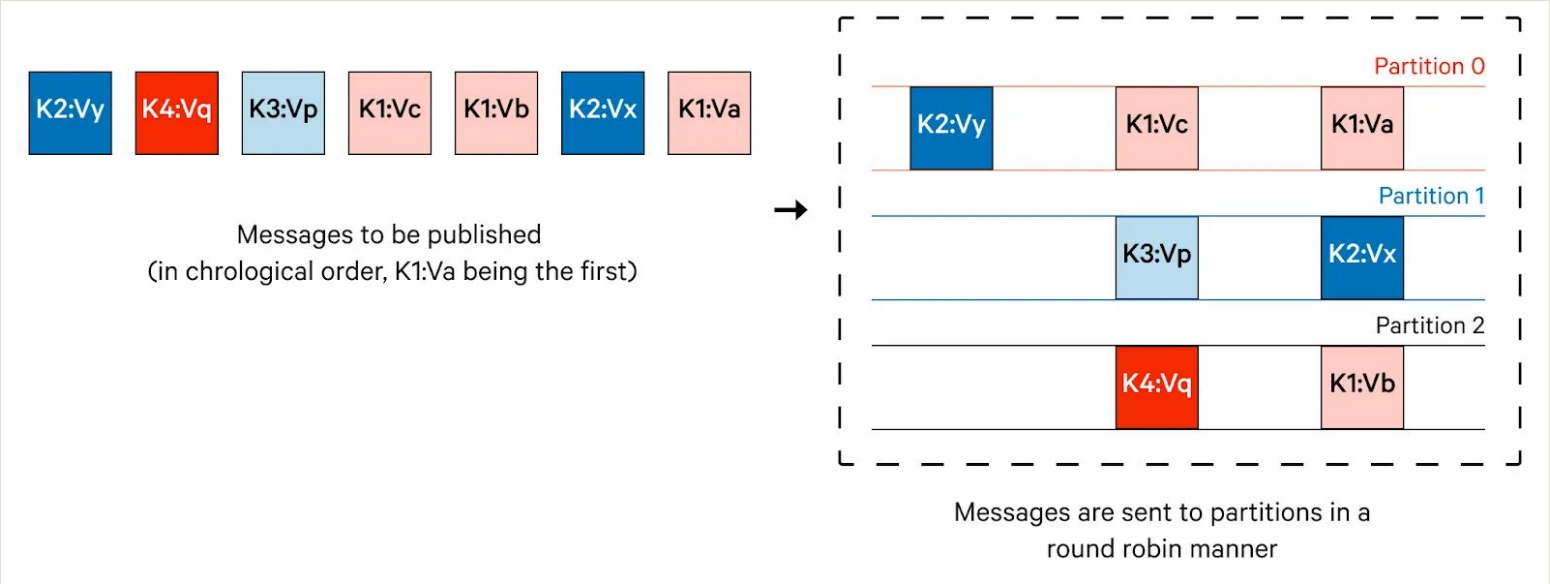
Kafka: Partitioning by Message Key

https://www.confluent.io/learn/kafka-message-key/



Kafka: Default Partitioner

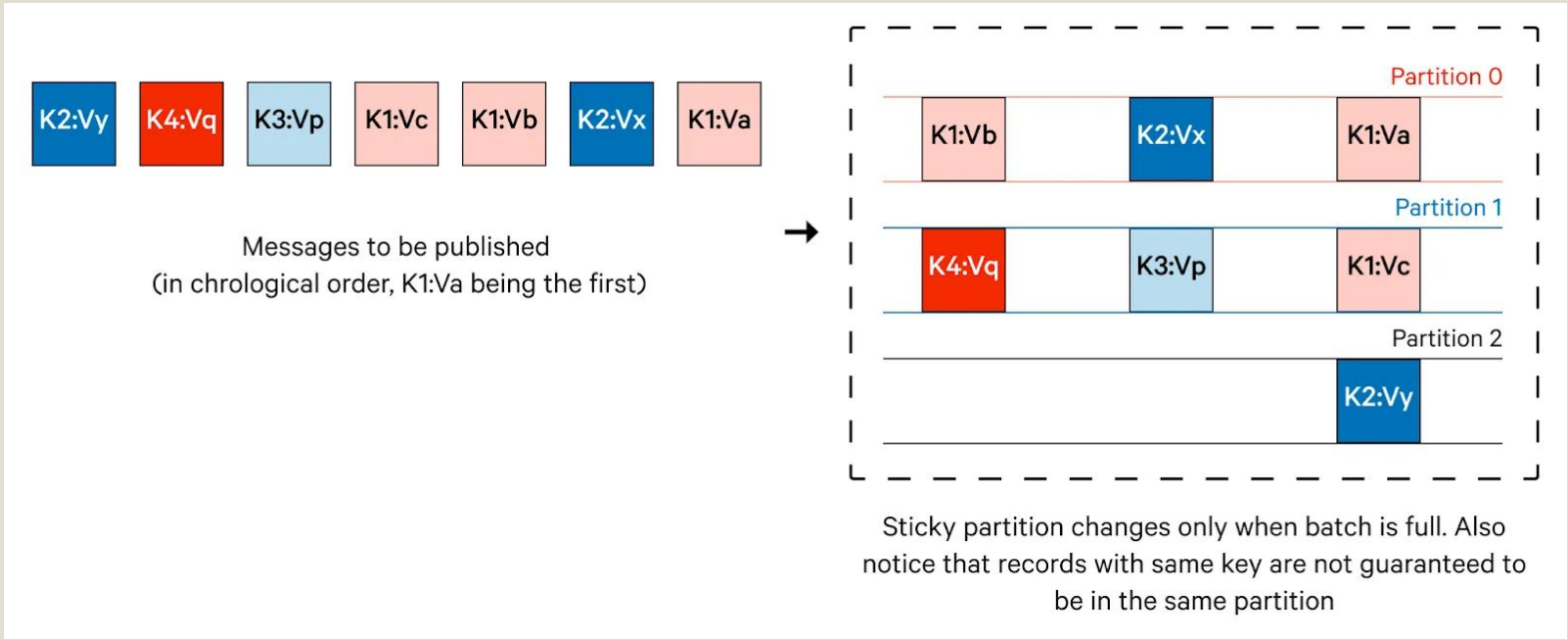
https://www.redpanda.com/guides/kafka-tutorial-kafka-partition-strategy



Kafka: Round Robin Partitioner

https://www.redpanda.com/guides/kafka-tutorial-kafka-partition-strategy

Similar to round-robin, but in batches



Kafka: Uniform Sticky Partitioner

https://www.redpanda.com/guides/kafka-tutorial-kafka-partition-strategy

```
public class PowerUserPartitioner implements Partitioner {

 public void configure(Map<String, ?> configs) {}

 public int partition(String topic, Object key, byte[] keyBytes,
 Object value, byte[] valueBytes,Cluster cluster) {

 List<PartitionInfo> partitions = cluster.partitionsForTopic(topic);
 int numPartitions = partitions.size();
 if ((keyBytes == null) || (!(key instanceOf String)))
      throw new InvalidRecordException("Record must have a valid string key");

 if (((String) key).equals("CEO"))
      return numPartitions - 1; // Messages with key “CEO” will always go to the last
partition

 // Other records will get hashed to the rest of the partitions
 return Math.abs(Utils.murmur2(keyBytes)) % (numPartitions - 1);
  }
}
```

Kafka: Custom Partitioner

https://www.redpanda.com/guides/kafka-tutorial-kafka-partition-strategy

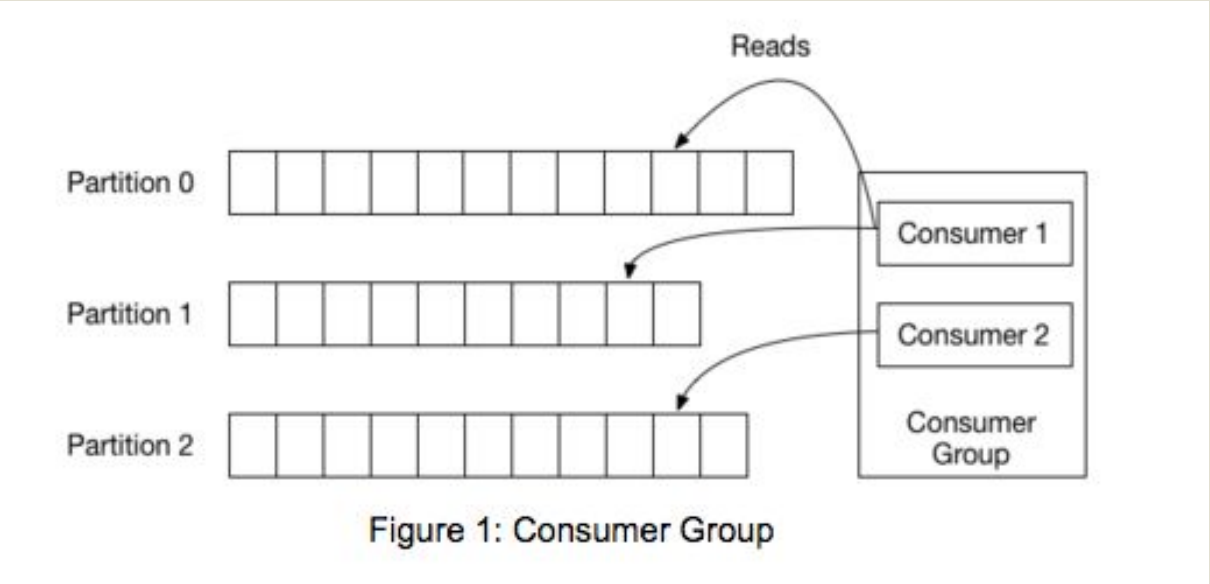
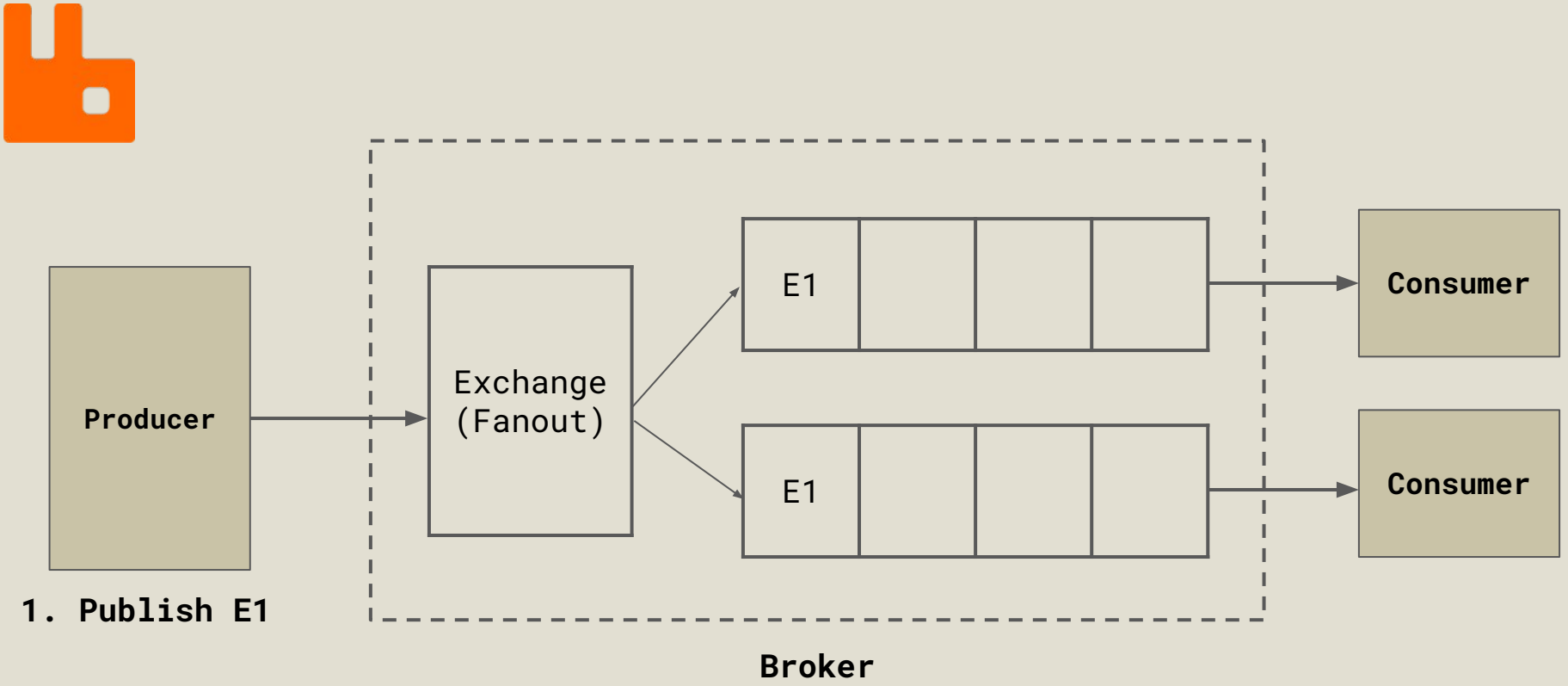


Figure 1: Consumer Group

Kafka: Suitable for Large Scale Fanout at High Throughput



Fanout: More Troublesome with RabbitMQ
Need to bind queues to consumers

Version: 4.0

Streams and Super Streams (Partitioned Streams)

What is a Stream

RabbitMQ Streams is a persistent replicated data structure that can complete the same tasks as queues: they buffer messages from producers that are read by consumers. However, streams differ from queues in two important ways: how messages are stored and consumed.

Streams model an append-only log of messages that can be repeatedly read until they expire. Streams are always persistent and replicated. A more technical description of this stream behavior is “non-destructive consumer semantics”.

To read messages from a stream in RabbitMQ, one or more consumers subscribe to it and read the same messages as many times as they want.

RabbitMQ: Streams offer something similar

5.2.1 reliability options

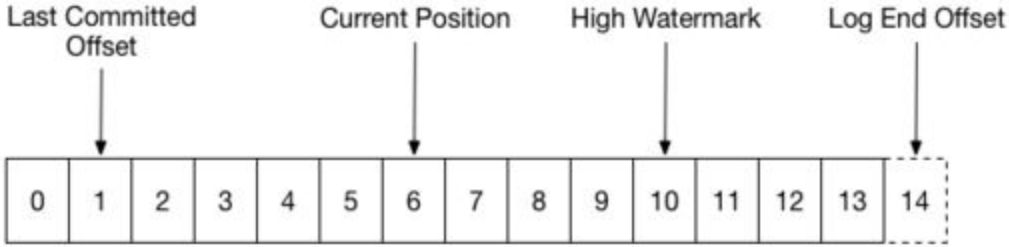


Figure 2: The Consumer's Position in the Log

Consumer crashes: what happens?

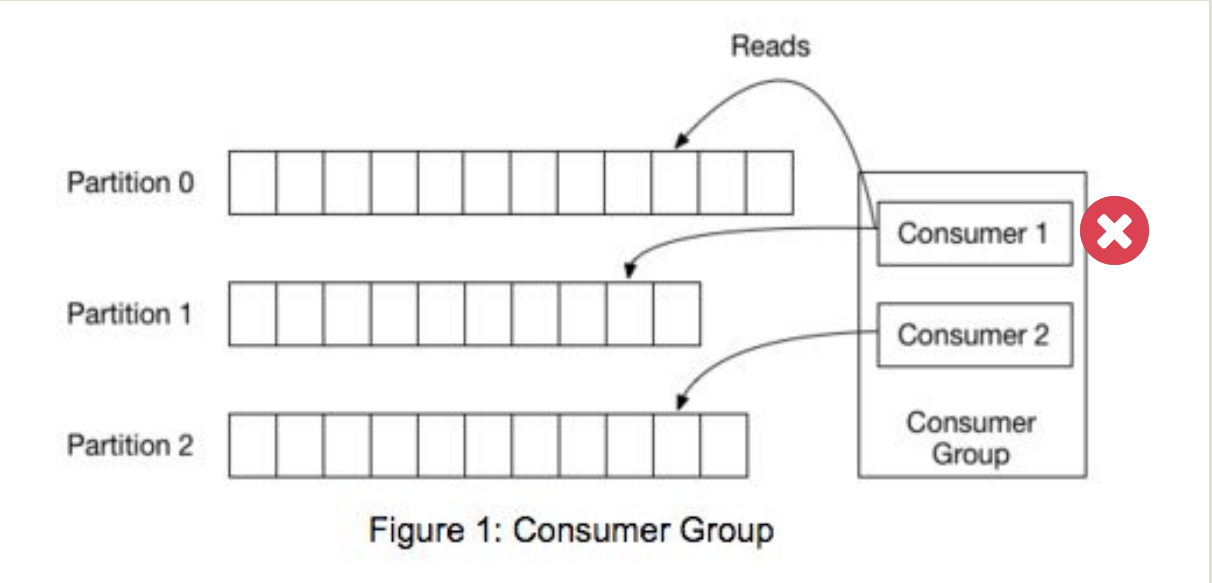


Figure 1: Consumer Group

Kafka reassigns another consumer in the same group to take over

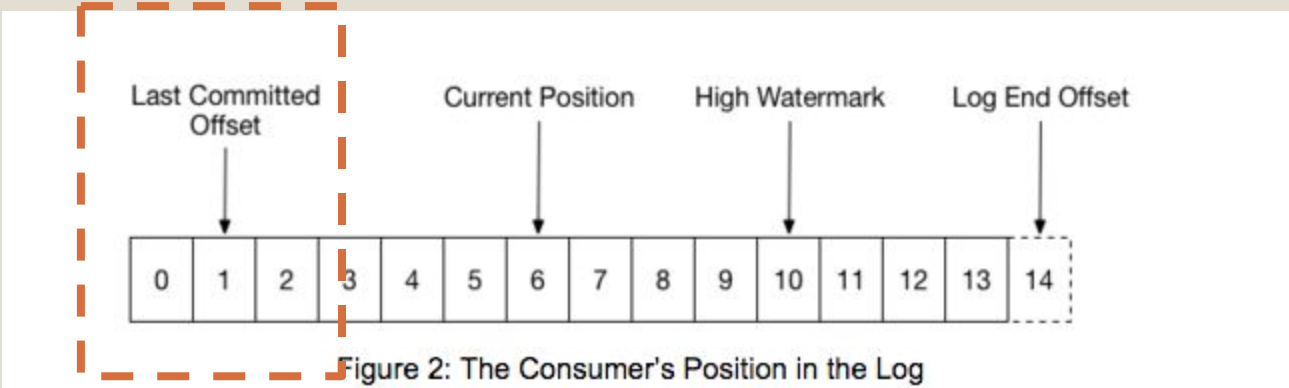


Figure 2: The Consumer's Position in the Log

New Consumer restarts from offset 1

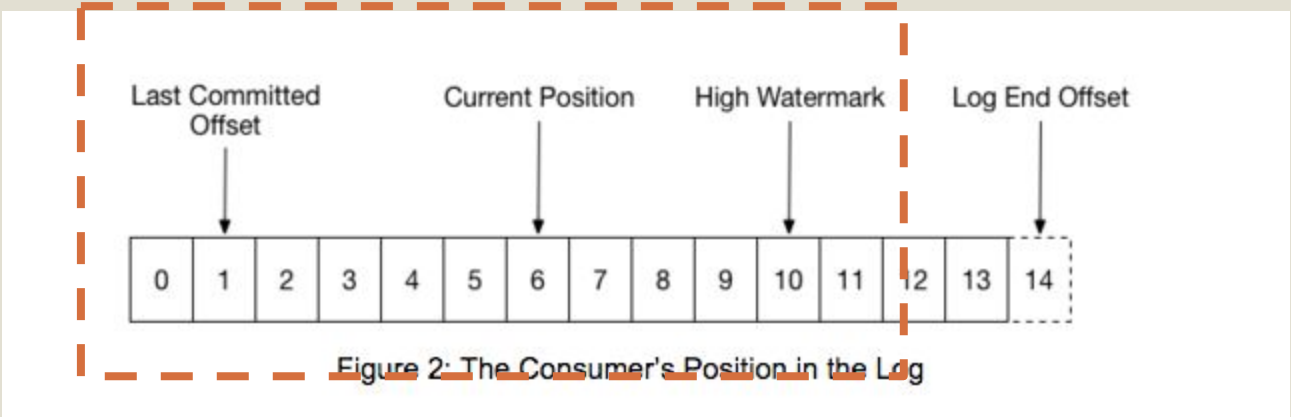
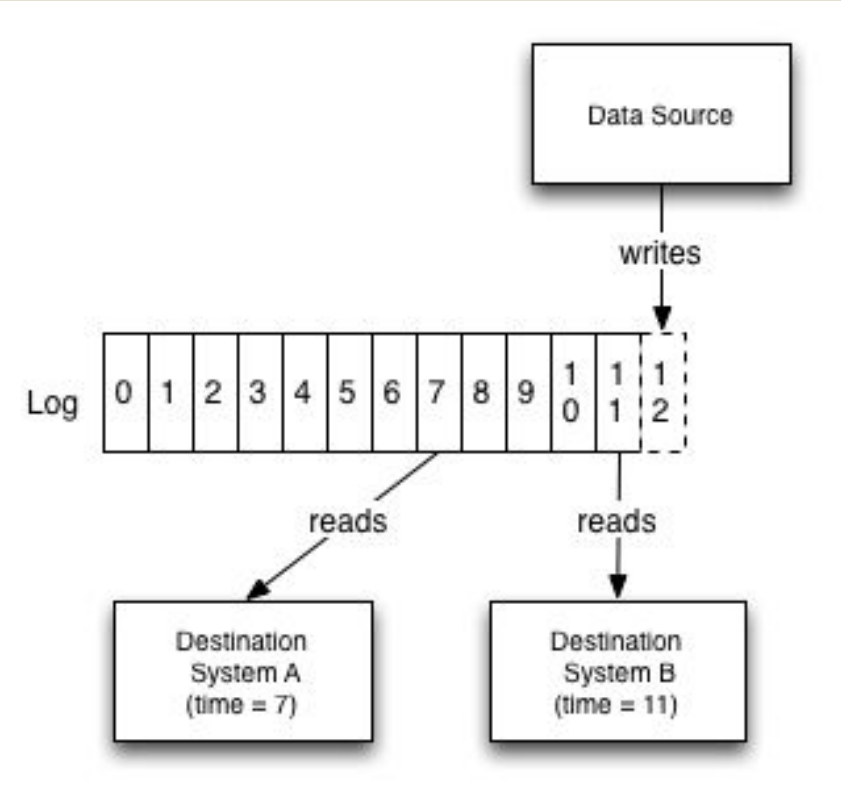


Figure 2: The Consumer's Position in the Log

Can process until high watermark point

auto.commit.interval.ms. The default interval is 5 seconds.

Consumer offset default: Auto Commit 5 seconds



Publisher Acknowledgement

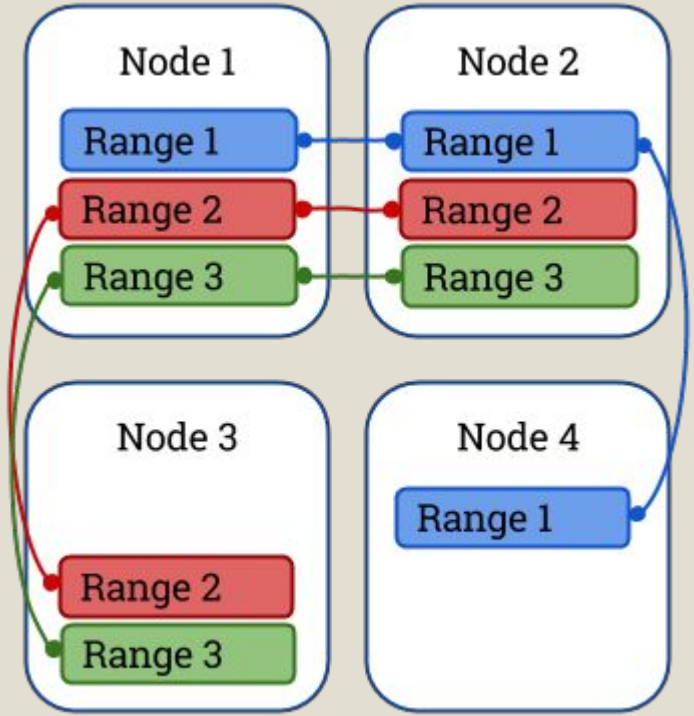
`acks=0` (No acknowledgment): The producer doesn't wait for any acknowledgment. It sends the message and doesn't care if it's received.

`acks=1` (Leader acknowledgment): The producer waits for an acknowledgment from the leader broker of the topic partition where the message is sent. This means the message is considered sent once the leader broker confirms receipt.

`acks=all` (Replica acknowledgment): The producer waits for acknowledgment from all in-sync replicas of the partition. This provides the highest level of durability but can introduce more latency.

Publisher Acknowledgement: Levels

https://medium.com/@monish.koppa/message-acknowledgement-in-apache-kafka-8daa8524c8e7



Range Replication in CockroachDB

Number of Nodes and **Number of Replicas (Replication Factor)** available determines the number of node failures we can tolerate.

Number of replicas = 3 by default in CockroachDB

Range Replication on Nodes

https://www.cockroachlabs.com/blog/the-new-stack-meet-cockroachdb-the-resilient-sql-database/

“Apache Kafka replicates the event log for each topic’s partitions across a configurable number of servers.

This replication factor is configured at the topic level, and the unit of replication is the topic partition.

This enables automatic failover to these replicas when a server in the cluster fails so messages remain available.”

Kafka: Replication Factor

https://kafka.apache.org/documentation/#replication

Topics are added and modified using the topic tool:

```
$ bin/kafka-topics.sh --bootstrap-server localhost:9092
--create --topic my_topic_name \
    --partitions 20 --replication-factor 3 --config x=y
```

Kafka: Replication Factor

https://kafka.apache.org/documentation/#replication

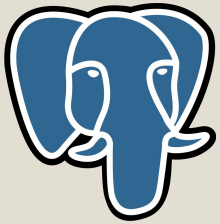
Kafka: Summary

1. **Kafka's key data structure is the append-only log.** Append-only logs allow for high-throughput fanout event systems to be built easily.

2. **Different Partitioners provide flexibility for message routing to different partitions**. If required, you can write a custom partitioner to give you the ability to to control how you wish to balance your partitions.

3. **Consumer groups and offset tracking provide flexibility for consumers to stop and resume**. Kafka can easily reassign consumers to replace dead consumers and resume events processing.

p.s.
Current Trends

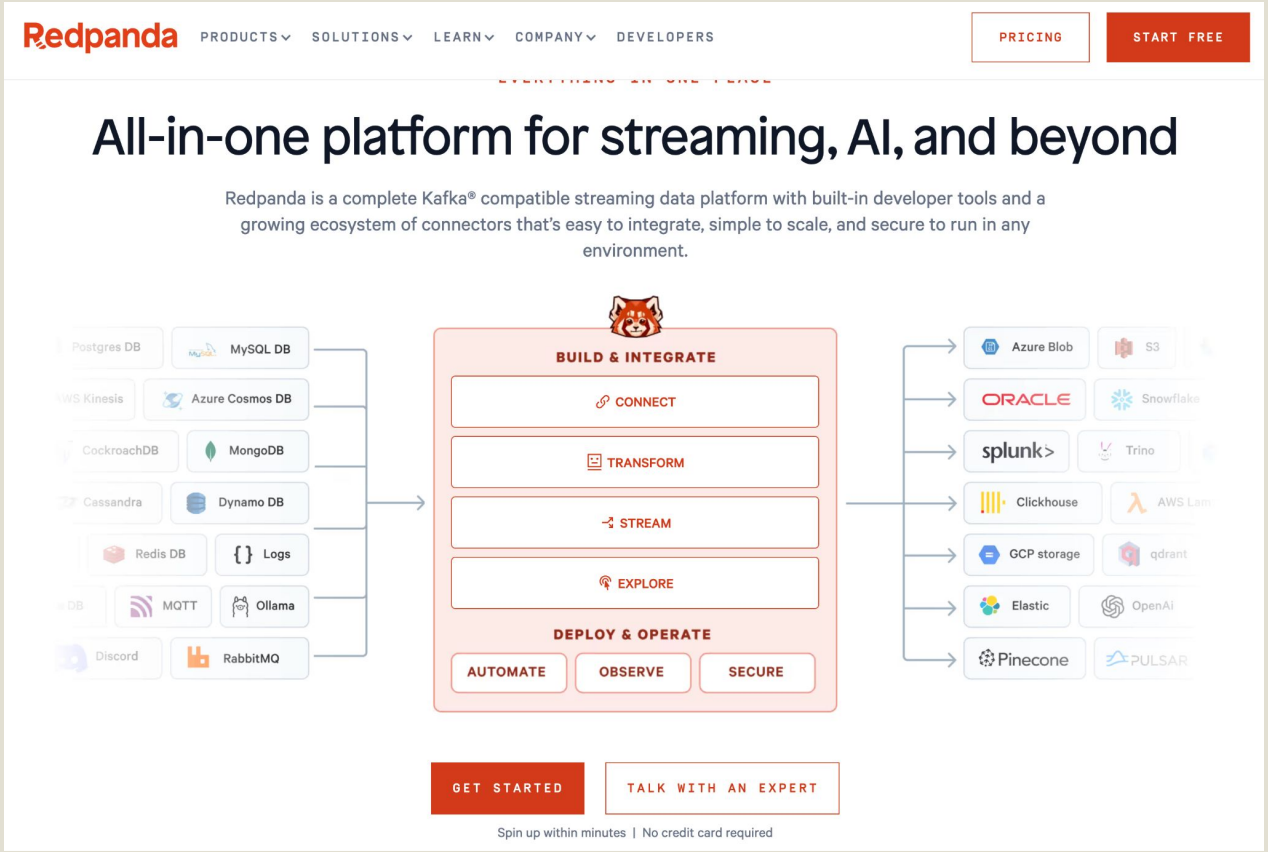


```
LISTEN virtual;
NOTIFY virtual;
Asynchronous notification "virtual" received from server process with
PID 8448.

NOTIFY virtual, 'This is the payload';
Asynchronous notification "virtual" with payload "This is the
payload" received from server process with PID 8448.
```

PostgreSQL for Events

https://www.postgresql.org/docs/current/sql-notify.html

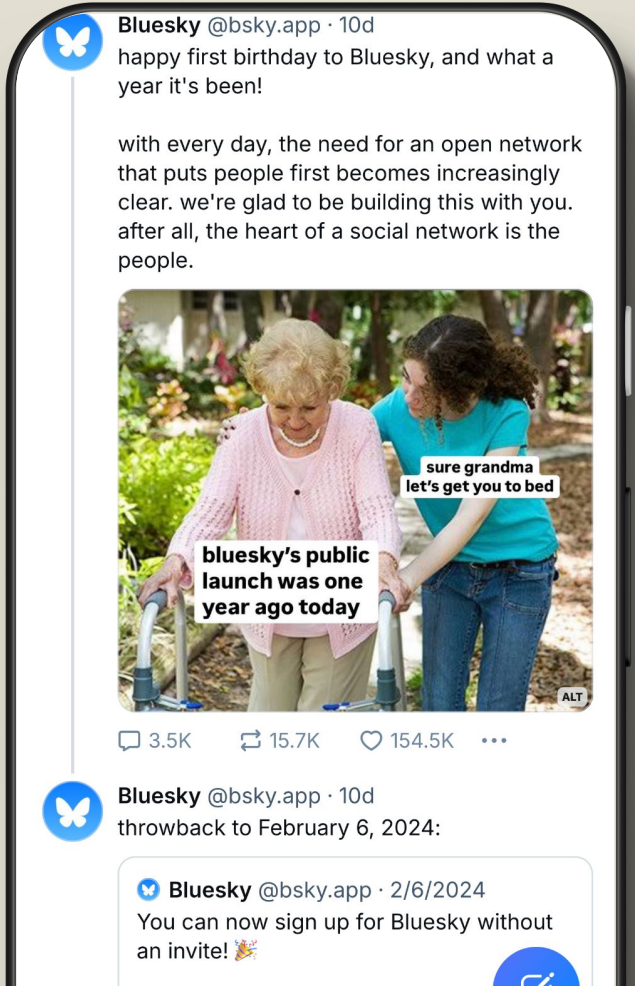


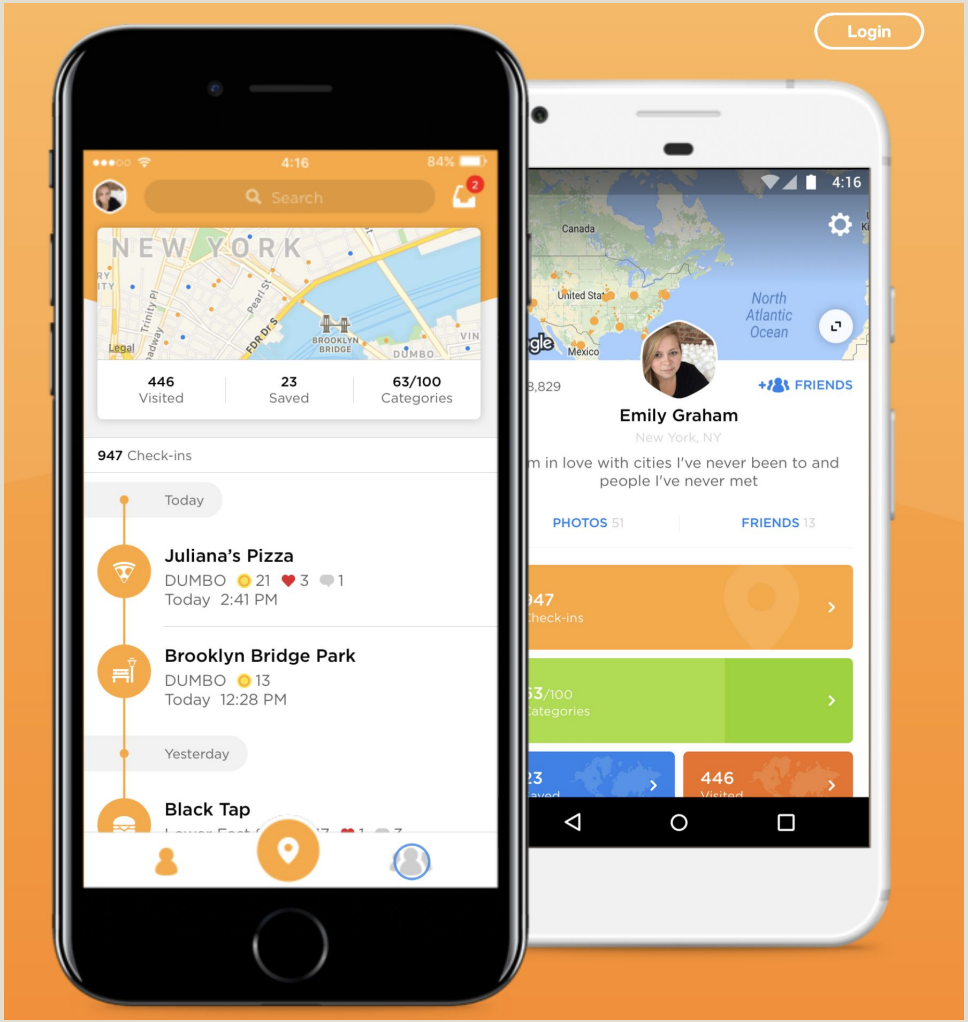
Redpanda: Kafka Compatible

https://www.redpanda.com/

Assignment

Extend a Distributed Social Media Platform




Login
4:16
84%
Search
NEW YORK
446
Visited
23
Saved
63/100
Categories
947 Check-ins
Today
Juliana's Pizza
DUMBO 21 3 1
Today 2:41 PM
Brooklyn Bridge Park
DUMBO 13
Today 12:28 PM
Yesterday
Black Tap
4:16
Canada
North Atlantic Ocean
Mexico
FRIENDS
Emily Graham
New York, NY
PHOTOS 51
FRIENDS 13
446

Assignment

W1 → Distributed Social Media Research
W2 → Review W1 + Build PoC
W3 → Design 1st Draft of System + Sharing
W4 → Implementation
W5 → Implementation (Queues)
W6 → Implementation (Load Testing)
W7 → Technical Retrospective
W8 → Complete System Implementation

Assignment #4

- Integrate an event / queue system into your system for at least one or more of these use cases:
  - Handle and process incoming Posts or events, eg. Firehose or Jetstream for AT Protocol
  - Process outgoing Posts or events, eg. Outbox for ActivityPub
  - Any other use cases you can think of

- Share your implementation in code, and share a link to your project (eg. GitHub or Gitlab)