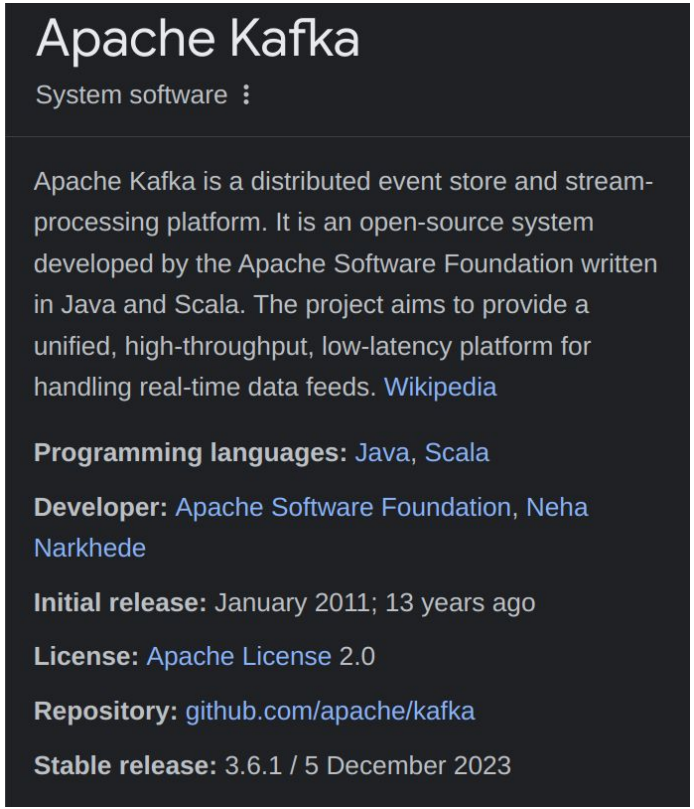
Kafka & It’s Internal Architecture

What is Strimzi ?

Strimzi **simplifies** the process of running Apache Kafka in a Kubernetes cluster.

What is Apache Kafka ?



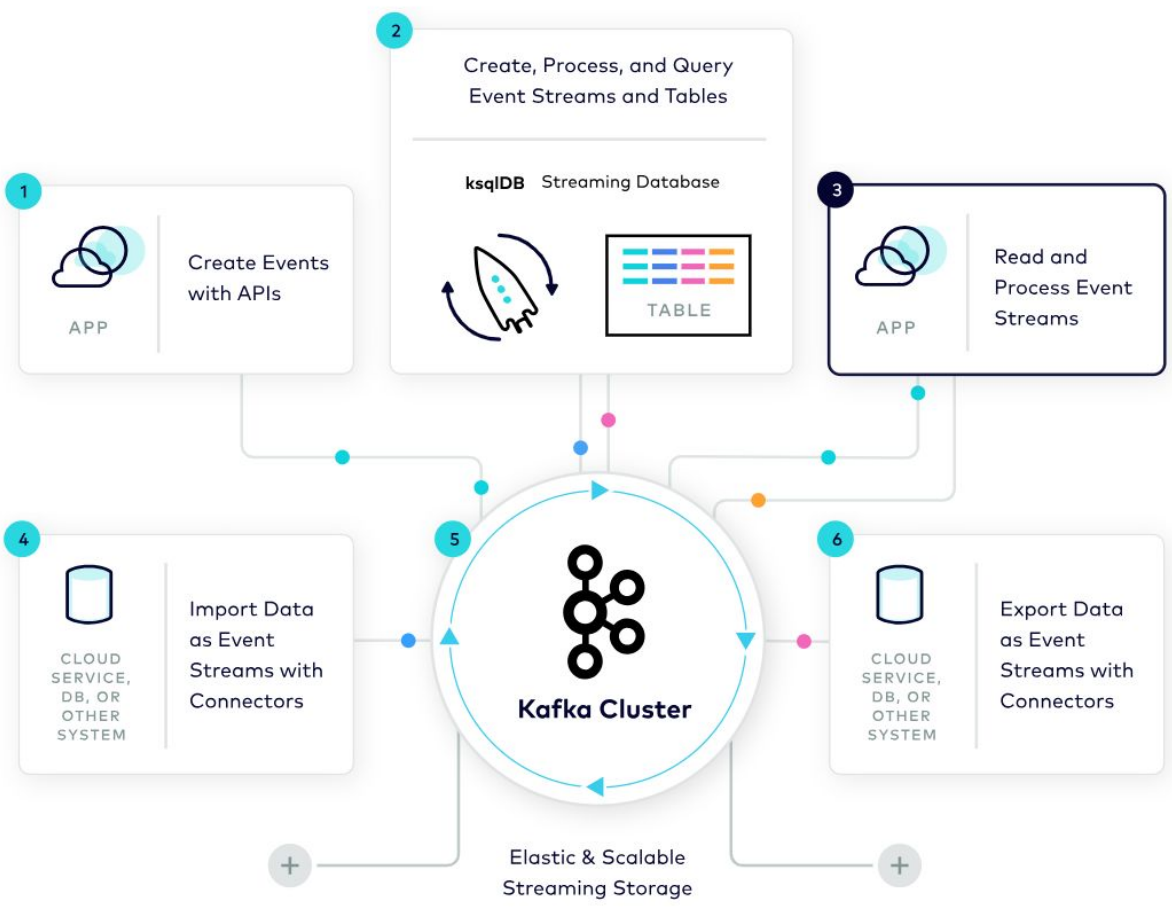
Kafka capabilities

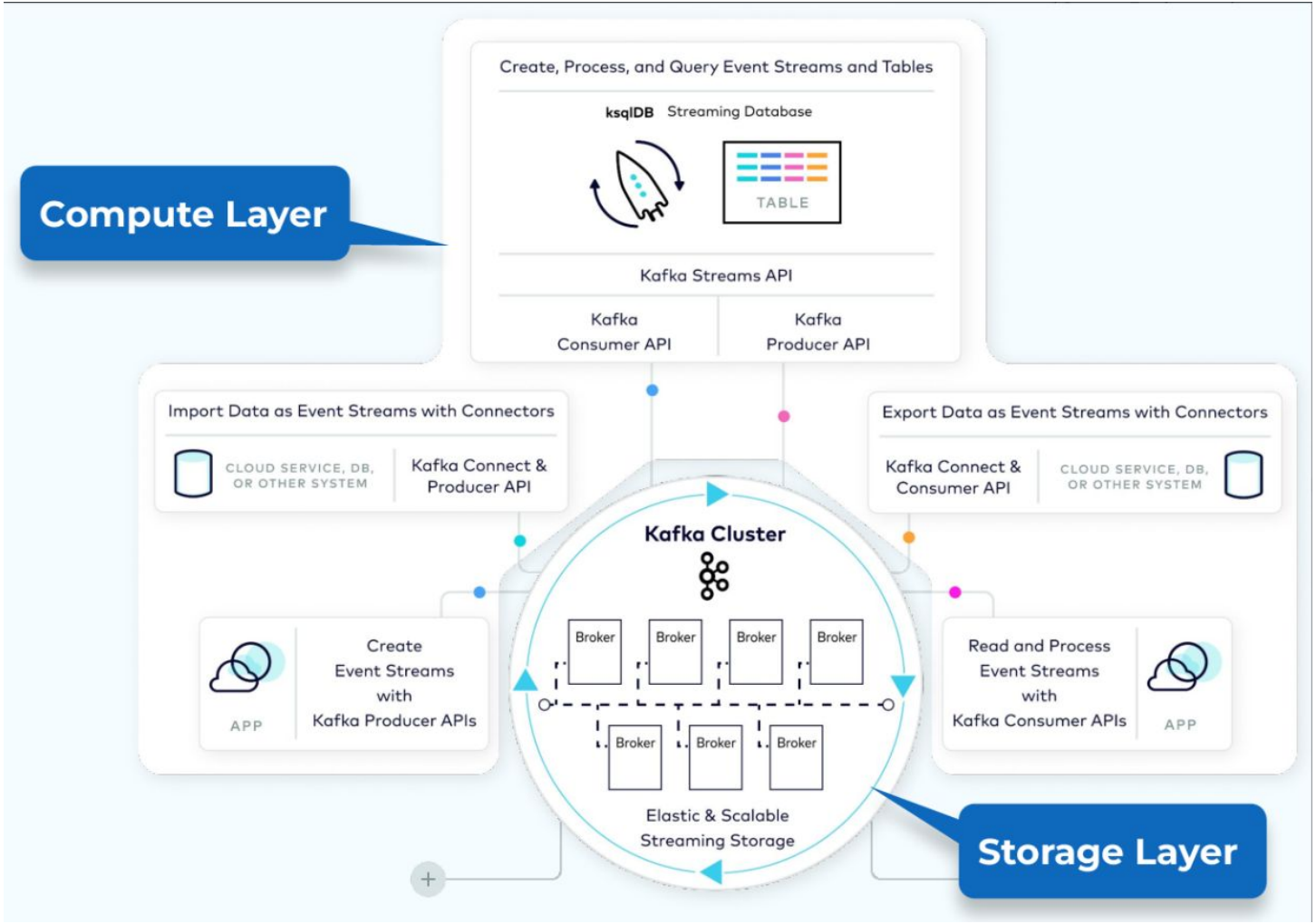
- Microservices and other applications to share data with extremely high throughput and low latency
- Message ordering guarantees
- Message rewind/replay from data storage to reconstruct an application state
- Message compaction to remove old records when using a key-value log
- Horizontal scalability in a cluster configuration
- Replication of data to control fault tolerance
- Retention of high volumes of data for immediate access

Kafka use cases

- Event-driven architectures
- Event sourcing to capture changes to the state of an application as a log of events
- Message brokering
- Website activity tracking
- Operational monitoring through metrics
- Log collection and aggregation
- Commit logs for distributed systems
- Stream processing so that applications can respond to data in real time

Overview


2
Create, Process, and Query
Event Streams and Tables
ksqlDB Streaming Database
TABLE
1
APP
Create Events
with APIs
3
APP
Read and
Process Event
Streams
4
CLOUD
SERVICE,
DB, OR
OTHER
SYSTEM
Import Data
as Event
Streams with
Connectors
5
Kafka Cluster
6
CLOUD
SERVICE,
DB, OR
OTHER
SYSTEM
Export Data
as Event
Streams with
Connectors
Elastic & Scalable
Streaming Storage


Compute Layer
Create, Process, and Query Event Streams and Tables
ksqlDB Streaming Database
TABLE
Kafka Streams API
Kafka Consumer API
Kafka Producer API
Import Data as Event Streams with Connectors
CLOUD SERVICE, DB, OR OTHER SYSTEM
Kafka Connect & Producer API
Export Data as Event Streams with Connectors
Kafka Connect & Consumer API
CLOUD SERVICE, DB, OR OTHER SYSTEM
Kafka Cluster
Broker
Broker
Broker
Broker
Broker
Broker
Broker
APP
Create Event Streams with Kafka Producer APIs
Read and Process Event Streams with Kafka Consumer APIs
APP
Elastic & Scalable Streaming Storage
Storage Layer

The **storage layer** is designed be distributed system, such that if your storage needs grow over time you can easily scale out the system to accommodate the growth

The **compute layer** consists of four core components—

1. Producer Api
2. Consumer Api
3. Streams Api
4. Connector Api

which allow Kafka to scale applications across distributed systems.

Producer and Consumer APIs

The foundation of Kafka's powerful application layer is two primitive APIs for accessing the storage.

- **the producer API for writing events**
- **the consumer API for reading them.**

Kafka Streams Api

For processing events as they arrive, we have Kafka Streams, a Java library that is built on top of the producer and consumer APIs.

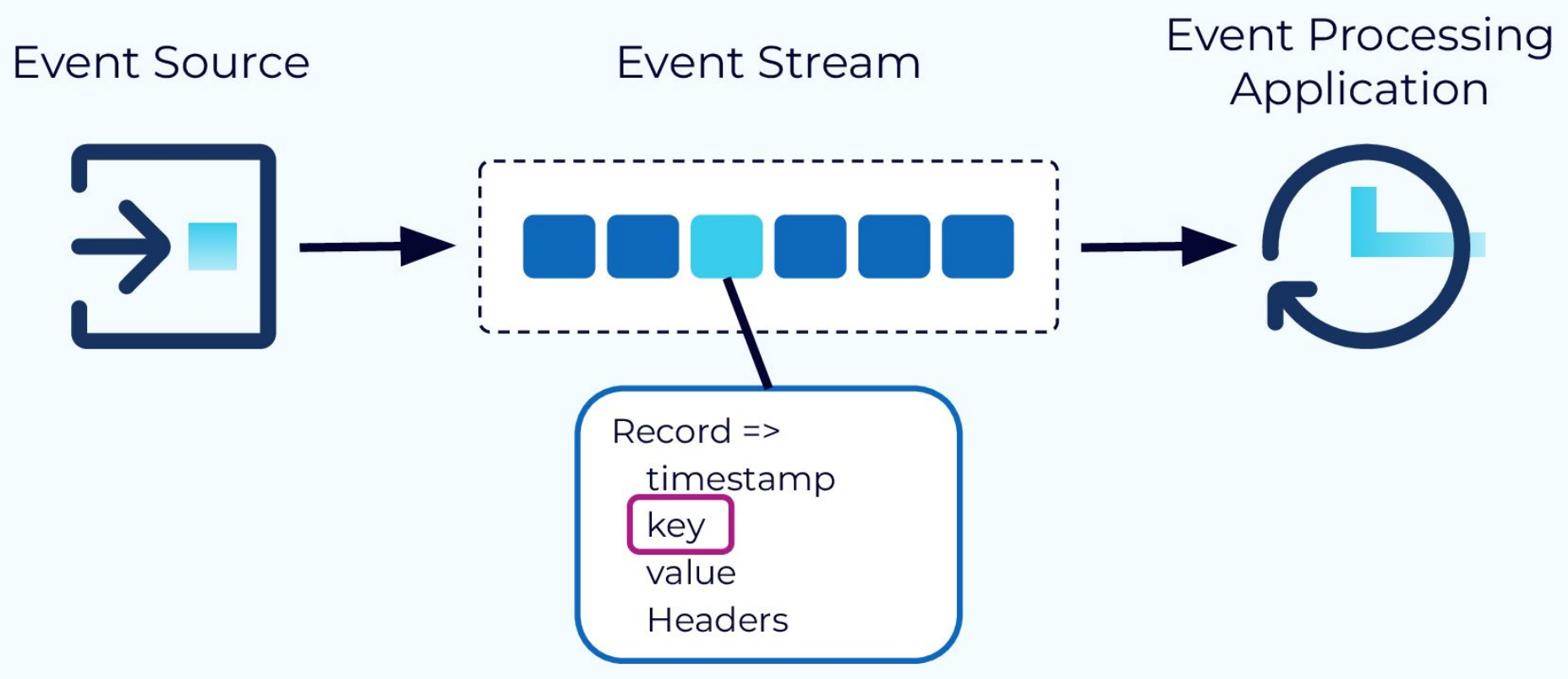
Kafka Streams allows you to perform real-time stream processing, powerful transformations, and aggregations of event data.

Kafka Connect Api

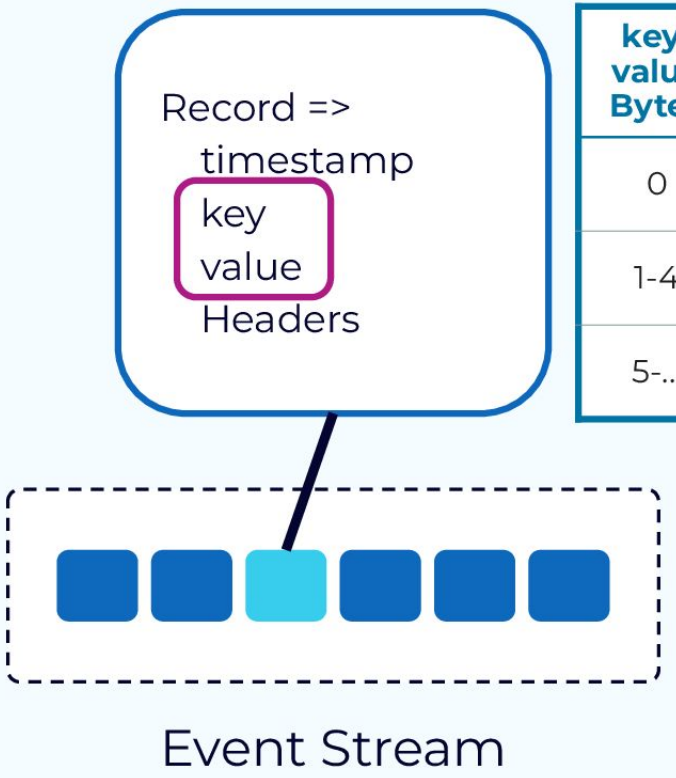
Kafka Connect, which is built on top of the producer and consumer APIs, provides a simple way to integrate data across Kafka and external systems.

- **Source connectors** bring data from external systems and produce it to Kafka topics.
- **Sink connectors** take data from Kafka topics and write it to external systems.

What is an **Event** in Stream Processing?

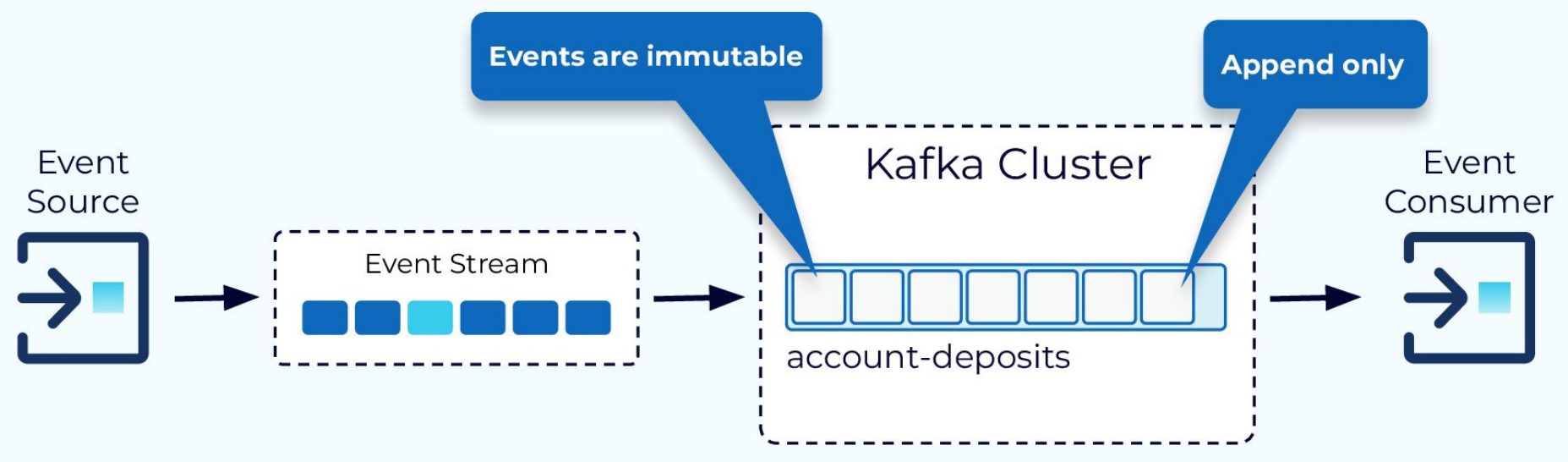


Record Schema

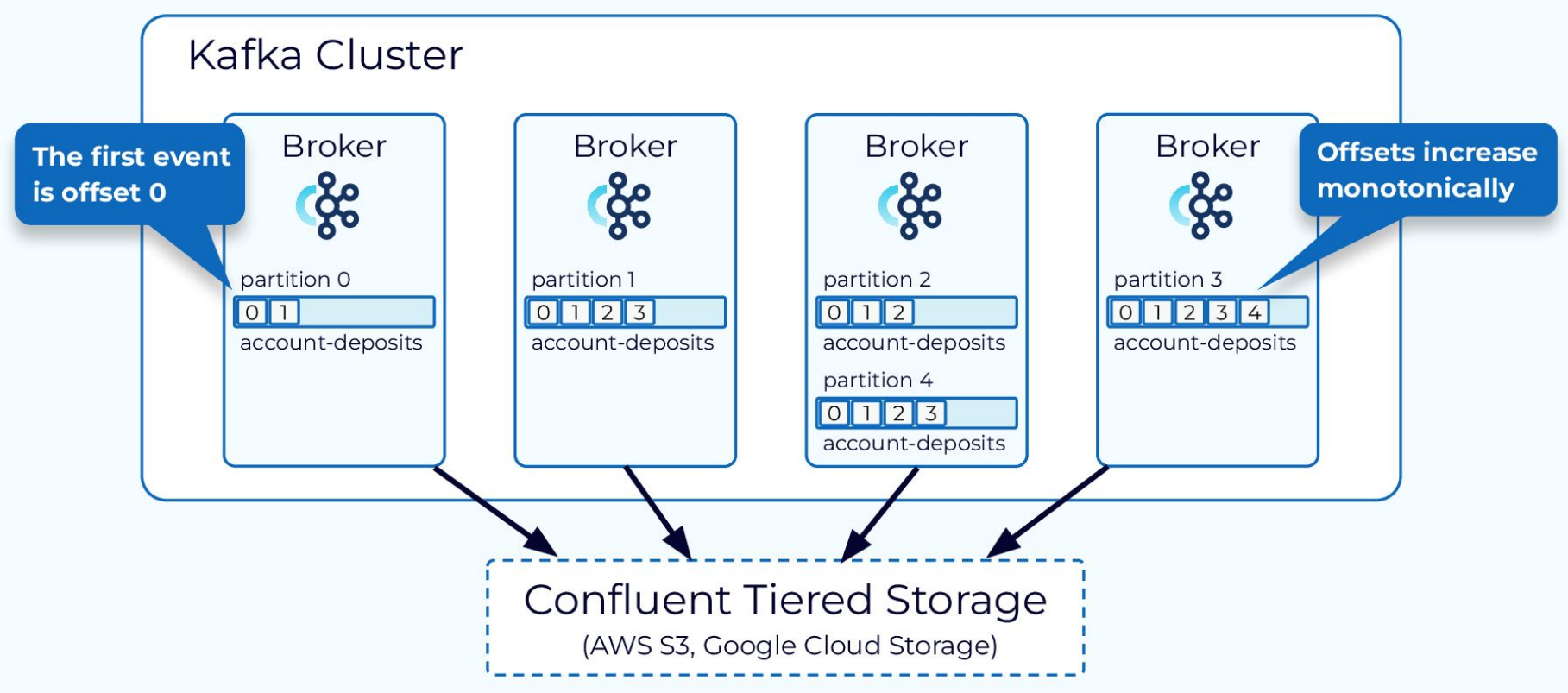


key/ value Bytes	Area	Description
0	Magic Byte	Confluent serialization format version number; currently always 0.
1-4	Schema ID	4-byte schema ID as returned by Schema Registry.
5-...	Data	Serialized data for the specified schema format.

Kafka Topics



Kafka Topic Partitions



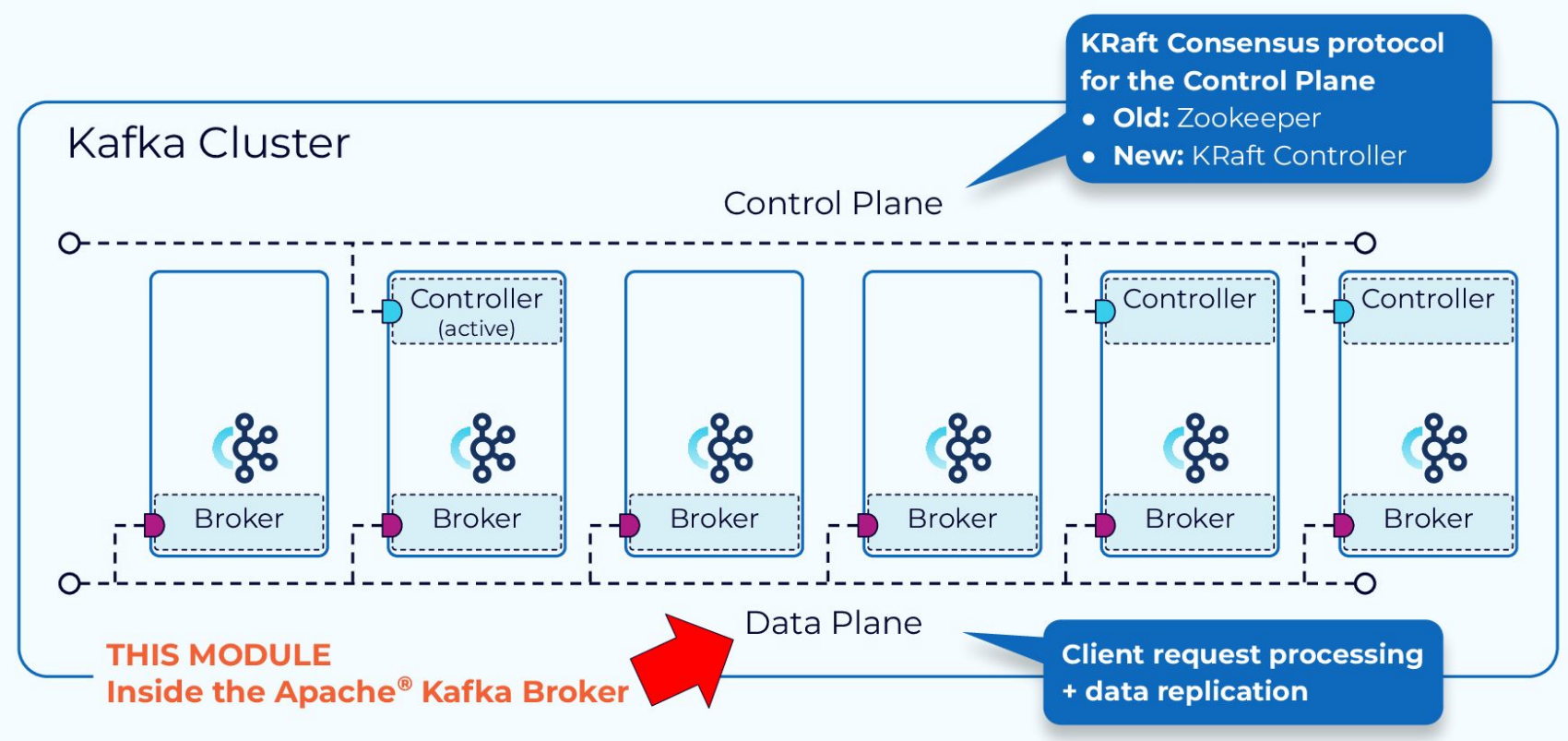
Producer is to decide which partition to send the messages to.

1. **No key specified** :- producer will randomly decide partition and would try to balance the total number of messages on all partitions
2. **Key Specified :-** the producer uses Consistent Hashing to map the key to a partition.
3. **Partition Specified :-** You can hardcode the destination partition as well.
4. **Custom Partitioning logic:-** We can write some rules depending on which the partition can be decided.

Any questions?

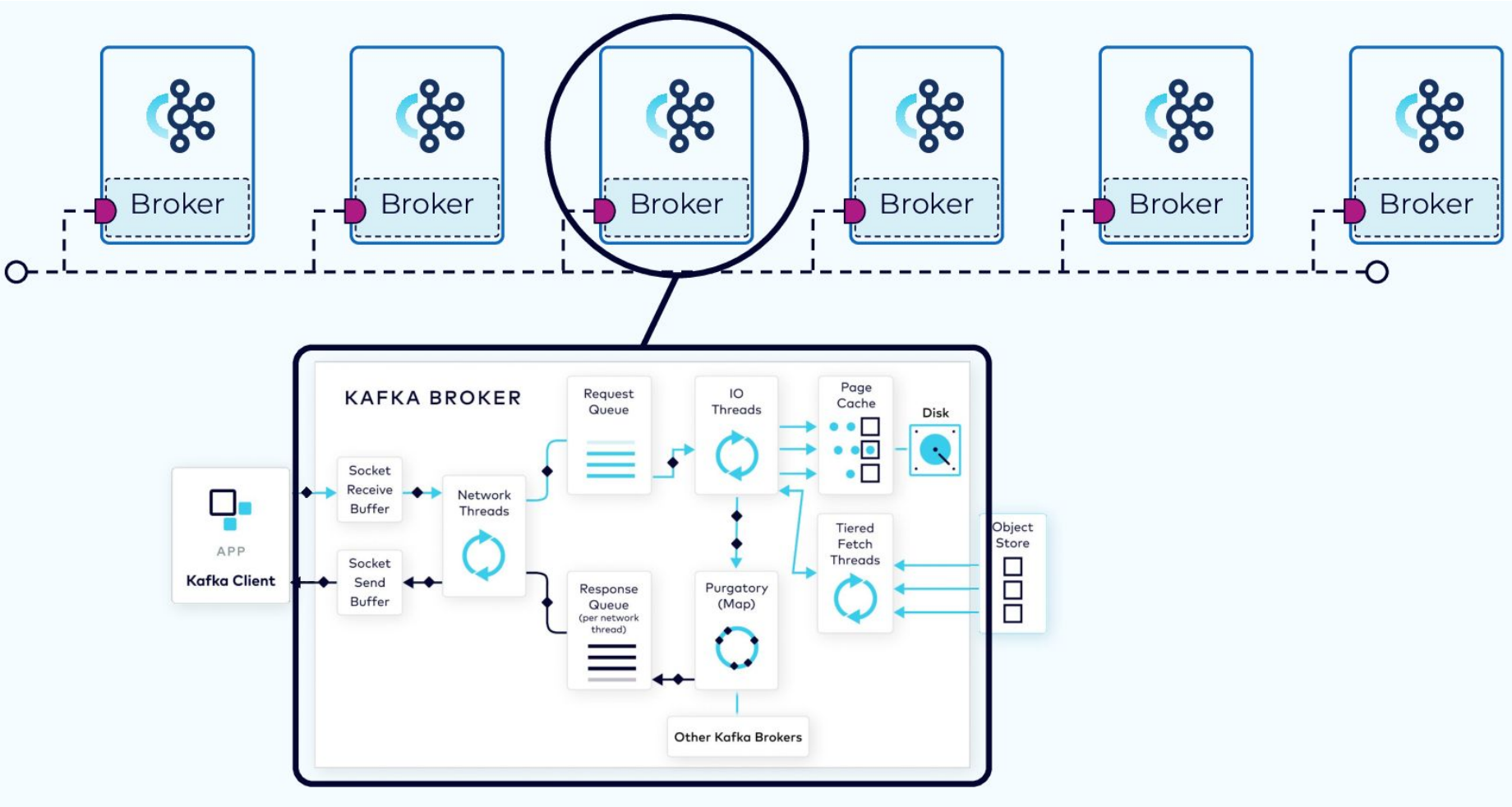
Inside the Apache Kafka Broker

Kafka manages Data & Metadata Separately

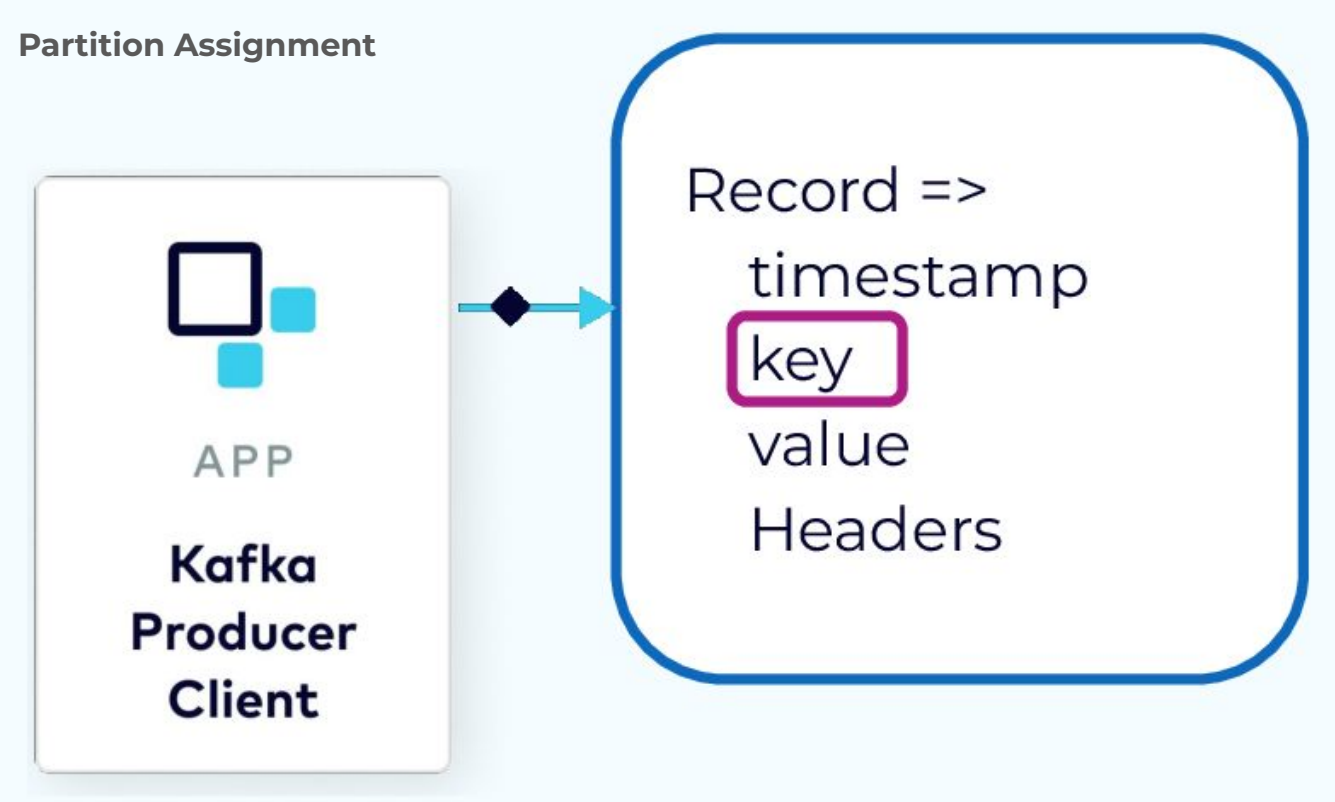


Apache Kafka Broker

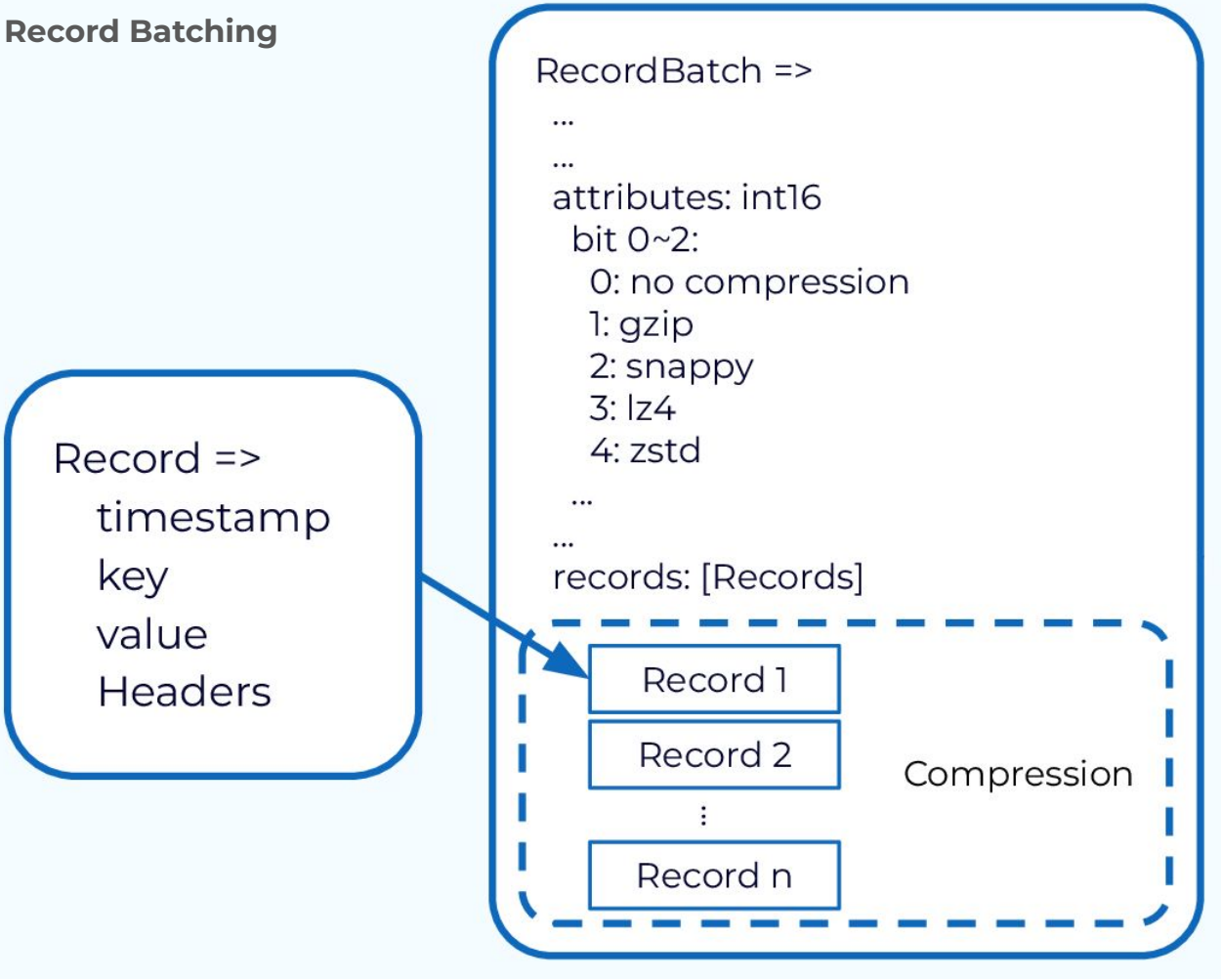
- An computer, instance, or container running the Kafka process
- Manage partitions
- Handle write and read requests
- Manage replication of partitions
- Intentionally very simple

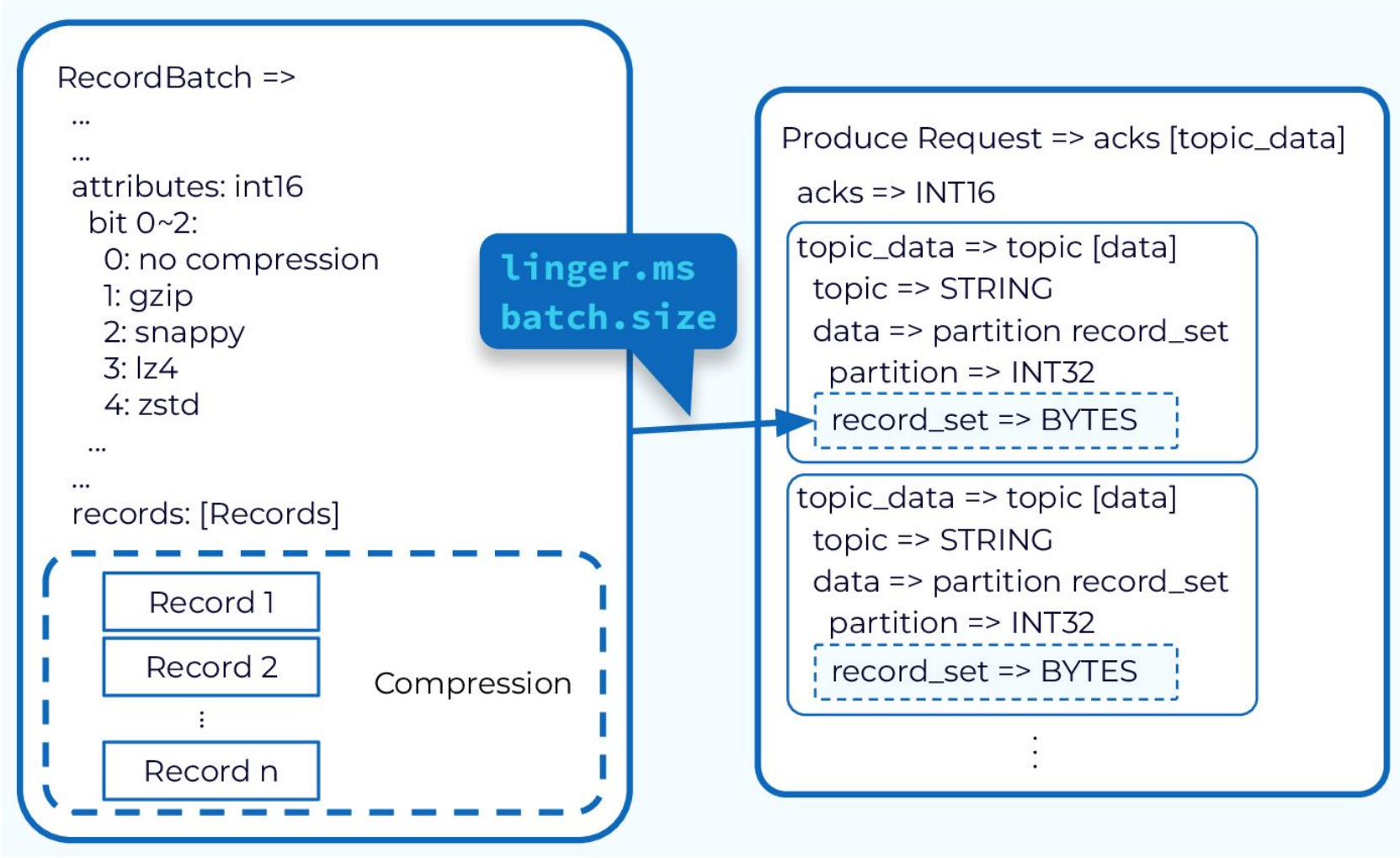

Broker
Broker
Broker
Broker
Broker
Broker
KAFKA BROKER
Request Queue
IO Threads
Page Cache
Disk
Socket Receive Buffer
Network Threads
Tiered Fetch Threads
Object Store
APP
Kafka Client
Socket Send Buffer
Response Queue
(per network thread)
Purgatory (Map)
Other Kafka Brokers

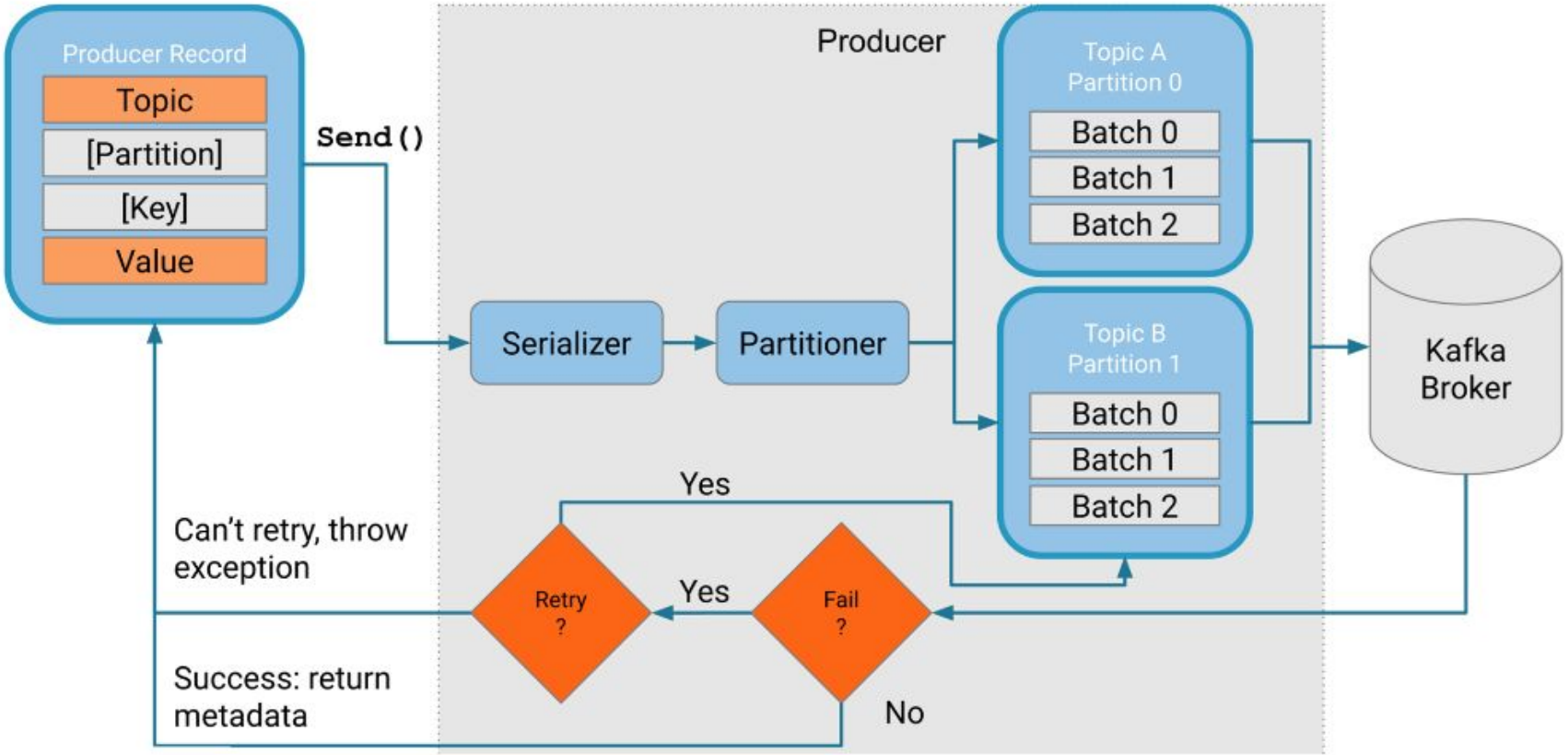
The Produce Request

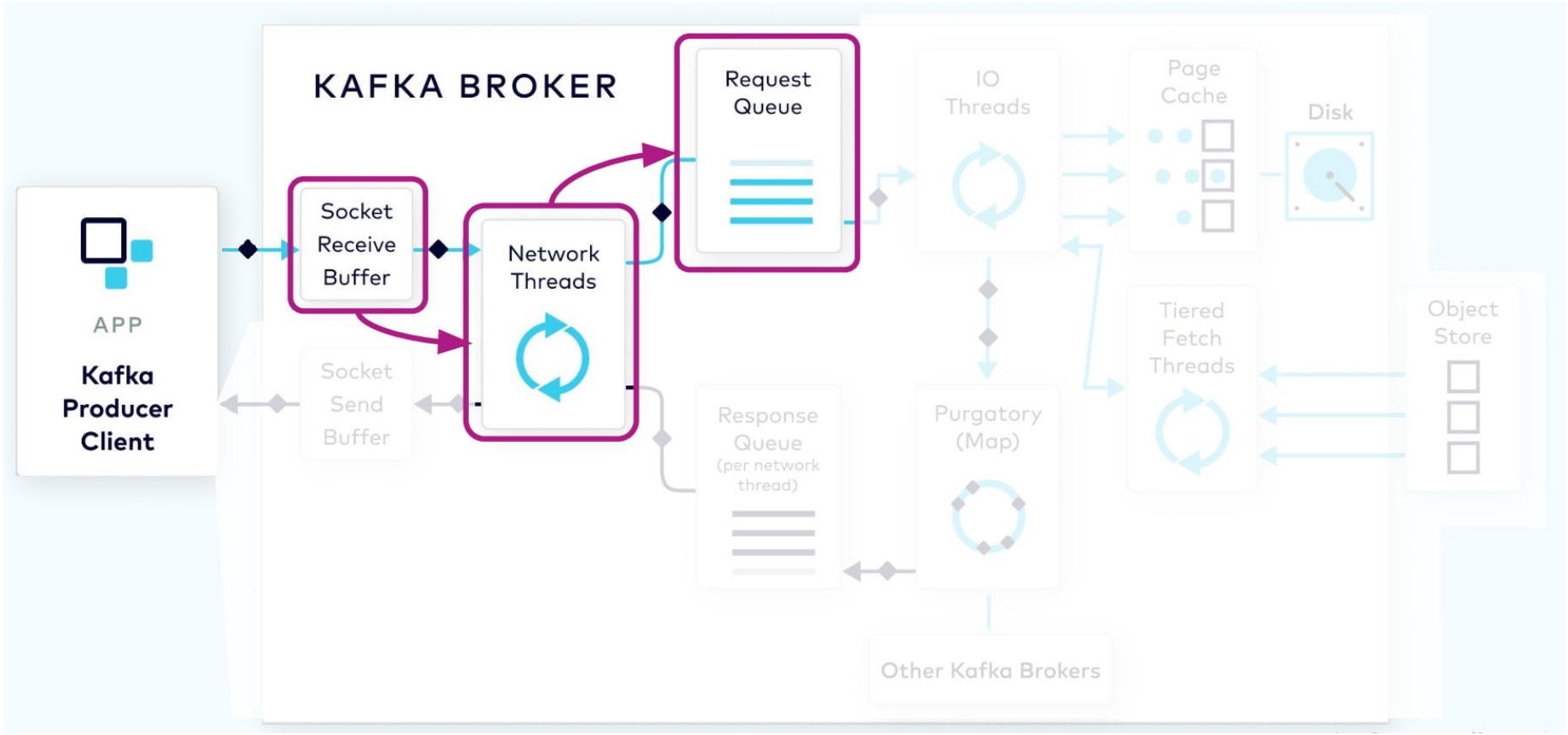


Record Batching



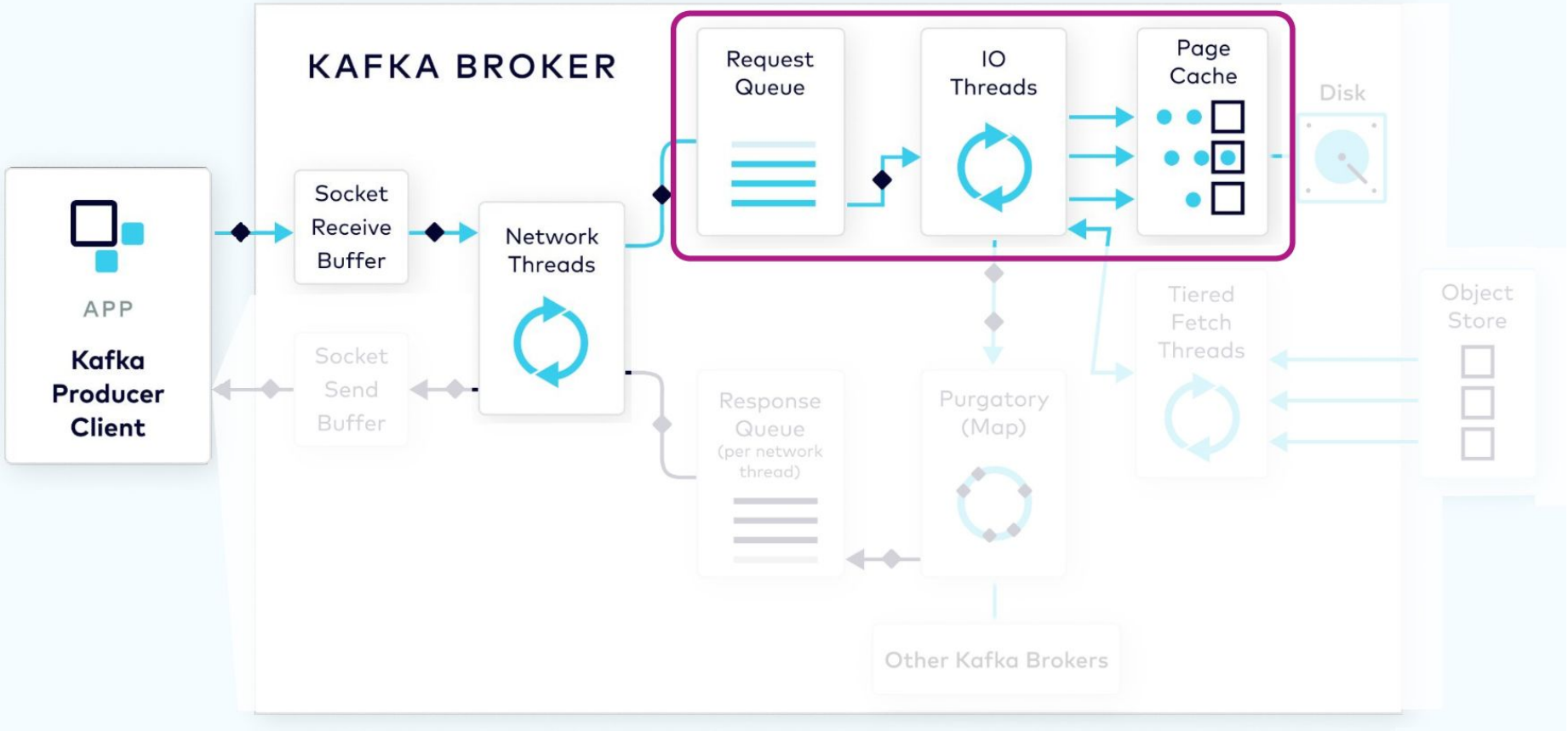

RecordBatch =>
...
...
attributes: int16
bit 0~2:
0: no compression
1: gzip
2: snappy
3: lz4
4: zstd
...
...
records: [Records]
Record 1
Record 2
Record n
Compression
linger.ms
batch.size
Produce Request => acks [topic_data]
acks => INT16
topic_data => topic [data]
topic => STRING
data => partition record_set
partition => INT32
record_set => BYTES
topic_data => topic [data]
topic => STRING
data => partition record_set
partition => INT32
record_set => BYTES


Producer Record
Topic
[Partition]
[Key]
Value
Send()
Producer
Topic A
Partition 0
Batch 0
Batch 1
Batch 2
Topic B
Partition 1
Batch 0
Batch 1
Batch 2
Kafka
Broker
Serializer
Partitioner
Yes
Can't retry, throw
exception
Retry
?
Yes
Fail
?
Success: return
metadata
No

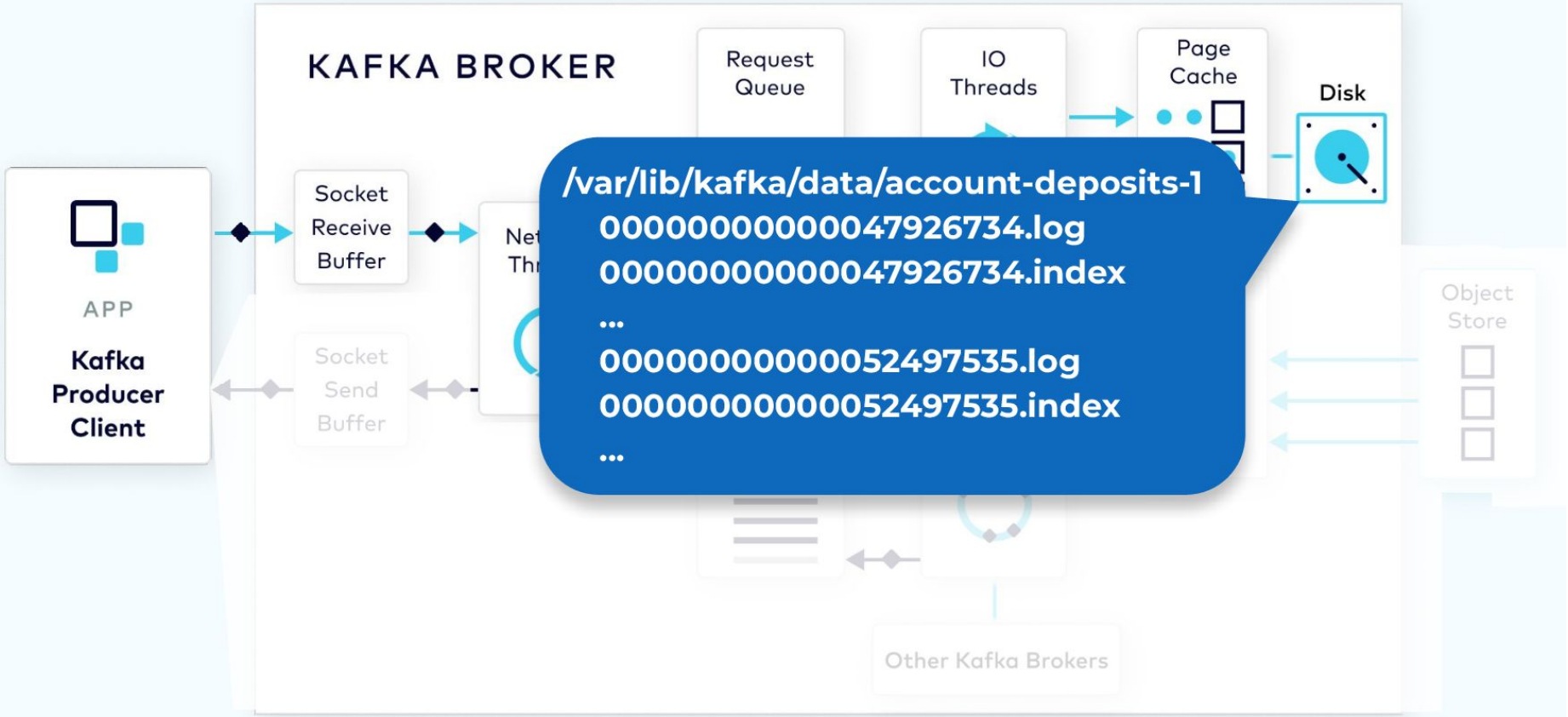


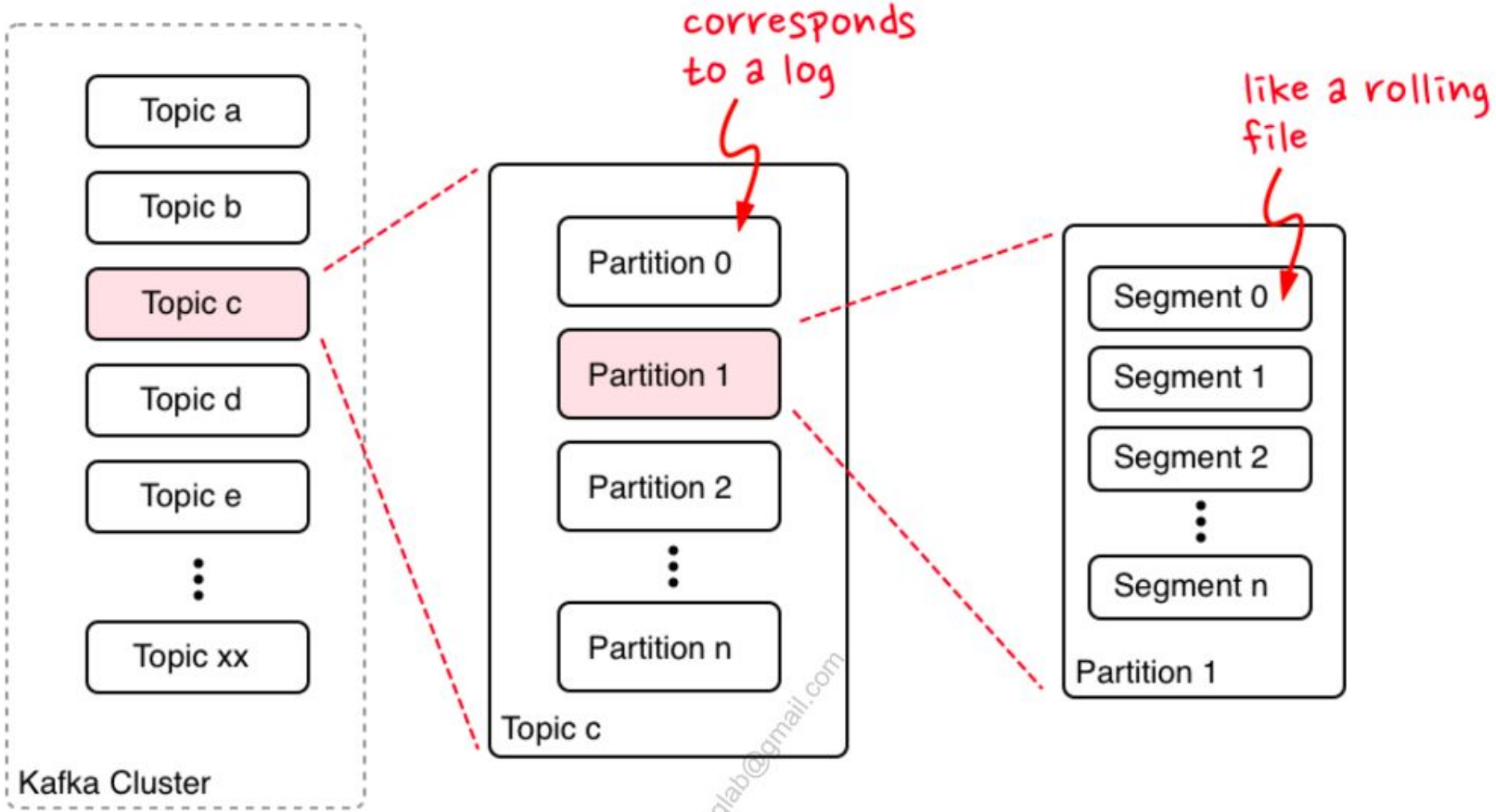
Network Thread Adds Request to Queue

I/O Thread Verifies and Stores the Batch

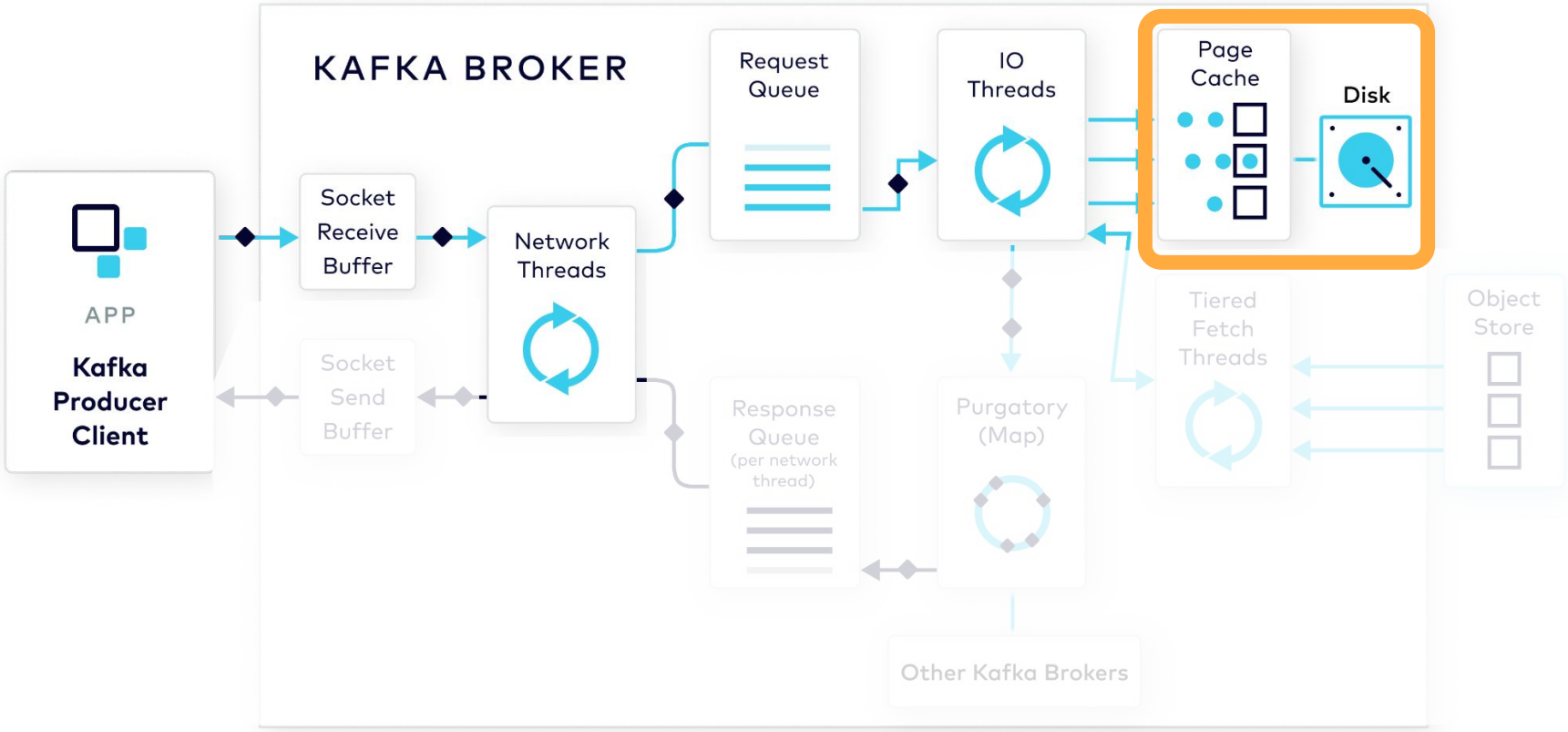


Kafka Physical Storage

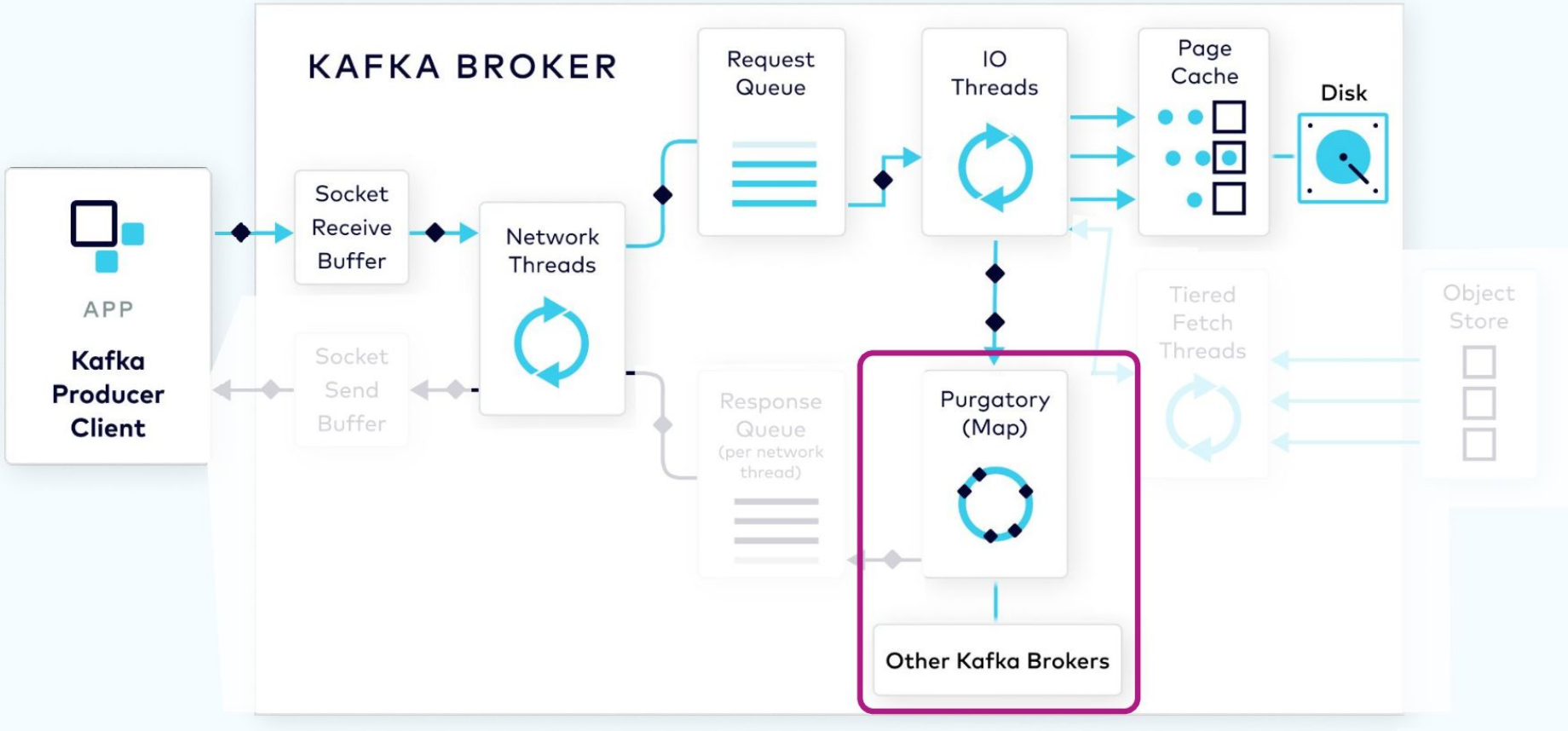



corresponds to a log
like a rolling file
Topic a
Topic b
Topic c
Topic d
Topic e
Topic xx
Kafka Cluster
Partition 0
Partition 1
Partition 2
Partition n
Topic c
Segment 0
Segment 1
Segment 2
Segment n
Partition 1

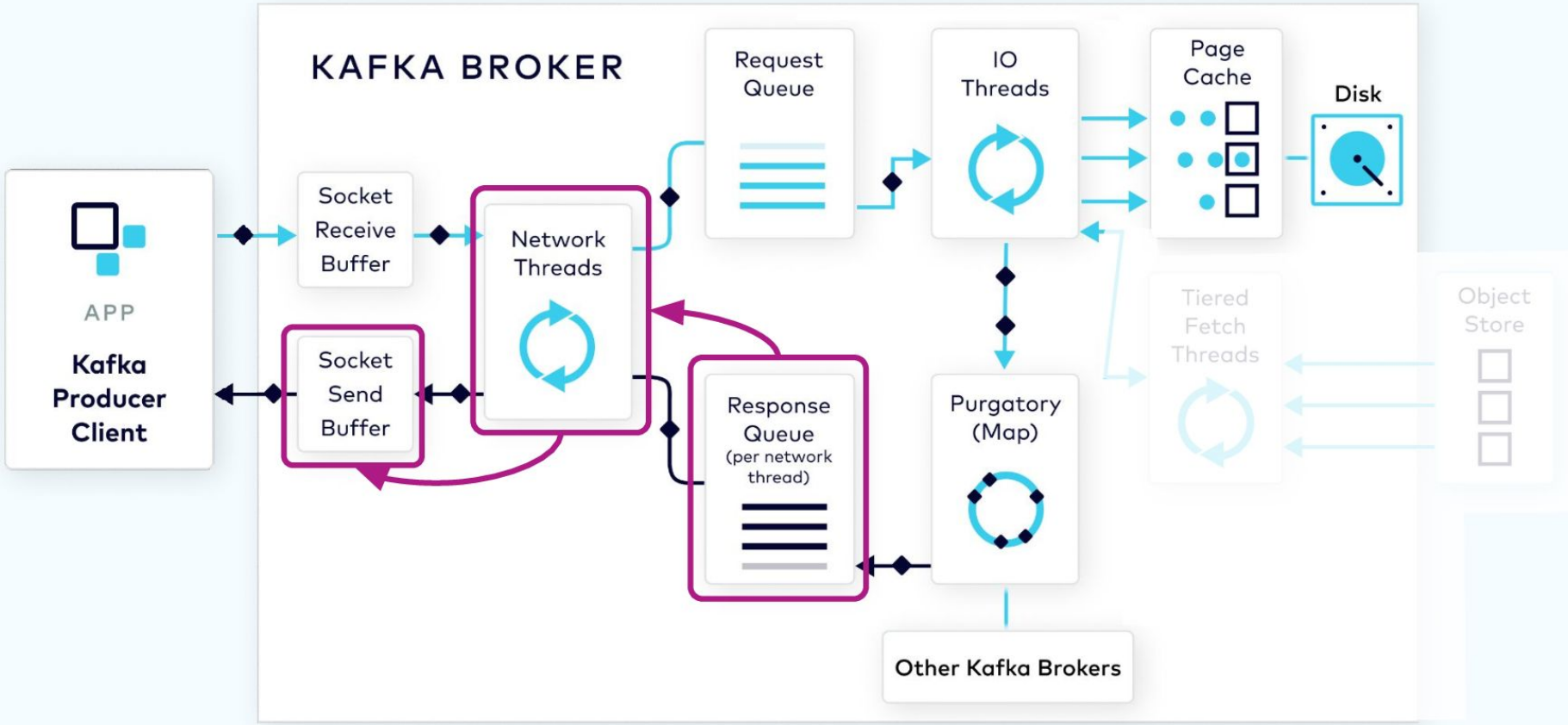
By Default, Data is Written Async to Disk



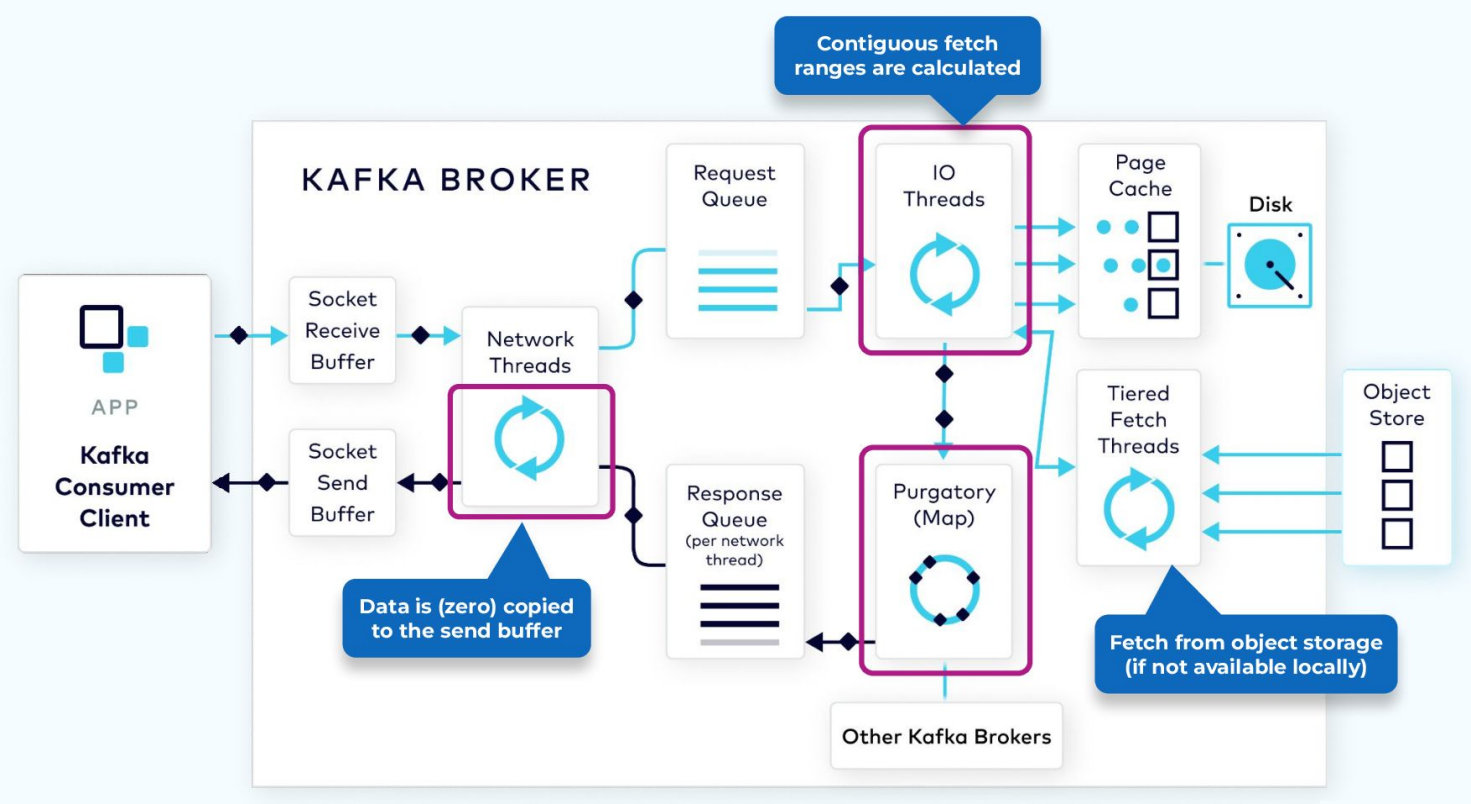
Purgatory Holds Requests Until Replicated



Response Added to Socket



The Fetch Request



Zookeeper

Zookeeper is essentially a service for distributed systems offering a hierarchical key-value store, which is used to provide a distributed configuration service, synchronization service, and naming registry for large distributed systems.

Role of Zookeeper in Kafka

1. Brokers registration, with heartbeat mechanism to keep the broker list updated
2. Maintaining a list of topics
   - ➔ Topic configuration (Partitions, Replication factor, additional configs etc.)
   - ➔ The list of In-sync replicas for Partitions
3. Performs leader election in case any broker is down
4. Store Access Control Lists, if security is enabled
   - ➔ Topics
   - ➔ Consumer groups
   - ➔ Users

Demo-1:

Kafka Cluster & Producer Client

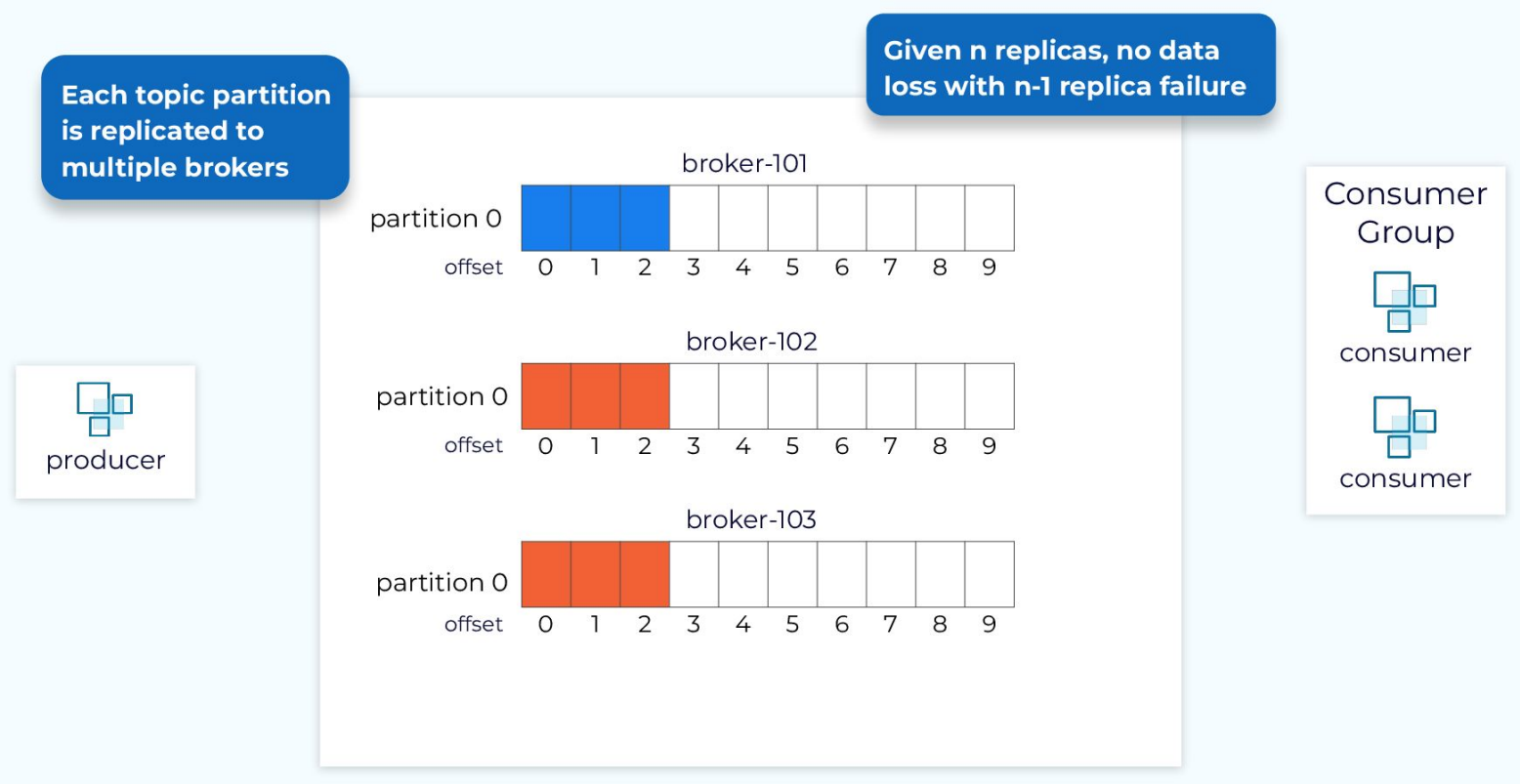
Any questions?

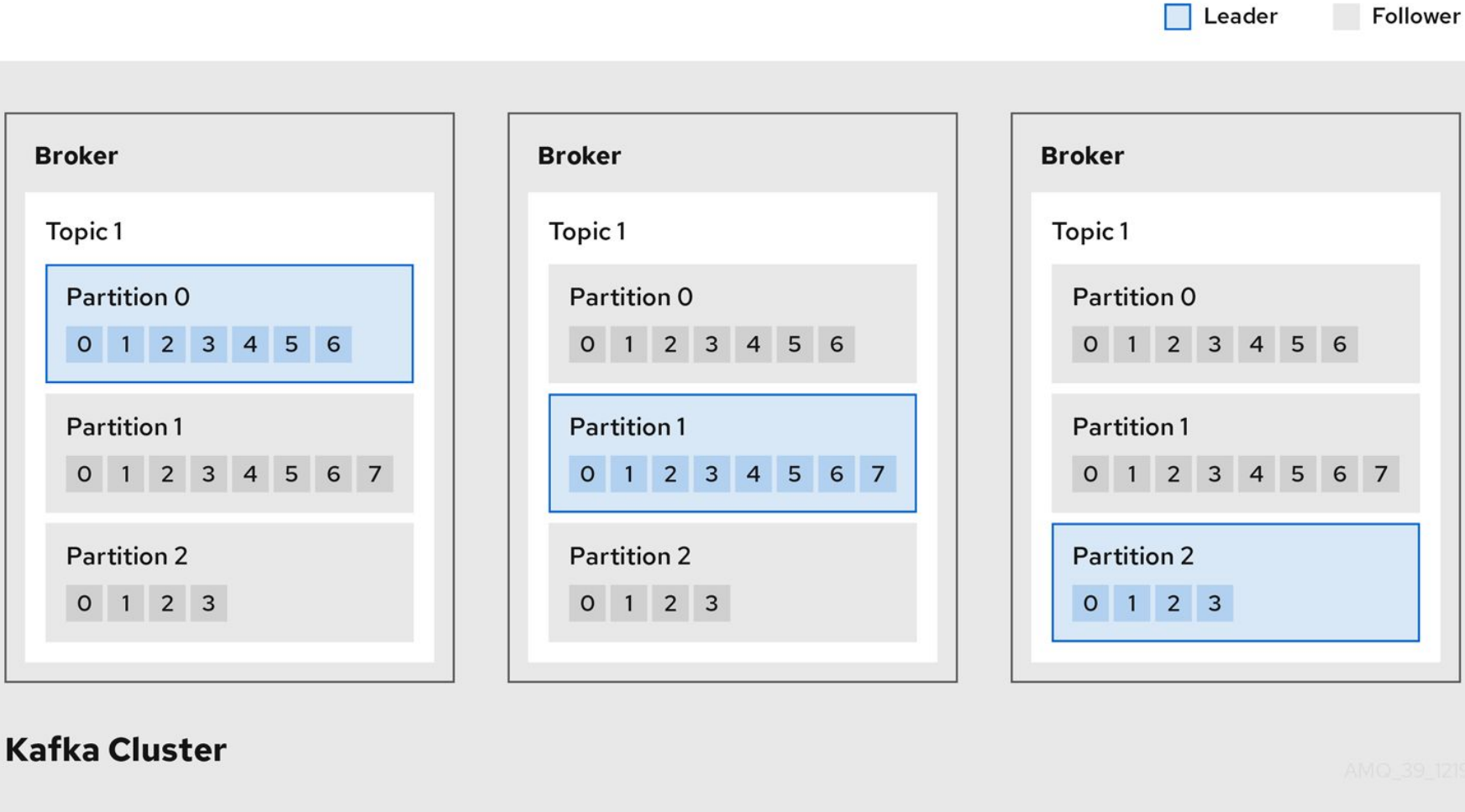
Data Plane: Replication Protocol

Replication

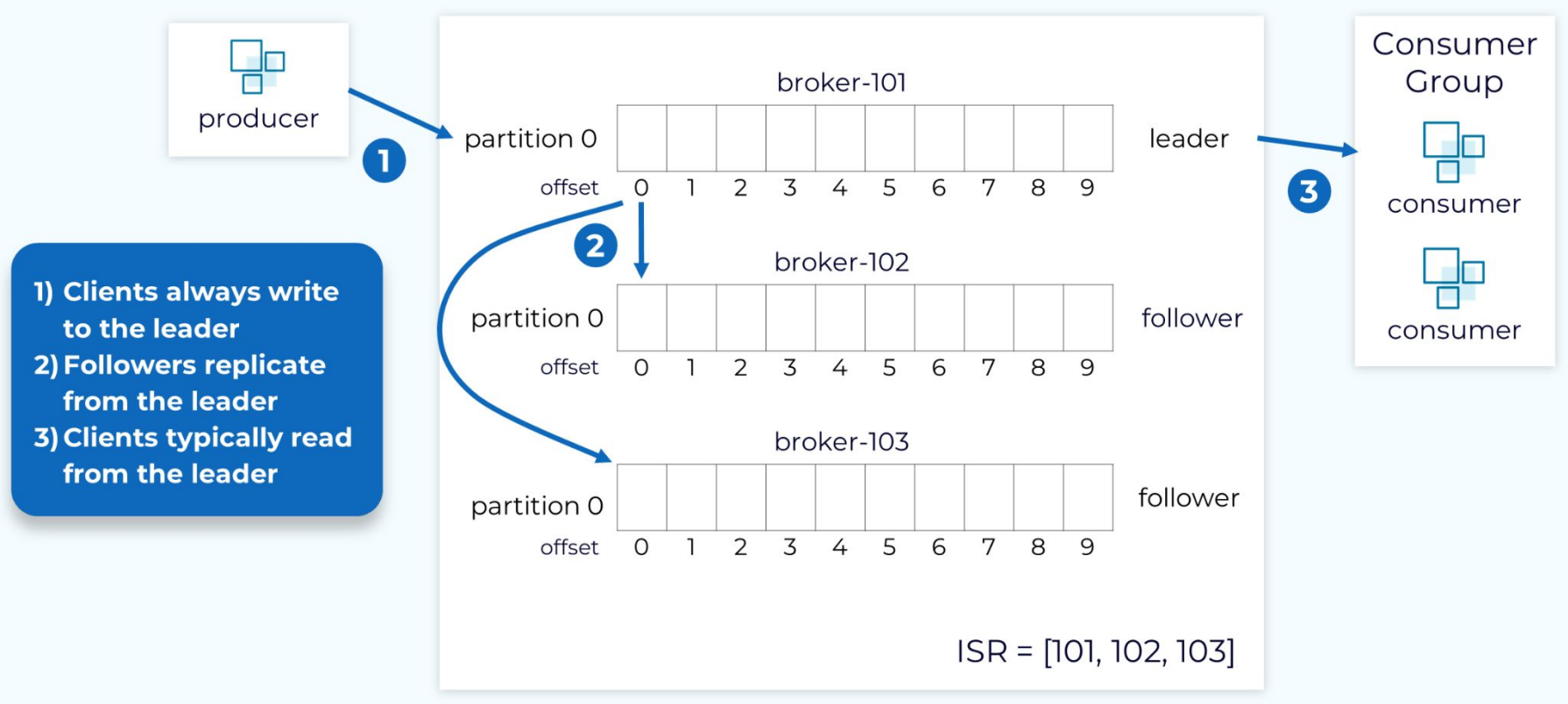
- Copies of data for fault tolerance
- One lead partition and N-1 followers
- In general, writes and reads happen to the leader
- An invisible process to most developers
- Tunable in the producer

Kafka Data Replication

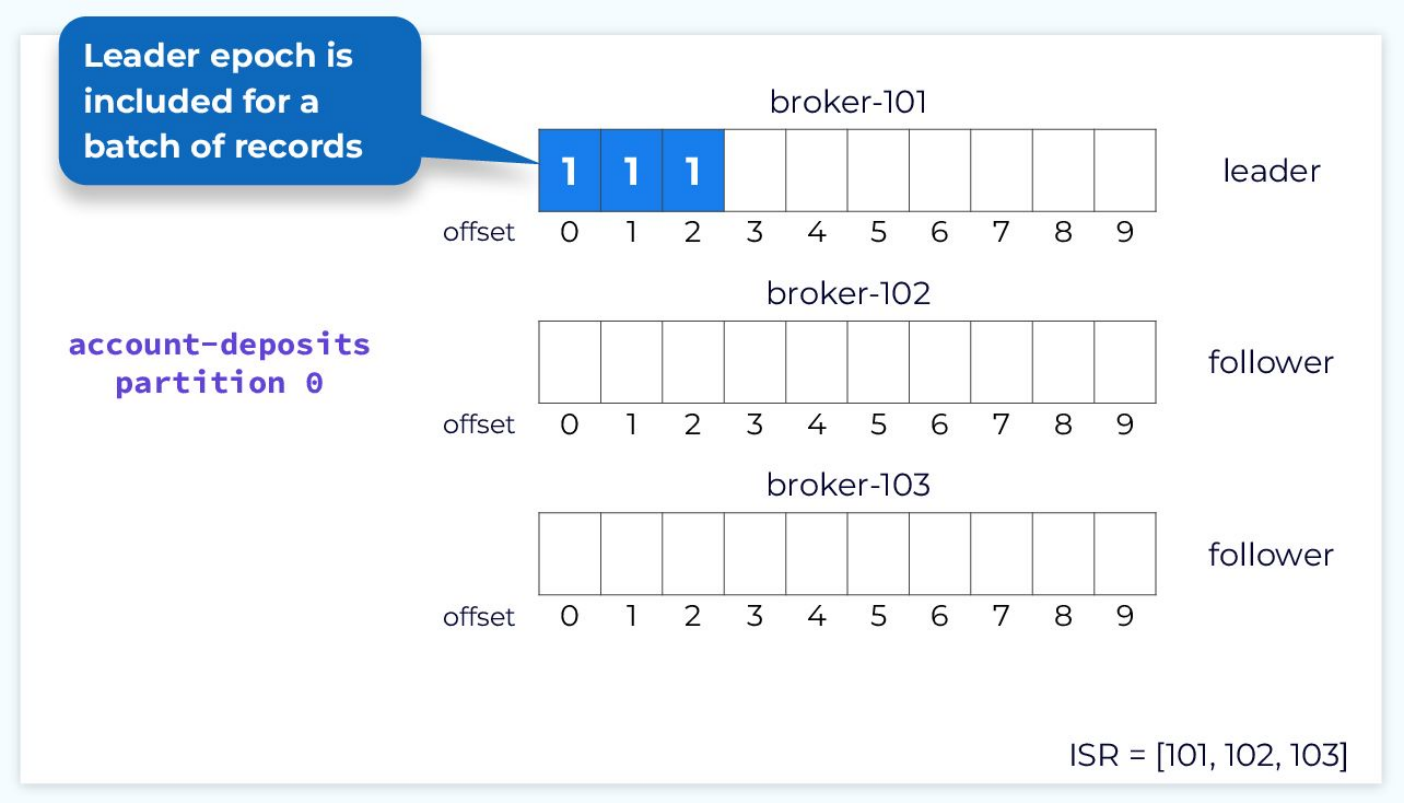



Leader
Follower
Broker
Topic 1
Partition 0
0 1 2 3 4 5 6
Partition 1
0 1 2 3 4 5 6 7
Partition 2
0 1 2 3
Broker
Topic 1
Partition 0
0 1 2 3 4 5 6
Partition 1
0 1 2 3 4 5 6 7
Partition 2
0 1 2 3
Broker
Topic 1
Partition 0
0 1 2 3 4 5 6
Partition 1
0 1 2 3 4 5 6 7
Partition 2
0 1 2 3
Kafka Cluster
AMQ_39_1219

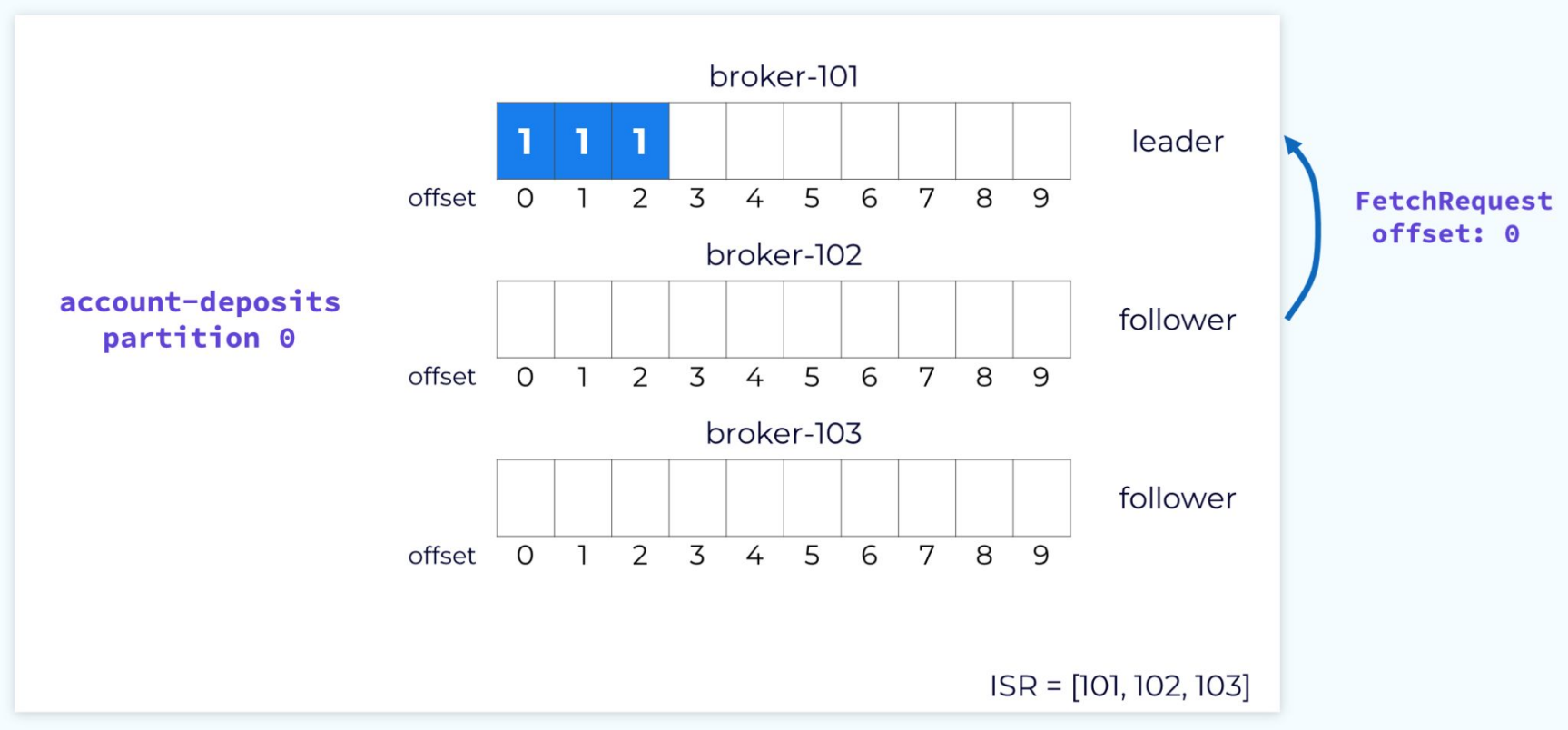
Leader,Follower, and In-Sync Replica (ISR) List



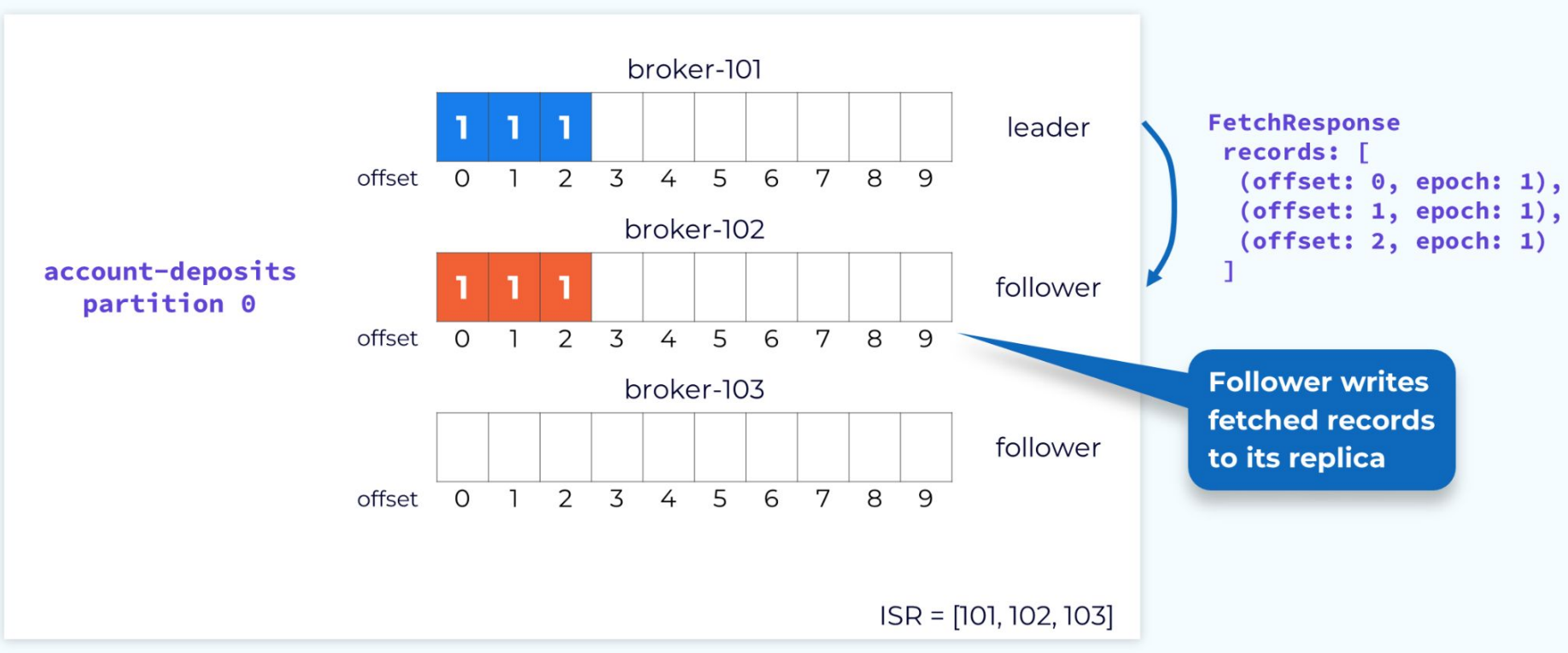
Leader Epoch



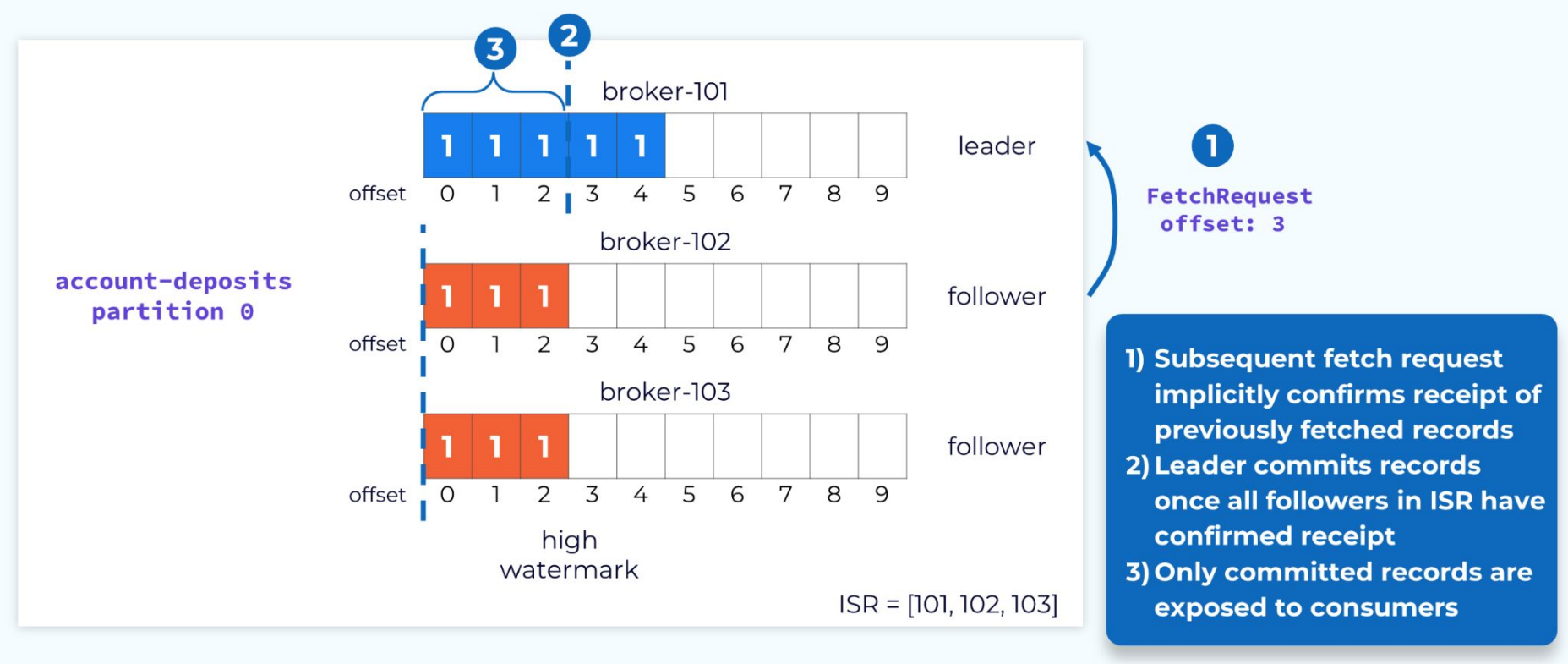
Follower Fetch Request



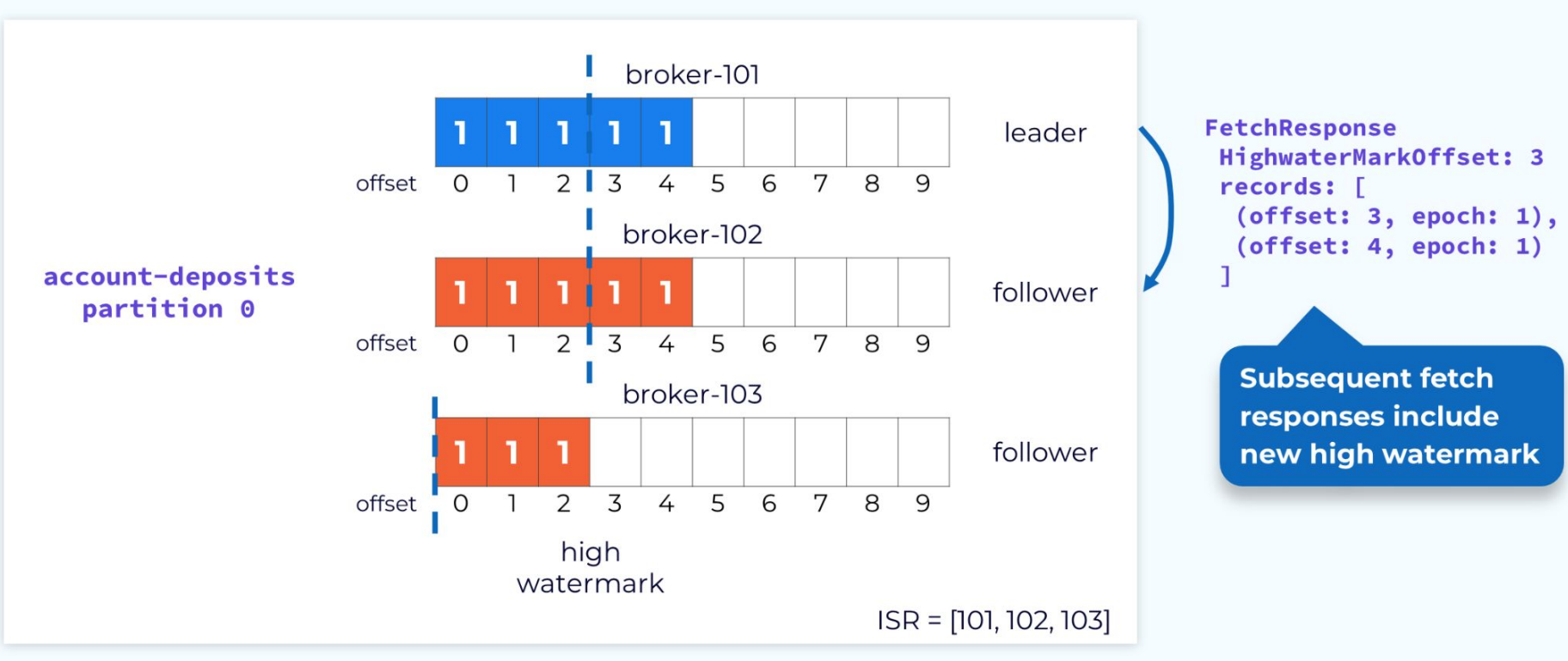
Follower Fetch Response



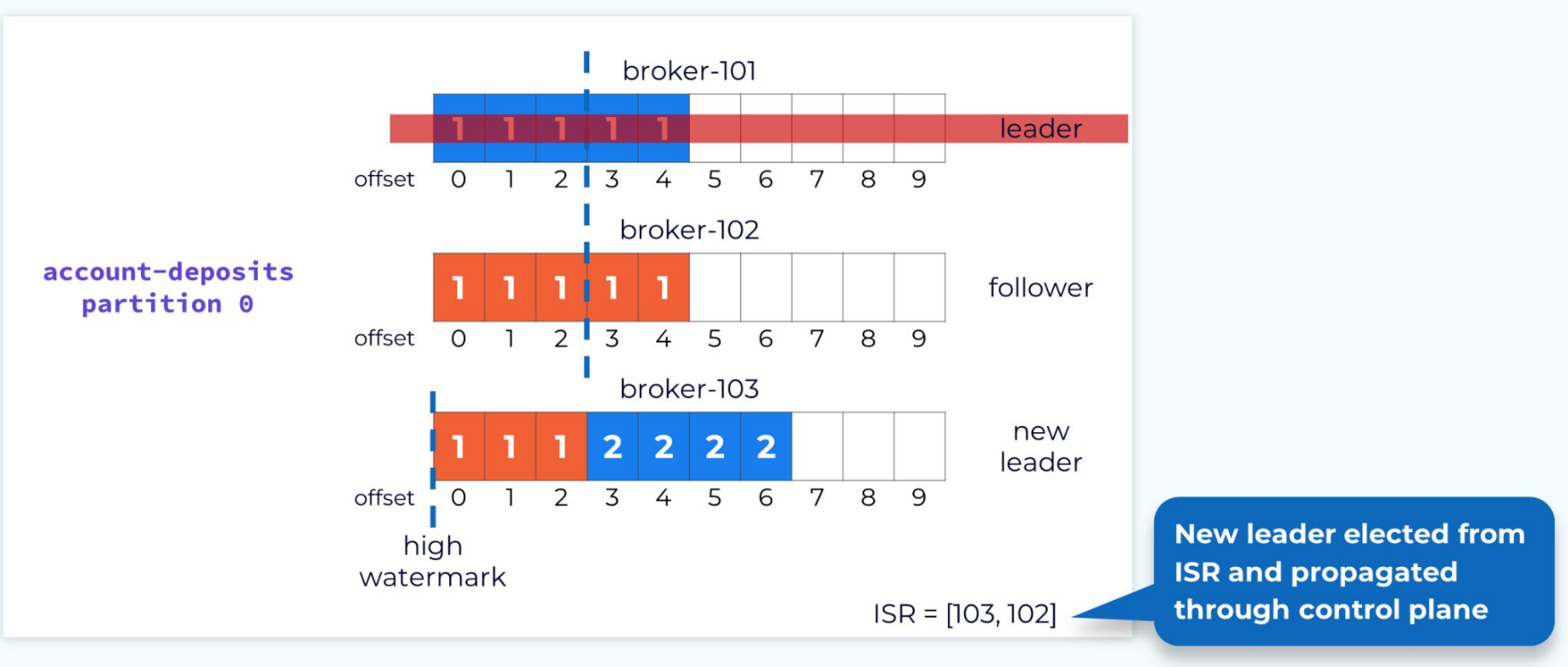
Committing Partition Offsets



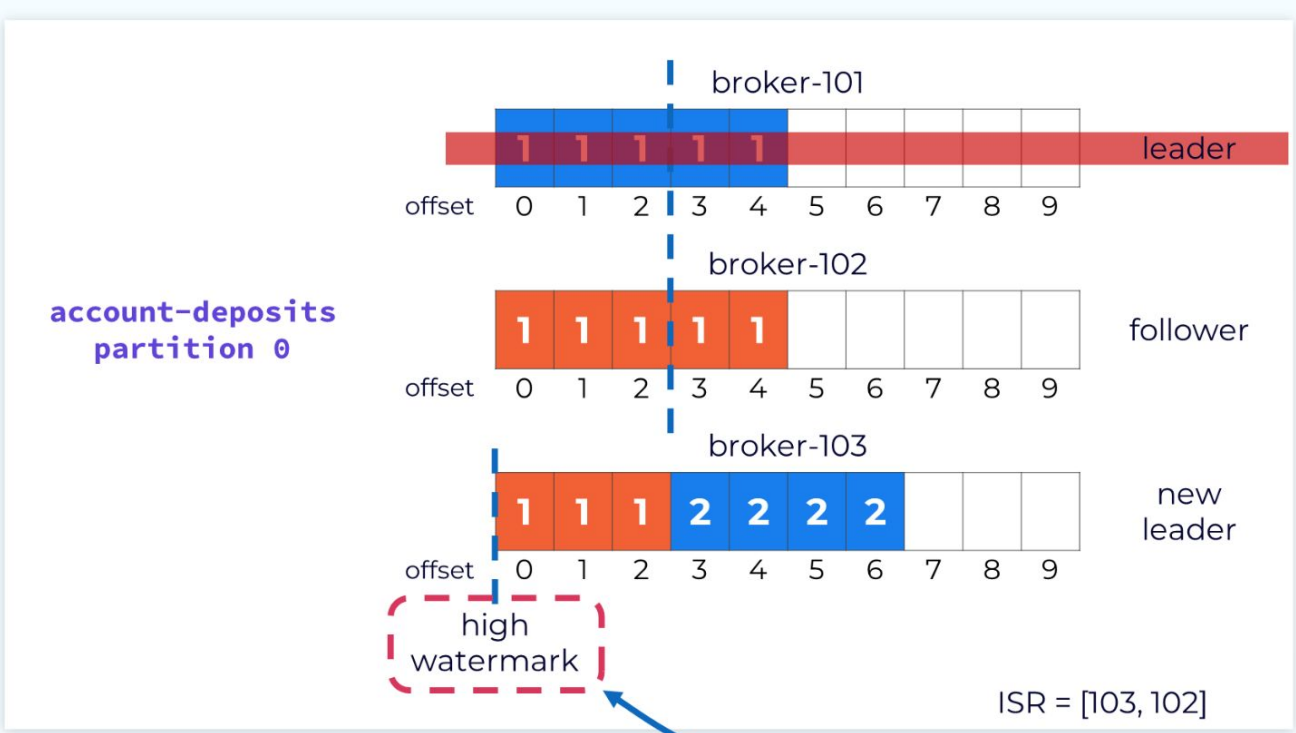
Advancing the Follower High Watermark



Handling Leader Failure



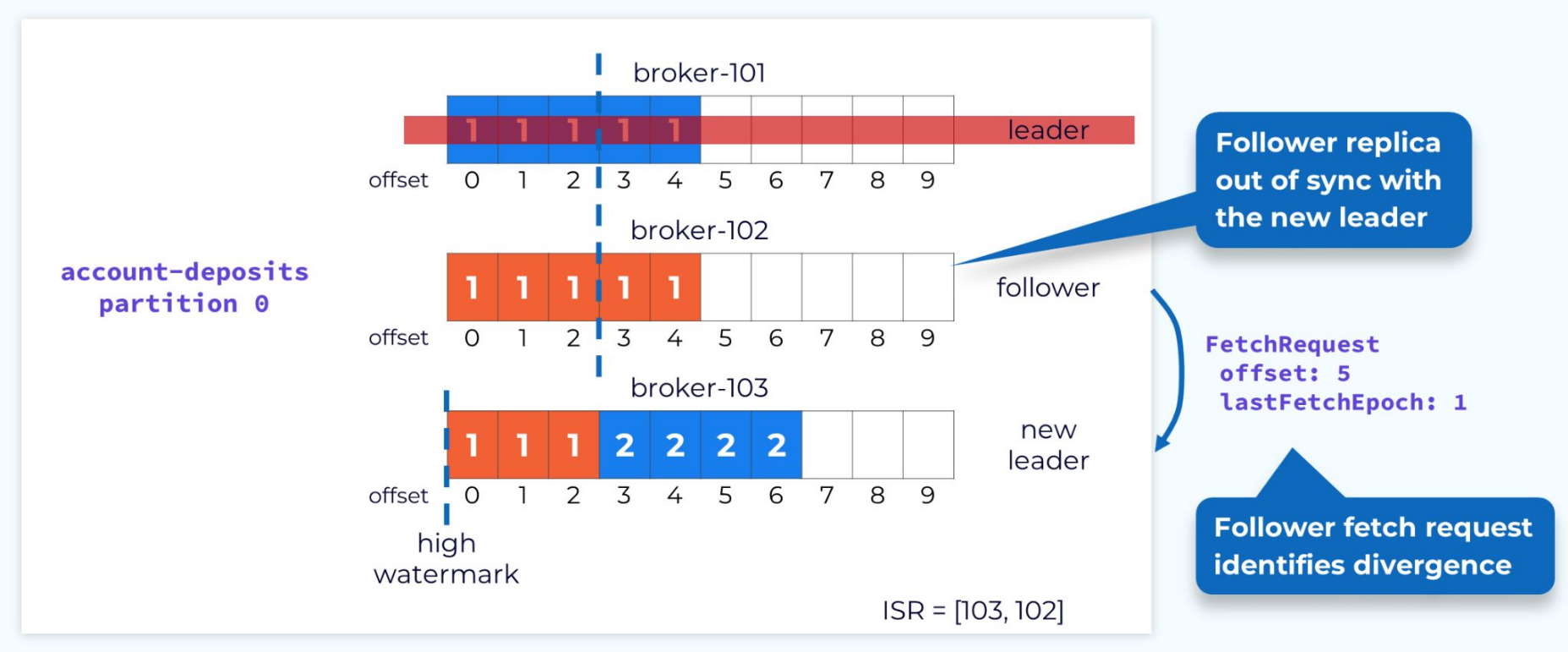
Temporary Decreased High Watermark



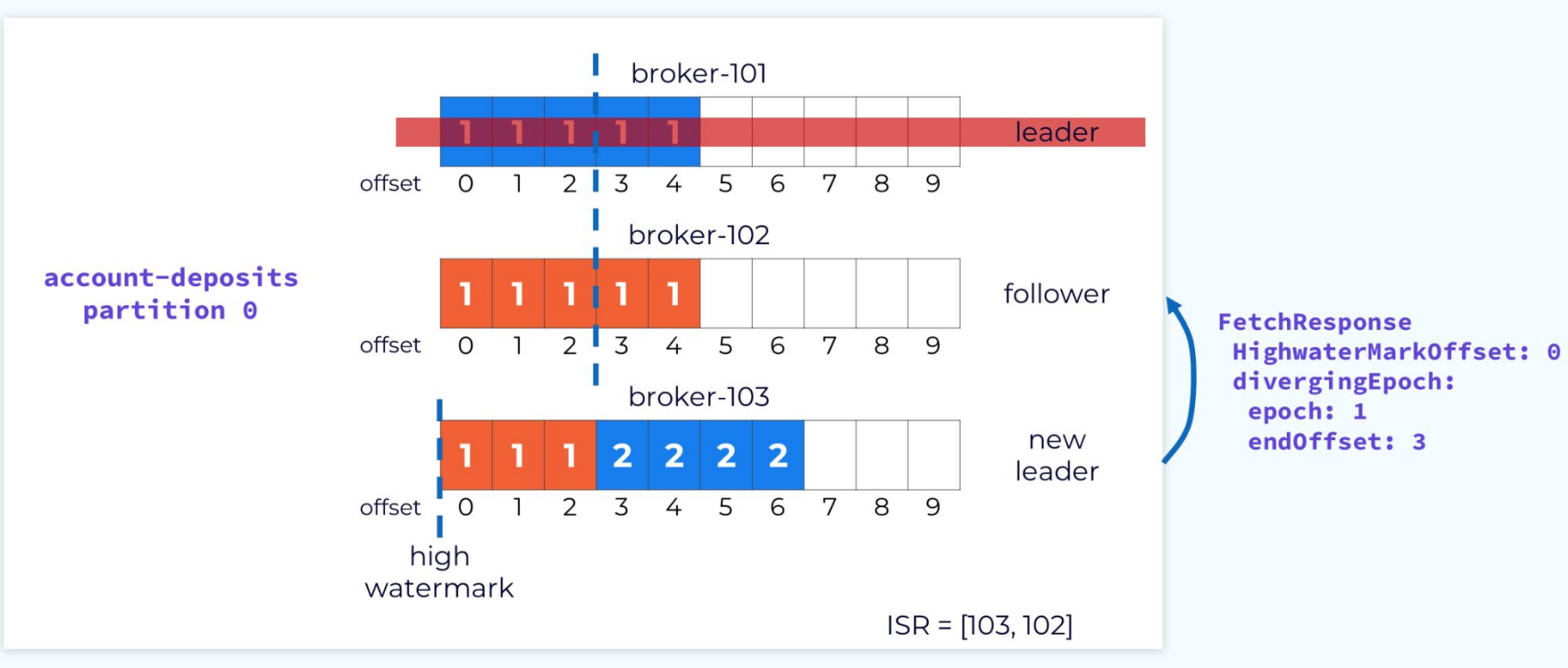
New leader high watermark could be less than true high watermark

- **If consumer tries to read data in between, it gets a retriable OFFSET_NOT_AVAILABLE error**
- **Once high watermark catches up, normal consumption continues**

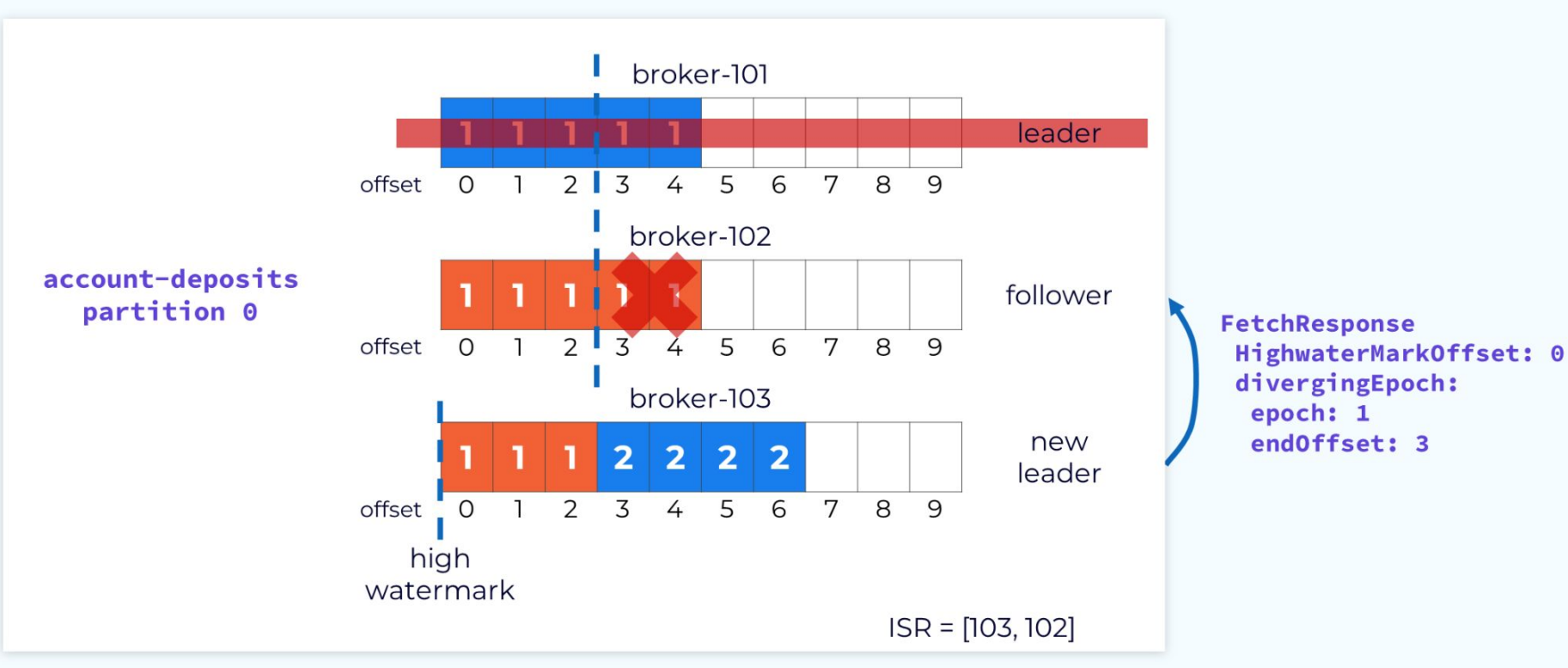
Partition Replica Reconciliation



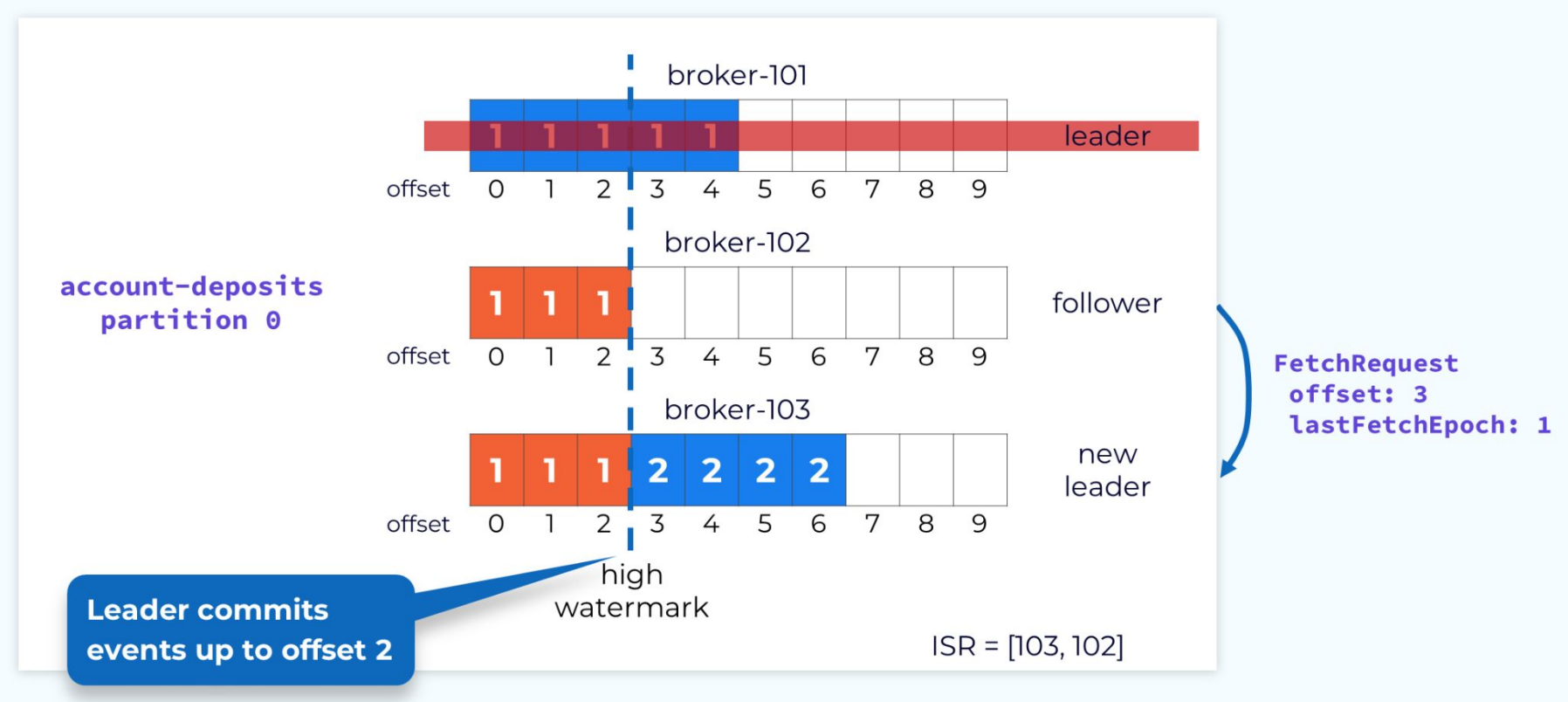
Fetch Response Informs Follower of Divergence



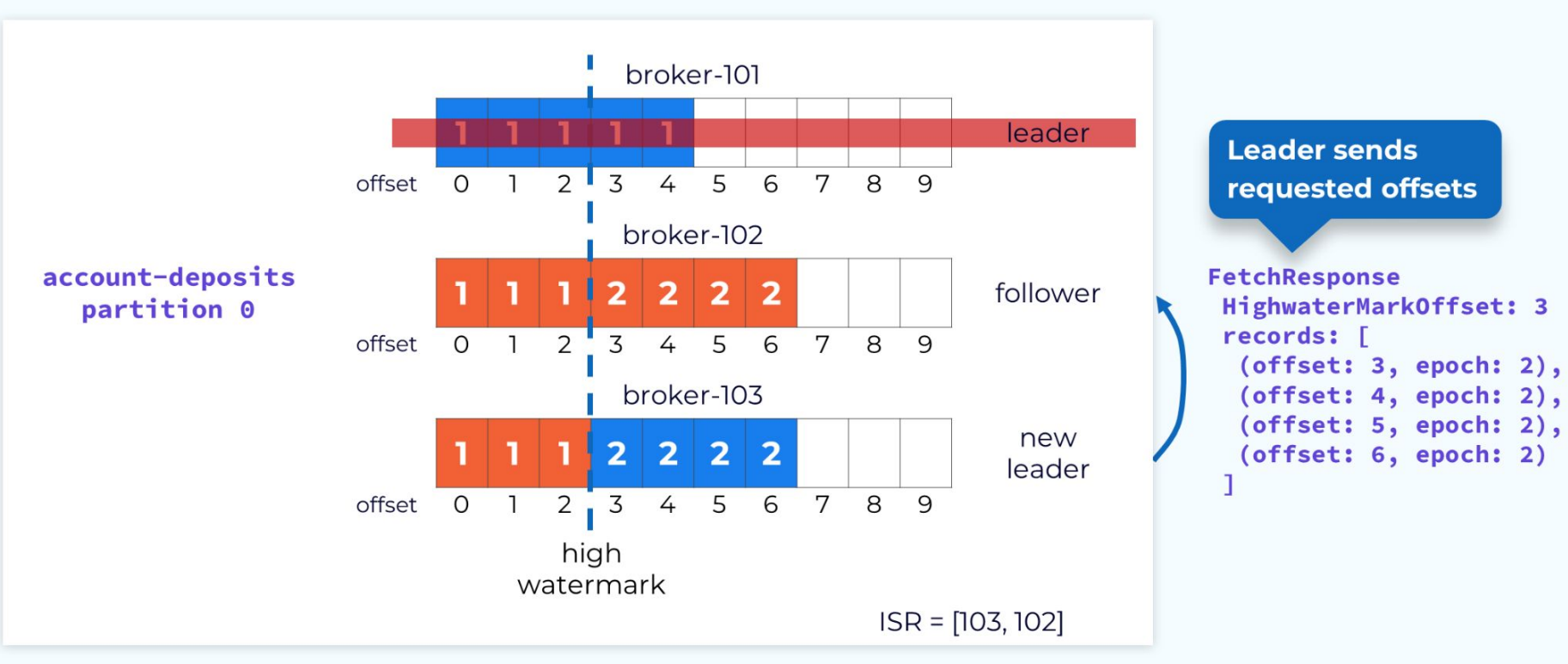
Follower Truncates Log to Match Leader log



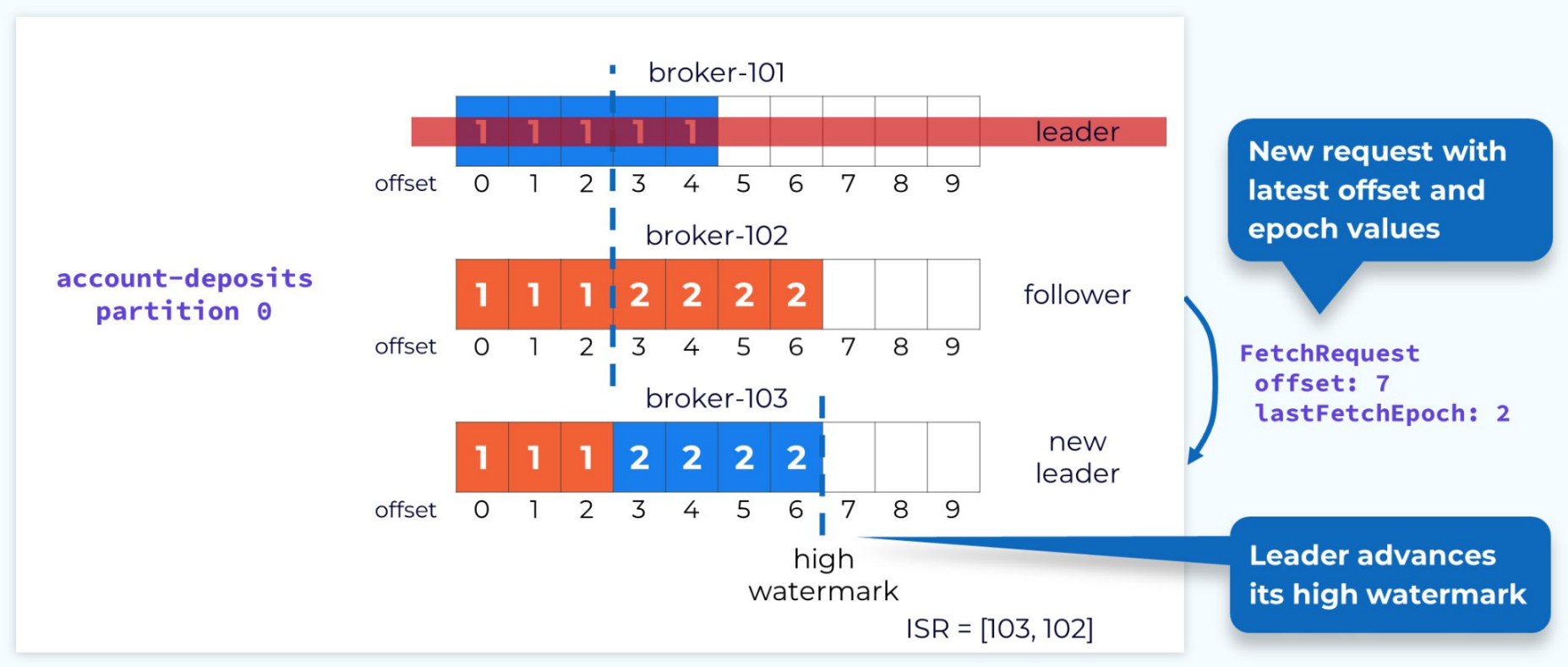
Subsequent Fetch with Updated Offset and Epoch



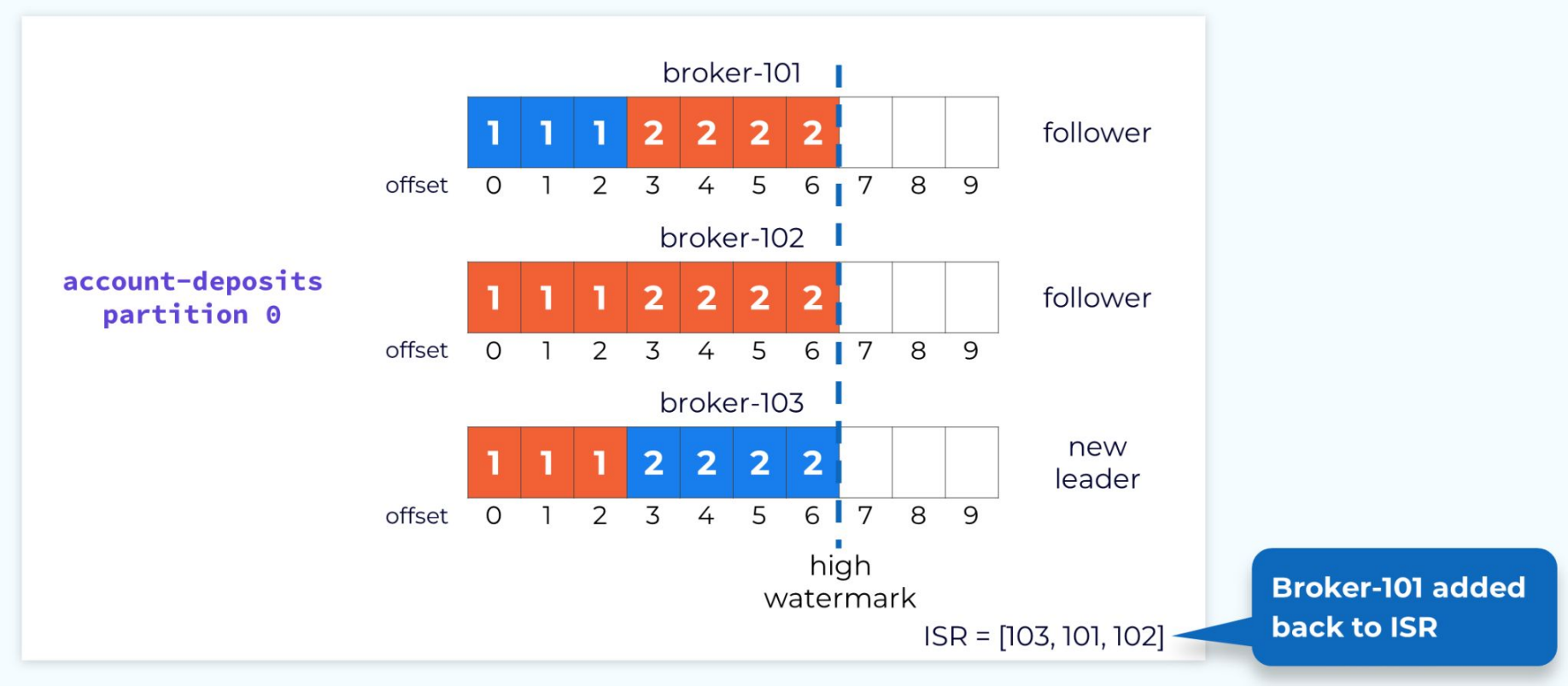
Follower 102 Reconciled



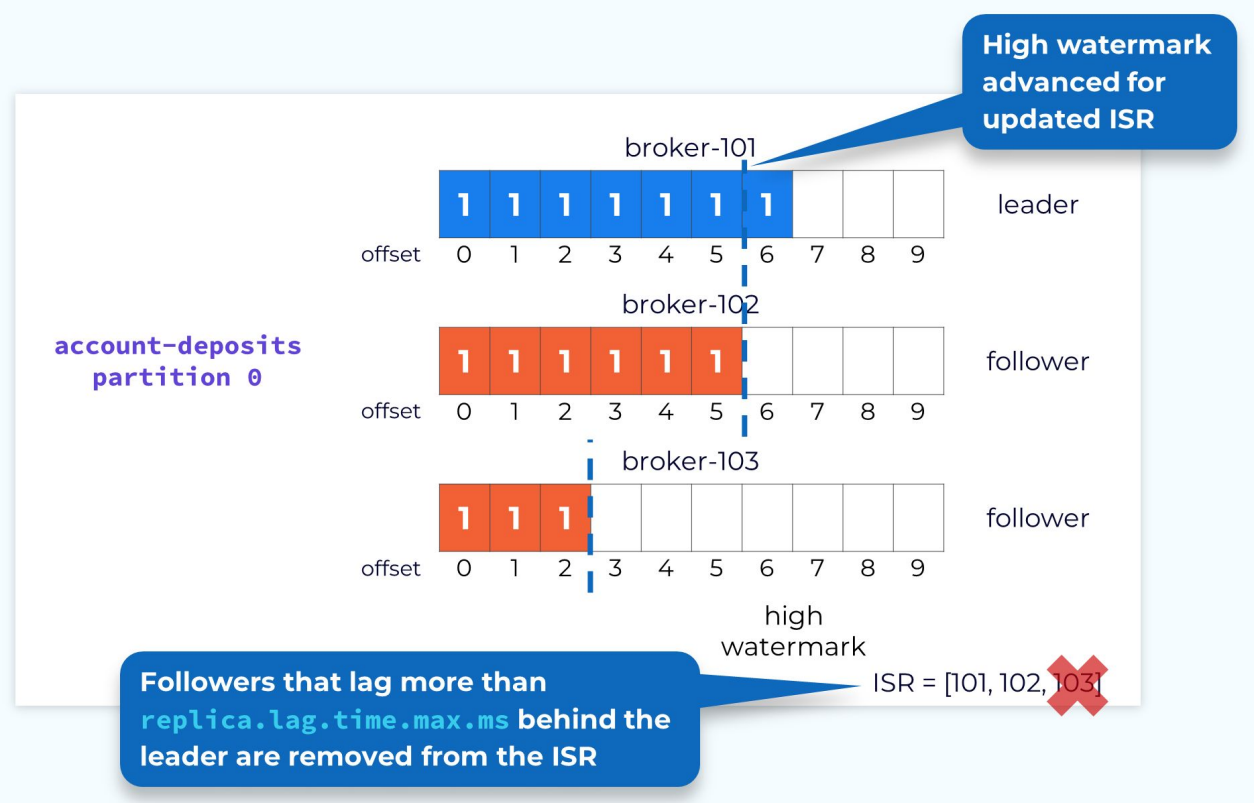
Follower 102 Acknowledges New Records



Follower 101 Rejoins the Cluster

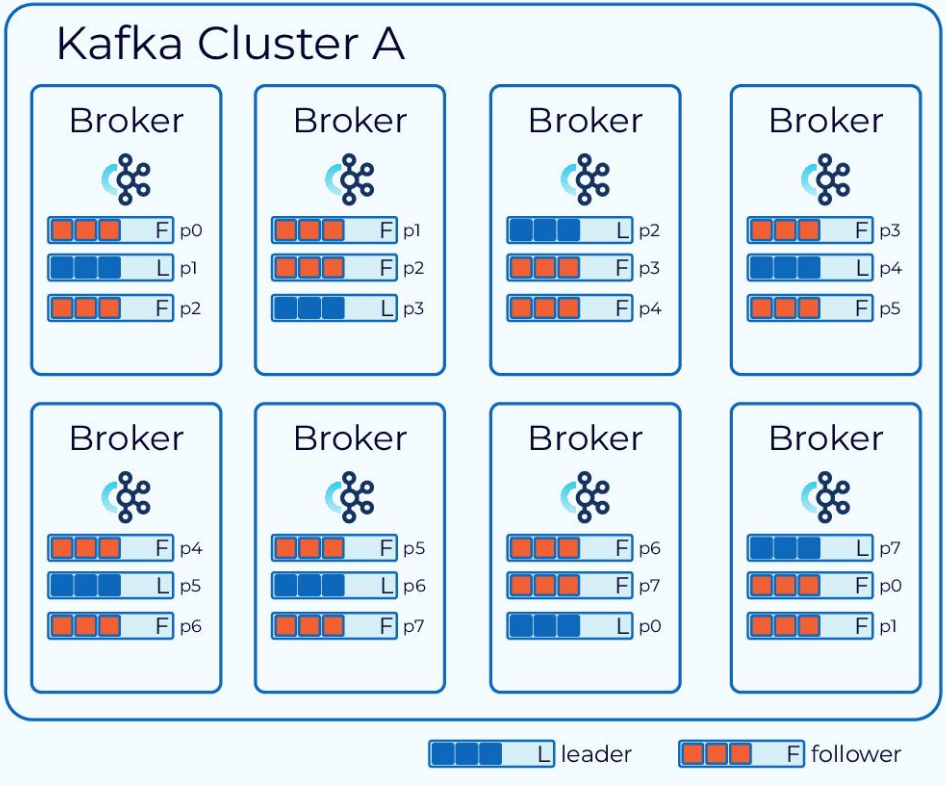


Handling Failed or Slow Followers



Partition Leader Balancing

- First replica considered preferred
- Preferred replica distributed evenly during assignment
- Background thread moves leader to preferred replica when it's in-sync



Demo-2:

Replication Protocol

Any questions?

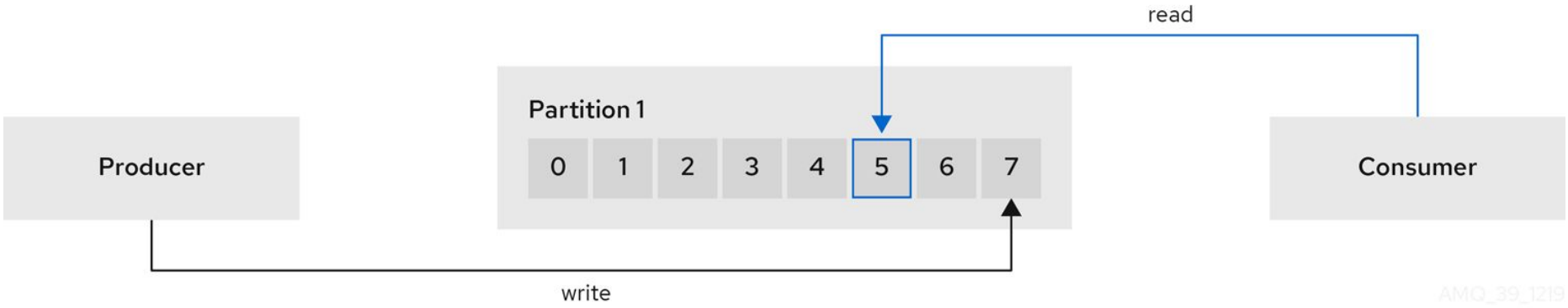
Consumer Group Protocol

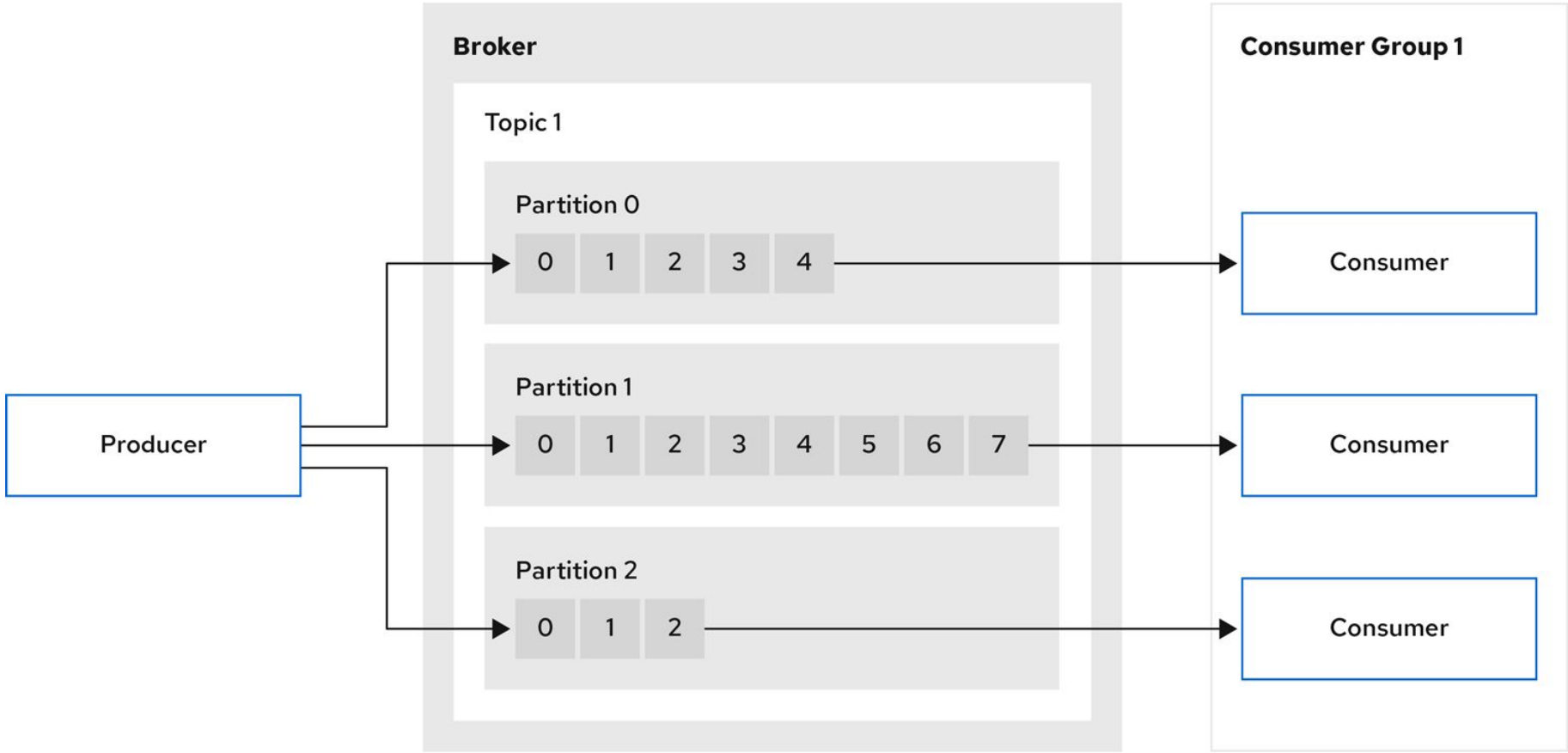
Consumer Group Protocol

Kafka separates **storage** from **compute**.

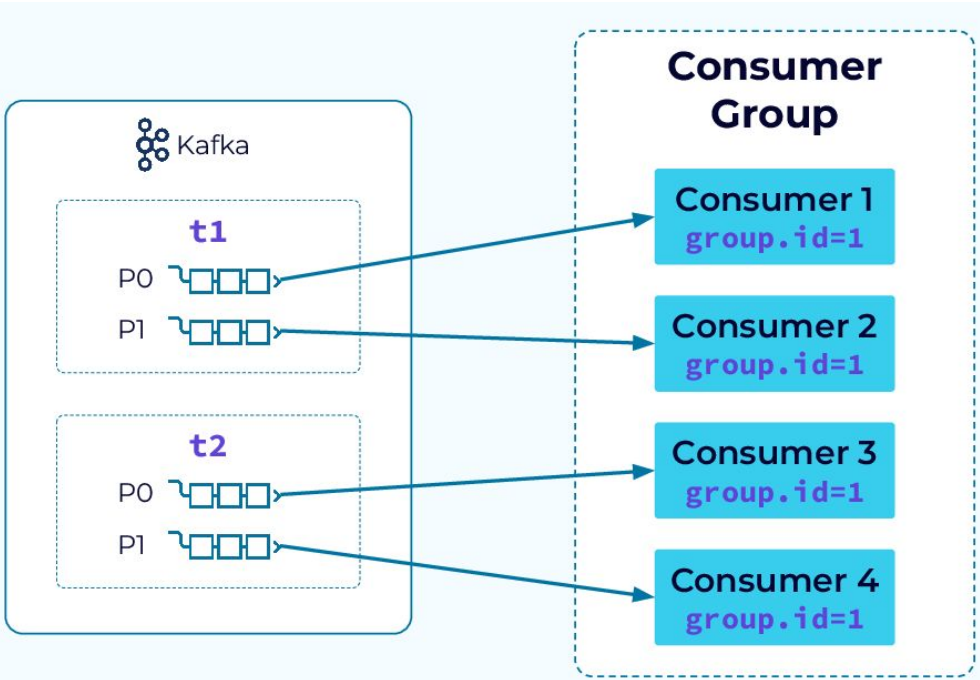
Storage is handled by the brokers and compute is mainly handled by consumers or frameworks built on top of consumers (Kafka Streams, ksqlDB).

Consumer groups play a key role in the effectiveness and scalability of Kafka consumers


read
Partition 1
0
1
2
3
4
5
6
7
Producer
Consumer
write
AMQ_39_1219


Broker
Topic 1
Partition 0
0
1
2
3
4
Partition 1
0
1
2
3
4
5
6
7
Partition 2
0
1
2
Producer
Consumer Group 1
Consumer
Consumer
Consumer

Kafka Consumer Group



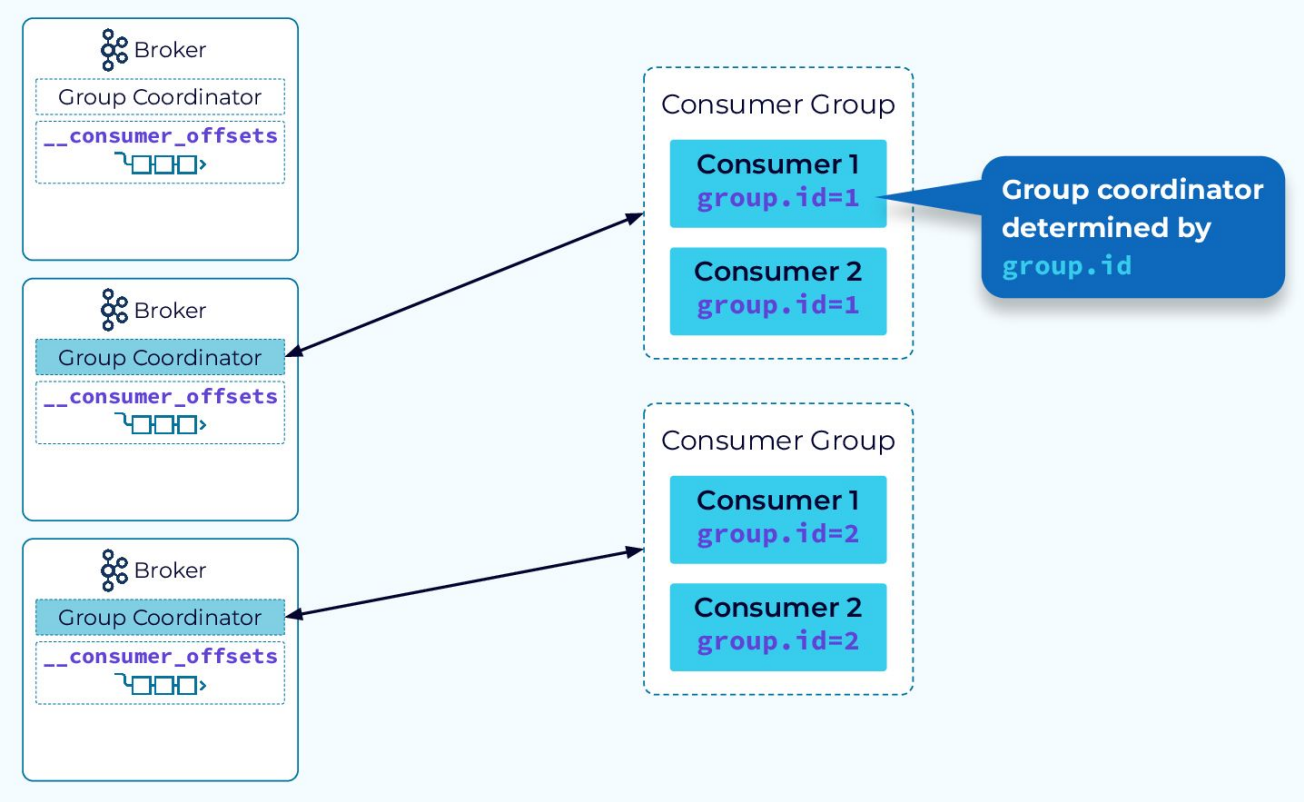
To enable group consumption:

1) Configure `group.id`
2) `topics={"t1", "t2"};`
   `consumer.subscribe(topics)`

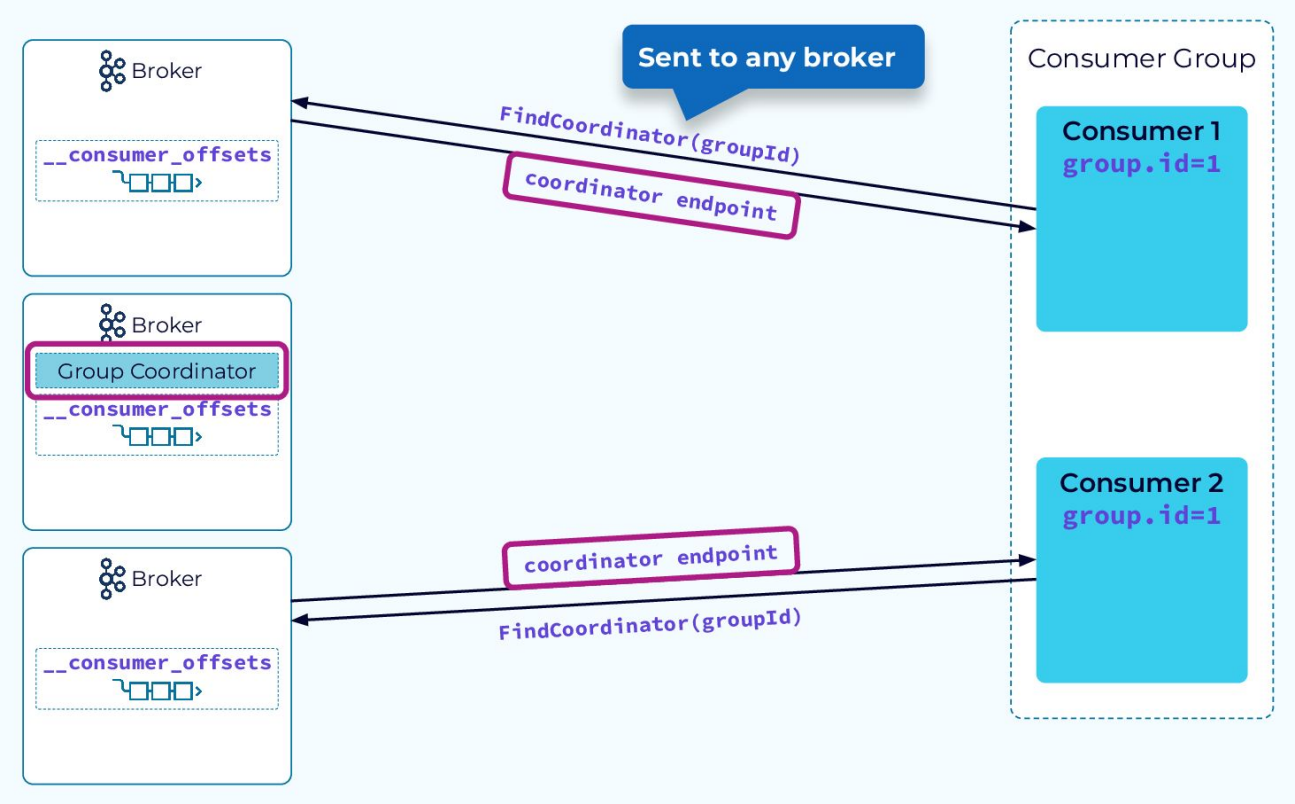
Benefits:

1) Scalability
2) Elasticity
3) Fault tolerance

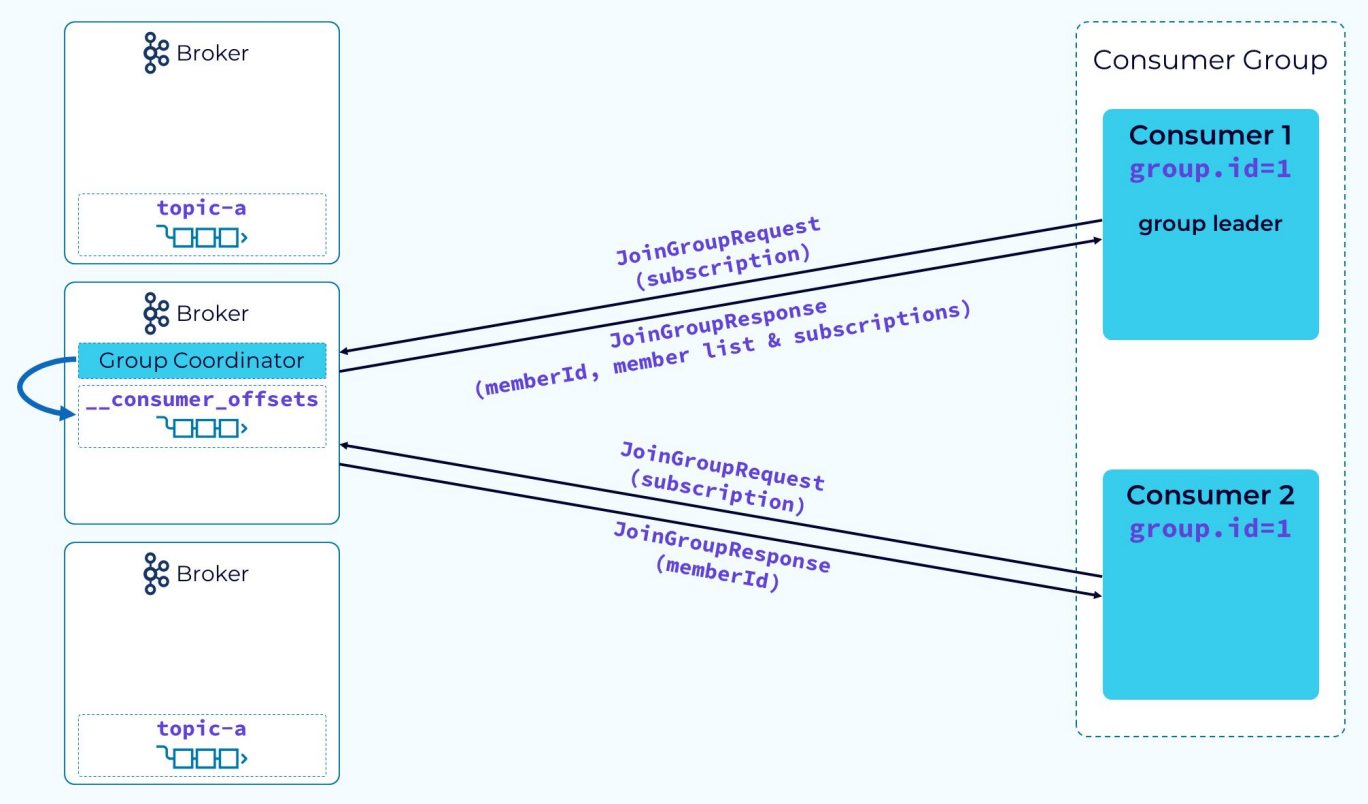
Group Coordinator



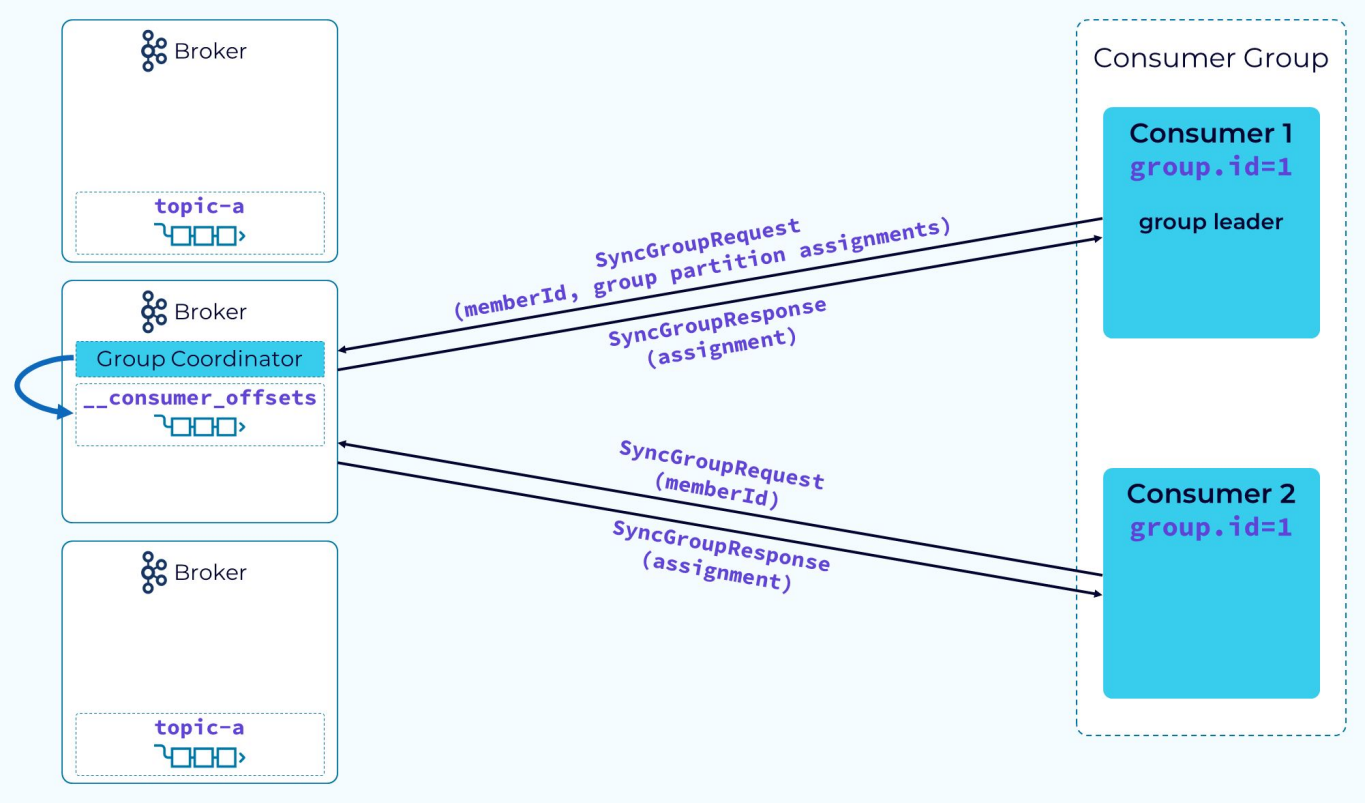
Group Startup: step-1: Find Group coordinator



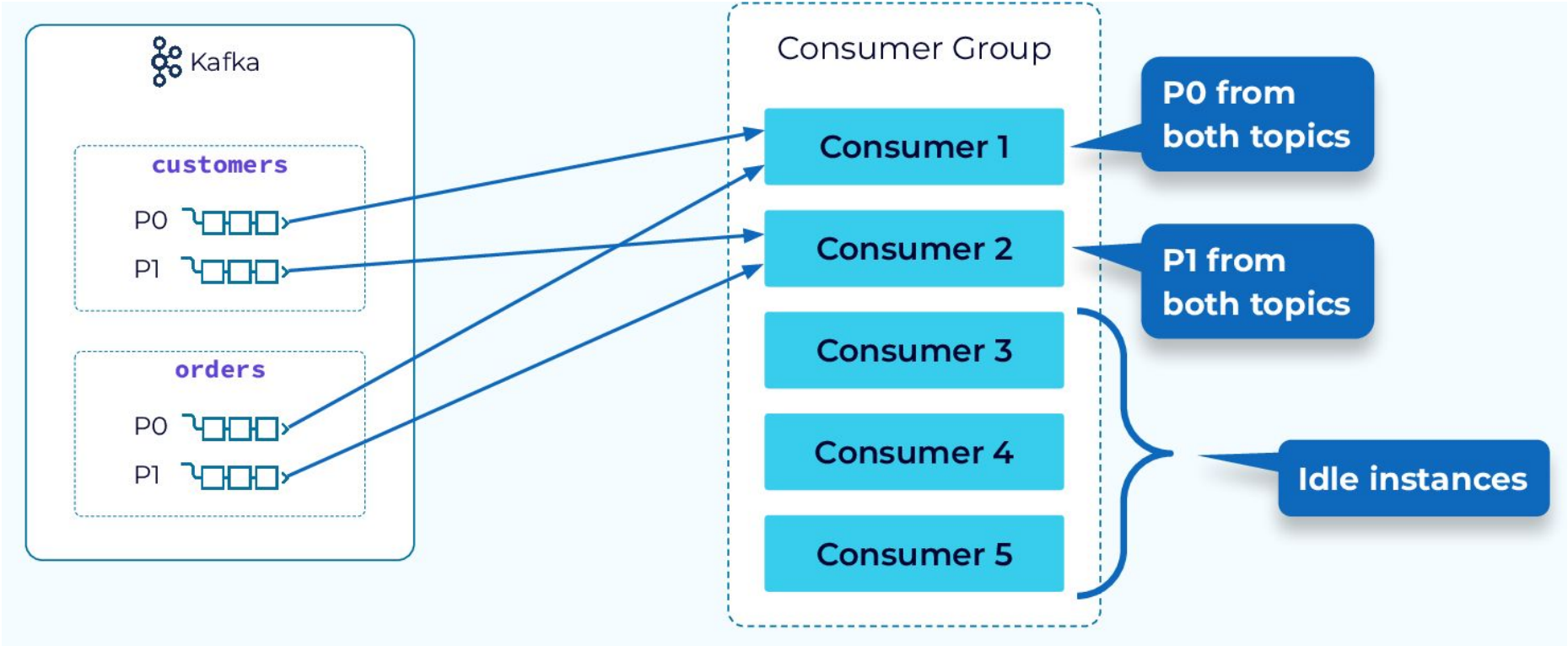
Group Startup: step- 2: Members Join



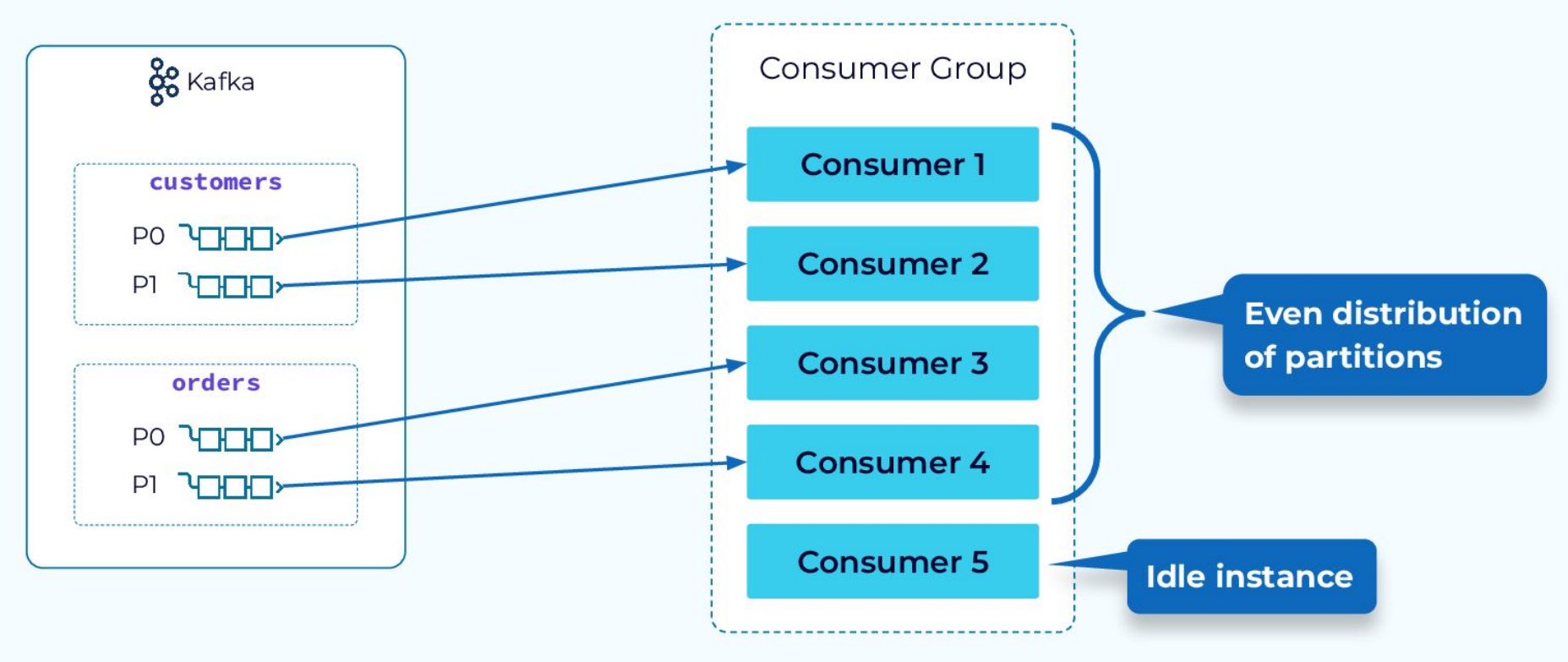
Group Startup: step- 3: Partitions Assigned



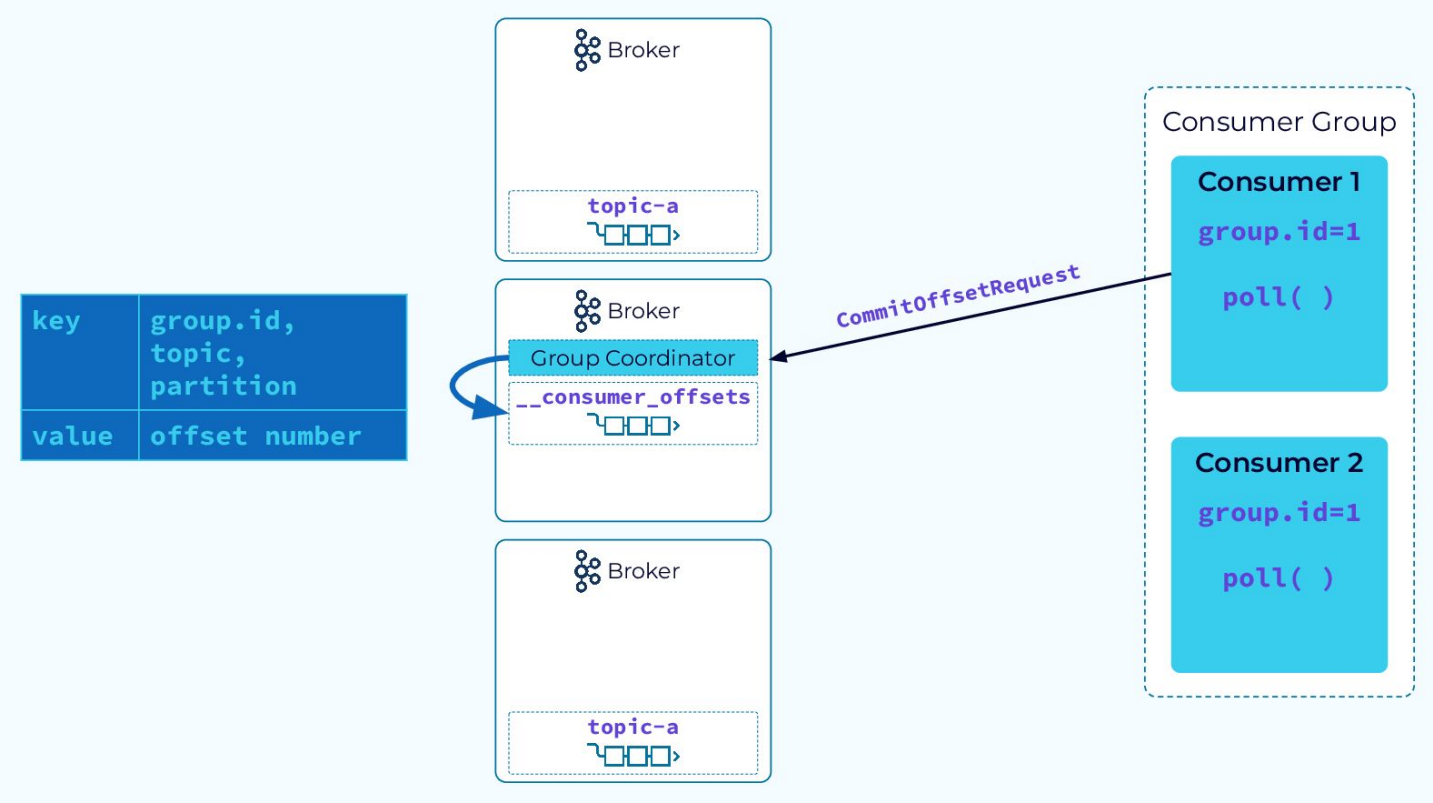
Range Partition Assignment Strategy



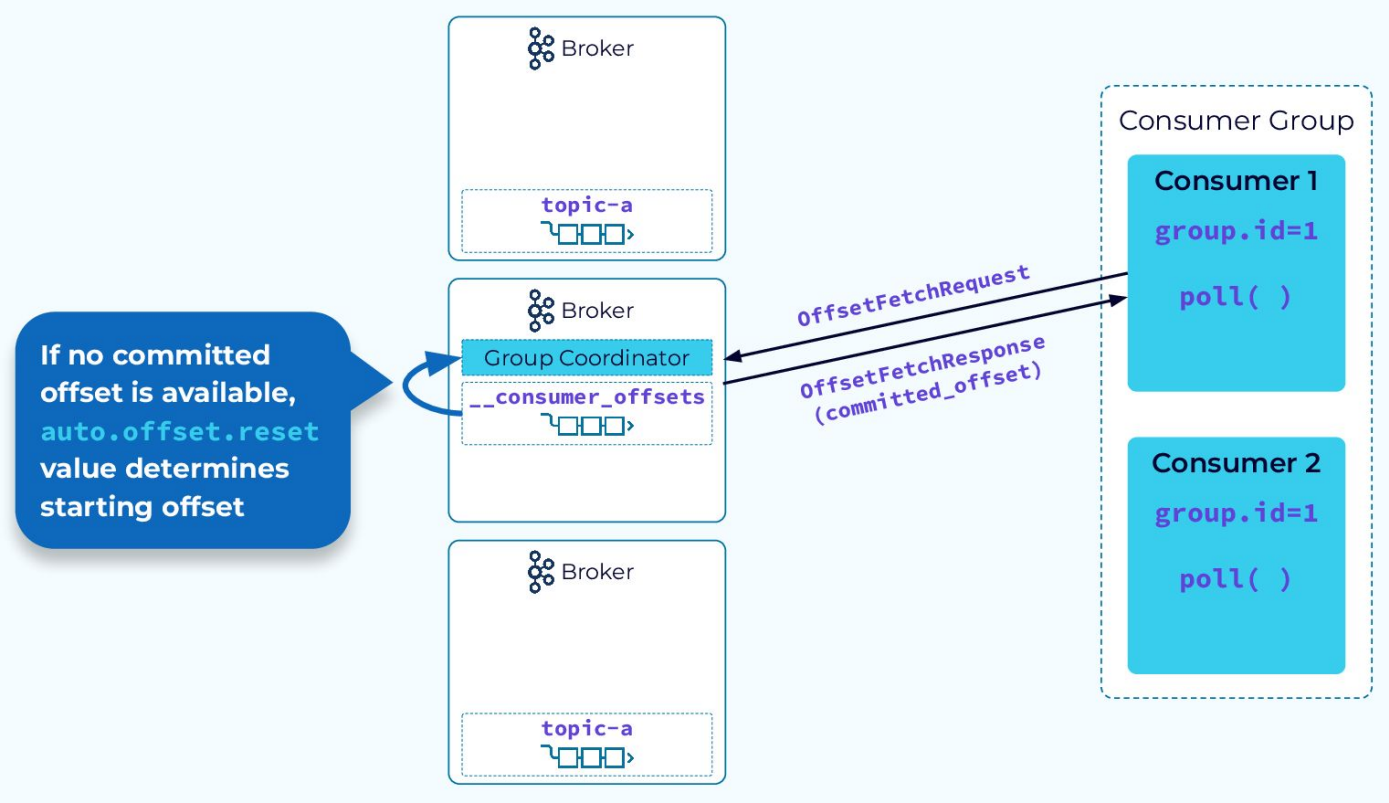
Round Robin & Sticky Partition Assignment



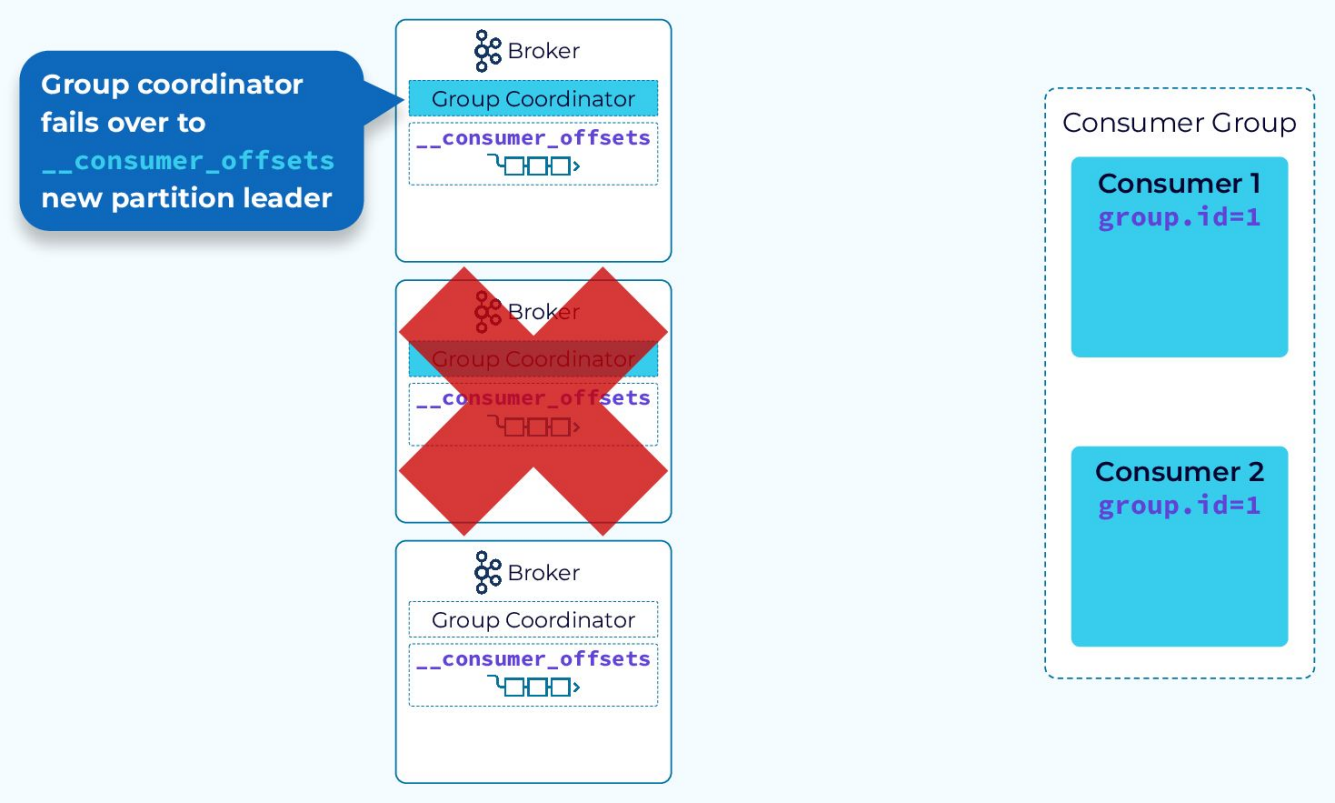
Tracking Partition Consumption



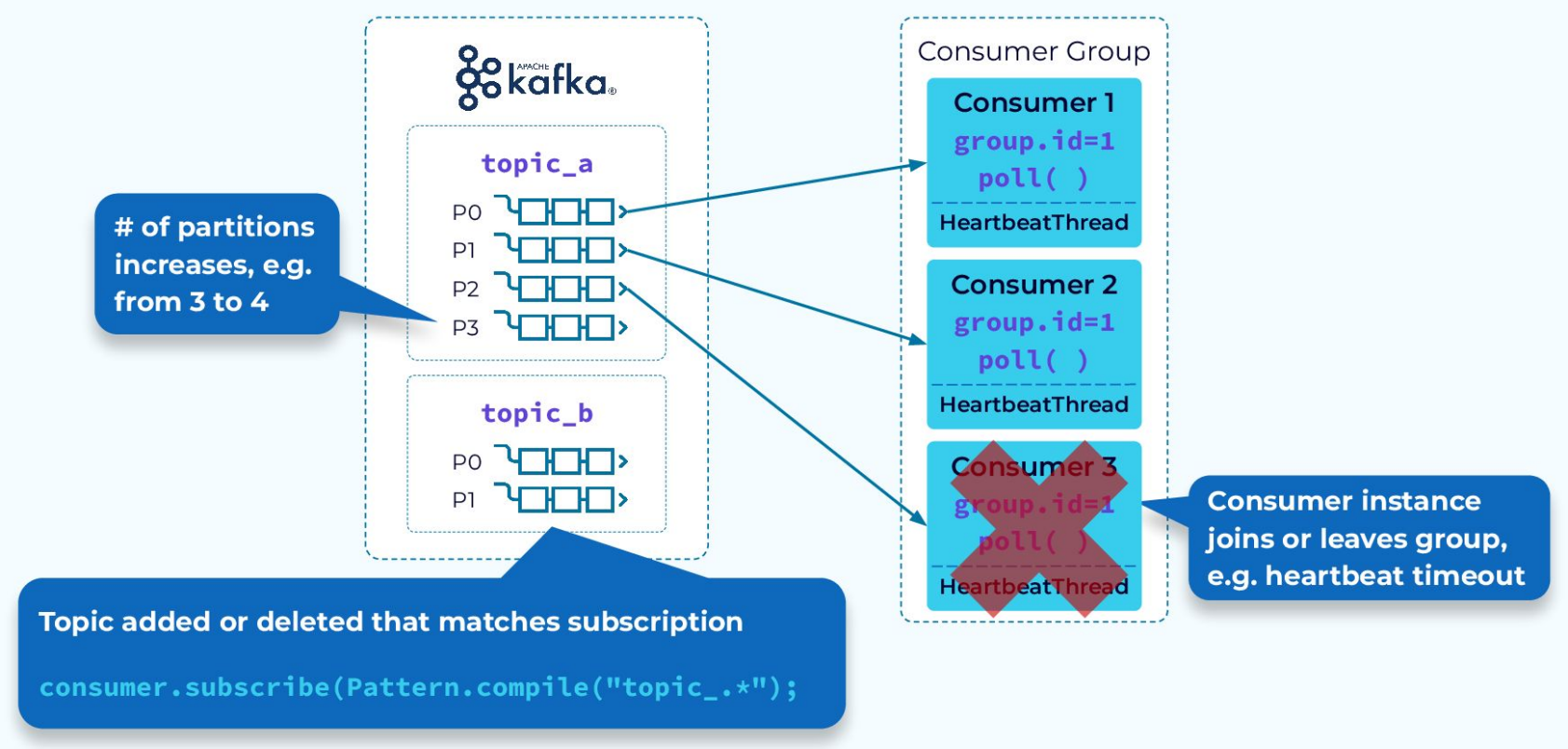
Determining Starting Offset to Consume



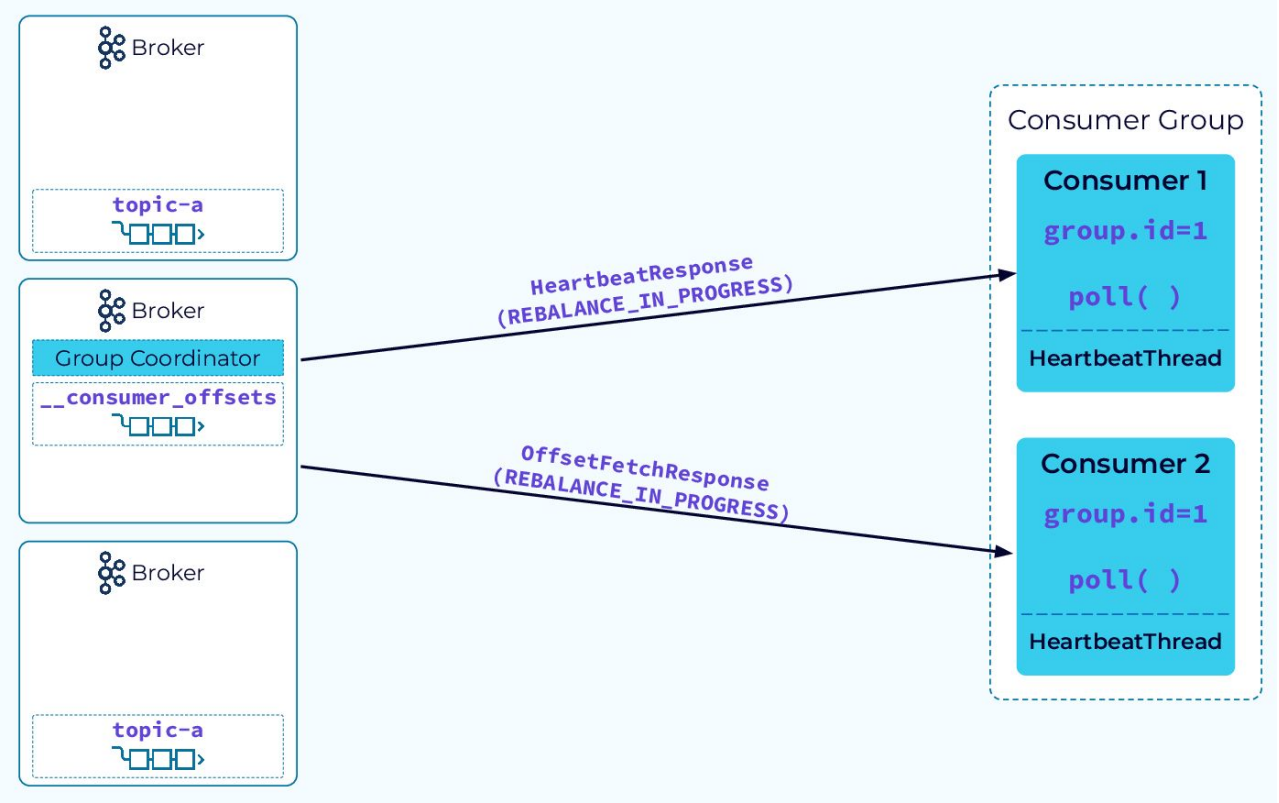
Group Coordinator Failover



Consumer Group Rebalance Triggers



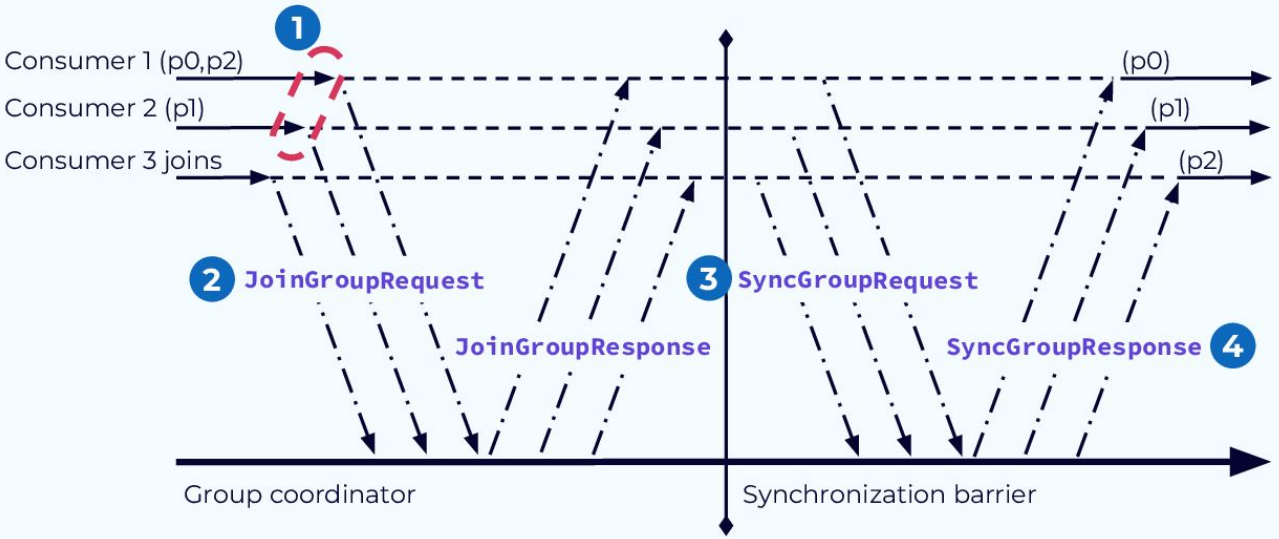
Consumer Group Rebalance Notification



Stop-the-World Rebalance

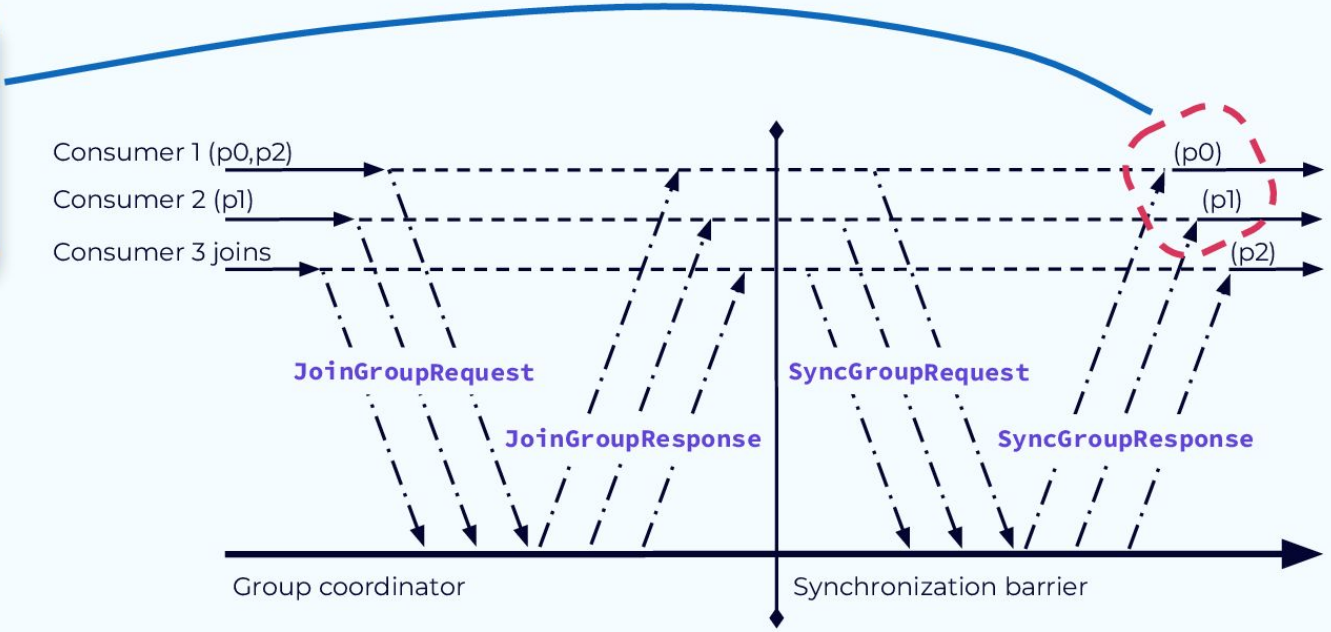
Consumers:

1) Revoke current partition assignment and clean up the partition states
2) Join the group
3) Sync with the group
4) Receive new partition assignments
   a) Build the partition state
   b) Resume consumption



Stop-the-World Problem #1 - Rebuilding State

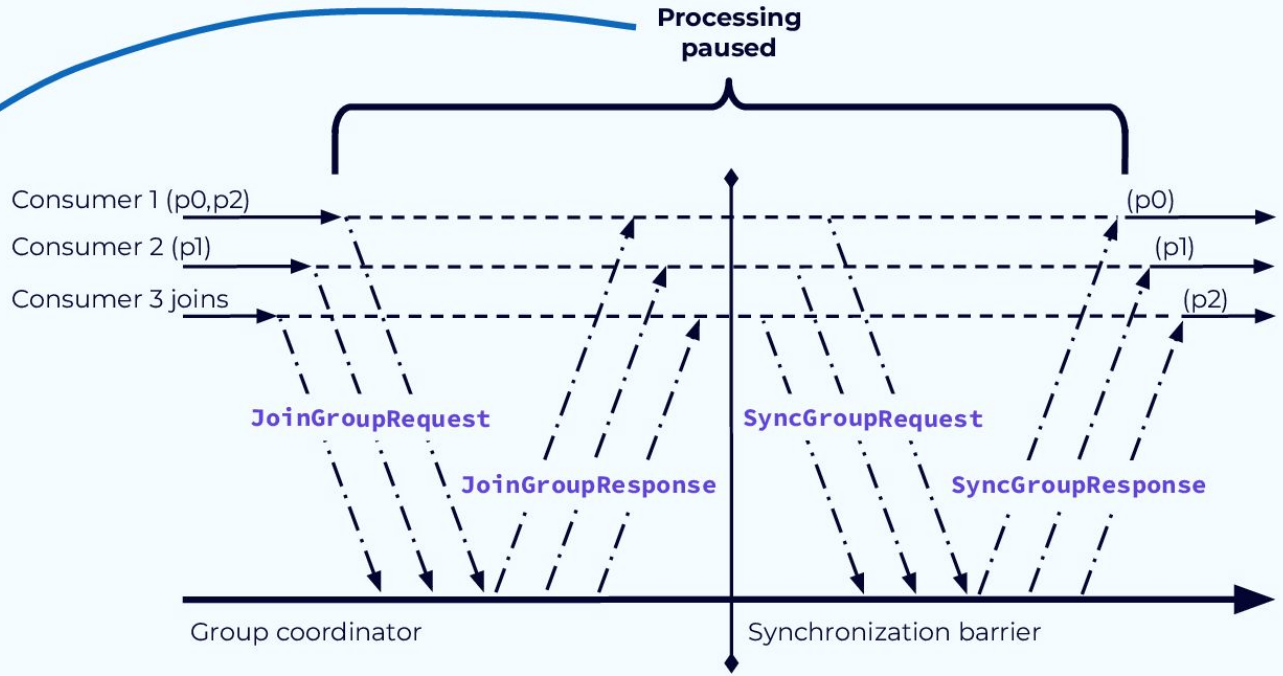
Since partitions p0 and p1 are assigned to the same consumer instance, rebuilding the state is unnecessary



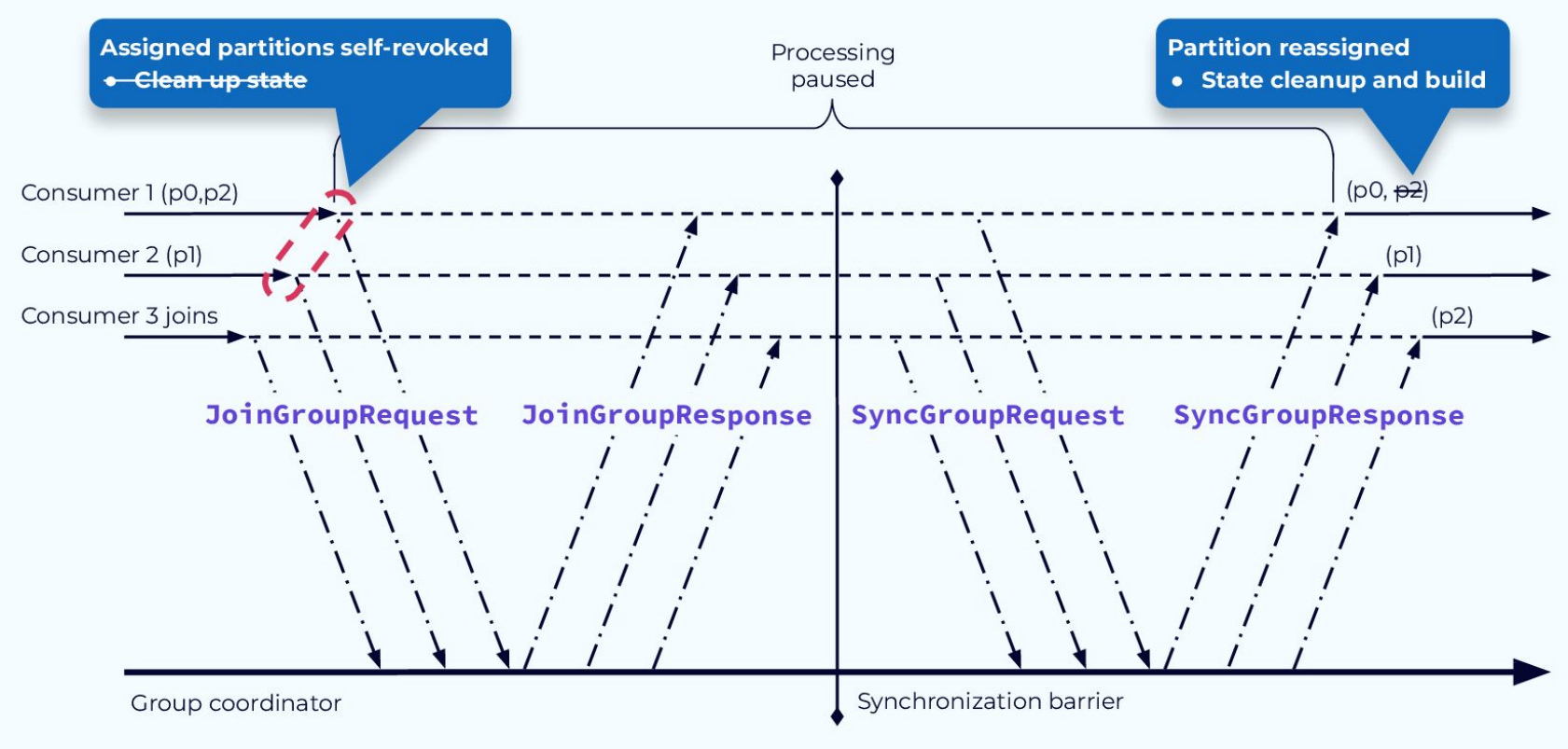
Stop-the-World Problem #2 - Paused Processing

Processing pauses for all subscribed partitions for the duration of the rebalance

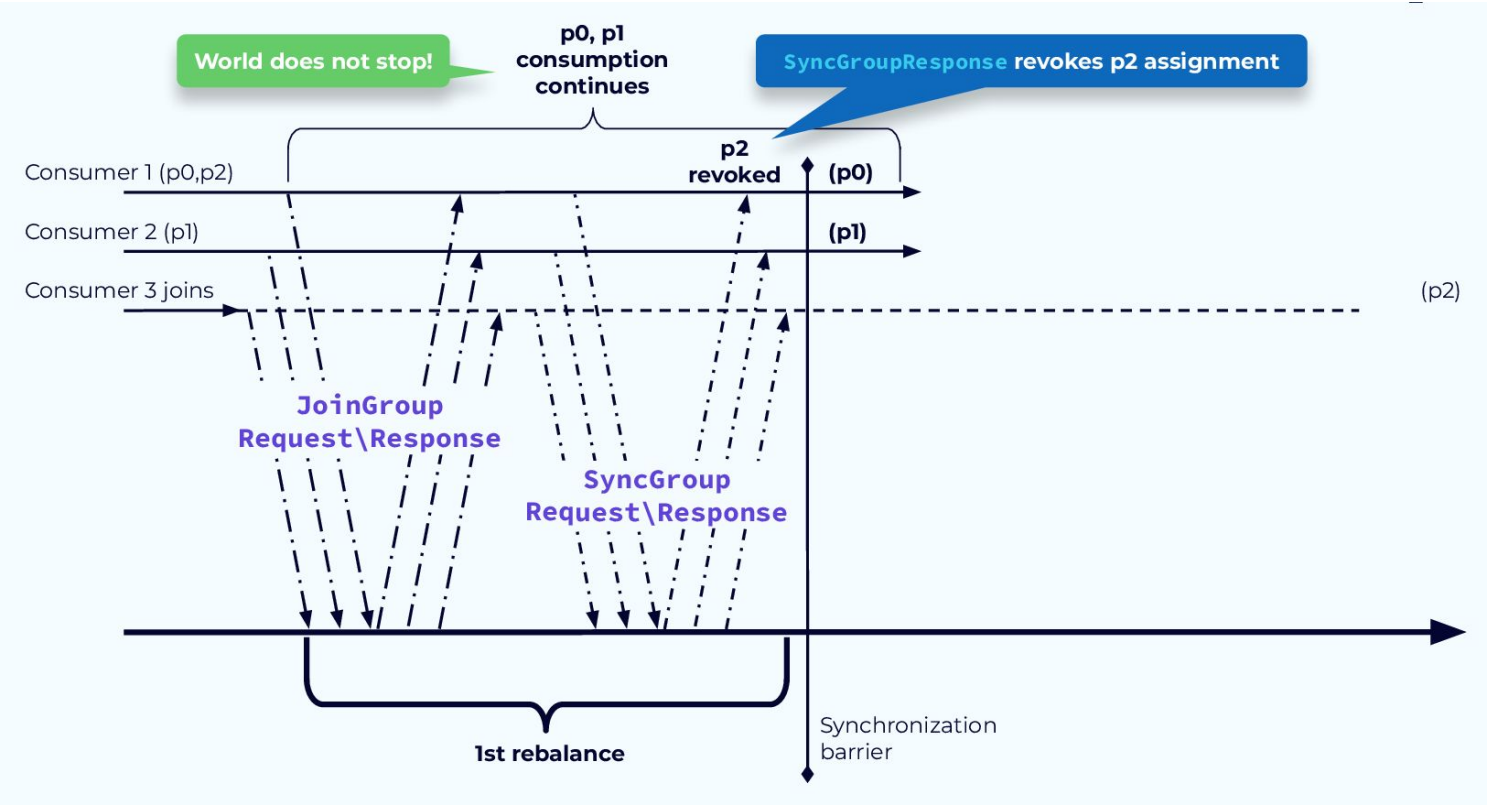
- **The pausing for p0 and p1 is unnecessary**



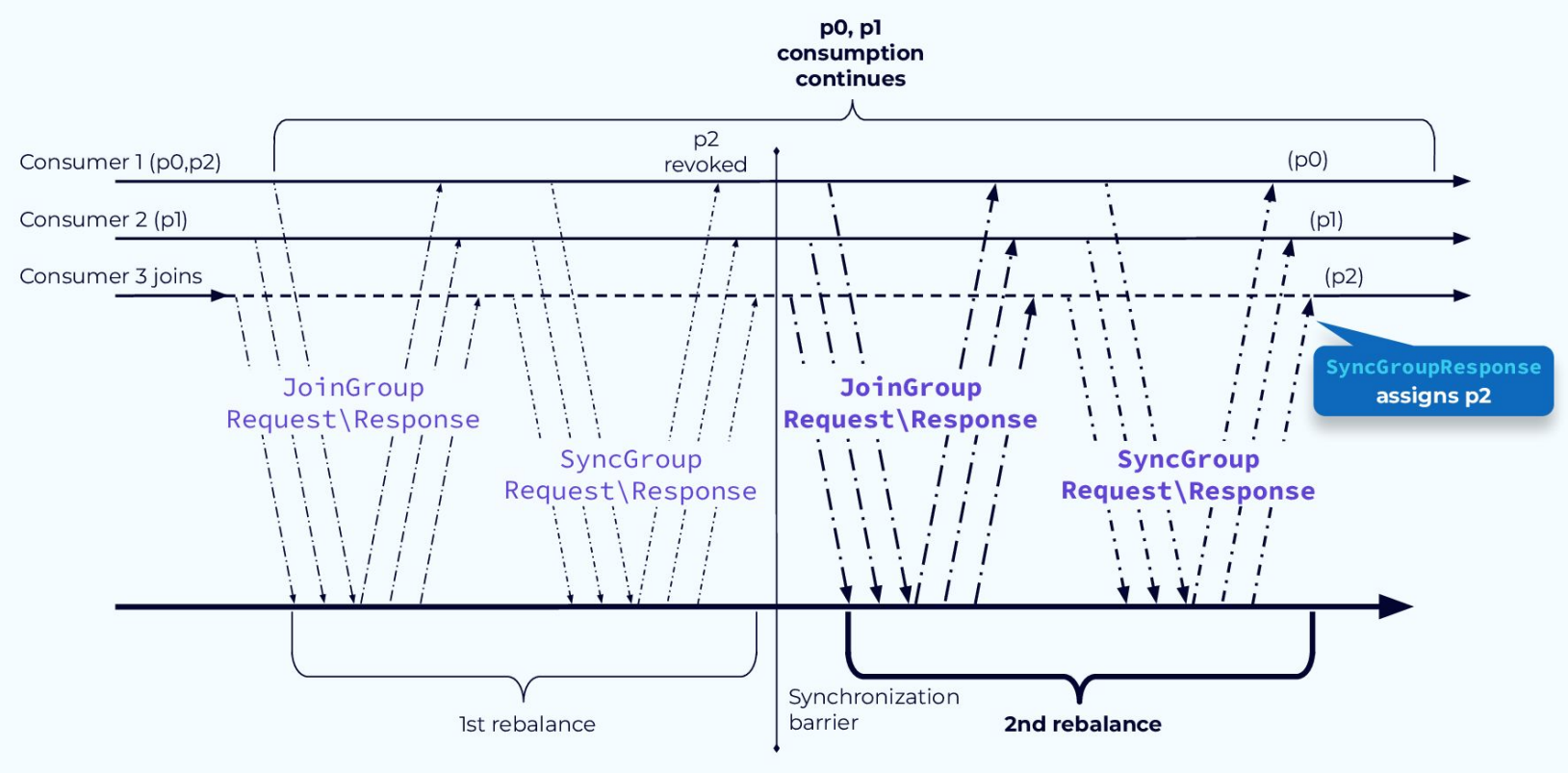
Avoid Needless State Rebuild with **StickyAssignor**



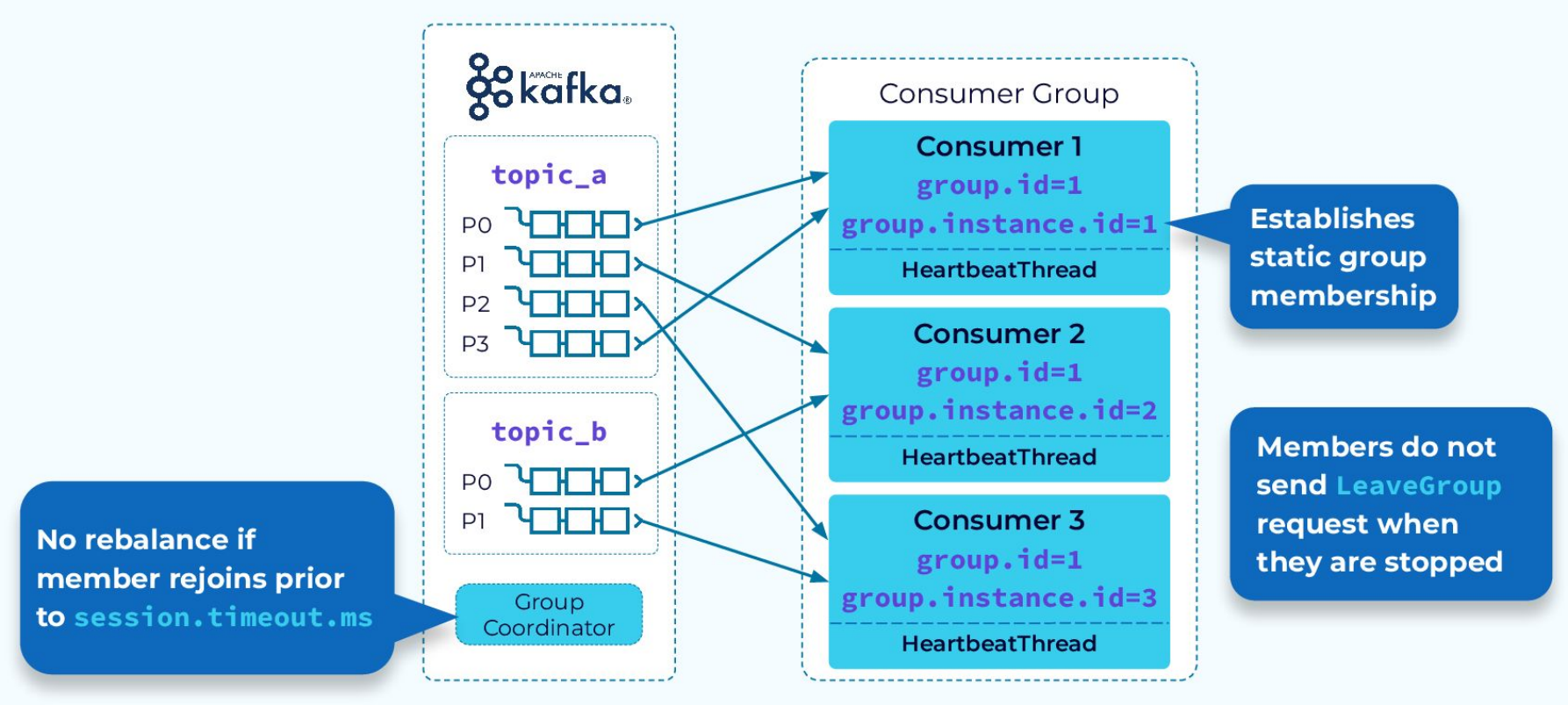
Avoid Pause with CooperativeStickyAssignor Step 1



Avoid Pause with CooperativeStickyAssignor Step 2



Avoid Rebalance with Static Group Membership



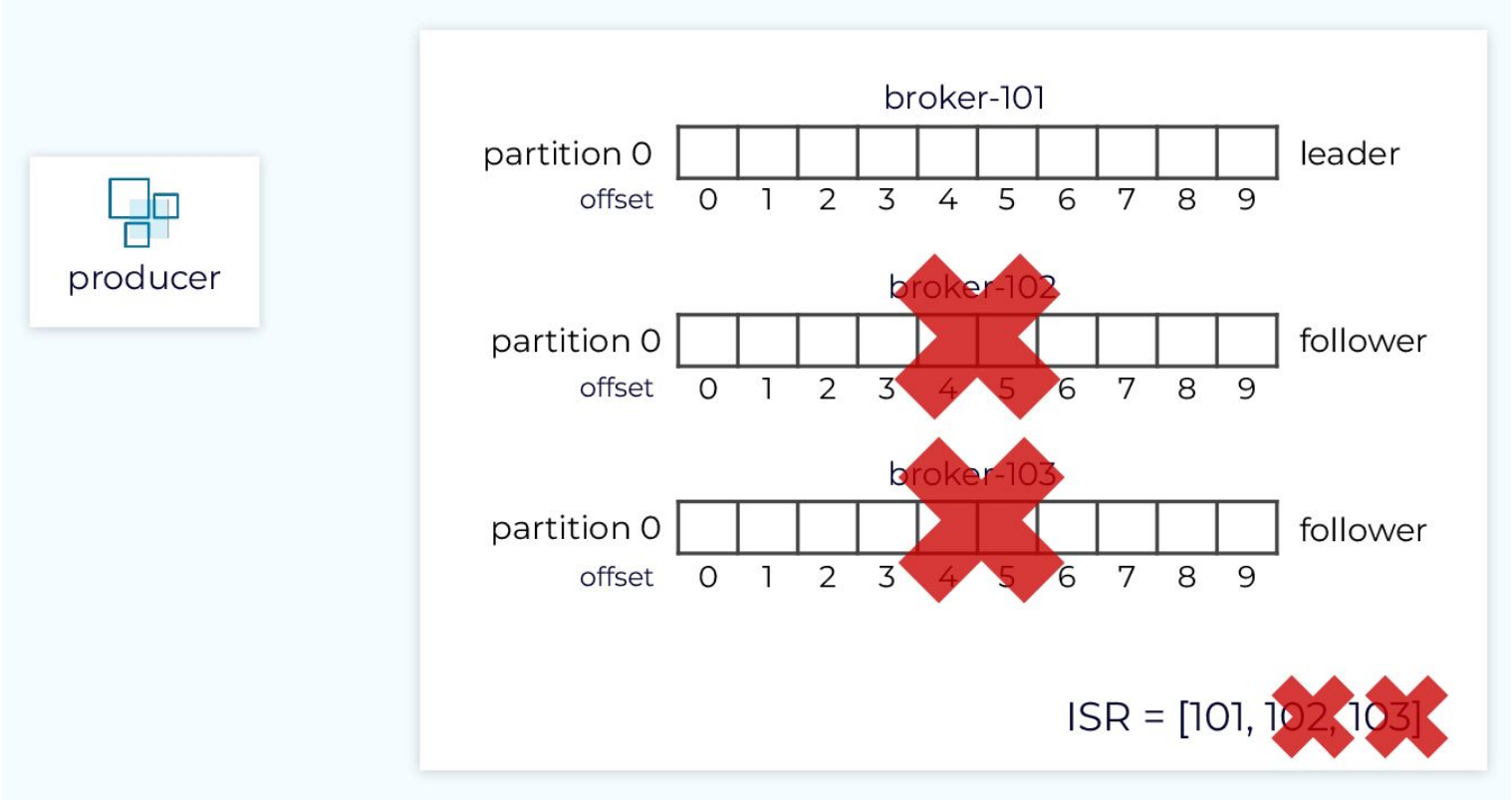
Demo-3:

Consumer Group Protocol

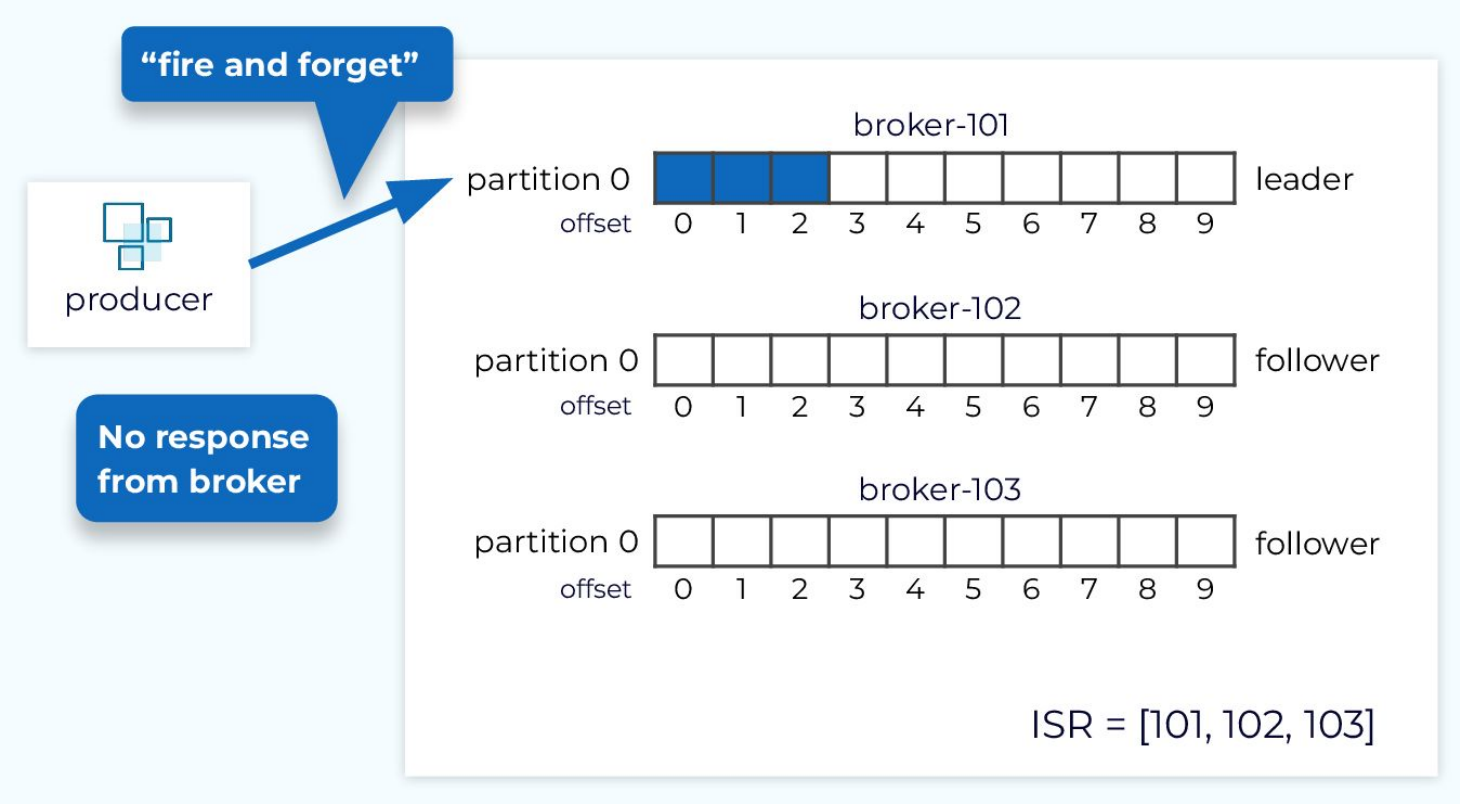
Any questions?

Durability,Availability & Ordering Guarantees

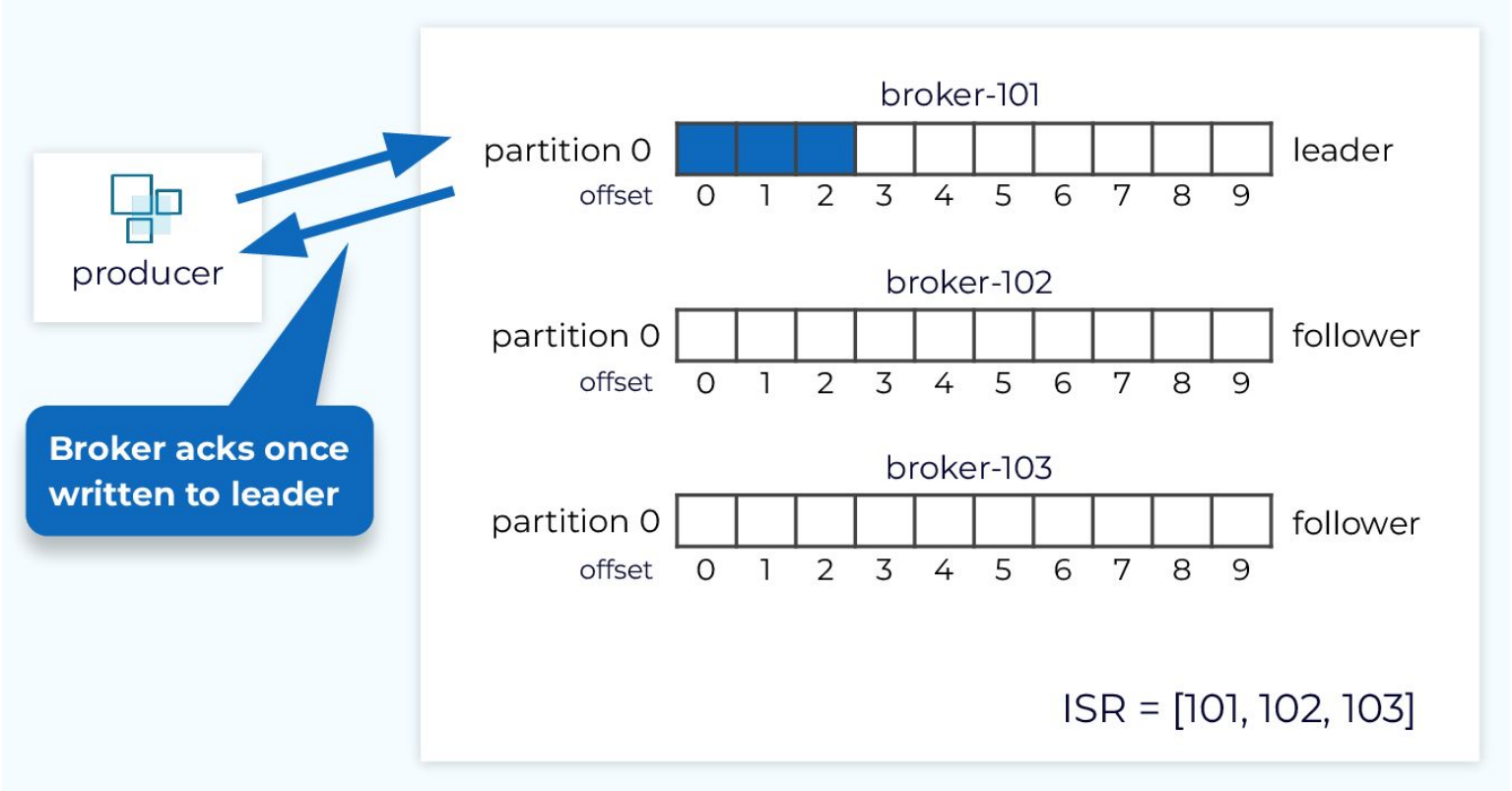
Data Durability and Availability Guarantees



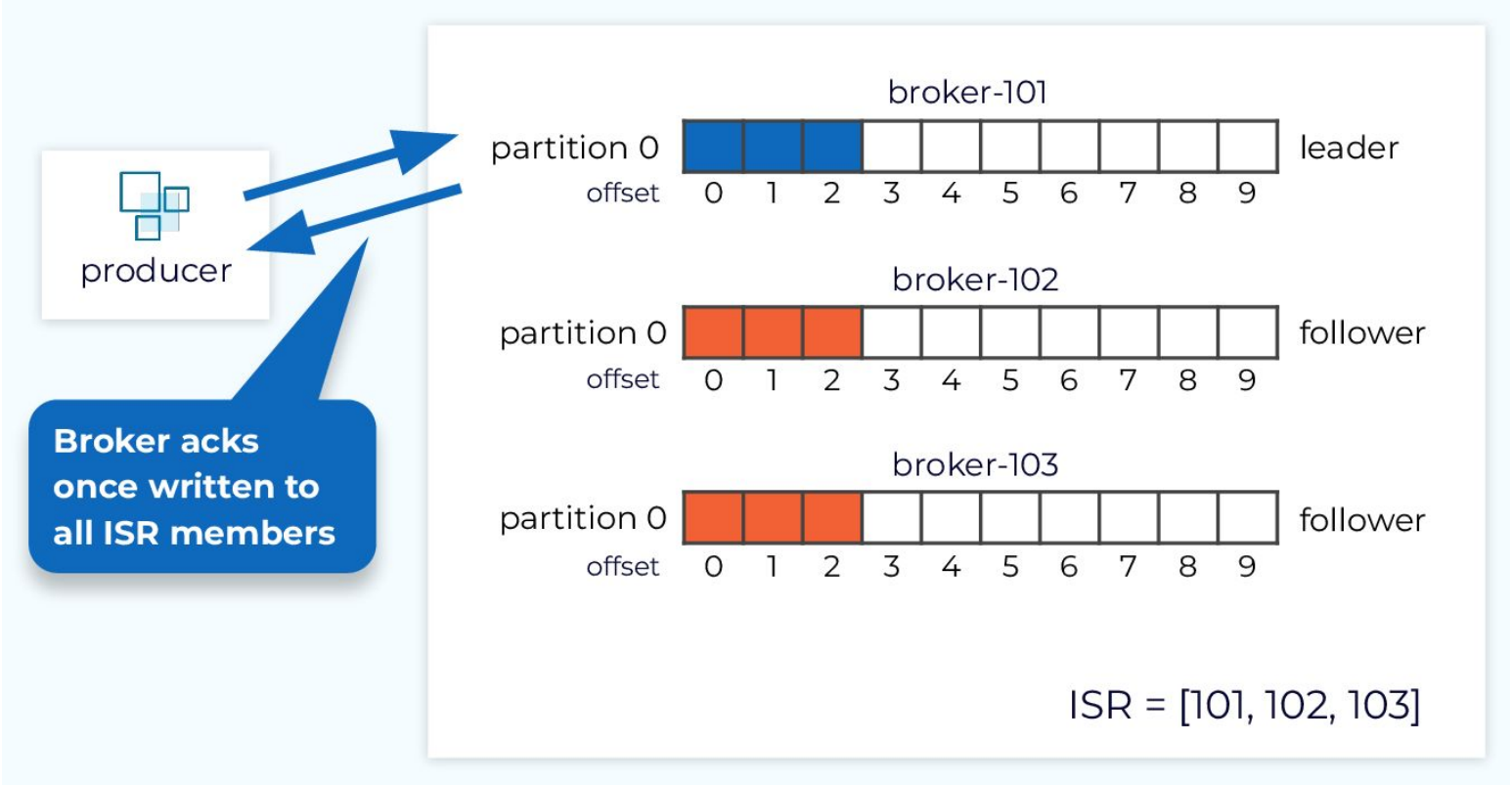
Producer acks = 0



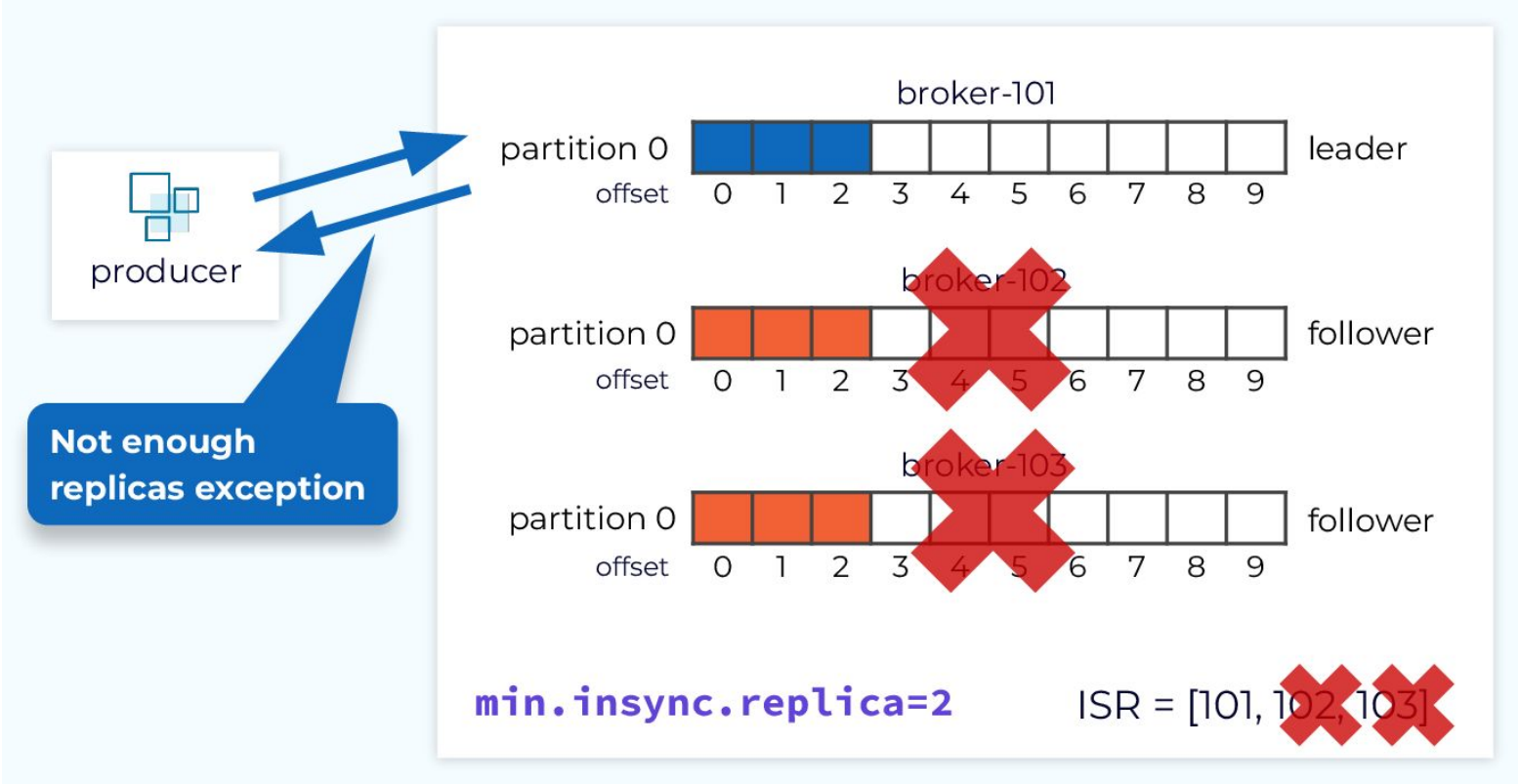
Producer acks = 1



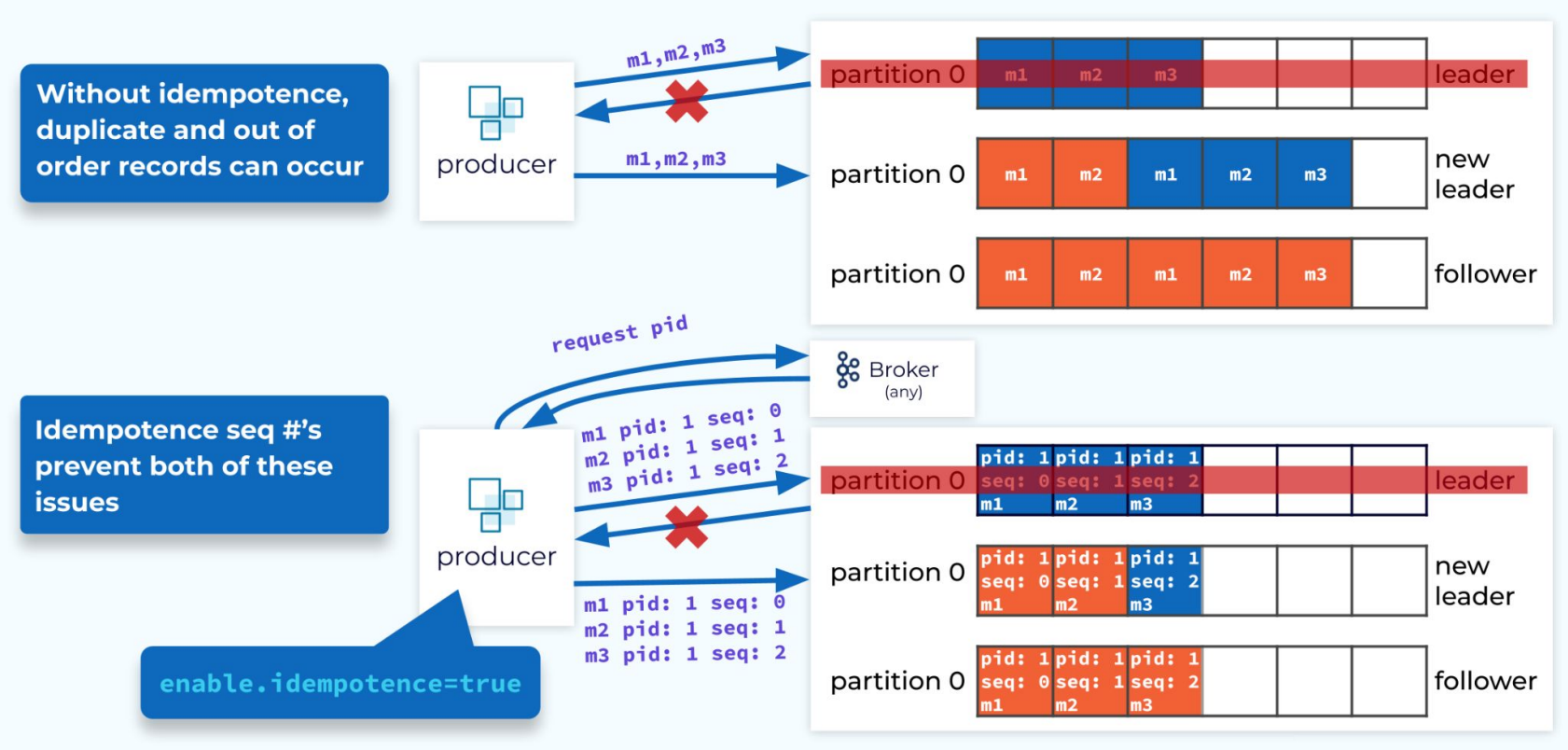
Producer acks = all



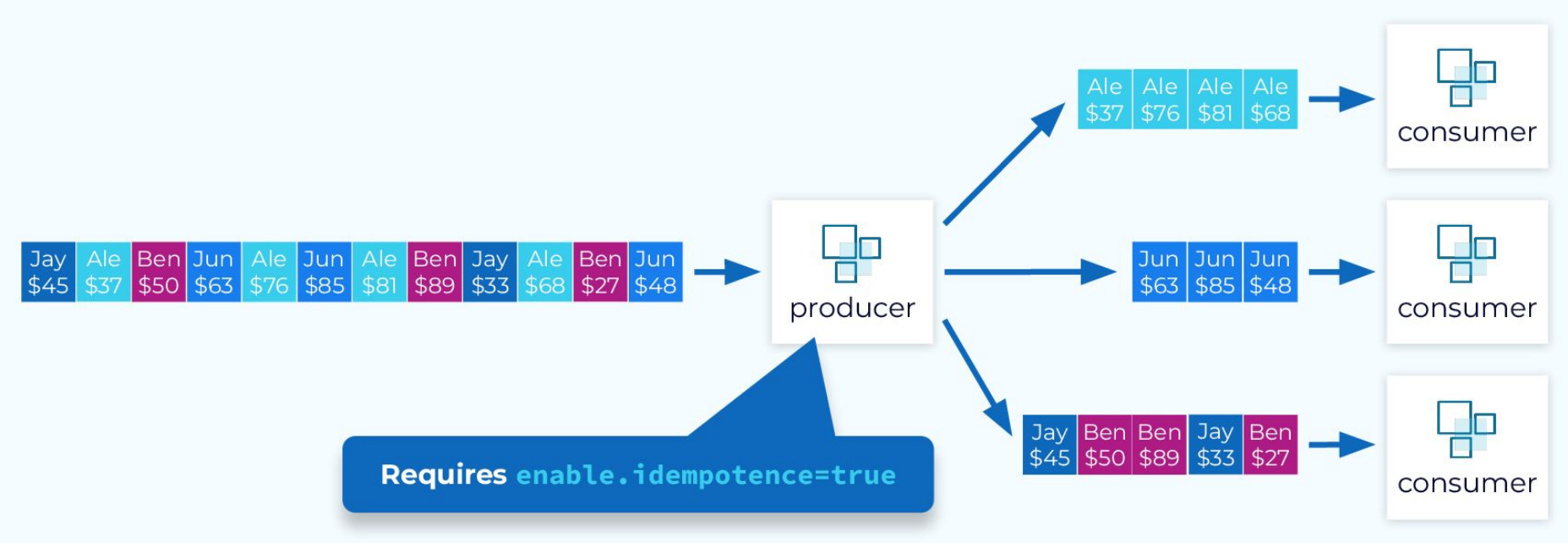
Topic **min.insync.replicas**



Producer Idempotence



End-to-End Guarantee



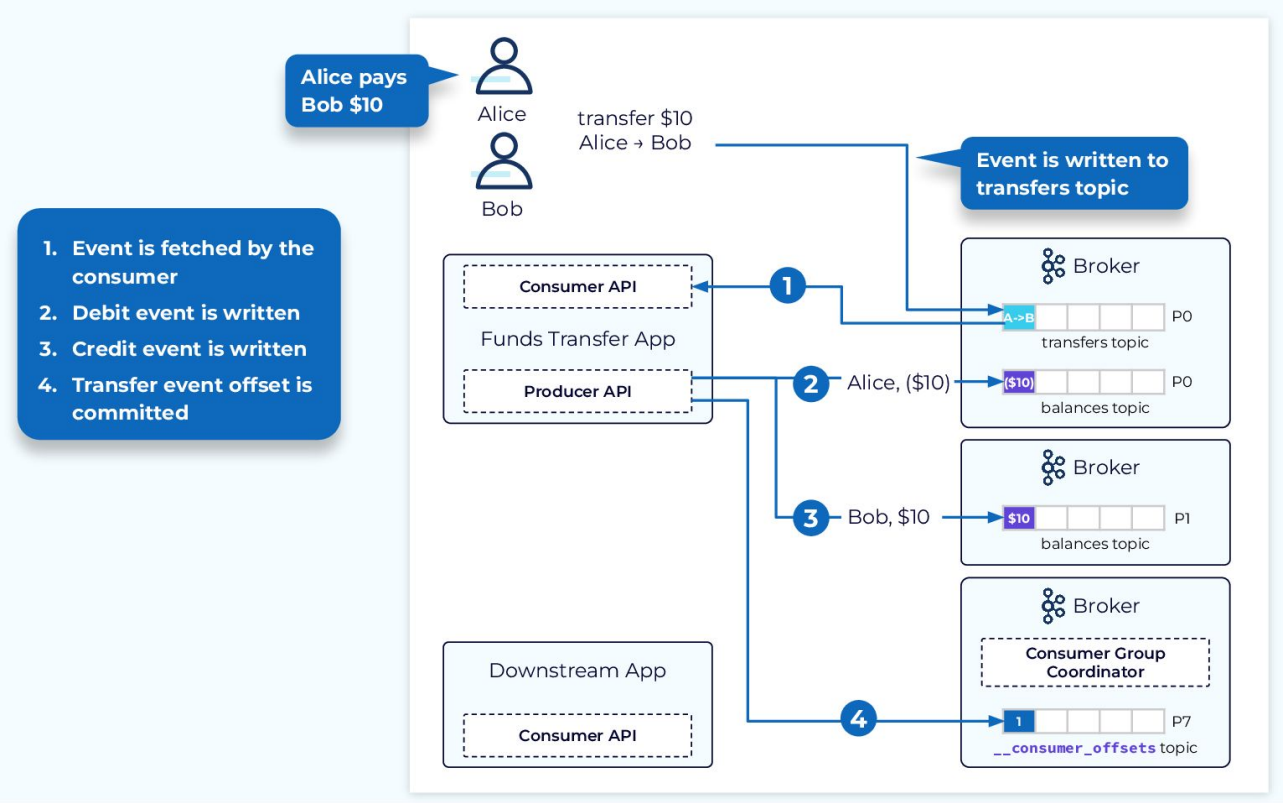
Demo-4:

Durability,Availability & Ordering Guarantees

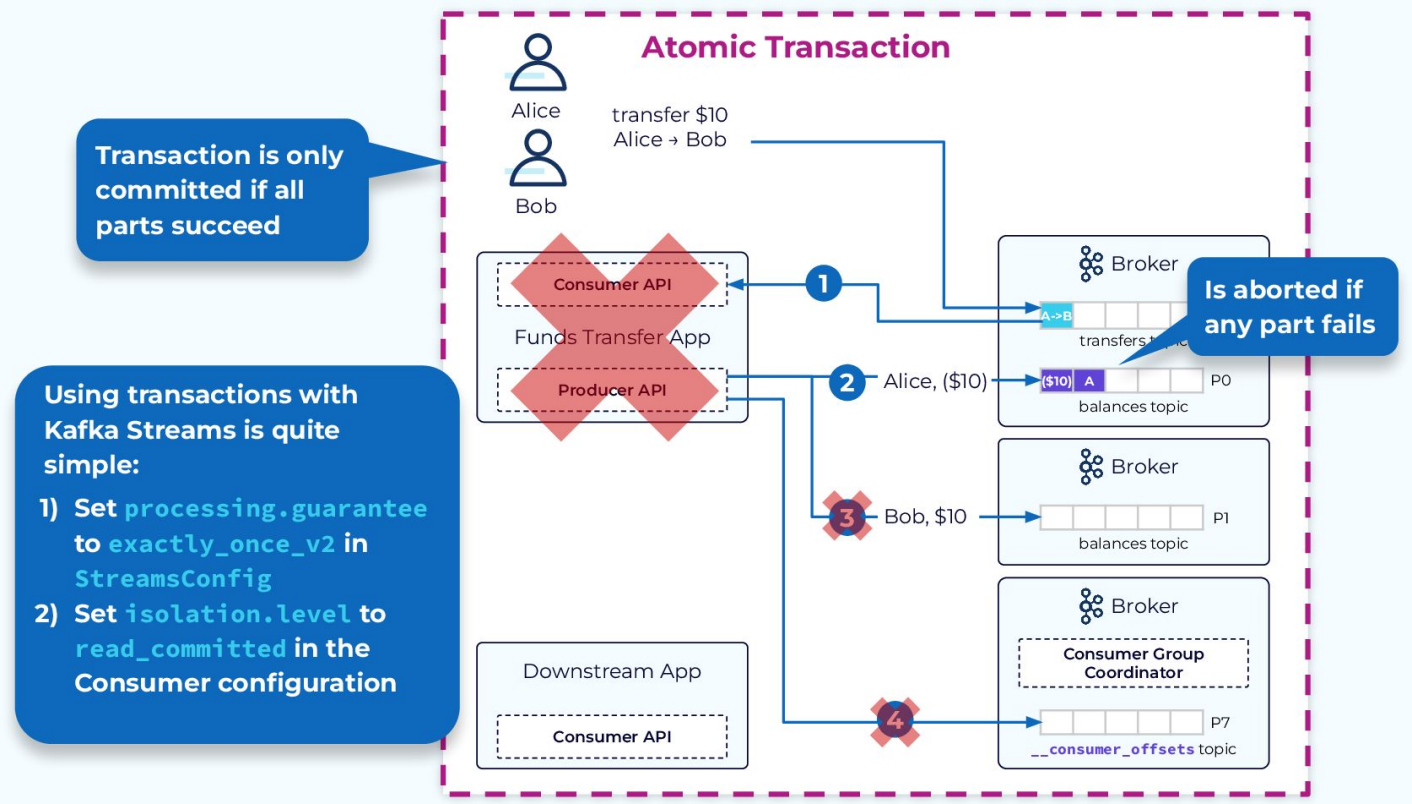
Any questions?

Transactions

Why Are Transactions Needed?

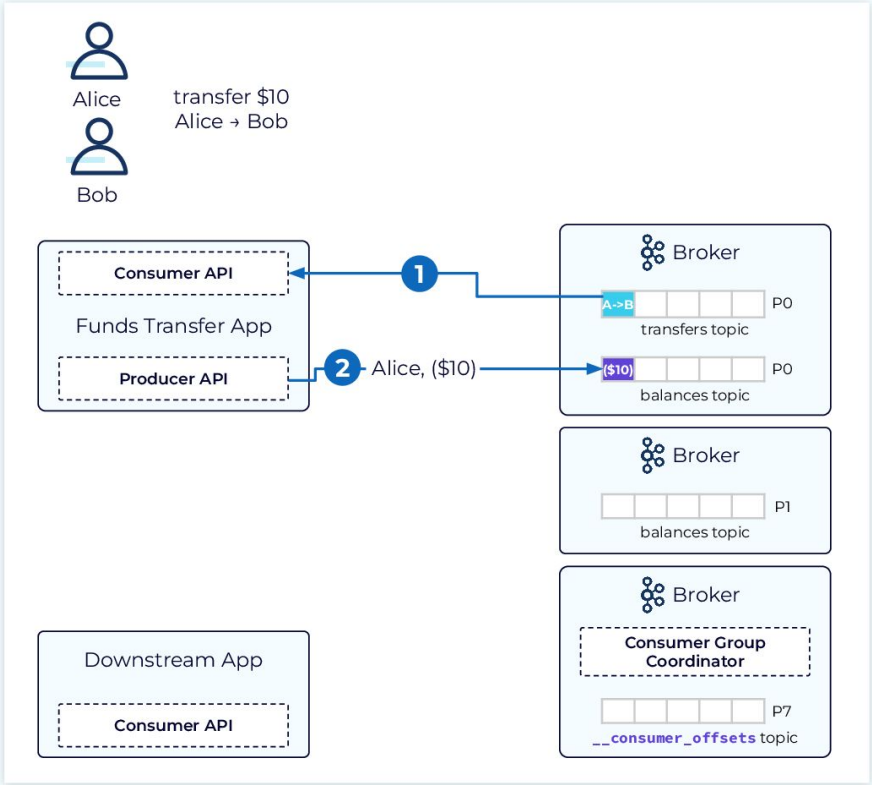


Kafka Transaction Deliver Exactly Once



System Failure Without Transactions

1. Event fetched by consumer
2. Alice's account debited

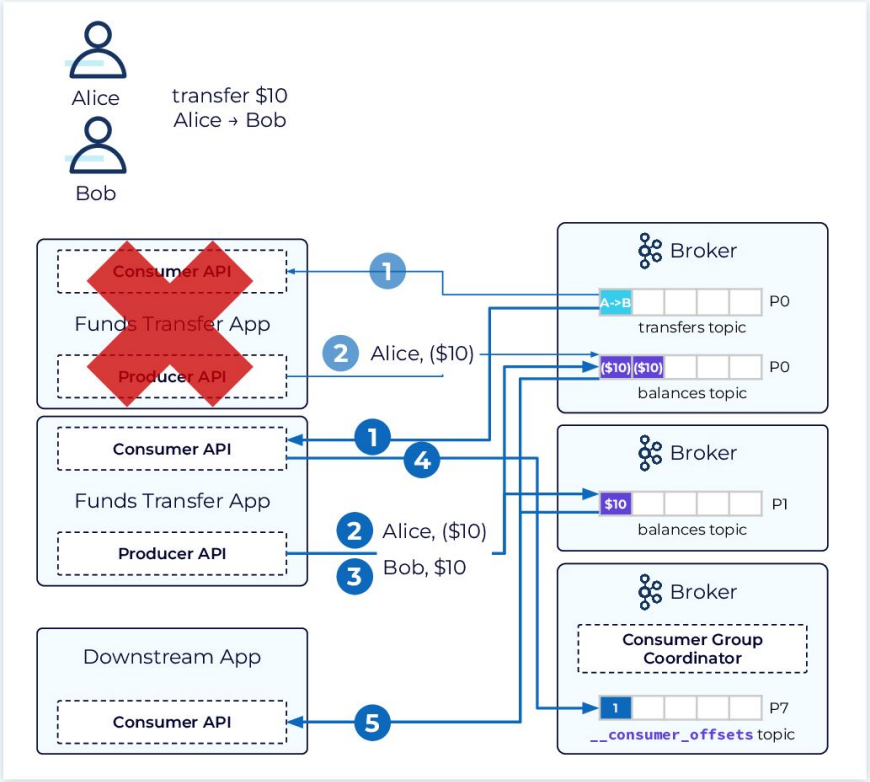


System Failure Without Transactions

1. Event fetched by consumer
2. Alice's account debited

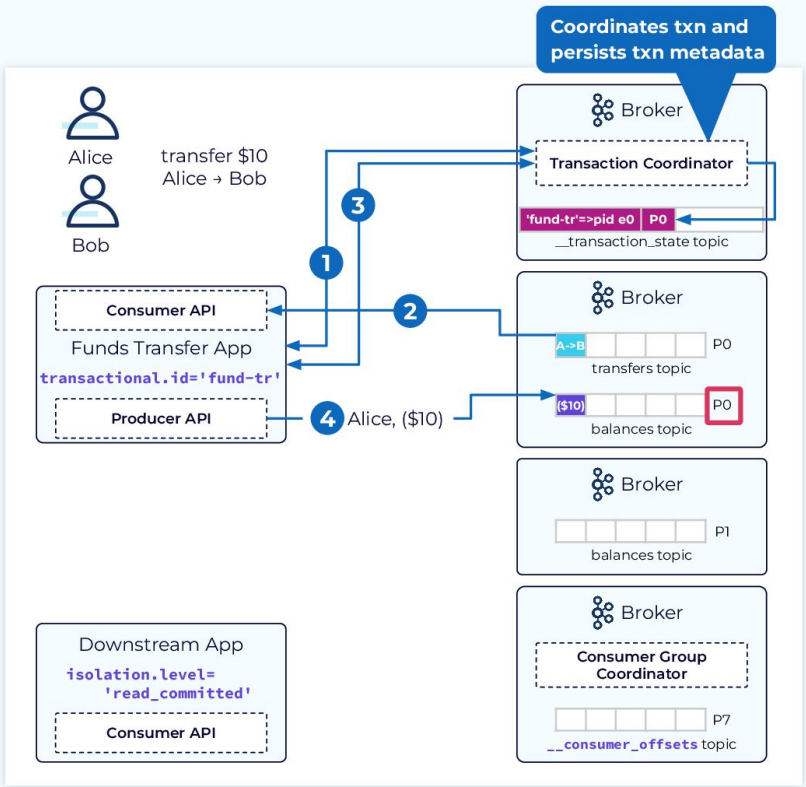
Application instance fails without committing offset and new application instance starts

1. **Event fetched by consumer**
2. **Alice's account is debited a second time**
3. **Bob's account is credited**
4. **Consumer offset committed**
5. **Two debit events processed by downstream consumer**



System Failure With Transactions

1. Requests txn ID, is returned PID and txn epoch
2. Event fetched by consumer
3. Notifies coordinator of partition being written to
4. Alice's account debited

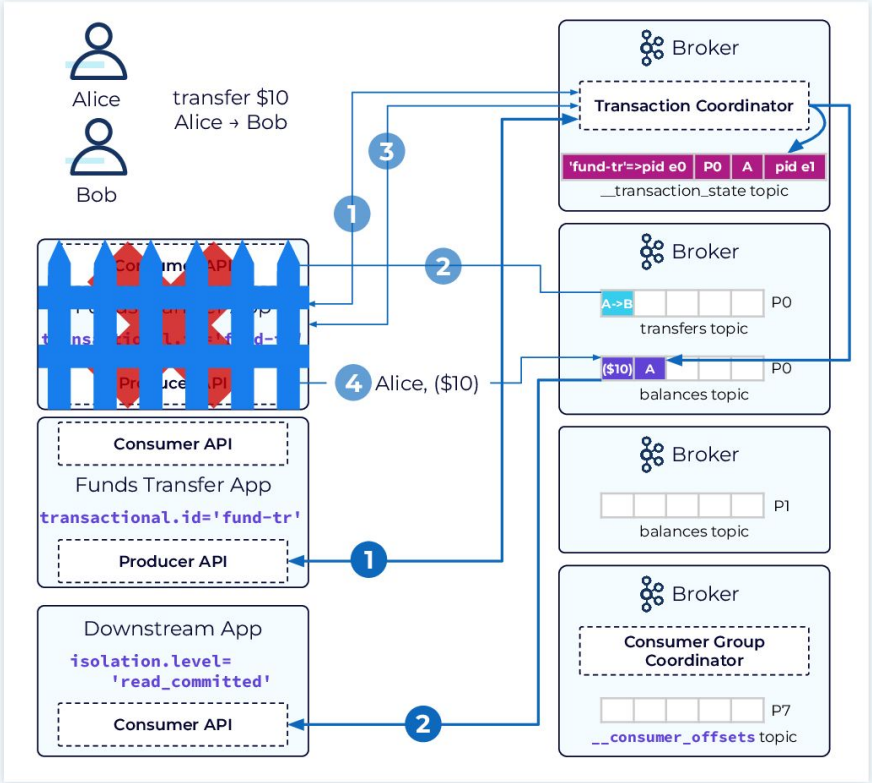


System Failure with Transactions

1. Requests txn ID, is returned PID and txn epoch
2. Event fetched by consumer
3. Notifies coordinator of partition being written to
4. Alice's account debited

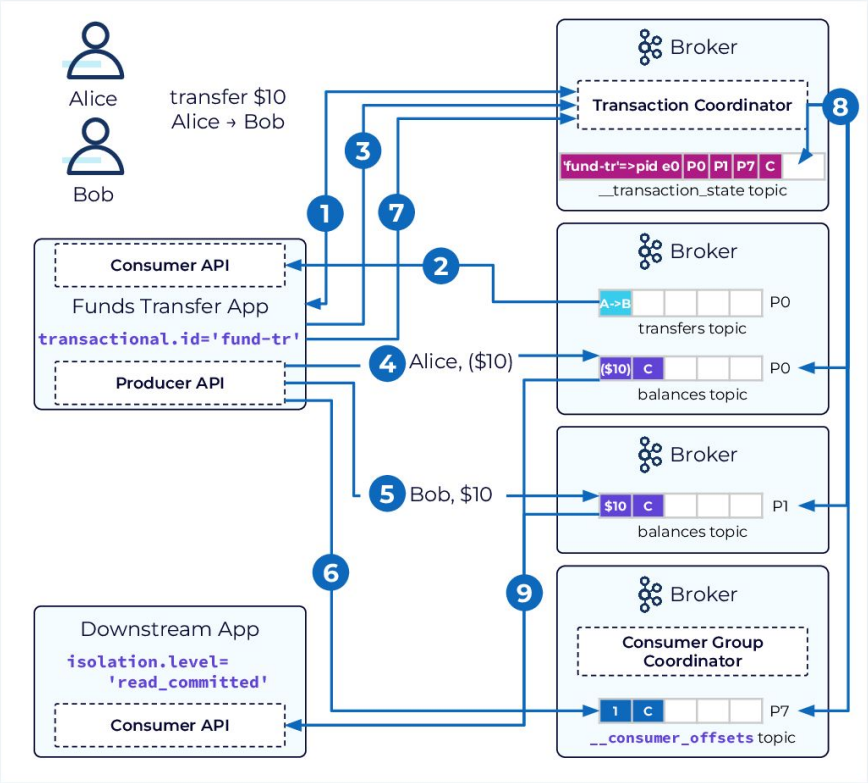
Application instance fails without committing offset and new application instance starts

1. **New instance requests txn ID**
   a. **Coordinator fences previous instance by aborting pending txn and bumping up epoch**
2. **Downstream consumer with `read_committed` discards aborted events**



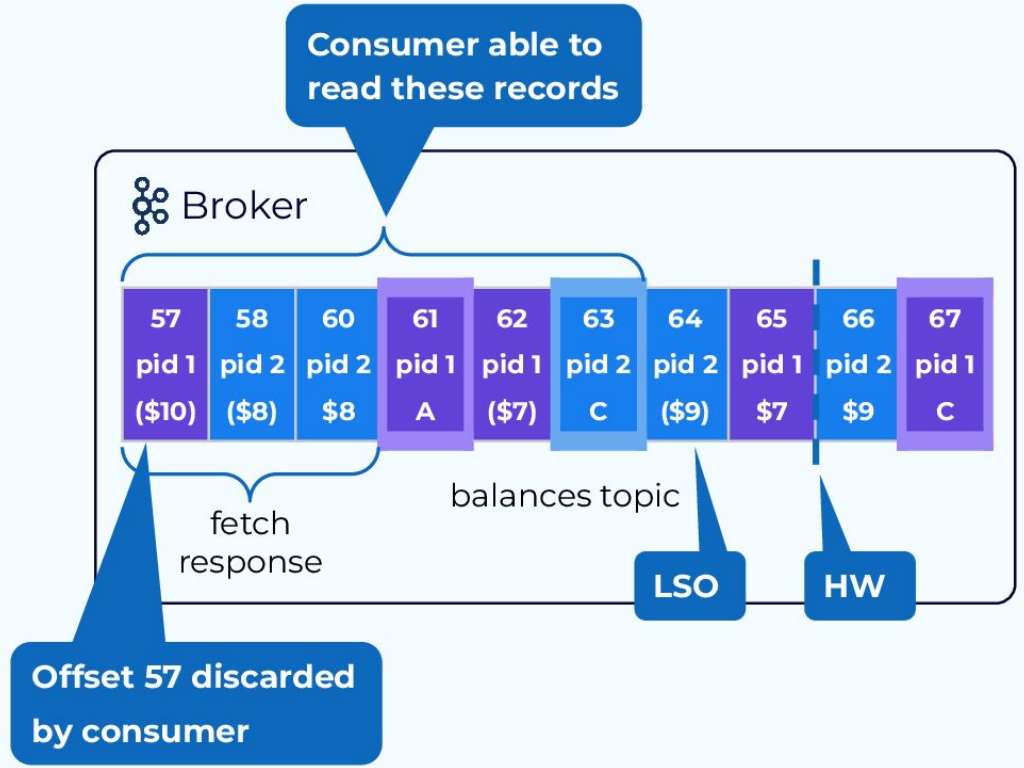
System with Successful Committed Transaction

1. Requests txn ID and assigned PID and epoch
2. Event fetched by consumer
3. Notifies coordinator of partition being written to
4. Alice's account debited
5. Bob's account credited
6. Consumer offset committed
7. Notify coordinator that transaction is complete
8. Coordinator writes commit markers to p0, p1, p7
9. Downstream consumer with `read_committed` processes committed events

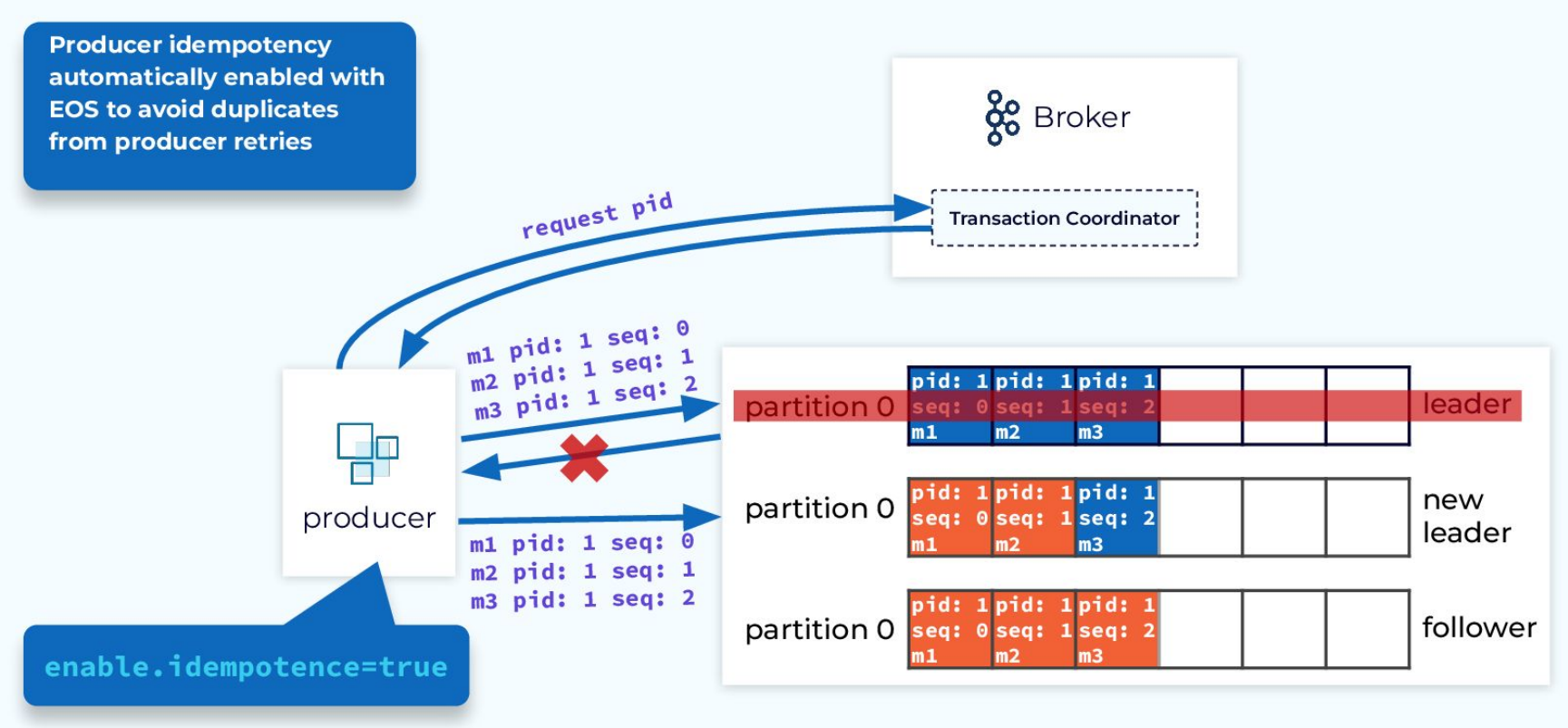


Consuming Transactions with **read_commited**

- Leader maintains last stable offset (LSO), the smallest offset of any open transaction
- Fetch response includes
  - only records up to LSO
  - metadata for skipping aborted records



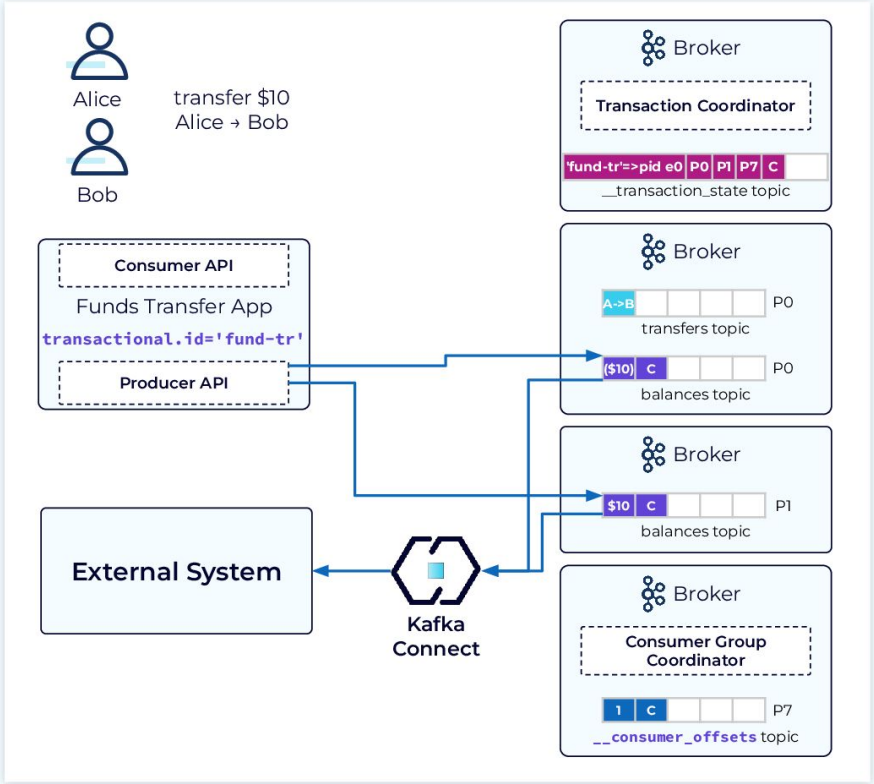
Transactions: Producer Idempotency



Interacting with External Systems

Atomic writes to Kafka and external systems are not supported

- **Instead, write the transactional output to a Kafka topic first**
- **Rely on idempotency to propagate the data from the output topic to the external system**



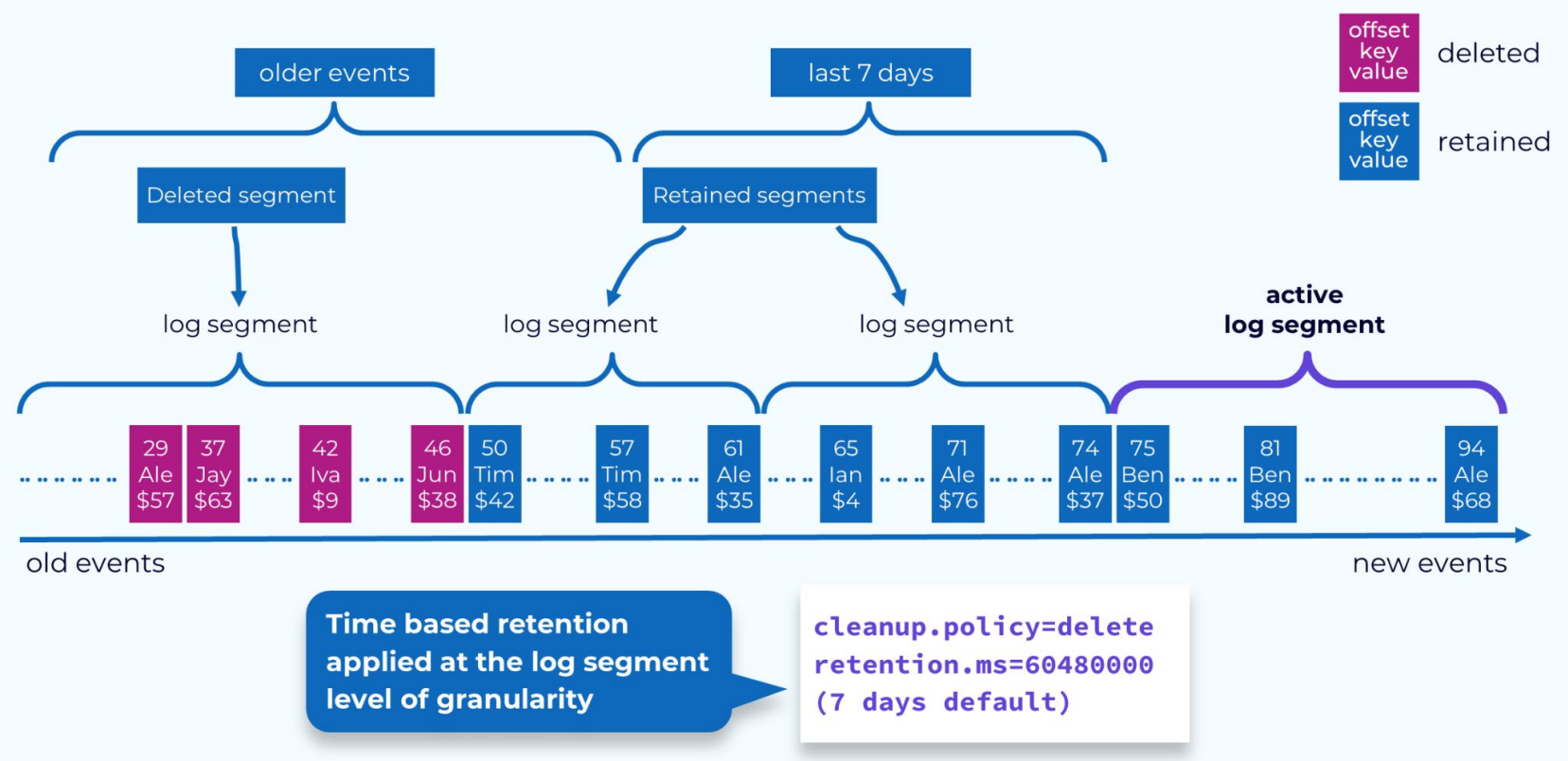
Demo-5:

Transactions

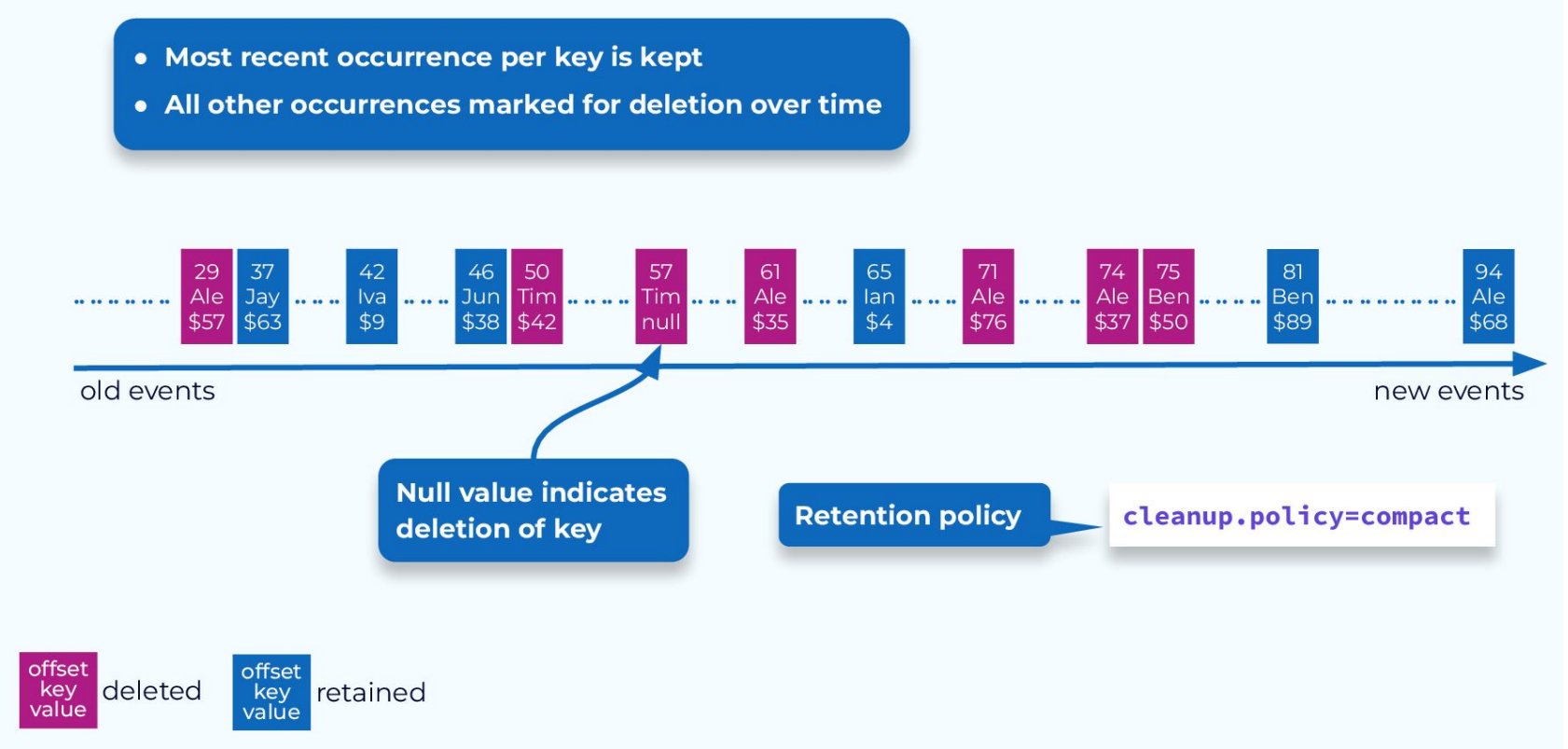
Any questions?

Topic Compaction

Time-Based Retention



Topic Compaction: Key-Based Retention



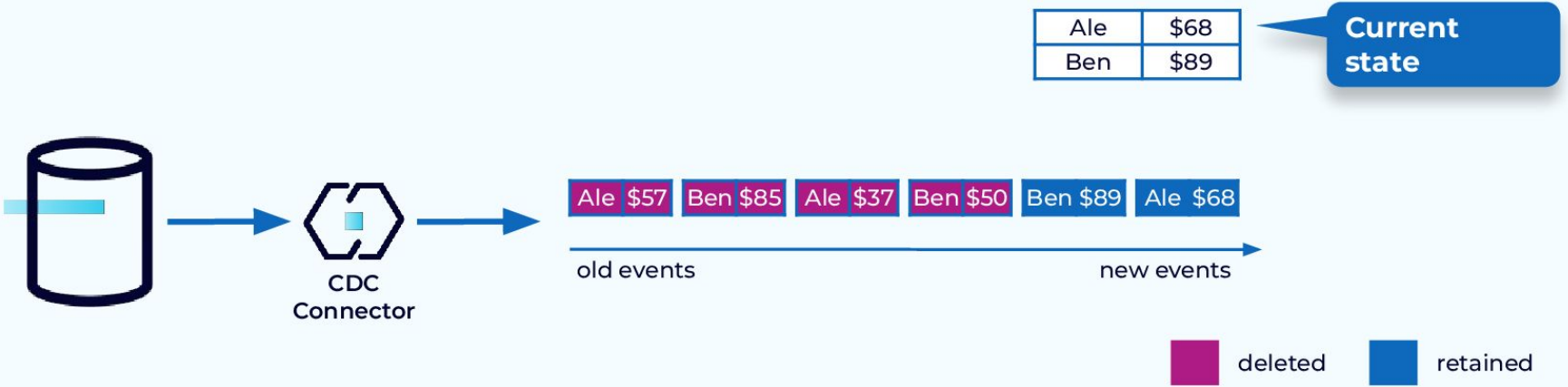
Usage and Benefits of Topic Compaction

Usage:

- Streaming updatable data sets (e.g. profiles, catalogue)
- ksqlDB state and tables

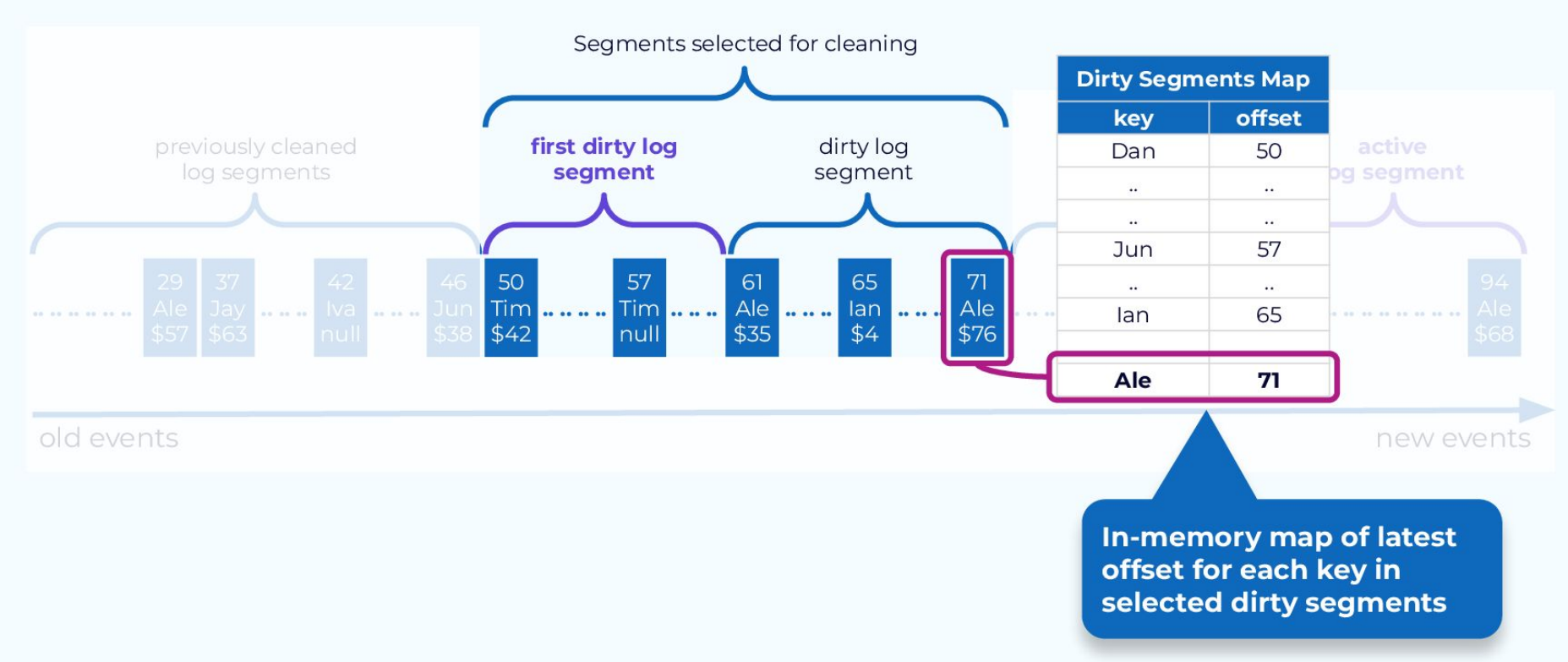
Benefits:

- Keeping latest info with bounded storage
- Allows incremental consumption and bootstrap to be done the same way

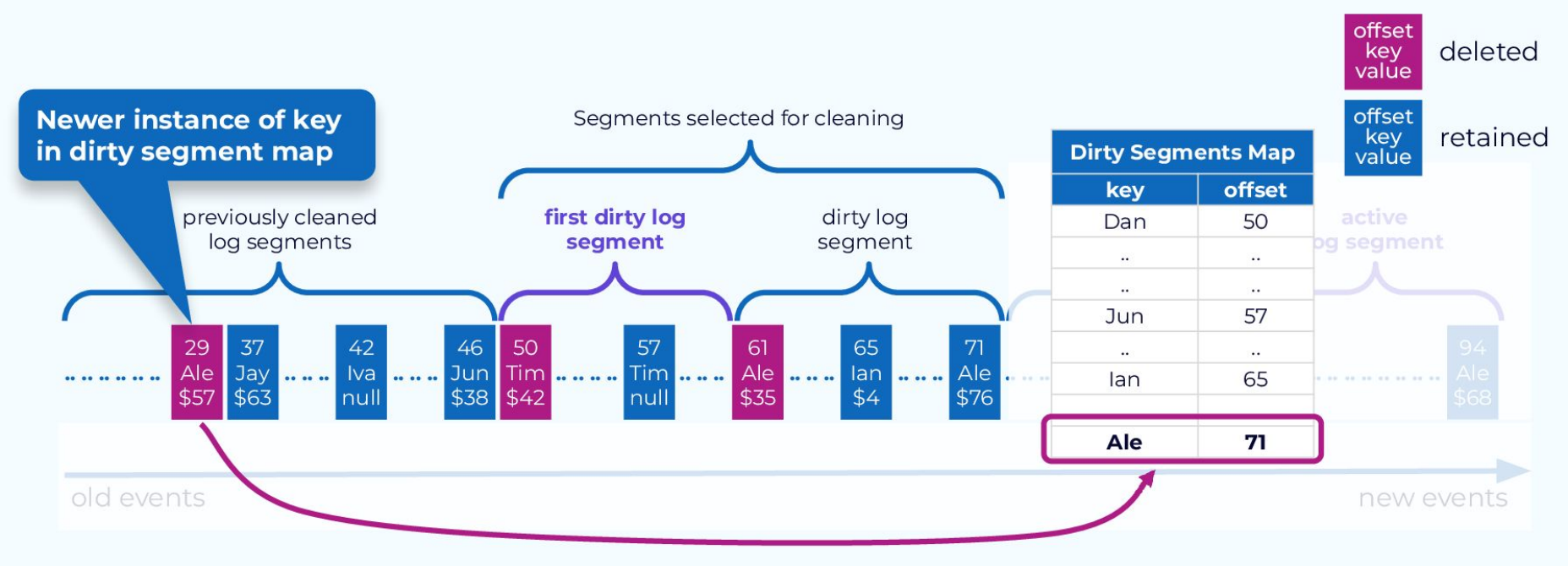


Compaction Process - Segments to Clean

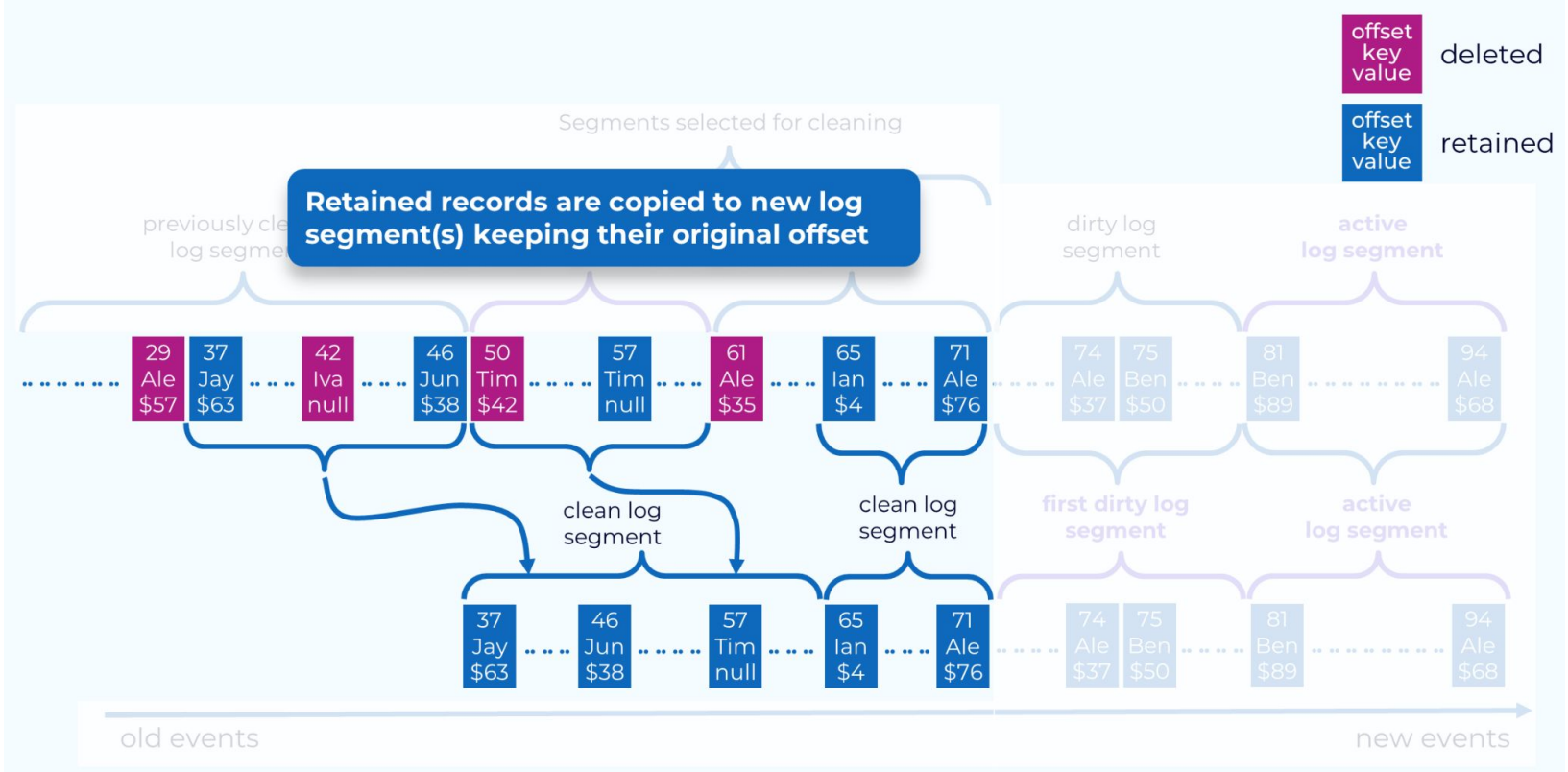
Compaction Process - Build Dirty Segment Map



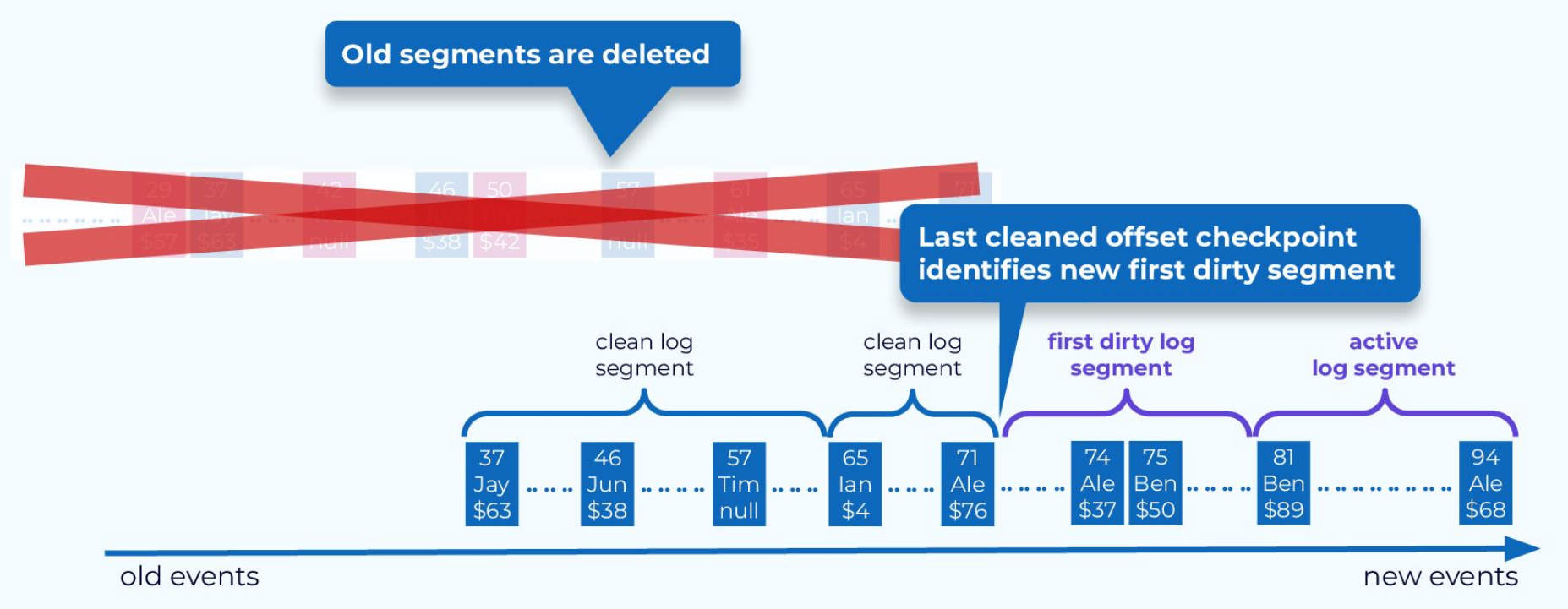
Compaction Process - Deleting Events



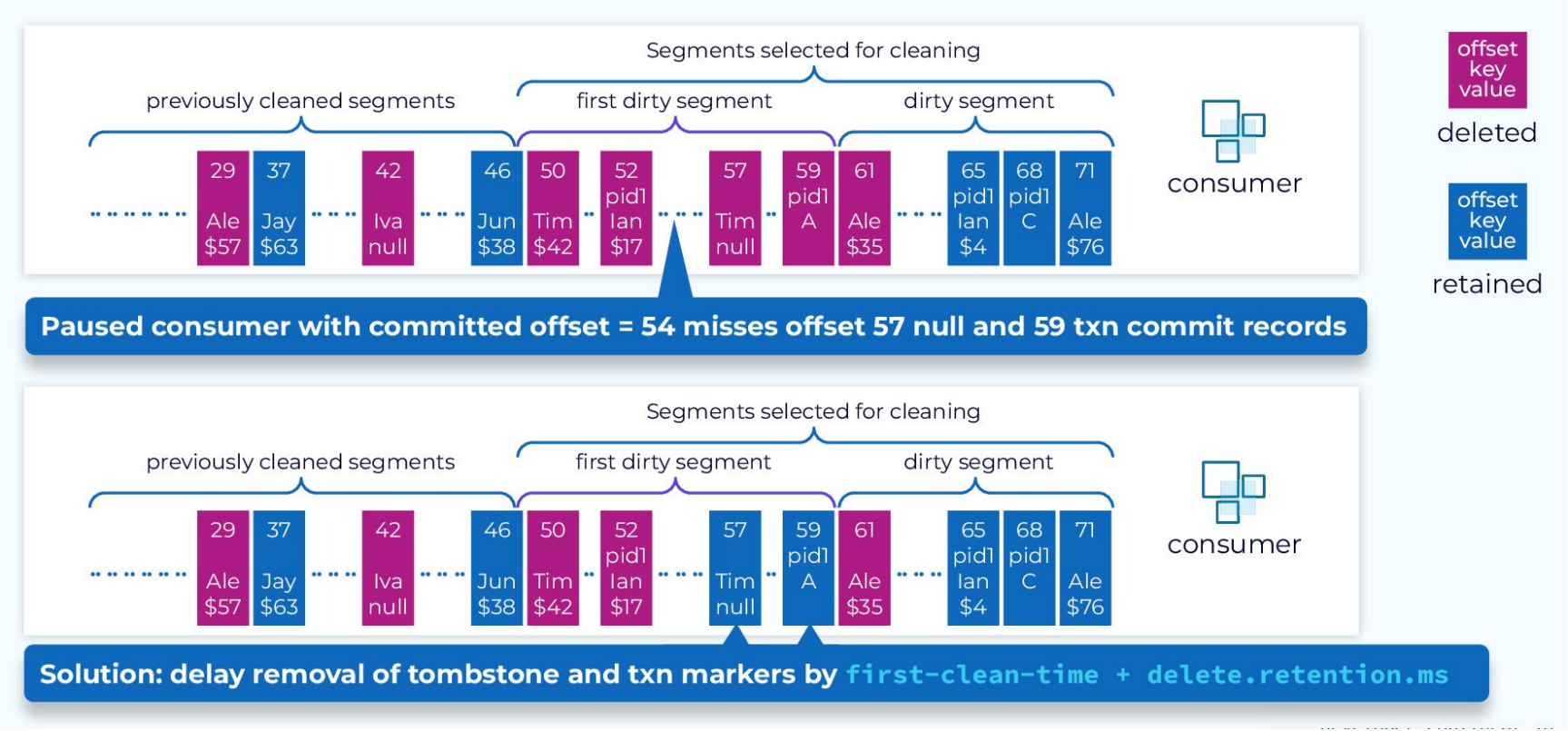
Compaction Process - Retaining Events



Compaction Process - Replace Old Segments



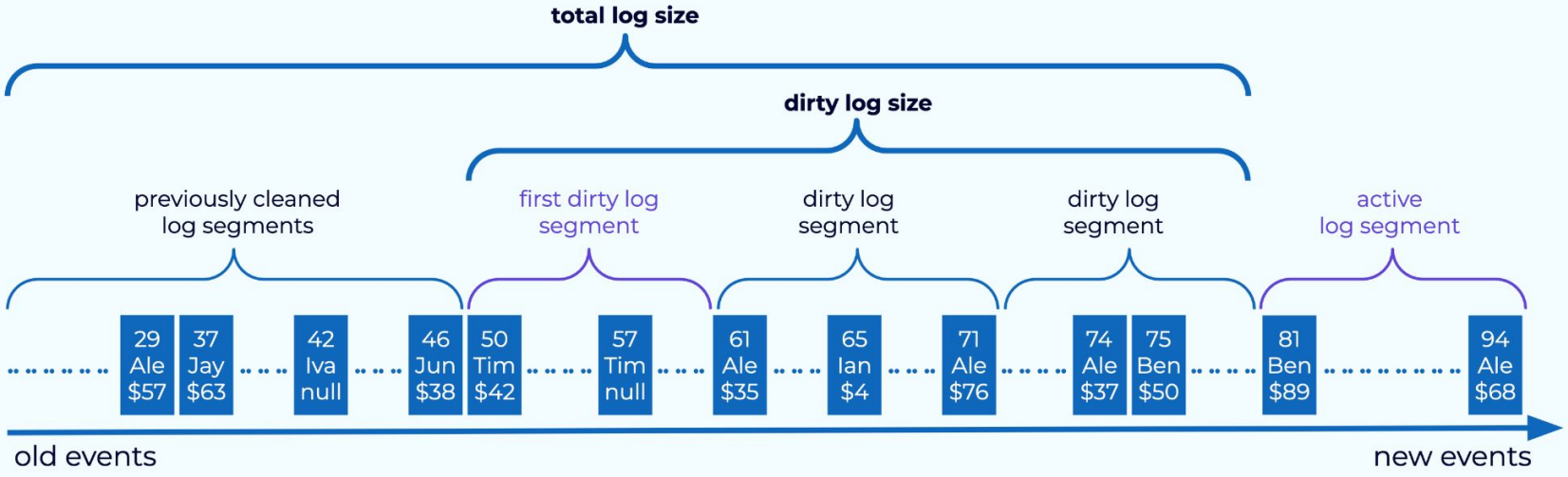
Cleaning Tombstone and Transaction Markers



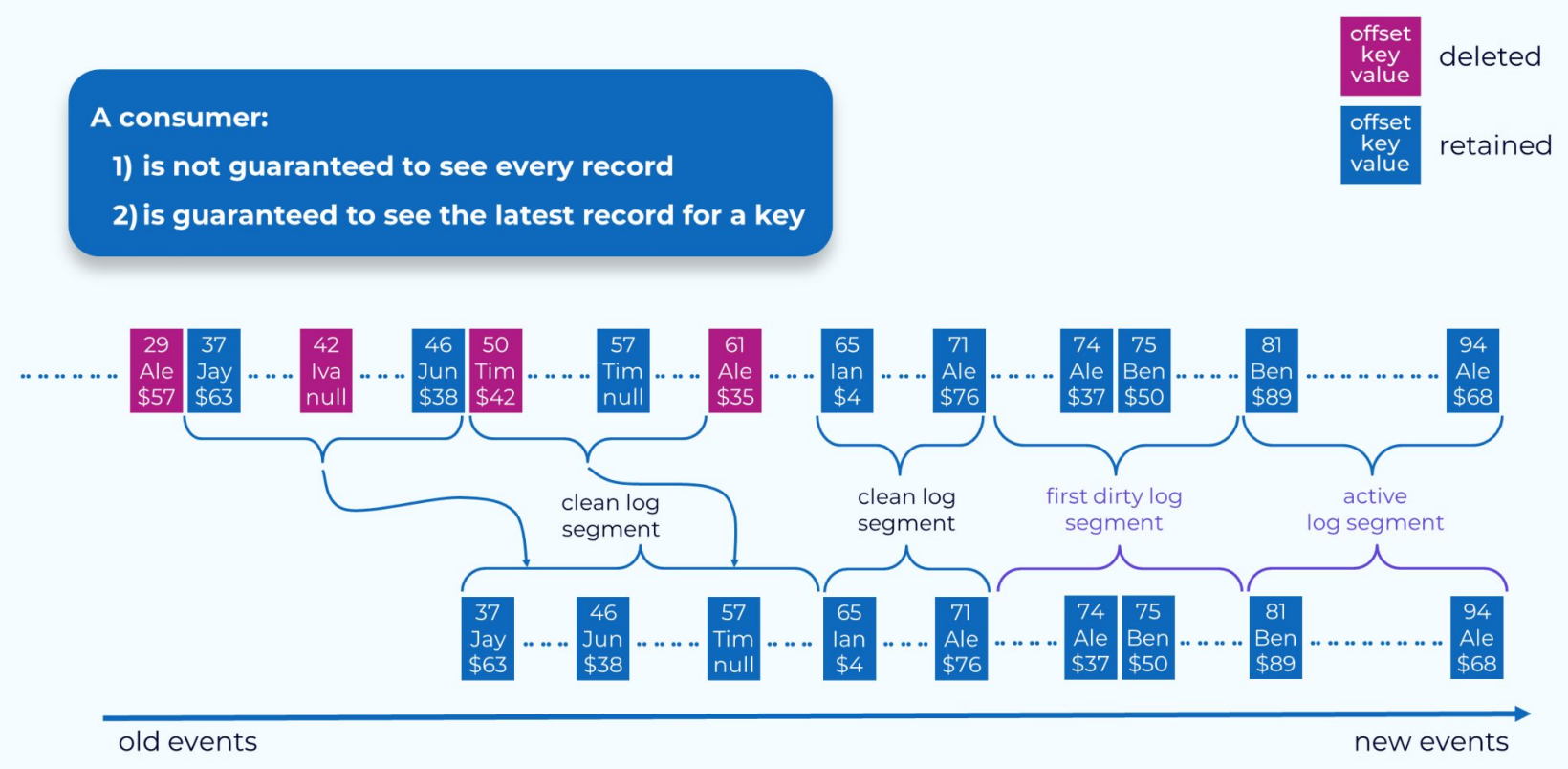
When Compaction Is Triggered

A topic partition is compacted if one of the following is true:

1. `dirty / total > min.cleanable.dirty.ratio` and
   `message timestamp < current time - min.compaction.lag.ms`
2. `message timestamp < current time - max.compaction.lag.ms`



Topic Compaction Guarantees



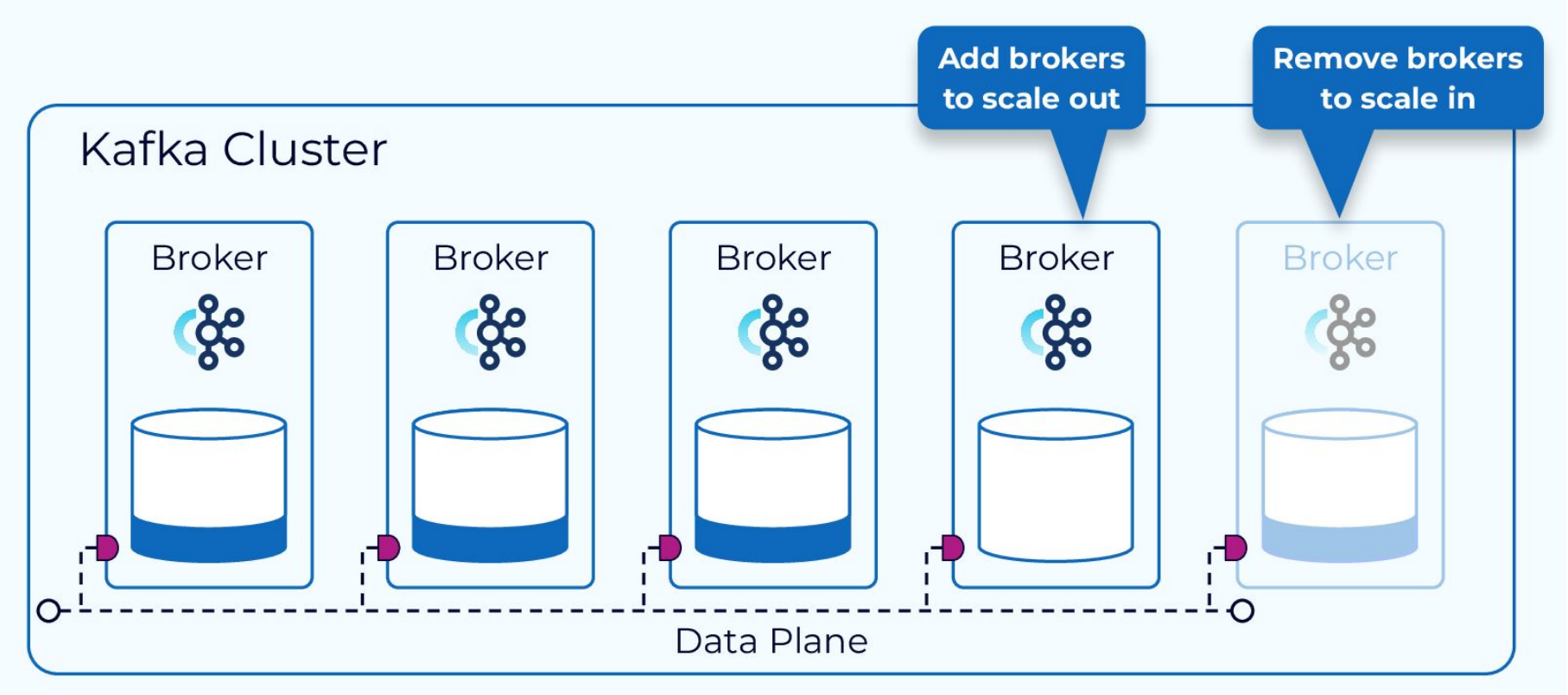
Demo-6:

Topic Compaction

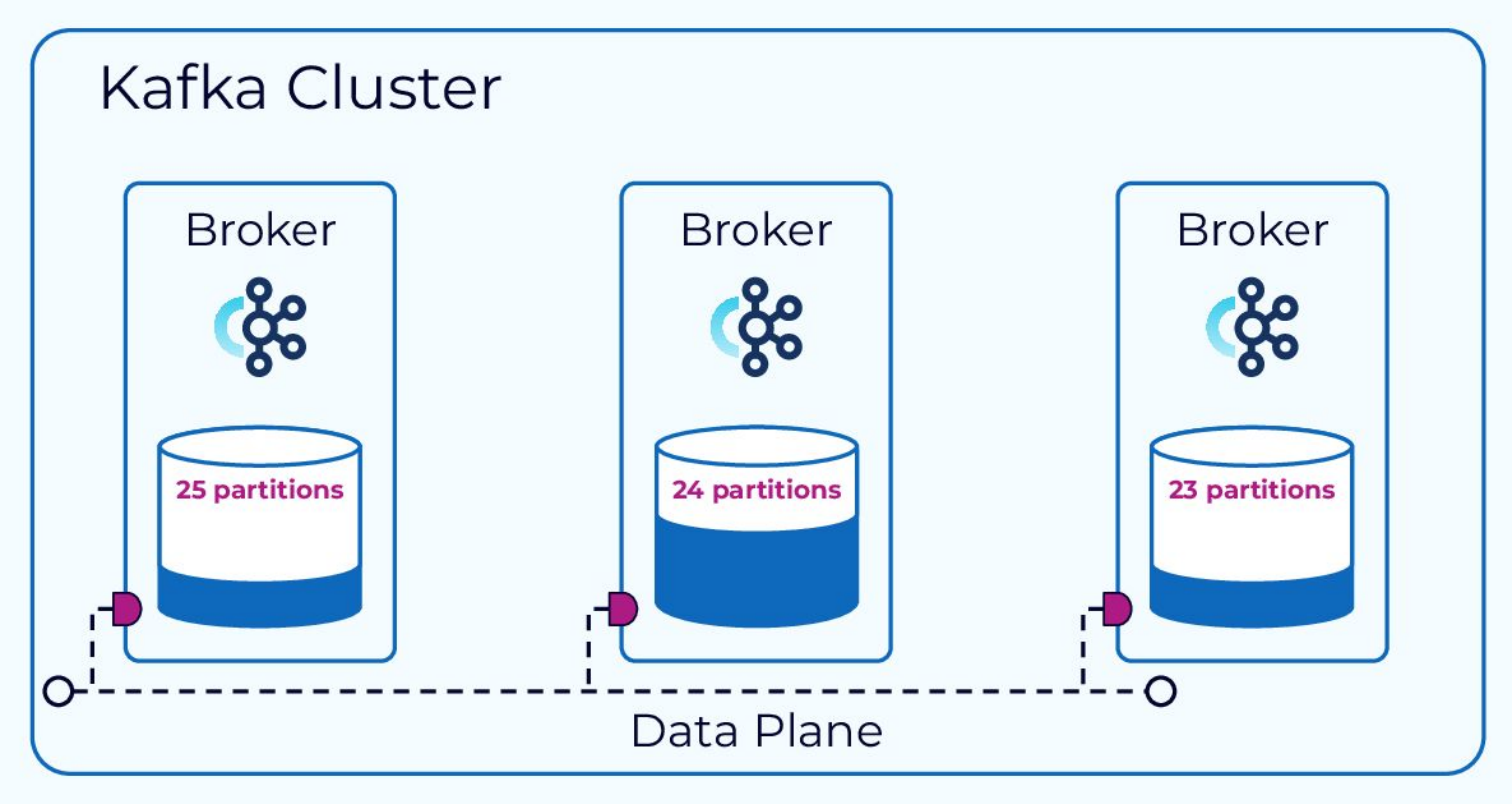
Any questions?

Cluster Elasticity

Cluster Scaling



Unbalanced Data Distribution



Automatic Cluster Scaling with Confluent Cloud

Kafka-reassign-partitions.sh

Included with Kafka core

1) Relocates partitions based upon JSON configuration file
2) Provides granular control
3) Manual process to create JSON configuration file

```
kafka-reassign-partitions \
    --bootstrap-server server-1:39094 \
    --reassignment-json-file reassign-topic-1.json \
    --execute
```

```
{"version":1,
    "partitions":[
        {"topic":"topic-1","partition":0,"replicas":[1,2,3]},
        {"topic":"topic-1","partition":1,"replicas":[2,3,1]},
        {"topic":"topic-1","partition":2,"replicas":[3,1,2]},
        {"topic":"topic-1","partition":3,"replicas":[1,2,3]},
        {"topic":"topic-1","partition":4,"replicas":[2,3,1]},
        {"topic":"topic-1","partition":5,"replicas":[3,1,2]}
    ]
}
```

Confluent Auto Data Balancer

Included with Confluent Platform

1) Automatically generates redistribution plan based on stats
2) When the plan is executed, partitions are rebalanced across brokers in the cluster
3) These are both manual steps done with the `confluent-rebalancer` command

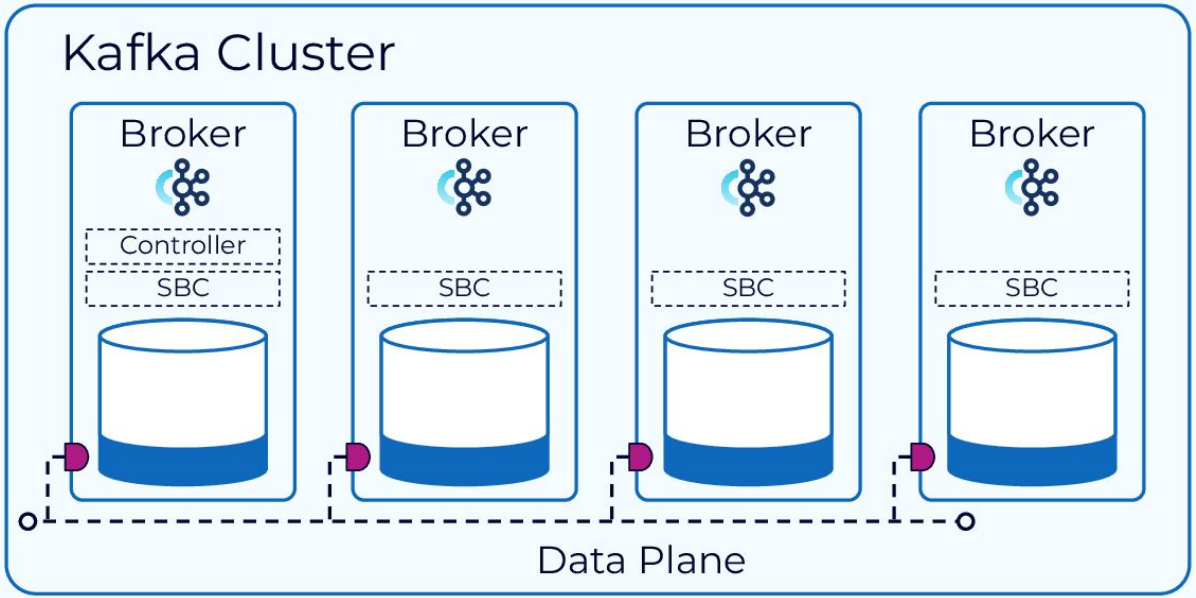
```
confluent-rebalancer execute \
    --bootstrap-server kafka-1:9092 \
    --metrics-bootstrap-server kafka-1:9092 \
    --throttle 1000000 \
    --verbose
```

```
Computing the rebalance plan (this may take a while) ...
You are about to move 512 replica(s) for 512 partitions to 1 broker(s) with total size 233.9 MB.
The preferred leader for 192 partition(s) will be changed.
In total, the assignment for 515 partitions will be changed.
The minimum free volume space is set to 20.0%.

The following brokers will have less than 40% of free volume space during the rebalance:
    Broker  Current Size (MB)  Size During Rebalance (MB)  Free % During Rebalance  Size After Rebalance (MB)  Free % After Rebalance
    102     311.7              311.7                       37.6                     233.8                      37.6
    104     0                  233.9                       36.8                     233.9                      36.8
    103     311.7              311.7                       37.6                     233.7                      37.6
    101     311.7              311.7                       37.6                     233.8                      37.6

Min/max stats for brokers (before -> after):
    Type  Leader Count                  Replica Count                 Size (MB)
    Min   0 (id: 104) -> 171 (id: 101) 0 (id: 104) -> 510 (id: 102) 0 (id: 104) -> 233.7 (id: 103)
    Max   229 (id: 102) -> 171 (id: 101) 684 (id: 102) -> 515 (id: 101) 311.7 (id: 101) -> 233.9 (id: 104)
No racks are defined.

Broker stats (before -> after):
    Broker     Leader Count    Replica Count   Size (MB)           Free Space (%)
    101        226 -> 171      683 -> 515      311.7 -> 233.8      37.6 -> 37.8
    102        229 -> 171      684 -> 510      311.7 -> 233.8      37.6 -> 37.8
    103        229 -> 171      683 -> 513      311.7 -> 233.7      37.6 -> 37.8
    104        0 -> 171        0 -> 512        0 -> 233.9          37.6 -> 36.8

Would you like to continue? (y/n):

The rebalance has been started, run `status` to check progress.

Warning: You must run the `status` or `finish` command periodically, until the rebalance completes, to ensure the throttle is
removed. You can also alter the throttle by re-running the execute command passing a new value.
```

Confluent Self-Balancing Clusters (SBC)

Advantages over Auto Data Balancer

1) No additional tools to run (built into the brokers)
2) Cluster balance is continuously monitored so that rebalances run whenever they are needed
3) Much faster rebalancing (several orders of magnitude)



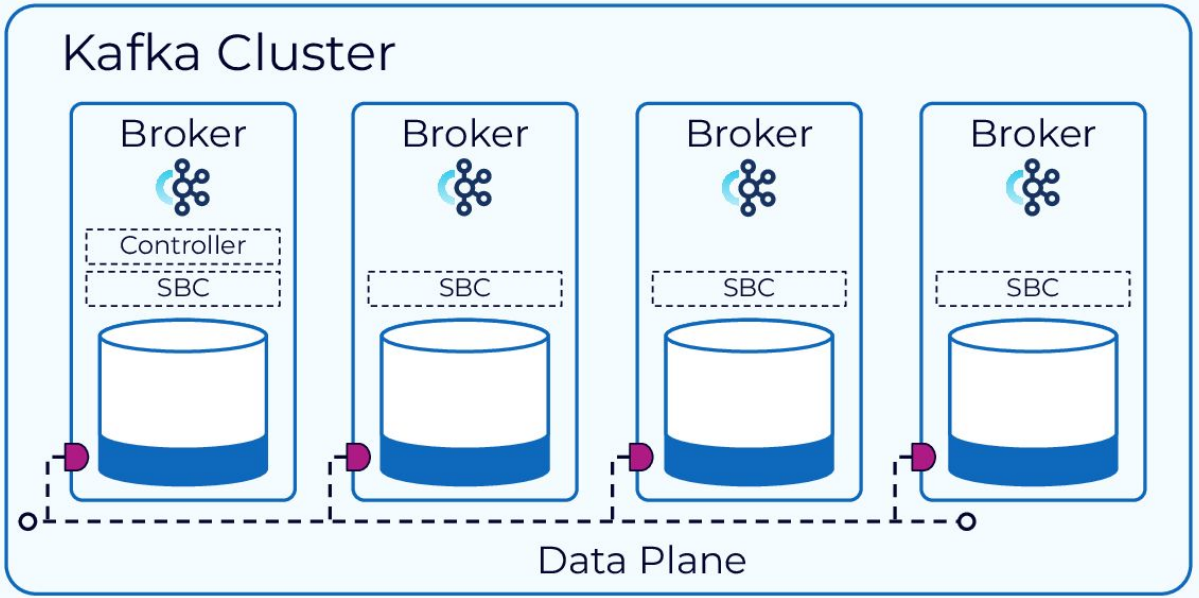
SBC Metrics Collection & Processing

SBC runs on each broker:

1) Collects metrics for that broker
2) Writes the metrics to an internal metrics collection topic

SBC on the controller node:

1) Aggregates the metrics
2) Generates load balance plans based on goals
3) Executes the plan and exposes monitoring data to Control Center



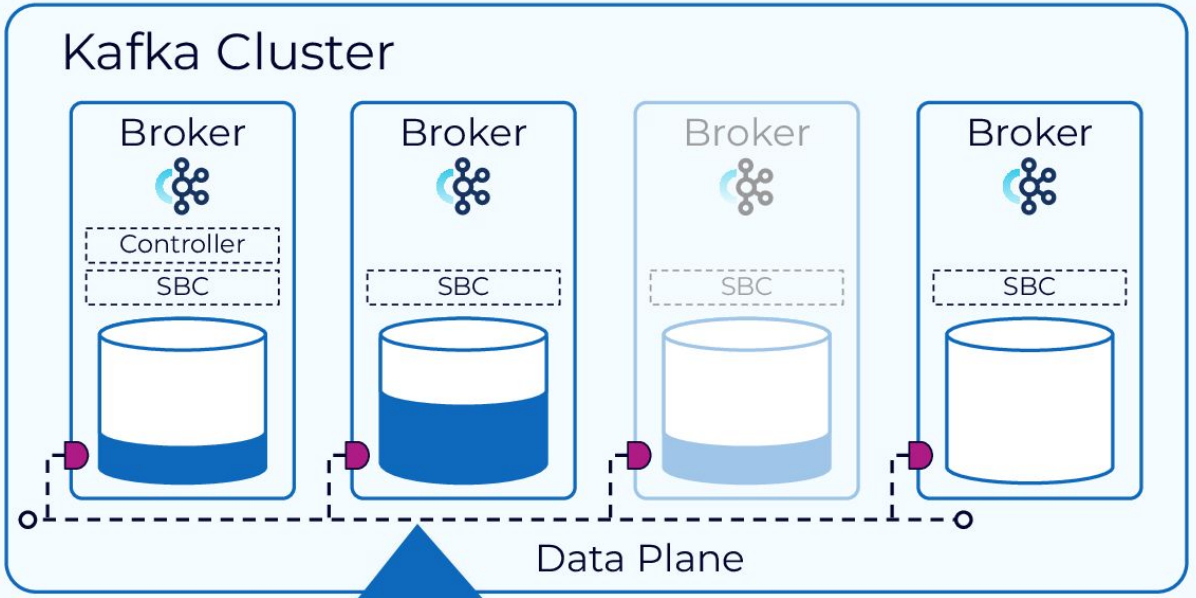
SBC Rebalance Triggers

Auto rebalance trigger options:

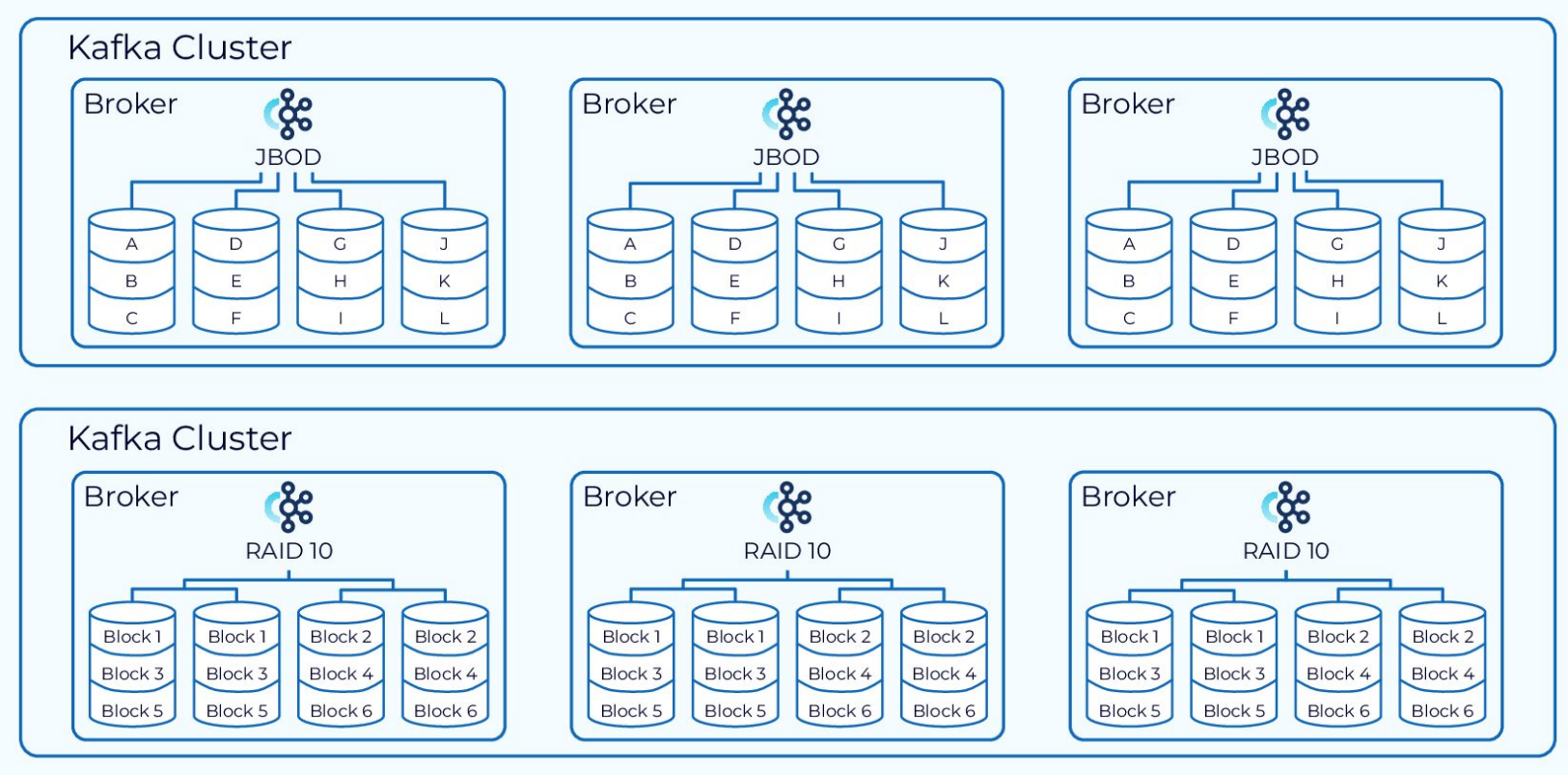
1) Added and removed brokers
2) Any uneven load

Uneven load condition is based upon:

1) Disk usage and network usage
2) Number of partitions/replicas per broker
3) Leadership and rack awareness

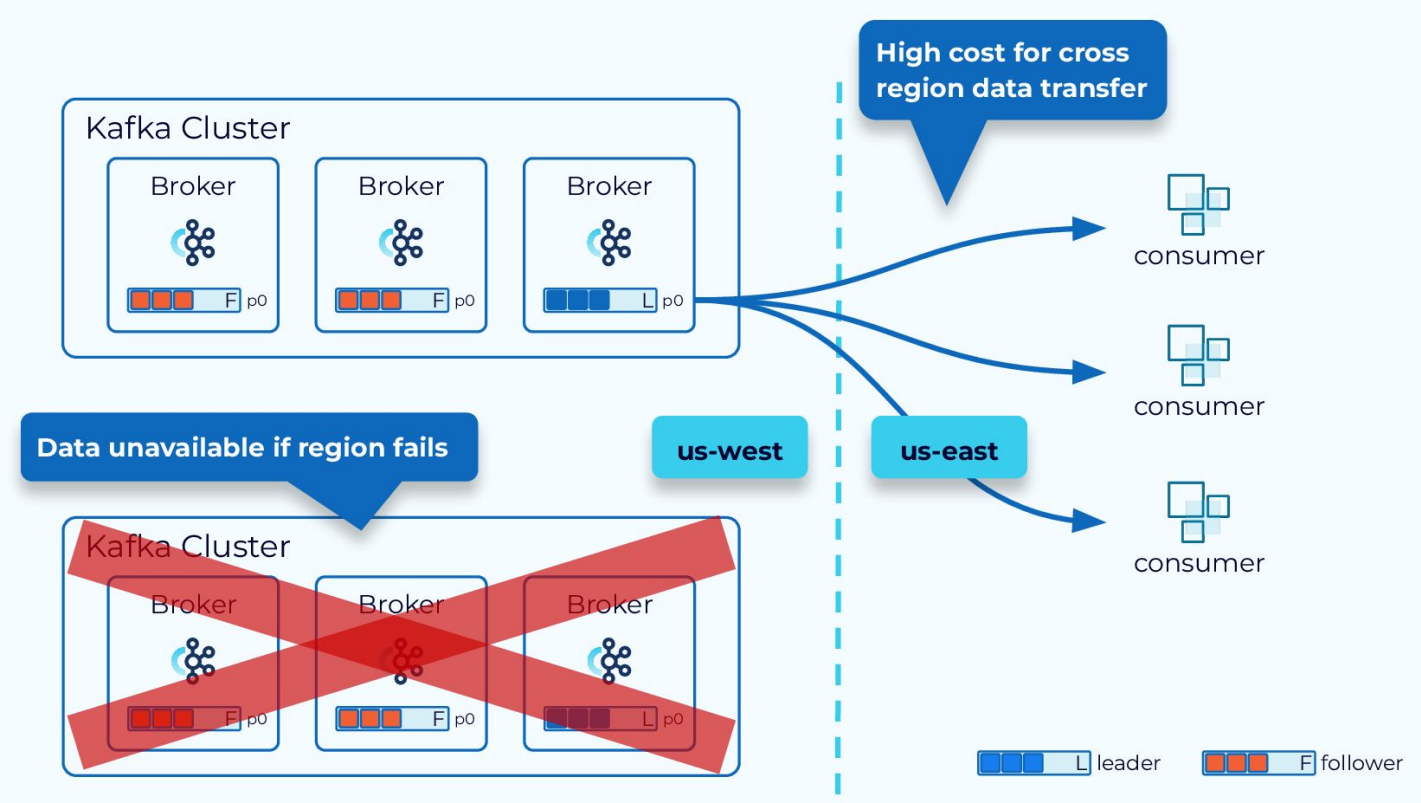


JBOD vs RAID

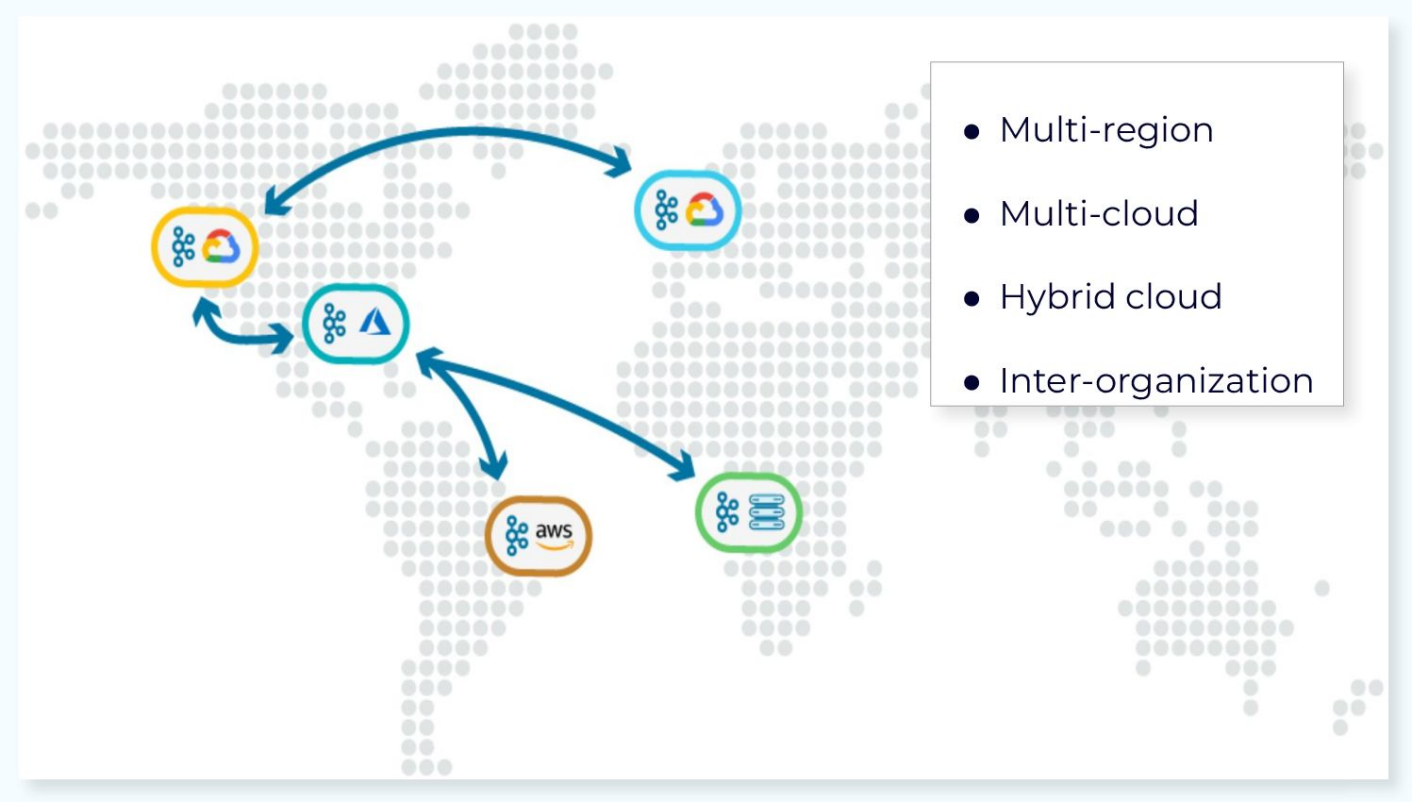

Kafka Cluster
Broker
JBOD
A
B
C
D
E
F
G
H
I
J
K
L
Broker
JBOD
A
B
C
D
E
F
G
H
I
J
K
L
Broker
JBOD
A
B
C
D
E
F
G
H
I
J
K
L
Kafka Cluster
Broker
RAID 10
Block 1
Block 3
Block 5
Block 1
Block 3
Block 5
Block 2
Block 4
Block 6
Block 2
Block 4
Block 6
Broker
RAID 10
Block 1
Block 3
Block 5
Block 1
Block 3
Block 5
Block 2
Block 4
Block 6
Block 2
Block 4
Block 6
Broker
RAID 10
Block 1
Block 3
Block 5
Block 1
Block 3
Block 5
Block 2
Block 4
Block 6
Block 2
Block 4
Block 6

Geo-Replication

Single-Region Cluster Concerns

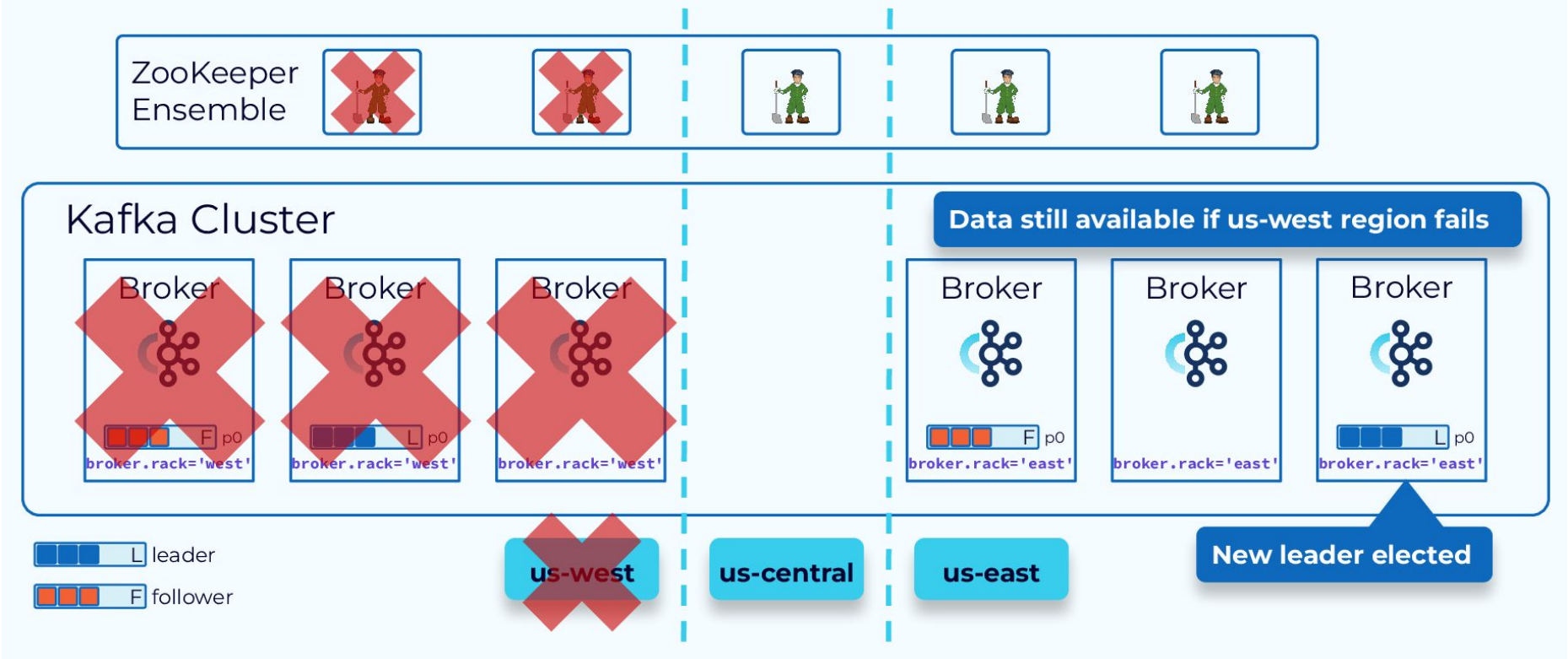


Geo-Replication

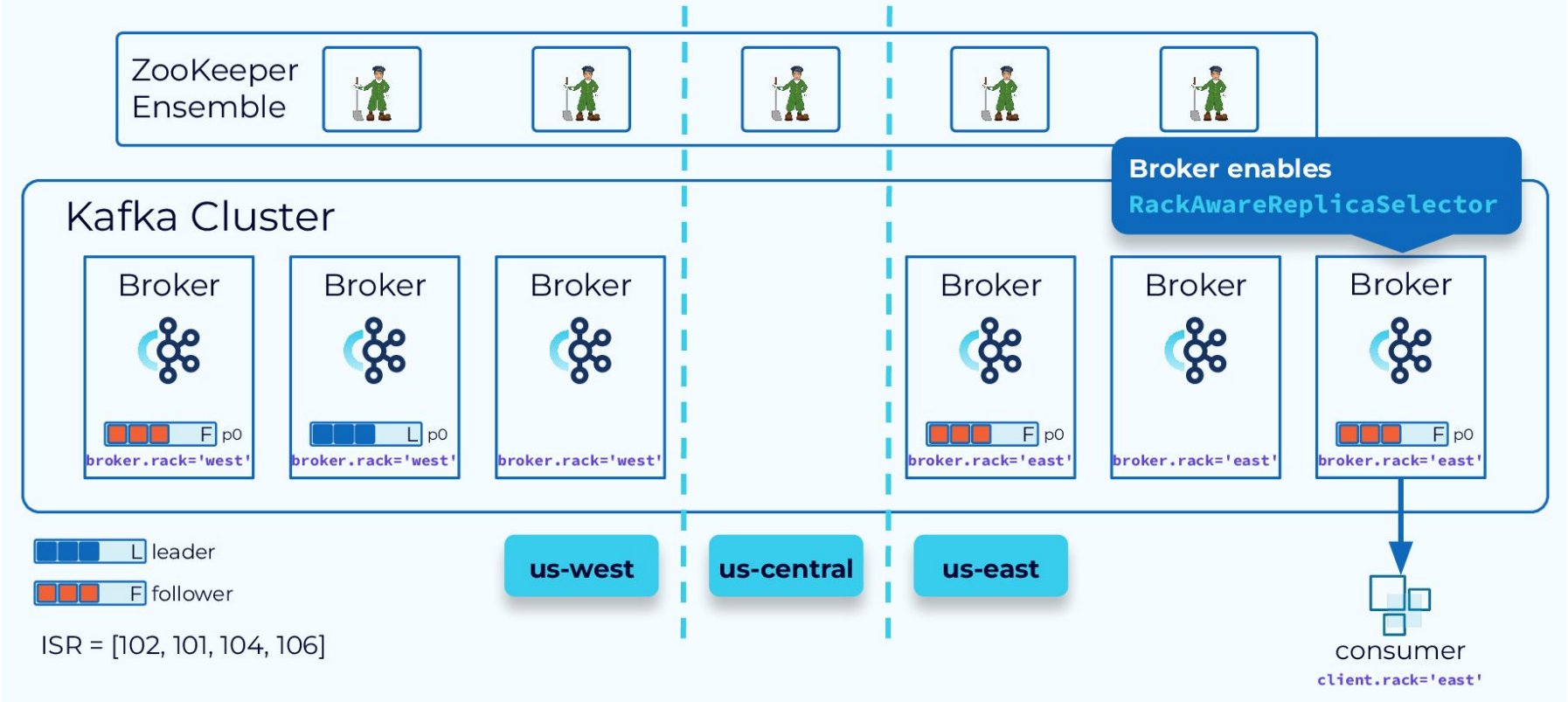


- Multi-region
- Multi-cloud
- Hybrid cloud
- Inter-organization

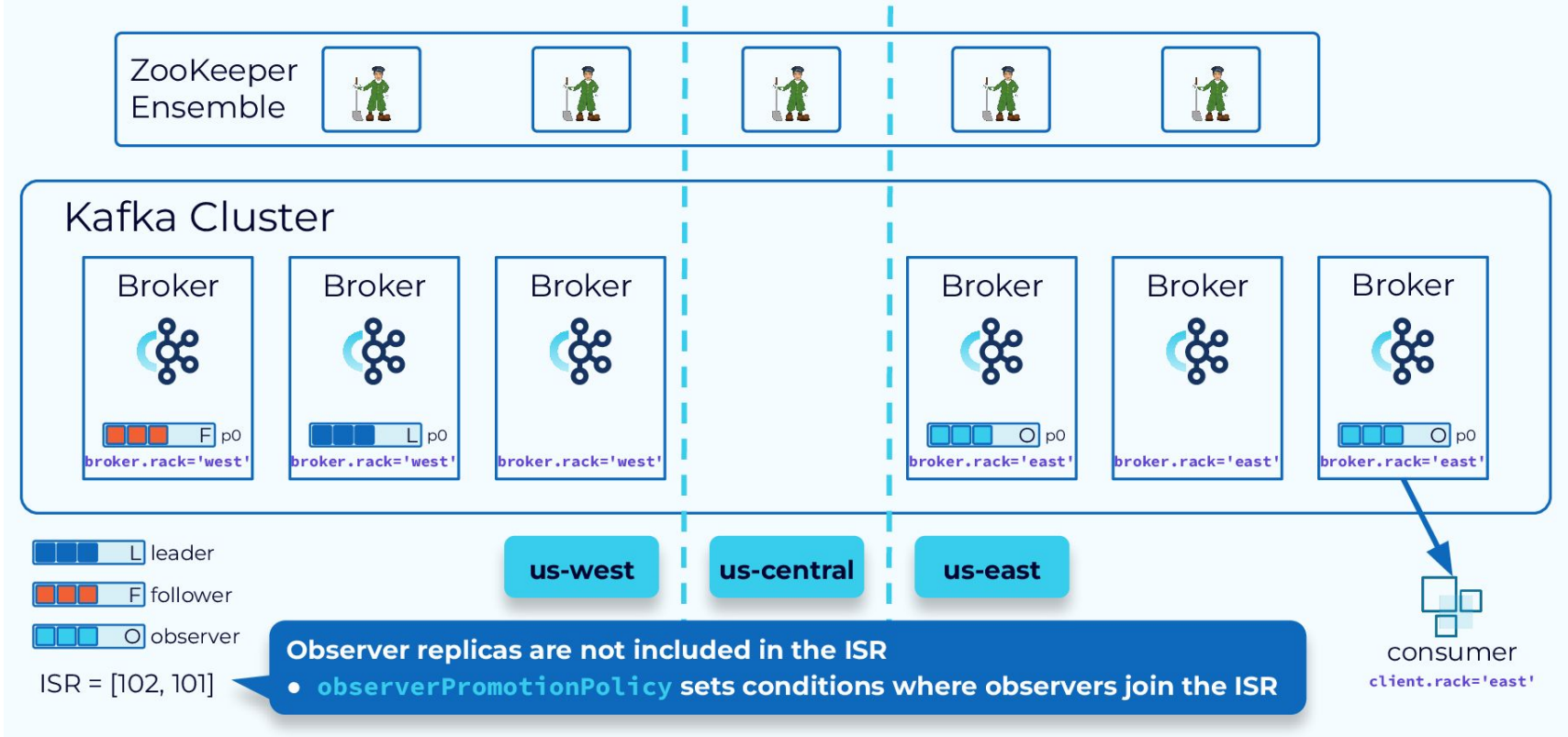
Confluent Multi-Region Cluster (MRC)



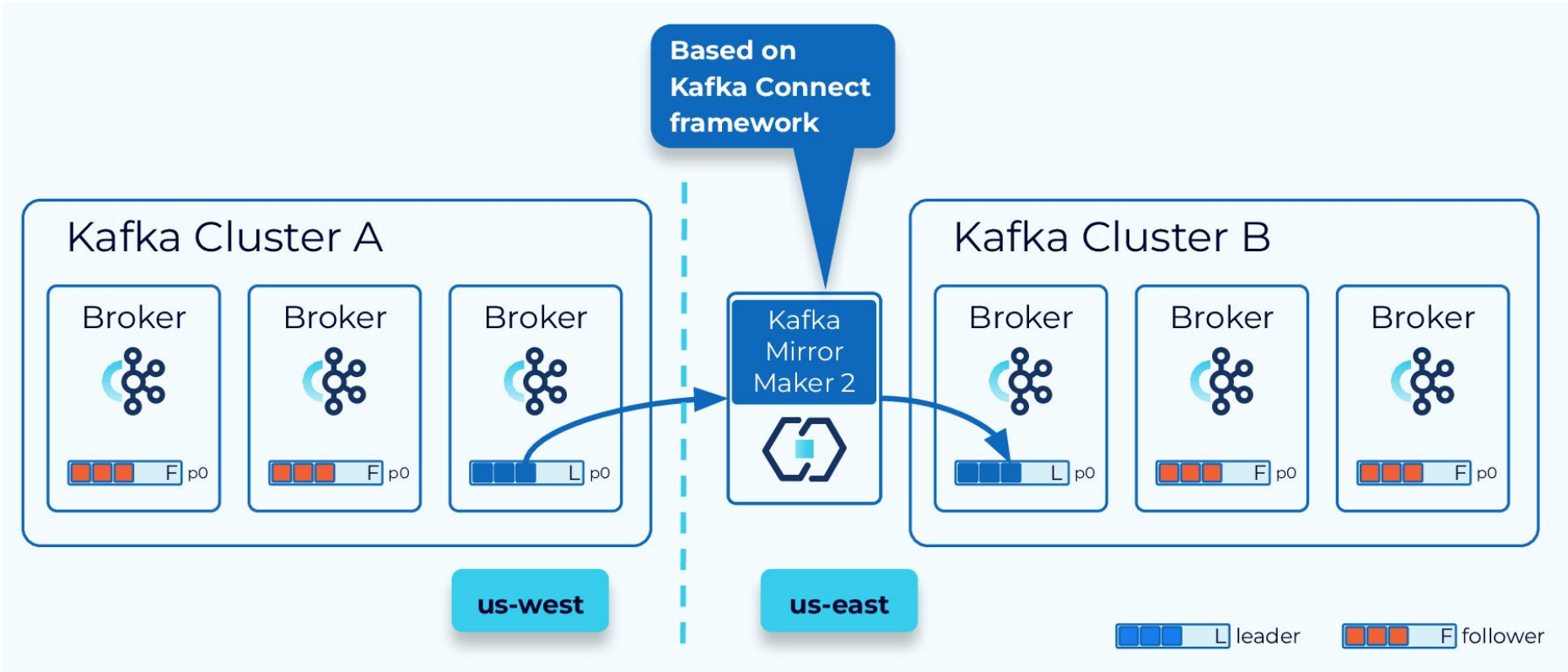
Better Locality with Fetch From Follower



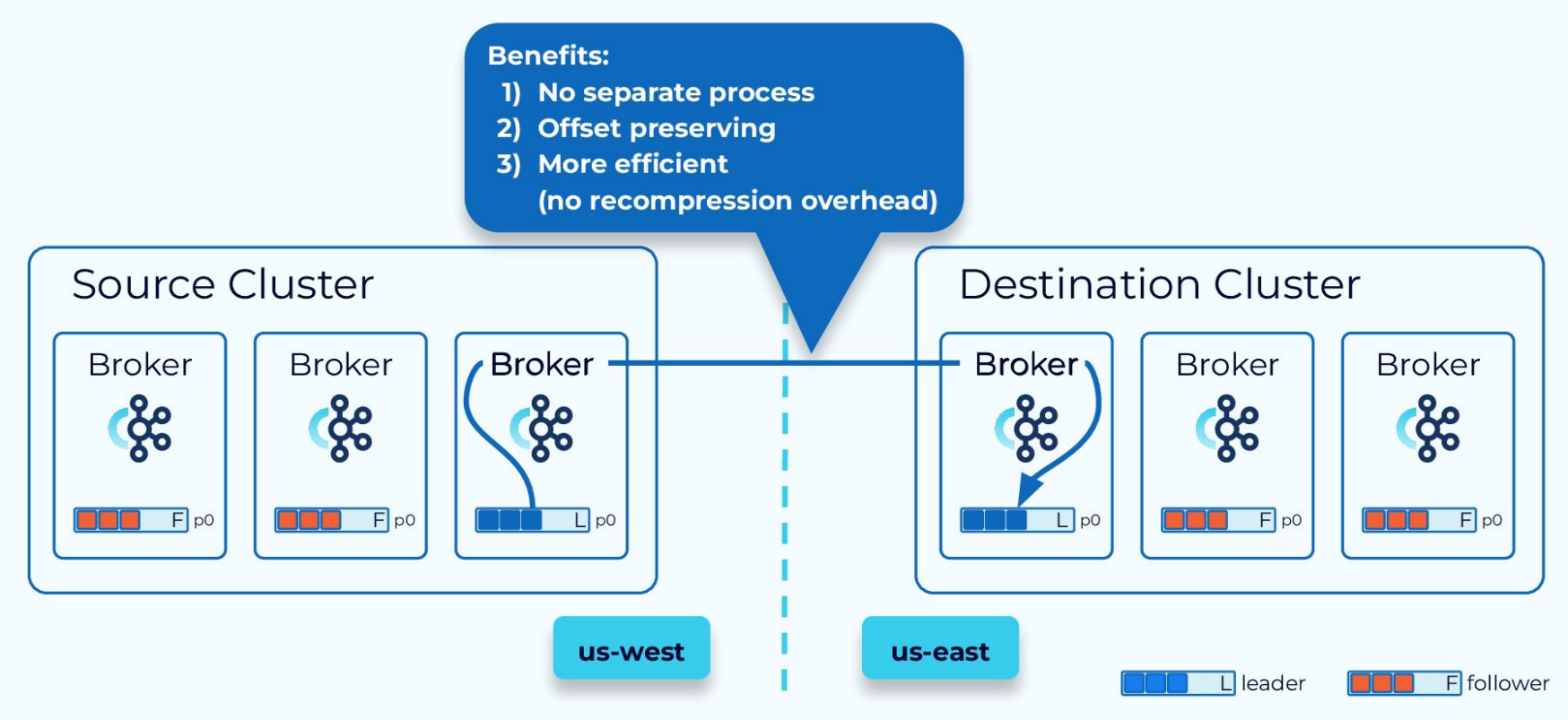
Async Replication with Observers



Kafka MirrorMaker 2



Confluent Cluster Linking



Recap

Feature	Sync vs. Async	Offset Preserving	Built-In to the Cluster	Latency Tolerant
Confluent Multi-Region Cluster	Sync or Async	Yes	Yes	No (dark fiber recommended)
Kafka MirrorMaker 2	Async	No (manual offset translation)	No	Yes
Confluent Replicator	Async	No (auto offset translation for Java consumers)	No	Yes
Confluent Cluster Linking	Async	Yes	Yes	Yes