



# Apache Kafka

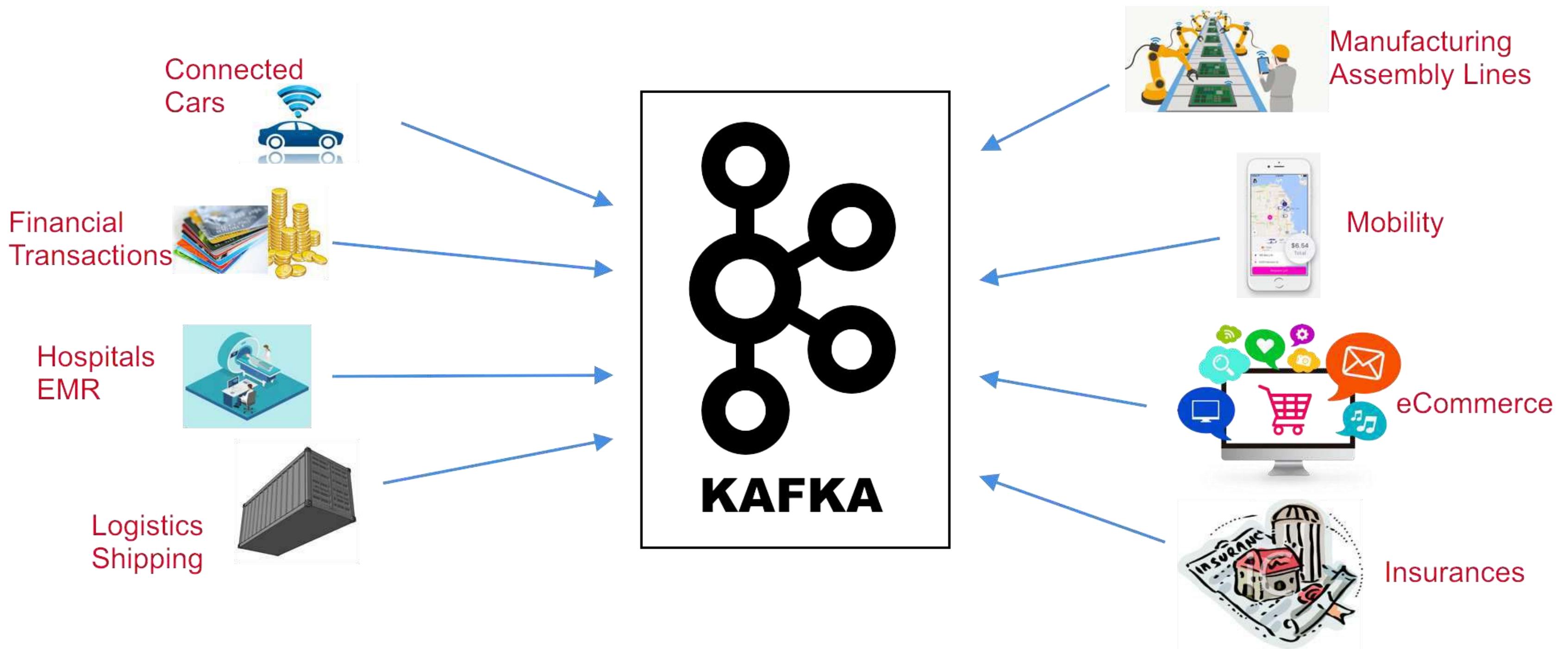
Architecture, API, Development, Debugging

# Agenda

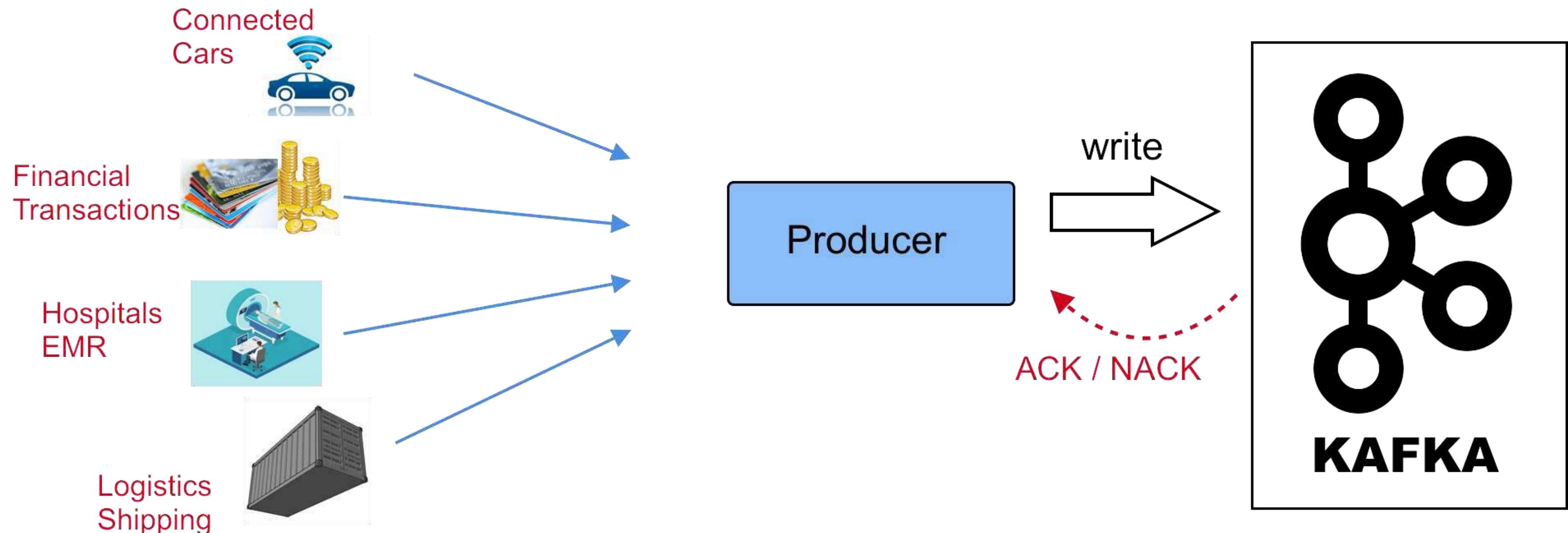
- Topics Covered
  - **Automating Certificate Management (in Kafka on the Cloud -MKS and Cloudwatch (Terraform integration)**
  - **Automation – Topic Creation, Schema Registries, CI/CD Pipelines**
  - **Kafka Observability and Log Correlation**
  - **Replays and Offsets**

# Core Kafka Architecture & Understanding

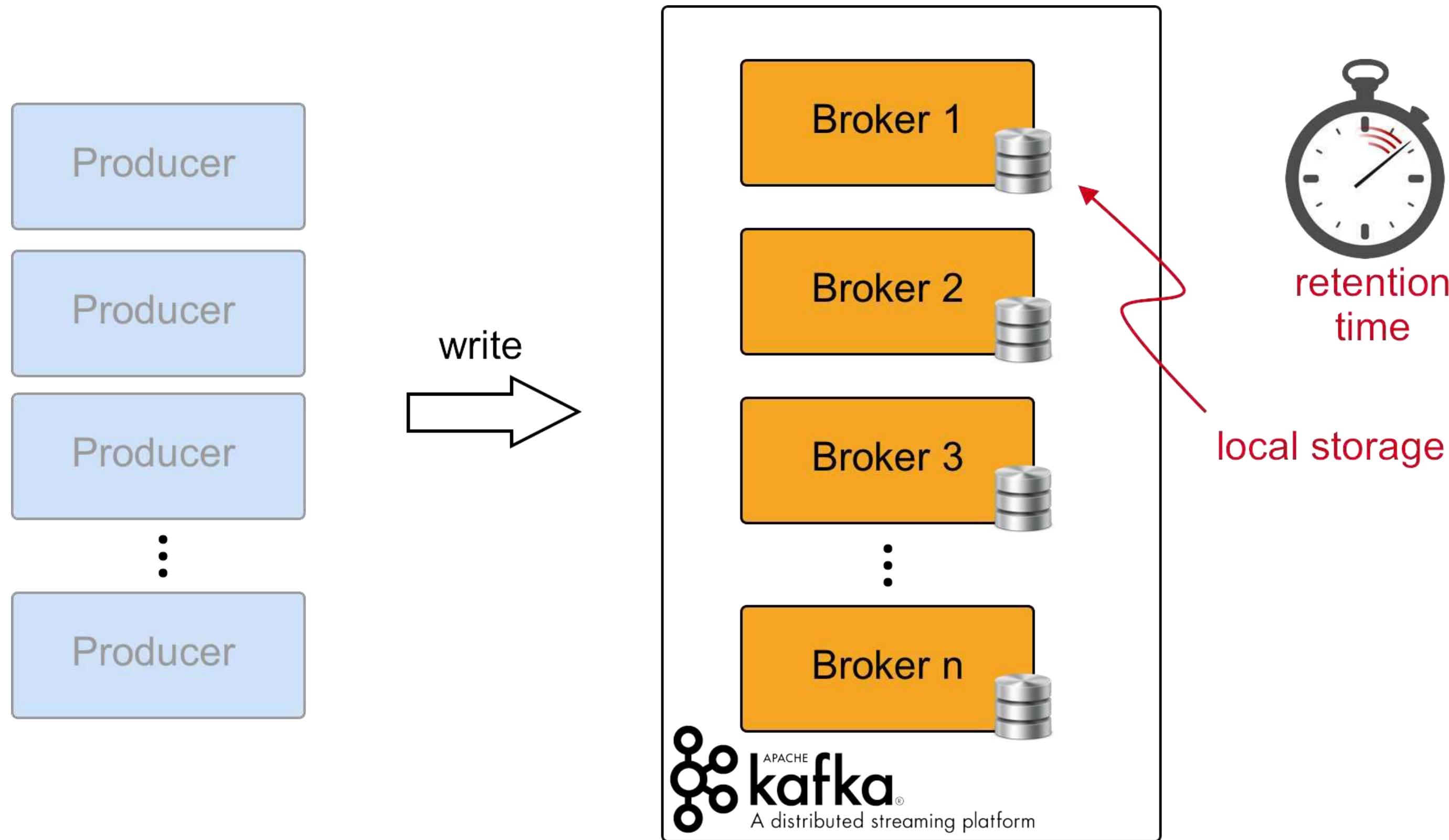
# The World Produces Data



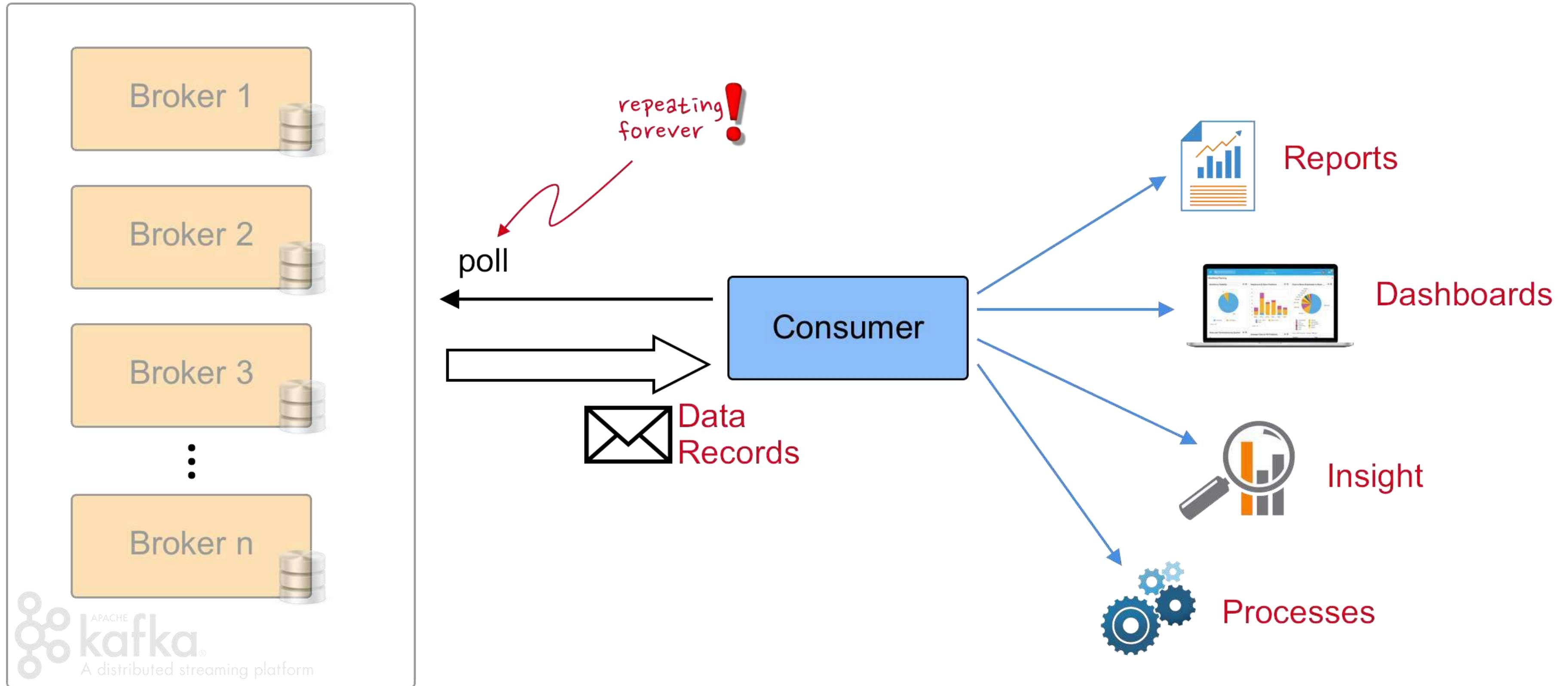
# Producers



# Brokers

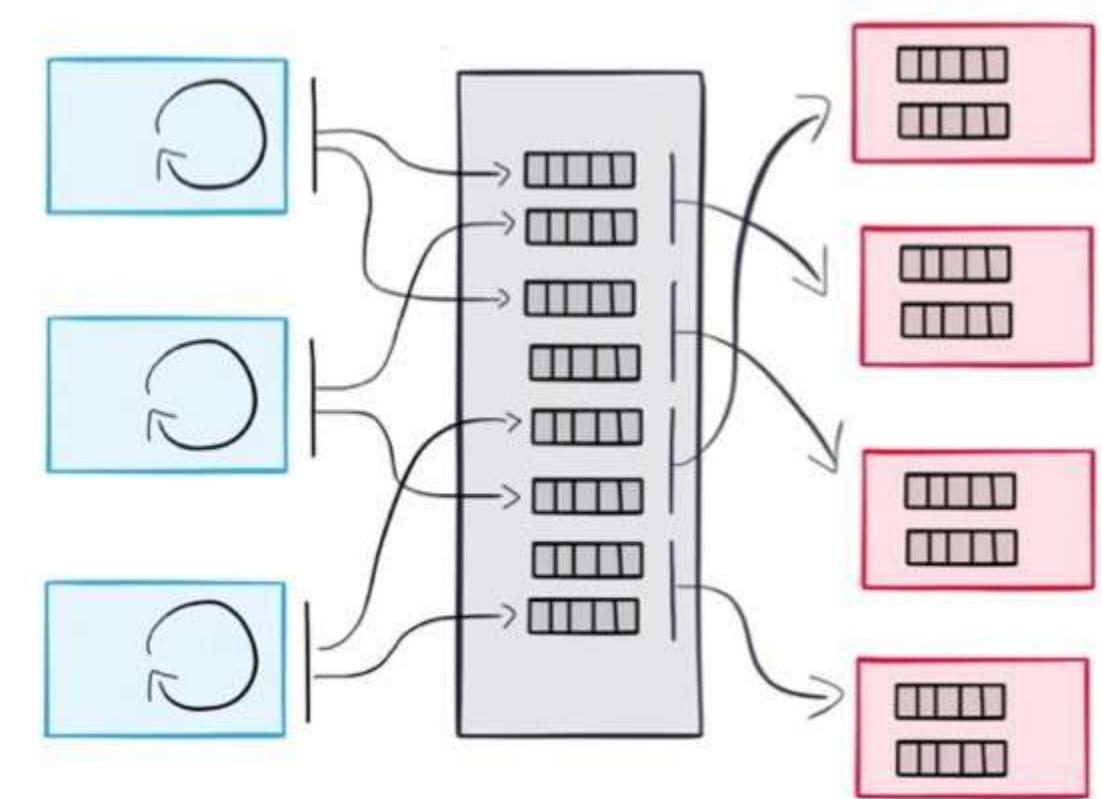


# Consumers



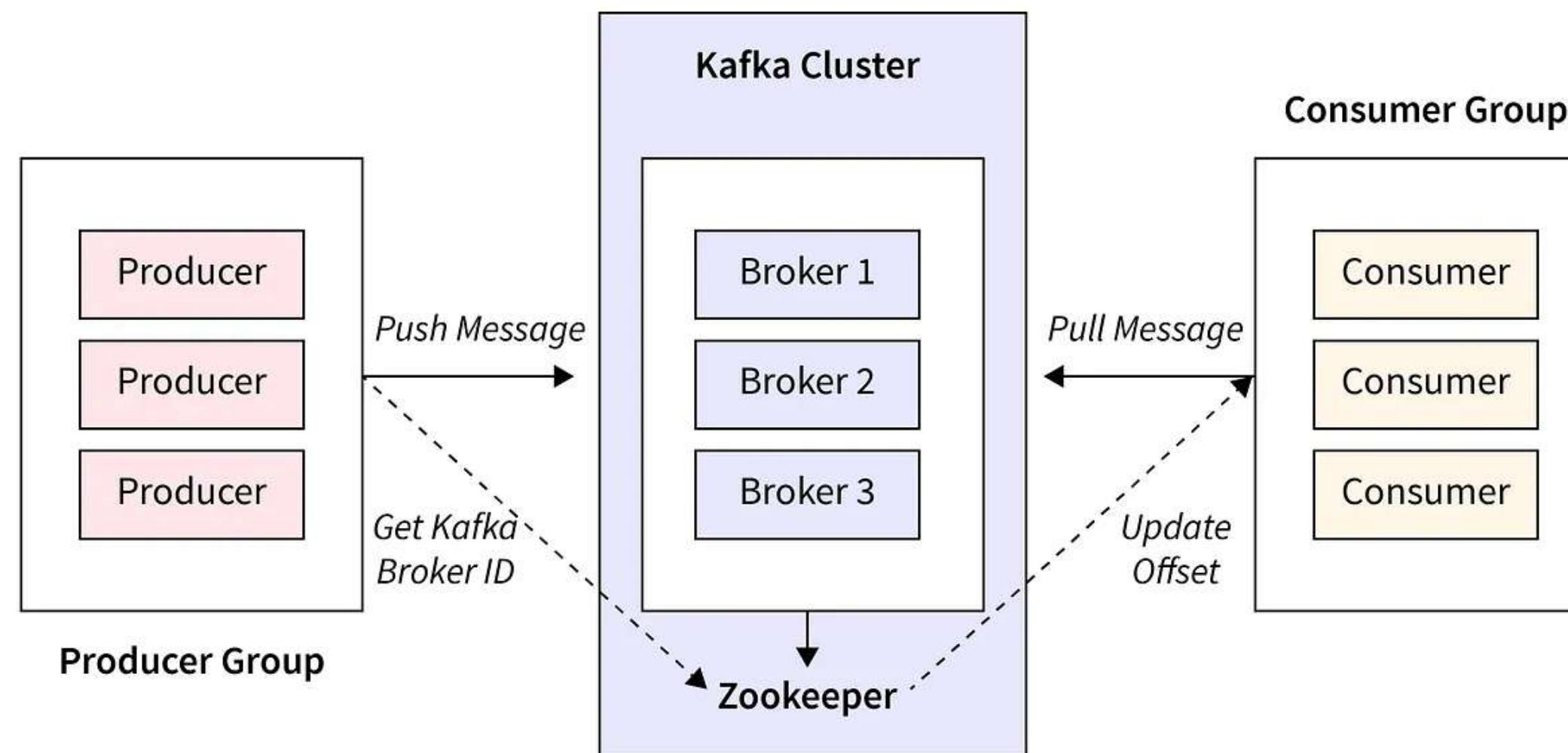
# Producers and Consumers are Decoupled

- Producers and Consumers are decoupled
- Slow Consumers do not affect Producers
- Add Consumers without affecting Producers
- Failure of Consumer does not affect System



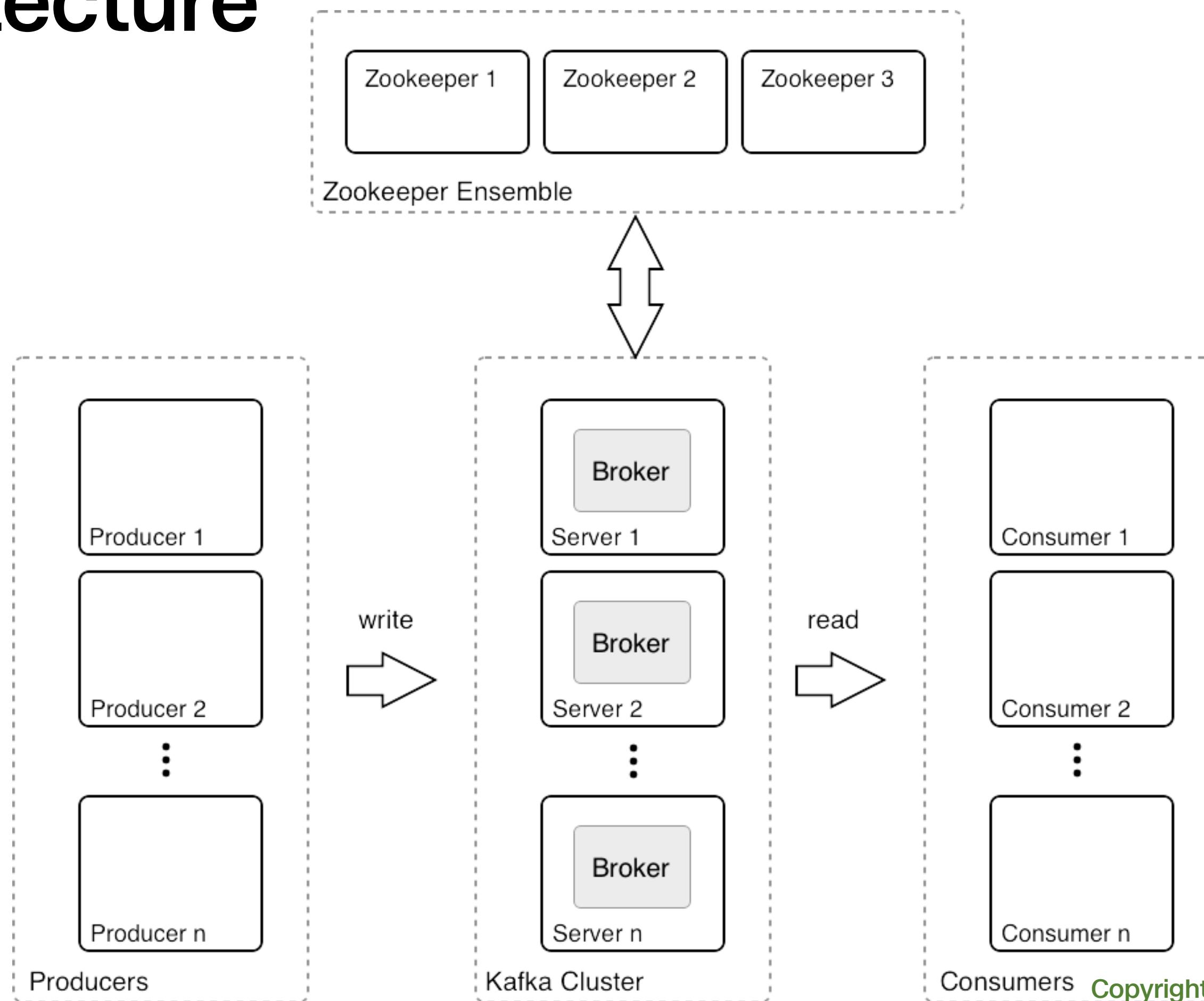
# Apache KAFKA

## Kafka Ecosystem

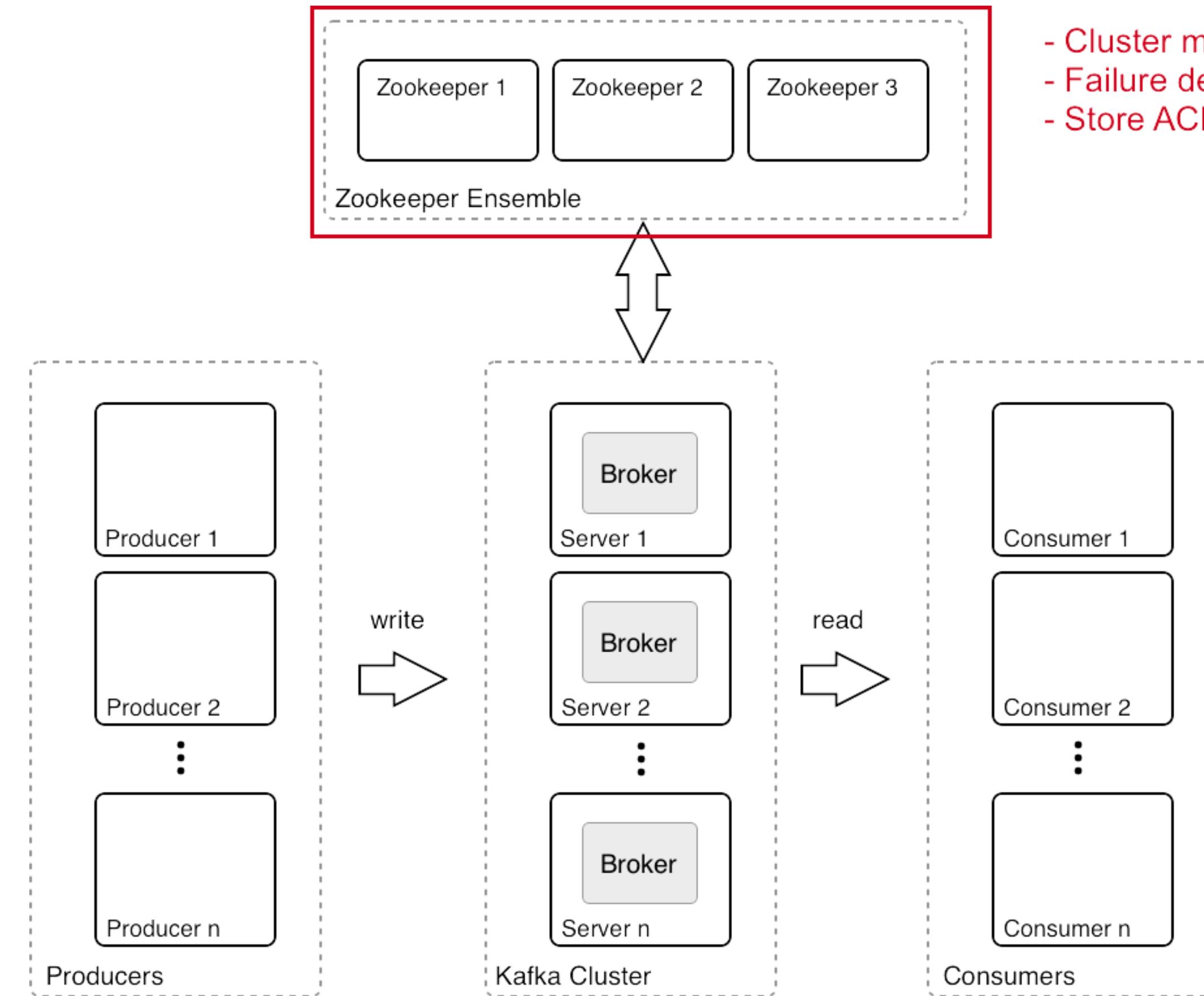


SCALER  
*Topics*

# Architecture



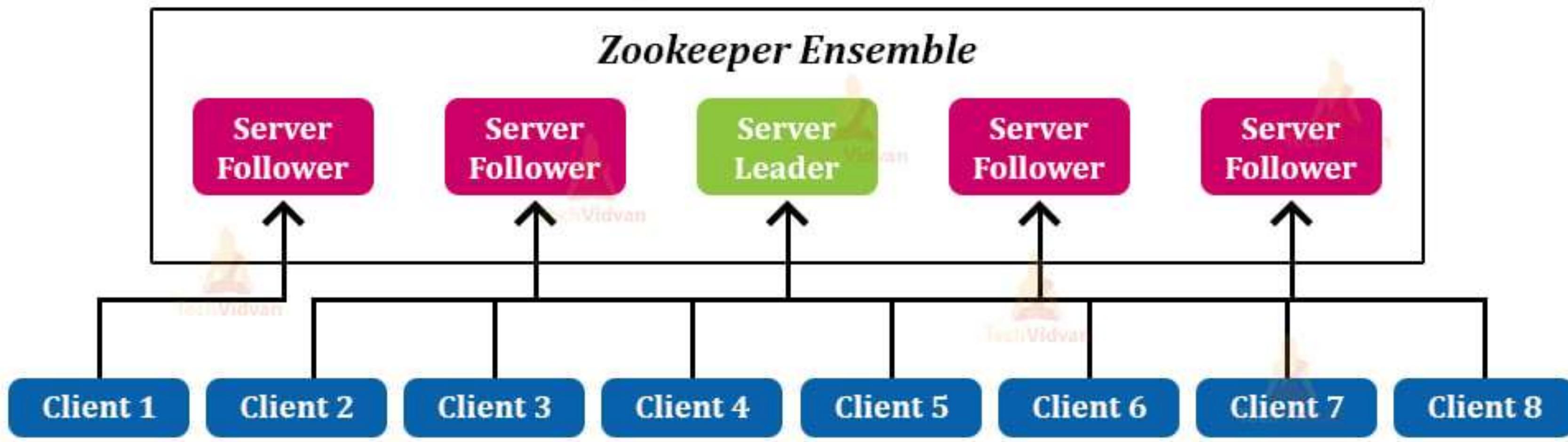
# How Kafka Uses ZooKeeper



- Cluster management
- Failure detection & recovery
- Store ACLs & secrets

# Apache Kafka

## Zookeeper Architecture

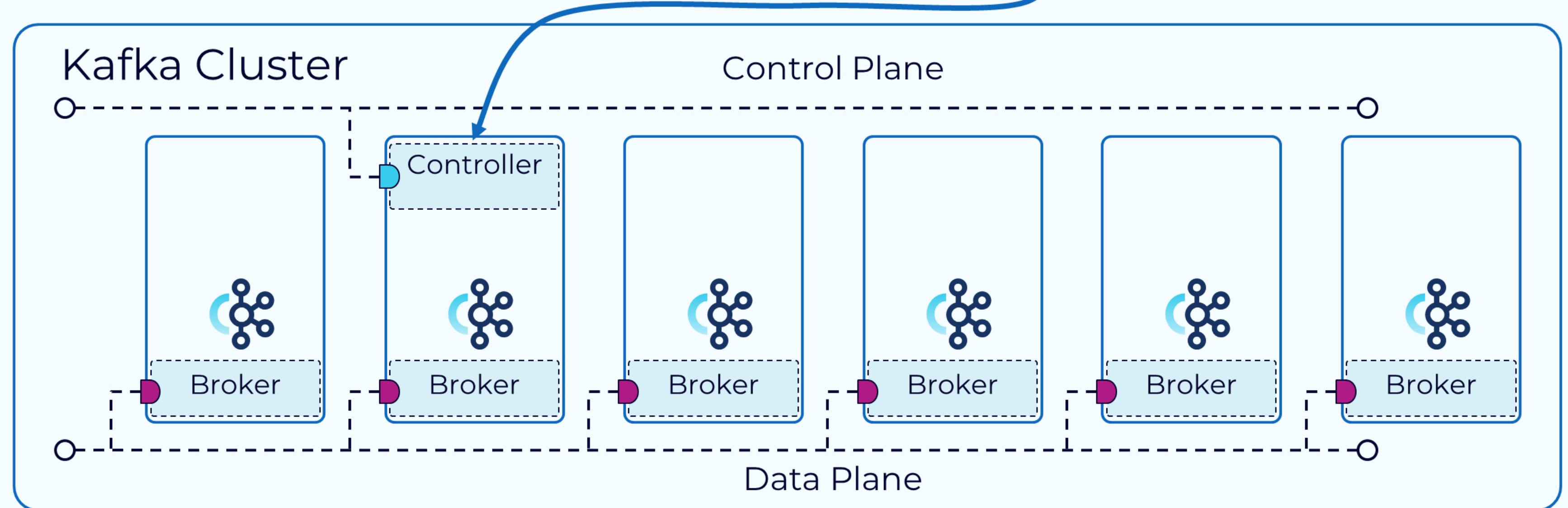
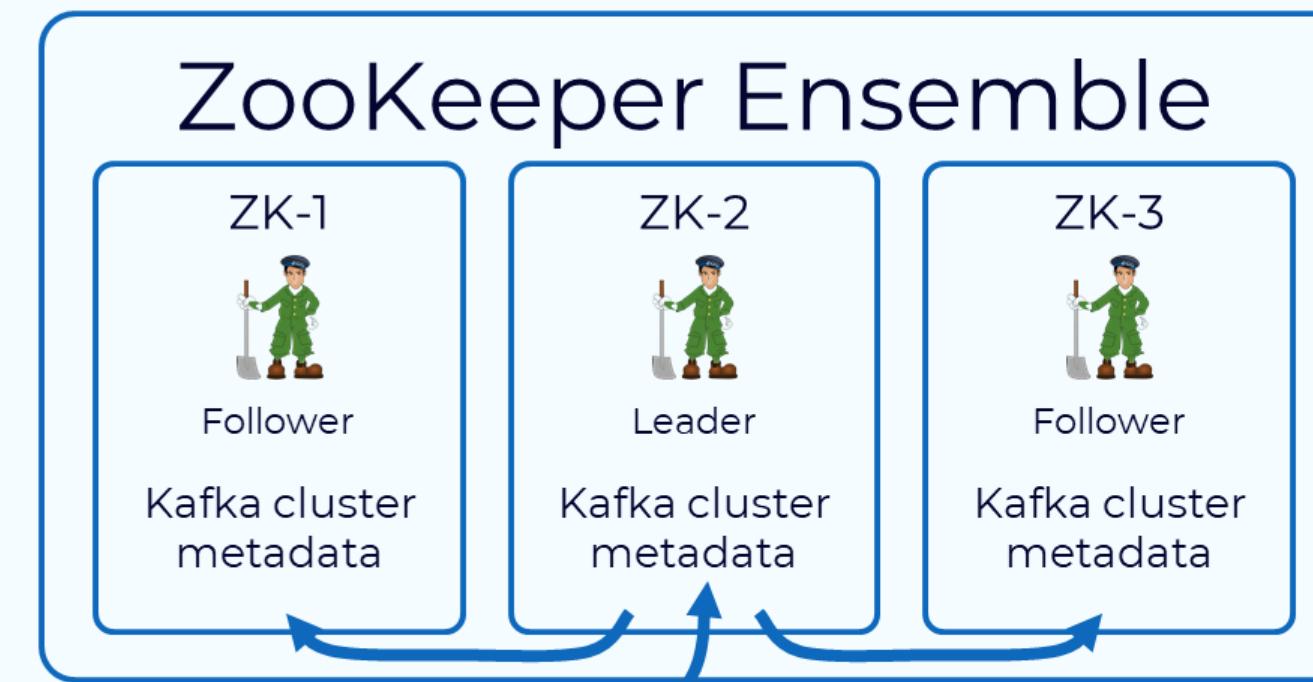


# Zookeeper Basics

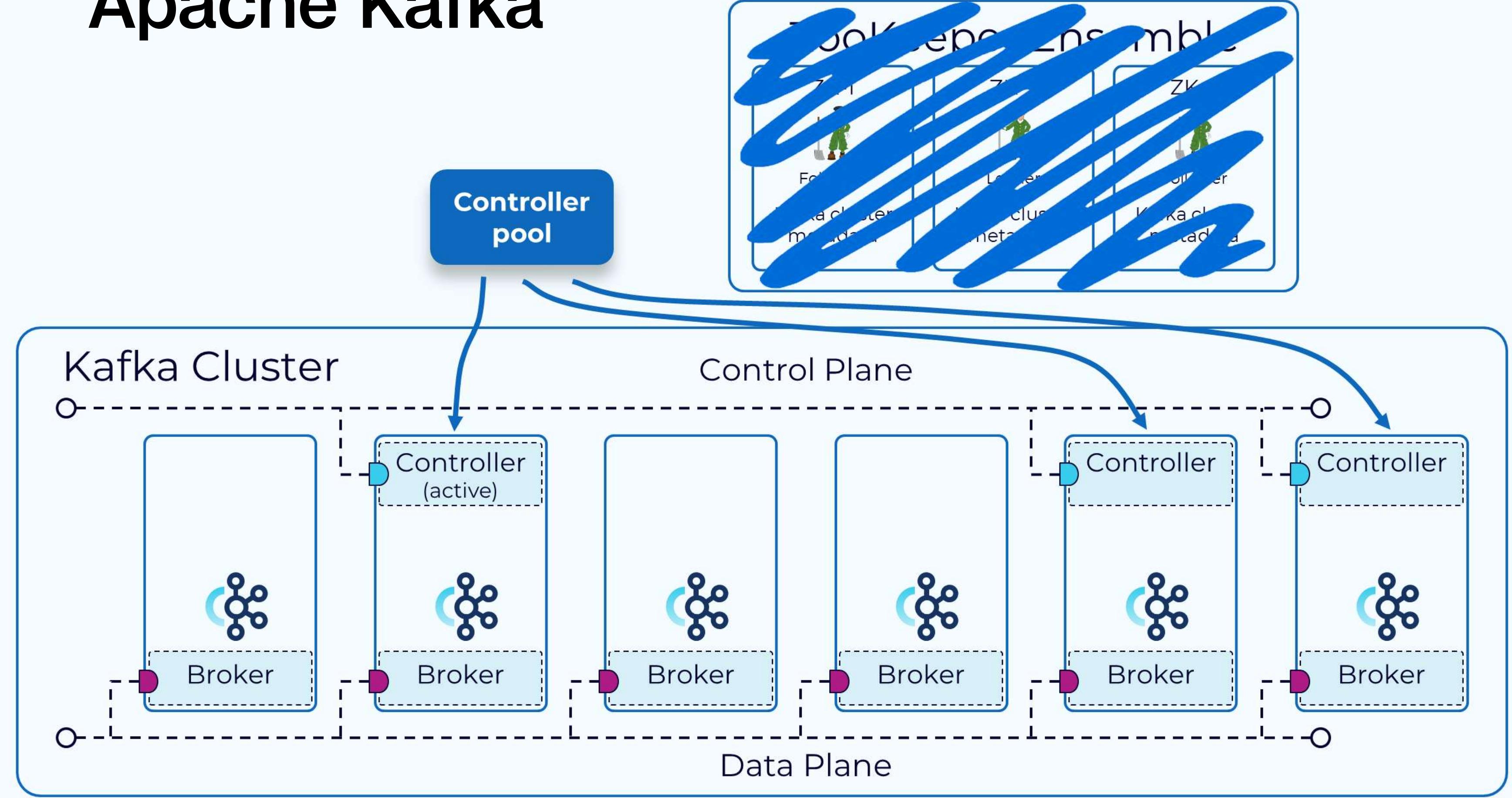
- **Open Source Apache Project**
- **Distributed Key Value Store**
- **Maintains configuration information**
- **Stores ACLs and Secrets**
- **Enables highly reliable distributed coordination**
- **Provides distributed synchronization**
- **Three or five servers form an ensemble**



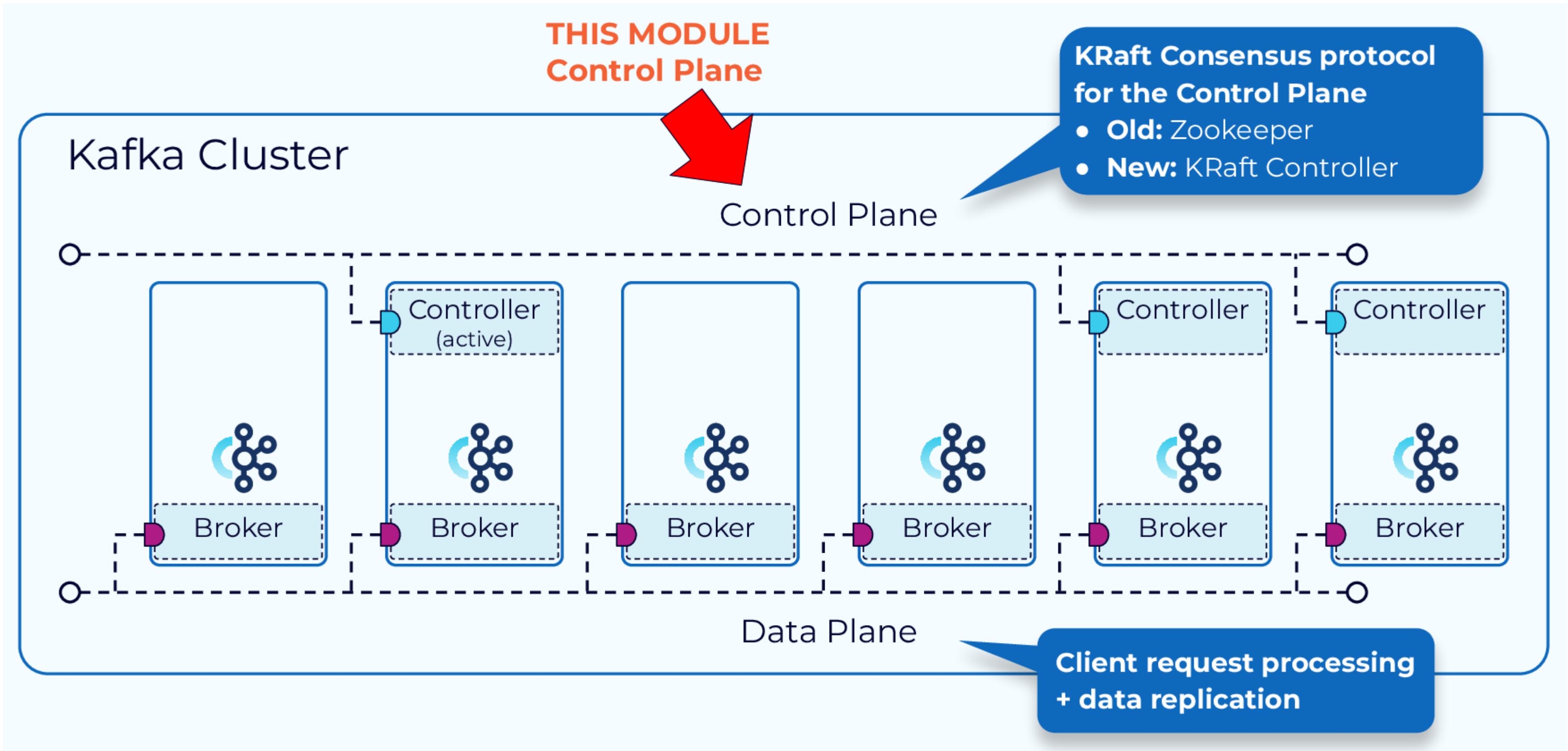
# Apache Kafka



# Apache Kafka

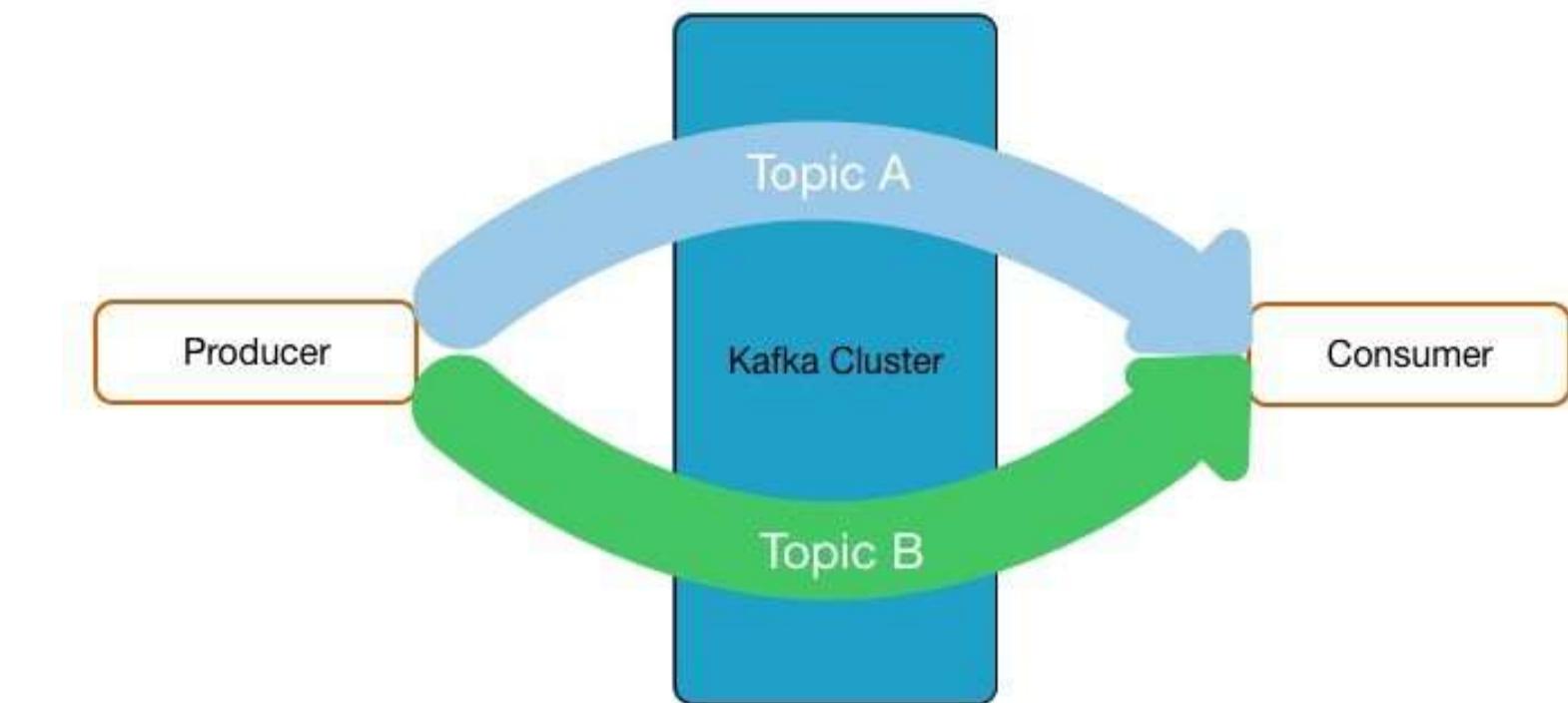


# Apache Kafka

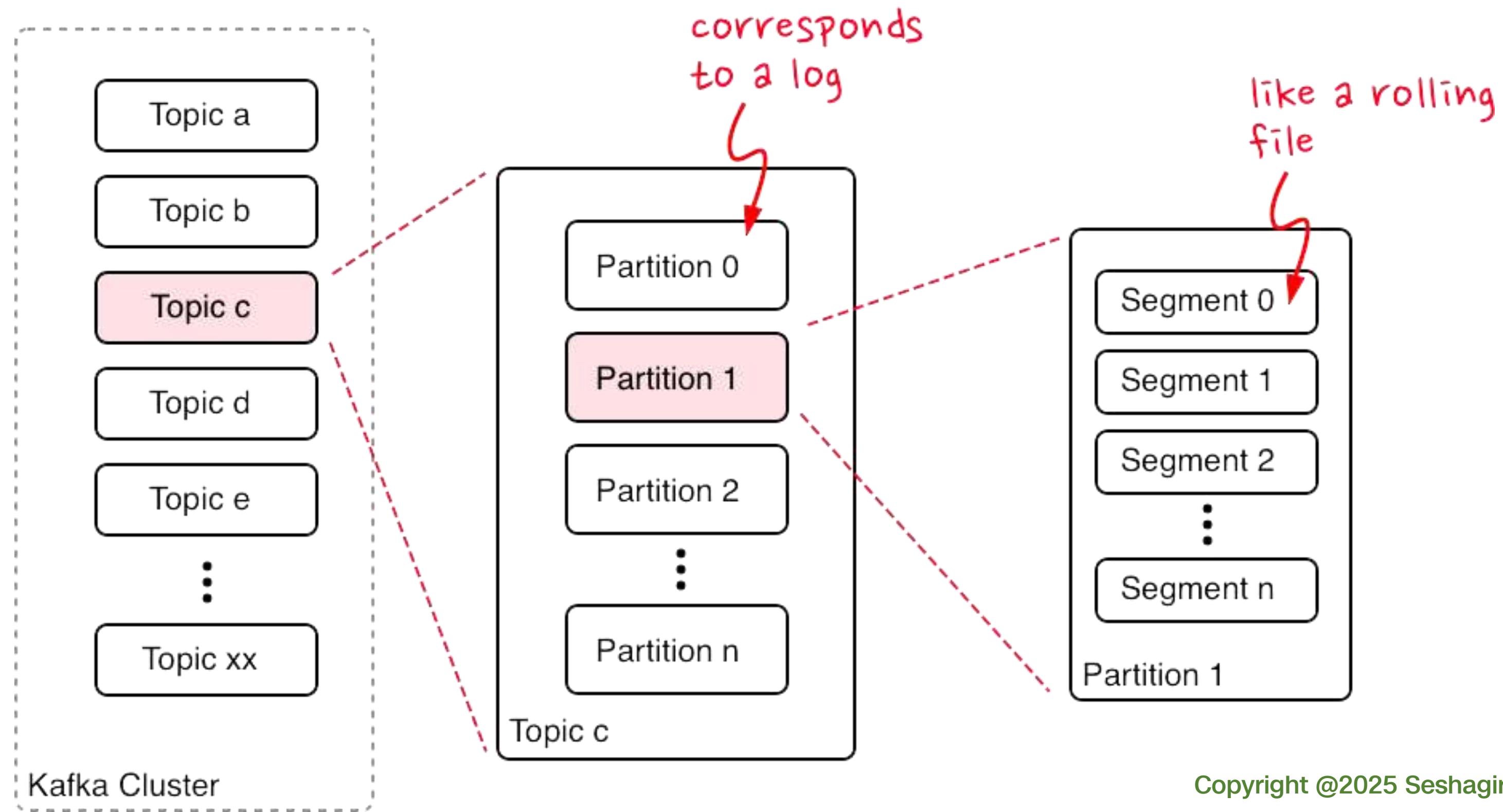


# Topics

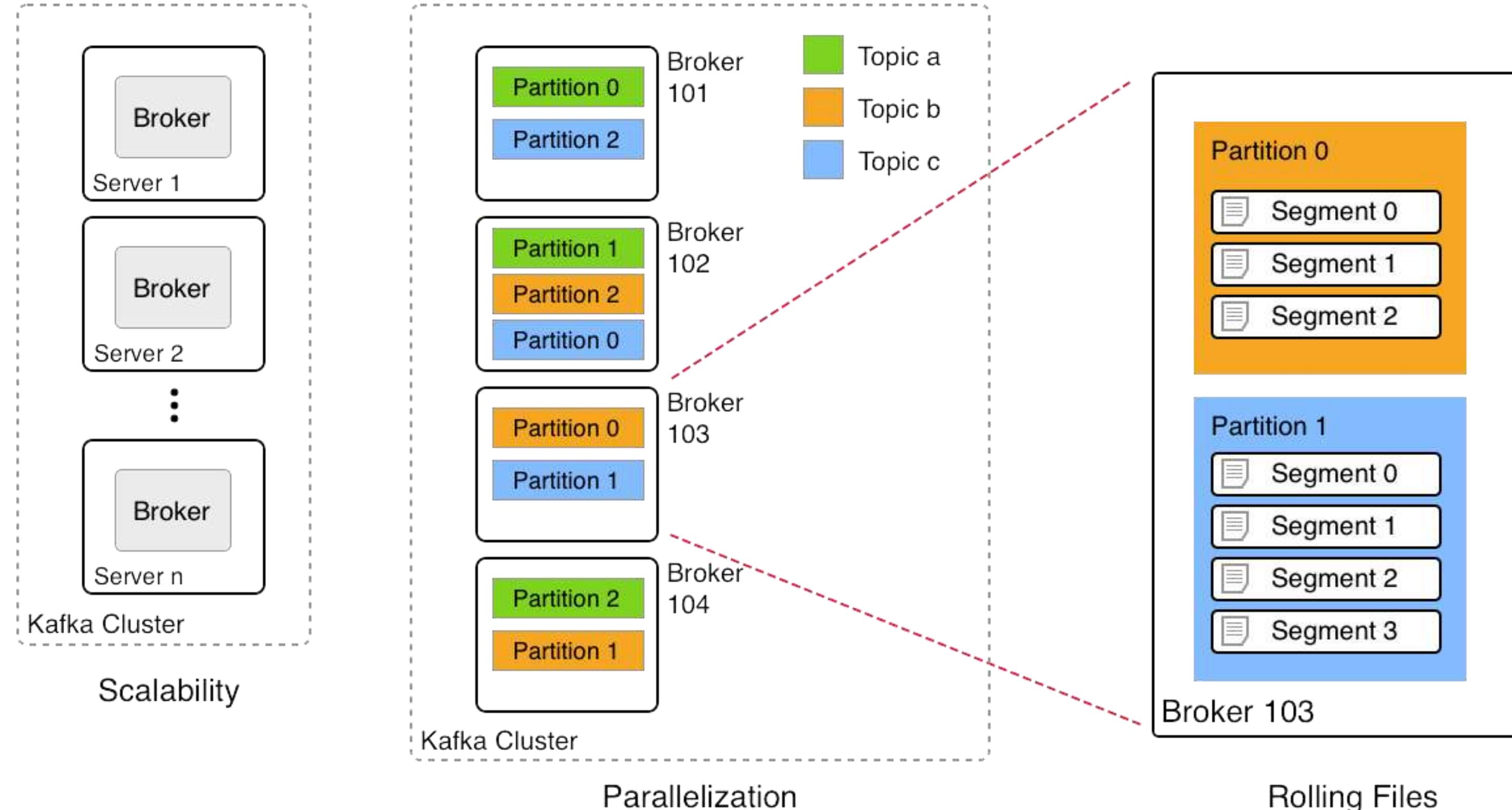
- **Topics:** Streams of “related” Messages in Kafka
  - Is a Logical Representation
  - Categorizes Messages into Groups
- Developers define Topics
- Producer ↔ Topic: N to N Relation
- Unlimited Number of Topics



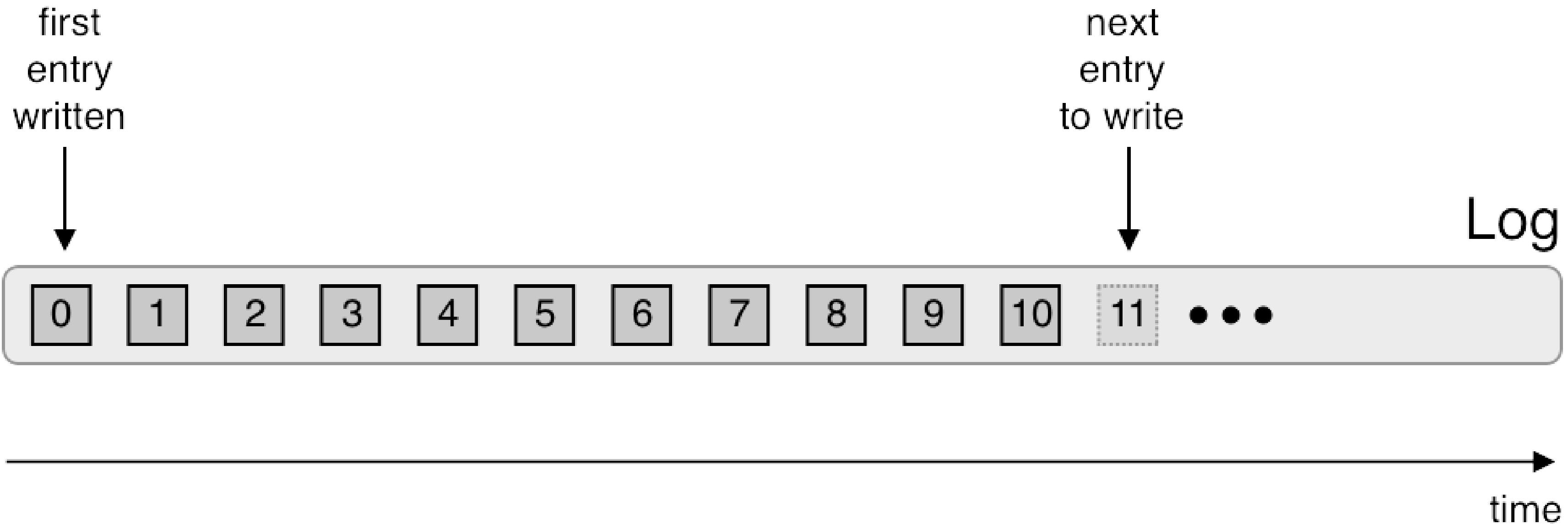
# Topics, Partitions and Segments



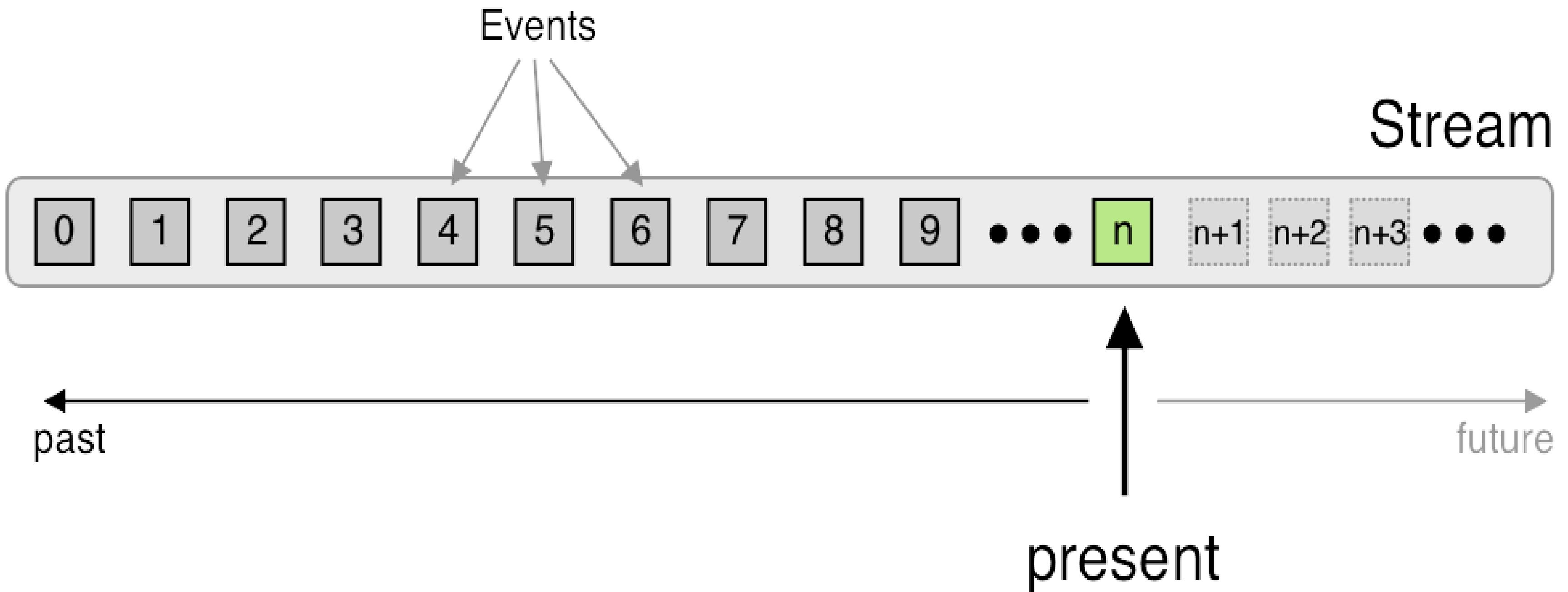
# Topics, Partitions and Segments



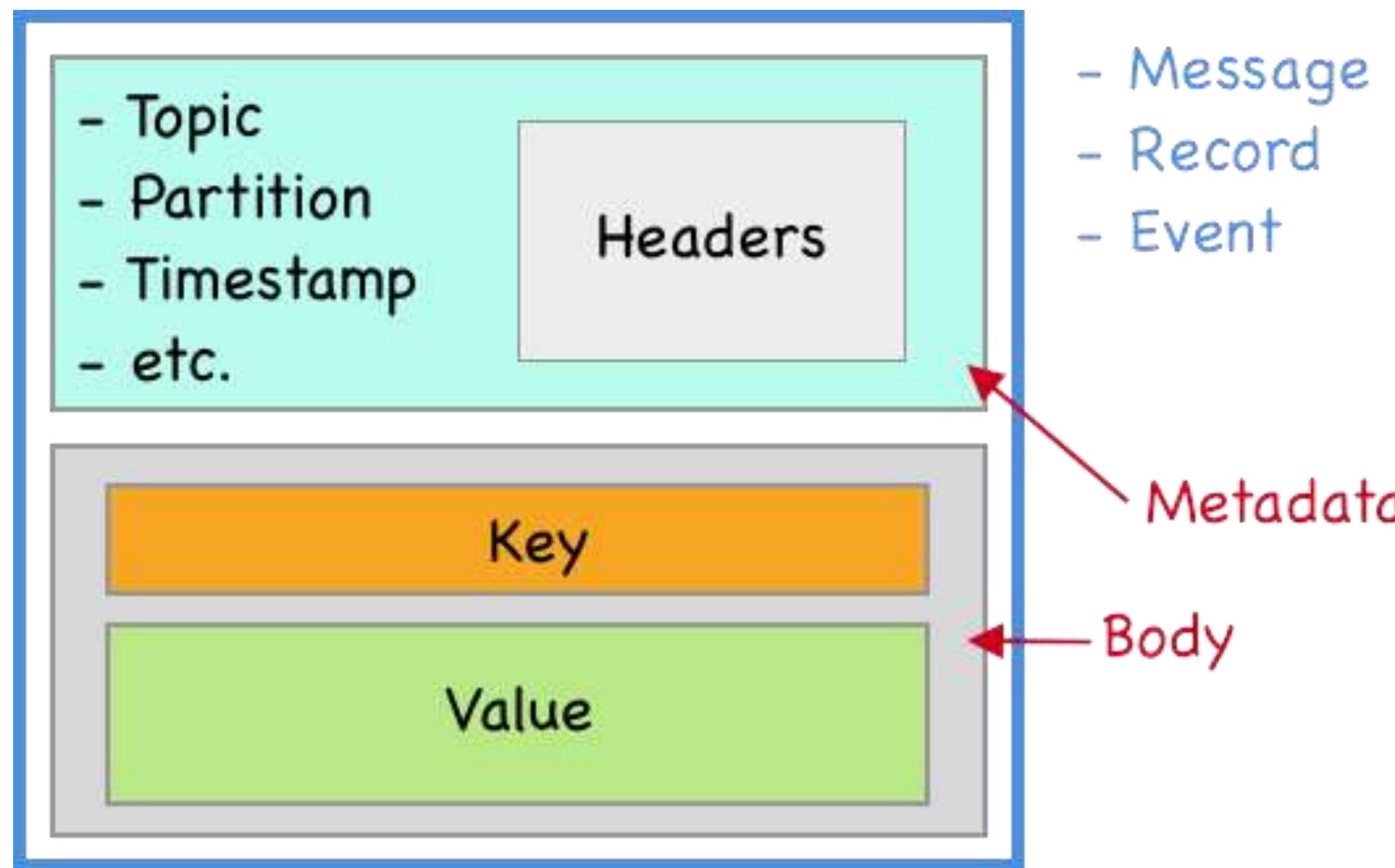
# Kafka Logs



# Streams

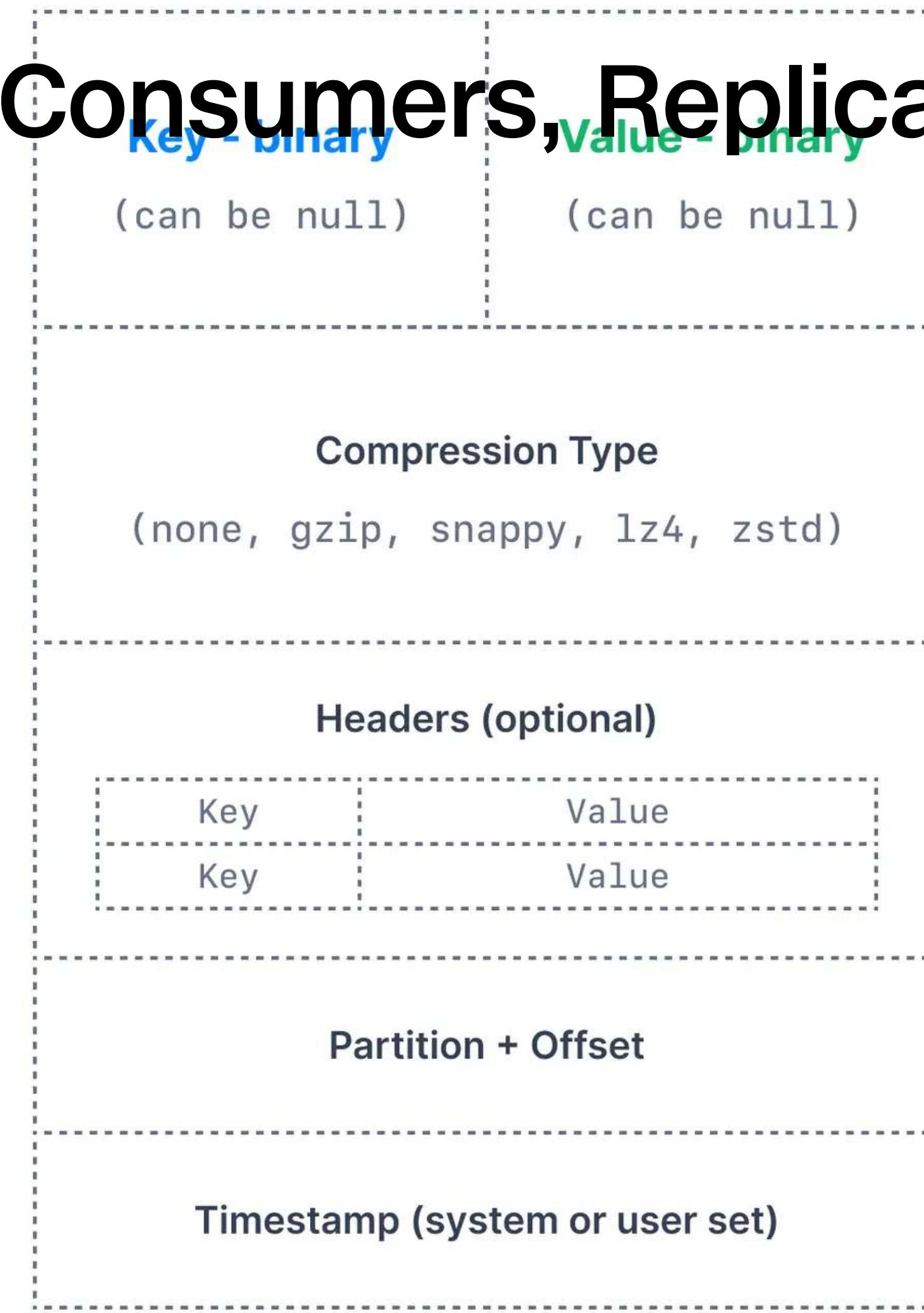


# Data Elements



# Producers, Consumers, Replicas

Kafka  
Message  
Created by  
the producer

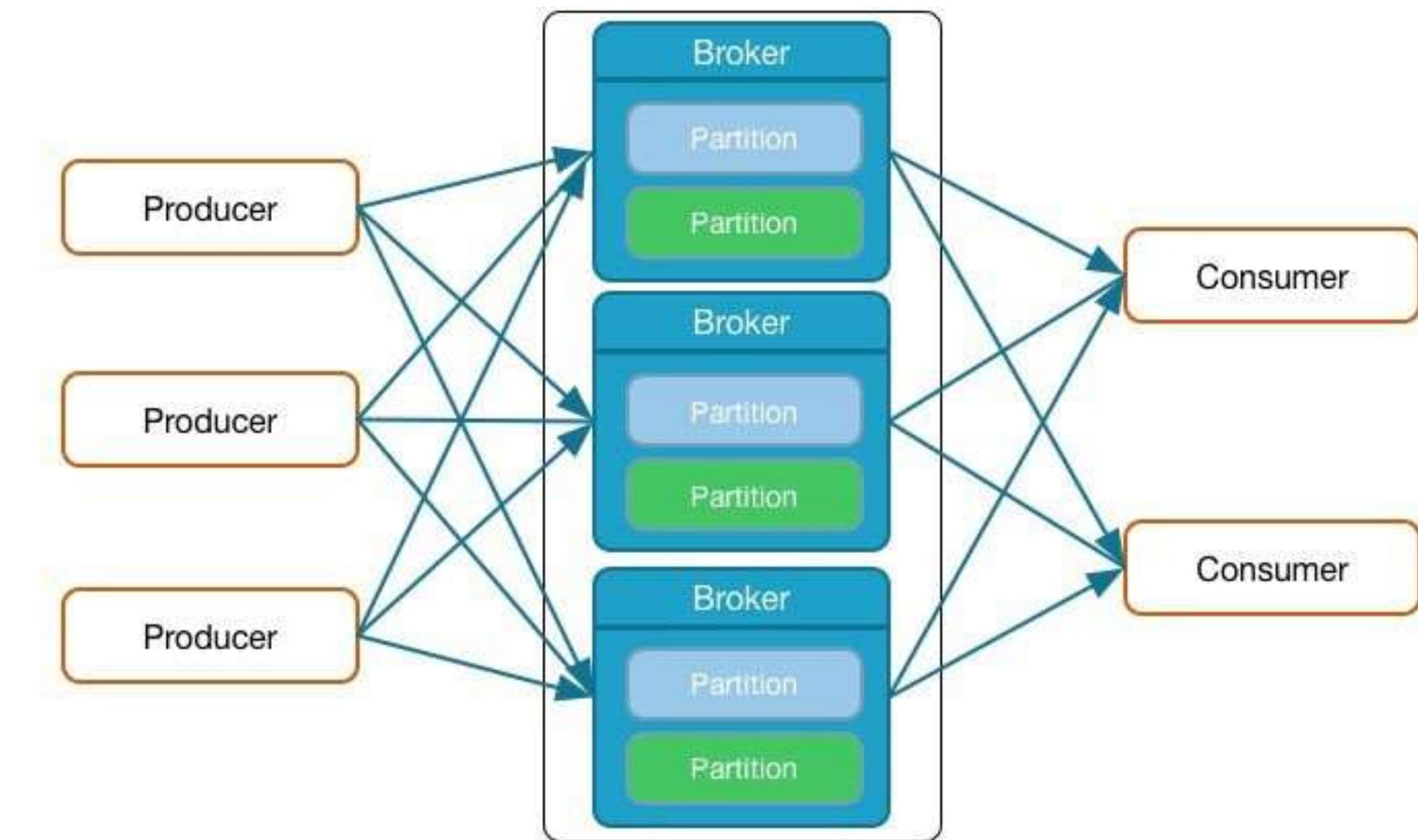


# Brokers And Partitions

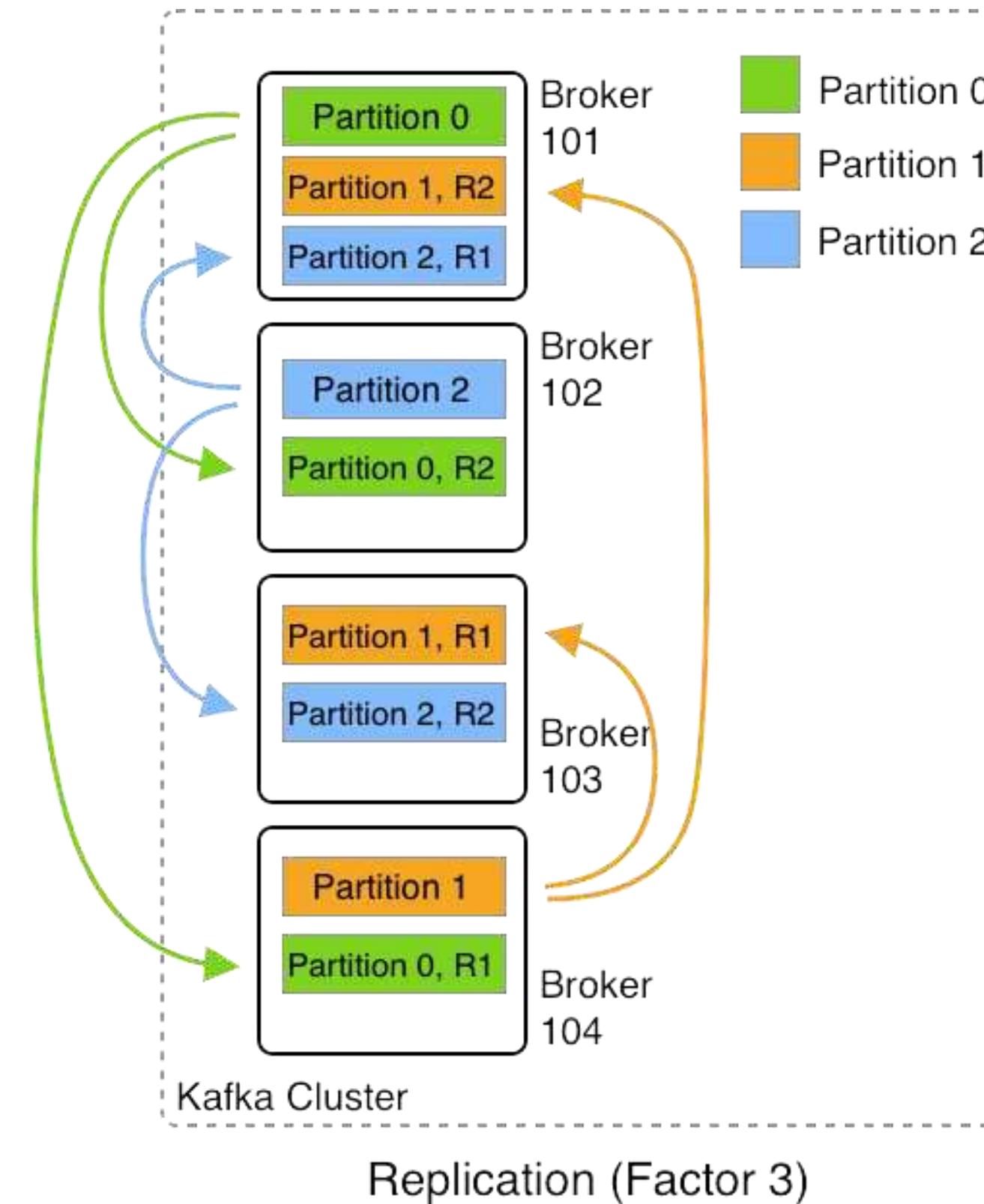
- Messages of Topic spread across Partitions
- Partitions spread across Brokers
- Each Broker handles many Partitions
- Each Partition stored on Broker's disk
- Partition: 1..n log files
- Each message in Log identified by *Offset*
- Configurable Retention Policy

# Broker Basics

- Producer sends Messages to Brokers
- Brokers receive and store Messages
- A Kafka Cluster can have many Brokers
- Each Broker manages multiple Partitions



# Broker Replication



# Producer Basics

- Producers write Data as Messages
- Can be written in any language
  - Native: Java, C/C++, Python, Go,, .NET, JMS
  - More Languages by Community
  - REST Server for any unsupported Language
- Command Line Producer Tool

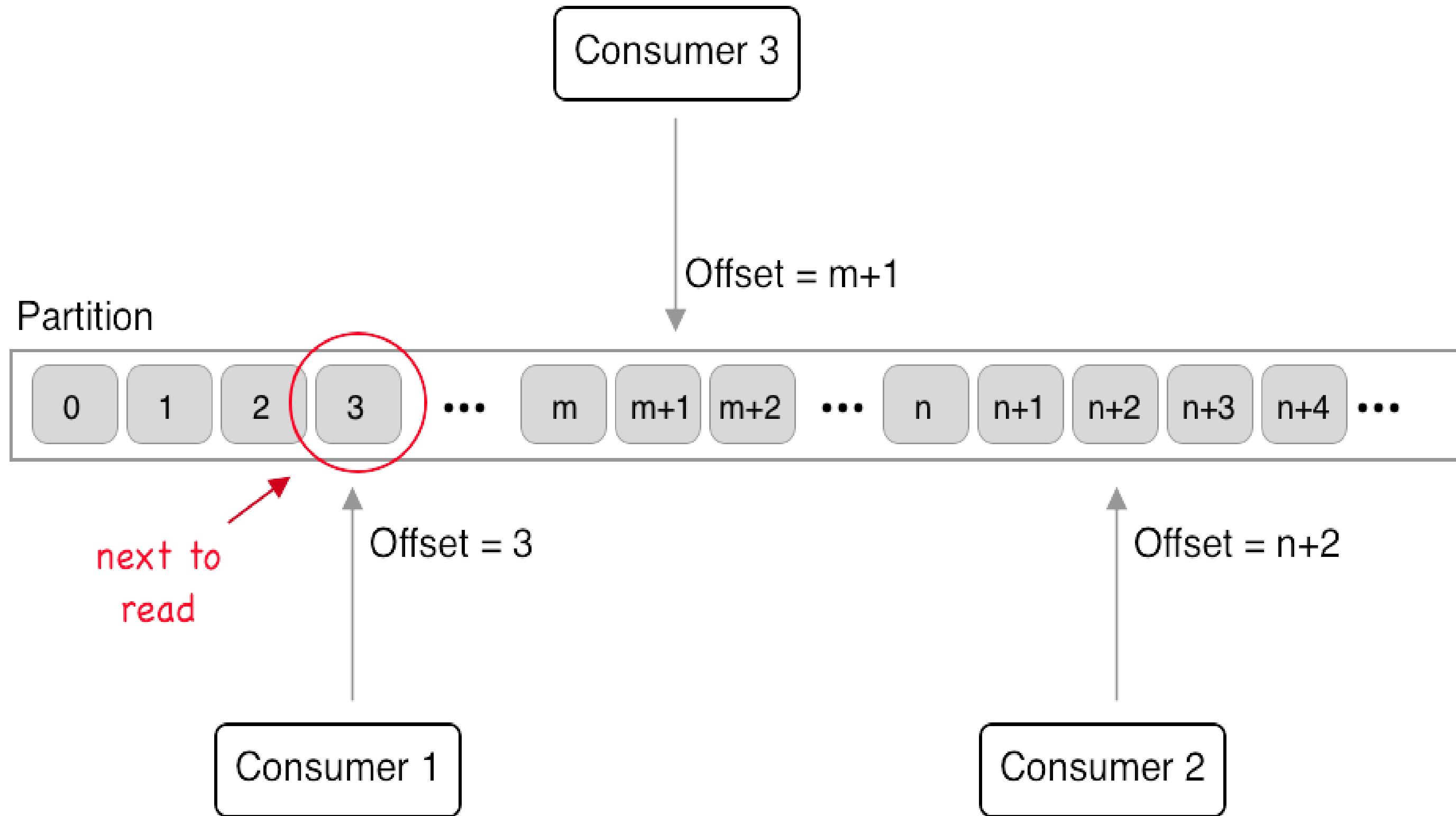
# Load Balancing and Semantic Partitioning

- Producers use a Partitioning Strategy to assign each message to a Partition
- Two Purposes:
  - Load Balancing
  - Semantic Partitioning
- Partitioning Strategy specified by Producer
  - Default Strategy: `hash(key) % number_of_partitions`
  - No Key → Round-Robin
- Custom Partitioner possible

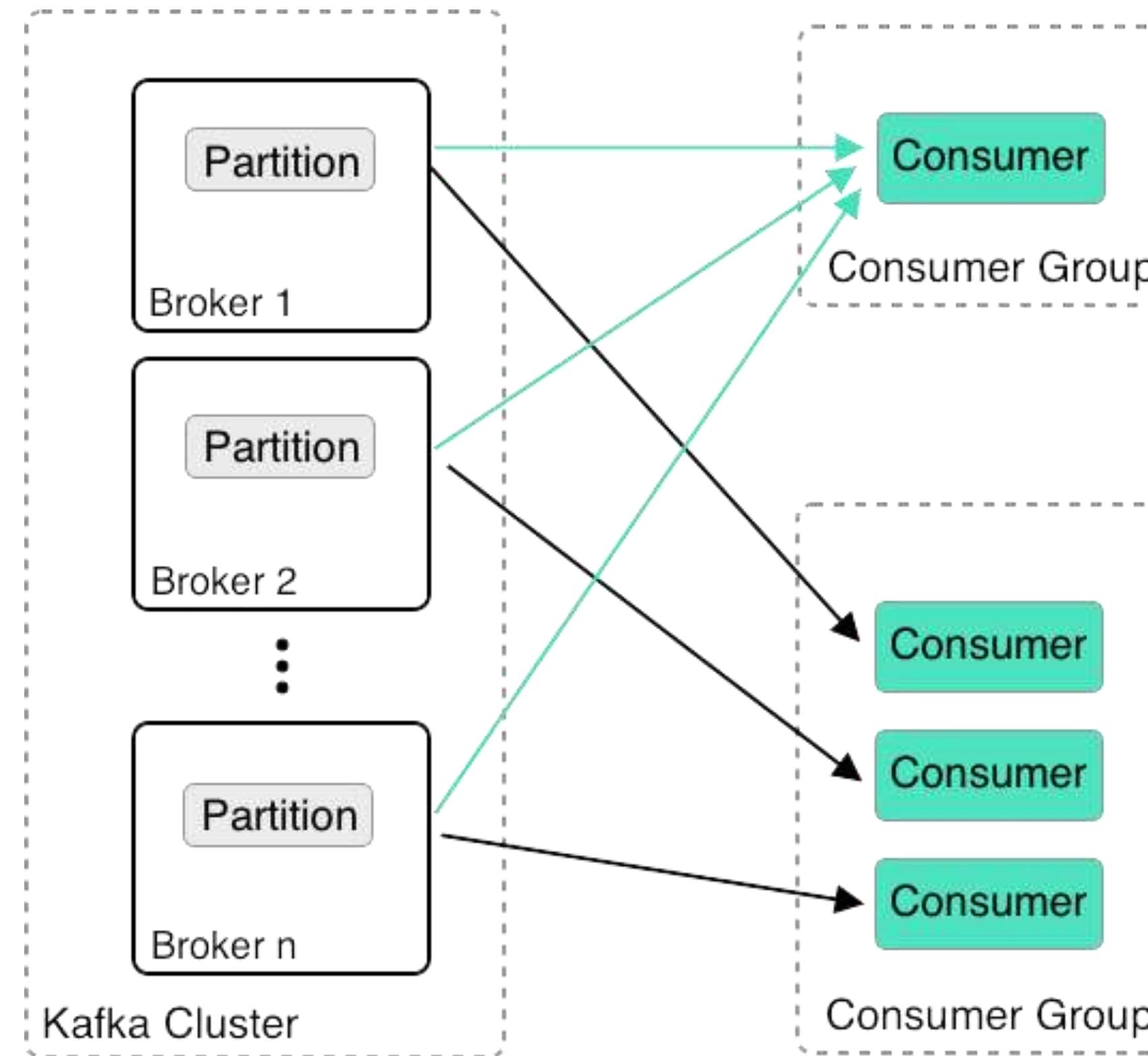
# Consumer Basics

- Consumers **pull** messages from 1..n topics
- New inflowing messages are automatically retrieved
- Consumer offset
  - Keeps track of the last message read
  - Is stored in special topic
- CLI tools exist to read from cluster

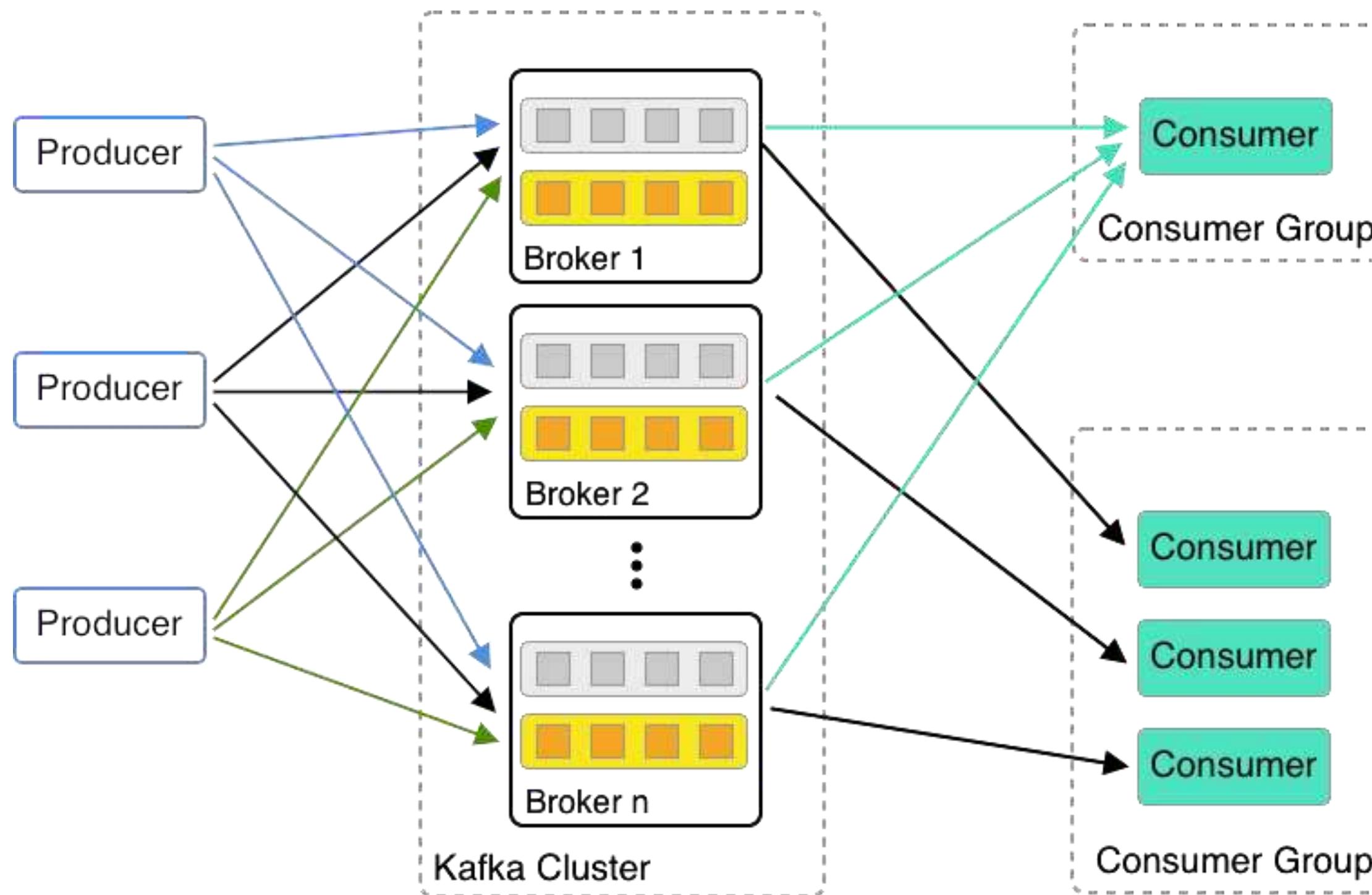
# Consumer Offsets



# Distributed Consumption



# Scalable Data Pipelines



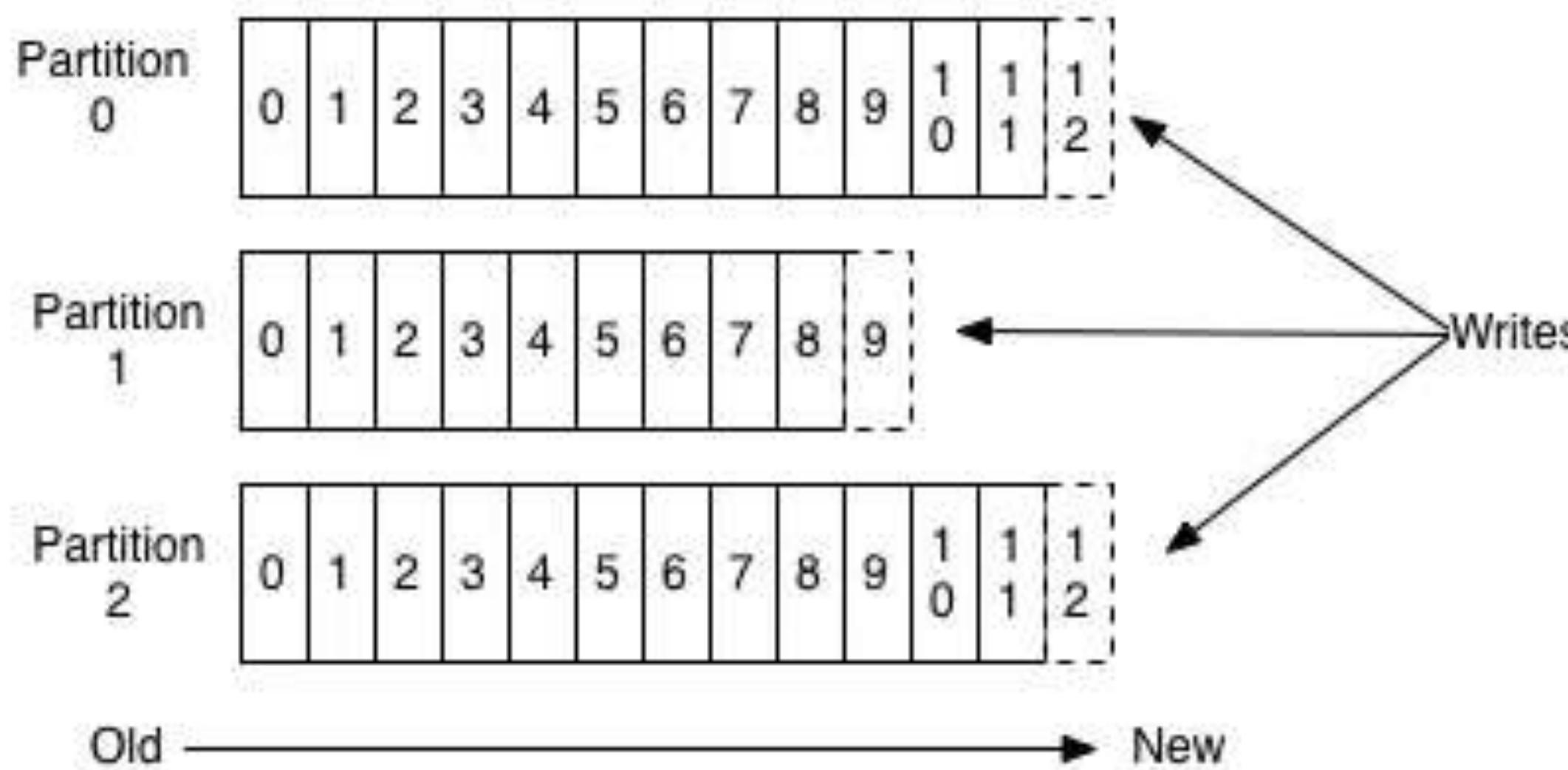
# Q & A



- **Questions:**
- Why do we need an odd number of ZooKeeper nodes?
- How many Kafka brokers can a cluster maximally have?
- How many Kafka brokers do you minimally need for high availability?
- What is the criteria that two or more consumers form a consumer group?

# KAFKA TOPICS AND PARTITIONS

## Anatomy of a Topic



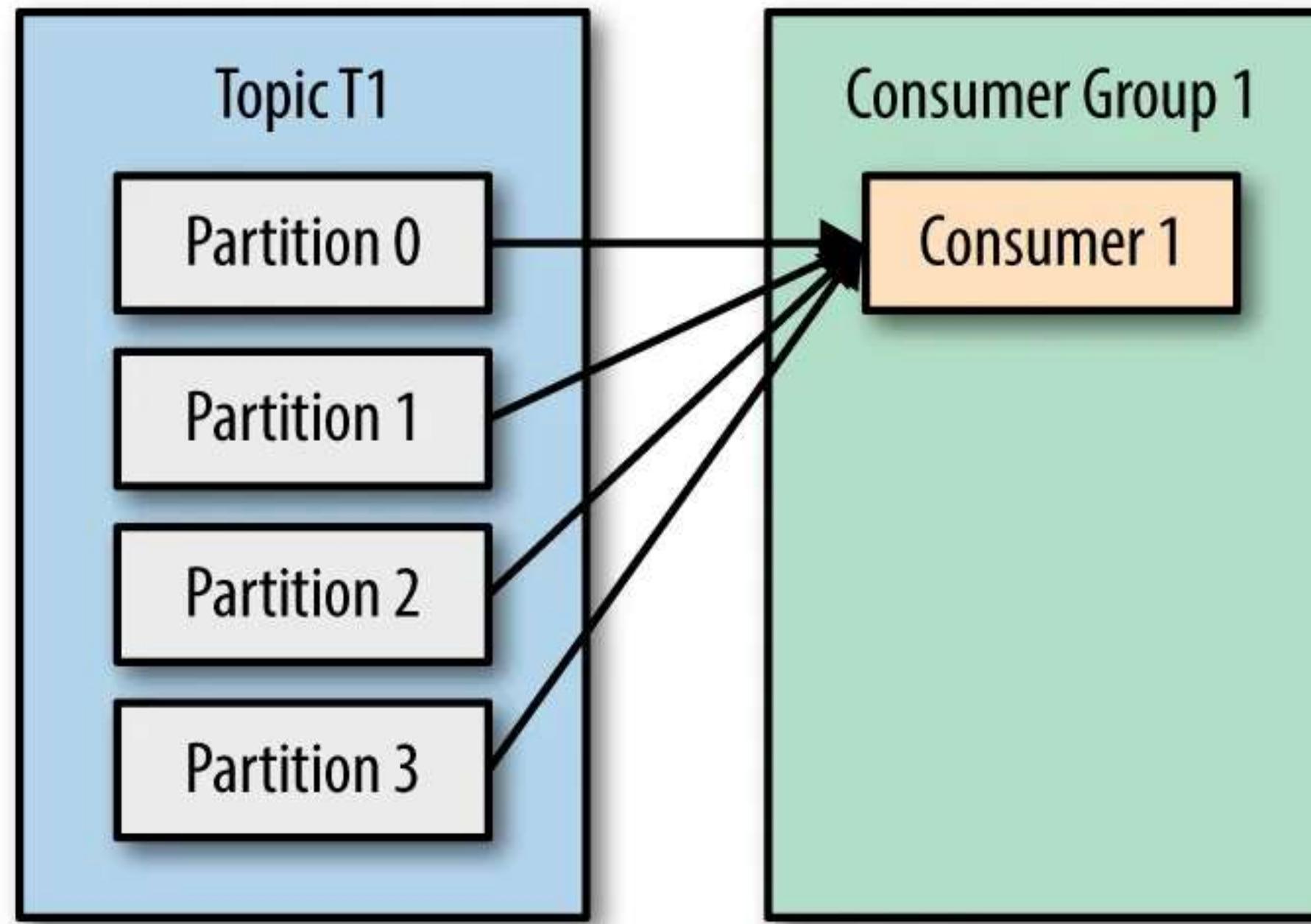
- Every Topic has one or more partitions.
- Default is 1
- No 2 Partitions will have same data
- Every Partition has its own offset.

# KAFKA TOPICS AND PARTITIONS

- A bunch of consumers can form a group in order to cooperate and consume messages from a set of topics.
- This grouping of consumers is called a **Consumer Group**.
- If two consumers have subscribed to the same topic and are present in the same consumer group, then these two consumers would be assigned a different set of partitions and none of these two consumers would receive the same messages.

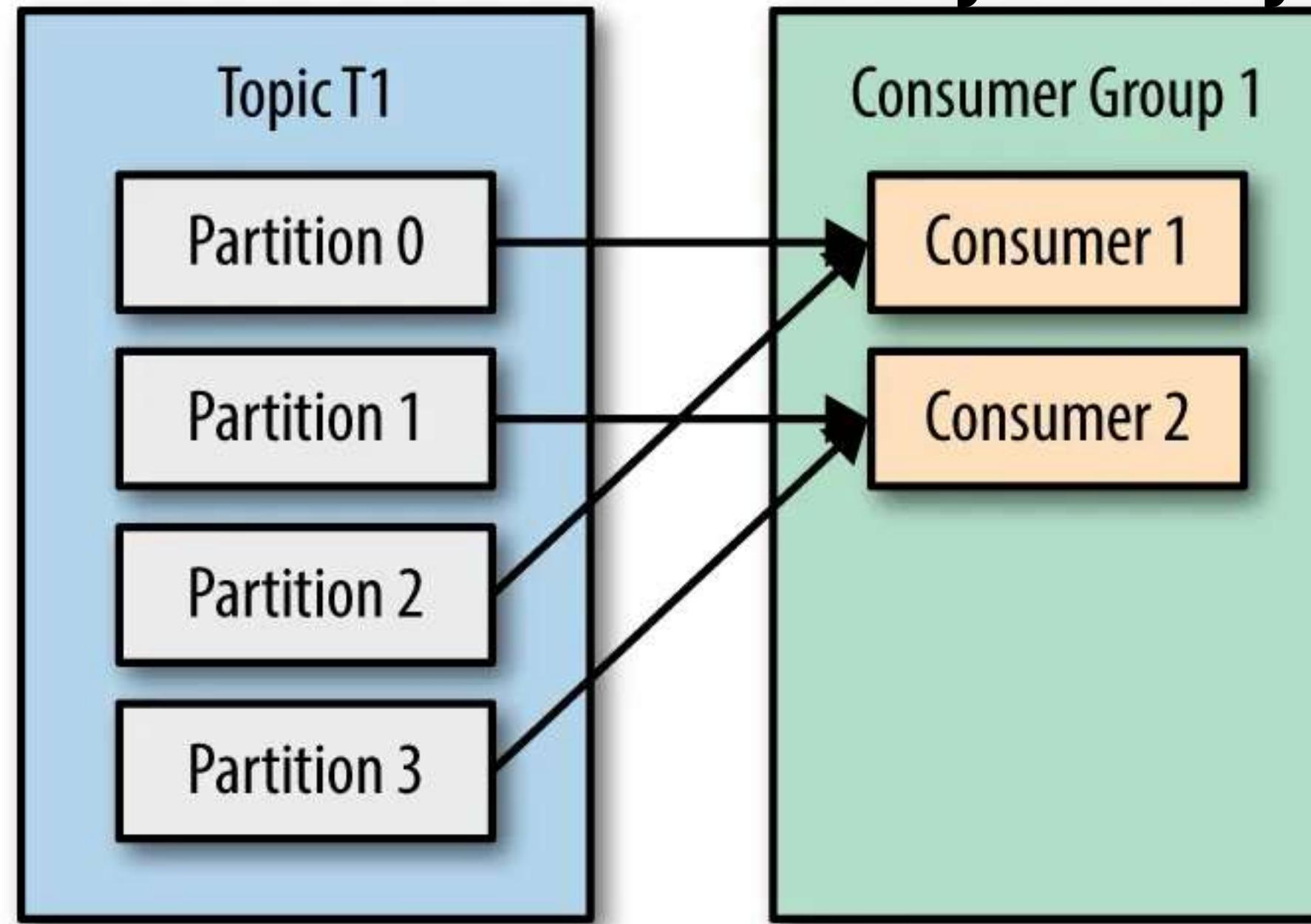
# KAFKA TOPICS AND PARTITIONS

- Scenario #1 = 4 Partitions, 1 Consumer Group, 1 Consumer



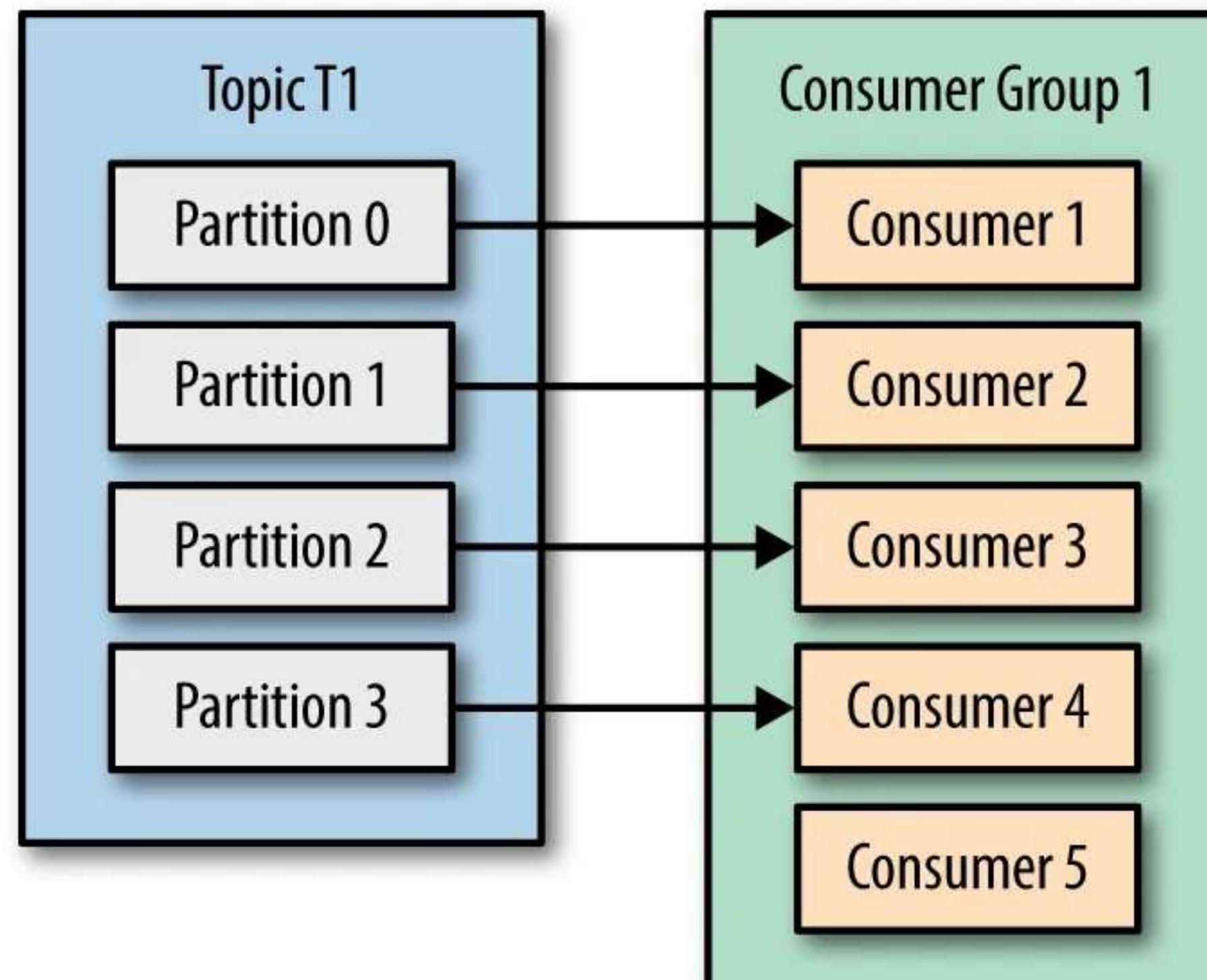
# KAFKA TOPICS AND PARTITIONS

- Scenario #2: 4 Partitions, 1 Consumer Group, 2 Consumers
- EACH Partition is consumed by exactly 1 Consumer



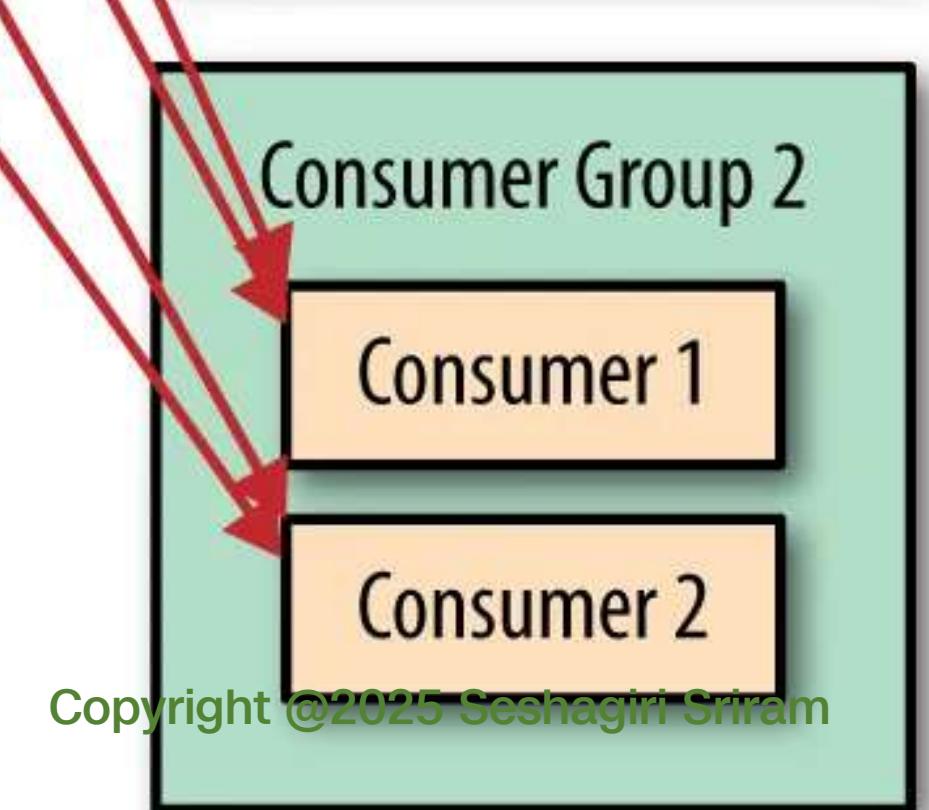
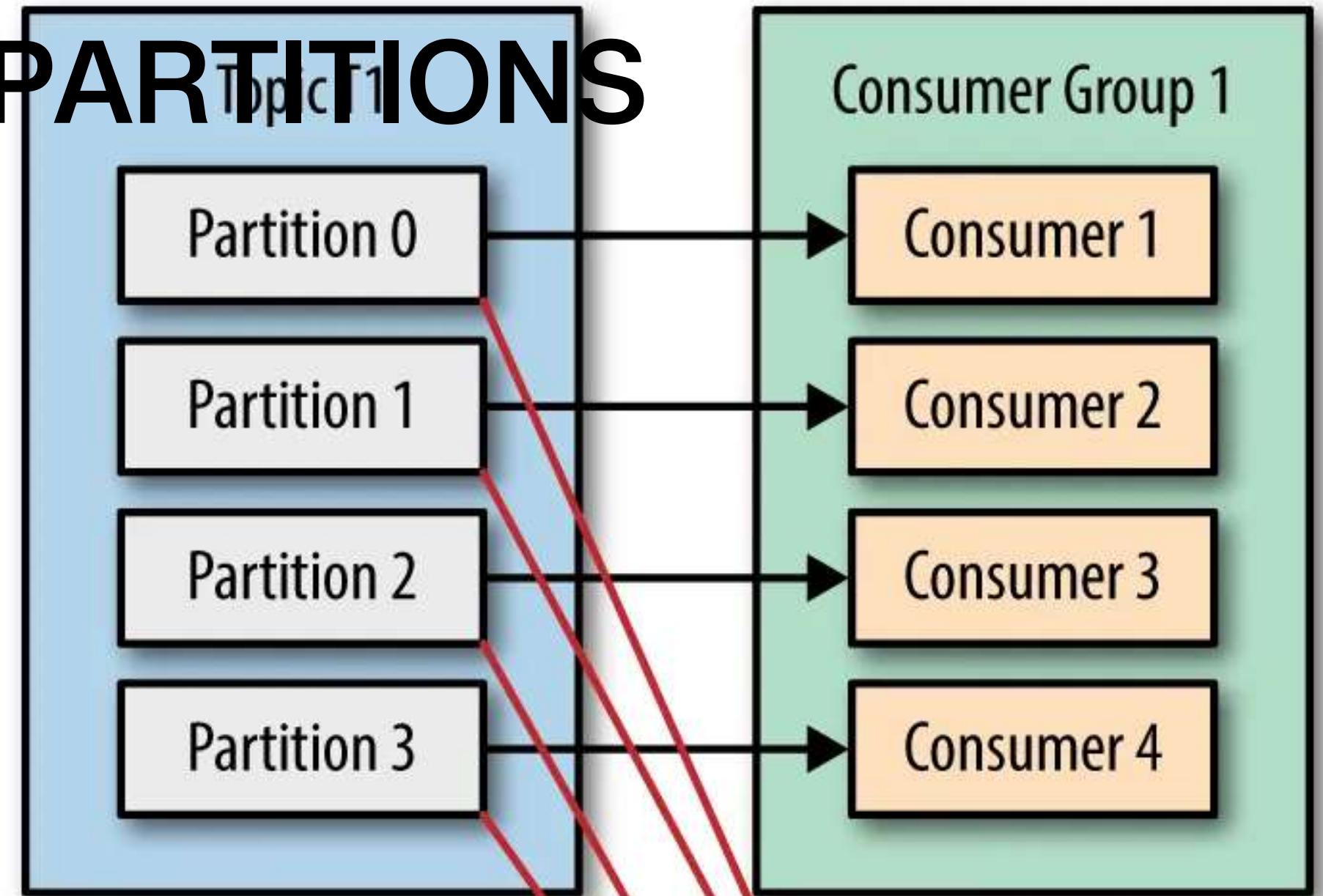
# KAFKA TOPICS AND PARTITIONS

- Scenario #3: More Consumers than there are partitions.



# KAFKA TOPICS AND PARTITIONS

- SCENARIO #4:  
Multiple Consumers  
need to Read from  
Same Partition.



# KAFKA TOPICS AND PARTITIONS

- If your goal is to attain a higher throughput or increase the consumption rate for a particular topic, then adding multiple consumers in the same consumer group would be the way to go.
- You can have at max consumers equal to the number of partitions of a topic in a consumer group, adding more consumers than the number of partitions would cause the extra consumers to remain idle, since Kafka maintains that no two partitions can be assigned to the consumer in a consumer group.
- When the the number of consumers consuming from a topic is equal to the number of partitions in the topic, then each consumer would be reading messages from a single topic, and the message consumption would be happening in parallel, thereby increasing the consumption rate.

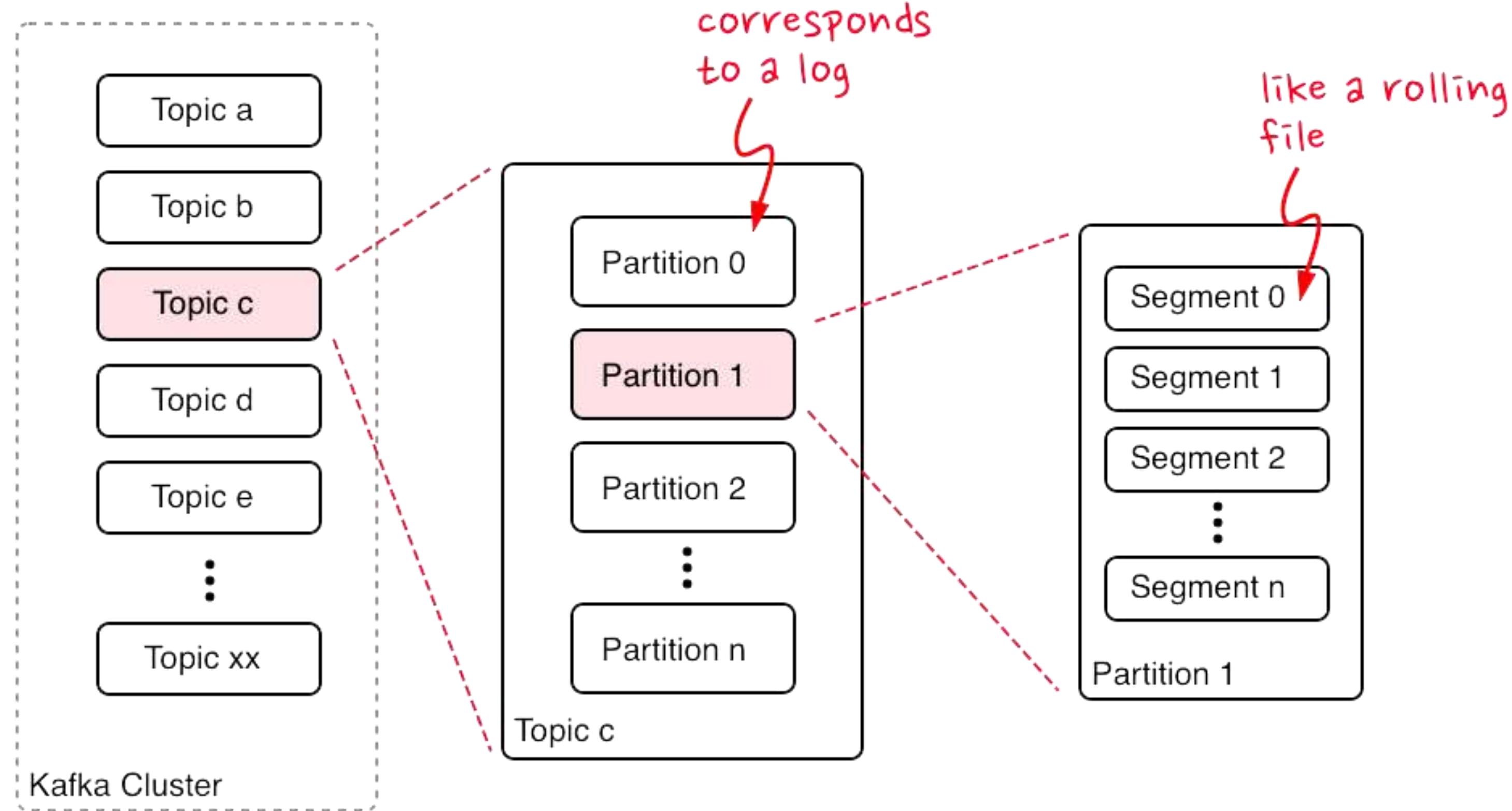
# KAFKA TOPICS AND PARTITIONS

- All Consumers are part of a Consumer Group.
- When Consumers are added or removed/stopped, a process of re-balancing occurs.
- **Rebalancing happens**
  - Consumers are added
  - Consumer go down
  - Consumer is considered Dead by the group Coordinator
  - New Partitions are added.
- Note Partitions cannot be decreased.

# Indexes, Retention, Compaction, Logs

- Kafka stores data in **log files**, which are essentially append-only files that hold messages for each partition.
- A Kafka log for a partition isn't a monolithic file but is broken down into **segments**.
- Each segment is a sequence of messages with a monotonically increasing offset.
- Kafka appends new messages to the last segment of a partition.

# Indexes, Retention, Compaction, Logs



# Indexes, Retention, Compaction, Logs



```
./bin/kafka-run-class.sh kafka.tools.GetOffsetShell --broker-list  
<your_broker_list> --topic <your_topic_name> --time -1
```

*See log.dirs folder – there will be 1 folder / topic partition.*

**AN offset represents the position of a message within a partition.**

# Indexes, Retention, Compaction, Logs

File Explorer View			
Name	Date modified	Type	Size
configured-topic-0	11/15/2021 4:56 PM	File folder	
configured-topic-1	11/15/2021 4:56 PM	File folder	
configured-topic-2	11/15/2021 4:56 PM	File folder	
.lock	11/15/2021 4:48 PM	LOCK File	0 KB
cleaner-offset-checkpoint	11/15/2021 4:48 PM	File	0 KB
log-start-offset-checkpoint	11/15/2021 7:27 PM	File	1 KB
meta.properties			
recovery-point-offset-checkpoint			
replication-offset-checkpoint			

Path: This PC > OS (C) > tmp > kafka-logs > configured-topic-0

Name	Date modified	Type	Size
00000000000000000000.index	11/15/2021 4:56 PM	INDEX File	10,240 KB
00000000000000000000	11/15/2021 4:56 PM	Text Document	0 KB
00000000000000000000.timeindex	11/15/2021 4:56 PM	TIMEINDEX File	10,240 KB
leader-epoch-checkpoint	11/15/2021 4:56 PM	File	0 KB

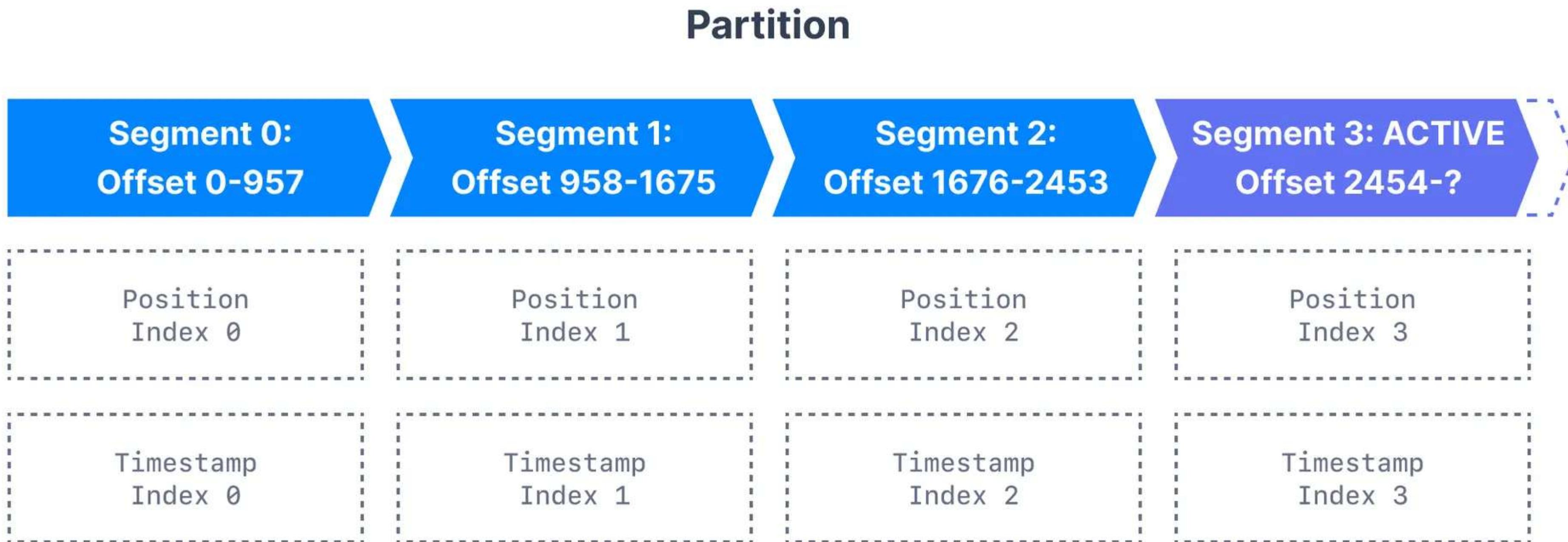
# Indexes, Retention, Compaction, Logs

- Each segment is stored as a pair of files
  - .log
    - Actual Messages
  - .index
    - Manages message offsets to their position
- The naming convention is <offset of 1<sup>st</sup> msg within segment>
- If Base offset of first message is 5000, then files are
  - 00000000000005000.log
  - 00000000000005000.index

# Indexes, Retention, Compaction, Logs

- There are actually 2 indices
  - An offset to position index - It helps Kafka know what part of a segment to read to find a message
  - A timestamp to offset index - It allows Kafka to find messages with a specific timestamp

# Indexes, Retention, Compaction, Logs



# Indexes, Retention, Compaction, Logs

- Kafka will not keep data forever
- It will not keep it in a single file
- It rolls over segment files based on
  - Time
  - Segment size (default 1GB)
- Time is controlled by log.roll.ms
- Size is controlled by log.segment.bytes

# Indexes, Retention, Compaction, Logs

- Improved Manageability: Smaller segment files are easier to handle, particularly for deletion and retention.
- More Frequent Compaction: Kafka's compaction process, which removes obsolete data, becomes more frequent with time-based rolling
- Trade-off: Too frequent log rolling increases the number of small files, which may slow down performance due to increased file system overhead. On the other hand, rolling too infrequently creates larger files that are slower to process during reads and compaction.

# Indexes, Retention, Compaction, Logs

- Kafka will store messages before they are deleted.
- Time Based retention - `log.retention.hours` – how long before they are deleted (also see `log.retention.ms`)
- Size Based retention – `log.retention.bytes` (default 10 GB)
- Compaction – Only latest version of key will be stored in logs.
- Question: Assume `log.retention.ms` has been set and segment is not closed because `log.segment.bytes` or `log.roll.ms` not set, what should happen? Why?

# Indexes, Retention, Compaction, Logs

```
[2020-12-24 15:51:17,808] INFO [Log partition=Topic-2,  
dir=/Folder/Kafka_data/kafka] Found deletable segments with base  
offsets [165828] due to retention time 604800000ms breach  
(kafka.log.Log)
```

```
[2020-12-24 15:51:17,808] INFO [Log partition=Topic-2,  
dir=/Folder/Kafka_data/kafka] Scheduling segments for deletion  
List(LogSegment(baseOffset=165828, size=895454171,  
lastModifiedTime=1608220234000, largestTime=1608220234478))  
(kafka.log.Log)
```

```
$KAFKA_HOME/bin/kafka-configs.sh --bootstrap-server :9092 --entity-type  
topics --entity-name my-topic --describe --all | grep segment.bytes
```

# Indexes, Retention, Compaction, Logs

- Broker Configs: `log.segment.bytes` and `log.roll.ms`
- Broker Configs: `log.retention.ms` and `log.retention.bytes`
- Per Topic: `retention.ms`, `retention.bytes`, `segment.bytes`, and `segment.ms`.

# Indexes, Retention, Compaction, Logs

## Common Modes

*(a) 1 Week of Retention*

`log.retention.hours=168`

`log.retention.bytes = -1`

`log.retention.ms = 604800000`

*(b) Indefinite retention time bounded by 500 MB*

`log.retention.hours = -1`

`log.retention.ms = -1`

`log.retention.bytes=524288000`

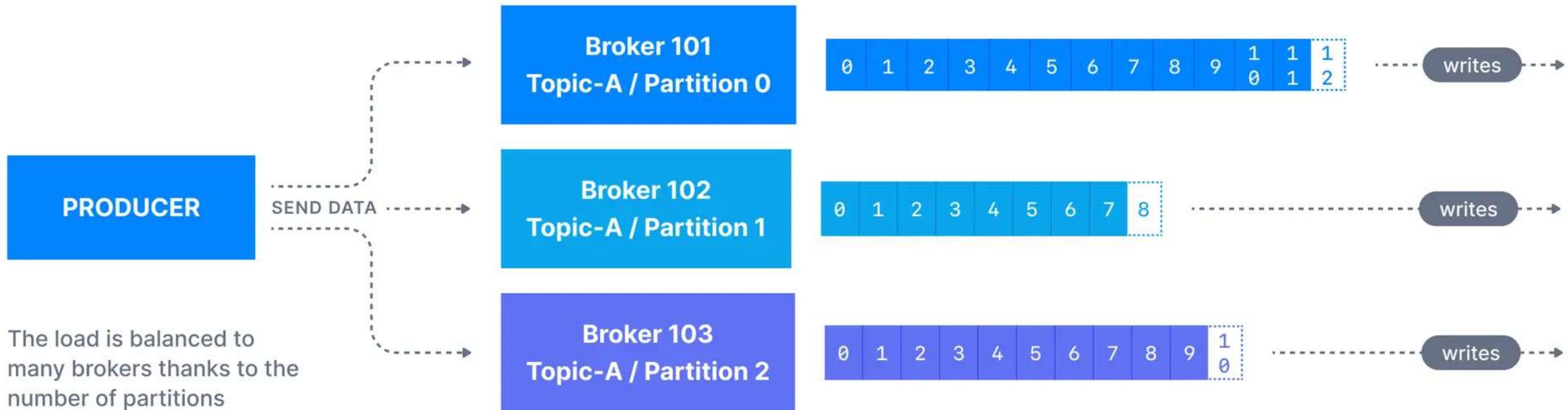
# Indexes, Retention, Compaction, Logs

- **Segment Size**
  - Smaller Segments: Can improve deletion times but increase the number of files, which adds file system overhead.
  - Larger Segments: Reduce the number of files but can slow down deletion and lead to slower recovery times after failures
- **Rolling Frequency**
  - Time-based Rolling: Ensures fresh segments are created at regular intervals, making it easier to manage log compaction and retention.
  - Size-based Rolling: Prevents segments from becoming too large, which improves write and read performance.
- **Retention Policies**
  - Time-based Retention: Suitable for streaming applications where you need to retain data for a fixed period.
  - Size-based Retention: Ideal for environments with limited storage capacity.
  - Compaction: Useful for use cases where only the latest state of each key is needed, such as changelog topics in stateful applications.

# Replicas

- By Default, partitions only have one replica.
- If partition fails, data is lost.
- For High availability, the number of replicas (or replication factor) is set to  $> 1$
- More on this when we talk of producers.
- In later sections we will talk about HWM and LWM for replicas when discussing logs and log analysis

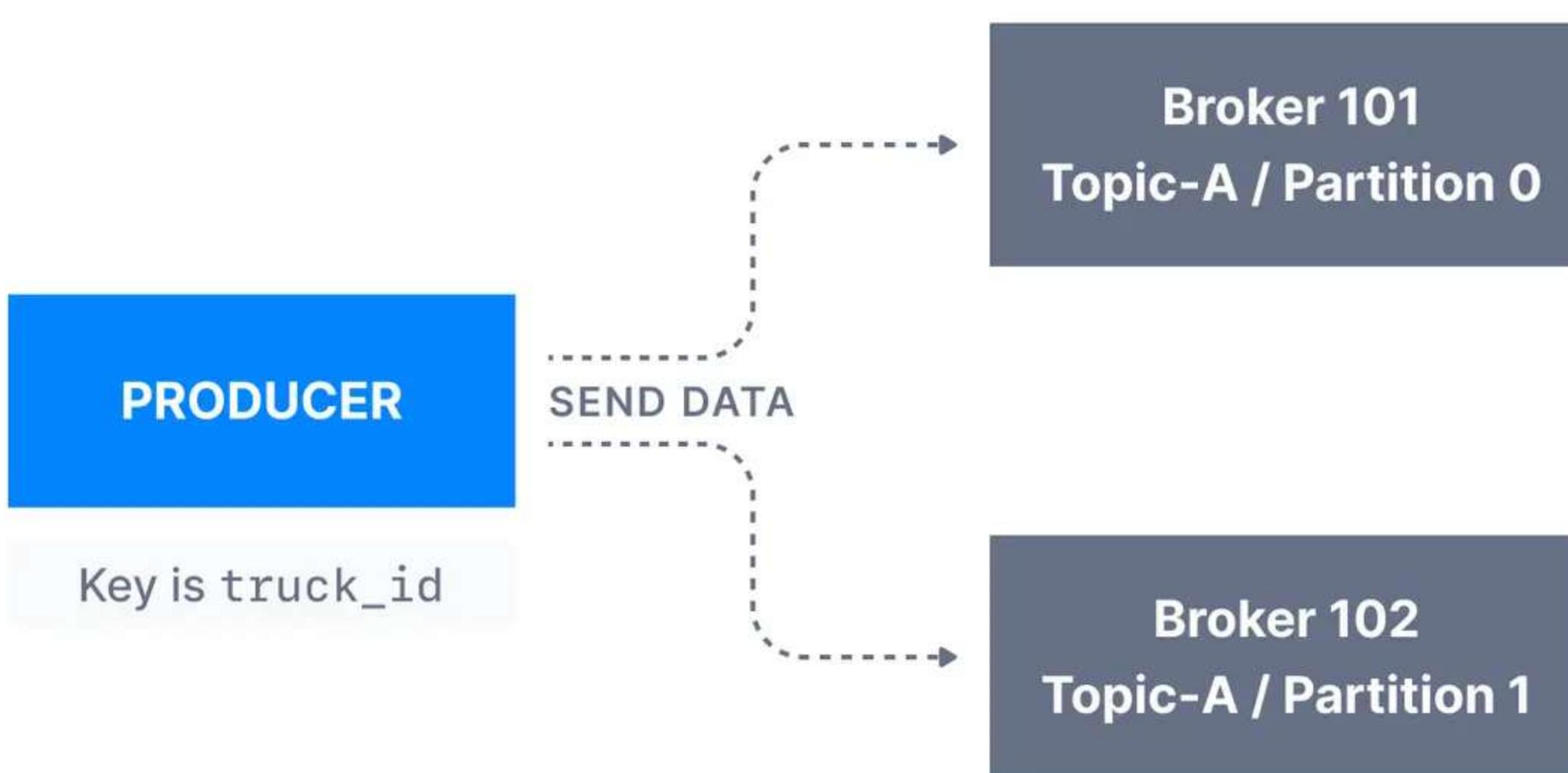
# Producers, Consumers, Replicas



# Producers, Consumers, Replicas

- For a message to be written into a topic, a producer should specify a level of acknowledgment (ACKS)
- Every Message contains an optional key and a value.
- Key selection is very critical to Kafka Performance and load balancing.
- E.g. Customer\_ID, Vehicle\_ID

# Producers, Consumers, Replicas



## Example:

truck\_id\_123 data will always be in partition 0

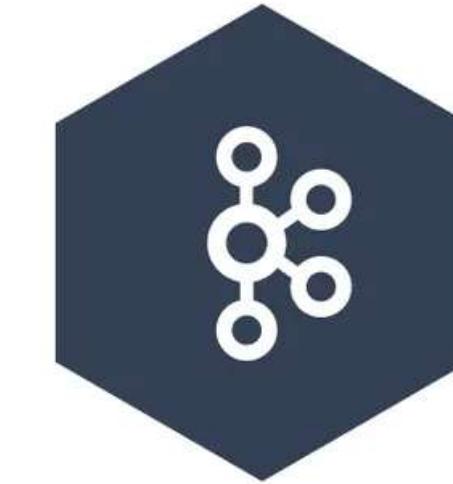
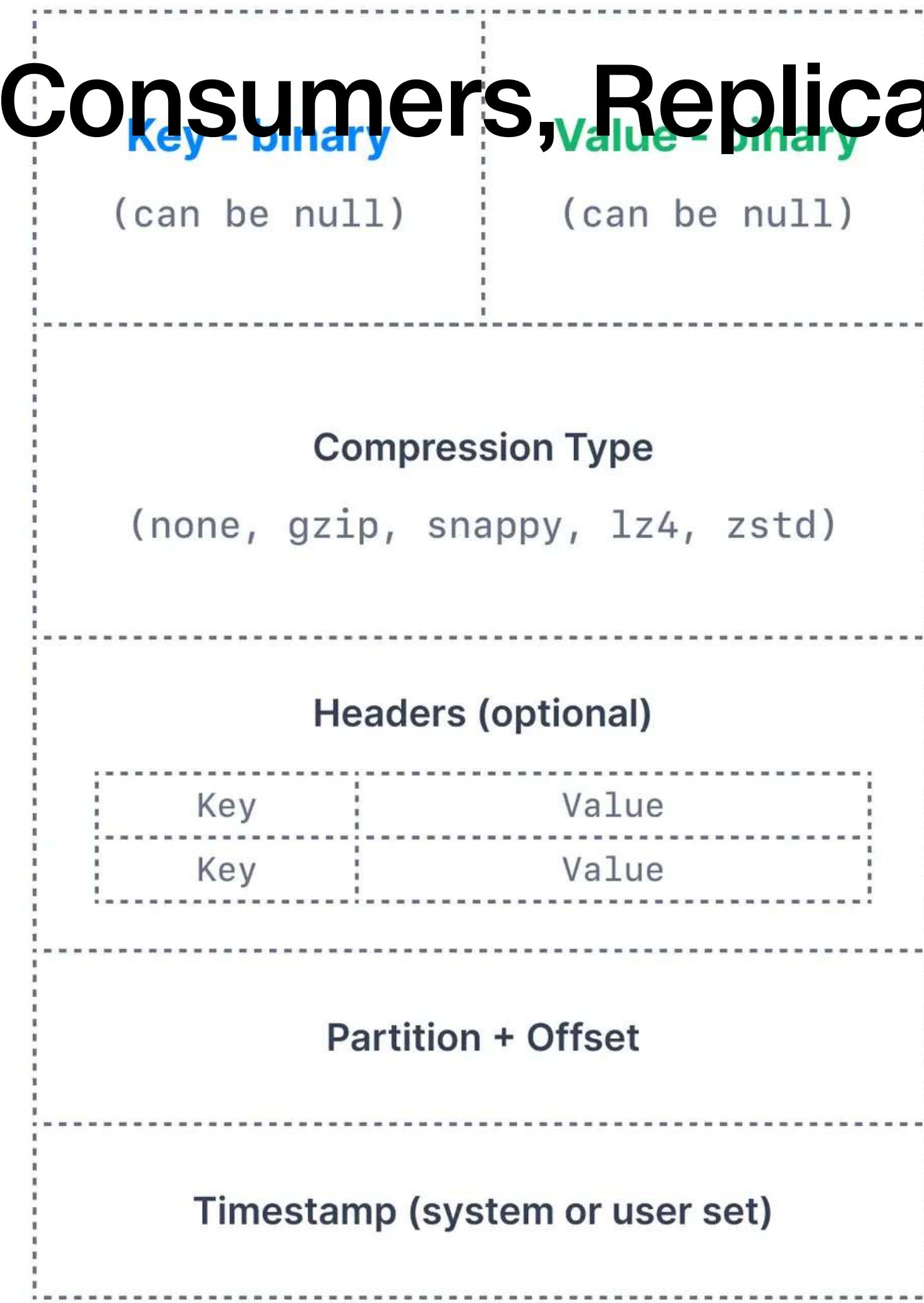
truck\_id\_234 data will always be in partition 0

truck\_id\_345 data will always be in partition 1

truck\_id\_456 data will always be in partition 1

# Producers, Consumers, Replicas

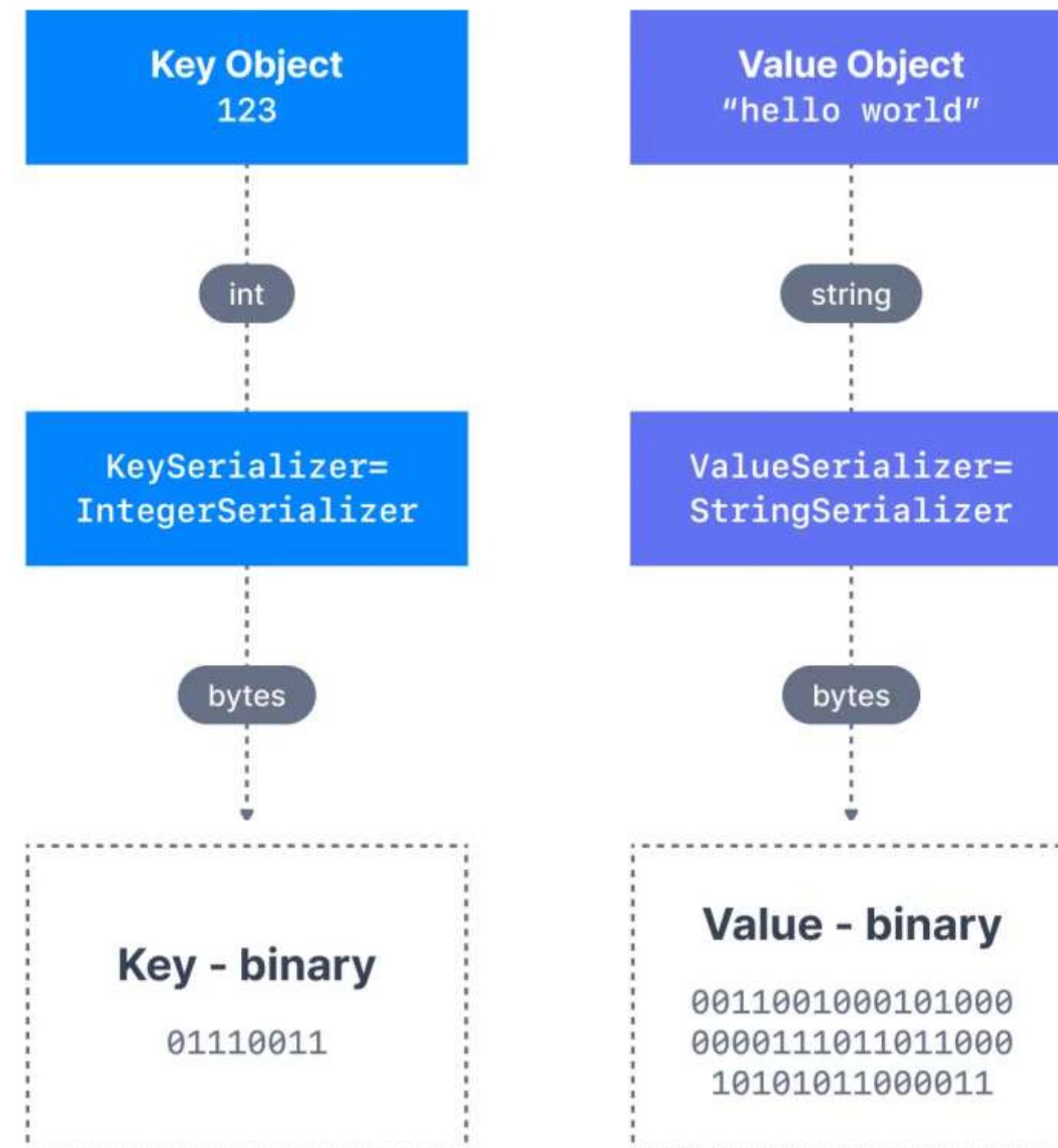
Kafka  
Message  
Created by  
the producer



# Producers, Consumers, Replicas

- Key+Value are really objects ☺
- They are in essence passed as byte[]
- Serialize/Deserialize

# Producers, Consumers, Replicas



# Producers, Consumers, Replicas

- Several serializers already exist, such as string (which supersedes JSON), integer, float.
- Other serializers may have to be written by the users, but commonly distributed Kafka serializers exist and are efficiently written for formats such as
  - JSON-Schema,
  - Apache Avro
  - and Protobuf, thanks to the Confluent Schema Registry.

# Producers, Consumers, Replicas

- So how is a key Hashed?

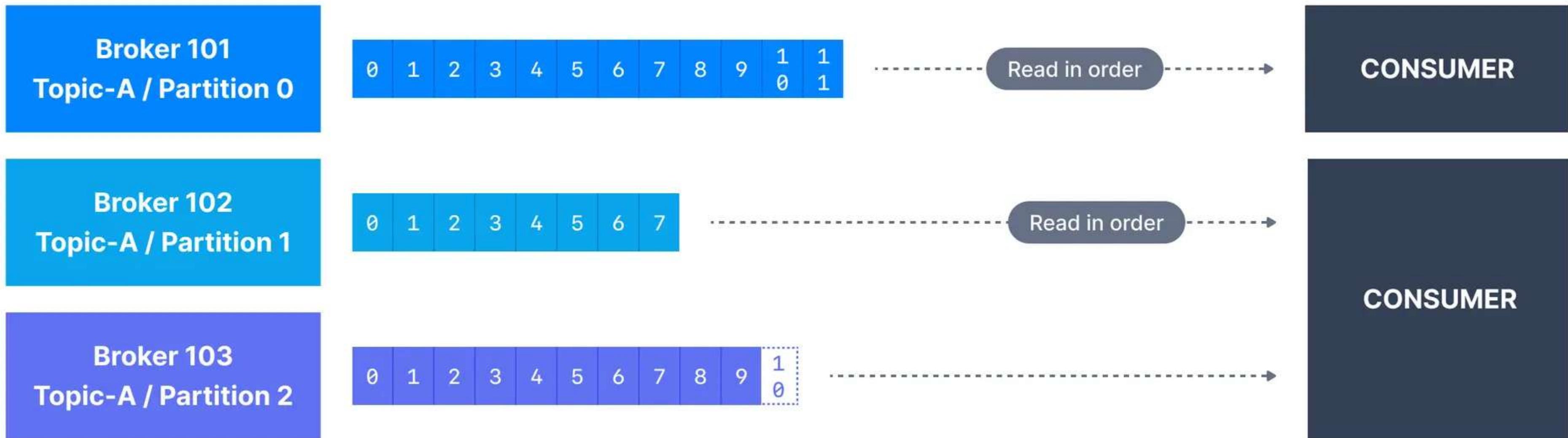


# Producers, Consumers, Replicas

- Default uses an algorithm : murmur2 algorithm,
- `targetPartition = Math.abs(Utils.murmur2(keyBytes)) % (numPartitions - 1)`
- <https://murmurhash.shorelabs.com/>
- <https://md5hashing.net/hash/murmur3/>

# Producers, Consumers, Replicas

- We will cover ACKS for later.
- Consumers are apps that read from partitions.



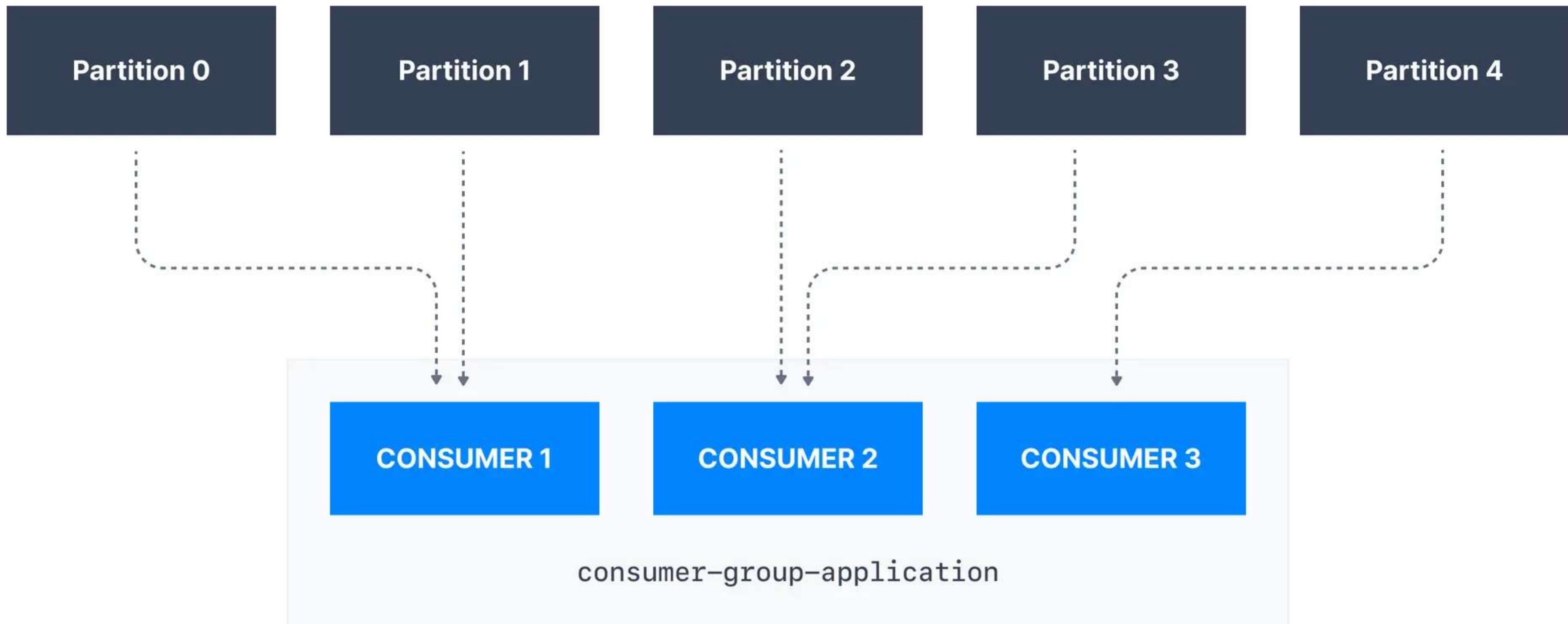
# Producers, Consumers, Replicas

- Consumers cannot read backwards.
- Data read from multiple partitions is not guaranteed to be in orders.
- By default, consumers read only data after they first connect to Kafka.
- Issues – Data serialized by producers have to be deserialized in the same format.
- Format should not change during topic life cycle. If you do need to do that, consider moving to new topics

# Producers, Consumers, Replicas

- Consumers that are part of the same application and therefore performing the same "logical job" can be grouped together as a Kafka consumer group.
- A topic usually consists of many partitions. These partitions are a unit of parallelism for Kafka consumers.
- The benefit of leveraging a Kafka consumer group is that the consumers within the group will coordinate to split the work of reading from different partitions.

# Producers, Consumers, Replicas



# Producers, Consumers, Replicas

- Consumer offsets (not to be confused with message offsets) are used to “checkpoint” how far the consumer has been reading the partition.
- Consumers frequently commit the latest processed message – consumer offsets
- The commit is usually turned on by default.
- Data for this can be found in the `_consumer_offsets` topic.

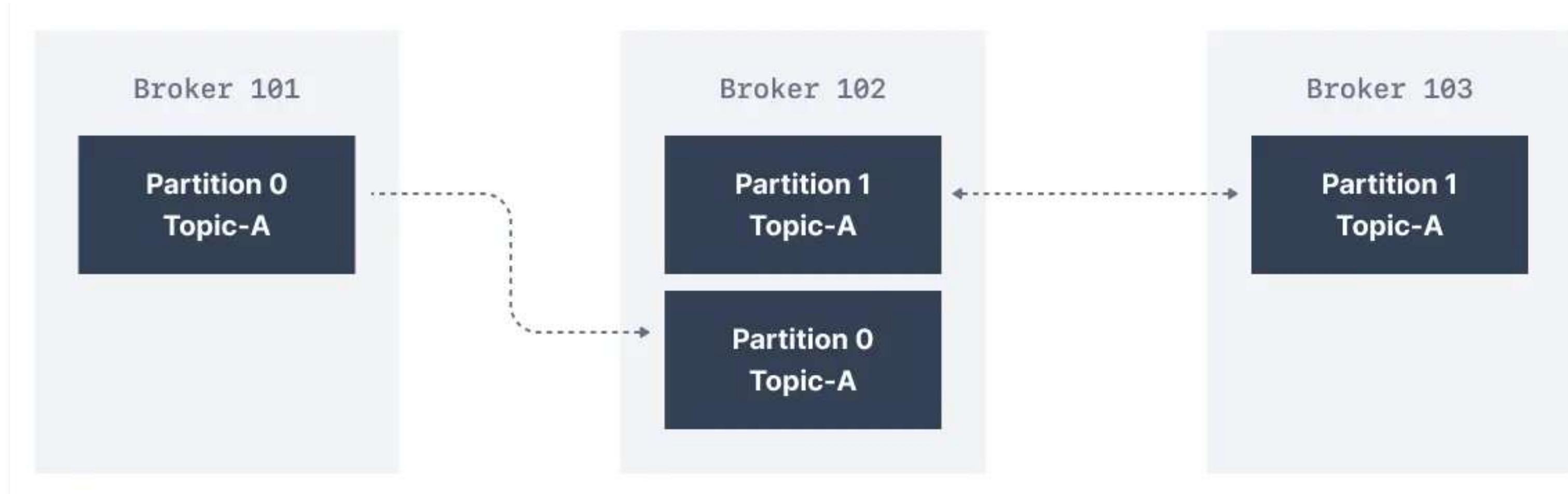
# Producers, Consumers, Replicas

- There are 2 issues that come in from here
  - A. Consumer lag – how far behind the actual message header
  - B. Crash Recovery
- The commit behavior controlled by
  - **enable.auto.commit = true|false**
  - **auto.commit.interval.ms (default 5 s)**

# Producers, Consumers, Replicas

- Depending on these, delivery to Customer is
- At Most once – Offsets are committed as soon as message is received. Processing fails, message is gone 😊
- At least once (preferred) – Offsets are committed after message processing. If failure, message is re-read with possible duplicates. → Commit anyways and send to DLQs
- Exactly Once – Topic to Topic Workflow using the **Transactions API**. In Kafka Streams, set `processing.guarantee=exactly_once_v2`

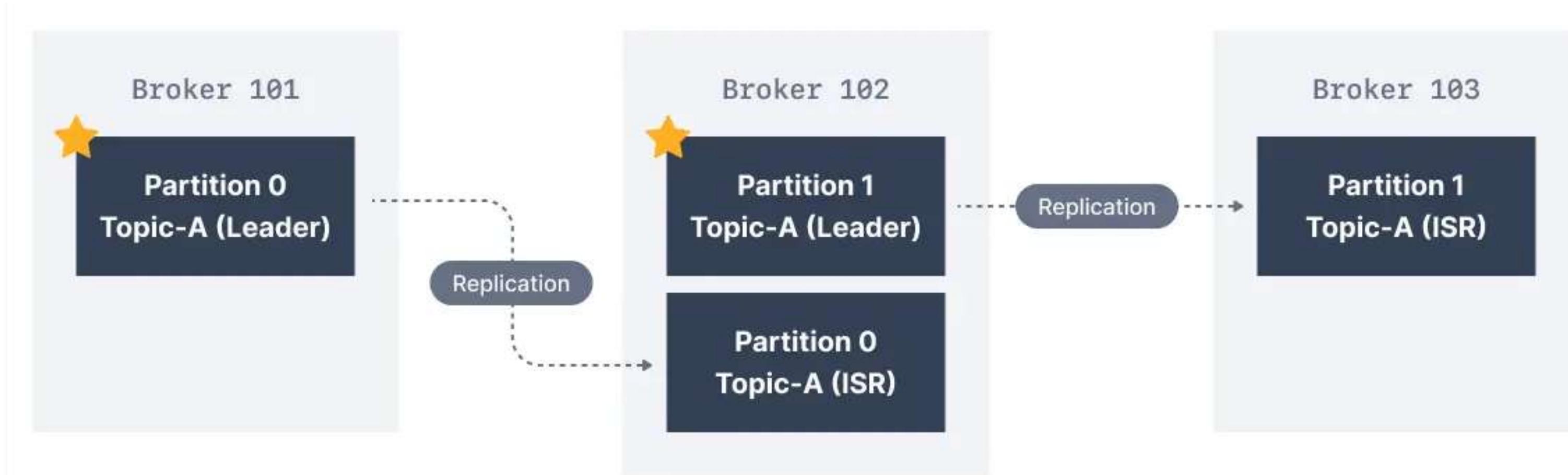
# Producers, Consumers, Replicas



# Producers, Consumers, Replicas

- In practice a replication factor of 3 allows for 2 failures.
- It is a via medium between resiliency and performance
- For a given topic/partition, one broker is designated as a leader. (Others are called replicas)
- An ISR or In-Sync Replicas, is a replica that's up-to-date with leader broker.

# Producers, Consumers, Replicas



# Producers, Consumers, Replicas

- Kafka Producers will only write to the current leader.
- They must also specify ACK level before message is written
  - essentially to how many replicas should it be written before it's considered good.
- For Kafka  $\geq 3.0$ , default value is all
- For Kafka  $< 3.0$ , default ack value is 1
- An acks value of 0 – do not wait for the broker to acknowledge.

# Producers, Consumers, Replicas

- Parameters to consider
- `min.insync.replicas` = min no of ISR to replicate to
  - Set at both Topic as well as Broker.
  - Question: if `min.insync.replicas` is set to 2 and total number of replicas are only 2, what do you expect the behavior to be?

# Producers, Consumers, Replicas

- A producer can throw an exception. Here's the scenario
  - A. Producer sends data to broker
  - B. Brokers persists data – no issue.
  - C. Broker dies before ACK is sent or
  - D. Network issue cause timeout
- Producer has not received the ACK and resends the message which is a duplicate message.

# Producers, Consumers, Replicas

- `enable.idempotence = true`

Kafka producer uses the following mechanisms to ensure exactly-once delivery:

- Producer ID (PID)
  - Each producer instance is assigned a unique Producer ID (PID) when it connects to a Kafka broker. This ID helps Kafka track messages sent by the producer
- Sequence Number
  - Each message sent by the producer to a specific partition is assigned a monotonically increasing sequence number.
- Broker Validation
  - Brokers track the latest sequence number for each partition and producer. Any record with a duplicate sequence number is discarded.

# Producers, Consumers, Replicas



- **Duplicate-Free Delivery:**
  - Ensures messages are delivered exactly once, even during retries.
- **Simplified Error Handling:**
  - Eliminates the need for custom deduplication logic in applications.
- **Increased Reliability:**
  - Improves consistency and reliability in distributed systems.
- **Foundation for Transactions:**
  - Idempotence is a prerequisite for Kafka's transactional messaging feature, enabling atomic writes across topics and partitions.

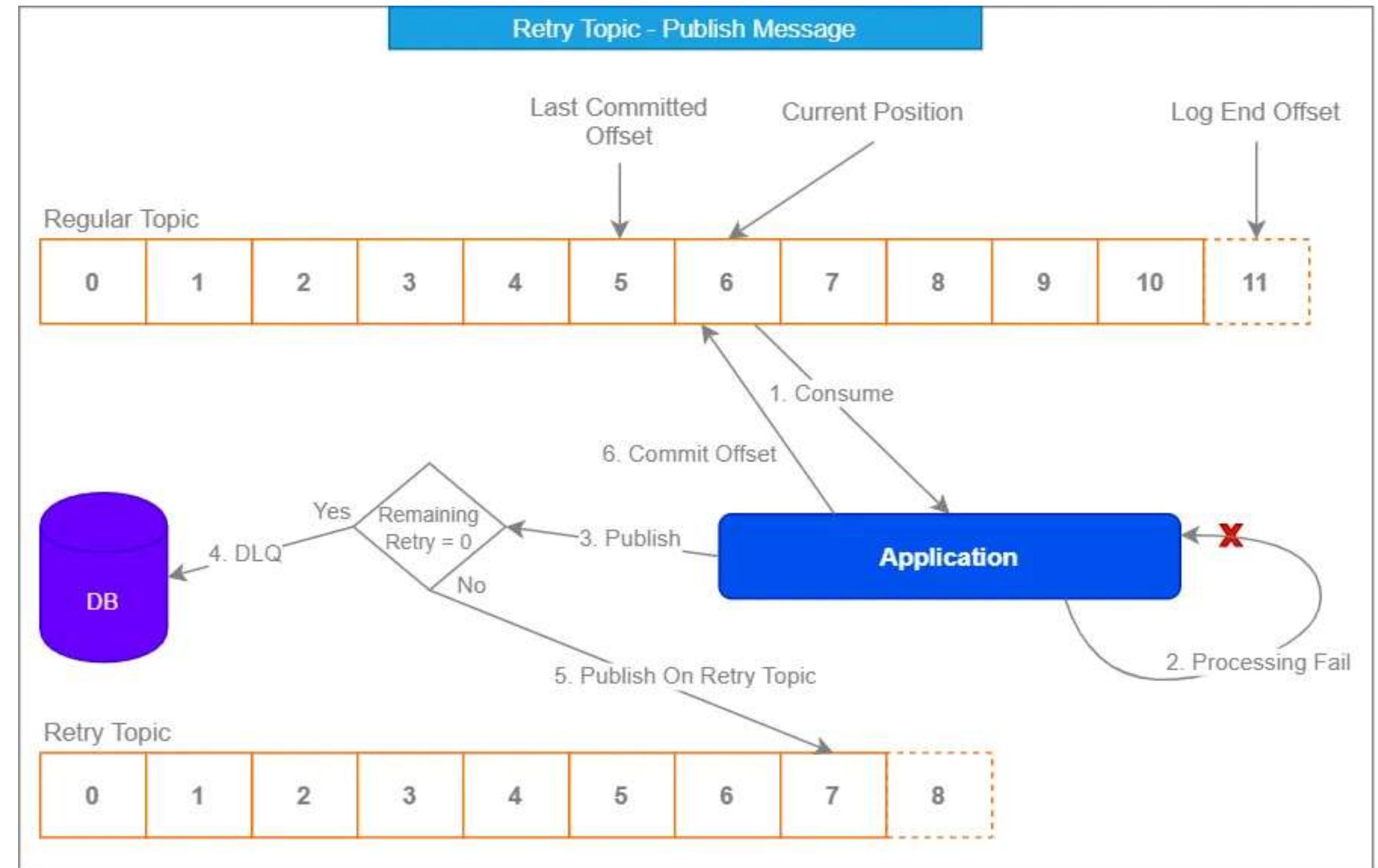
# Producers, Consumers, Replicas



- **Partition-Specific Guarantee:**
  - Idempotence works at the partition level. Duplicate messages sent across different partitions are not deduplicated.
- **Does Not Cover Consumers:**
  - Idempotence guarantees apply to producer-to-broker communication. To achieve exactly-once semantics for consumers, you need to use Kafka's transactions feature.
- **Increased Latency:**
  - With `acks=all` and sequence tracking, latency may increase slightly compared to non-idempotent producers.

# Producers, Consumers, Replicas

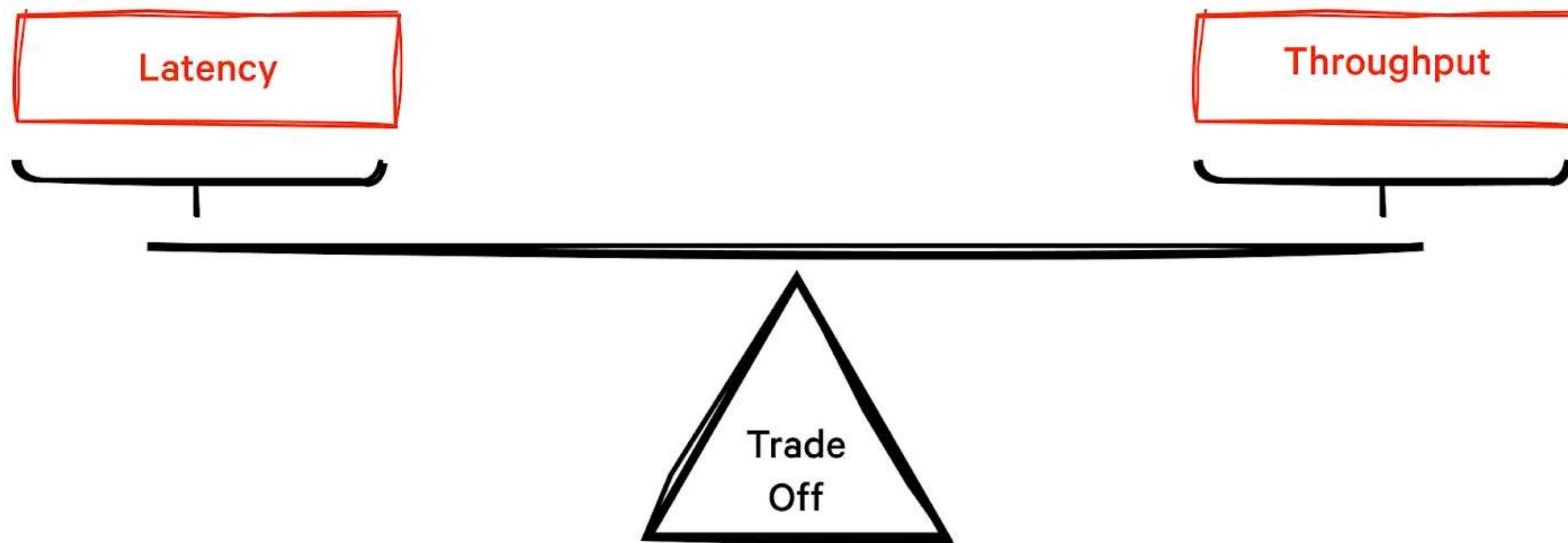
- One Pattern is the use of Retry Logics and DLQs.



# Producers, Consumers, Replicas

- Retries Can be enriched with the following
  - Metadata about number of tries left
  - Metadata about consumer group id
  - Next timestamp
  - Exponential backoffs
  - Retry topic / topic (or single Retry Topic for all)
- Once no more retries are possible, put it into DLQ and be done.

# Producers, Consumers, Replicas



# Producers, Consumers, Replicas

- Latency measures how long it takes for Kafka to fetch or pull a single message.
- It is the time gap between the producer generating a message and the consumer consuming it.
- Low latency is critical for real-time applications, where delays in processing have significant consequences.
- To reduce latency, you optimize your Kafka configuration to minimize the time it takes for Kafka to process a single event. You can consider strategies like:
  - Tuning the number of partitions and replication factors
  - Optimizing the hardware and network configurations
  - Using compression to reduce the size of the data Kafka processes.

# Producers, Consumers, Replicas

- Throughput measures how many messages Kafka can process in a given period.
- High throughput is essential for applications that process large amounts of data quickly.
- To maximize throughput, you optimize your Kafka configuration to handle as many events as possible within a given time frame.
- Strategies include:
  - Increasing the batch size
  - Increasing the number of producer threads
  - Increasing the number of partitions

# Producers, Consumers, Replicas

- Batch Requests for Producers
  - batch.size and linger.ms e.g.  
*Properties props = new Properties();  
props.put("batch.size", 16384);  
props.put("linger.ms", 5);*
- Compress data before sending to Kafka e.g.
  - *props.put("compression.type", "snappy");*
- Use Idempotent Producers.
  - Tradeoff in latency is IMHO worth hassle of De-duplicating applications ☺

# Producers, Consumers, Replicas

Compression type	Compression ratio	CPU usage	Compression speed	Network bandwidth usage
Gzip	Highest	Highest	Slowest	Lowest
Snappy	Medium	Moderate	Moderate	Medium
Lz4	Low	Lowest	Fastest	Highest
Zstd	Medium	Moderate	Moderate	Medium

# Producers, Consumers, Replicas

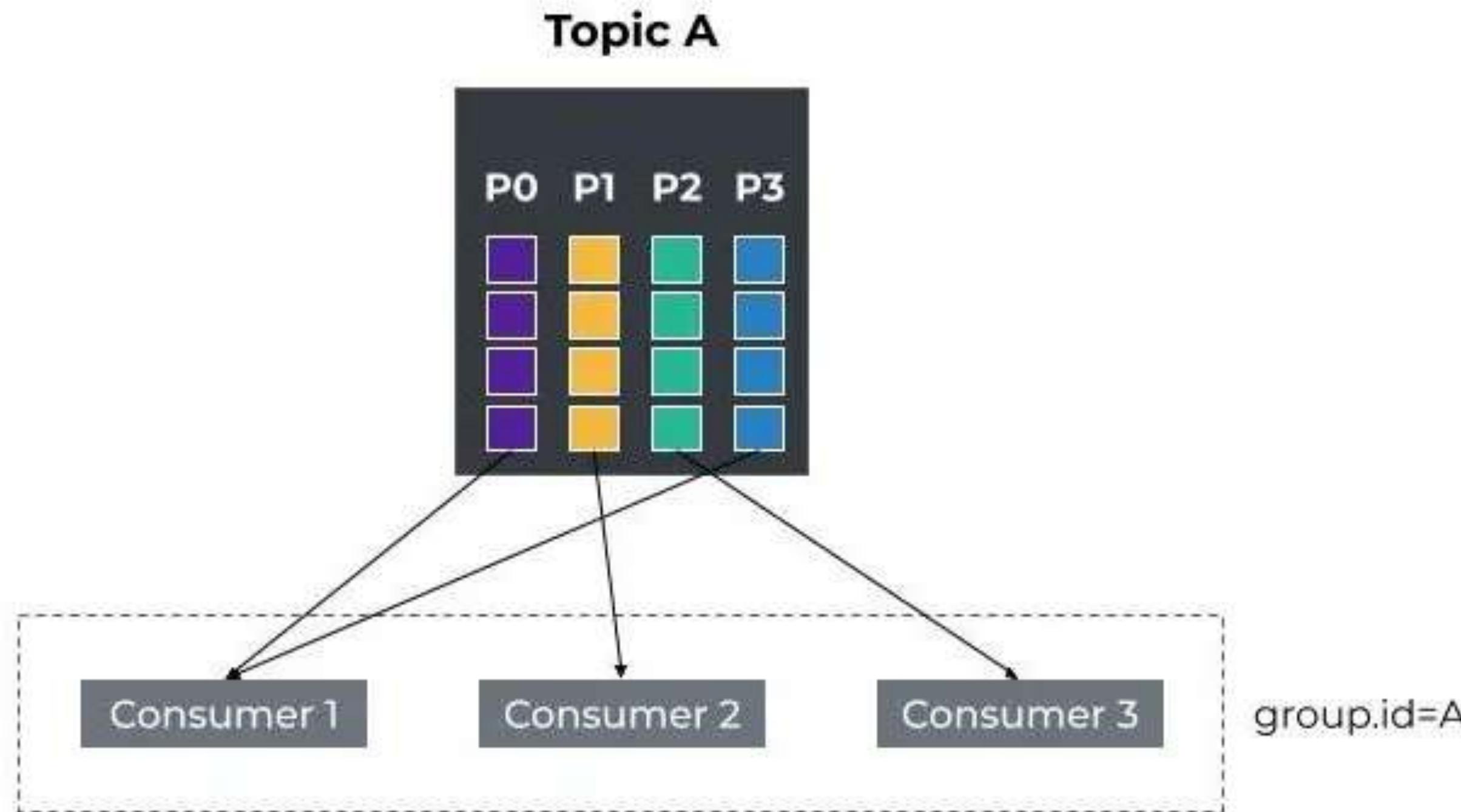
- Trade-off between Compression and CPU usage
- BTW, none is the default -

# KAFKA PARTITIONING STRATEGIES

- On the producer side, the partitions allow writing messages in parallel.
- If a message is published with a key, then, by default, the producer will hash the given key to determine the destination partition.
- This provides a guarantee that all messages with the same key will be sent to the same partition.
- In addition, a consumer will have the guarantee of getting messages delivered in order for that partition.

# Understanding Kafka Replicas

# KAFKA PARTITIONING STRATEGIES



# KAFKA PARTITIONING STRATEGIES

- ***Rebalance/Rebalancing***: the procedure that is followed by a number of distributed processes that use Kafka clients and/or the Kafka coordinator to form a common group and distribute a set of resources among the members of the group

(source : [Incremental Cooperative Rebalancing: Support and Policies](#)).

# KAFKA PARTITIONING STRATEGIES

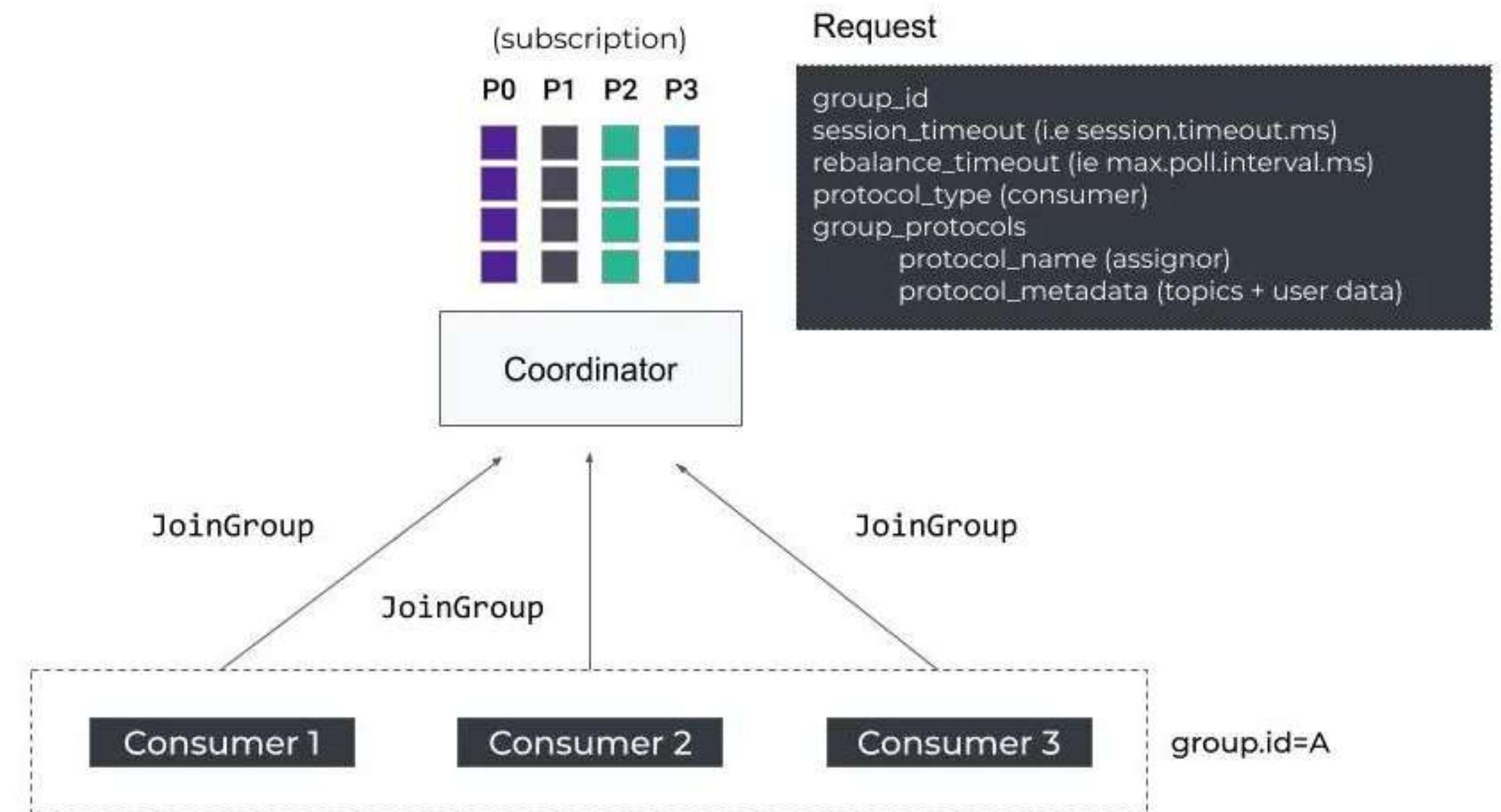


# UNDERSTANDING

- The Group Membership Protocol is in charge of the coordination of members within a group.
- The clients participating in a group will execute a sequence of requests/responses with a Kafka broker that acts as coordinator.
- Client Embedded Protocols run on the client side.

# KAFKA PARTITIONING STRATEGIES

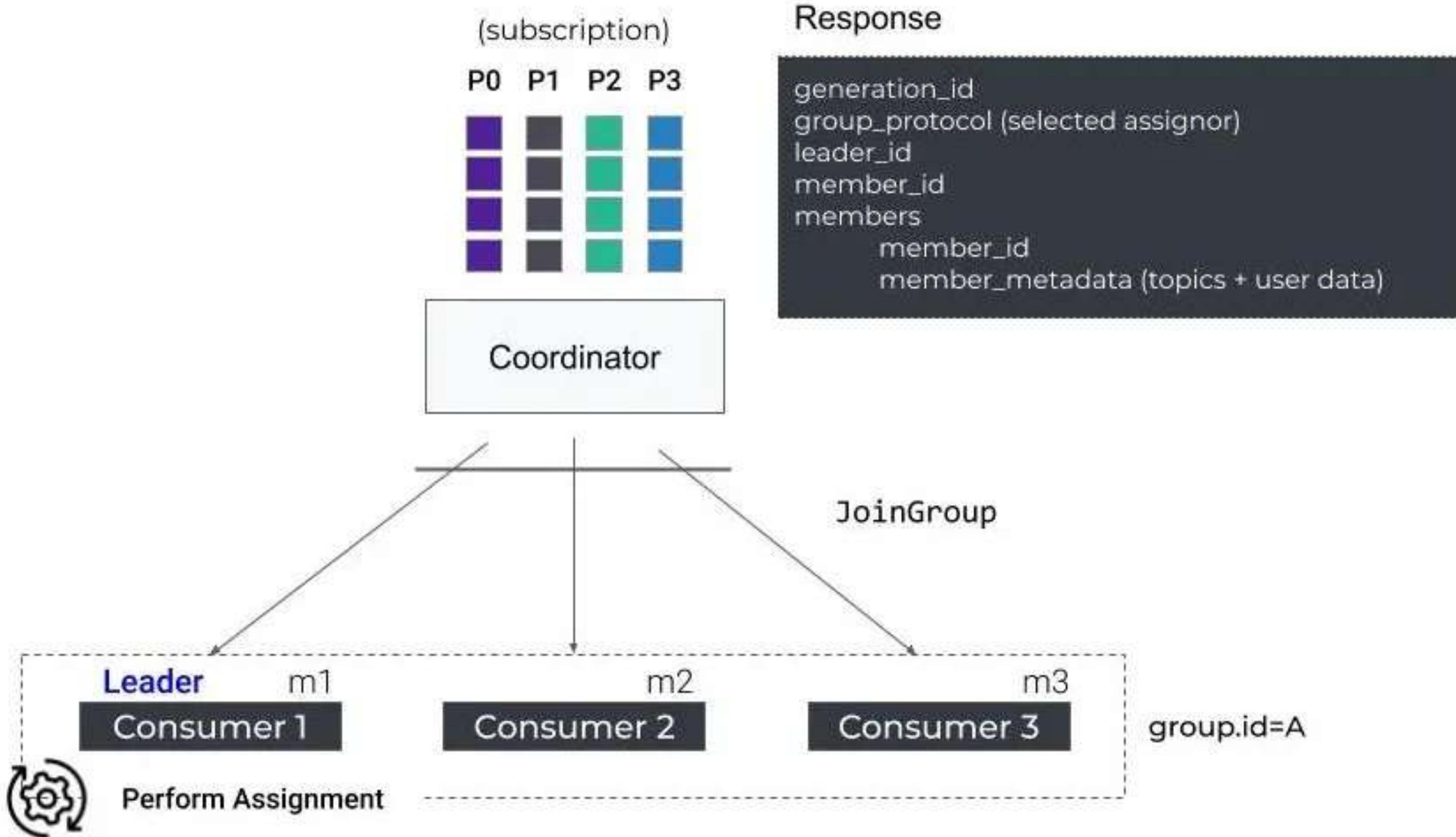
- A consumer sends the **FindCoordinator** request.
- It then sends a **JoinGroup** Request



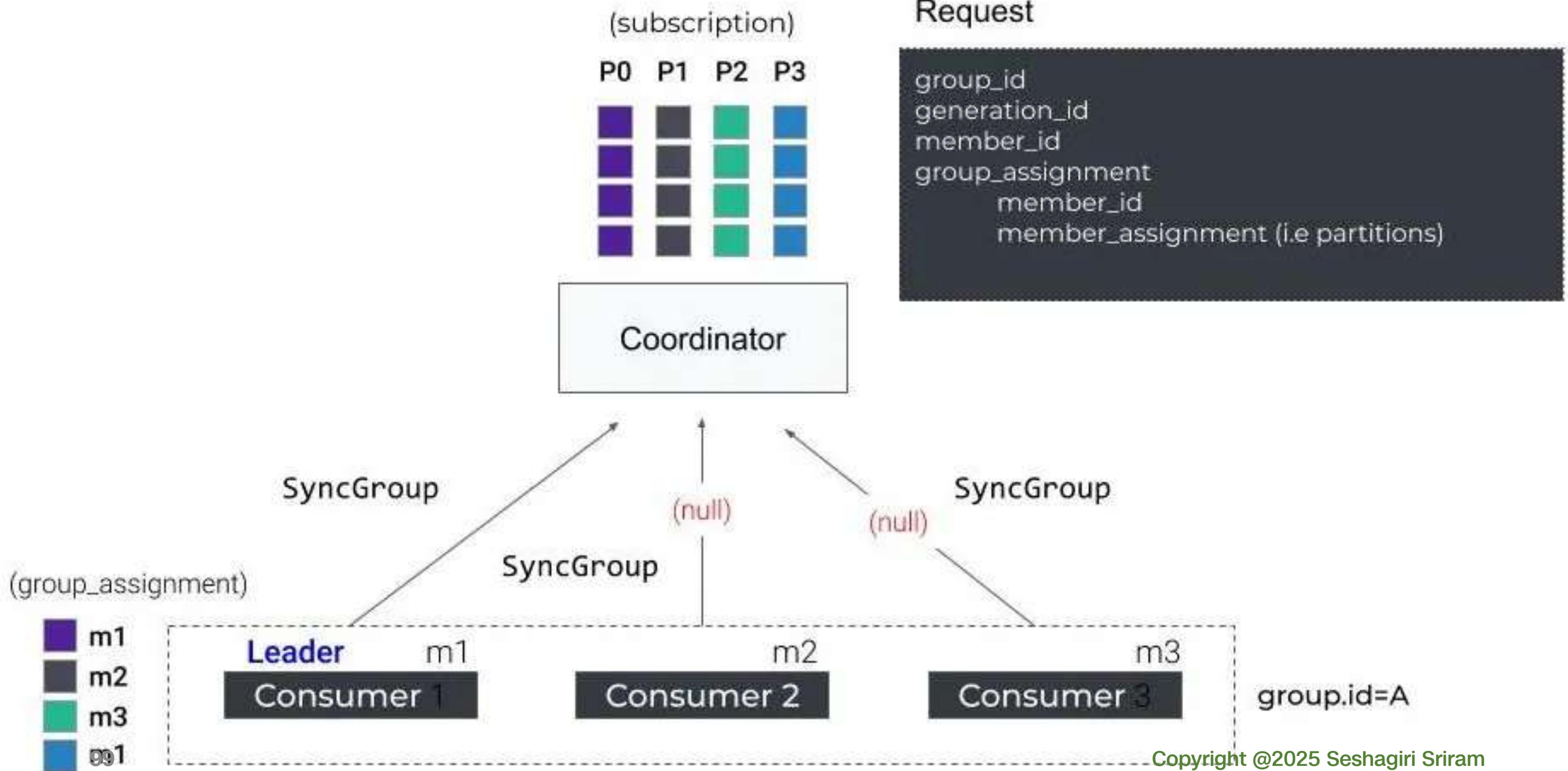
# KAFKA PARTITIONING STRATEGIES

- JoinGroup contains Consumer client Configuration like timeout, Max Poll interval etc.
- It also contains
  - List of protocols e.g. Partitioning Strategy
  - Metadata e.g. List of Topics consumer has subscribed to
- Coordinator does not send response immediately.
  - Rebalance timeout (or) group.initial.rebalance.delay has to be exceeded.

# KAFKA PARTITIONING STRATEGIES



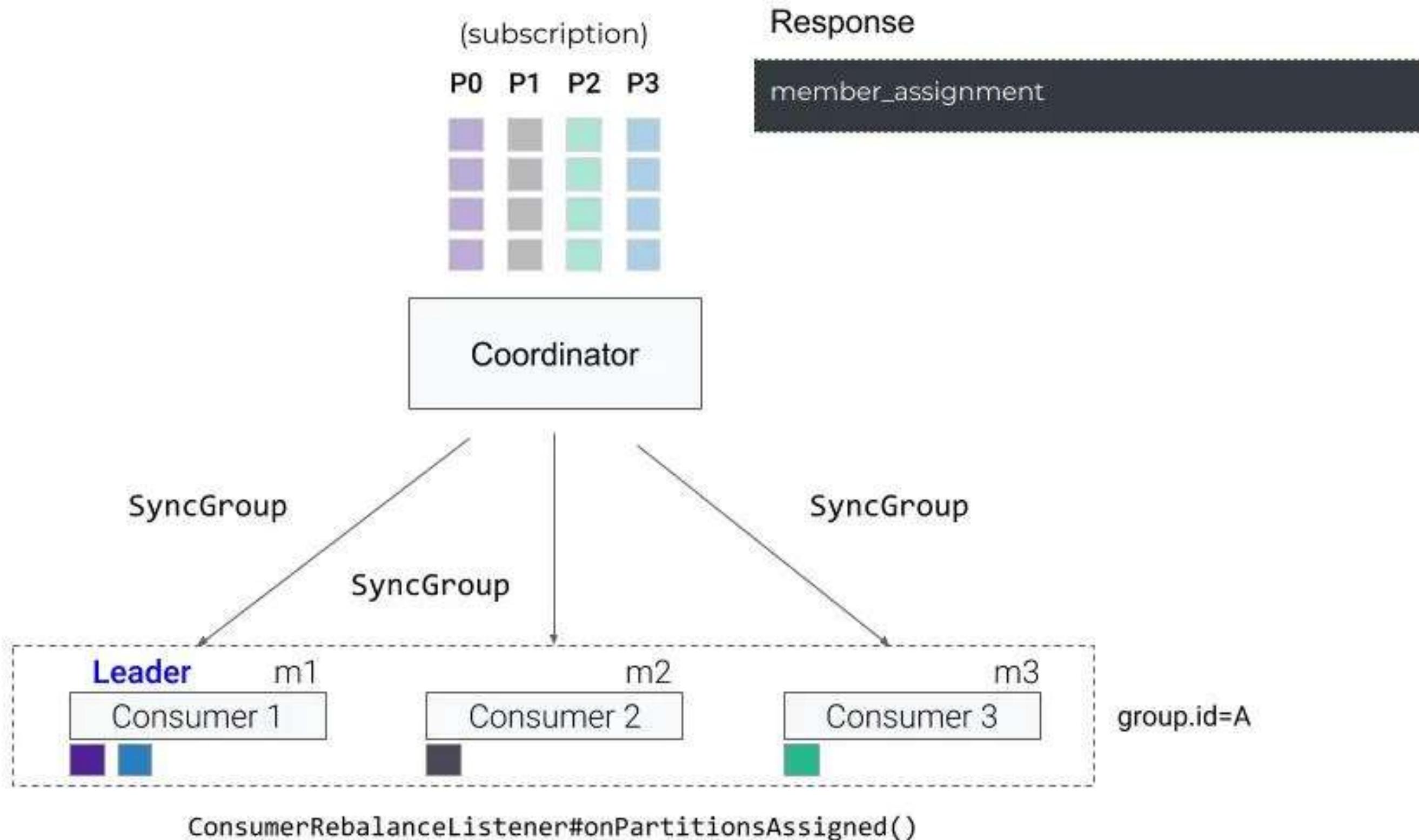
# KAFKA PARTITIONING STRATEGIES



# KAFKA PARTITIONING STRATEGIES

- Once Coordinator has received and responded to all SyncGroup Requests, each consumer get information on the assigned partition
- They start the `onPartitionAssignedMethod` on configured listener and fetch messages.

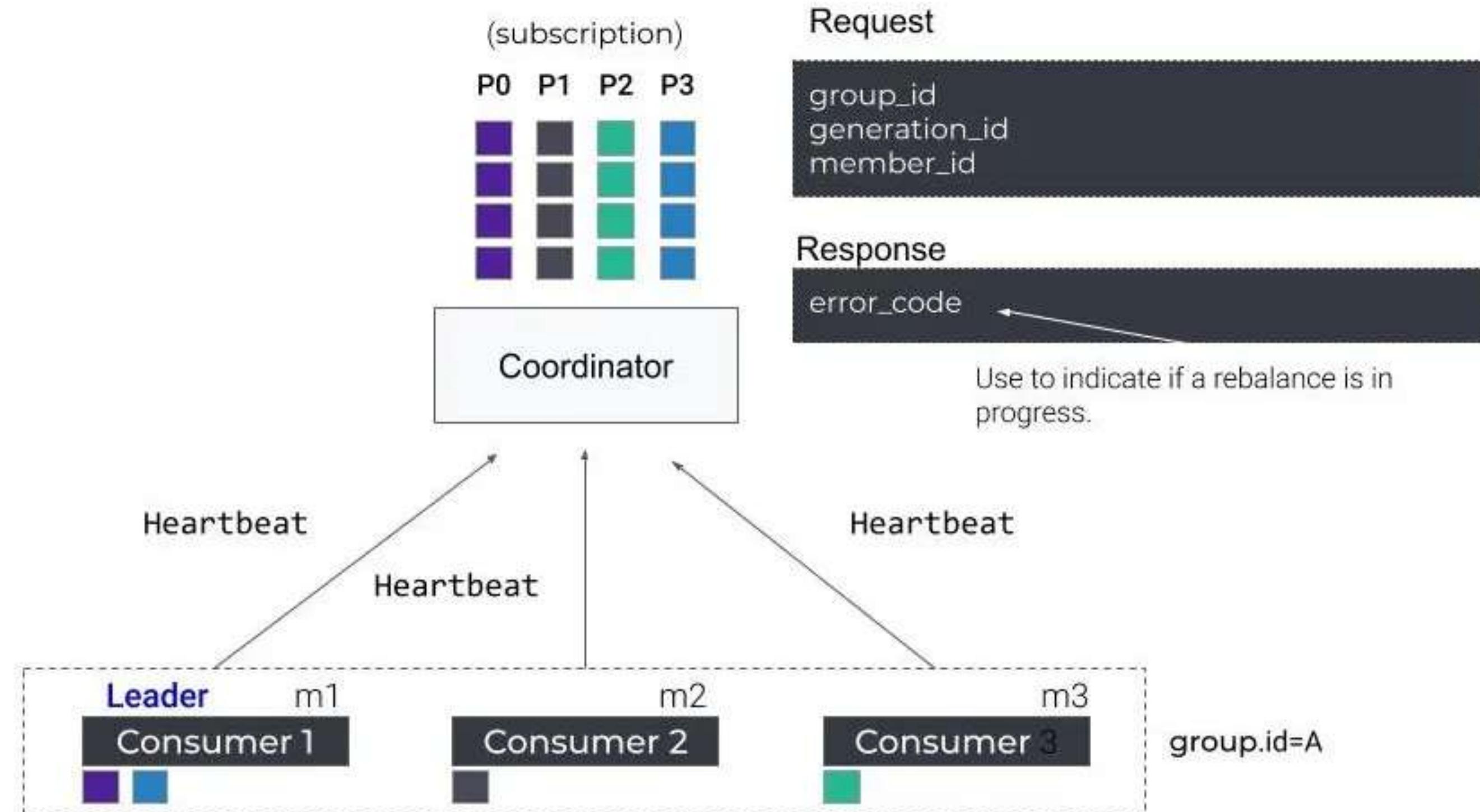
# KAFKA PARTITIONING STRATEGIES



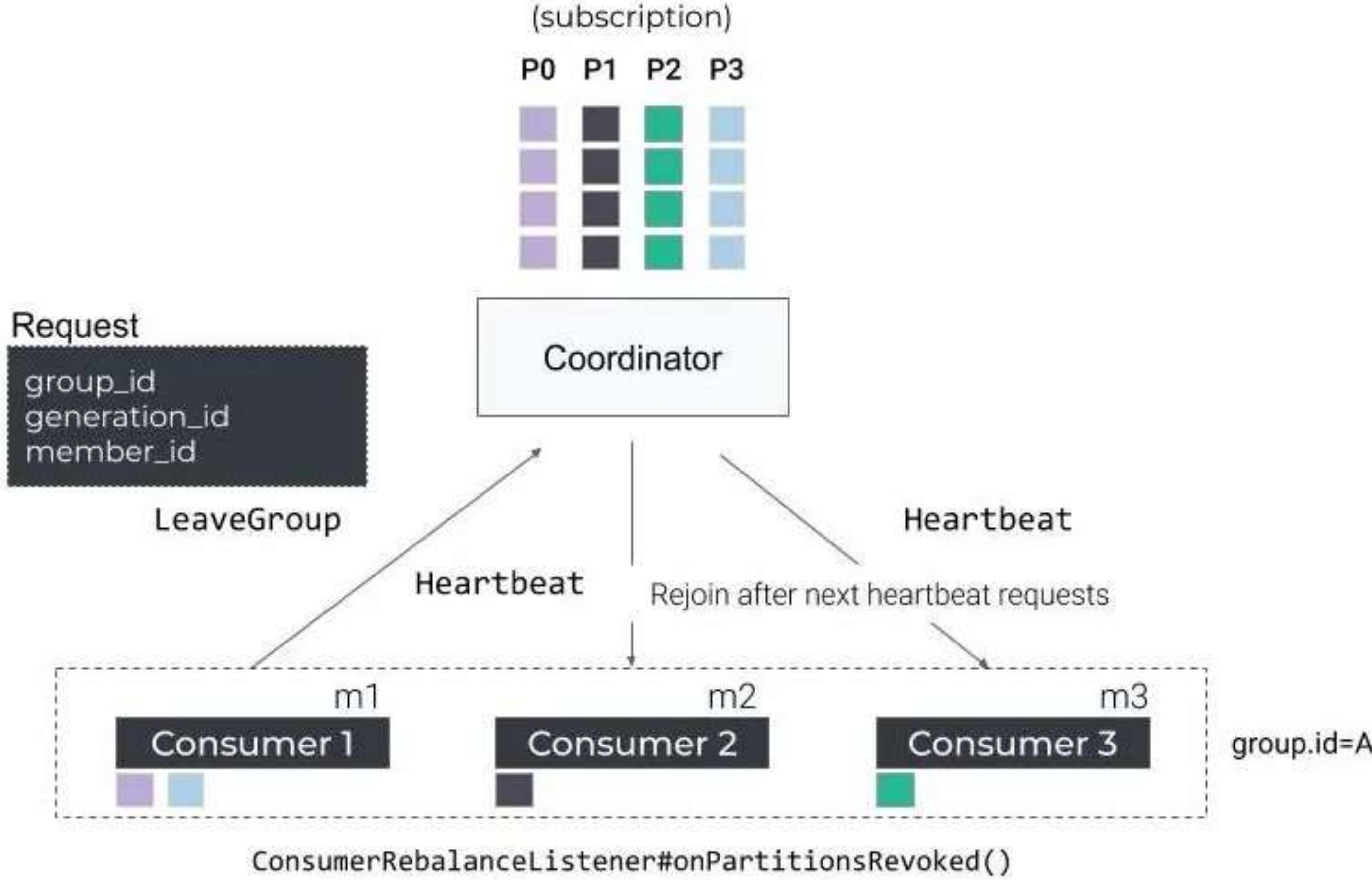
# KAFKA PARTITIONING STRATEGIES

- Last but not least, each consumer periodically sends a Heartbeat request to the broker coordinator to keep its session alive (see : `heartbeat.interval.ms`).
- If a rebalance is in progress, the coordinator uses the Heartbeat response to indicate to consumers that they need to rejoin the group.

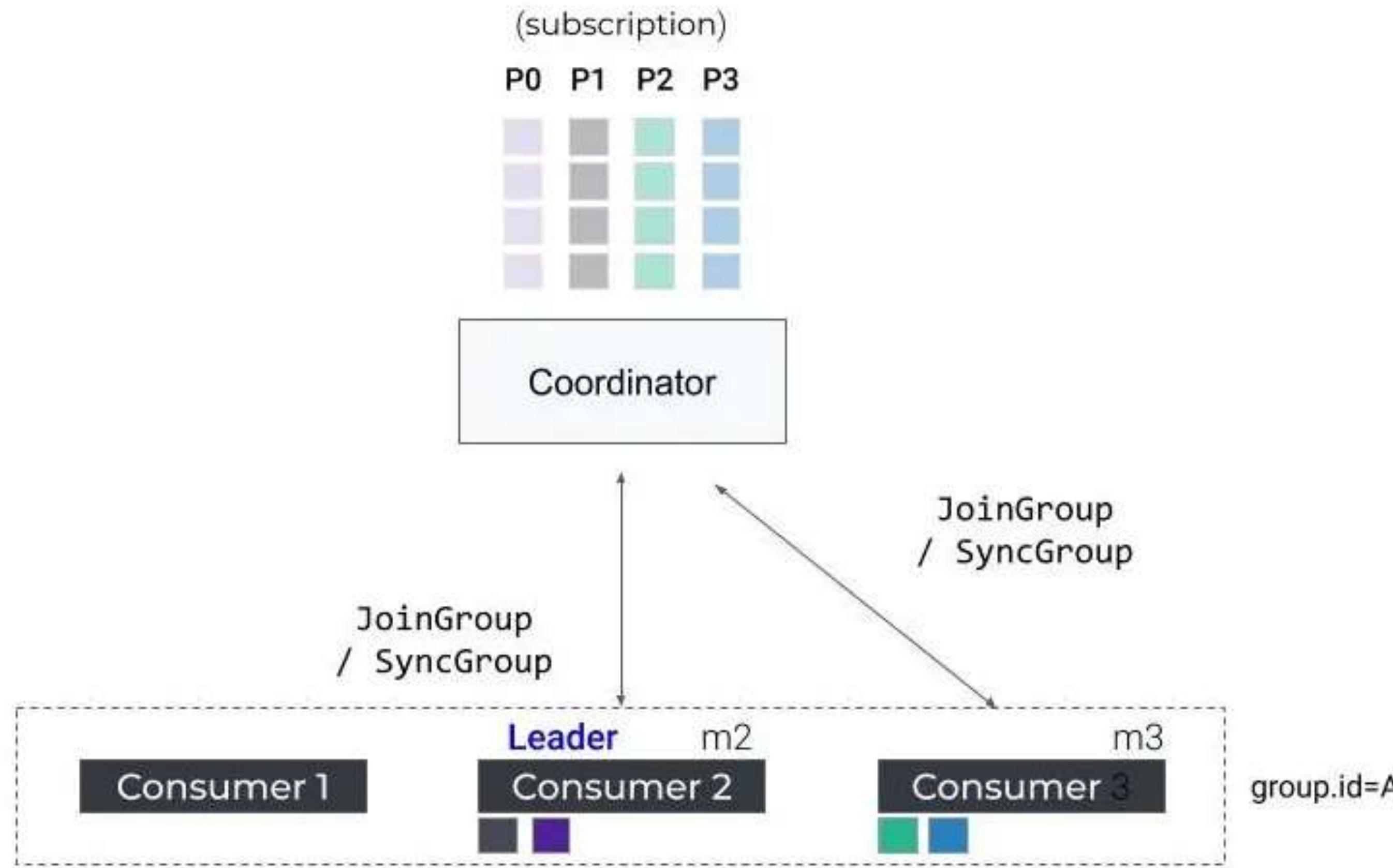
# KAFKA PARTITIONING STRATEGIES



# KAFKA PARTITIONING STRATEGIES



# KAFKA PARTITIONING STRATEGIES



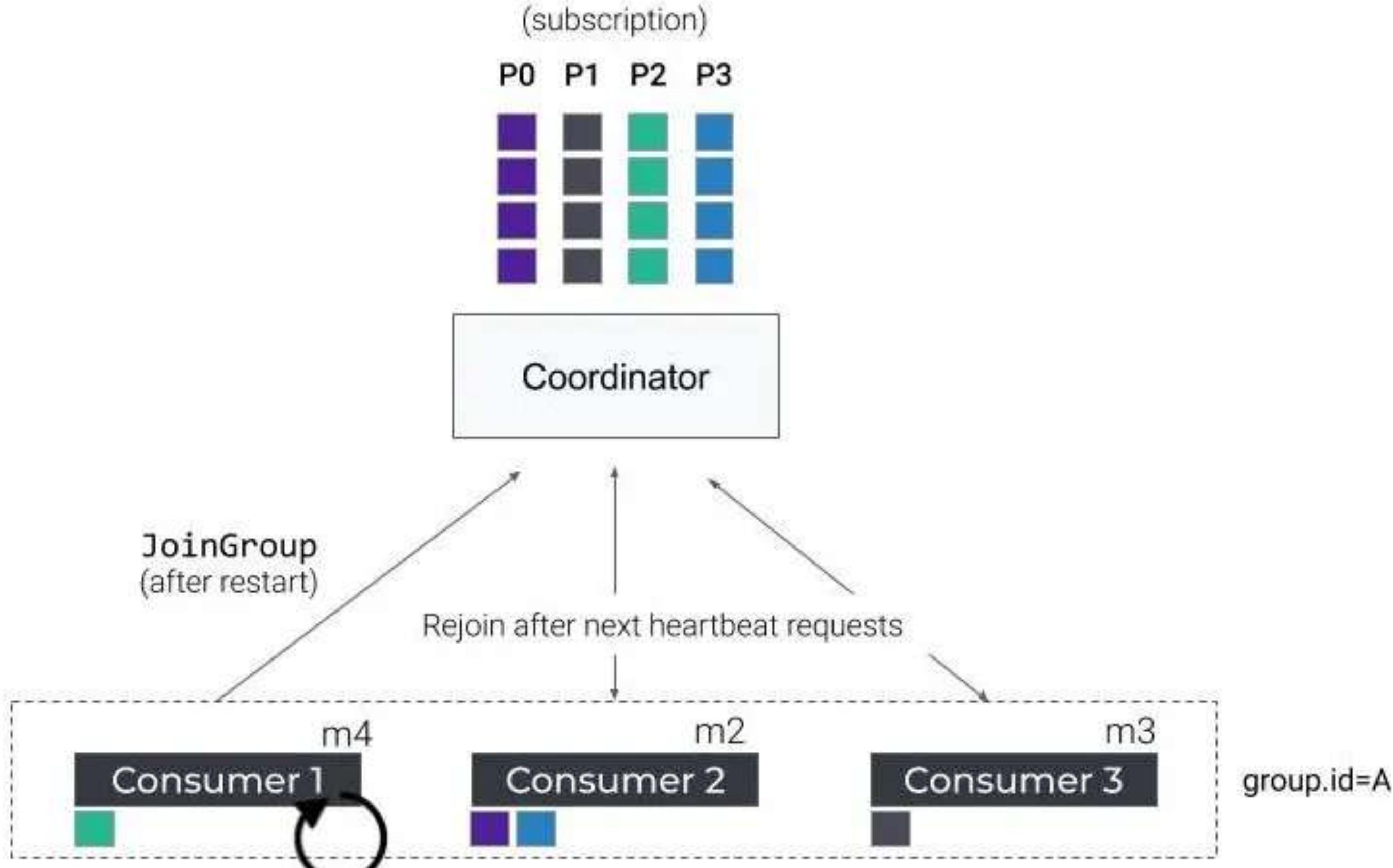
# KAFKA PARTITIONING STRATEGIES

- During the entire rebalancing process, i.e. as long as the partitions are not reassigned, consumers no longer process any data.
- By default, the rebalance timeout is fixed to 5 minutes which can be a very long period during which the increasing consumer-lag can become an issue.

# KAFKA PARTITIONING STRATEGIES

- Let's just assume it's a transient issue.
- Consumer fails->Restarts.
- Will Trigger Re-balance all over again.
- → Consumers will be stopped all over again.

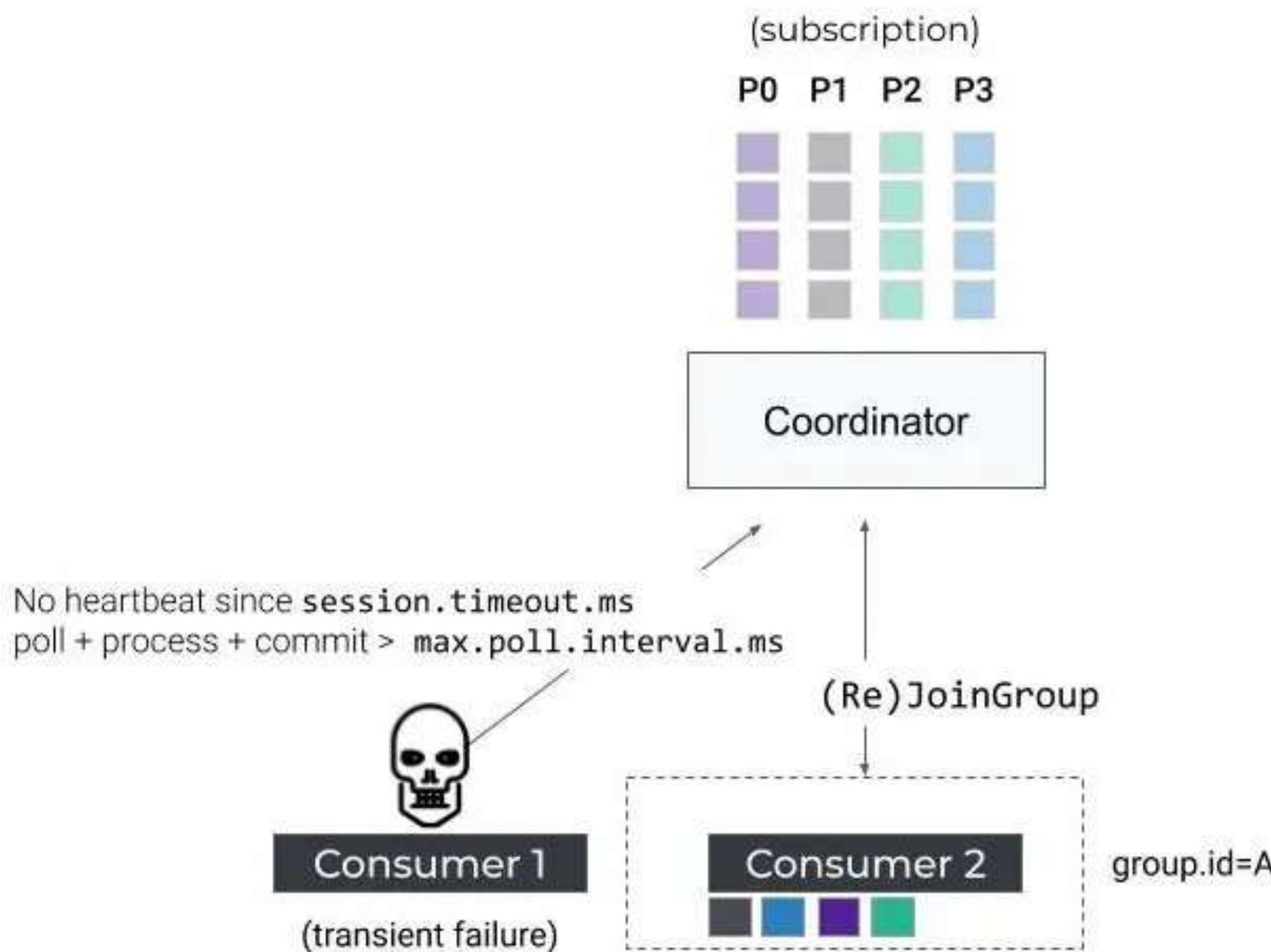
# KAFKA PARTITIONING STRATEGIES



# KAFKA PARTITIONING STRATEGIES

- A consumer group with n members will trigger  $2^*n$  Rebalance Requests during a rolling upgrade.
- Other issues (in Java)
  - Missing Hearbeat requests
  - Long GC Pauses
  - Network Issues
  - Non-Invoking of KafkaConsumer#Poll() due to long processing times.
  - See: `session.timeout.ms` and `max.poll.interval.ms`

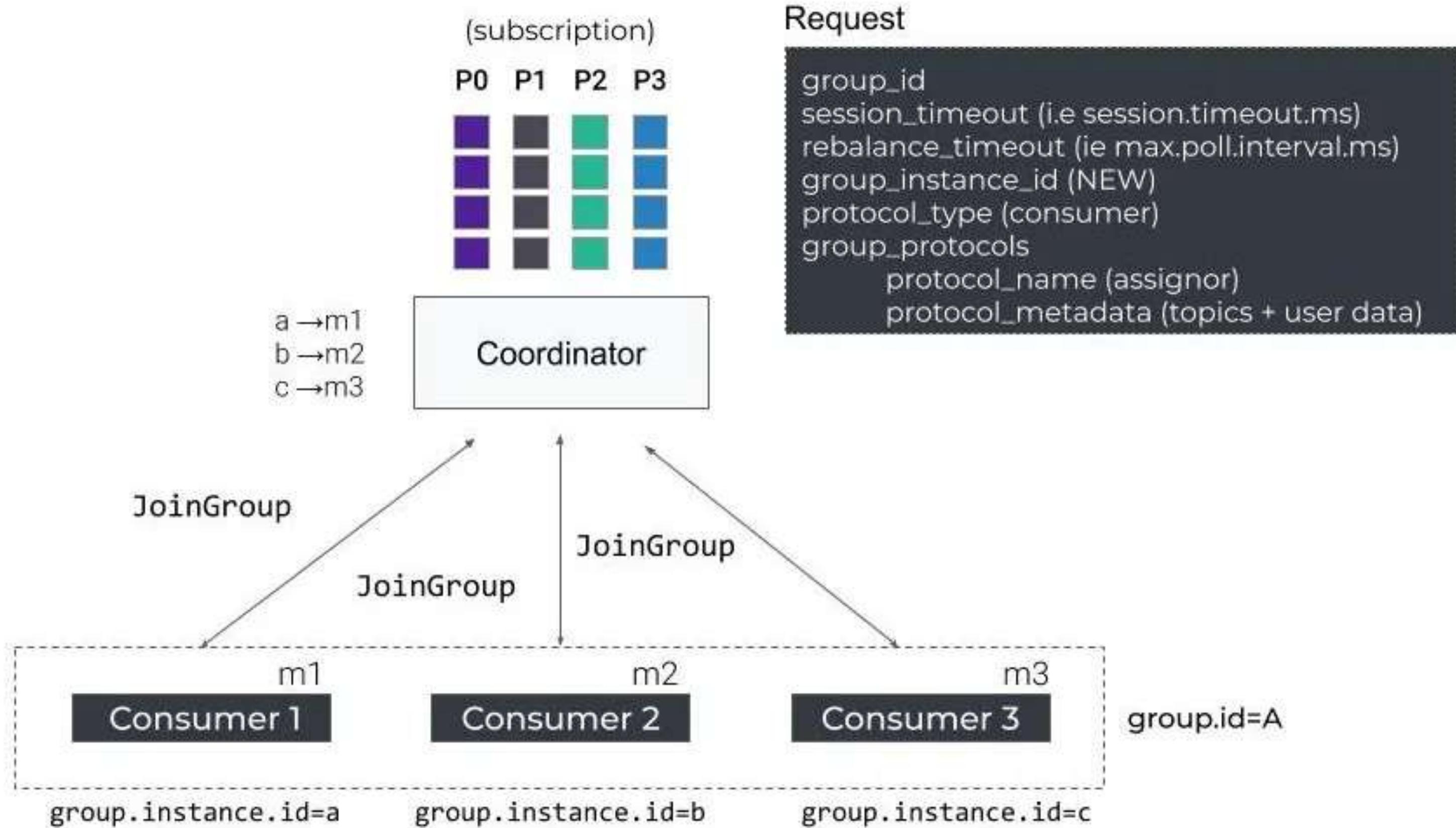
# KAFKA PARTITIONING STRATEGIES



# KAFKA PARTITIONING STRATEGIES

- Consider Static Membership (since Apache Kafka 2.3)
- Unique identifier along with group.instance.id

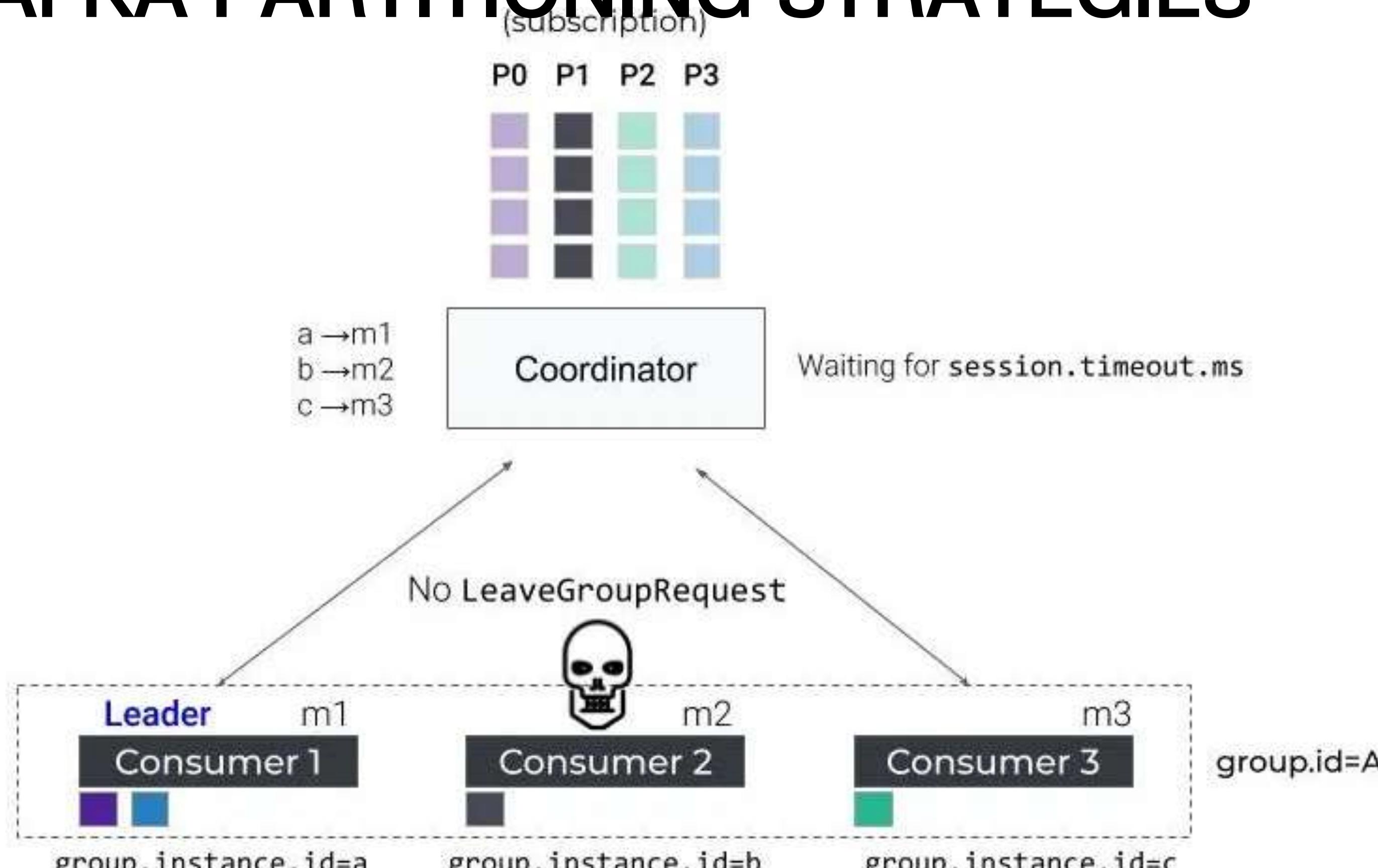
# KAFKA PARTITIONING STRATEGIES



# KAFKA PARTITIONING STRATEGIES

- When a consumer is killed or restarted, others will not be notified for a re-balance until `session.timeout.ms` is reached.
- Notes that Consumers will not send `LeaveGroup` when they are stopped.

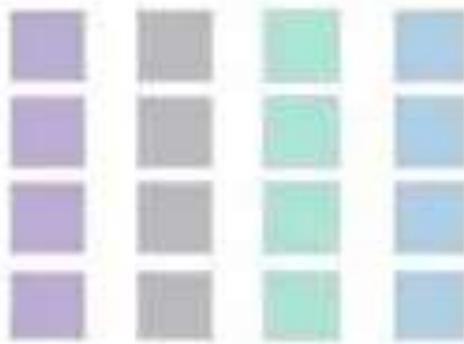
# KAFKA PARTITIONING STRATEGIES



# KAFKA PARTITIONING STRATEGIES

(subscription)

P0 P1 P2 P3



a → m1  
b → m4  
c → m3

Coordinator

no rebalance

Leader

m1

Consumer 1



group.instance.id=a

m4

Consumer 2



group.instance.id=b

m3

Consumer 3



group.instance.id=c

Is restarting

No additional re-balance

Adjust session.timeout.ms

# KAFKA PARTITIONING STRATEGIES

- Can be very useful for limiting the number of undesirable rebalances and thus minimizing stop-the-world effect.
- On the other hand, this has the disadvantage of increasing the unavailability of partitions because the coordinating broker may only detect a failing consumer after a few minutes (depending on `session.timeout.ms`).
- Unfortunately, this is the eternal trade-off between availability and fault-tolerance you have to make in a distributed system.

# KAFKA PARTITIONING STRATEGIES

- Other topics
  - Incremental Cooperative Balancing
  - Kafka Connect Limitations

# KAFKA PARTITIONING STRATEGIES

- Strategies for partitioning = = `partition.assignment.strategy`
- E.g.

```
Properties props = new Properties();
...
props.put(ConsumerConfig.PARTITION_ASSIGNMENT_STRATEGY_CONFIG,
StickyAssignor.class.getName());
```

```
KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
//...
```

# KAFKA PARTITIONING STRATEGIES

- All Consumers in group must have same strategy
- This property accepts a comma-separated list of strategies.
- For example, it allows you to update a group of consumers by specifying a new strategy while temporarily keeping the previous one.
- As part of the Rebalance Protocol the broker coordinator will choose the protocol which is supported by all members.

# KAFKA PARTITIONING STRATEGIES

- 3 Built in

- Range
- Round Robin
- StickAssignor

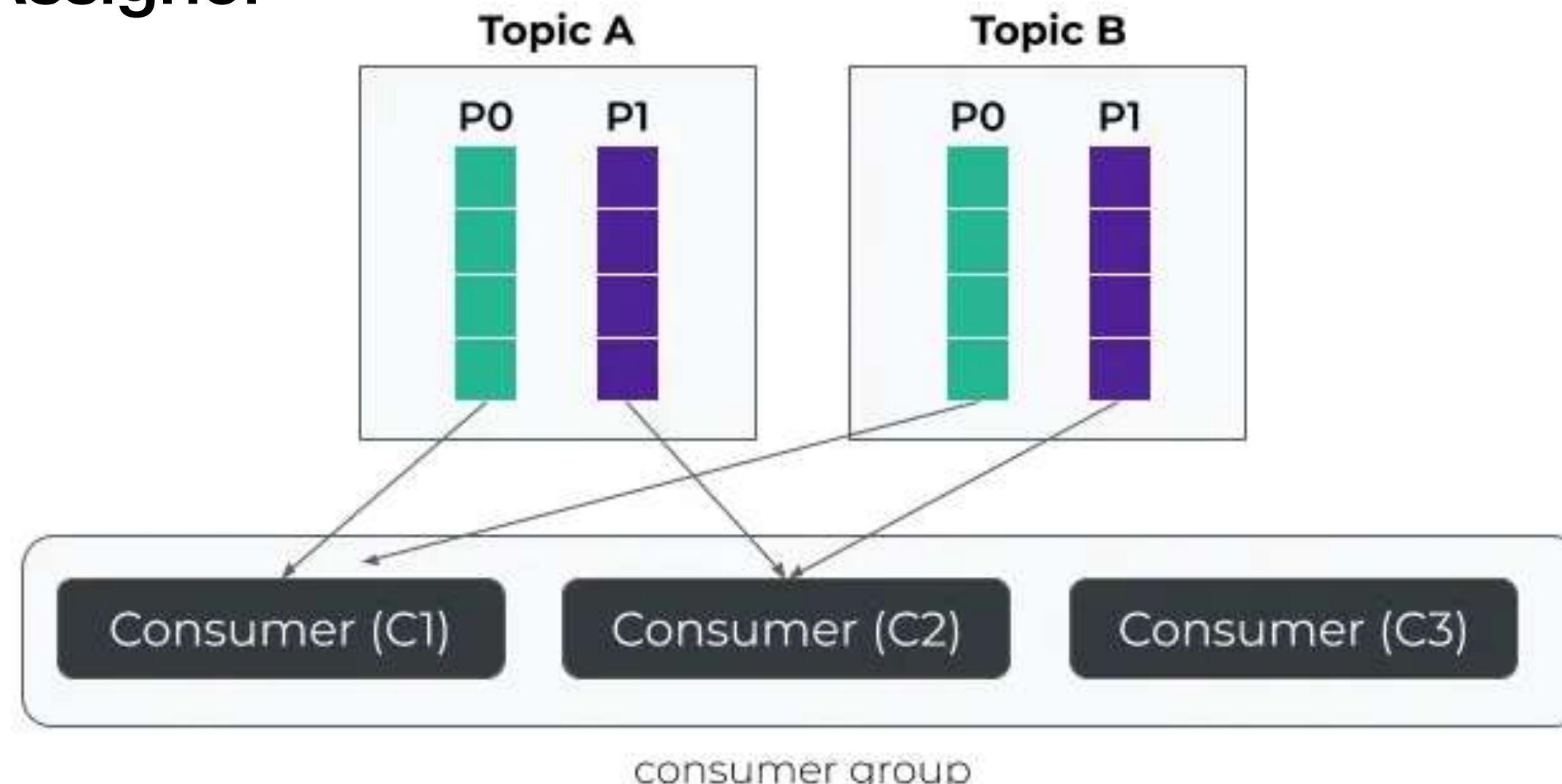
**Default if none is specified.**

# KAFKA PARTITIONING STRATEGIES

- RangeAssignor
- First put all consumers in lexicographic order using the *member\_id* assigned by the broker coordinator.
- Then, it will put available topic-partitions in numeric order.
- Finally, for each topic, the partitions are assigned starting from the first consumer .

# KAFKA PARTITIONING STRATEGIES

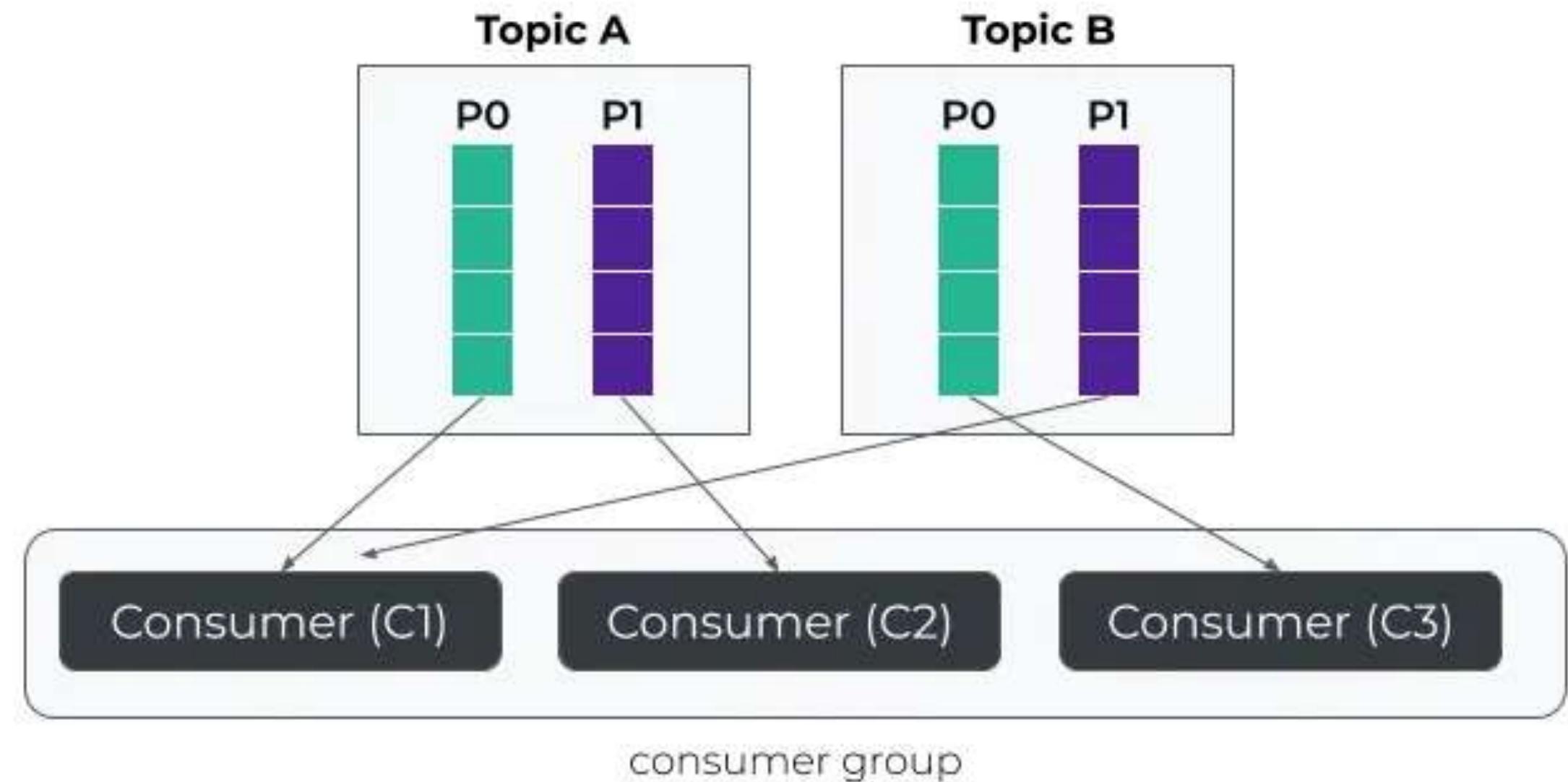
- RangeAssignor



Assignment —> C1 = {A-0, B-0}, C2 = {A-1, B-1}, C3 = {}

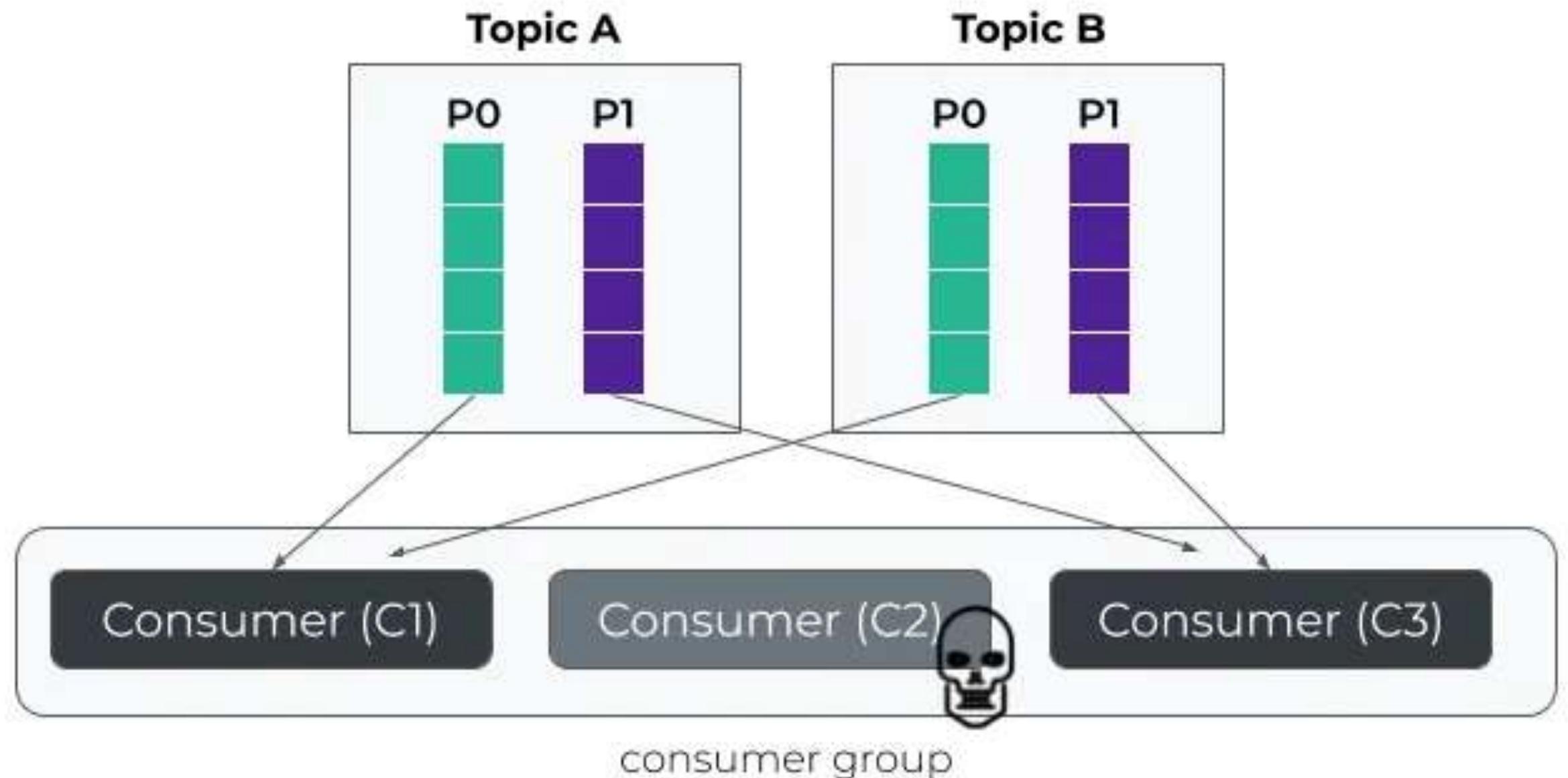
# KAFKA PARTITIONING STRATEGIES

- RoundRobinAssignor
- Distribute evenly across Consumers



# KAFKA PARTITIONING STRATEGIES

- RoundRobinAssignor does not attempt to reduce partition movements when re-balancing occurs.

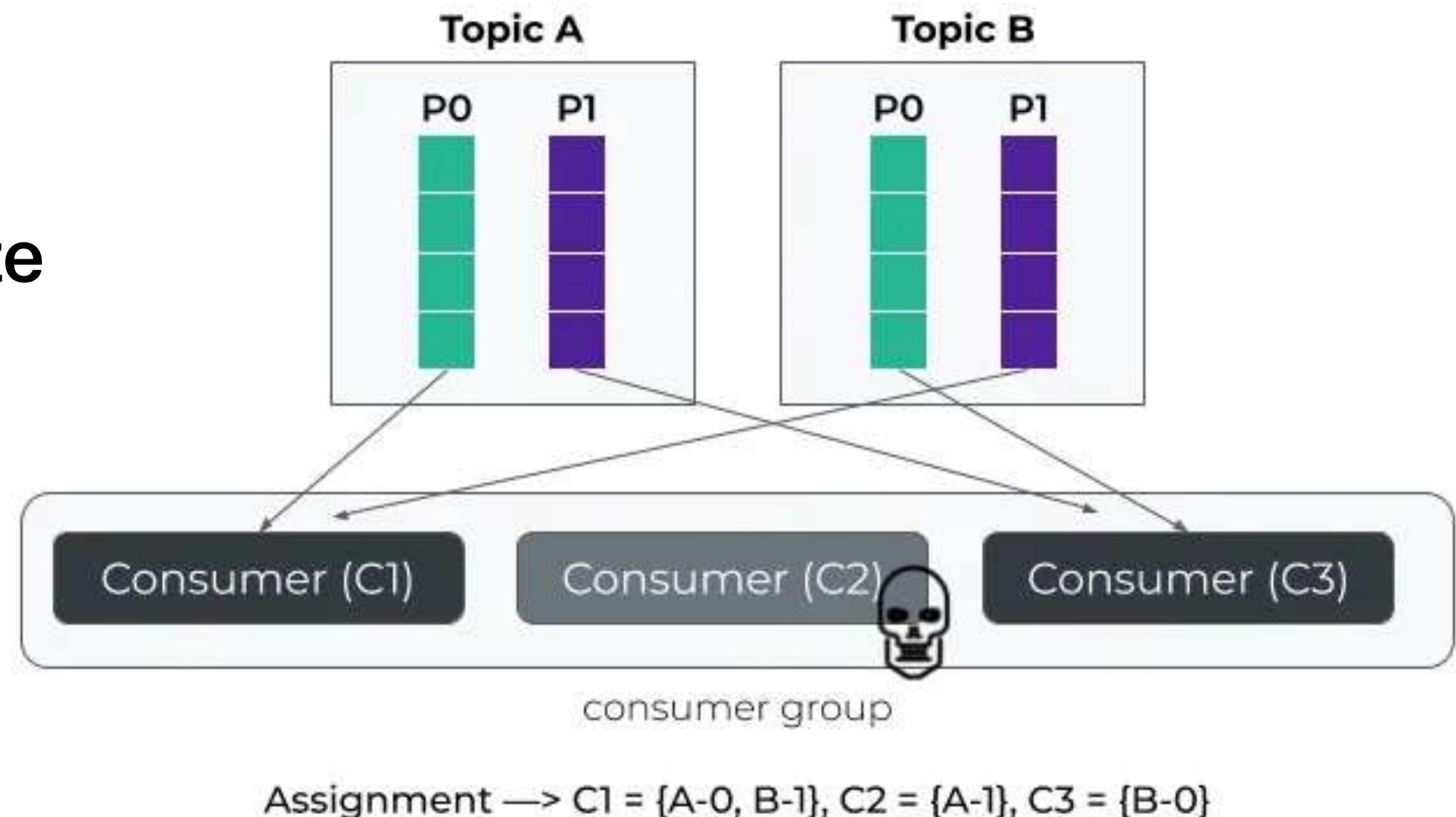


before —> C1 = {A-0, B-1}, C2 = {A-1}, C3 = {B-0}

after —> C1 = {A-0, B-0}, C3 = {A-1, B-1}

# KAFKA PARTITIONING STRATEGIES

- **StickyAssignor**  
Similar to  
RoundRobin but  
will try to minimize  
partition  
movement.



# KAFKA PARTITIONING STRATEGIES

- Others not being covered here
- **StreamsPartitionAssignor**
- Custom Assignment Strategies

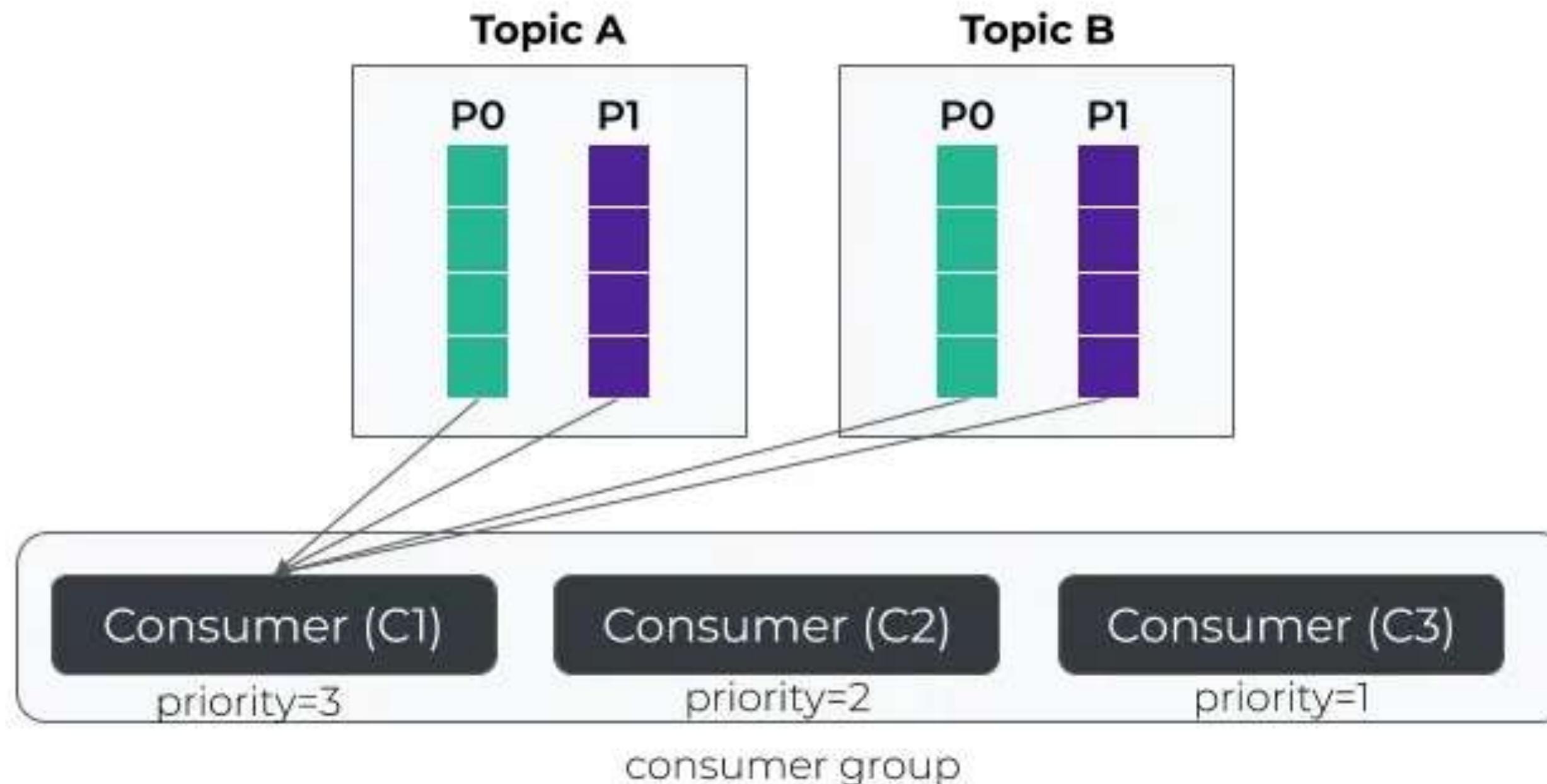
# KAFKA PARTITIONING STRATEGIES

- All Consumers act as if in Active/Active Mode.
- For some production scenarios, it might be useful to have an Active/Passive mode.
- Similar to a FailoverAssignor.

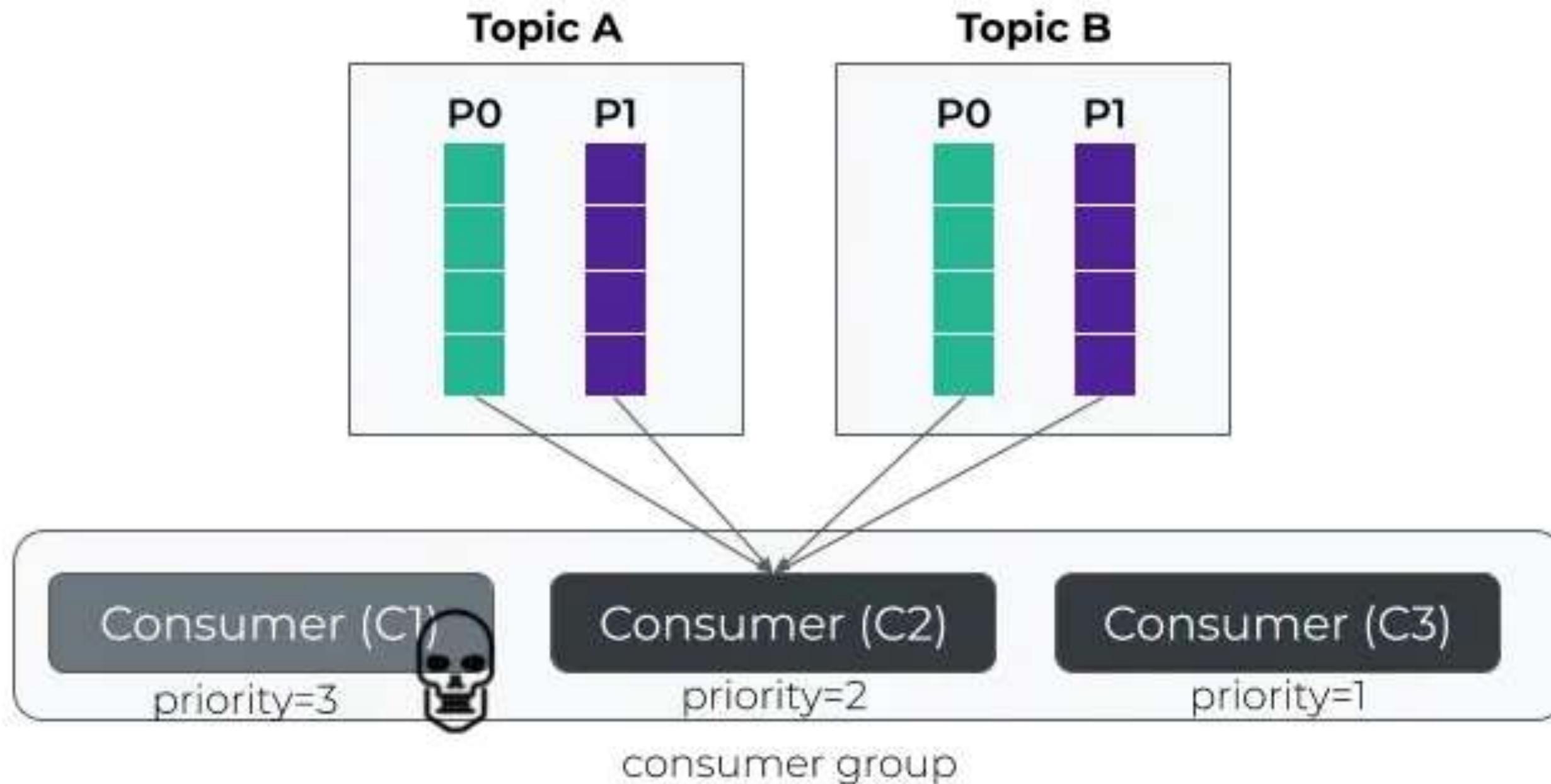
# KAFKA PARTITIONING STRATEGIES

- Multiple consumers can join a same group.
- However, all partitions are assigned to a single consumer at a time.
- If that consumer fails or is stopped then partitions are all assigned to the next available consumer.
- Usually, partitions are assigned to the first consumer but for our example we will attach a priority to each of our instance. Thus, the instance with the highest priority will be preferred over others.

# KAFKA PARTITIONING STRATEGIES



# KAFKA PARTITIONING STRATEGIES

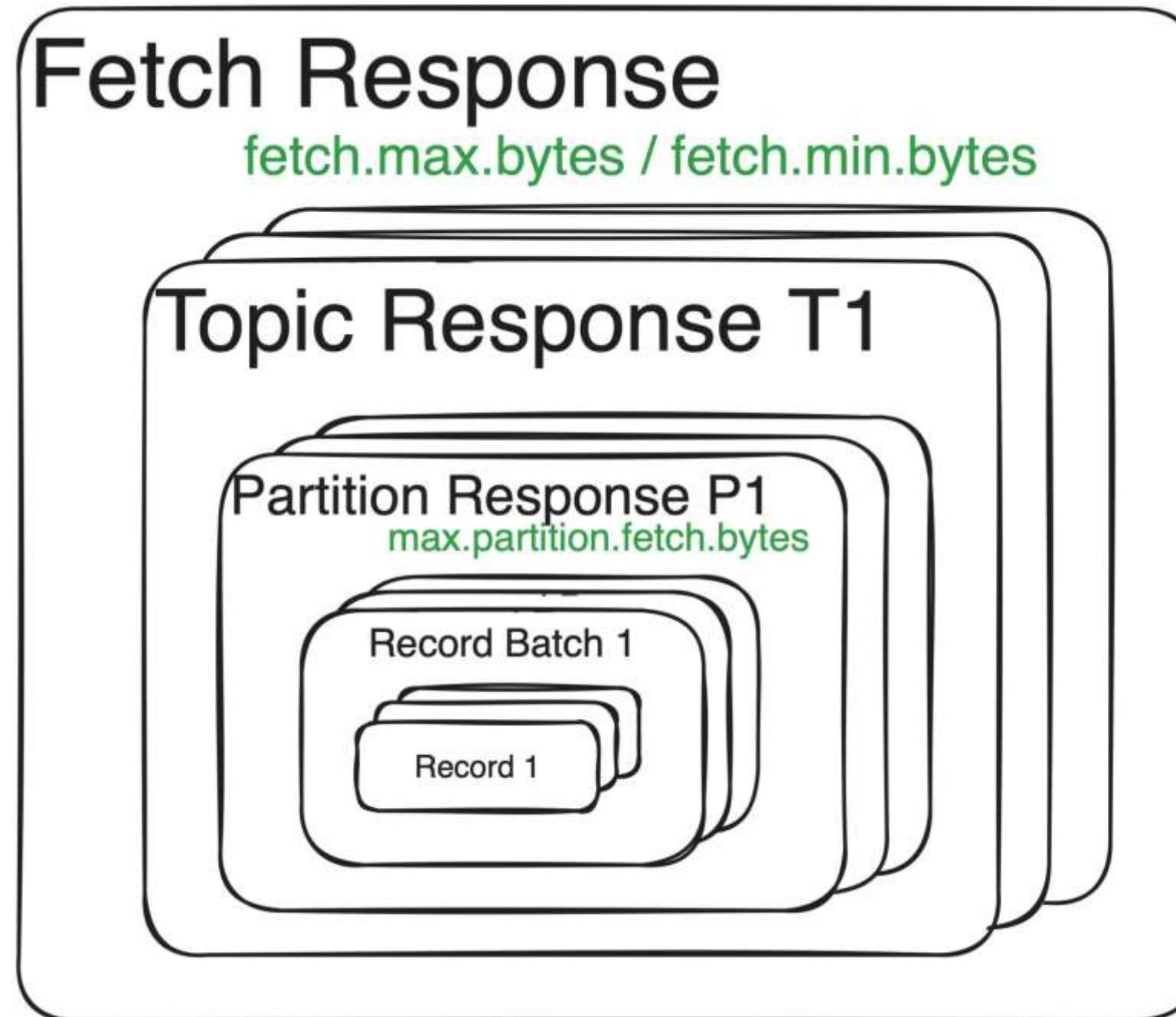


# KAFKA PARTITIONING STRATEGIES

- Instead of implementing PartitionAssignor interface, extend the abstract class AbstractPartitionAssignor.
- assign() method is already implemented here 😊
- Left as an exercise to the class 😊

# Some Notes on Optimization

# Customer Optimization by Fetch Size



# Customer Optimization by Fetch Size

- Kafka Consumers send fetch requests to Kafka Brokers, asking for records from one or more topics and one or more partitions within each topic.
- Each fetch response can contain a **batch of record batches** for each topic and partition tuple that was in the request.
- Kafka Consumers may send multiple fetches in parallel but only one fetch to each broker that has data it's subscribed or assigned to.

# Customer Optimization by Fetch Size

- fetch.max.bytes
- max.partition.fetch.bytes
- fetch.min.bytes
- fetch.max.wait.ms

# Customer Optimization by Fetch Size

- An application may underutilize CPU resources if throughput is limited by the Kafka Consumer.
- An application may scale out faster than necessary if throughput is limited by the Kafka Consumer.
- An application may put excess load on the Kafka Brokers, and waste Kafka Broker CPU, if it polls for data too frequently.

# Customer Optimization by Fetch Size

- The metrics to monitor include
  - Average Fetch Latency
  - Max Fetch Latency
  - Average Fetch request Rate
  - Average Fetch Size
  - Max Fetch Size
- <<ADD HERE>>

# Consumer Tuning

- **Fetch size**
- The fetch size directly impacts the number of messages a consumer fetches from the broker in a single request. Configuring fetch size based on the expected message size optimizes throughput.  
**fetch.min.bytes**
- This configuration defines the minimum amount of data, in bytes, that the broker should return for a fetch request.
- Increasing this value leads to fewer fetch requests, reducing the overhead of network communication and I/O operations.
- However, it may also increase latency as the consumer waits for enough messages to accumulate before making a fetch request.
- You should experiment with different fetch sizes to find the ideal balance between throughput and latency for your specific use case.

# Consumer Tuning

- **Max poll records**
- Controlling the maximum number of records fetched in a single poll request helps balance the processing time and consumer lag.  
**max.poll.records**
- This configuration defines the maximum number of records a consumer fetches in a single poll.
- By adjusting this value, you control the trade-off between the time spent processing records in the application and the potential for consumer lag.
- A smaller value leads to more frequent polls and lower consumer lag but also increases the overhead of processing records.
- Conversely, a larger value improves throughput but results in higher consumer lag if the consumer cannot process records fast enough.
- You should set this value based on your application's processing capabilities/resources and tolerance for consumer lag.

# Consumer Tuning

- **Client-side buffering**
- You use buffering to reduce the impact of network latency on consumer processing.

## **fetch.max.bytes**

- This configuration defines the maximum amount of data, in bytes, that the consumer buffers from the broker before processing it.
- Increasing this value helps the consumer absorb temporary spikes in network latency, but it may also expand the memory footprint of the consumer.
- You should choose a buffer size that balances the trade-off between network latency and memory usage for your specific use case.

# Consumer Tuning

- **Consumer group rebalancing**
  - Configuring session and heartbeat timeouts optimizes group rebalancing, ensuring consumer groups maintain a stable membership and quickly detect failed consumers.
- session.timeout.ms**
- This configuration defines the maximum amount of time, in milliseconds, that a consumer can be idle without sending a heartbeat to the group coordinator.
  - If no heartbeat is received within this time, the consumer is considered failed, and the group triggers a rebalance.
  - Rebalancing in Kafka can impact performance, as it may cause temporary disruption in message processing while consumer groups reassign partitions to ensure even distribution of workload across consumers.
  - During this process, the overall throughput and latency may be affected, making it essential to monitor and manage rebalancing events carefully. You should set this value based on your application's processing capabilities and the desired frequency of rebalances.

**heartbeat.interval.ms**

- This configuration defines the interval, in milliseconds, at which the consumer sends heartbeats to the group coordinator.
- A shorter interval helps the group coordinator detect failed consumers more quickly but also increases the coordinator's load. You should choose a heartbeat interval that balances the trade-off between failure detection and coordinator load for your specific use case.

# Some Notes on Metrics

- If the average fetch size (fetch-size-avg) metric is consistently around 1 MiB \* number of partitions per fetch request, the consumer is likely hitting `the max.partition.fetch.bytes limit`.
- The default value for `max.partition.fetch.bytes` is 1 MiB, which means the broker will send you a response with at most 1 MiB per partition even if it has more data for the partition(s) you're asking for.
- The result of this is you will end up issuing significantly more requests and your throughput could be bound by the round trip time to your broker, because Kafka Consumers only allow one fetch in-flight (or buffered) per broker.

# Some Notes on Metrics

- A reasonable fetch rate is somewhere around 1 to 5 requests per second per broker because it corresponds to 200 to 1,000 milliseconds of fetch latency.
- If your average fetch size is consistently hitting a limit and your average fetch request rate is higher than five requests per second per broker, you're likely creating many small fetch requests instead of fewer large ones.
- This not only limits the throughput of your application, it can very quickly consume a large amount of your Kafka Broker capacity due to the high volume request rate which directly translates to significant costs.

# Some Notes on Metrics

- $\text{max.partition.fetch.size} = 50\text{MiB} / \text{partitions per fetch}$
- How is partitions per fetch calculated?
- average partitions per fetch =  $\min(\text{ceil}(\text{assigned-partitions} / \text{broker count}), \text{assigned-partitions})$

# Some Notes on Metrics

- Kafka Metrics are exposed via JMX
- Note that Consumer Lag Metrics are only available via the Kafka Lag Exporter.
- Also note that Kafka Exporter is not same as Kafka Lag Exporter and different from the standard JMX exporter.

# Some Notes on Metrics

- First get all base metrics you are interested In before you start playing around with the parameters usually

# Some Notes on Commit Mechanisms

# Offset Commits

- **Concerns**
  - Which messages should be read?
  - Duplicate Message Reading and Processing
  - Message Loss
- **Kafka keeps track of the messages that consumers read.**
- **This is done by offsets**
- Offsets are integers starting from zero that increment by one as the message gets stored.

# Offset Commits

- 4 ways to commit offsets
- Method #1: Auto-Commit (Default after every 5 seconds)
- Commits the largest offset returned by poll()

```
KafkaConsumer<Long, String> consumer = new  
KafkaConsumer<>(props);consumer.subscribe(KafkaConfigProperties.getTopic());  
ConsumerRecords<Long, String> messages =  
consumer.poll(Duration.ofSeconds(10));  
for (ConsumerRecord<Long, String> message : messages) { // processed message}
```

# Offset Commits

- Assume 100 records came in.
- 60 have been processed.
- Consumer crashes/stop/fails
- It comes back live – records 61 to 100 are lost

# Offset Commits

- **Method #2: Manual Sync Commit**
- First off, turn auto Commit to be false

```
Properties props = new Properties();props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, "false");
```

```
KafkaConsumer<Long, String> consumer = new KafkaConsumer<>(props);consumer.subscribe(KafkaConfigProperties.getTopic());ConsumerRecords<Long, String> messages = consumer.poll(Duration.ofSeconds(10));//process the messagesconsumer.commitSync();
```

# Offset Commits

- Commits only after processing the messages.
- Consumer can still crash before commit is called => Duplicate Processing
- Impacts consumer performance badly.
- Blocking code and will retry => Low throughput.

# Offset Commits

- **Method #3: Manual Async Commit**

```
KafkaConsumer<Long, String> consumer = new  
KafkaConsumer<>(props);consumer.subscribe(KafkaConfigProperties.get  
Topic());  
ConsumerRecords<Long, String> messages =  
consumer.poll(Duration.ofSeconds(10));  
//process the messages  
consumer.commitAsync();  
// We do assume that Auto Commit has been disabled
```

# Offset Commits

- Suppose 300 is the largest offset, but *commitAsync()* fails due to some issue.
- It could be possible that before it retries, another call of *commitAsync()* commits the largest offset of 400 as it is asynchronous.
- When failed *commitAsync()* retries and if it commits offsets 300 successfully, it will overwrite the previous commit of 400, resulting in duplicate reading.
- That is why *commitAsync()* doesn't retry

# Offset Commits

- **Method #4: Commit Specific Offset**
- Processing the messages in small batches
- Commit the offsets as soon as messages are processed.

# Offset Commits

```
KafkaConsumer<Long, String> consumer = new KafkaConsumer<>(props);
consumer.subscribe(KafkaConfigProperties.getTopic());

Map<TopicPartition, OffsetAndMetadata> currentOffsets = new HashMap<>();

int messageProcessed = 0;

// So now we have a Hashmap
```

# Offset Commits

```
while (true)
{
    ConsumerRecords<Long, String> messages =
    consumer.poll(Duration.ofSeconds(10));
    // our logic will go here...
}
```

# Offset Commits

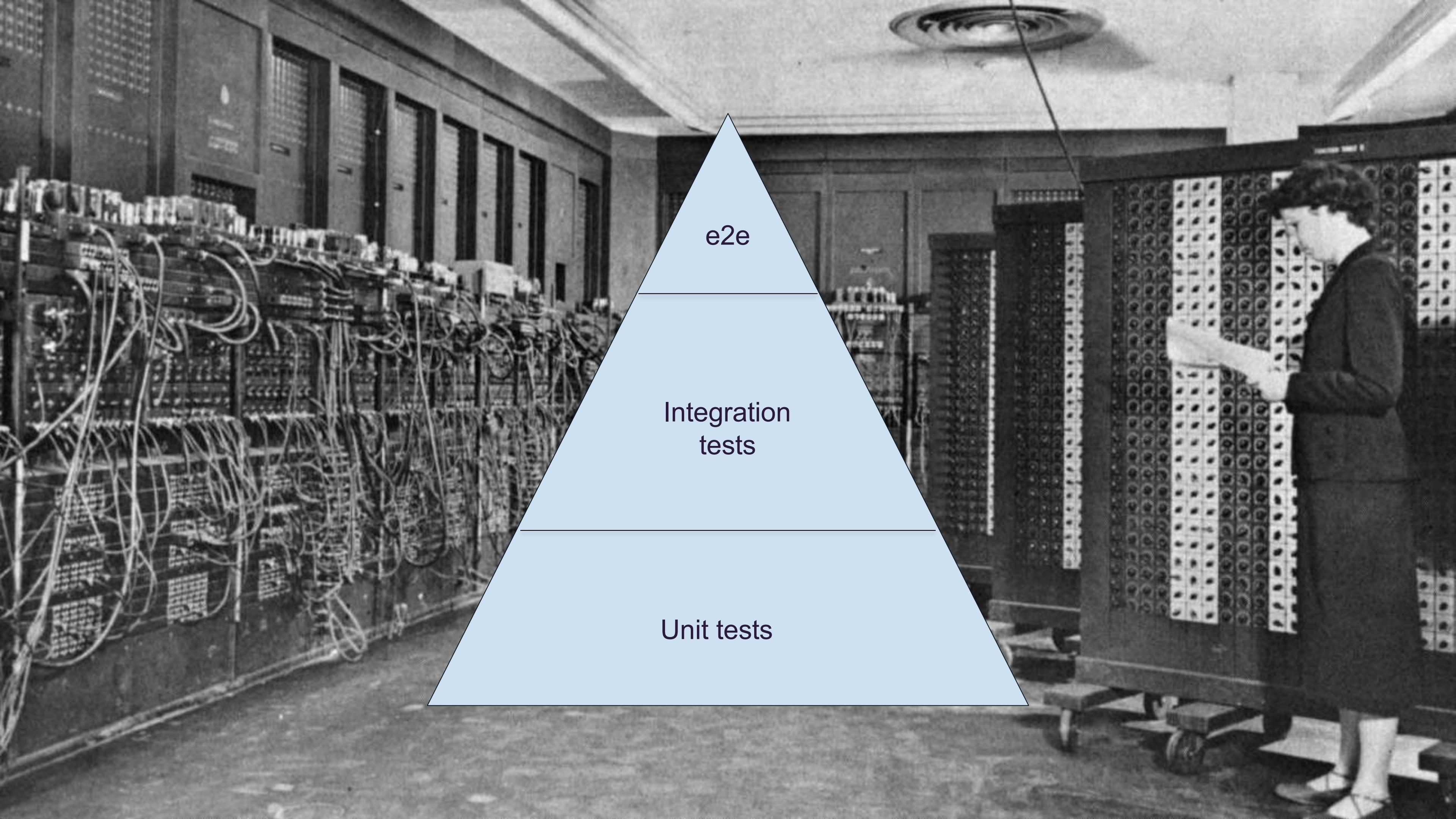
Our logic -→

```
for (ConsumerRecord<Long, String> message : messages)
{
    // processed one message
    messageProcessed++;
}
```

```
currentOffsets.put(
    new TopicPartition(message.topic(), message.partition()),
    new OffsetAndMetadata(message.offset() + 1));
```

```
if (messageProcessed%50==0)
{
    consumer.commitSync(currentOffsets);
}
```

# Getting Started with Testing

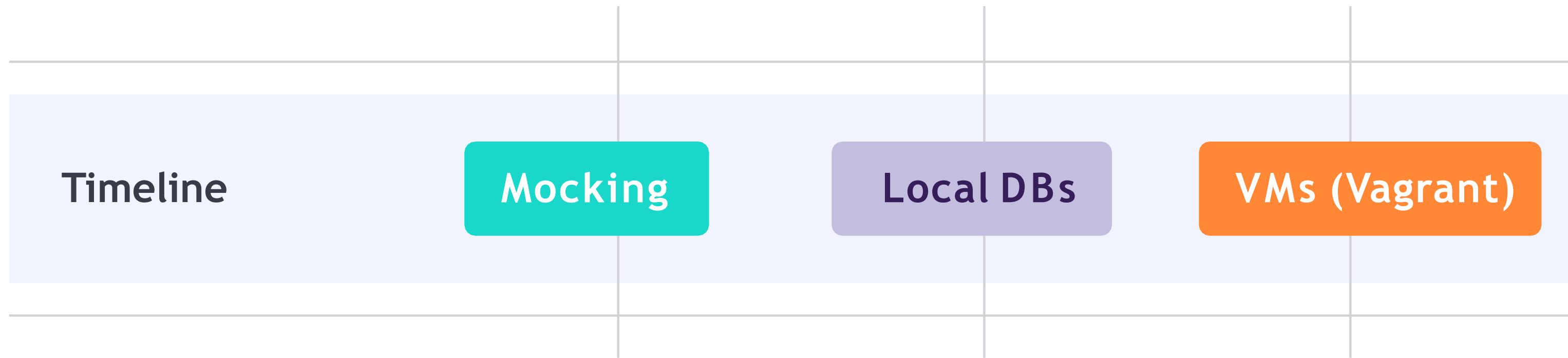


e2e

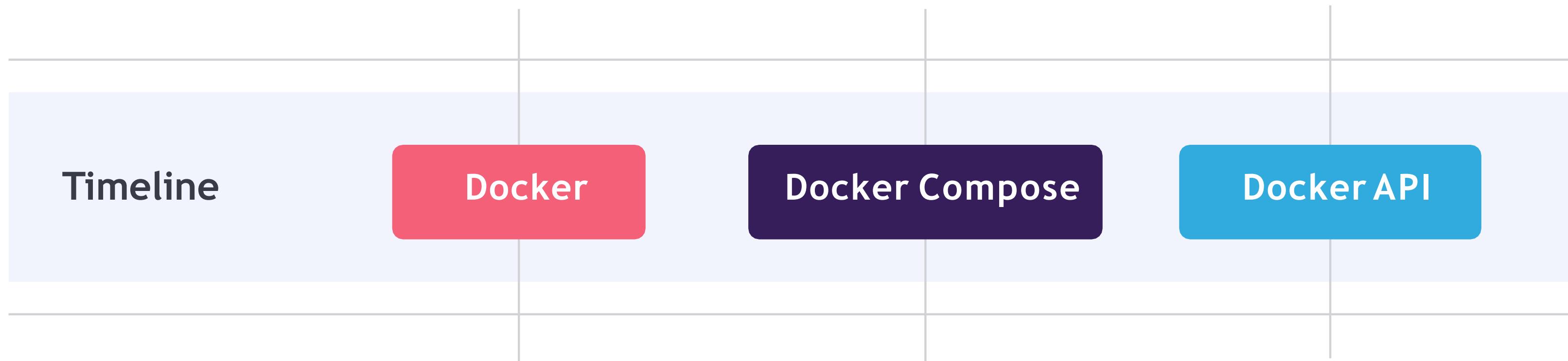
Integration  
tests

Unit tests

# Integration Testing transformation over the years (i)



# Integration Testing transformation over the years (ii)

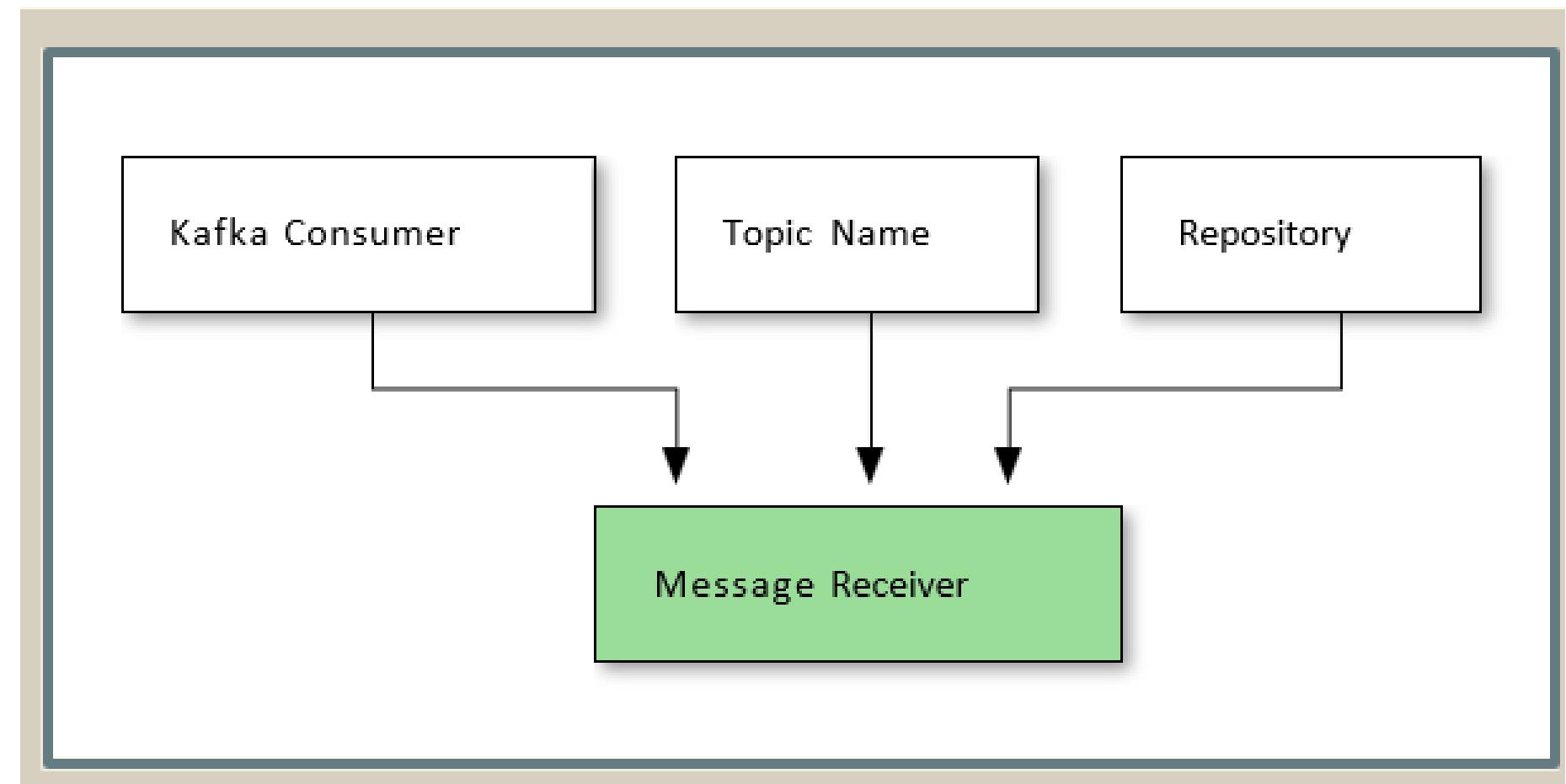


# Getting Started with Testing

```
public class Main {  
    public static void main(String[] args){  
  
        var consumer = new KafkaConsumer<String, String>(props); 1  
        var topic = "my-topic";  
        var repository = new Repository("jdbc:...");  
  
        var messageReceiver =  
            new MessageReceiver(consumer, topic, repository); 2  
        messageReceiver.run(); 3  
    }  
}
```

# Getting Started with Testing

- Constructor Injection



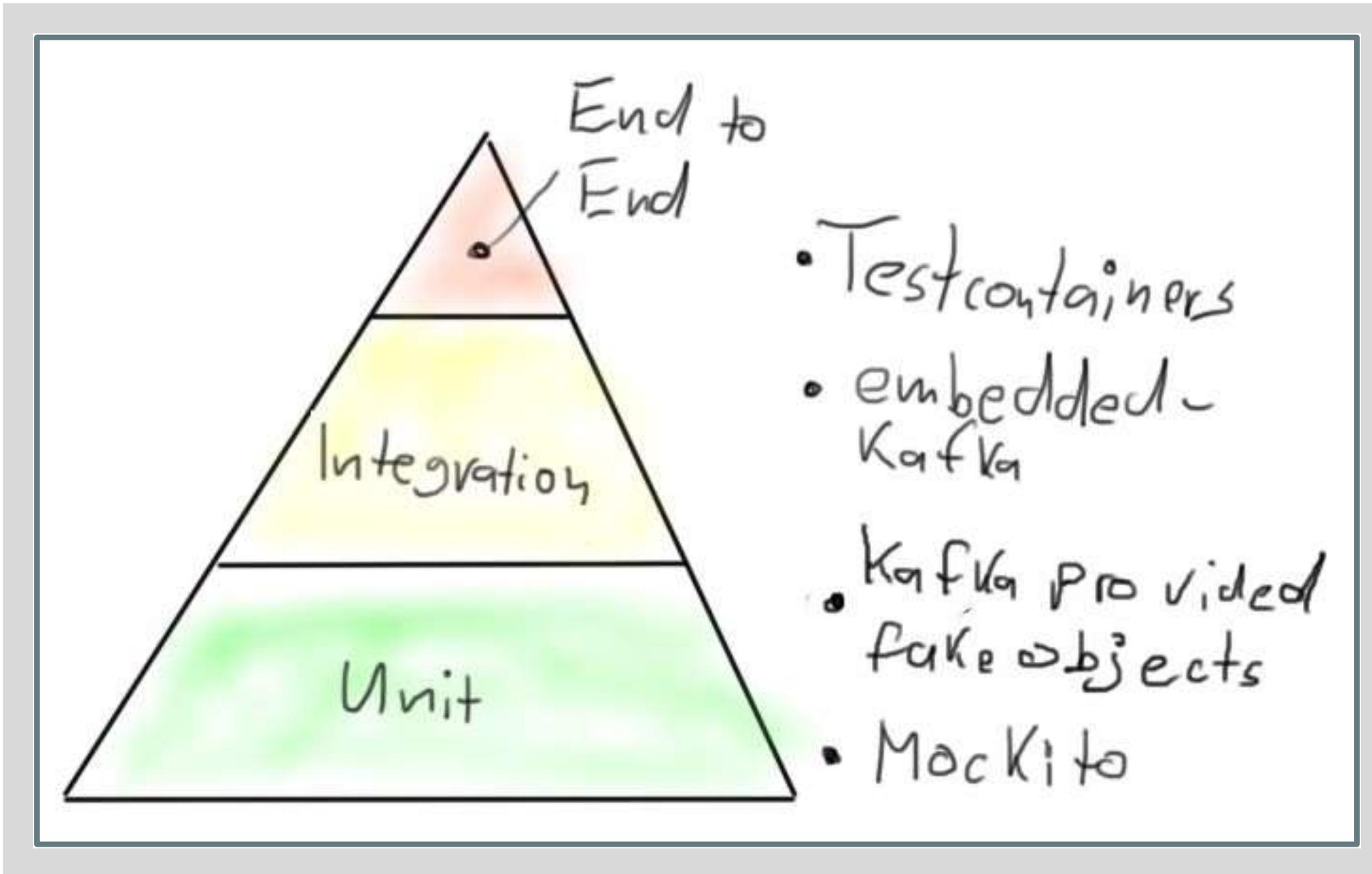
- Everything can be changed
- Makes testing Easier

# Getting Started with Testing

- Now it is easy to test like so

```
@Mock Repository mockRepository;  
@Mock KafkaConsumer<String, String> mockConsumer;  
  
@Test  
public void writesReceivedMessage() {  
    // arrange  
    var records = buildConsumerRecords("my-topic", "HELLO")  
    when(mockConsumer.poll(any())).thenAnswer(notUsed-> records);  
    // act (sut = System Under Test)  
    var sut = new MessageReceiver(mockConsumer, mockRepository, "my-topic")  
    // we don't call run() directly  
    sut.processRecords();  
    // assert  
    verify(mockRepository, times(1)).saveMessage("HELLO WORLD");  
}
```

# Getting Started with Testing

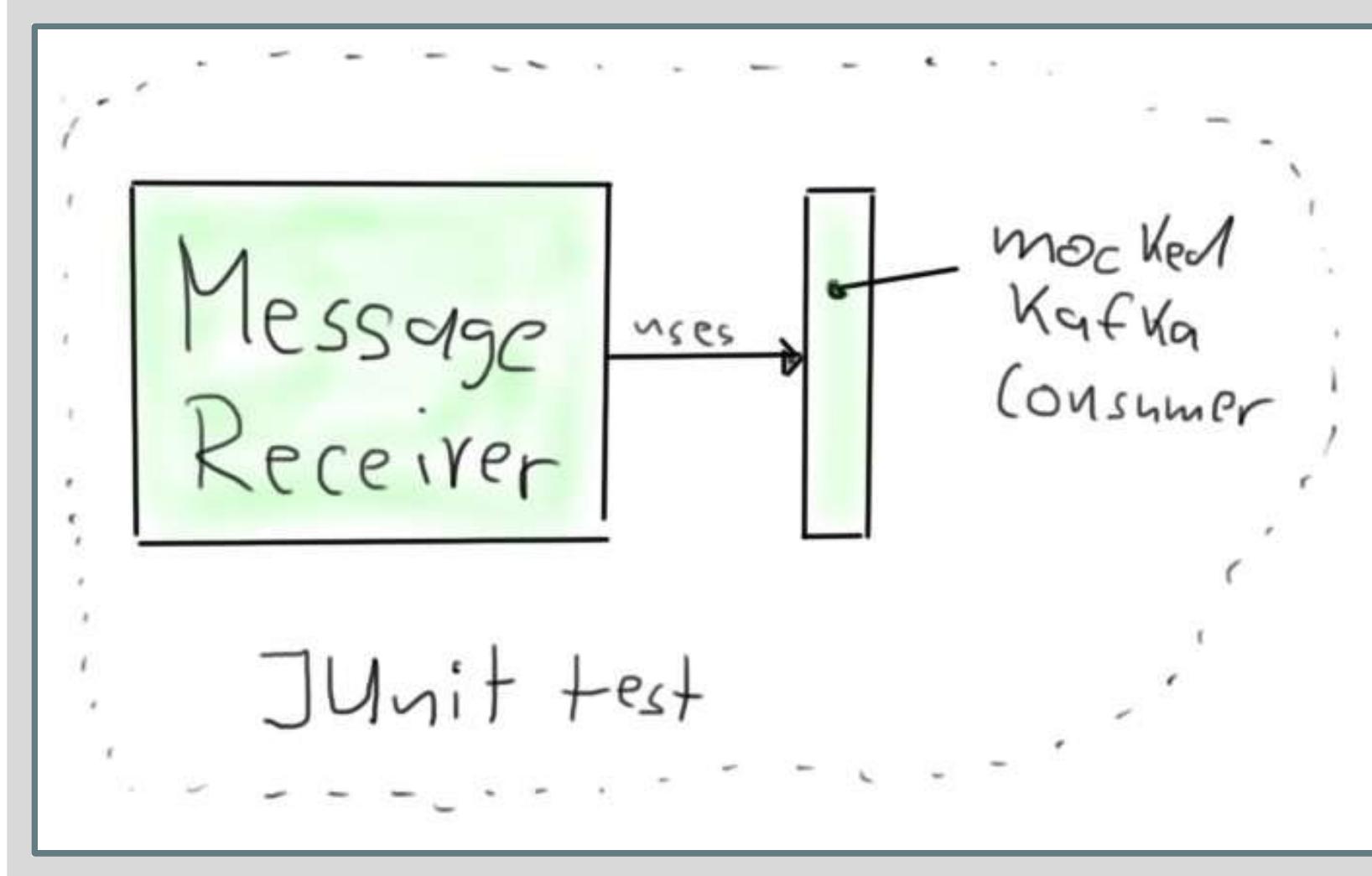


# Getting Started with Testing

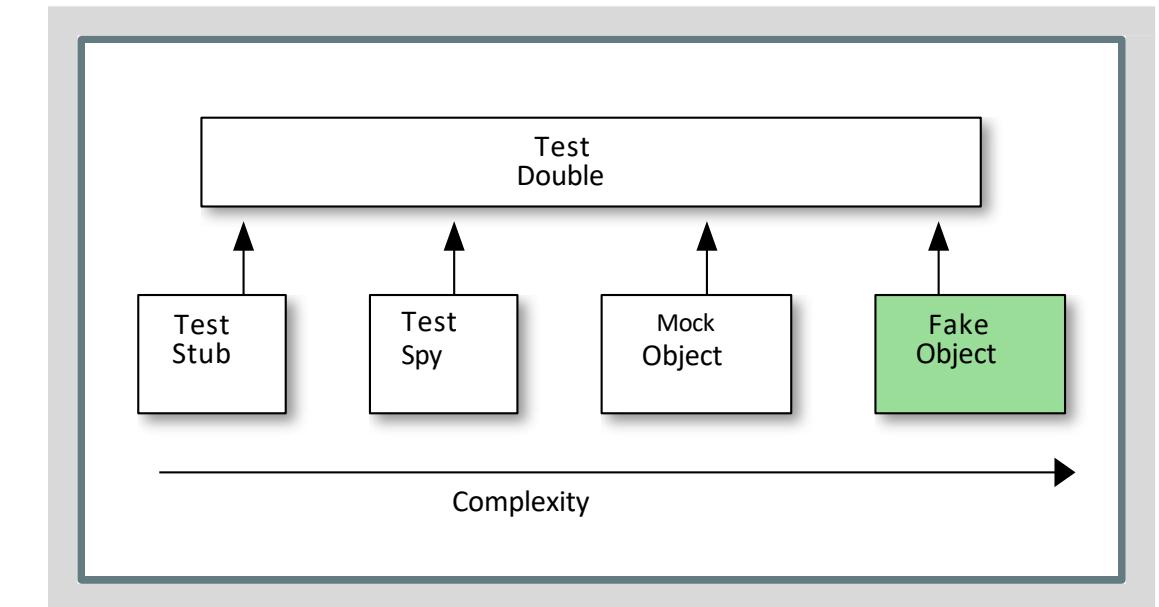


- defacto standard for *mocking* in Java

# Getting Started with Testing

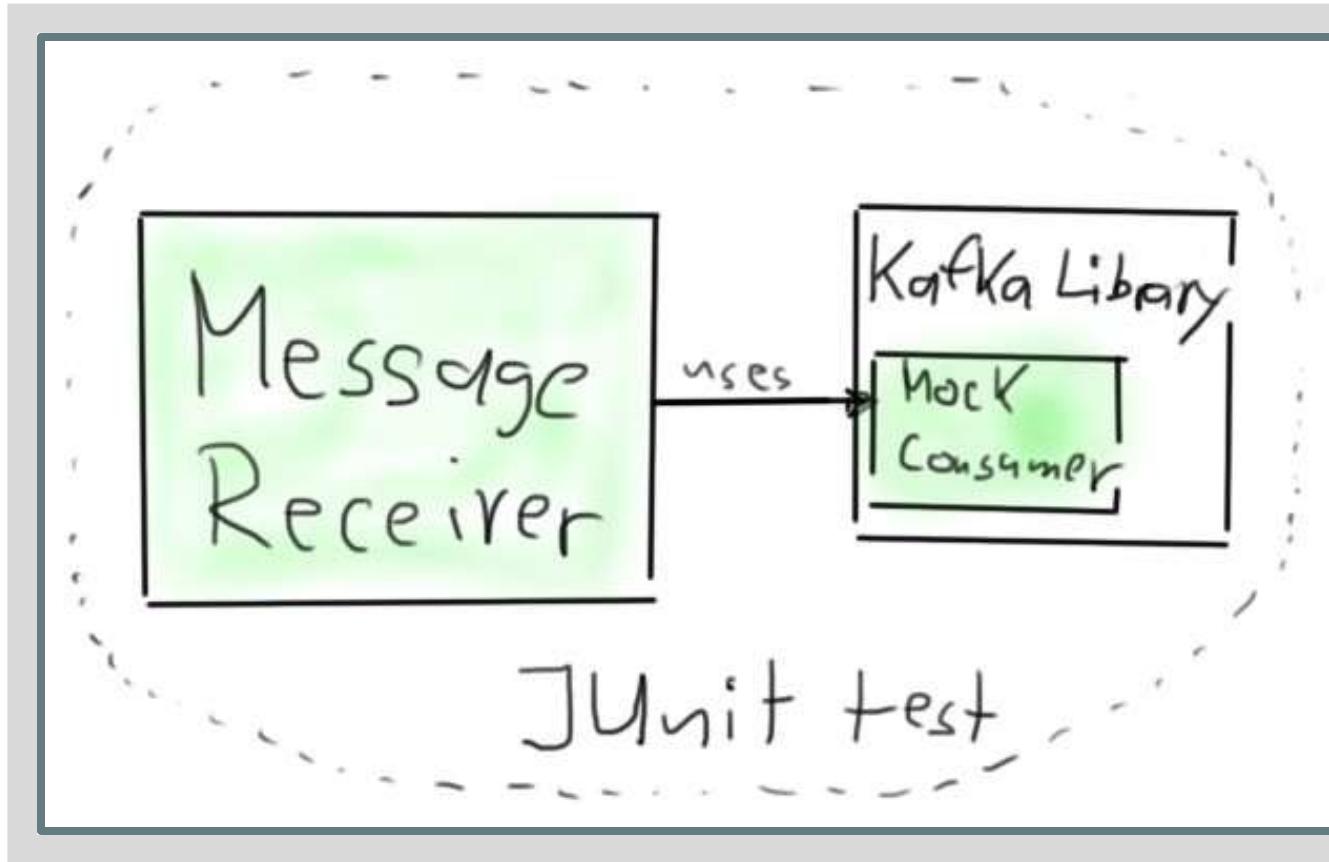


**Mockito**



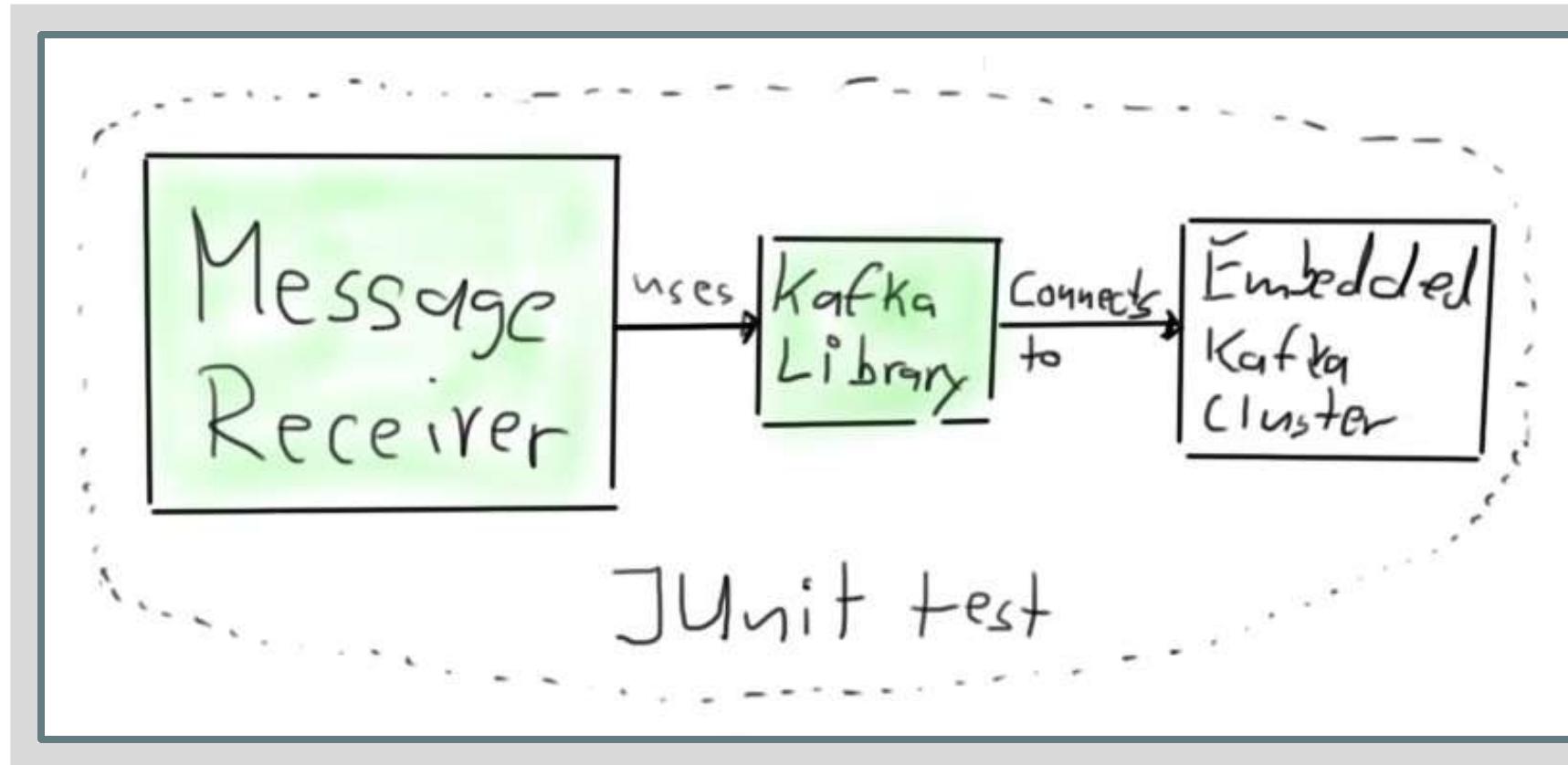
**Fake Objects**

# Getting Started with Testing



All Code for testing MockConsumer and MockProducer as fake objects is given in the provided github repository.

# Getting Started with Testing



All Code for testing MockConsumer and MockProducer as fake objects is given in the provided github repository.

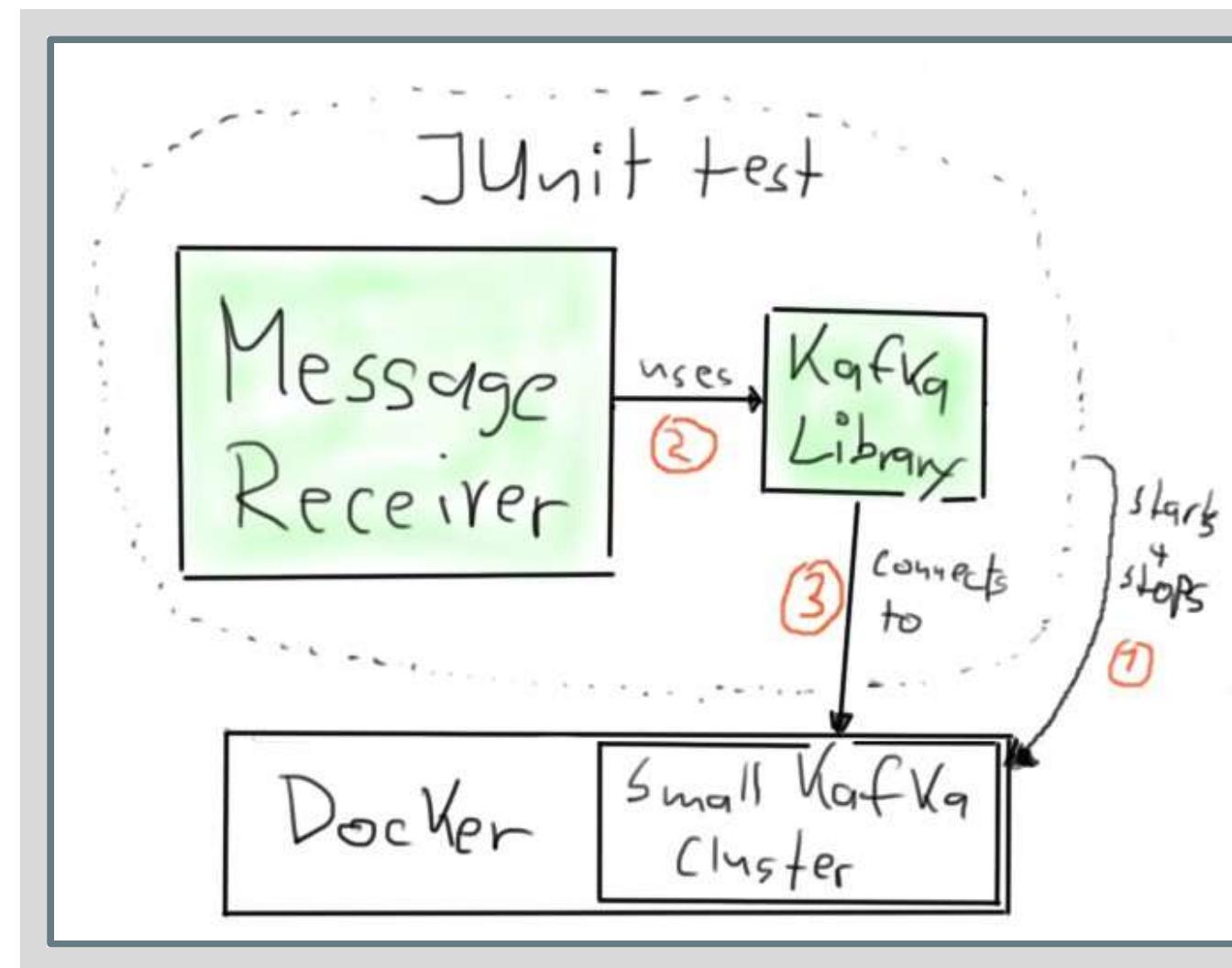
# Getting Started with Testing

- **TestContainers** – A Java Library that supports Junit test and provides throwaway instances of anything else that can run in Docker



All Code for testing MockConsumer and MockProducer as fake objects is given in the provided github repository.

# Getting Started with Testing



All Code for testing MockConsumer and MockProducer as fake objects is given in the provided github repository.

# Getting Started with Testing

- Stress Testing
- kafka-producer-perf-test --topic TOPIC -  
-record-size SIZE\_IN\_BYTES

# Getting Started with Testing

```
$ bin/kafka-producer-perf-test --topic testtopic --record-size  
1000 --num-records 10000 --throughput 1000 --producer-props  
bootstrap.servers=localhost:9092
```

```
5003 records sent, 1000.4 records/sec (0.95 MB/sec),  
1.6 ms avg latency,  
182.0 ms max latency.
```

```
10000 records sent, 998.801438 records/sec (0.95  
MB/sec), 1.12 ms avg  
latency, 182.00 ms max latency, 1 ms 50th, 2 ms  
95th, 19ms 99th, 23 ms 99.9th.
```

# Getting Started with Testing

- Stress Testing
- kafka-consumer-perf-test      --broker-list  
host1:port1,host2:port2      --topic TOPIC

# Load Testing

- JMETER + KLOADGEN
- Basic Approach
  - Create Topics
  - Provide Test Data and let KLOADGEN generate synthetic test data

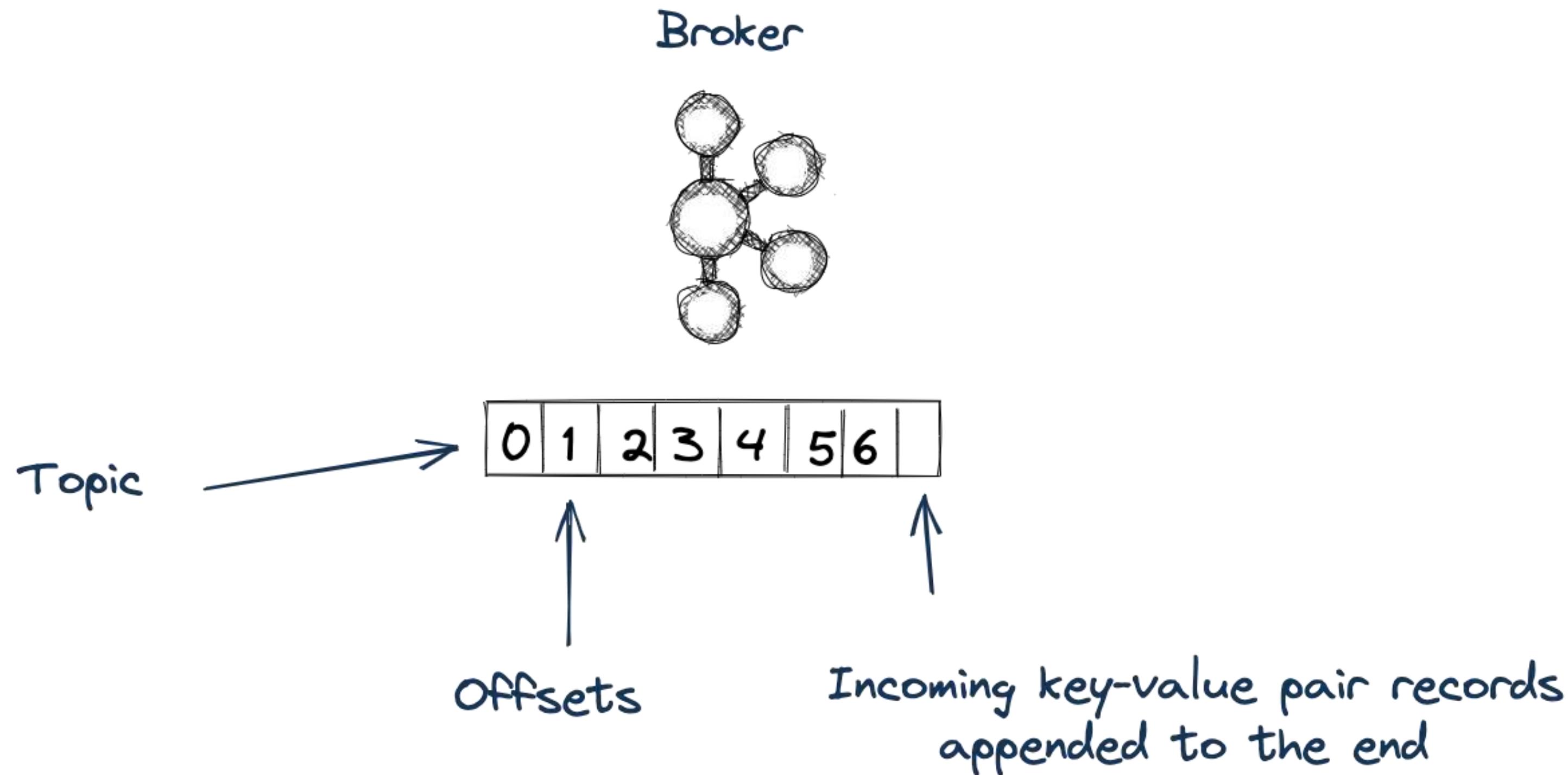
All Code for setting up load testing with instructions is given in the provided github repository.

# Schema Registry 101

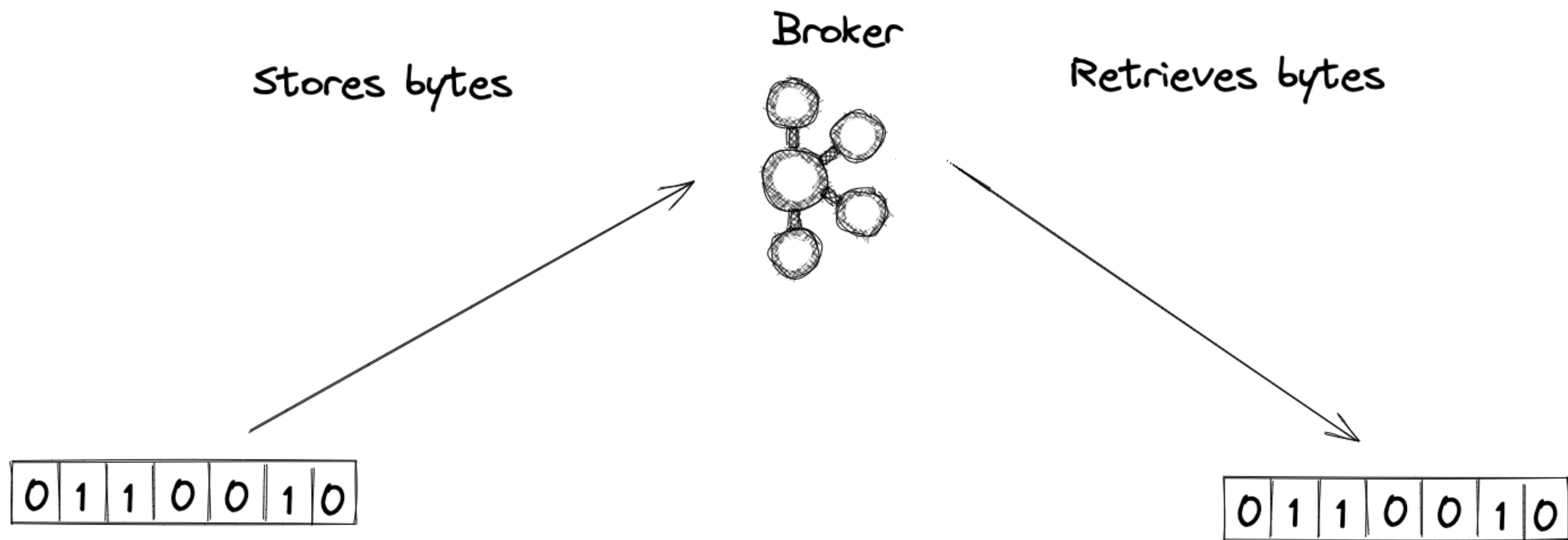
# Why Schema Registry?

A quick Apache Kafka® review

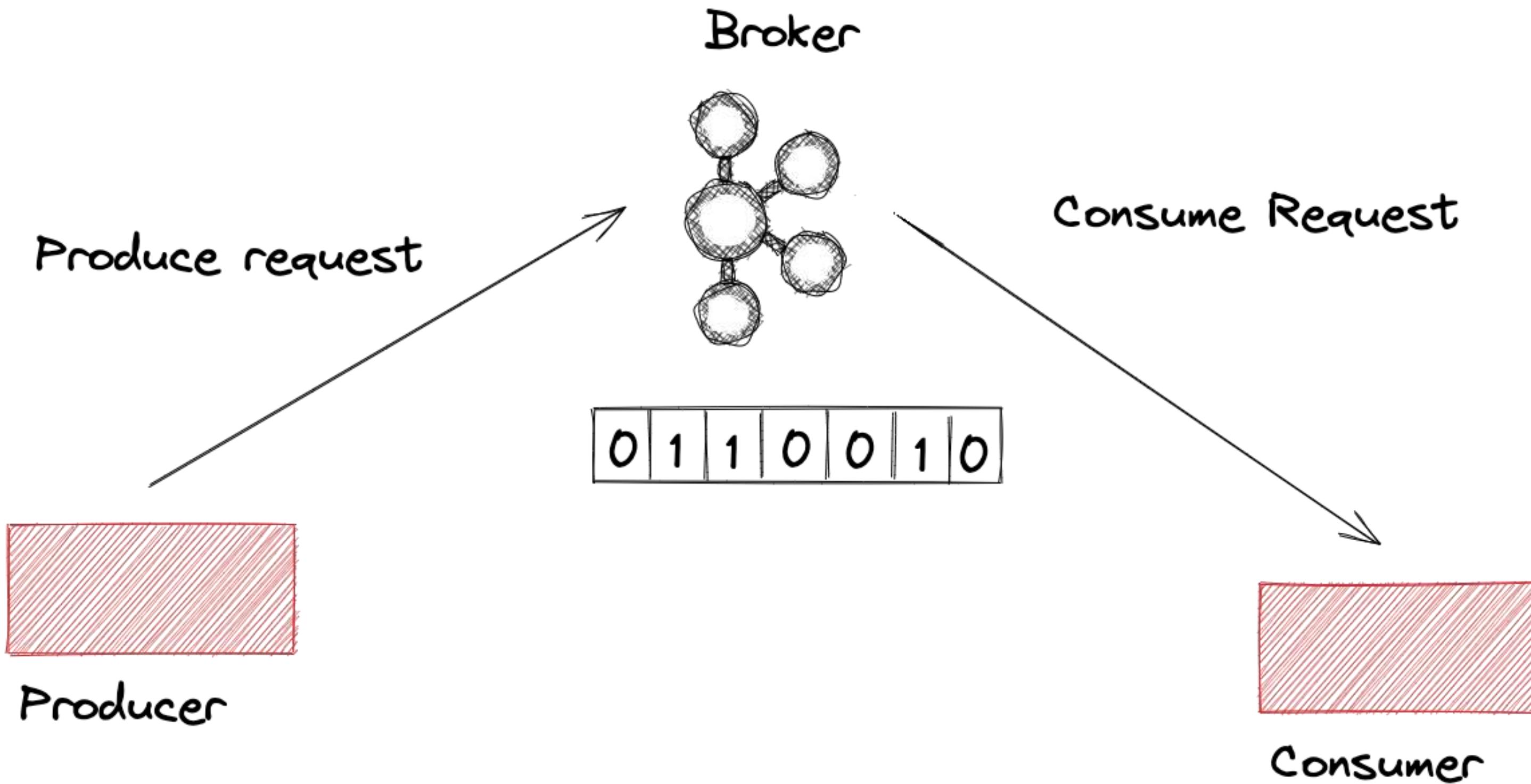
# Kafka - an append only distributed log



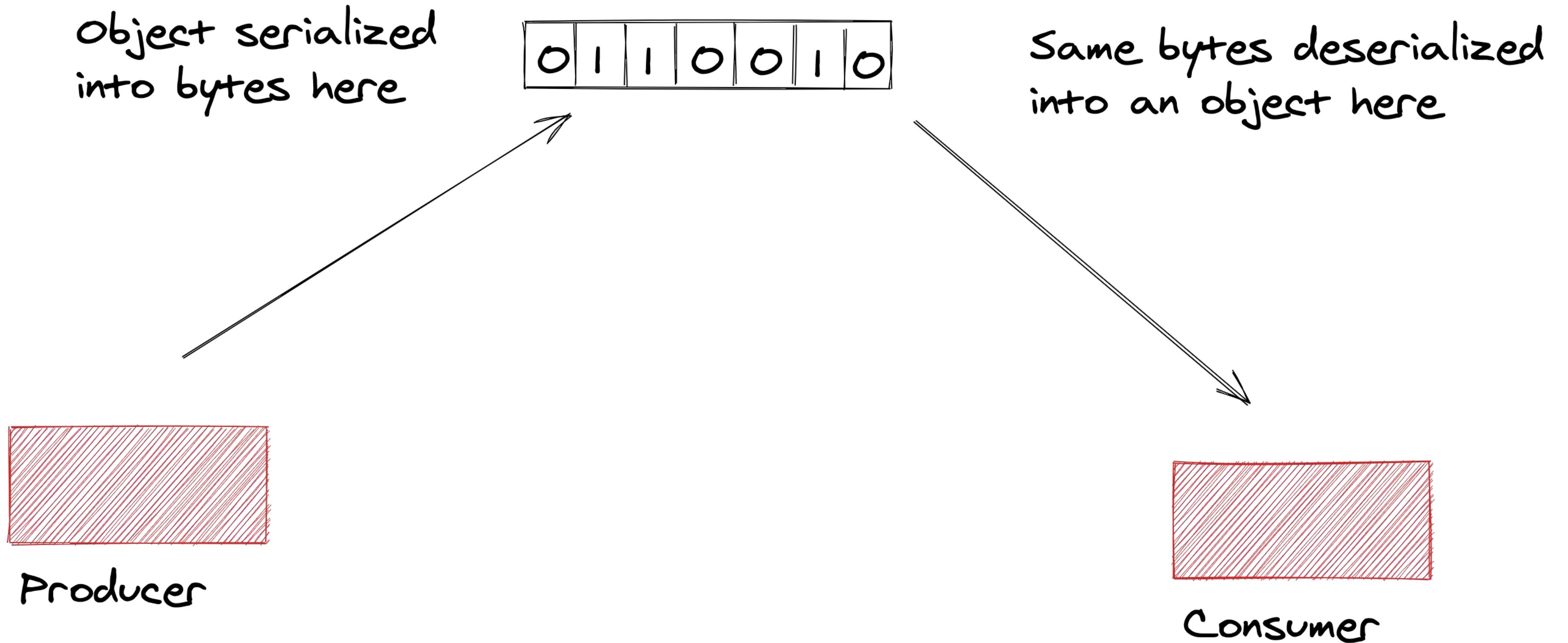
# Kafka brokers work with bytes only



# Clients produce and consume

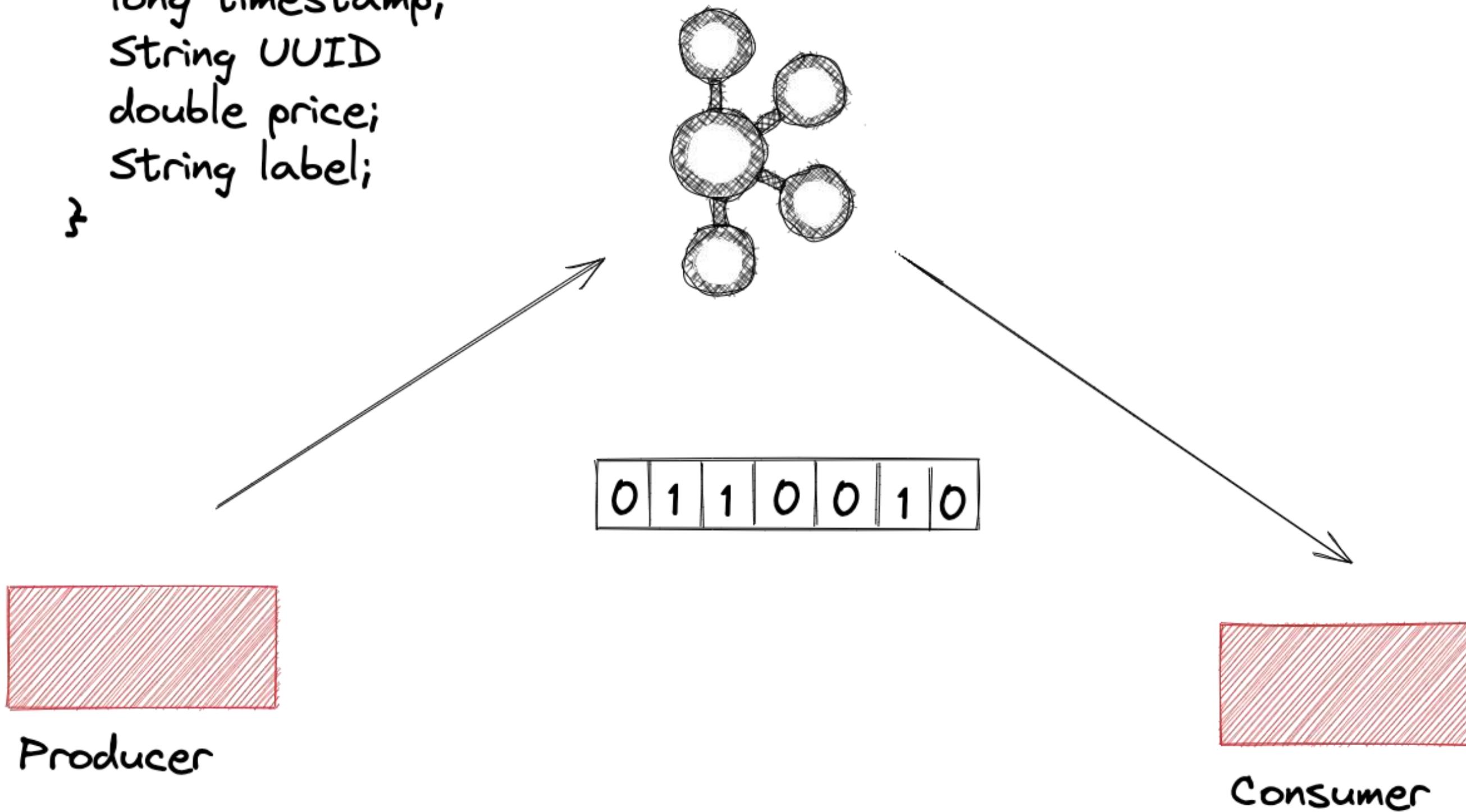


# An implicit contract



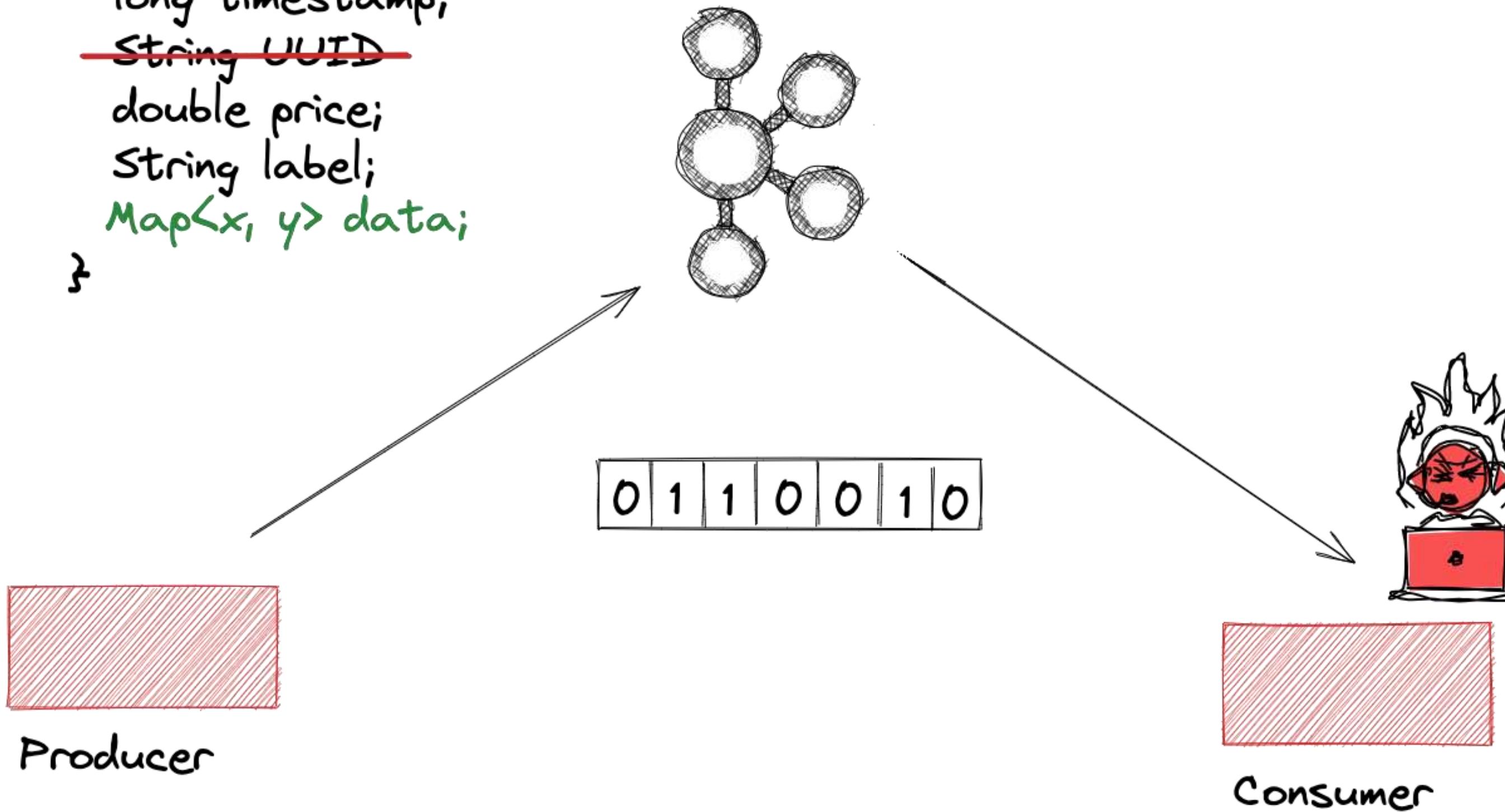
# Expectation of object structure

```
public class Foo {  
    long timestamp;  
    String UUID;  
    double price;  
    String label;  
}
```

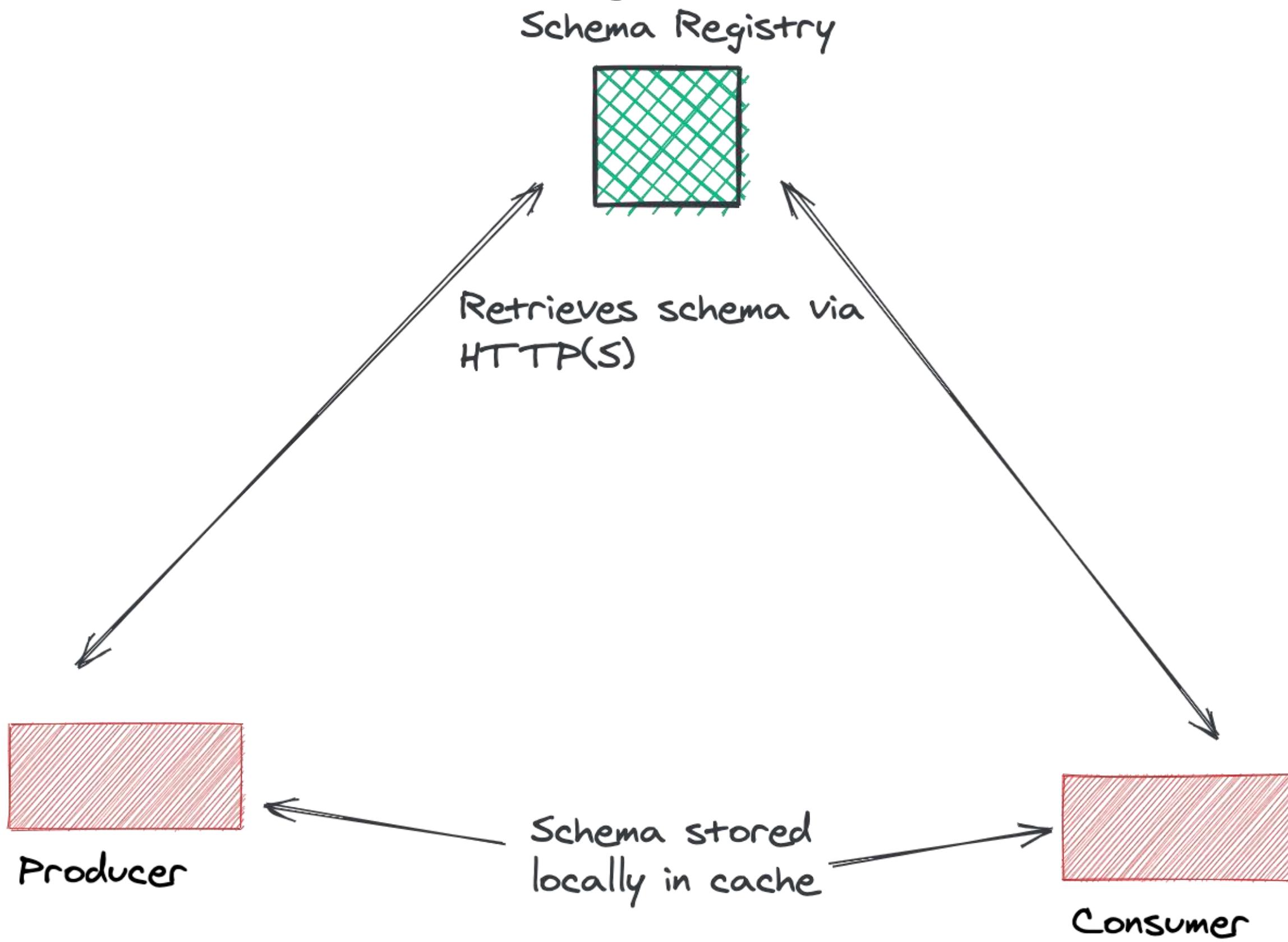


# Potentially make unexpected changes

```
public class Foo {  
    long timestamp;  
String UUID  
    double price;  
    String label;  
    Map<x, y> data;  
}
```



# Use Schema Registry FTW!



# Working with Schema Registry

1. Write/download a schema
2. Test and upload the schema
3. Generate the objects
4. Configure clients (Producer, Consumer, Kafka Streams)
5. Write your application!

# Build a Schema

- Avro
- Protocol Buffers
- JSON Schema

# Build a Schema

```
{  
  "type": "record",  
  "namespace": "io.confluent.developer.avro",  
  "name": "Purchase",  
  "fields": [  
    {"name": "item", "type": "string"},  
    {"name": "amount", "type": "double", "default": 0.0},  
    {"name": "customer_id", "type": "string"}  
]  
}
```

# Build a Schema (Protobuf)

```
syntax = "proto3";  
  
package io.confluent.developer.proto;  
option java_outer_classname = "PurchaseProto";  
  
message Purchase {  
    string item = 1;  
    double amount = 2;  
    string customer_id = 3;  
}
```

# Register

```
confluent schema-registry schema create --subject purchases-value \
--schema src/main/proto/purchase.proto \
--type PROTOBUF \
--api-key YYY \
--api-secret ZZZ

jq '. | {schema: toJson}' src/main/avro/purchase.avsc | \
curl -u API_KEY:API_SECRET \
-X POST https://CLUSTER_IP/subjects/purchases-value/versions \
-H "Content-Type:application/json" \
-d @-
```

# Register

```
./protoJsonFmt.sh src/main/proto/purchase.proto | \
curl -u API_KEY:API_SECRET \
-X POST https://<SERVER>/subjects/purchases-value/versions \
-H "Content-Type:application/json" \
-d @- \
PROTO_JSON=$(awk '{gsub(/\n/, "\\\n"); gsub(/\ /, "\\\\""); print}' $1) \
&& SCHEMA='{"schemaType":"PROTOBUF","schema":'\"$PROTO_JSON'"}' \
&& echo ${SCHEMA}
```

# Register

**Producer clients can “auto-register”**

```
producerConfigs.put("auto.register.schemas", true)
```

Not for production!!!

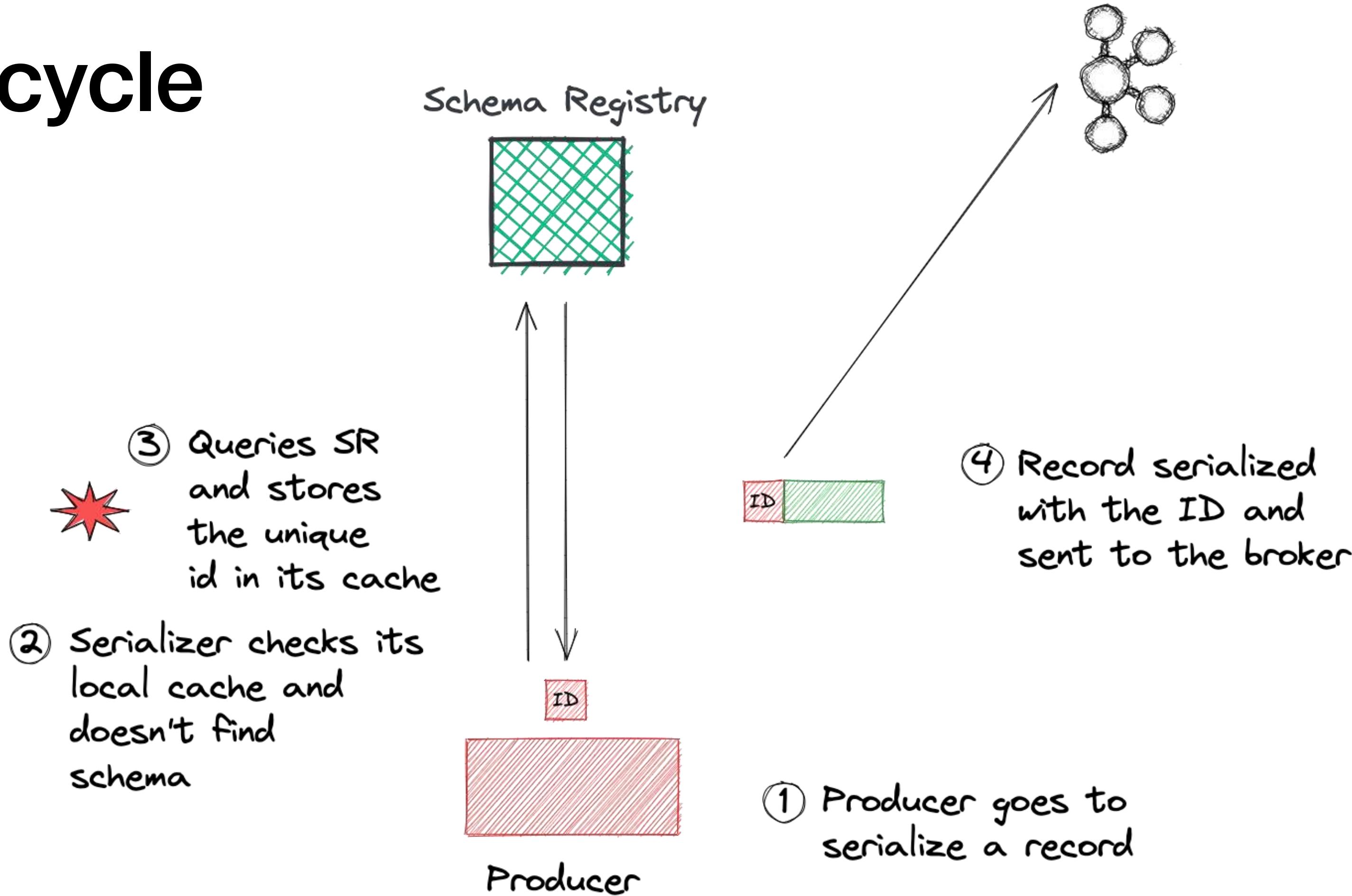
# View/Retrieve

```
confluent schema-registry schema describe --subject purchases-value \
--version [version number | "latest"]
--api-key YYY \
--api-secret ZZZ
```

```
curl -u API_KEY:API_SECRET \
-s "https://CLUSTER_IP/subjects/purchases-value/versions/latest" | jq .'
```

```
curl -u API_KEY:API_SECRET \
-s "https://CLUSTER_IP/subjects/purchases-value/versions/N" | jq .'
```

# Lifecycle



# Clients

- . basic.auth.credentials.source=USER\_INFO
- . schema.registry.url=  
    https://<CLUSTER>.us-east-2.aws.confluent.cloud
- . basic.auth.user.info=API\_KEY:API\_SECRET

# Clients - Producer

```
producerConfigs.put("value.serializer", ? );
```

- . KafkaAvroSerializer.class
- . KafkaProtobufSerializer.class
- . KafkaJsonSchemaSerializer.class

# Clients - Consumer

```
consumerConfigs.put("value.deserializer", ? );
```

- . KafkaAvroDeserializer.class
- . KafkaProtobufDeserializer.class
- . KafkaJsonSchemaDeserializer.class

# Clients - Consumer

- . specific.avro.reader = true|false
- . specific.protobuf.value.type = proto class name
- . json.value.type = class name

# Clients - Consumer Returned Types

## Avro

- . SpecificRecord – myObj.getFoo()
- . GenericRecord - generic.get("foo")

## Protobuf

- . Specific
- . Dynamic

# Clients - Kafka Streams

- SpecificAvroSerde
- GenericAvroSerde
- KafkaProtobufSerde
- KafkaJsonSchemaSerde

# Clients - Kafka Streams

```
Serde<Generated> serde = new SpecificAvroSerde<>();  
  
Map<String, String> configs =  
    Map.of("schema.registry.url", "https..", ...)  
  
serde.configure(configs)
```

# Clients - Command Line Produce

- confluent kafka topic produce purchases \
  - --value-format protobuf \
    - --schema src/main/proto/purchase.proto \
      - --sr-endpoint https://.... \
        - --sr-api-key xxxyyyy\
          - --sr-api-secret abc123 \
            - --cluster lkc-45687
    - > {"item":"pizza", "amount":17.99, "customer\_id":"lombardi"}

# Clients - Command Line Produce

```
./bin/kafka-protobuf-console-producer \
  --topic purchases \
  --bootstrap-server localhost:9092 \
  --property schema.registry.url = http://... \
  --property value.schema ="$(<src/main/proto/purchase.proto)"

> {"item":"pizza", "amount":17.99, "customer_id":"lombardi"}
```

# Clients - Command Line Consume

- **confluent kafka topic consume purchases \**
  - **--from-beginning \**
  - **--value-format protobuf \**
  - **--schema src/main/proto/purchase.proto \**
  - **--sr-endpoint https://.... \**
  - **--sr-api-key xxxyyyy\**
  - **--sr-api-secret abc123 \**
  - **--cluster lkc-45687**

# Clients - Command Line Consume

- `./bin/kafka-protobuf-console-consumer \`
  - `--from-beginning \`
  - `--topic purchases \`
  - `--bootstrap-server localhost:9092 \`
  - `--property schema.registry.url = http://...`

# Clients -Testing

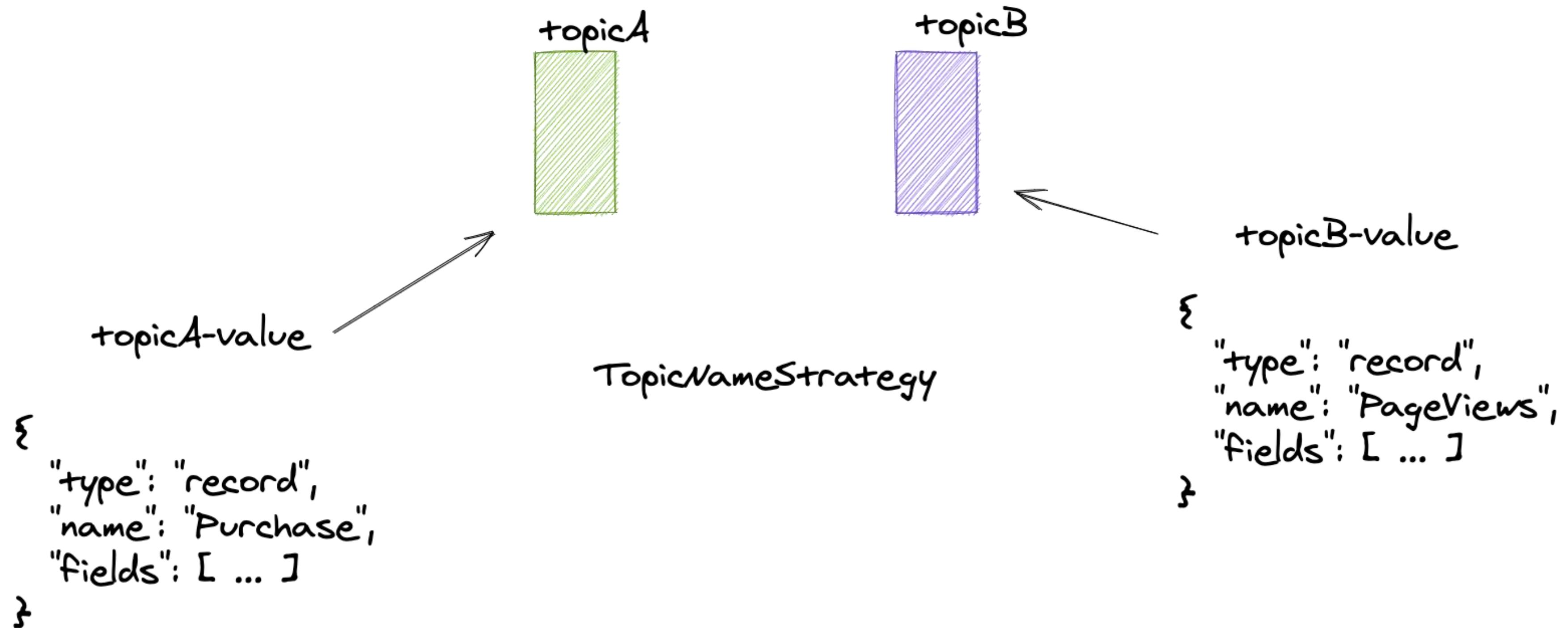
For testing you can use a mock schema registry

- . schema.registry.url= “mock://scope-name”

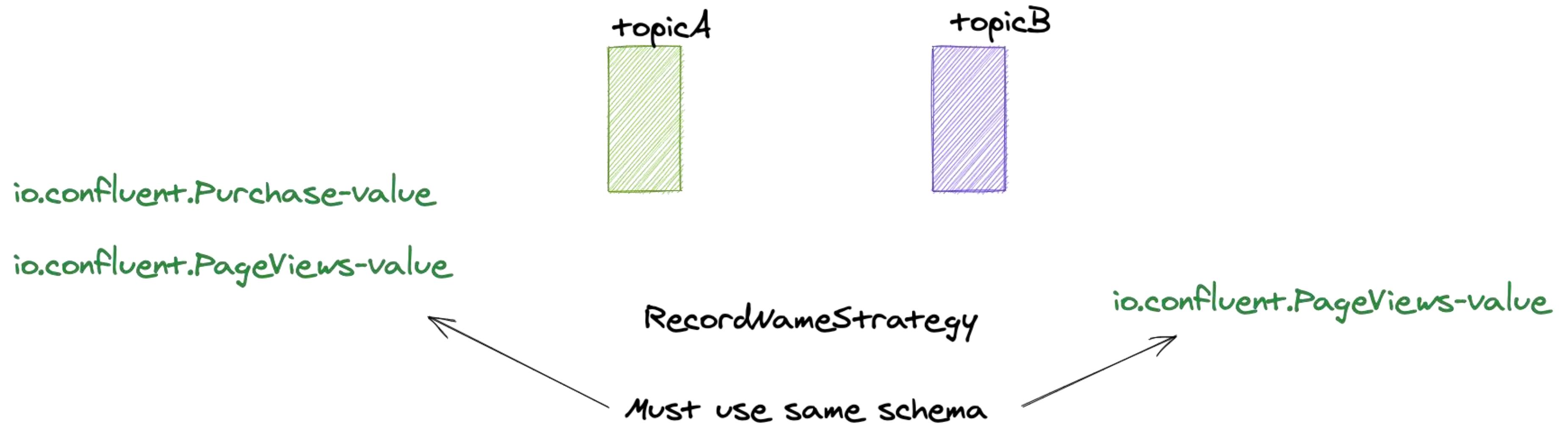
# What is a Subject ?

- . Defines a namespace for a schema
- . Compatibility checks are per subject
- . Different approaches – subject name strategies

# Subjects - TopicNameStrategy



# Subjects - RecordNameStrategy



# Subjects - TopicRecordNameStrategy

topicA-io.confluent.Purchase-value

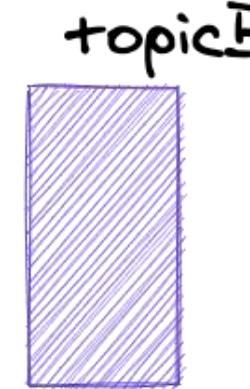


topicA-io.confluent.PageViews-value

version 1

TopicRecordNameStrategy      version 2

topicB-io.confluent.PageViews-value



Schemas scoped to the topic level, so now they can evolve individually

# Schema Compatibility

- . Schema Registry provides a mechanism for safe changes
- . Evolving a schema
- . Compatibility checks are per subject
- . When a schema evolves, the subject remains, but the schema gets a new ID and version

# Schema Compatibility

- . Backward - default
  - . Forward
  - . Full
- Latest version will work with previous version
- . But not necessarily with versions beyond that

# Schema Compatibility

- Backward transitive
- Forward transitive
- Full transitive

Latest version will work with all previous versions

# Schema Compatibility

Backward	Forward	Full
Delete fields Add fields with default values	Delete fields with default values Add fields	Delete or add fields either must have default values.
Update consumer clients first	Update producer clients first	Order of update doesn't matter

# *Testing a Schema for compatibility*

```
confluent schema-registry compatibility validate --schema src/main/proto/purchase.proto \
--type PROTOBUF \
--subject purchases-value \
--version latest \
--api-key YYY \
--api-secret ZZZ

jq '. | {schema: toJson}' src/main/avro/purchase.avsc | \
curl -u API_KEY:API_SECRET \
-X POST https://CLUSTER_IP/compatibility/subjects/purchases-value/version/latest \
-H "Content-Type:application/json" \
-d @-
```

# *Summary*

- . Kafka broker works with bytes only
- . Event objects form an implicit contract
- . Using Schema Registry allows for an explicit contract
- . Schema Registry provides for keeping domain objects in sync

# *Resources*

- . Documentation - <https://docs.confluent.io/platform/current/schema-registry/index.html#sr-overview>
- . Multiple Event Types Presentation - <https://www.confluent.io/en-gb/events/kafka-summit-europe-2021/managing-multiple-event-types-in-a-single-topic-with-schema-registry/>
- . Multiple Events tutorial - <https://developer.confluent.io/tutorials/multiple-event-type-topics/confluent.html>
- . Multiple Event Type code - <https://github.com/bbejeck/multiple-events-kafka-summit-europe-2021>

# *Resources*

- . Martin Kleppmann - <https://www.confluent.io/blog/put-several-event-types-kafka-topic/>
- . Robert Yokota - <https://www.confluent.io/blog/multiple-event-types-in-the-same-kafka-topic/>
- . Avro - <https://avro.apache.org/docs/current/>
- . Protocol Buffers - <https://developers.google.com/protocol-buffers>
- . JSON Schema - <https://json-schema.org/>

# Build tools

- Schema Registry Maven
  - <https://docs.confluent.io/platform/current/schema-registry/develop/maven-plugin.html#sr-maven-plugin>
- Schema Registry Gradle
  - <https://github.com/lmFlog/schema-registry-plugin>
- Protobuf Gradle
  - <https://github.com/google/protobuf-gradle-plugin>
- Avro Gradle
  - <https://github.com/davidmc24/gradle-avro-plugin>

# Build tools

## JSON Schema

- <https://github.com/jsonschema2dataclass/js2d-gradle>
- <https://github.com/joelittlejohn/jsonschema2pojo>

# Kafka Logging

# KAFKA LOGGING

- One Use Case is when you want to process your application logs.
- Being written in Microservices, apps logs are distributed across all services/containers

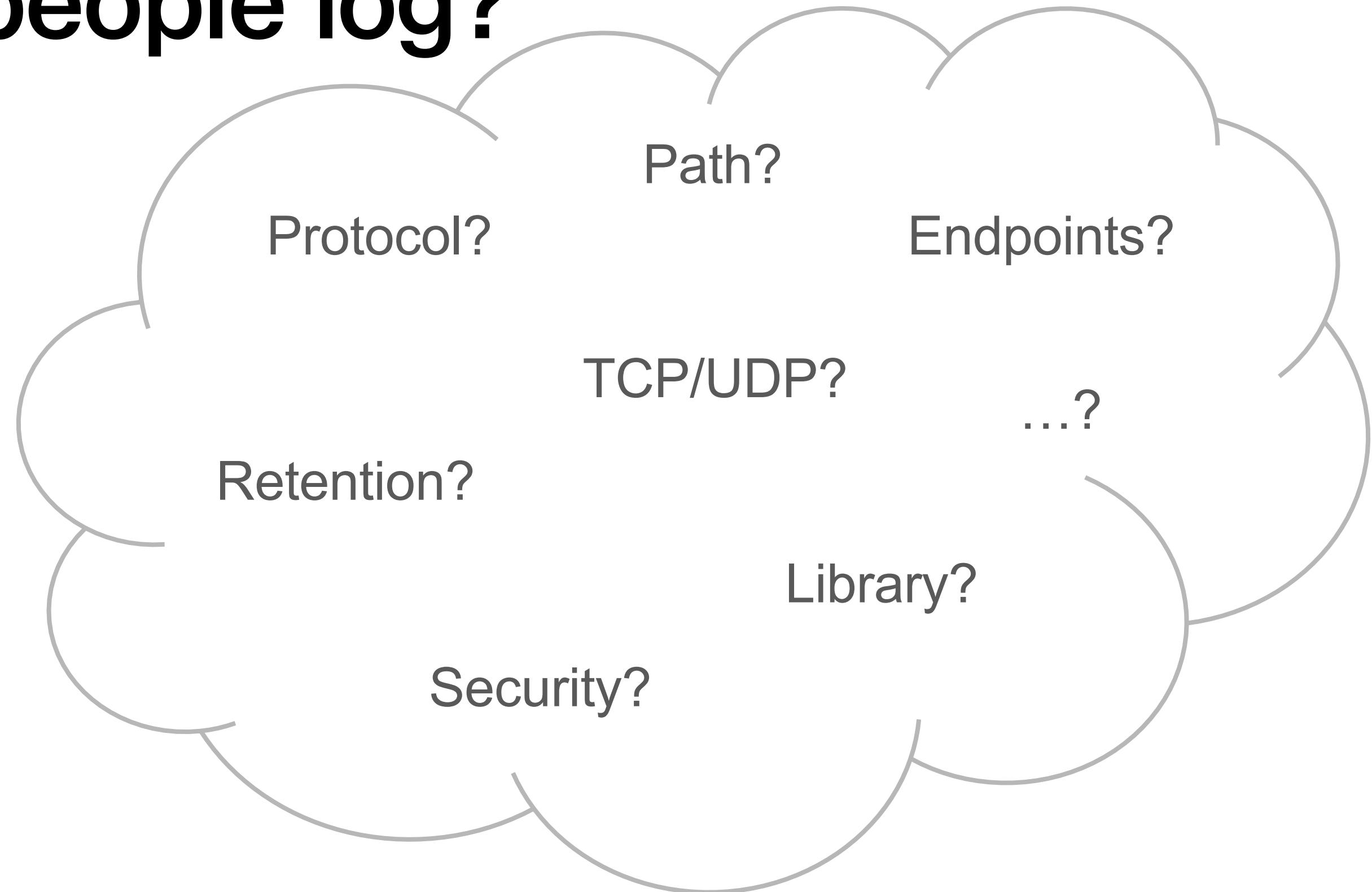
# Why do people log?

- To ensure their application is running
- To debug running application
- To see activity traces
- To see errors or edge cases manifesting
- To gather audit information
- To collect runtime metrics/usage patterns

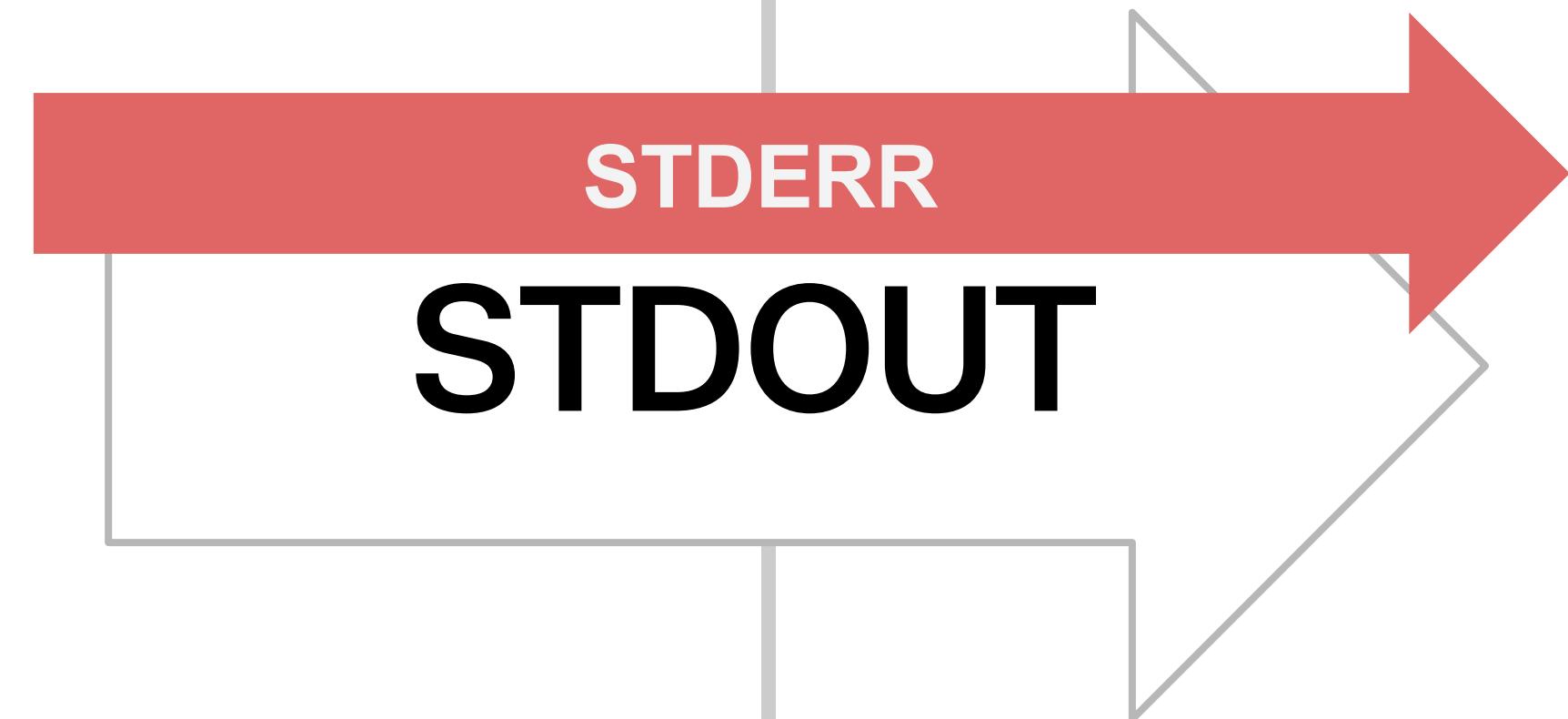
... ordered by subjective importance

# How do people log?

- STDOUT
- STDERR
- FILE
- NETWORK
- UNIX SOCKET



application



aggregator



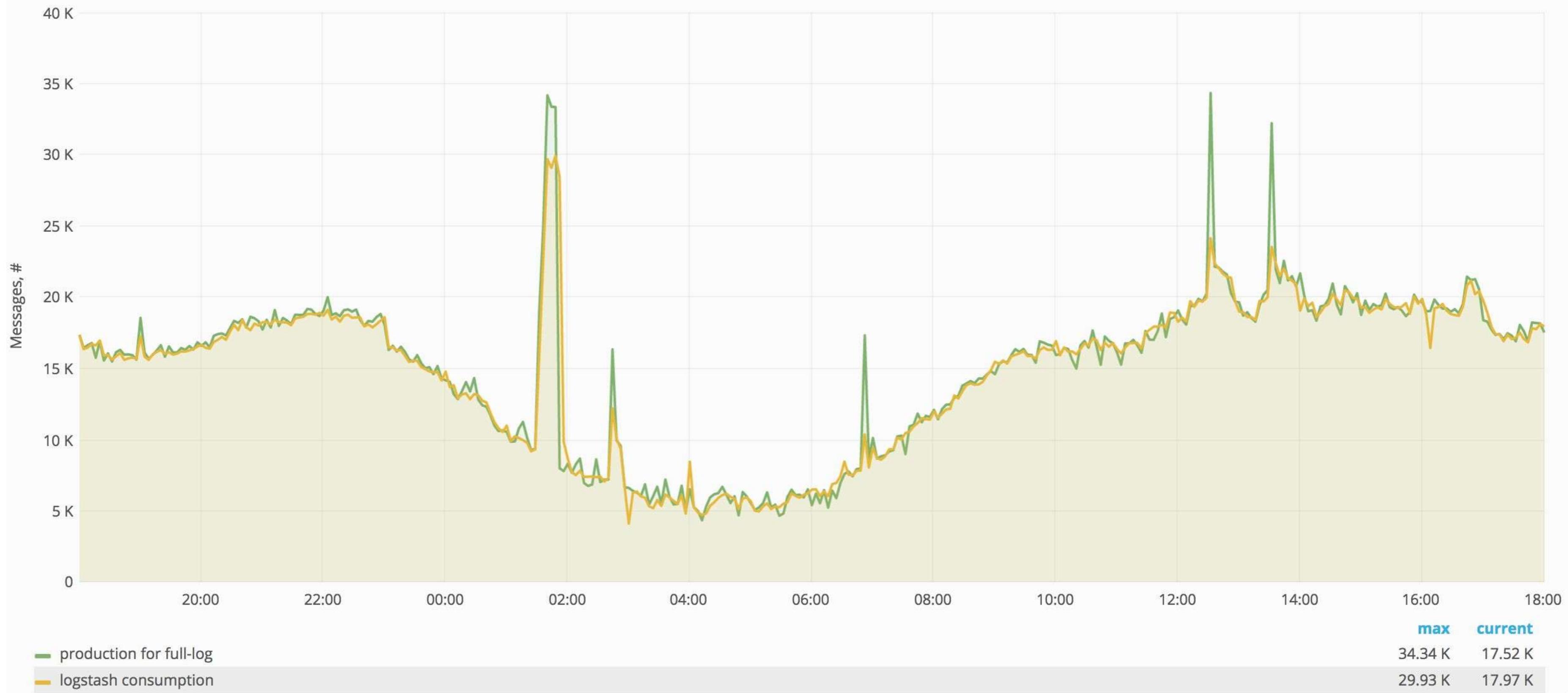
**But  
why...?**

# Because...

1. By decoupling logging transport from application you enable other team(s) to ensure your logs get to a central place without any effort from your side
2. By having log transport not bound to complicated assumptions your application is more portable
3. By forcing yourself away from standard logging protocols you can use elegant data encoding options (....JSON, ....)

**SEND DEBUGS**

### Total production/consumption rates



# TURN ON DEBUGGING ONLY IF NEEDED

- ~30k msg/s
- ~2k rps
- ~200GB/h
- ~32TB/week
- **+3000\$/month**

# What should be logged?

- Usually you filter out and only look for understandable logs
  - Searching for needle in a barn is not a fun activity
- Logs are not a good place to store sensitive data
  - “user %s authorised with password %s bought item SKU#%s” is my favourite.
- You want to see patterns, not individual errors
  - “%s caught for %s user, while in %s” is hard to find, when you are not 100% certain what are you looking for
  - “exception caught in purchase pipeline” is better, I’ll cover the details in next topic.

```
[pid: 5151|app: 0|req: 2166118/25642392] 10.5.244.32 () {42 vars in 865 bytes} [Wed Jul 19 18:54:39 2017] GET  
/api/v1/pp_preview/?models=reviews%2Cquestions&sku=XXXXXXXXXXXX&limit=50&country=by =>  
generated 1285 bytes in 20 msecs (HTTP/1.0 200) 5 headers in 154 bytes (1 switches on  
core 0)
```

VS

```
{"pid": 5151, "req_n": 2166118, "source_ip": "10.5.244.32", "response_size": 1285,  
"timestamp": "2017-07-19T18:54:39Z", "http_method": "GET",  
"http_url": "/api/v1/pp_preview/?models=reviews%2Cquestions&sku=XXXXXXXXXXXX&limit=50&c  
ountry=by", "response_time": "0.02", ... }
```

# Application Logging

- Understanding/predicting necessary information
  - Do you need a particular field?
  - Is it useful now?
  - How about future?
  - What can you aggregate from a field?
- Think about aggregations
  - Average over time? You need a number.
  - Histogram on field? You need limited set of possible values.
  - Pie chart? Limited set of values.
  - ... (yes, I don't mention geo based aggregations, because you can google that one out)
- Use Libraries for Tracing
  - OpenTelemetry is only one
  - Context should be propagated to allow for easy analysis

# Interpreting & Debugging Kafka Logs - RID THIS

- server.log
- controller.log
- state-change.log



- Notes on LogManager  
<https://jaceklaskowski.gitbooks.io/apache-kafka/content/kafka-log-LogManager.html>
- <https://jaceklaskowski.gitbooks.io/apache-kafka/content/kafka-log-cleanup-policies.html>

# Log Appenders

- Apache Kafka brokers and controllers use Log4j 2 for logging
- Can configure a variety of appenders to route logs to different destinations.

# Log Appenders

Appender Name	Purpose
KafkaAppender	General broker logs (INFO level and above)
StateChangeAppender	Logs state transitions in the Kafka cluster
RequestAppender	Captures client requests handled by the broker
CleanerAppender	Logs compaction and cleanup activity for log segments
ControllerAppender	Logs controller-specific events and decisions
AuthorizerAppender	Logs authorization decisions (useful for debugging ACLs)
MetadataServiceAppender	Tracks metadata changes across the cluster
AuditLogAppender	Captures audit logs, especially when audit messages fail to reach Kafka
DataBalancerAppender	Logs activity related to Kafka's data balancing mechanisms
ConsoleAppender	Outputs logs to stdout or stderr
RollingFileAppender	Writes logs to files with time-based rotation

# Log Appenders

- Logging options can be overridden in config files using
  - **KAFKA\_LOG4J\_OPTS="-Dlog4j.configuration=file:/path/to/log4j2.yaml"**
- Each Appender can be tuned with own layout, file pattern, log level, Rolling levels
- The additivity property should be set to false to avoid recursive logging. E.g. kafka.request.logger.

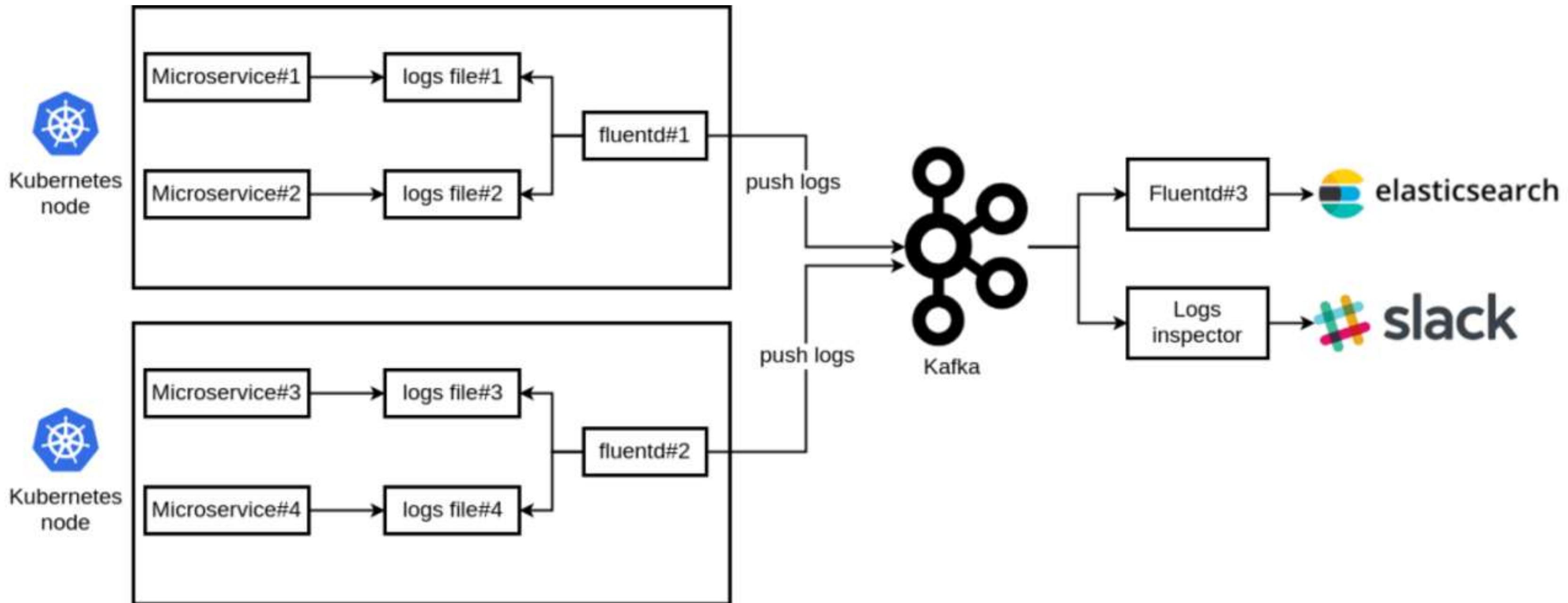
# LOG APPENDERS

- **Audit & Authorizer Logging:** Enable finer-grained access logs by setting `kafka.authorizer.logger` level to DEBUG or TRACE.
- **Network Traffic Inspection:** For debugging SASL/mTLS setups, add logging for `org.apache.kafka.common.network`.
- **DR & Replication Events:** Include `kafka.server.ReplicaFetcherThread` logger to observe replica lag and recovery.
- **Loop Prevention:** Log headers and interceptors via `org.apache.kafka.common.header` for custom routing logic.

# Logging

- If written to direct files, they need to be roll overed
- Importantly, these files need to be shipped to ELK.
- There are several tools
  - ELK
  - LOKI
  - Filebeats/FluentD

# KAFKA LOGGING ARCHITECTURE #1



# Kafka Security

# KAFKA Security

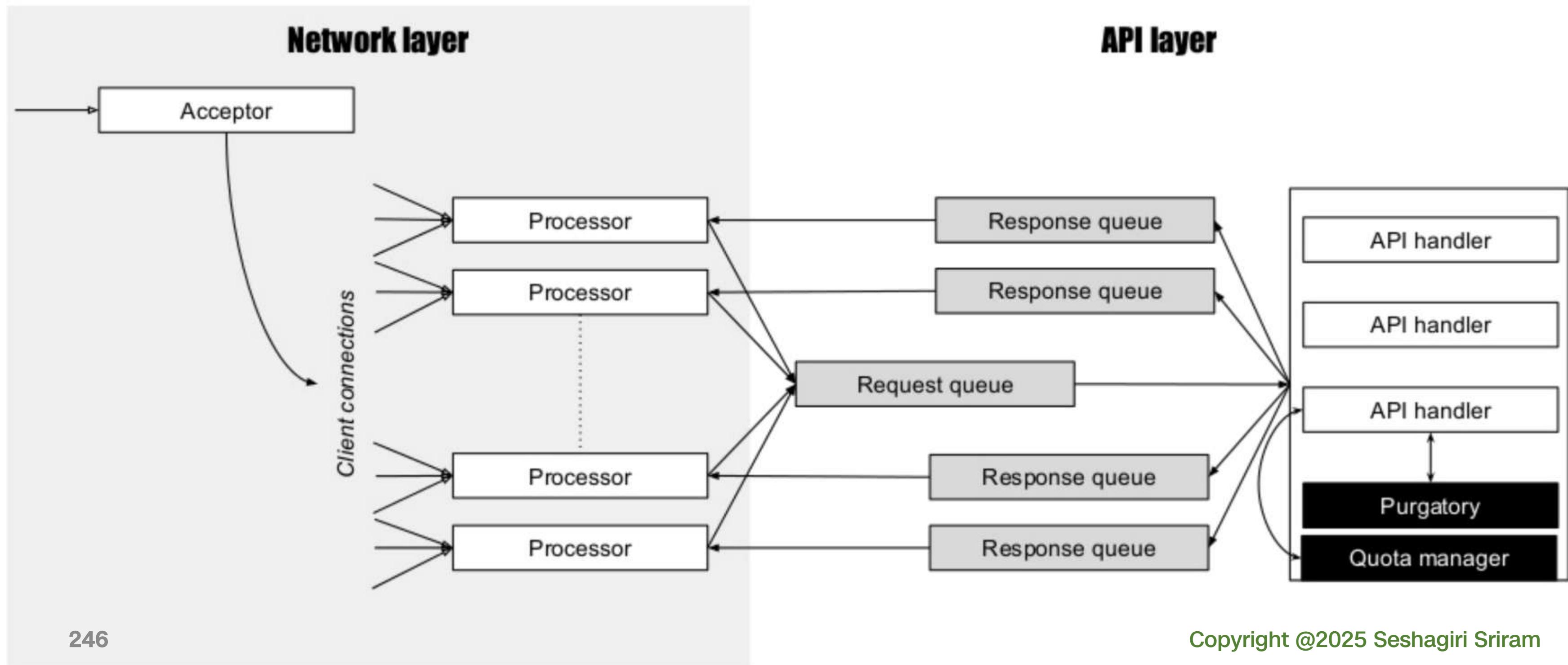
- How can we prevent rogue agents to publishing/consuming data from Kafka
- How can we encrypt the data that's flowing through the network
- How can we give permissions to a topic to specific group or users

# KAFKA

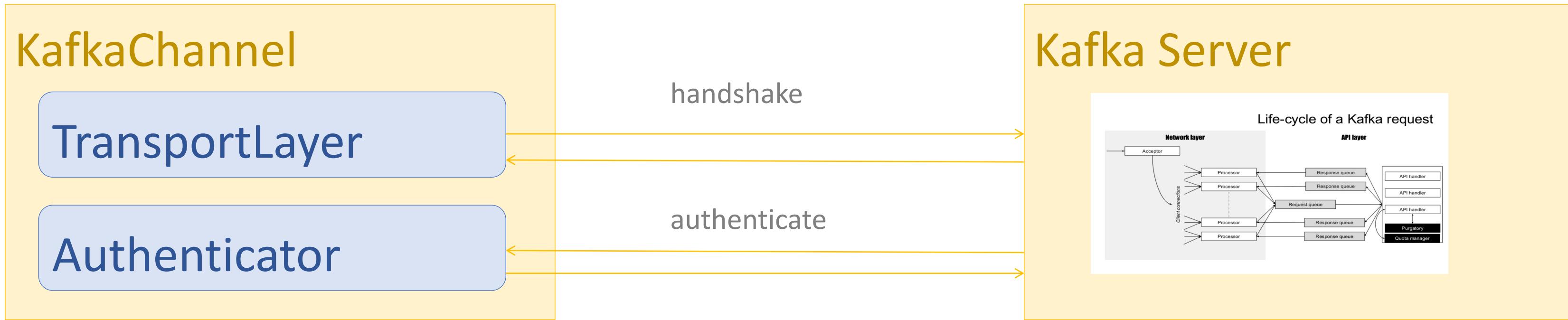
- By Default, we use PLAINTEXT
  - No Encryption
  - All in plain text
  - No Authentication or Authorization

# KAFKA Security

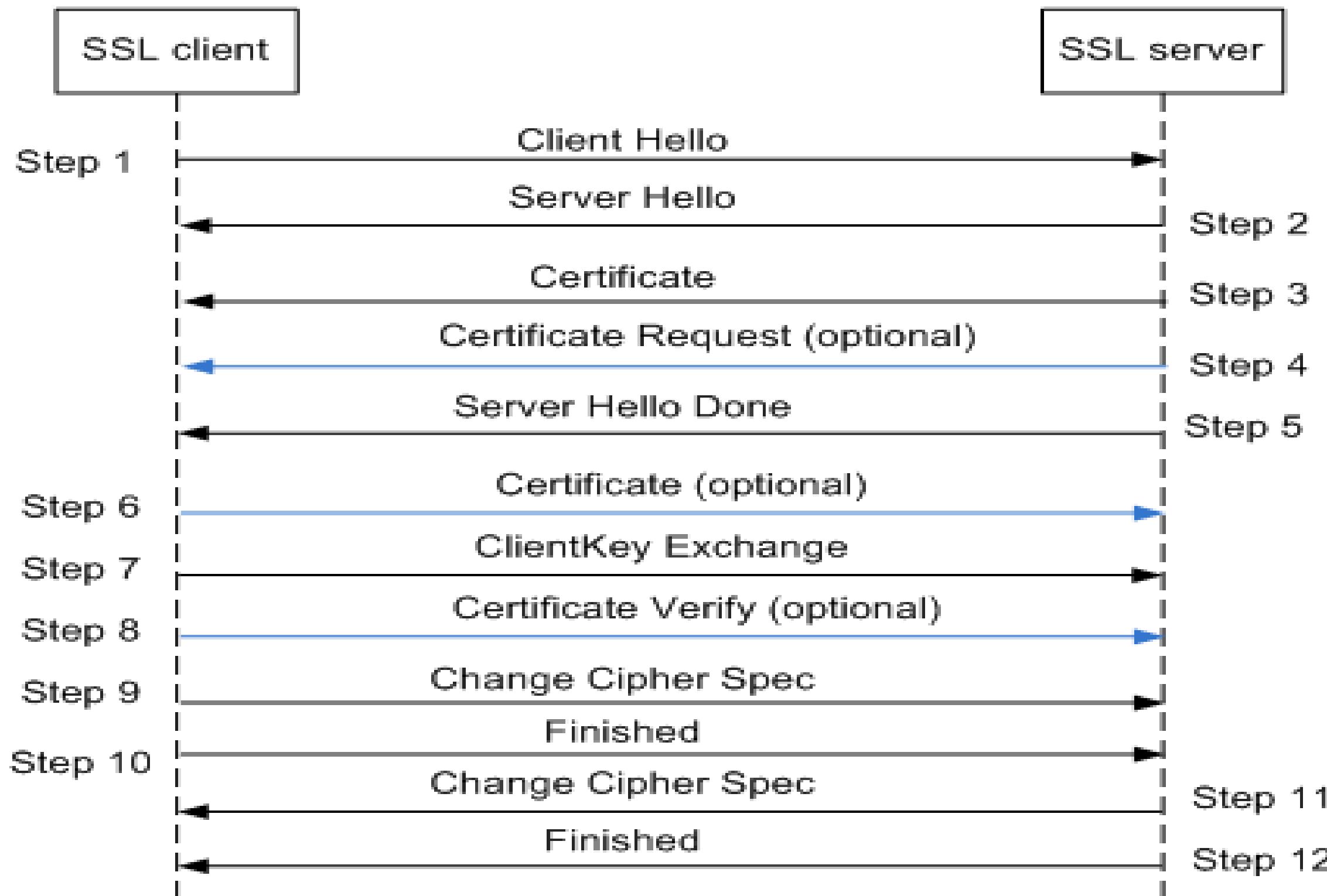
## Life-cycle of a Kafka request



# Kafka Networking



# Kafka Security - SSL



# Kafka Security

- **Simple Authentication and Security Layer, or SASL**
  - Provides flexibility in using Login Mechanisms
  - One can use Kerberos , LDAP or simple passwords to authenticate.
- **JAAS Login**
  - Before client & server can handshake , they need to authenticate with Kerberos or other Identity Provider.
  - JAAS provides a pluggable way of providing user credentials. One can easily add LDAP or other mechanism just by changing a config file.
-

# Kafka Security

```
KafkaServer {  
    com.sun.security.auth.module.Krb5LoginModule required  
    useKeyTab=true  
    storeKey=true  
    serviceName="kafka"  
    keyTab="/vagrant/keytabs/kafka1.keytab"  
    principal="kafka/host@EXAMPLE.COM";  
};
```

```
KafkaClient {  
    com.sun.security.auth.module.Krb5LoginModule required  
    useKeyTab=true  
    storeKey=true  
    serviceName="kafka"  
    keyTab="/vagrant/keytabs/client1.keytab"  
    principal="client/host@EXAMPLE.COM";  
};
```

# Let's Get our hands dirty

- **listeners=PLAINTEXT://:9092**
- **advertised.listeners=PLAINTEXT://your.host.name:9092**
- **listener.security.protocol.map=PLAINTEXT:PLAINTEXT**
- **inter.broker.listener.name=PLAINTEXT**

THIS IS PLAIN SIMPLE CONFIGURATION - No Authentication Every thing in plain text

# Let's Get our hands dirty

- Let's now add a single user (Still no Encryption)
- We need to change the broker properties and add a JAAS Config file
- Also clients can be changed.

# Let's Get our hands dirty

- **listeners=SASL\_PLAINTEXT://:9093**
- **advertised.listeners=SASL\_PLAINTEXT://your.host.name:9093**
- **listener.security.protocol.map=SASL\_PLAINTEXT:SASL\_PLAINTEXT**
- **inter.broker.listener.name=SASL\_PLAINTEXT**
- **sasl.mechanism.inter.broker.protocol=PLAIN**
- **security.inter.broker.protocol=SASL\_PLAINTEXT**

# Let's Get our hands dirty

- KafkaServer {
- 
- org.apache.kafka.common.security.plain.PlainLoginModule required
- username="admin"
- password="admin-secret"
- user\_admin="admin-secret"
- user\_alice="alice-password";
- };

# Let's Get our hands dirty

- **KAFKA\_OPTS="-Djava.security.auth.login.config=/path/to/kafka\_server\_jaas.conf"**

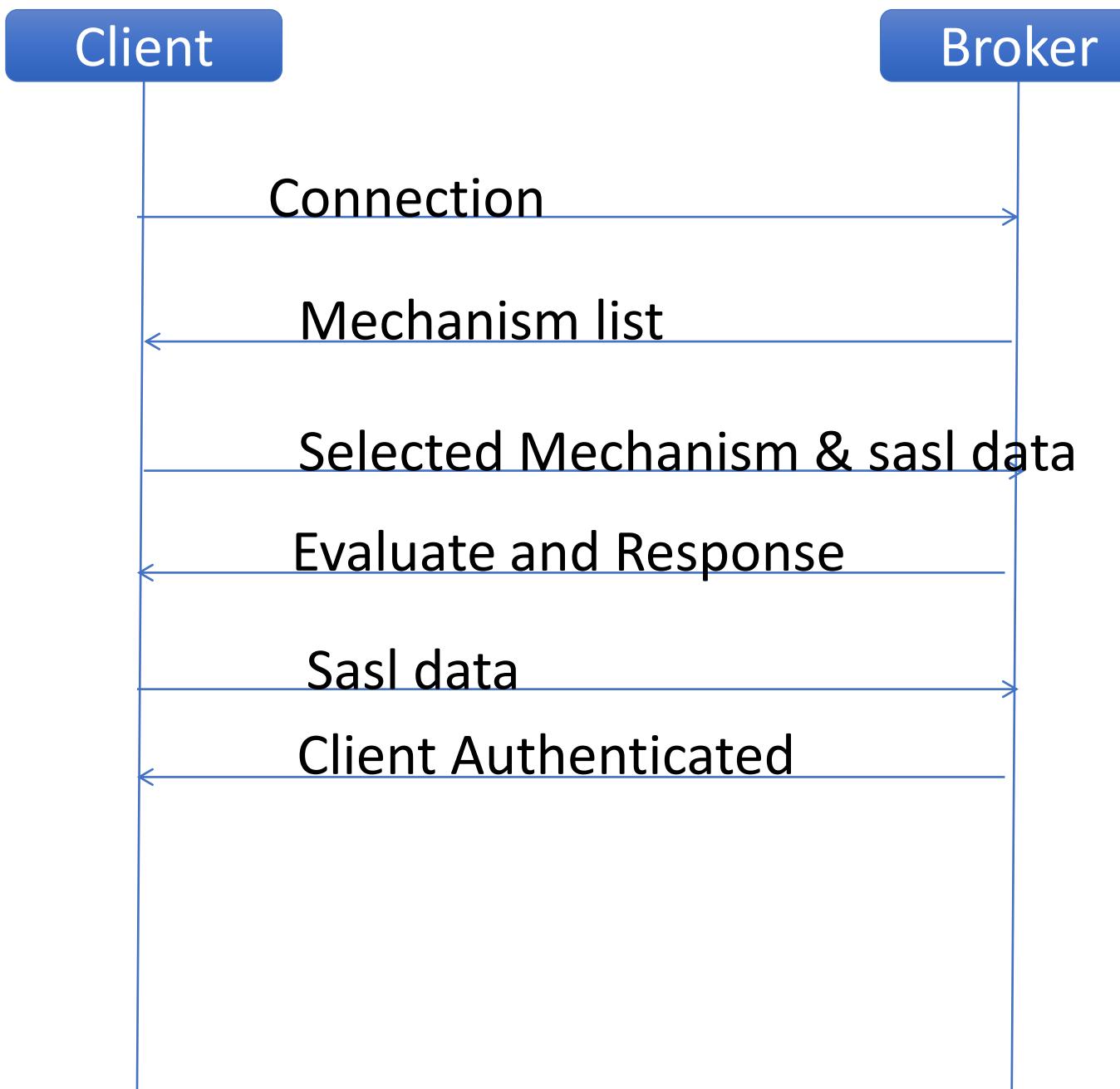
# Let's Get our hands dirty

- KafkaClient {  
org.apache.kafka.common.security.plain.PlainLoginModule  
required username="alice" password="alice-password";};
- export KAFKA\_OPTS="-  
Djava.security.auth.login.config=/path/to/client\_jaas.conf"  
bin/kafka-console-producer.sh \ --broker-list  
your.host.name:9093 \ --topic test \ --producer.config  
client.properties

# Let's Get our hands dirty

- The PlainLoginModule is never ever to be used in Public
- 
- What we have done so far is to use SASL/PLAIN and the broker will create an in-memory mapping
- E.g. user\_alice=“alice-password”
- The user alice has a password “password”

# Kafka Security



# Kafka Security

- Kafka doesn't natively support external databases for password verification in SASL/PLAIN.
- Can integrate a database-backed credential store by implementing a custom JAAS login module, which Kafka supports.

# Let's Get our hands dirty

- Alternatively
- Put a proxy service (e.g., Envoy, OAuth2 sidecar, or custom REST gateway) in front of Kafka
- Handle auth via database or identity provider (IdP)
- Only route verified clients to Kafka

This isolates your auth logic and can support:

- OAuth
- LDAP
- JWT
- Database credential validation

# Kafka Security

- Kafka's JAAS setup is designed for simplicity and JVM-local credential mapping:
- SASL/PLAIN: In-memory
- SASL/SCRAM: Credential hashes stored in ??

# Kafka Security

- With SASL/PLAIN,
  - There's no call to a file, LDAP, or database.
  - Everything stays inside the JVM memory via the user\_<username> entries.
  - It's simple, but risky for production unless network is restricted.

Recommended to move to SCRAM or SSL-based transport

Avoid hardcoding secrets (or wrap them with Docker secrets/env vars/vaults)

# Kafka Security

- With SASL/PLAIN,

Concern	Detail
 Passwords in clear	Sent over the wire in plaintext
 Stored unencrypted	Broker reads passwords from JAAS as raw strings
 No hashing	Passwords are not salted or hashed—just compared verbatim
 Memory-resident	User creds are held in JVM memory—visible via JMX/thread dumps

# KAFKA Security

- With SASL/SCRAM, credentials are hashed using PBKDF2
- These are stored in ---???
- Raw Passwords are not transmitted.

# KAFKA Security

- On server side.

```
listeners=SASL_PLAINTEXT://:9093
advertised.listeners=SASL_PLAINTEXT://your.host.name:9093
listener.security.protocol.map=SASL_PLAINTEXT:SASL_PLAINTEXT
inter.broker.listener.name=SASL_PLAINTEXT
sasl.mechanism.inter.broker.protocol=SCRAM-SHA-512
security.inter.broker.protocol=SASL_PLAINTEXT
```

# KAFKA Security

- On server side.

```
KafkaServer {  
    org.apache.kafka.common.security.scram.ScramLoginModule required  
    username="admin"  
    password="admin-secret";  
};
```

A few points to note here

User Credentials are not stored here.  
These users will need to be provisioned first.

# KAFKA Security

- On server side

```
# Create user 'alice' with password 'alice-password'  
bin/kafka-configs.sh --zookeeper localhost:2181 \  
--alter --add-config 'SCRAM-SHA-512=[password=alice-password]' \  
--entity-type users --entity-name alice
```

```
# (Optional) Add inter-broker user  
bin/kafka-configs.sh --zookeeper localhost:2181 \  
--alter --add-config 'SCRAM-SHA-512=[password=admin-secret]' \  
--entity-type users --entity-name admin
```

# KAFKA Security

- On client side.

```
KafkaClient {  
    org.apache.kafka.common.security.scram.ScramLoginModule required  
    username="alice"  
    password="alice-password";  
};
```

```
# Client Properties  
security.protocol=SASL_PLAINTEXT  
sasl.mechanism=SCRAM-SHA-512
```

# Kafka Security

Component	Status
Broker JAAS	SCRAM module
ZooKeeper/KRaft	User credentials stored securely
Client JAAS	Password passed to broker
Wire Protocol	Still PLAINTEXT (add SSL later if needed)

# Kafka Security - Review

- We started with PLAINTEXT – No Authentication
- Add a simple User WITH PLAINTEXT (SASL\_PLAIN)
- Then moved to SASL\_SCRAM
- Now Moving to SASL\_SSL

# KAFKA Security

- On server side.

```
listeners=SASL_SSL://:9093
advertised.listeners=SASL_SSL://your.host.name:9093
listener.security.protocol.map=SASL_SSL:SASL_SSL
inter.broker.listener.name=SASL_SSL
sasl.mechanism.inter.broker.protocol=SCRAM-SHA-512
security.inter.broker.protocol=SASL_SSL
ssl.keystore.location=/path/to/keystore.jks
ssl.keystore.password=keystore-password
ssl.key.password=key-password
ssl.truststore.location=/path/to/truststore.jks
ssl truststore password=truststore-password
```

# KAFKA Security

- On server side.

```
KafkaServer {  
    org.apache.kafka.common.security.scram.ScramLoginModule required  
    username="admin"  
    password="admin-secret";  
};
```

A few points to note here

User Credentials are not stored here.  
These users will need to be provisioned first.

# KAFKA Security

- On client side.

```
KafkaClient {  
    org.apache.kafka.common.security.scram.ScramLoginModule required  
    username="alice"  
    password="alice-password";  
};
```

```
security.protocol=SASL_SSL  
sasl.mechanism=SCRAM-SHA-512  
ssl.truststore.location=/path/to/truststore.jks  
ssl.truststore.password=truststore-password
```

# Kafka Security

Component	Status
Broker JAAS	SCRAM module
ZooKeeper/KRaft	User credentials stored securely
Client JAAS	Password passed to broker
Wire Protocol	Encrypted with SSL

# Kafka Security

- As you can see this is very cumbersome with one cluster.
- Factor this for a large number of clusters/brokers and you will see the need for automation

A Complete code solution automating this process with near zero down time and key rotation included is shared in the github repository

# Kafka Security

- Step #1 Generate Keystores and Truststore (You do need keytool – Unix based ☺)
- Step #2 These need to be propagated to all the brokers
- Step #3 Cert Rotation Scripts (NEAR ZERO DOWNTIME)

# Kafka Security

- Step #2
  - Symlink cert paths to `server.properties` via volume mounts
  - Inject broker-specific env vars (e.g. `BROKER_ID`, `CN`) to template configs
  - Optionally render JAAS and `server.properties` using `Jinja2` or shell

# Kafka Security

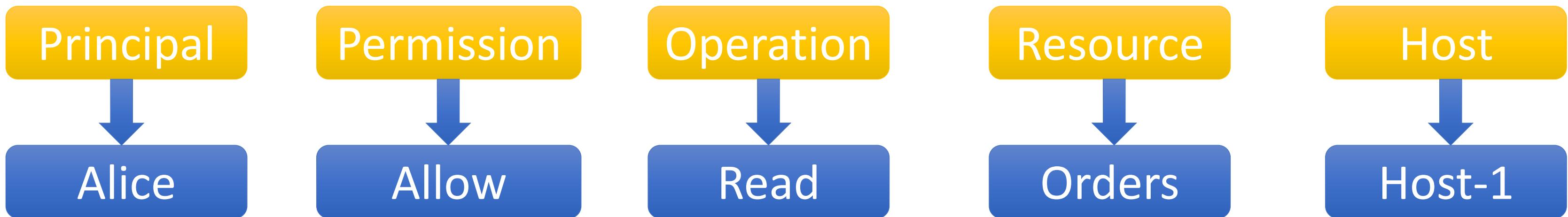
- Step #3
  - To rotate certs safely:
    - Generate new certs in parallel (with a different alias or CN)
    - Update keystore/truststore atomically
    - Bounce brokers one at a time to avoid cluster unavailability

# Kafka Security

- So far we have covered Authentication.
- Kafka allows to set authorization rules
  - Who can do what on which Objects
  - Pluggable Authorizers
  - Access Control List (ACL) based approach

# Access Control Lists

- Alice is Allowed to Read from Orders-topic from Host-1



# Principal

- PrincipalType:Name
- Supported types: User
- Extensible so users can add their own types
- Wild Card User:\*

# Operation

- **Read, Write, Create, Delete, Alter, Describe, ClusterAction, All**

# Resource

- ResourceType:ResourceName
- Topic, Cluster and ConsumerGroup
- Wild card resource ResourceType:\*

# Permissions

- Allow and Deny
- Anyone without an explicit Allow ACL is denied
- Then why do we have Deny?
- Deny works as negation
- Deny takes precedence over Allow Acls

# Hosts

- Why provide this granularity?
- Allows authorizer to provide firewall type security even in non secure environment.
- \* as Wild card.

# Configuration

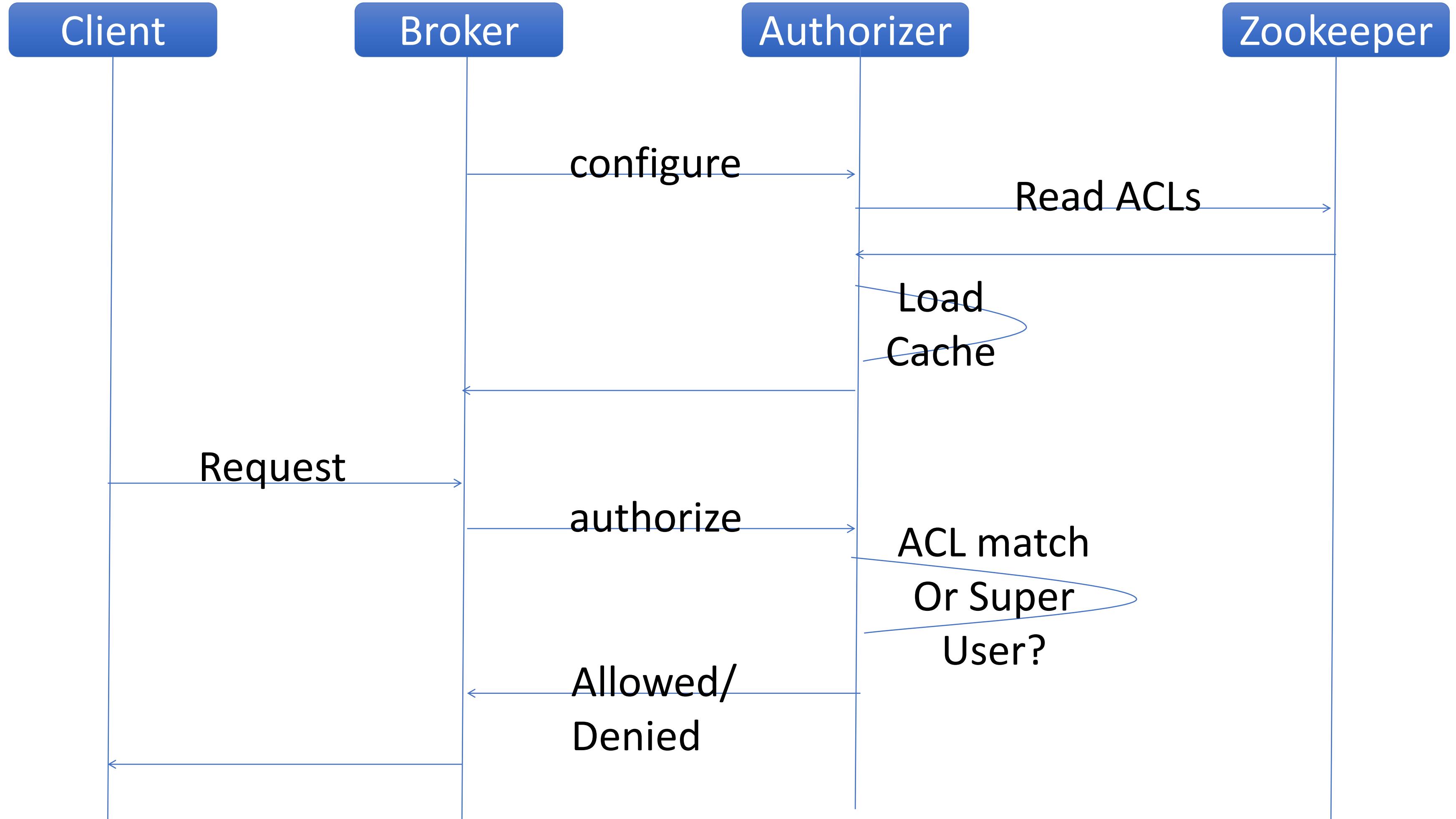
- Authorizer class
- Super users
- Authorizer properties
- Default behavior for resources with no ACLs

# SimpleAclAuthorizer

- Out of box authorizer implementation.
- Stores all of its ACLs in zookeeper.
- In built ACL cache to avoid performance penalty.
- Provides authorizer audit log.

# What about KRAFT mode?

- **StandardAuthorizer**
  - `org.apache.kafka.metadata.authorizer.StandardAuthorizer`
- `authorizer.class.name=org.apache.kafka.metadata.authorizer.StandardAuthorizer`



# CLI

- Add, Remove and List acls
- Convenience options:
  - producer and --consumer.

# Zookeeper

- Kafka's metadata store
- Has its own security mechanism that supports SASL and MD5-DIGEST for establishing identity and ACL based authorization
- Create , Delete directly interacts with zookeeper

# FAQs

- **How do you do end-to-end encryption in Kafka?**
  - Apache Kafka does not support end-to-end encryption natively, but you can achieve this by using a combination of TLS for network connections and disk encryption on the brokers.
- **Is data encrypted at rest in Kafka?**
  - By default, Apache Kafka data is not encrypted at rest. Encryption can be provided at the OS or disk level using third-party tools.

# FAQs

- **SASL Support in KAFKA**
  - Apache Kafka supports various SASL mechanisms, such as GSSAPI (Kerberos), OAUTHBEARER, SCRAM, LDAP, and PLAIN.
- **How do I set up ACLs in Kafka?**
  - To configure ACLs in Apache Kafka, first set the `authorizer.class.name` property in `server.properties`:  
`authorizer.class.name=kafka.security.authorizer.AclAuthorizer.`
  - Add and remove ACLs using the `kafka-acls.sh` script.

# FAQs

- **Does Kafka support Role-Based Access Control (RBAC)?**
  - Apache Kafka does not support Role-Based Access Control (RBAC) by default.
  - Confluent adds RBAC support to Kafka, allowing you to define group policies for accessing services (reading/writing to topics, accessing Schema Registry, etc.) and environments (dev/staging/prod, etc.), across all clusters.

# KAFKA Security Summary

- Apache Kafka Security Models
  - PLAINTEXT
  - SSL
  - SASL\_PLAINTEXT
  - SASL\_SSL
- Apache Kafka Security Models - A Ready Reckoner
- How to Troubleshoot security issues
- Most Common Errors
- Apache Kafka Security - Dos and Don'ts

# What Are We Securing?

- End-user Authentication (Is user who he/she claims to be?)
  - User Authorization (Does authenticated user have access to this resource?)
  - In-flight data i.e. Communication between
    - Kafka Broker <--> Kafka Clients (Consumer/Producers)
    - Kafka Broker <--> Kafka Broker
    - Kafka Broker <--> Zookeeper
- What Are We NOT Securing?
- Data persisted on-disk, e.g. security through data encryption

# Apache Kafka Security Models

## PLAINTEXT

- No Authentication / No Authorization / insecure channel => ZERO security
- Default security method
- To be used only for Proof-of-Concept
- Absolutely NOT recommended for use in Dev/Test/Prod environment

# Apache Kafka Security Models

## SSL

- X.509 Certificate based model - only secures the HTTP channel
- Performs certificate based host authorization
- No User Authentication / Authorization
- How to configure
  - Setup per-node certificate truststore/keystore for brokers & clients

Broker-side:	Client-side:
listeners=SSL://127.0.0.1:6667	security.protocol = SSL
inter.broker.protocol=SSL	

# Apache Kafka Security Models

## SASL\_PLAINTEXT (or PLAINTEXTSASL in older version)

- Supports user authentication via
  - Username / Password
  - GSSAPI (Kerberos Ticket)
  - SCRAM (Salted Password)
- Supports User authorization via Kafka ACLs or **Apache Ranger**
- Sends secrets & data over the wire in "**Plain**" format
- How to configure
  - Pre-configure authentication mechanism

Broker-side:	Client-side:
listeners=SASL_PLAINTEXT://127.0.0.1:6667	security.protocol = SASL_PLAINTEXT
inter.broker.protocol=SASL_PLAINTEXT sasl.enabled.mechanism=PLAIN   GSSAPI   SCRAM	sasl.mechanism = PLAIN   GSSAPI   SCRAM-SHA-256   SCRAM-SHA-512

# Apache Kafka Security Models

## SASL\_SSL

- Supports user authentication via
  - Username / Password
  - GSSAPI (Kerberos Ticket)
  - SCRAM (Salted Password)
- Supports User authorization via Kafka ACLs or **Apache Ranger**
- Sends secrets & data over the wire in "**Plain**" **Encrypted**
- How to configure
  - Pre-configure authentication mechanism
  - Setup per-node certificate truststore/keystore for broker(s) & client(s)

Broker-side:	Client-side:
listeners=SASL_SSL://127.0.0.1:6667	security.protocol = SASL_SSL
inter.broker.protocol=SASL_SSL sasl.enabled.mechanism=PLAIN   GSSAPI   SCRAM	sasl.mechanism = PLAIN   GSSAPI   SCRAM-SHA-256   SCRAM-SHA-512

# Apache Kafka Security Models - A Ready Reckoner

security.protocol	User Authentication	Authorization	Encryption Over Wire
PLAINTEXT	✗	✗	✗
SSL	✗	Host Based (via SSL certificates)	✓
SASL_PLAINTEXT	PLAIN   KRB5   SCRAM	Kafka ACLs / Ranger	✗
SASL_SSL	PLAIN   KRB5   SCRAM	Kafka ACLs / Ranger	✓

\* Available in Apache Kafka 0.9.0 and above

# How to troubleshoot security issues?

- Enable Krb debug for SASL clients (consumer/producer)
  - `export KAFKA_OPTS="-Dsasl.security.krb5.debug=true"`
- Enable SSL debug for clients
  - `export KAFKA_OPTS="-Djavax.net.debug=ssl"`
- Enable Krb / SSL debug for Kafka Broker (**AMBARI-24151**)
  - Enable this in console as 'kafka' user & start the Broker from command line:
    - `export KAFKA_KERBEROS_PARAMS="$KAFKA_KERBEROS_PARAMS -Dsasl.security.krb5.debug=true -Djavax.net.debug=ssl"`
    - `/usr/hdp/current/kafka-broker/bin/kafka-server-start.sh -daemon /etc/kafka/conf/server.properties`

\* Disable debug properties once you are done with troubleshooting, otherwise it's going to bloat the log files

# How to troubleshoot security issues?

- Enable Kafka Broker log4j debug
  - Set `log4j.logger.kafka=DEBUG, kafkaAppender` in `/etc/kafka/conf/log4j.properties`
- Enable Kafka Ranger log4j debug
  - Set `log4j.logger.org.apache.ranger=DEBUG, rangerAppender` in `/etc/kafka/conf/log4j.properties`
- Enable Kafka Client debug
  - Set `log4j.rootLogger=DEBUG, stderr` in `/etc/kafka/conf/tools-log4j.properties`

\* Disable debug properties once you are done with troubleshooting, otherwise it's going to bloat the log files

# How does Kerberos debug messages look like in Apache Kafka logs

```
[2018-06-21 08:43:04,581] INFO Waiting for keeper state SaslAuthenticated (org.I0Itec.zkclient.ZkClient$1@103434)
>>> KeyTabInputStream, readName(): LAB.HORTONWORKS.NET
>>> KeyTabInputStream, readName(): kafka
>>> KeyTabInputStream, readName(): bali2.openstacklocal
>>> KeyTab: load() entry length: 81; type: 23
>>> KeyTabInputStream, readName(): LAB.HORTONWORKS.NET
>>> KeyTabInputStream, readName(): kafka
>>> KeyTabInputStream, readName(): bali2.openstacklocal
>>> KeyTab: load() entry length: 73; type: 3
>>> KeyTabInputStream, readName(): LAB.HORTONWORKS.NET
>>> KeyTabInputStream, readName(): kafka
>>> KeyTabInputStream, readName(): bali2.openstacklocal
>>> KeyTab: load() entry length: 97; type: 18
>>> KeyTabInputStream, readName(): LAB.HORTONWORKS.NET
>>> KeyTabInputStream, readName(): kafka
>>> KeyTabInputStream, readName(): bali2.openstacklocal
>>> KeyTab: load() entry length: 81; type: 17
>>> KeyTabInputStream, readName(): LAB.HORTONWORKS.NET
>>> KeyTabInputStream, readName(): kafka
>>> KeyTabInputStream, readName(): bali2.openstacklocal
>>> KeyTab: load() entry length: 89; type: 16
Looking for keys for: kafka/bali2.openstacklocal@LAB.HORTONWORKS.NET
Java config name: null
Native config name: /etc/krb5.conf
Loaded from native config
Added key: 16version: 0
Added key: 17version: 0
Added key: 18version: 0
Found unsupported keytype (3) for kafka/bali2.openstacklocal@LAB.HORTONWORKS.NET
Added key: 23version: 0
>>> KdcAccessibility: reset
Looking for keys for: kafka/bali2.openstacklocal@LAB.HORTONWORKS.NET
Added key: 16version: 0
Added key: 17version: 0
Added key: 18version: 0
Found unsupported keytype (3) for kafka/bali2.openstacklocal@LAB.HORTONWORKS.NET
Added key: 23version: 0
Using builtin default etypes for default_tkt_enctypes
```

# Troubleshoot Using Kafka Console Consumer/Producer

- Create a Kafka topic
  - Should be run only on Kafka Broker node as 'kafka' user
- Use Kafka Console Producer to write messages to above Kafka topic
  - Can be run from any Kafka client node as any user
  - Make sure that authentication token is acquired and user has permission to 'Describe' & 'Publish'
- Use Kafka Console Consumer to read messages from the Kafka topic
  - Can be run from another or same Kafka client as the same or different user
  - Make sure that authentication token is acquired and user has permission to 'Consume'

# Most Common Errors

- **javax.security.auth.login.LoginException**
  - Check JAAS configuration
- **Could not login: the client is being asked for a password**
  - Again, issue with JAAS configuration - either Ticket not found or Bad / inaccessible user keytab
- **PKIX path building failed - unable to find valid certification path to requested target**
  - Issue with SSL truststore; most likely truststore not present or readable
- **No User Authentication / Authorization**

# Apache Kafka Security - Dos and Don'ts

- No Kerberos = No Security
  - All the pain is well worth it !
- Enabling SSL is only half the story
  - Having SSL without Authentication is meaningless
- Using any SASL (i.e. Authentication) without SSL is dangerous
- Use Apache Ranger for large deployments with many users

# Back to KAFKA security

- We have SSL traffic and JAAS configuration.
- We do not yet have mTLS (mutual TLS)
- Here clients pass Certificates to the broker and both sides verify each other (client verifies broker and broker verifies Client)
-

# Kafka Security

- Each Client should generate their own certificates

- **Client.properties**

*ssl.keystore.location=/path/to/client.keystore.jks*

*ssl.keystore.password=client-keystore-pass*

*ssl.key.password=client-key-pass*

- On the broker side, client authentication is required.

*ssl.client.auth=required*

- Optionally sign both broker and client certs via a shared CA for centralized trust.

# Kafka Security

- In TLS, only server authenticates itself with the client
- In MTLS, both server and client authenticate each other.
- This may seem an overkill – if so read this.
- <https://www.thehindubusinessline.com/info-tech/data-breach-at-pine-labs-exposes-500000-records/article35962296.ece>
- Whenever sensitive information esp. in finance domain is concerned this is a must

# KAFKA Security

- One way to secure KAFKA end to end with MTLS (almost end to end ☺) is to use 3<sup>rd</sup> party tools/components.
- This is needed as certificates can and will expire on both ends.
- Hashicorp Vault is one way of doing this. We are not covering this because

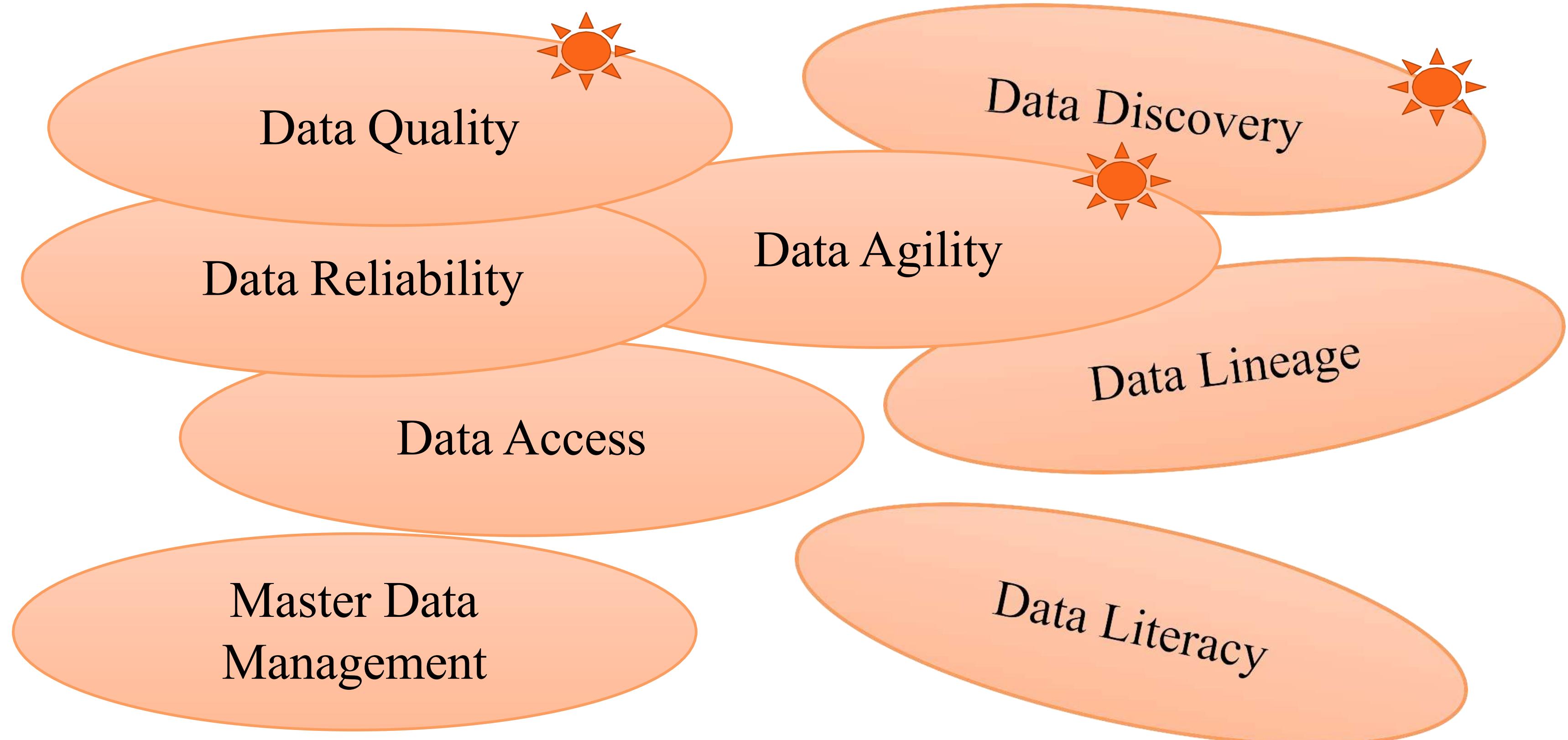
# KAFKA Security

- Workflows will change (Esp. with Certificate Revocation Lists) e.g.
  - Change KAFKA\_HOSTS to include
    - -Dcom.sun.security.enableCRLDP=true  
Dcom.sun.net.ssl.checkRevocation=true
- Components in your SW and infra architecture will change
- How you deploy software (+updates) will change

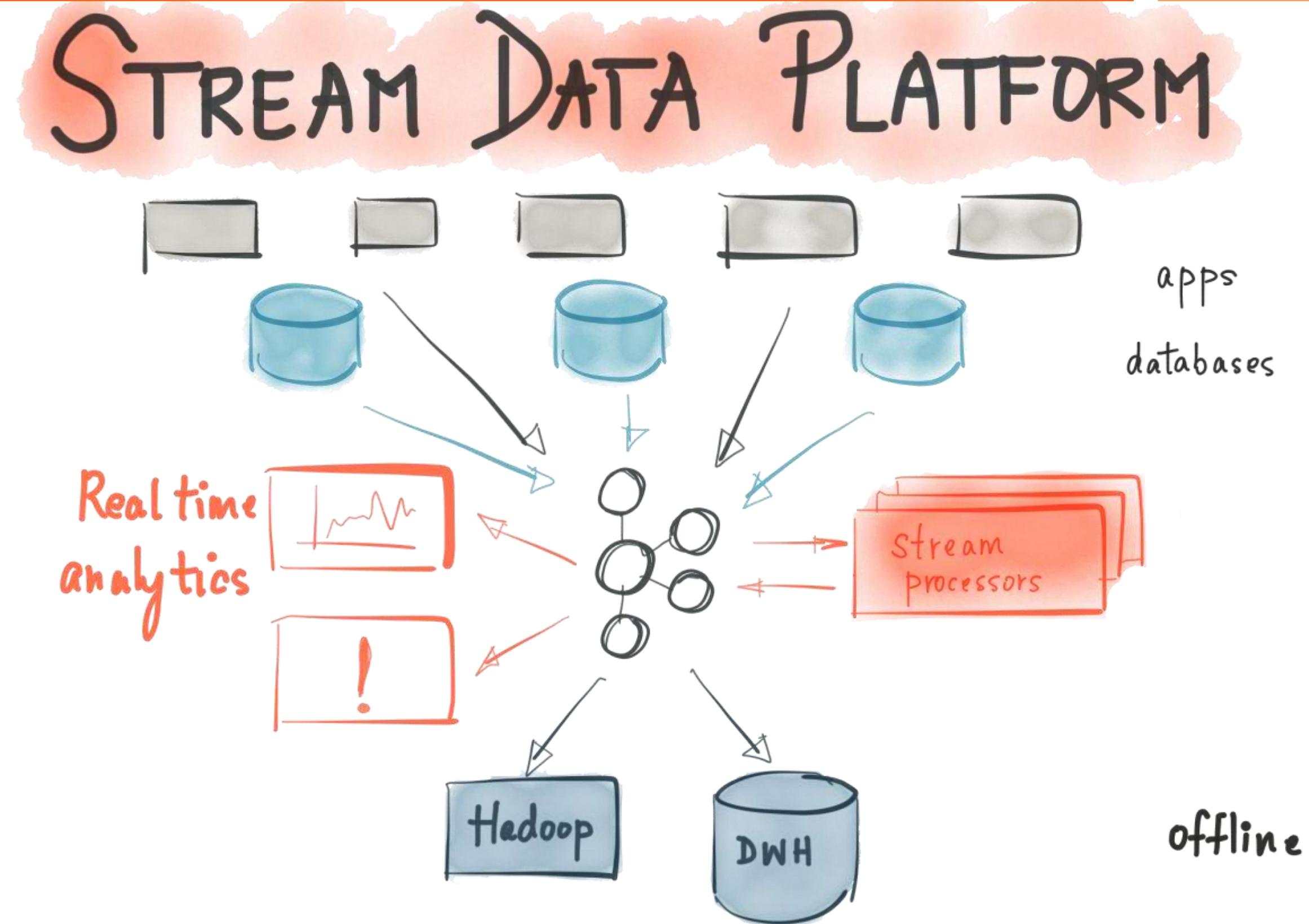
# Some Notes on Data Governance

# What is Data Governance?

---



# To Grow a Successful Data Platform, You need Some Guarantees



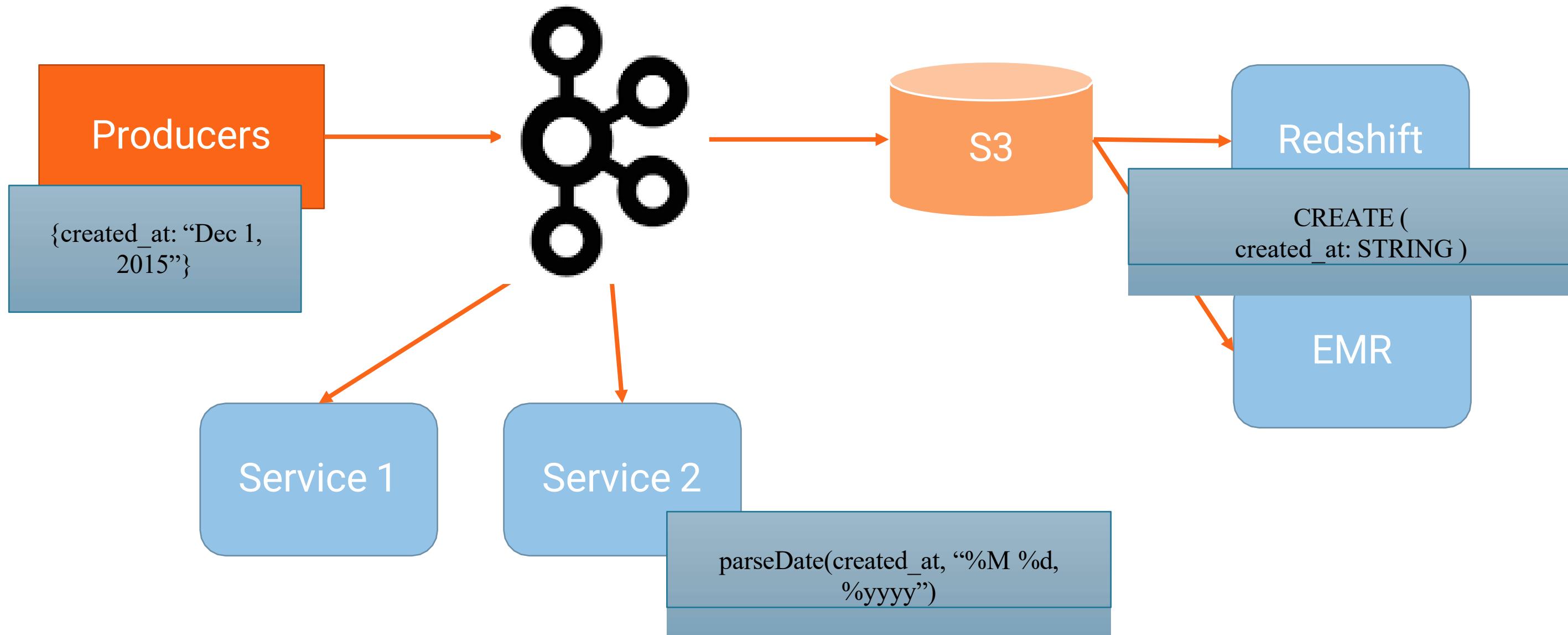
## Schemas Are a Key Part of This

---

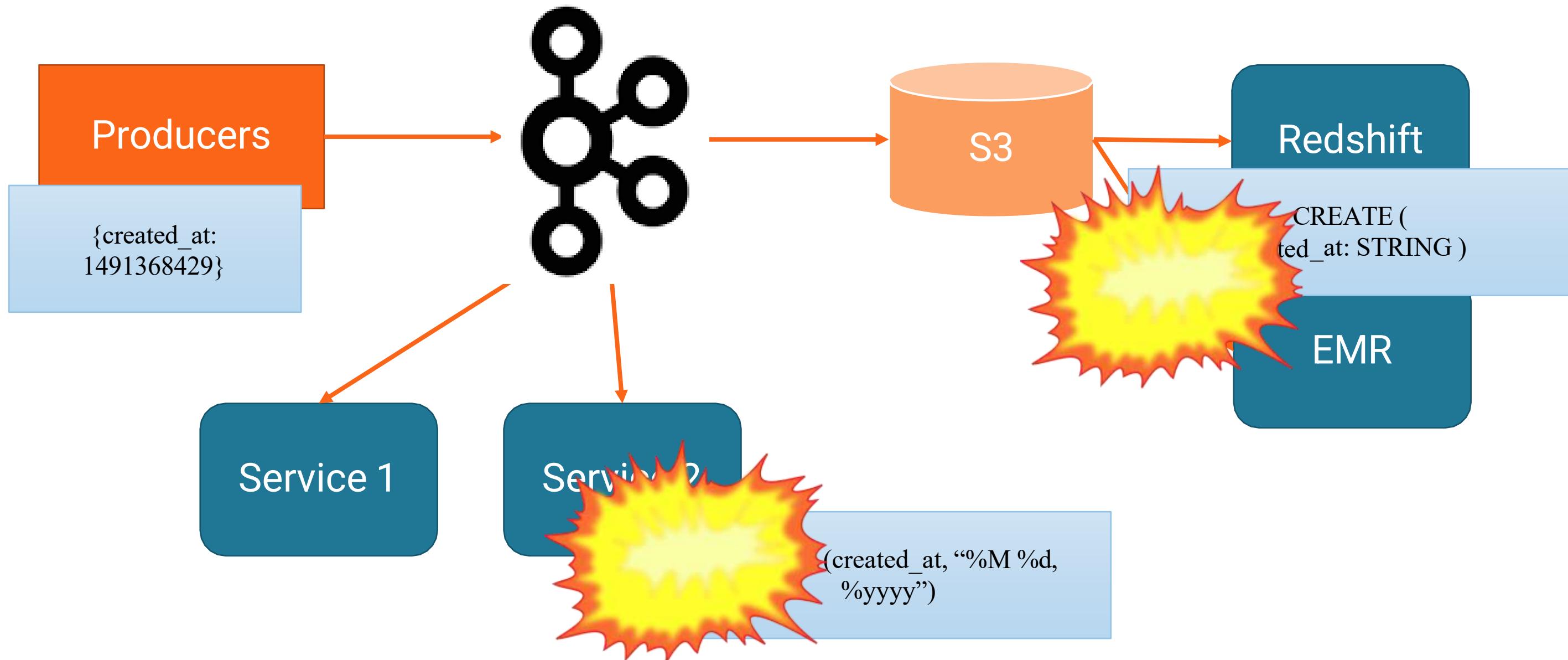
1. Schemas: Enforced Contracts Between your Components
2. Schemas need to be agile but compatible
3. You need tools & process that will let you:
  - Move fast and not break things



# Cautionary tale: Uber

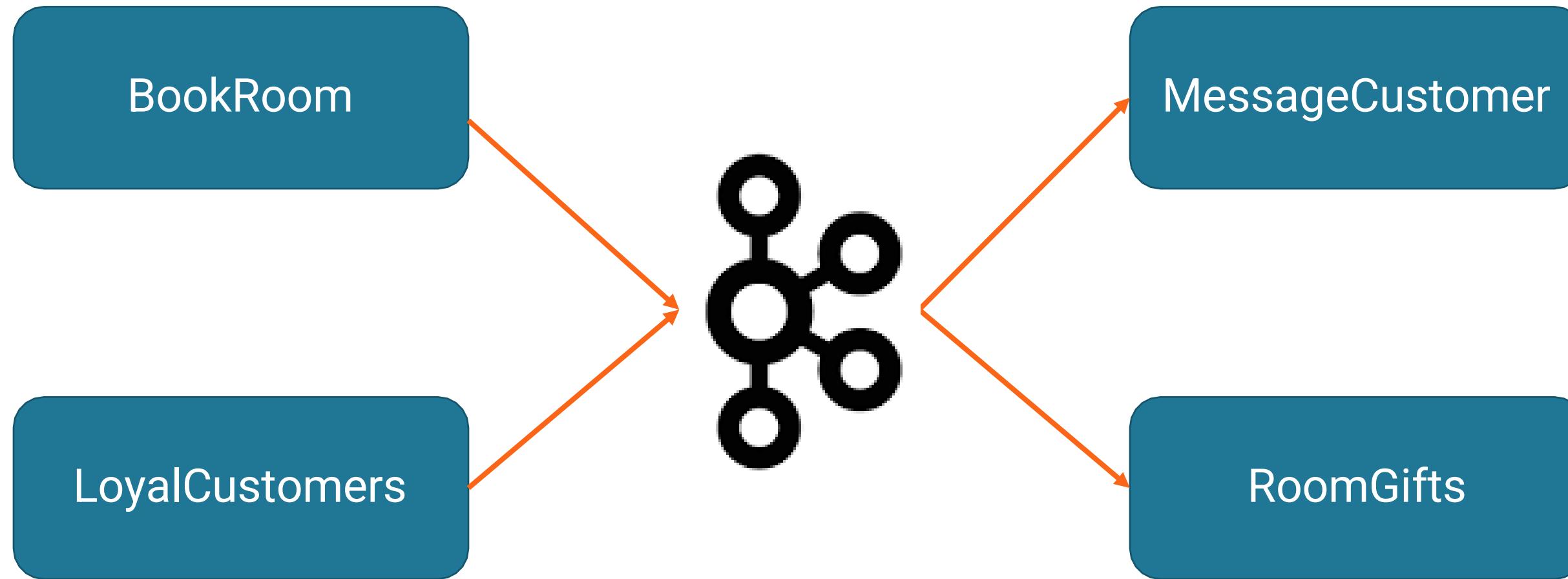


# Cautionary tale: Uber

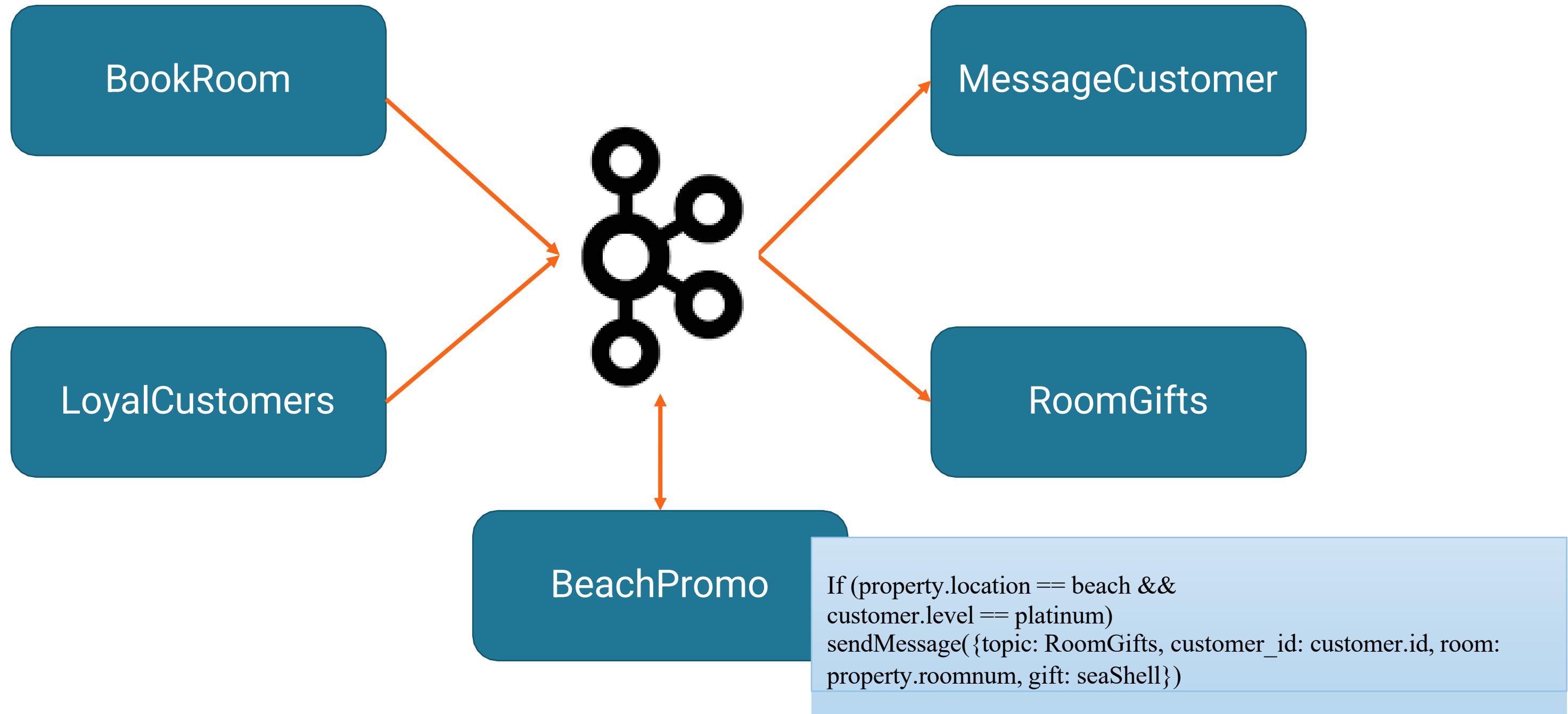


# The right way to do things

---



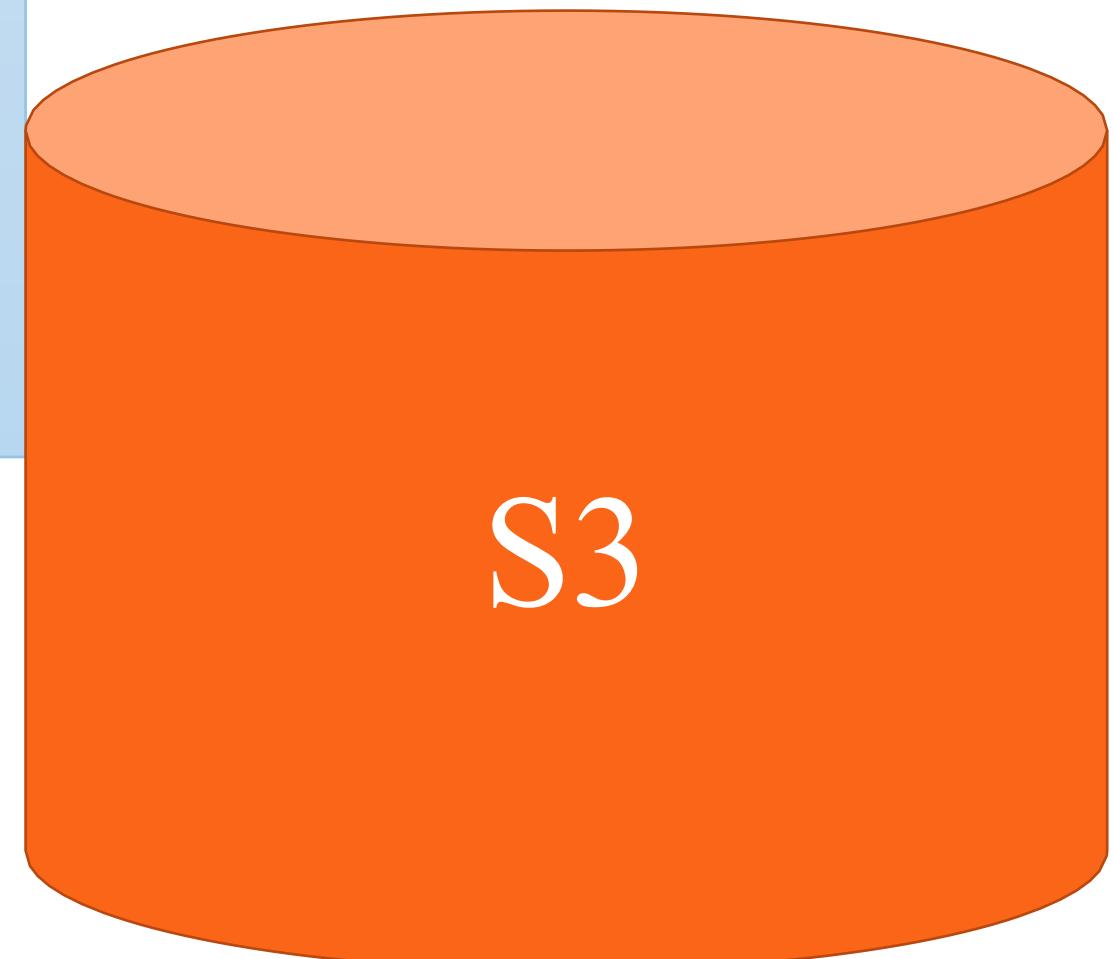
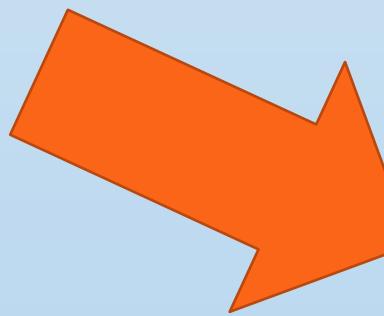
# The right way to do things



# There is special value in standardizing parts of schema

```
{ "type": "record",
  "name": "Event",
  "fields": [ { "name": "headers", "type": {
      "name" : "headers",
      "type": "record"
      "fields": [
        {"name": "source_app", "type": "string"},
        {"name": "host_app", "type": "string"},
        {"name": "destination_app", "type": ["null","string"]},
        {"name": "timestamp", "type": "long"}
      ]
    },
    { "name": "body", "type": "bytes" } ] }
```

Standard headers allowed  
analyzing patterns of communication



S3

---

In Request Response World – APIs between services are key

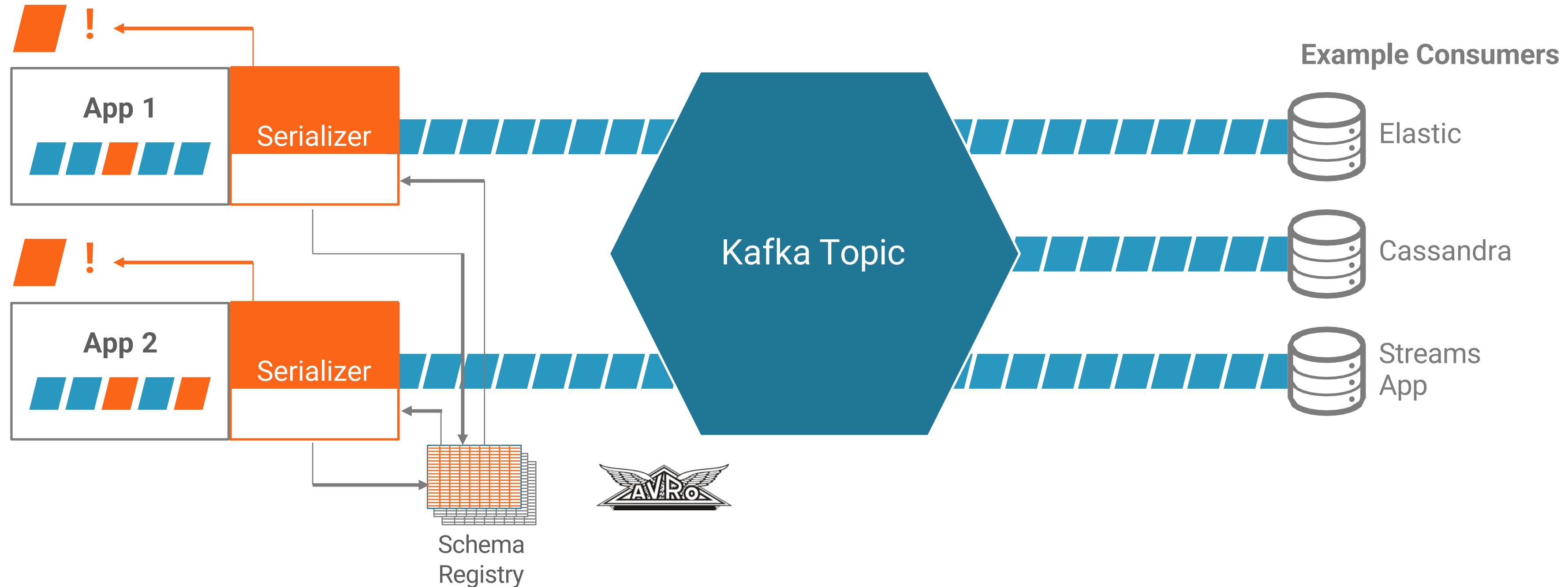
In Async Stream Processing World – Message Schemas ARE the API

...except they stick around for a lot longer

Lets assume you see why you need  
schema compatibility...

How do we make it work for us?

# Schema Registry



Define the expected fields for each Kafka topic

Automatically handle schema changes (e.g. new fields)

Prevent incompatible changes

Supports multi-datacenter environments

## Key Points:

---

- Schema Registry is efficient – due to caching
- Schema Registry is transparent
- Schema Registry is Highly Available
- Schema Registry prevents bad data at the source

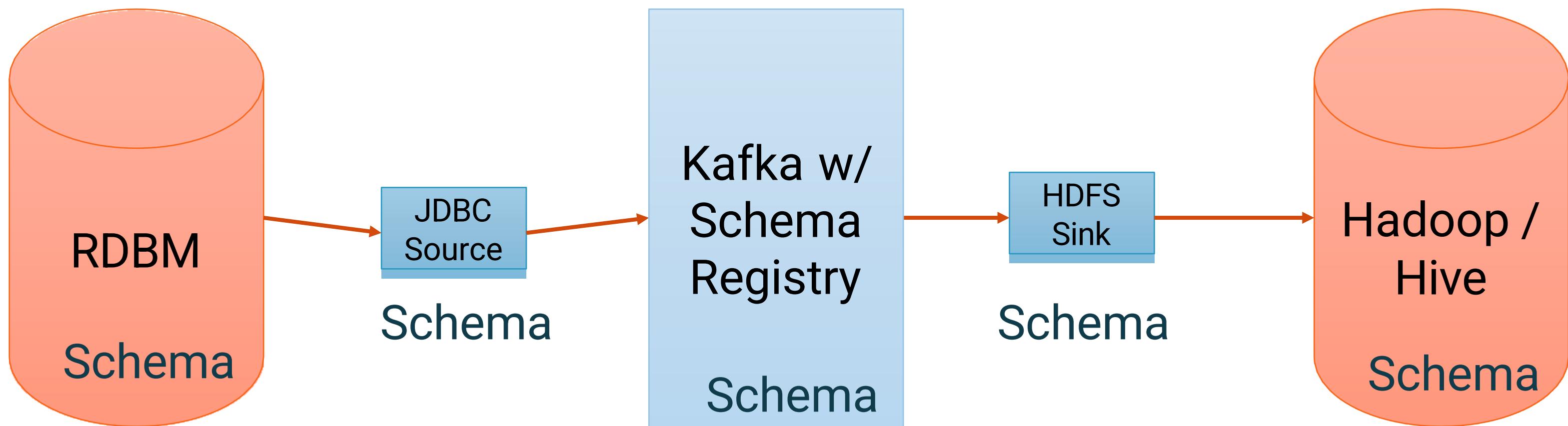
# Big Benefits

---

- Schema Management is part of your entire app lifecycle – from dev to prod
- Single source of truth for all apps and data stores
- No more “Bad Data” randomly breaking things
- Increased agility! add, remove and modify fields with confidence
- Teams can share and discover data
- Smaller messages on the wire and in storage

# But the Best Part is...

---

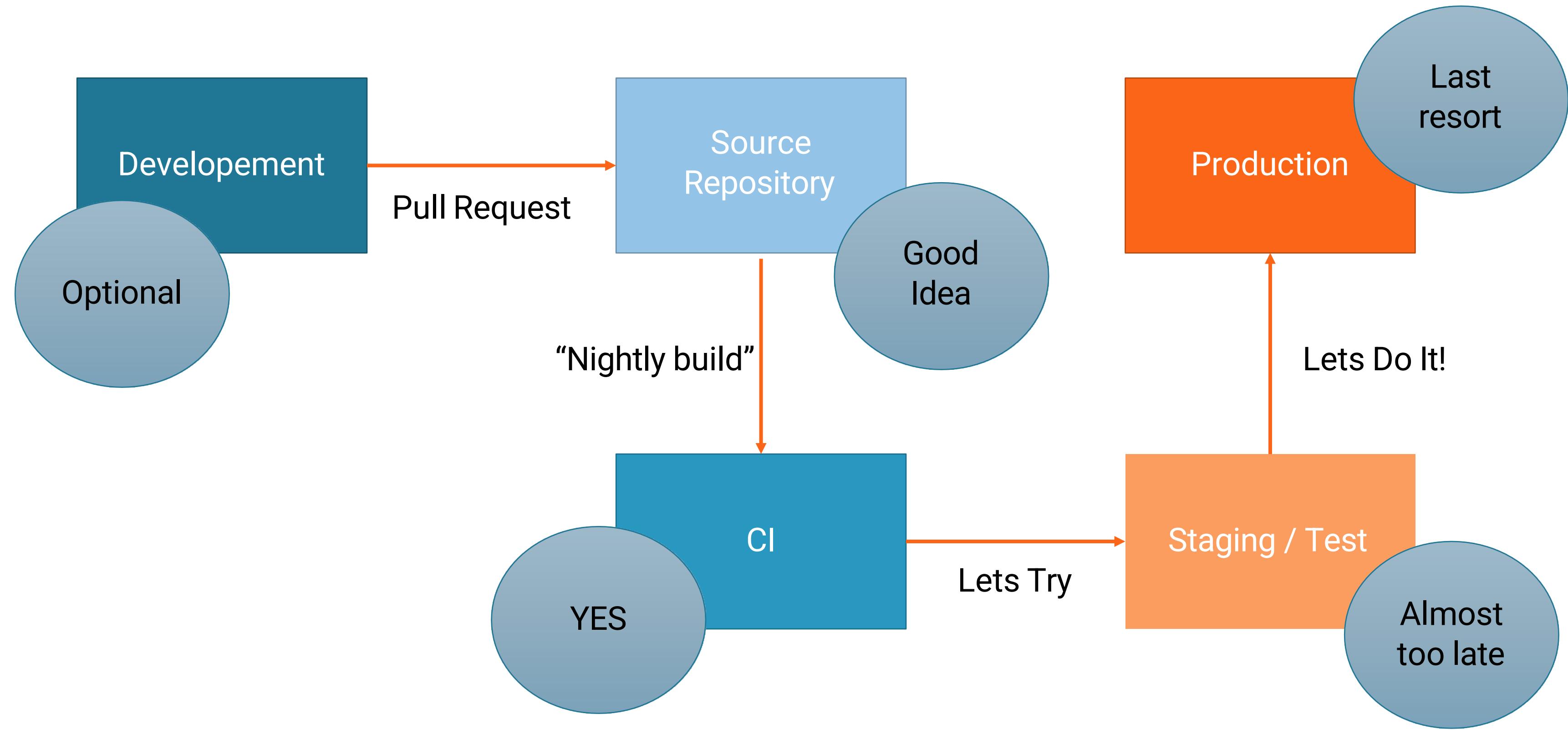


# Compatibility tips!

---

- **Do you want to upgrade producer without touching consumers?**
  - You want *\*forward\** compatibility.
  - You can add fields.
  - You can delete fields with defaults
- **Do you want to update schema in a DWH but still read old messages?**
  - You want *\*backward\** compatibility
  - You can delete fields
  - You can add fields with defaults
- **Both?**
  - You want *\*full\** compatibility.
  - Default all things!
  - Except the primary key which you never touch.

# App Lifecycle



# Data durability

Kafka is not waiting  
for a disk flush by  
default.

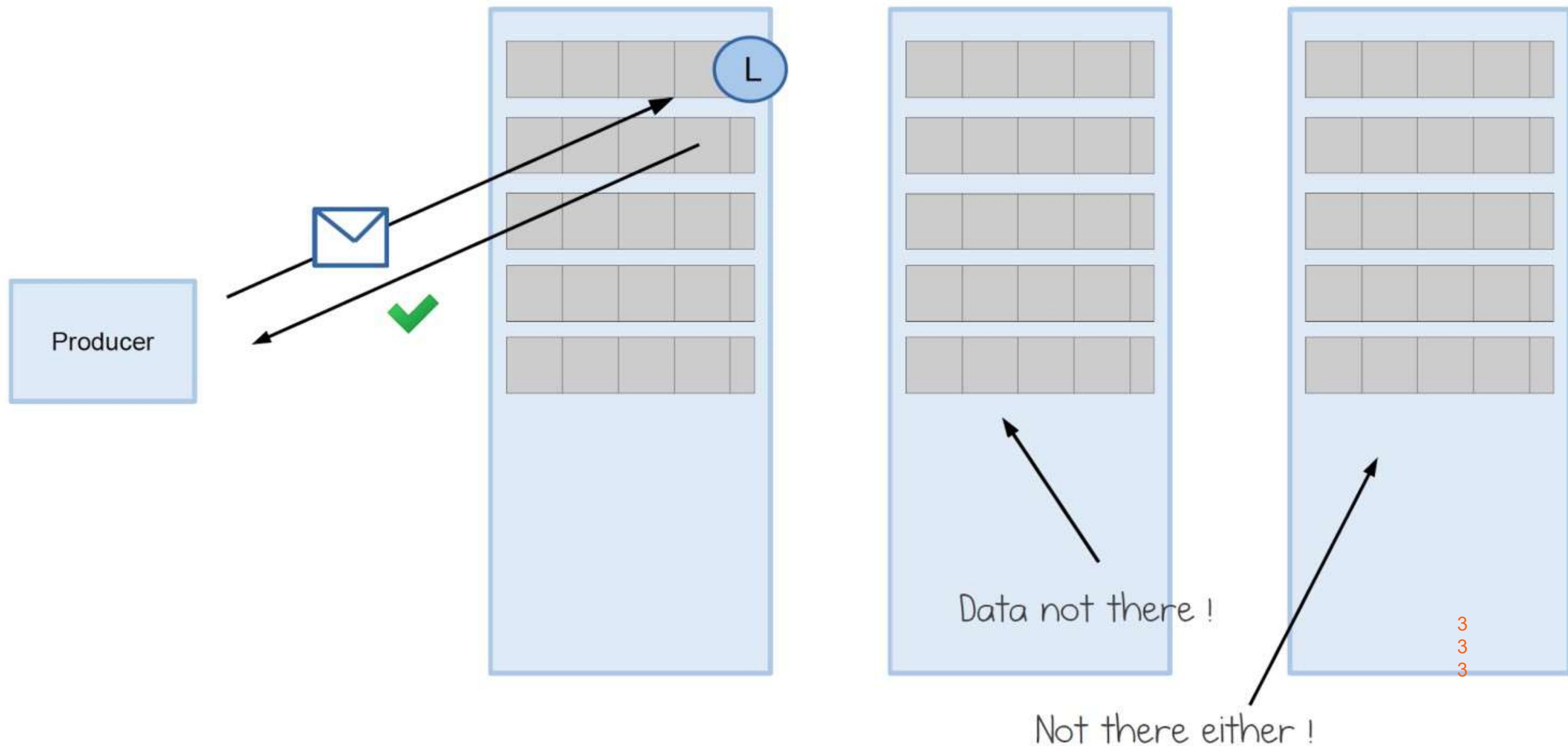
Durability is achieved  
through replication.

# Data durability

Is my data safe?

It depends on your configuration...

As soon as it is on the page cache of  
the leader...

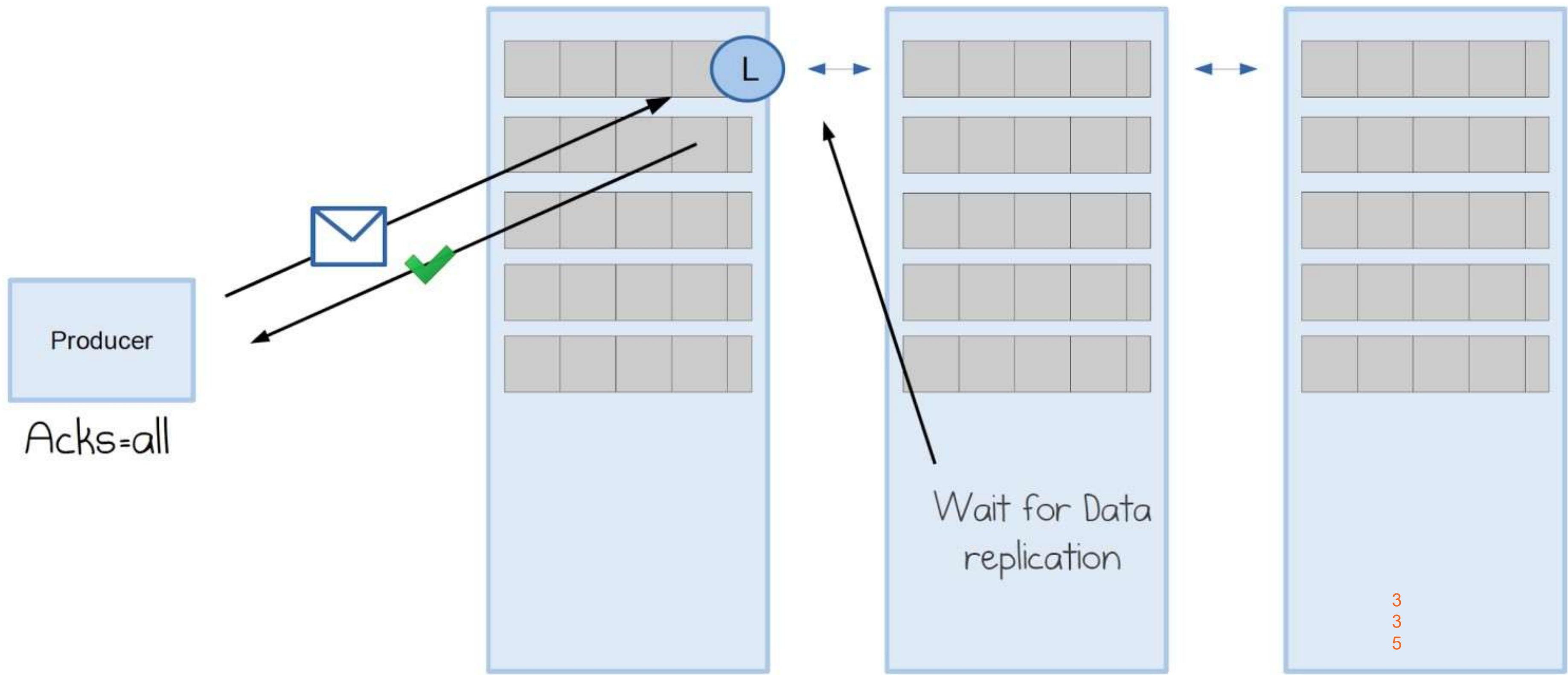


# Data durability

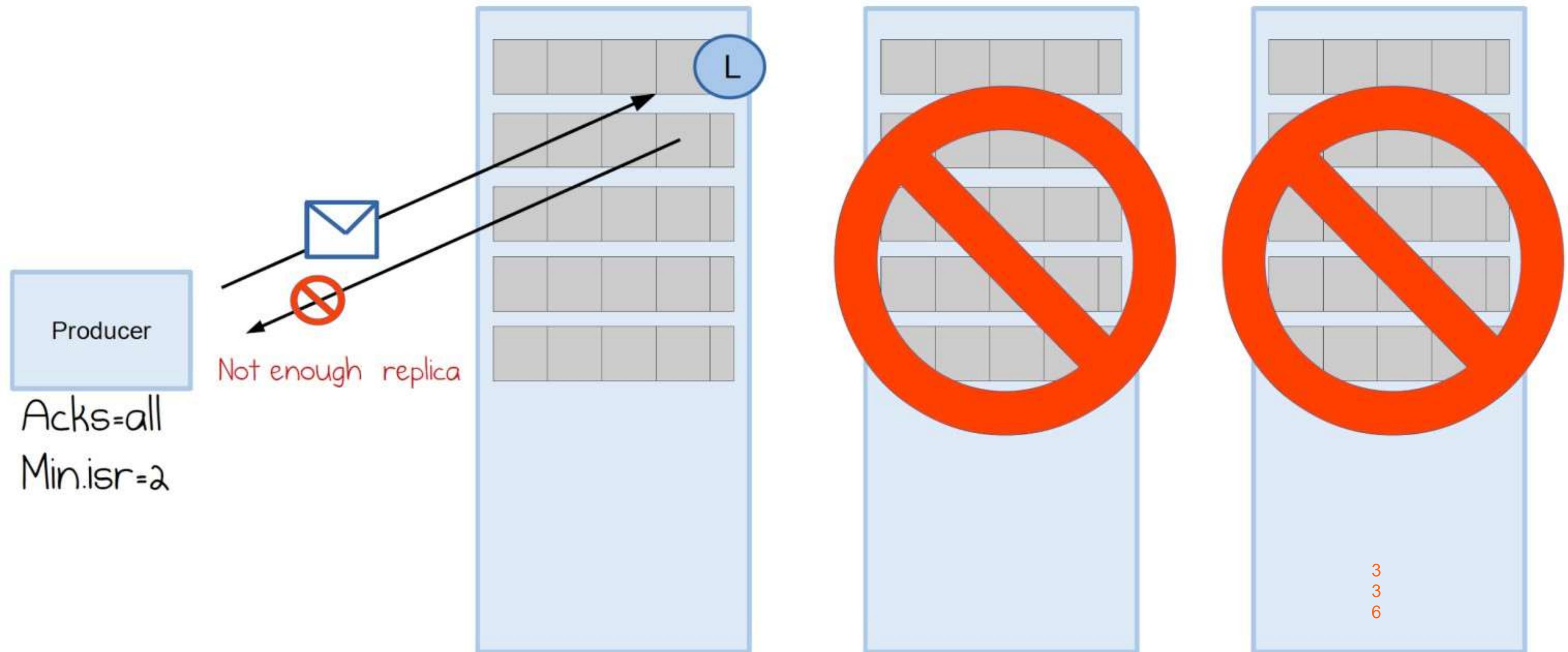
acks=1 (default value)  
good for **latency**

acks=all  
good for **durability**

# Replication before acknowledging



... which could be only one server



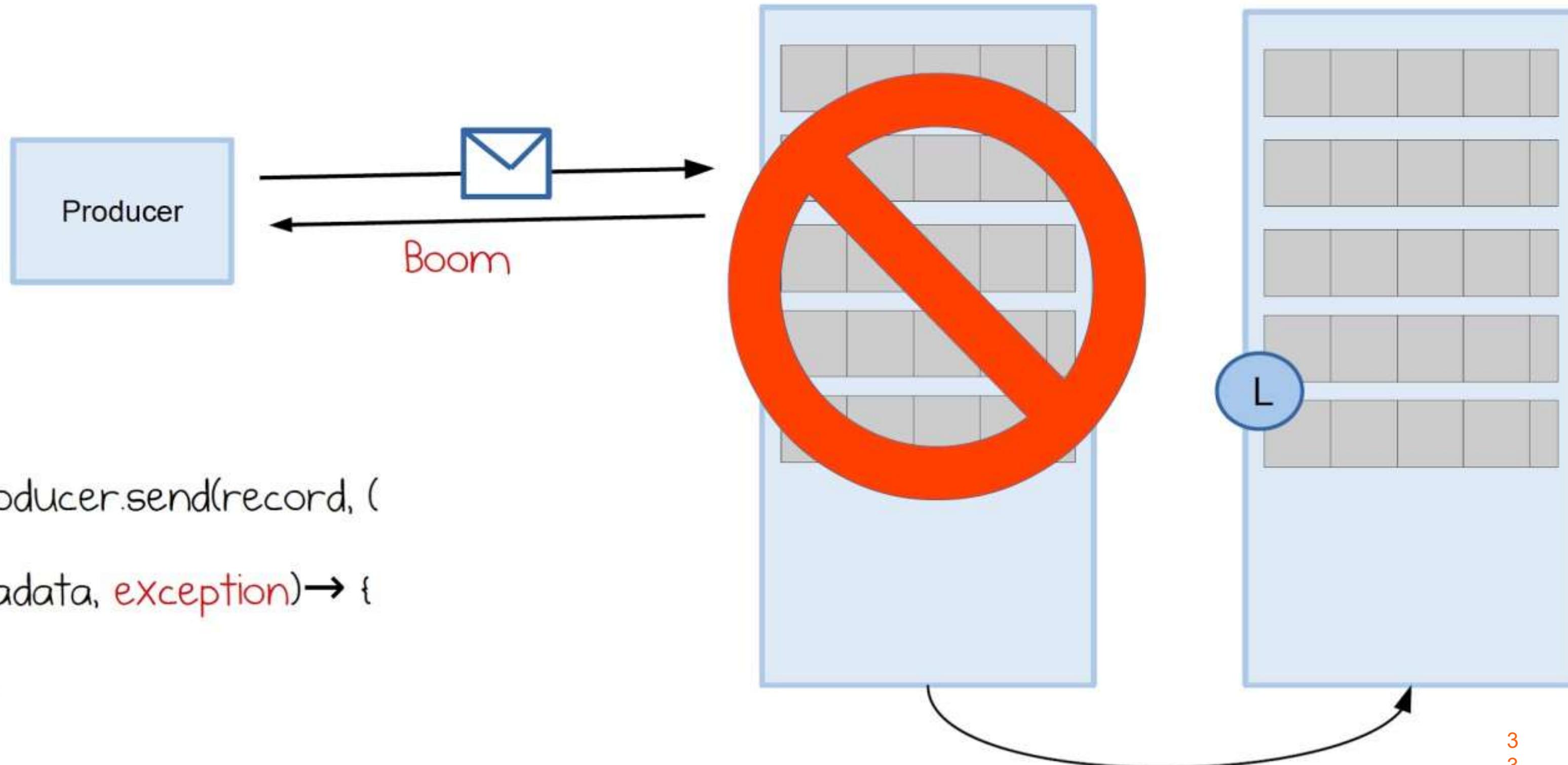
# defaults

The default values are optimized for availability & latency.

If durability is more important, tune it!

# What's happening in case of issue ?

```
kafkaProducer.send(record, (metadata, exception) → {  
    ...  
})  
)
```



The leader moved to a  
different broker

Parameter:  
**retries**

It will cause the client to resend any record whose send fails with a potentially transient error.

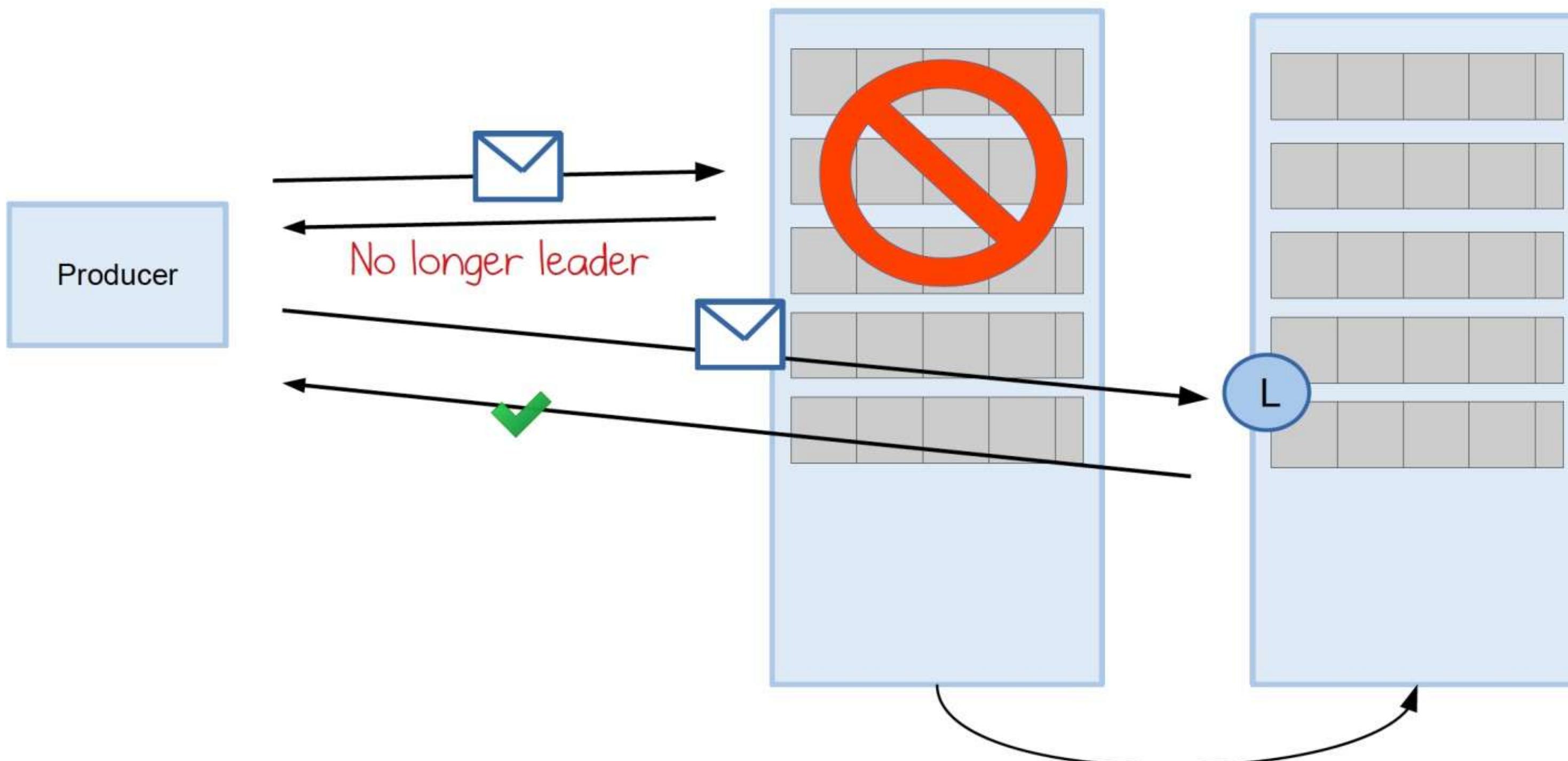
Default value : 0

Parameter:  
**retries**

It will cause the client to resend any record whose send fails with a potentially transient error.

Default value : 0

# What's happening in case of issue with retry ?



The leader moved to a  
different broker

3  
4  
1

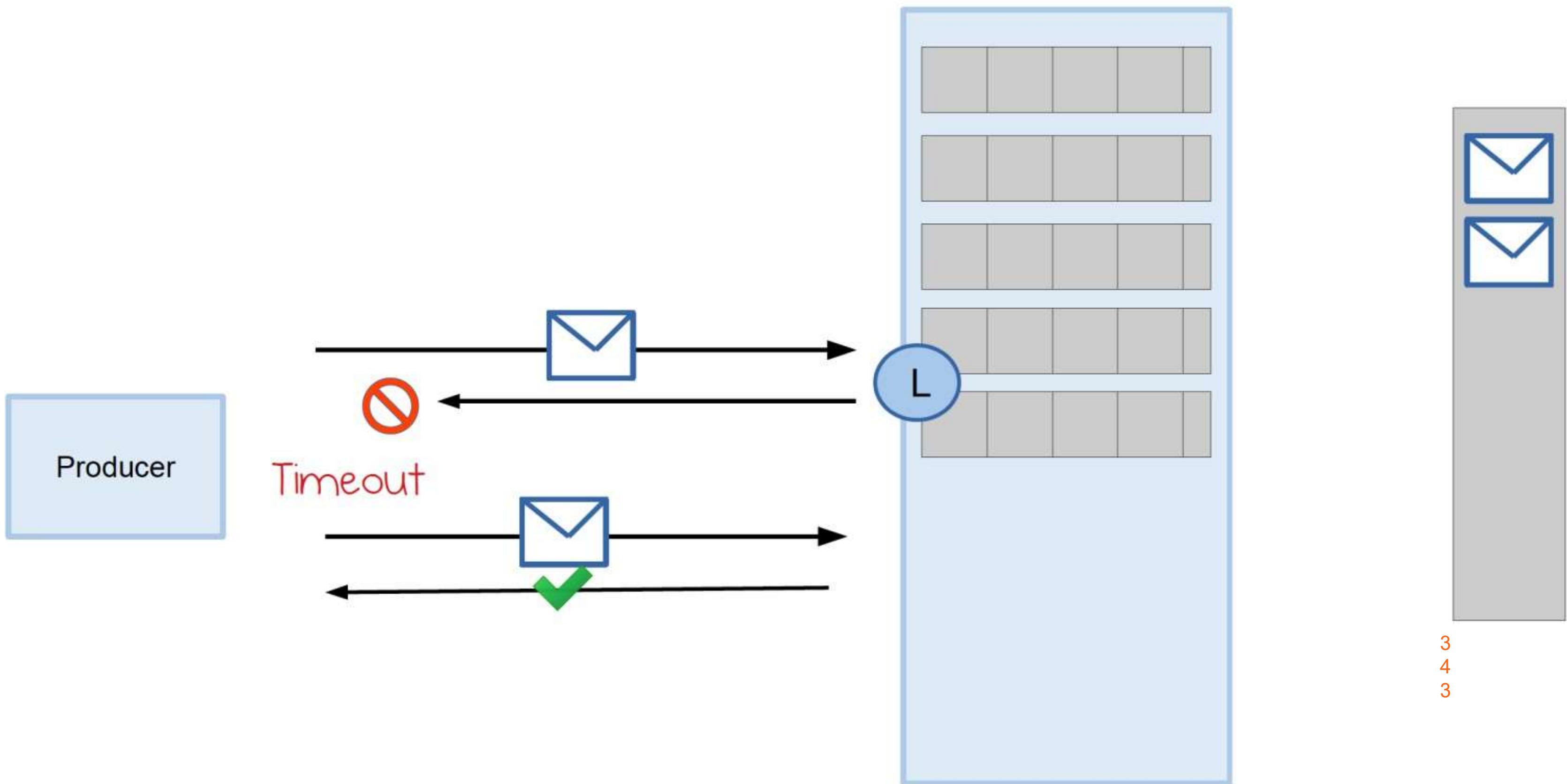
Parameter:

**retries**

Use built in retries !

Bump it from 0 to **infinity!**

# Message duplication

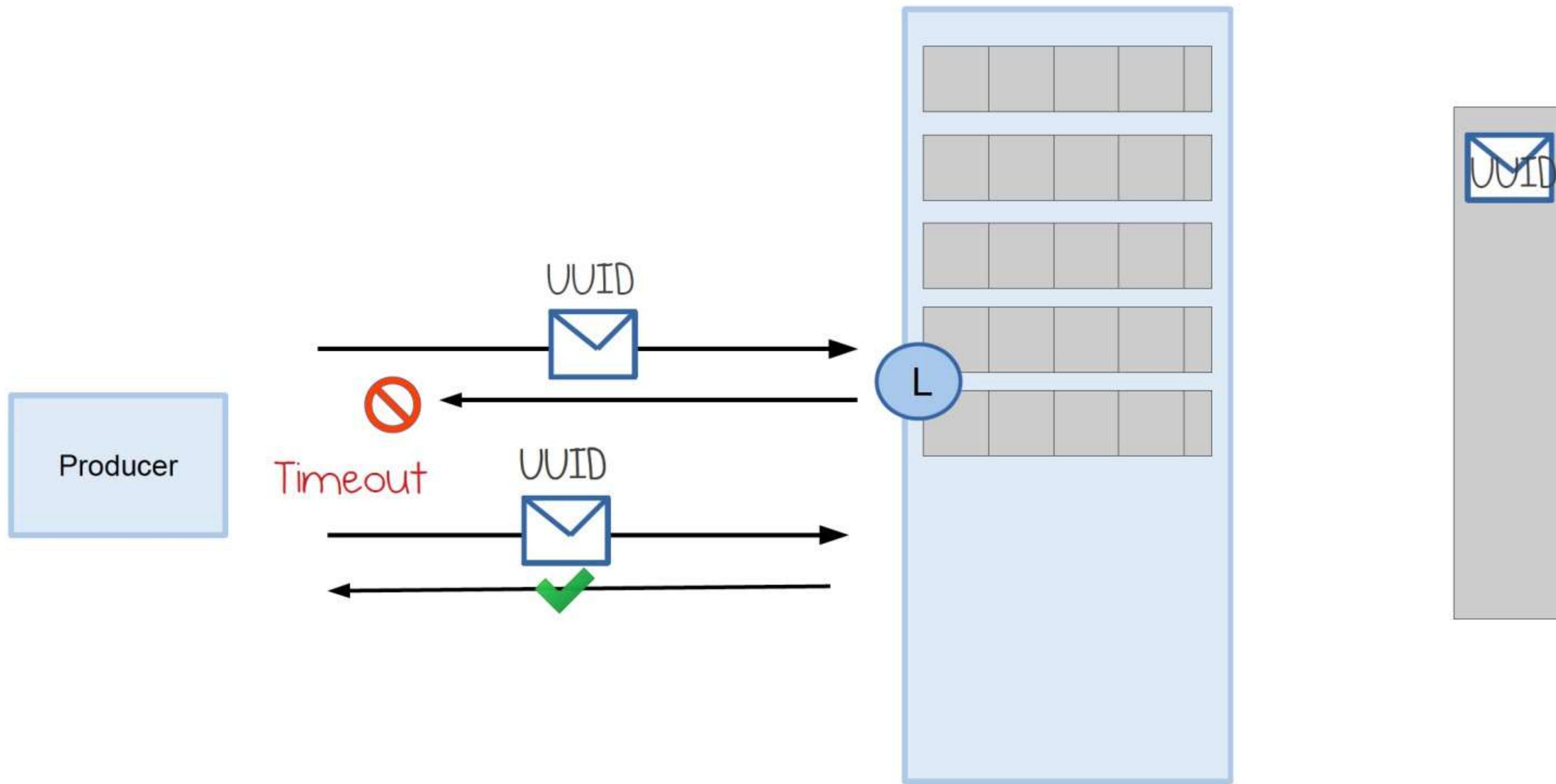


Parameter:  
**enable.idempotence**

When set to 'true',  
the producer will ensure  
that exactly  
**one** copy of each  
message is written.

**Default value: false**

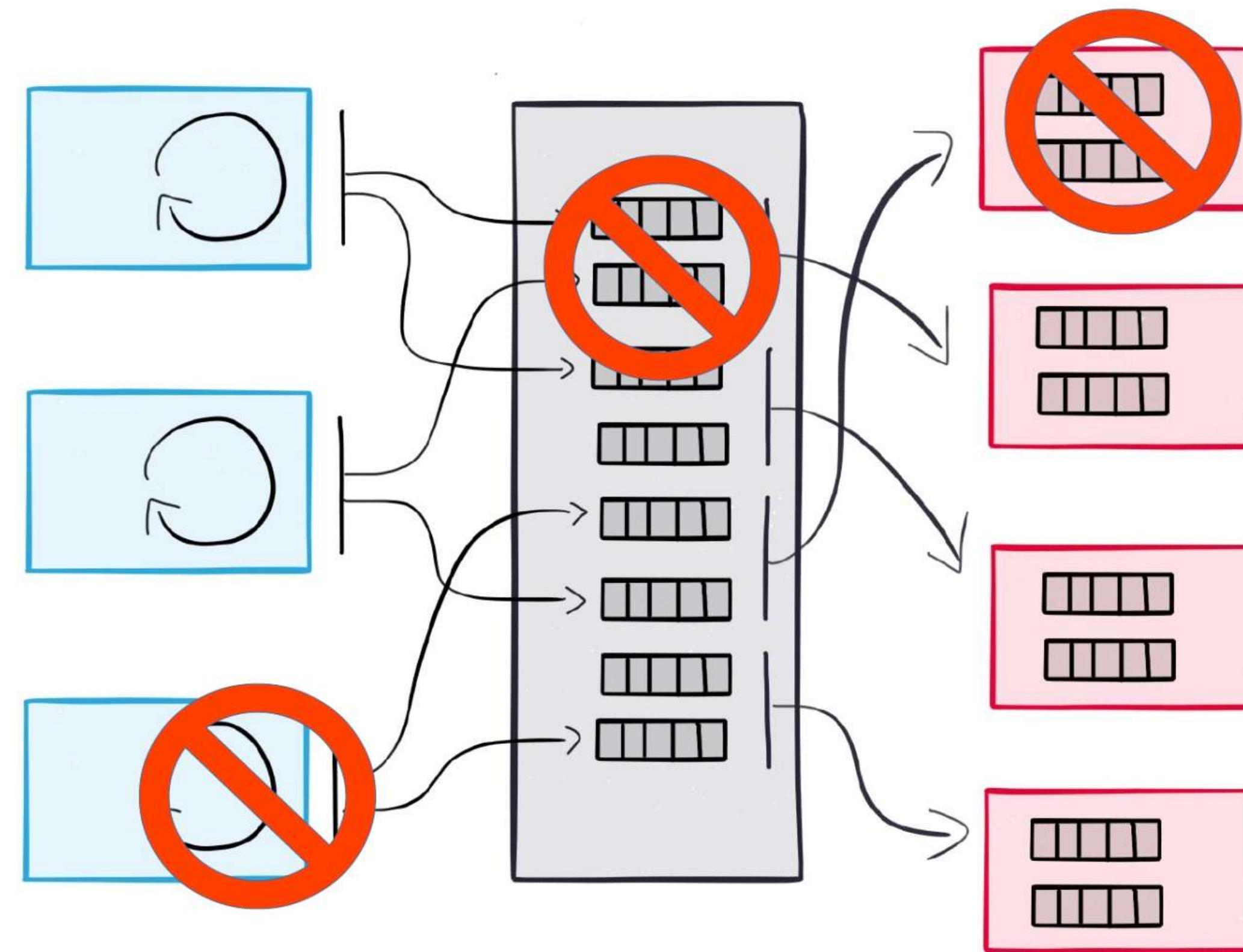
# With Retries and Idempotency



# What to do in case of an error ?

Producer

Consumer

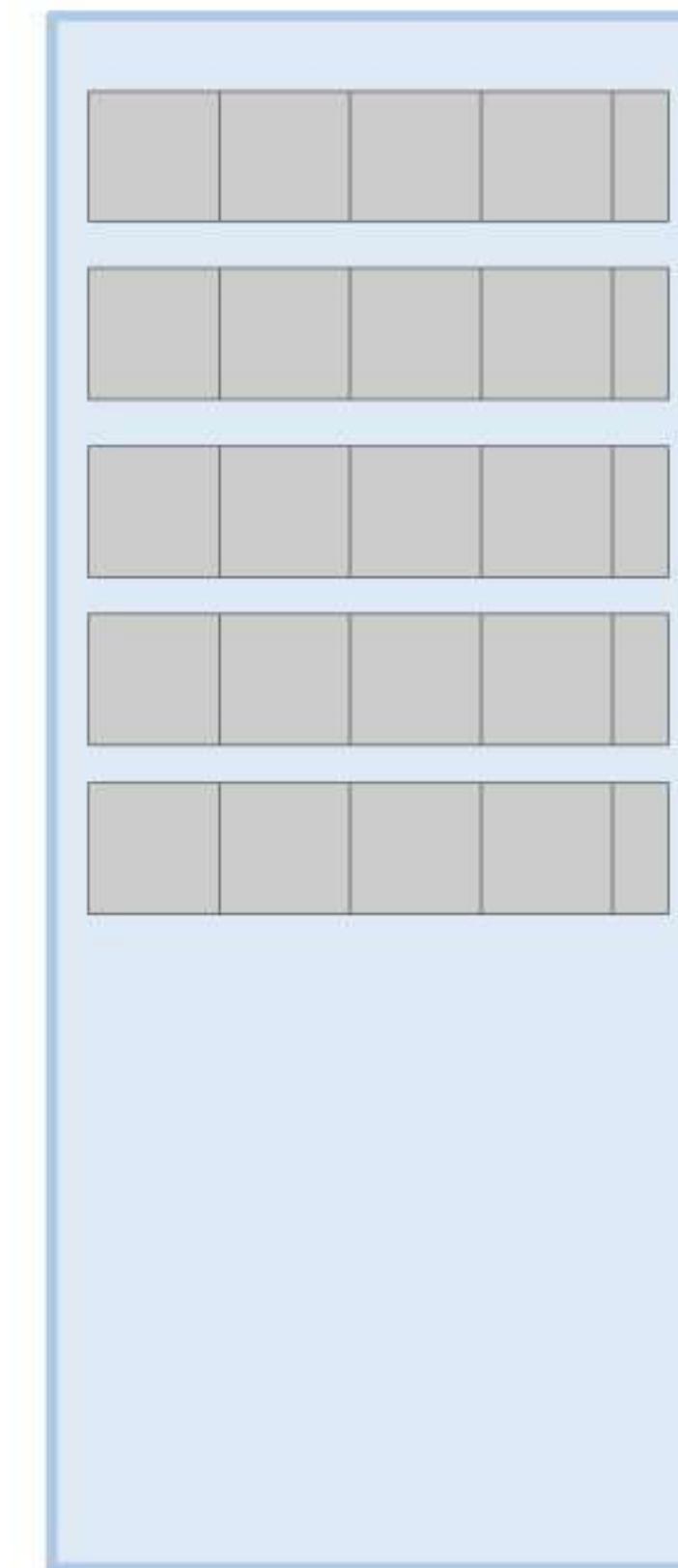


Infinite Retries

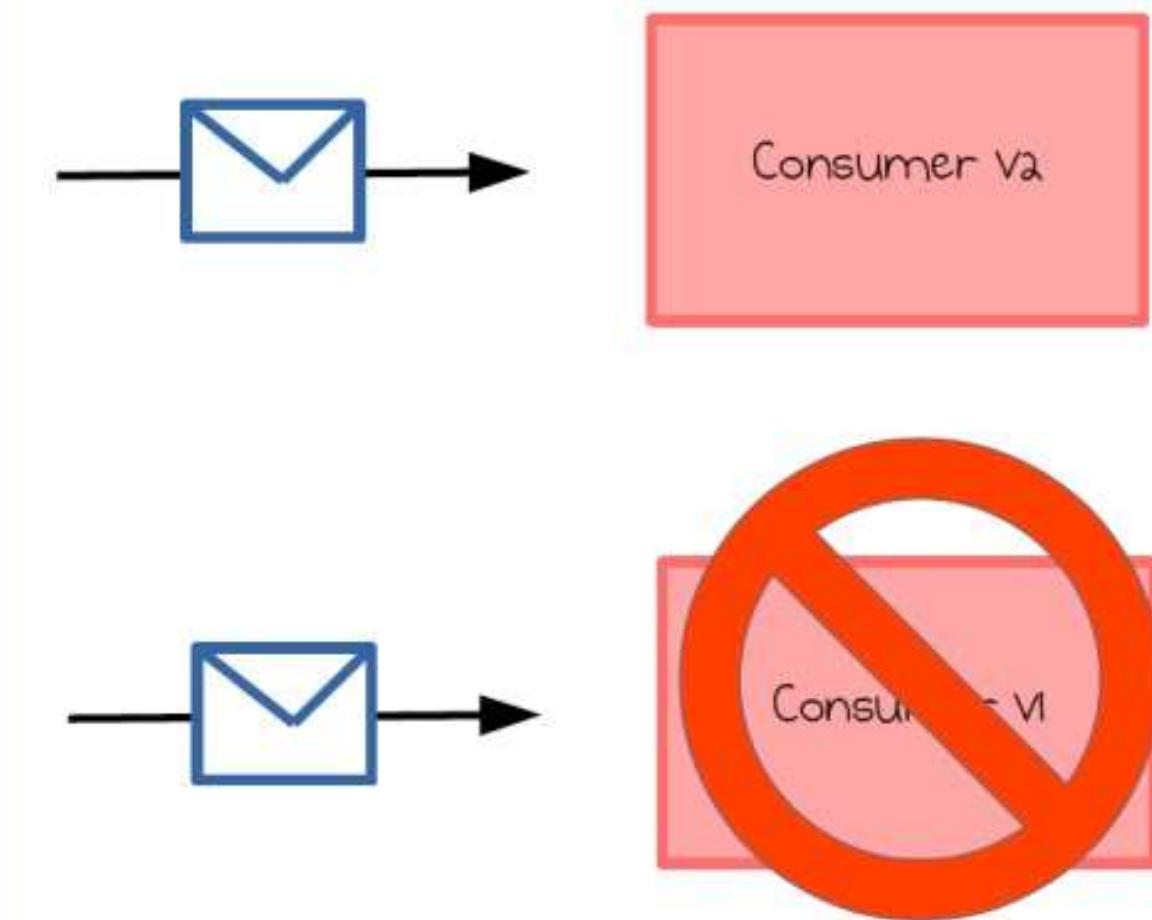
Write to DLQ and continue

Ignore and continue

Application I



```
{  
    Name : « Stan Lee »  
    DateOfBirth : Timestamp(...)  
}
```



Expected String got a  
Timestamp instead !

# governance

Changes in producers  
might impact consumers

## Schema registry

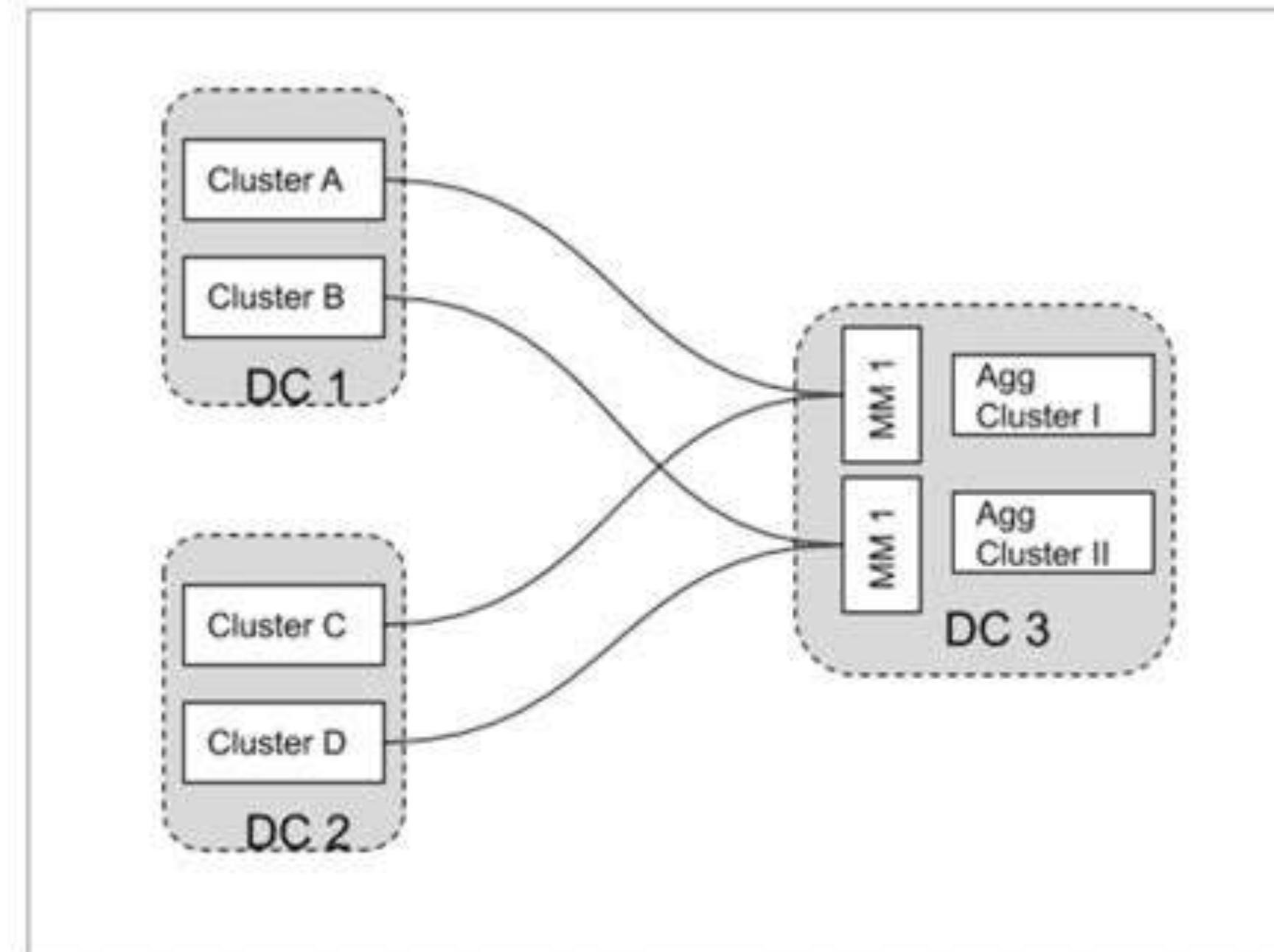
# Getting Started with Mirror Maker2

# Mirror Maker

- Kafka MirrorMaker is a tool used to replicate data between Kafka clusters, enabling seamless data migration, disaster recovery, and cross-region data synchronization.
- Using MirrorMaker to copy messages from one cluster to another in real time ensures high availability and fault tolerance for large deployments spanning multiple environments.

NB: A sample use case of Mirror Maker 2 with Active-Active Replication is provided in the [GITHUB Repository](#)

# Mirror Maker



Traditional Mirrormaker Aggregation Use Case

# Mirror Maker

- Key Features
- Data Replication
- Multi-Cluster Replication
- Support For Consumer Groups
- Selective Replication
- Automatic Recovery from Network/Cluster Failures

# Mirror Maker

- **Data Replication Capabilities**
  - MM2 efficiently replicates data across Kafka clusters, ensuring real-time availability of messages in multiple environments.
  - Better supports redundancy, disaster recovery, and hybrid or multicloud deployments.
- **Multi-Cluster Replication Across Different Regions or Environments**
  - Supports replication between geographically dispersed Kafka clusters.
  - Enables global event streaming, cross-region data availability, and compliance with data sovereignty regulations.

# Mirror Maker

- **Support for Consumer Groups**
  - MM2 replicates consumer group offsets, ensuring that applications consuming from replicated topics maintain their position.
  - This simplifies consumer failover and ensures seamless migration of applications between clusters.
- **Selective Replication**
  - Selection replication allows replication of specific topics instead of entire clusters as with MM1.
  - This allows precise control over data synchronization, allowing you to optimize bandwidth and storage costs.

# Mirror Maker

- **Automatic Recovery From Network or Cluster Failures**
  - MM2 automatically recovers and resynchronizes topics if a network or cluster issue occurs.
  - This ensures high availability and minimizes data loss during outages.

# Mirror Maker

- Multithreading for Parallel Processing
- MirrorMaker 2 allows parallelized data replication by increasing the number of consumer and producer threads.
- By running multiple consumer threads, MM2 can read from multiple partitions simultaneously, while additional producer threads ensure that data is written efficiently to the target cluster.
- Configuring `--num-streams 4` enables four parallel processing streams, improving throughput and reducing latency

# Some to watch for

- **Adjusting Consumer and Producer Settings**
- Tuning consumer fetch sizes and producer batch sizes is crucial for optimizing performance.
- E.g. setting `consumer.fetch.min.bytes=1048576` ensures that the consumer fetches larger batches of messages at a time, reducing network overhead.
- Similarly, setting `producer.batch.size=16384` allows the producer to send messages in efficient batch sizes, improving write performance.

# Some to watch for

- **Increasing Network Bandwidth Utilization**
- Ensuring adequate network bandwidth is essential for high-performance replication.
- Deploying MirrorMaker in a high-bandwidth environment reduces congestion and speeds up data transfer.
- Additionally, enabling compression techniques, such as setting `compression.type=lz4`, significantly reduces network payload size, lowering latency and improving efficiency.

# Some to watch for

- **Compressing Data Transmissions**
- Compression algorithms like lz4 or snappy help optimize MirrorMaker's replication efficiency.
- These algorithms reduce message size before transmission, minimizing bandwidth usage while maintaining fast decompression speeds on the receiving end.

# Some to watch for

- **Partition, Topic, and Kafka Buffer Tuning**
- Fine-tuning partition distribution and buffer configurations ensures optimal performance.
- E.g. setting `replication.factor=3` enhances data redundancy,
- E.g setting `log.retention.hours=168` configures the log retention period to maintain historical data without excessive storage overhead.

# Some to watch for

- **Optimizing for Large Messages and Workloads**
- For workloads involving large messages, adjusting Kafka's request size and buffer configurations is essential.
- Increasing `fetch.message.max.bytes=10485760` and `message.max.bytes=10485760` allows MM2 to handle large message payloads efficiently, preventing truncation and ensuring reliable delivery.
- **Adjusting Consumer Lag Thresholds**
- Monitoring and managing consumer lag is critical to preventing message delays.
- Setting `max.poll.records=500` optimizes the polling mechanism, ensuring consumers process messages at an optimal rate without excessive lag.

# Some to watch for

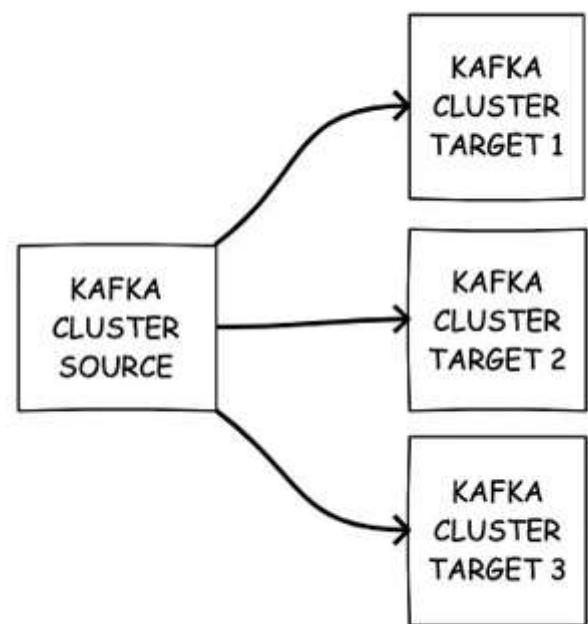
- Configuring multiple consumer groups within MirrorMaker enhances high availability and load balancing.
- Using `group.id=mirrormaker-group-1` ensures that messages are processed in a distributed manner, improving resilience and scalability.

# Mirror Maker

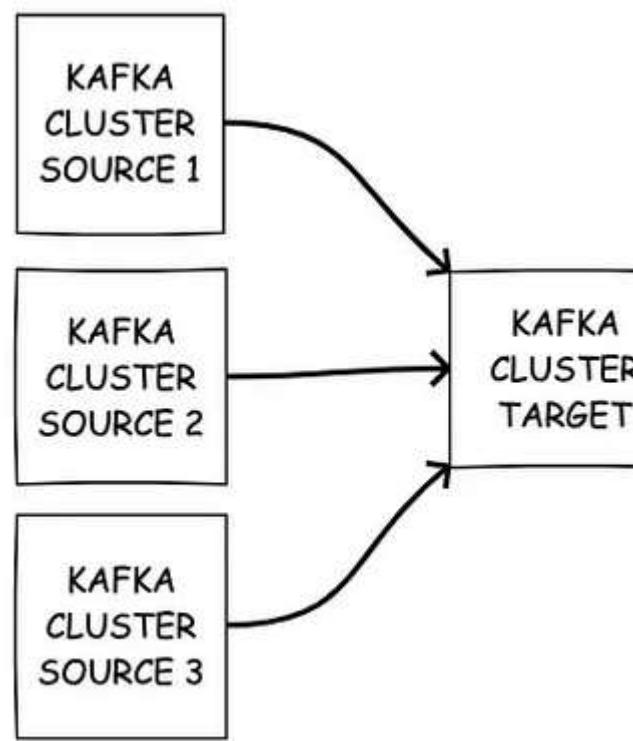
- Enable Auto – Restart
- Deploy Multiple MM2 instances
- Deploy MM2 across Multiple Availability Zones
- Set Higher Replication Factor  $\geq 3$
- Integrating Prometheus and Grafana for monitoring helps track consumer lag and network health, allowing proactive issue resolution
- Distributing replication tasks among multiple MirrorMaker nodes ensures balanced workloads, optimizing replication speed and resource utilization.

# Mirror Maker

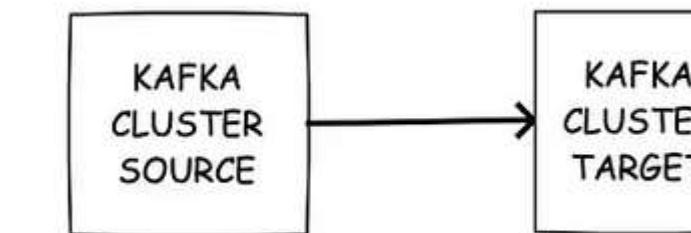
- Modes



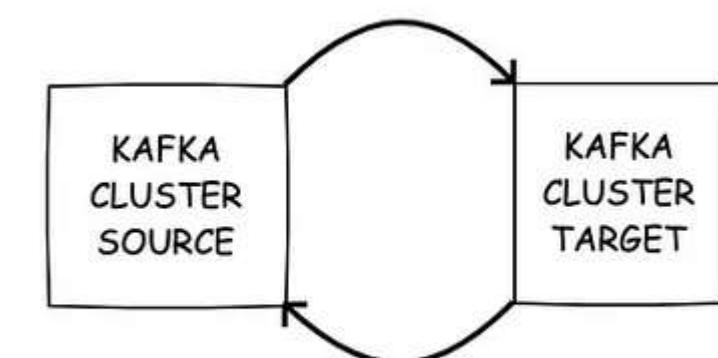
FAN-OUT



AGGREGATION



ACTIVE-PASSIVE



ACTIVE-ACTIVE

# Mirror Maker

- MirrorMaker 2 Configuration Issues
  - Signs: High error rates, failure to replicate data
  - Solution: Validate consumer and producer configurations, ensure correct ACLs and authentication settings
- Replication Lag and Delays
  - Signs: Delayed messages, inconsistencies in real-time applications
  - Solution: Tune `fetch.min.bytes`, `fetch.max.wait.ms`, and increase partitions to distribute load

# Mirror Maker

- Out of Memory Errors
  - Signs: JVM crashes, process restarts frequently
  - Solution: Increase heap size (-Xmx) for MirrorMaker, optimize batch.size, and enable compression
- Kafka Broker Connectivity Failures
  - Signs: MirrorMaker failing to connect to source or target clusters
  - Solution: Verify network connectivity, ensure bootstrap.servers is correctly configured

# Mirror Maker

- **Topic Mismatches Between Clusters**
  - **Signs:** Data missing in target cluster
  - **Solution:** Ensure topic auto-creation is enabled or create topics manually before replication
- **Partition Imbalance**
  - **Signs:** Some partitions overloaded while others remain idle
  - **Solution:** Increase number of MirrorMaker workers, enable round-robin partitioning

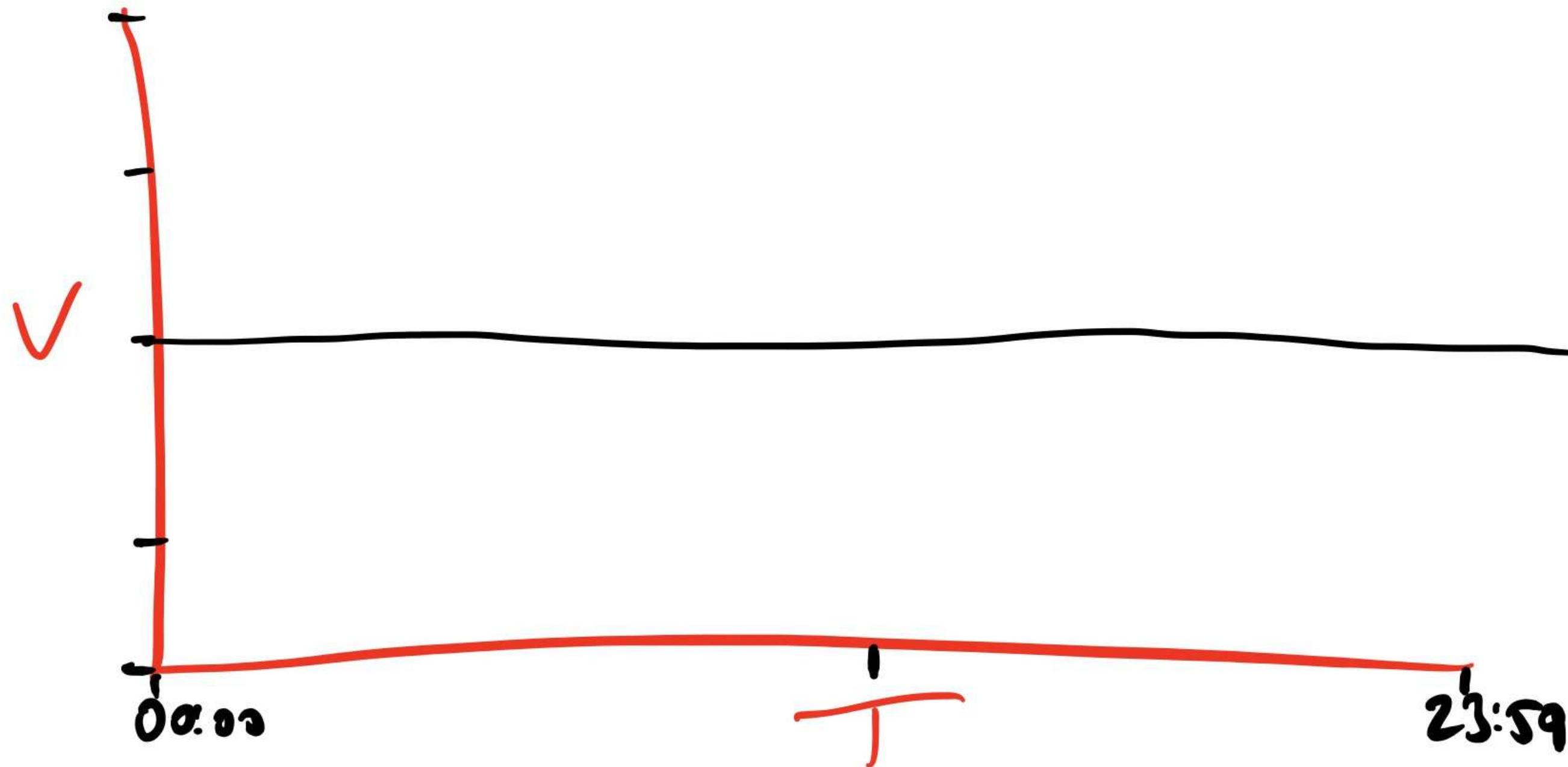
# Mirror Maker

- **Missing or Outdated Consumer Group Offsets**
  - Signs: Consumers reprocessing old messages
  - Solution: Use MirrorMaker's offset translation feature to map source offsets correctly
- **Disk Space and Storage Issues**
  - Signs: Kafka brokers running out of space, replication slowing down
  - Solution: Implement log retention policies, use tiered storage solutions

# Capacity Planning

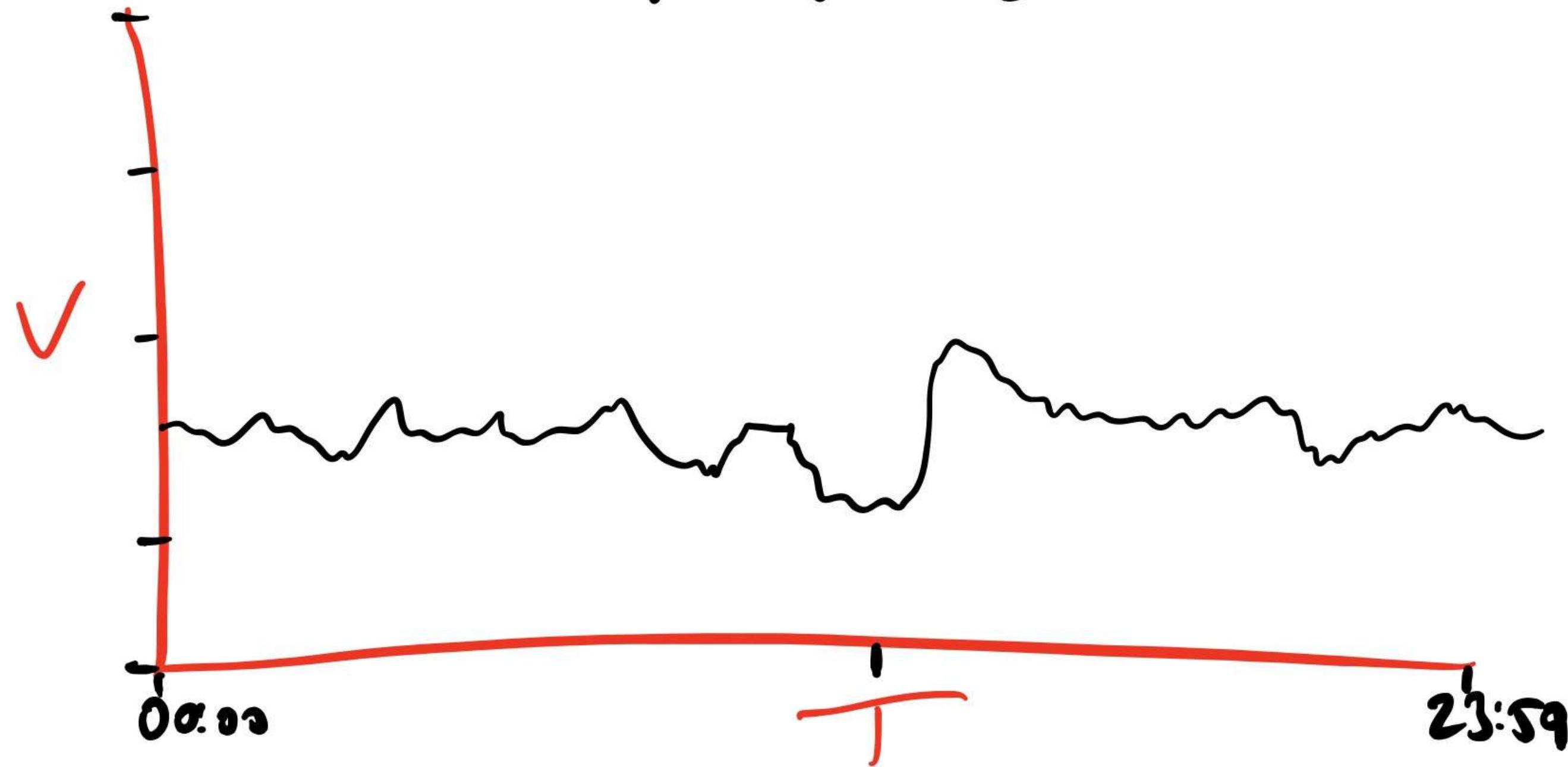
And when we say  
**streaming data** what do  
we *really* mean?

"On Yeah, Right."



The **reality** is usually different.

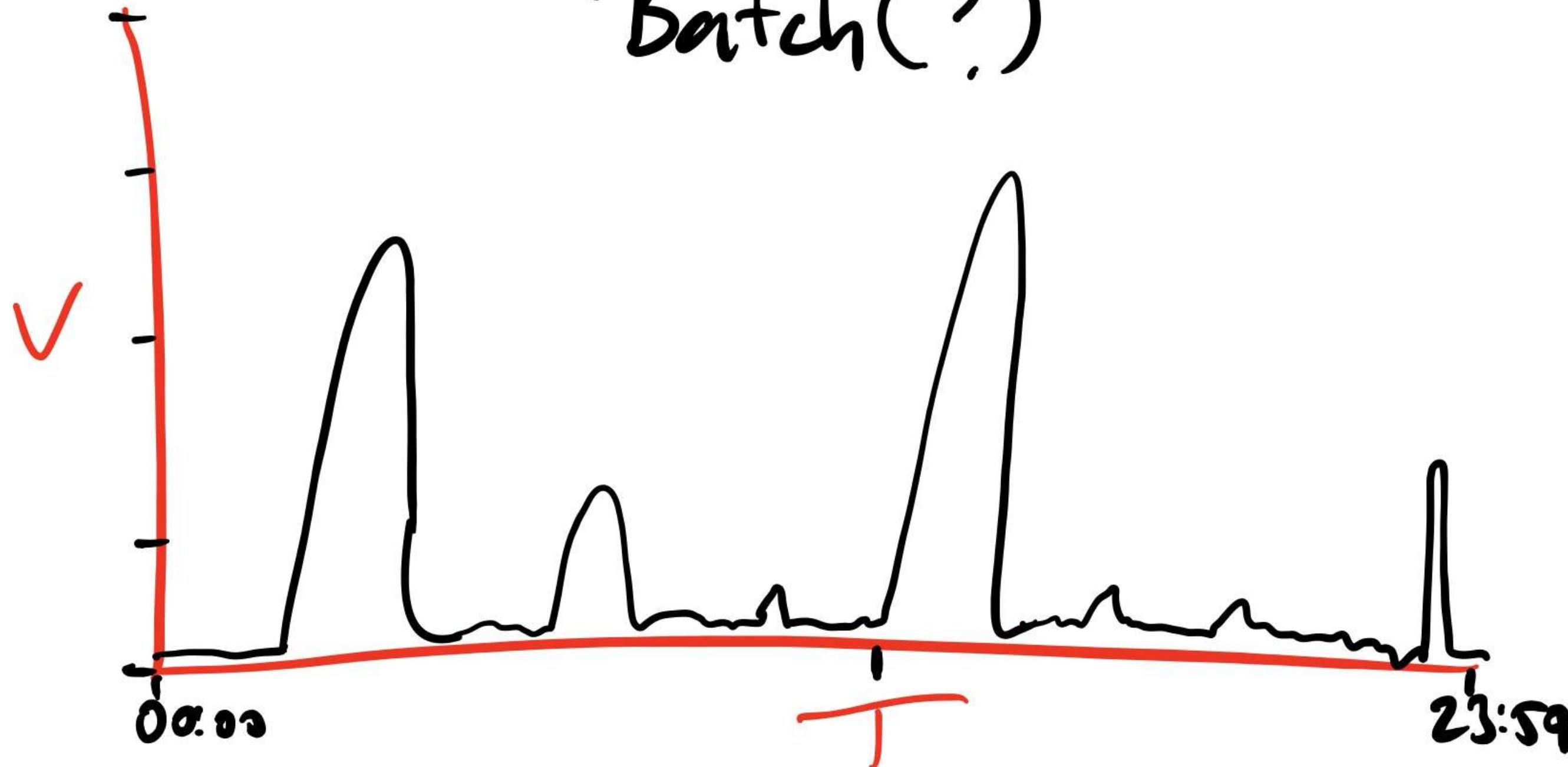
"Realtime"



"Retail"



"Batch(?)"





# What you should ask for

- Average Message Size
- Estimated Daily Quantity
- Any Peak Per Hour Quantity
- Desired Replication Factor
- Desired Partitions
- Minimum In-sync Replicas

# What you should ask for

- Average Message Size - (6 KB)
- Estimated Daily Quantity - (10,000,000/d)
- Any Peak Per Hour Quantity - (1,250,000)
- Desired Replication Factor - (4)
- Desired Partitions - (10)
- Minimum In-sync Replicas - (2)

# Estimated Capacity

**(Message size x 3)**

- \* Daily Qty**
- \* 1.4 (add 40%)**
- = Volume per replicated broker.**

# Estimated Capacity

(Message size x 3)

\* Daily Qty

\* 1.4 (add 40%)

= Volume per replicated broker.

(6KB x 3) x 10,000,000 = 184,320,000 KB

\* 1.4 (add 40%)

= 258,048,000 KB

= 248.09 GB

Roughly translates to 2.940 MB/sec

The x3 gives a payload size with key, header, timestamp and the value.  
It's just a rough calculation. Adding 40% overhead will give some breathing space .....

# Retention

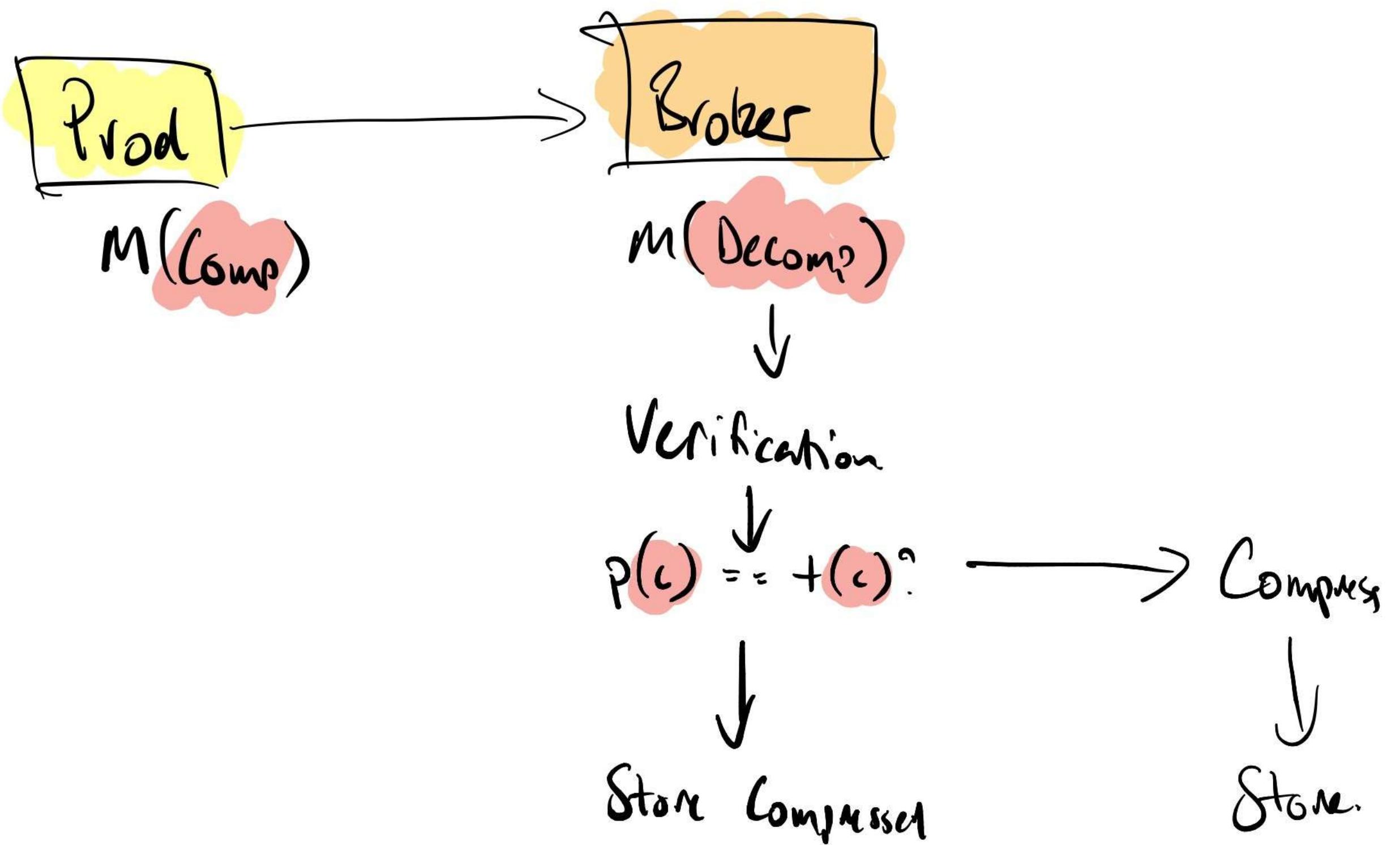
**(6KB x 3) x 10,0000,000 = 184,320,000 KB  
x 1.4 (add 40%)  
= 258,048,000 KB  
= 248.09 GB**

**248.09 GB/day x 14 days retention  
= 3.4 TB per broker.**

**Producer configuration** `compression.type` **defaults to**  
**“none”.**

**Your options are** `gzip`, `snappy`, `lz4` **and** `zstd`.

**Expect ~20%-40% message compression  
depending on the algorithm used.**

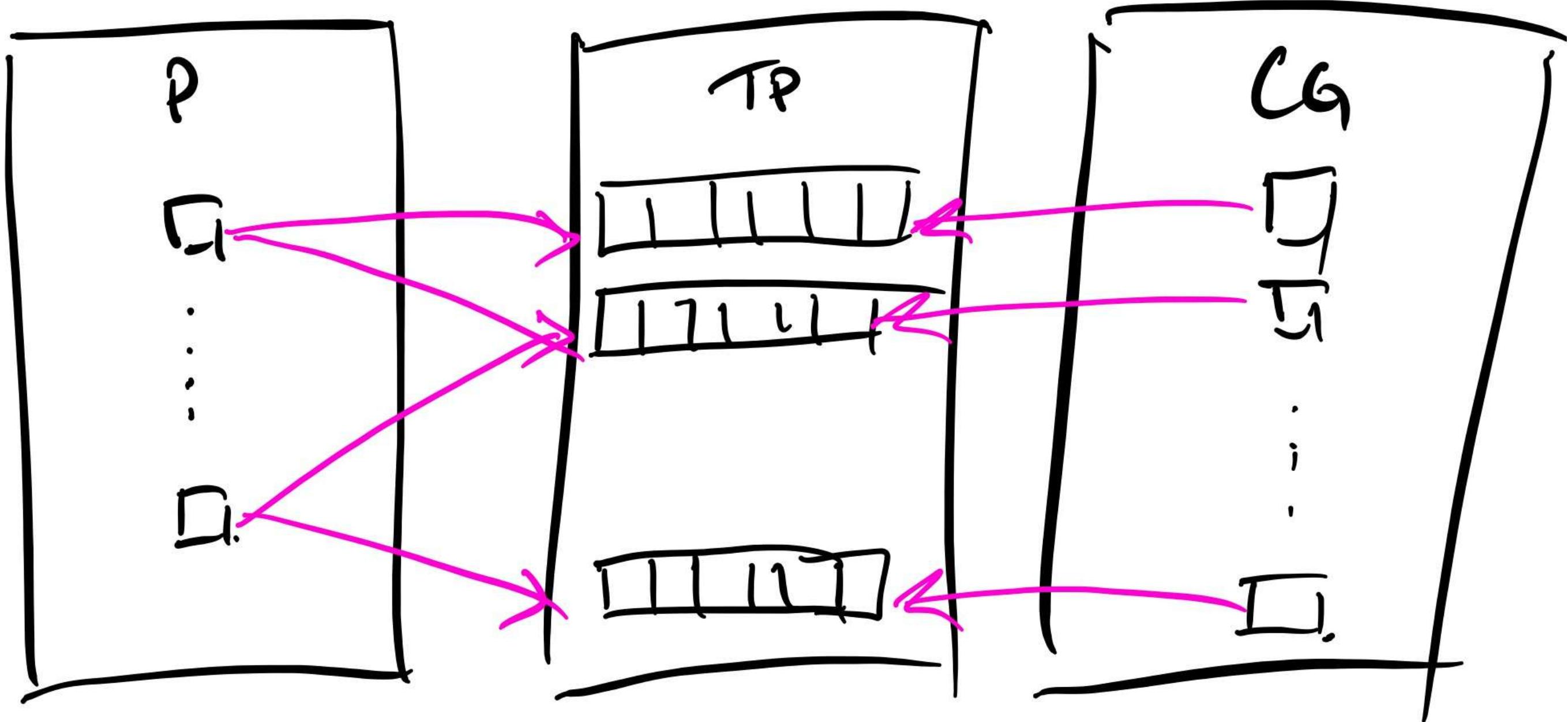


# Stress Testing

```
kafka-producer-perf-test --topic TOPIC --record-size SIZE_IN_BYT
```

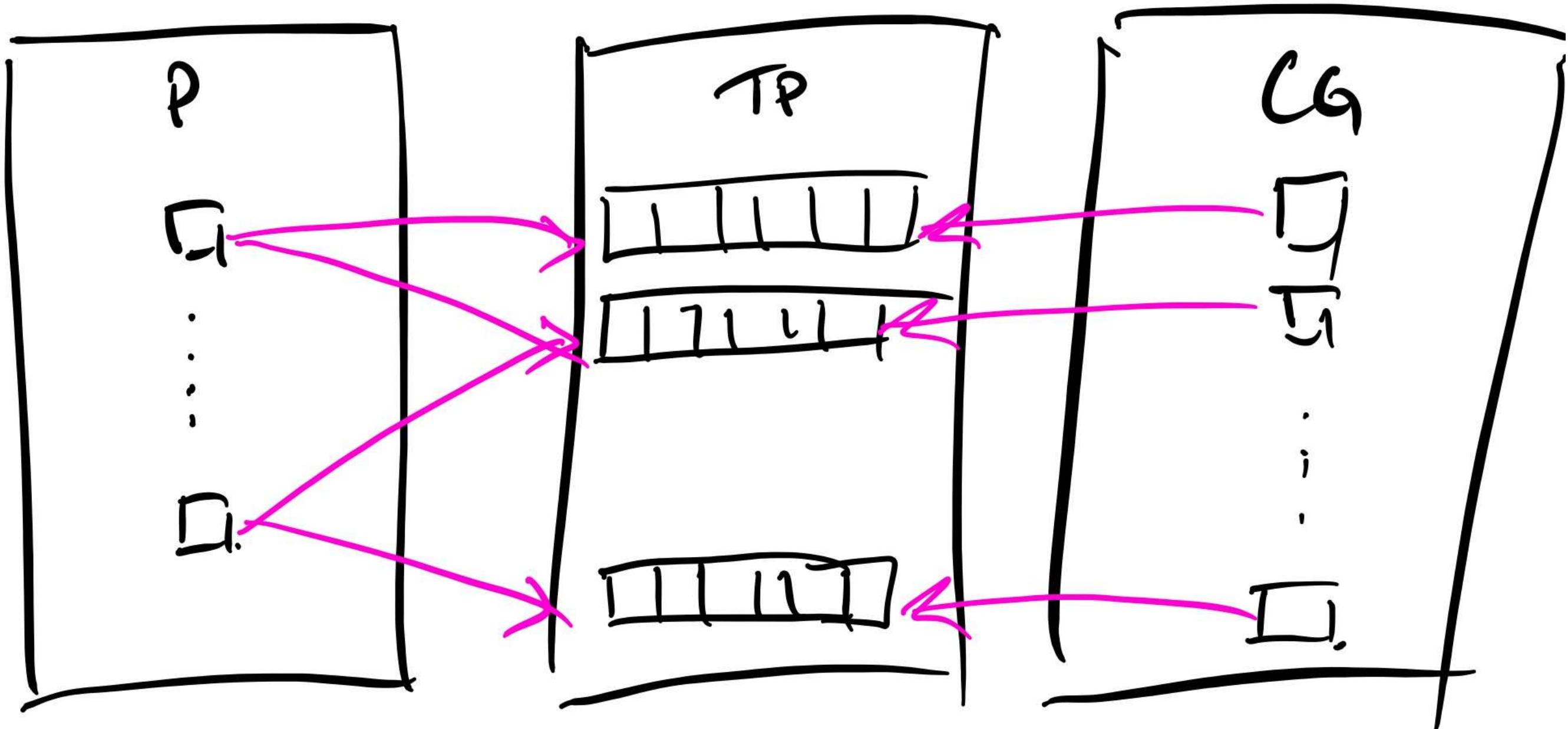
```
kafka-consumer-perf-test      --broker-list      host1:port1,host2:port2      --
topic TOPIC
```

# Topic Partitions



Per Consumer Reads @ 50ms/sec

$$16B \div 50ms = 20$$



Producer Writes @ 100 MB/sec.

$$16\text{GB} \div 100 = 10$$

**Having a large number of partitions will have effects on Zookeeper znodes.**

- **More network requests**
- **If leader or broker goes down it may affect startup time as the broker returns to the cluster.**