CONFLUENT
Developer

Apache Kafka®
Internal Architecture

Contents

CONFLUENT Developer

CONFLUENT
Developer

The Fundamentals

What's Covered

CONFLUENT Developer

1 Inside the Broker

Control Panel, Replication Protocols, Topic Compaction, etc.

2 Durability, Availability and Ordering Guarantees

3 Consumer Group Protocol

4 Transactions

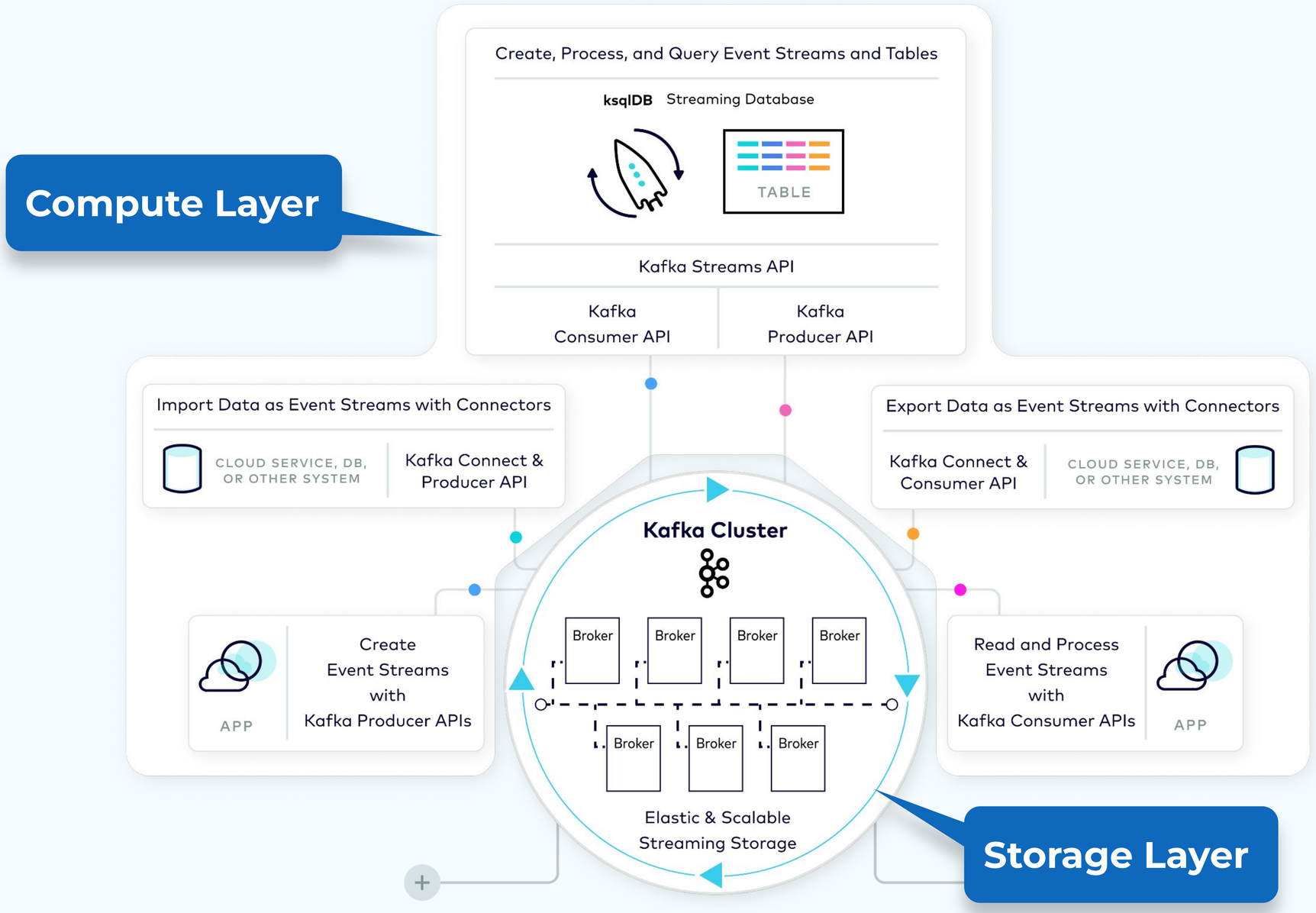
5 Tiered Storage

6 Elasticity

7 Geo-Replication

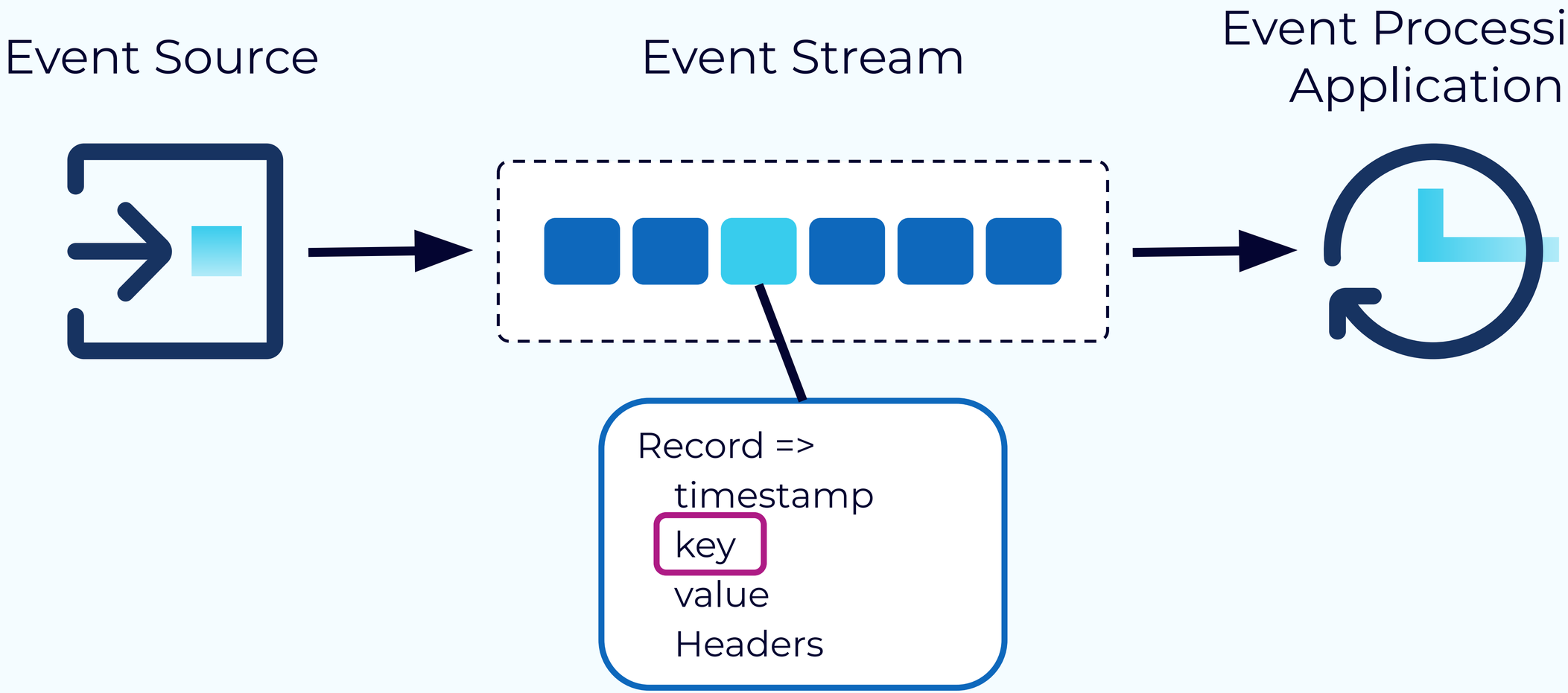
Overview of Kafka Architecture

CONFLUENT Developer



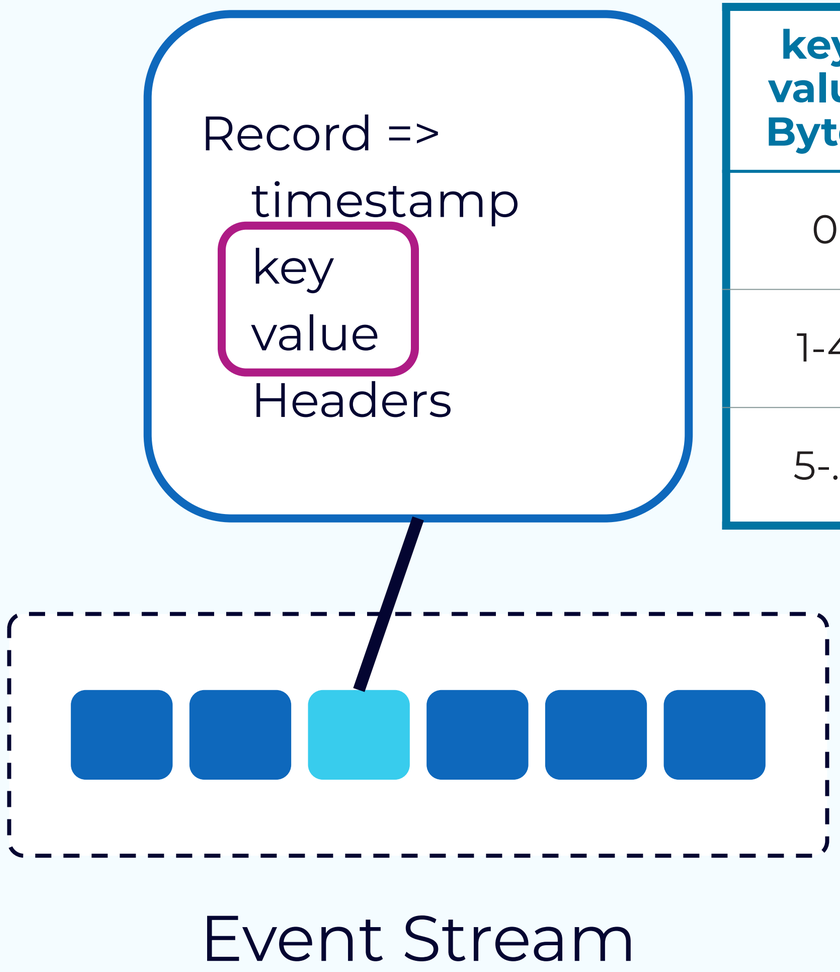
Events

CONFLUENT Developer



Record Schema

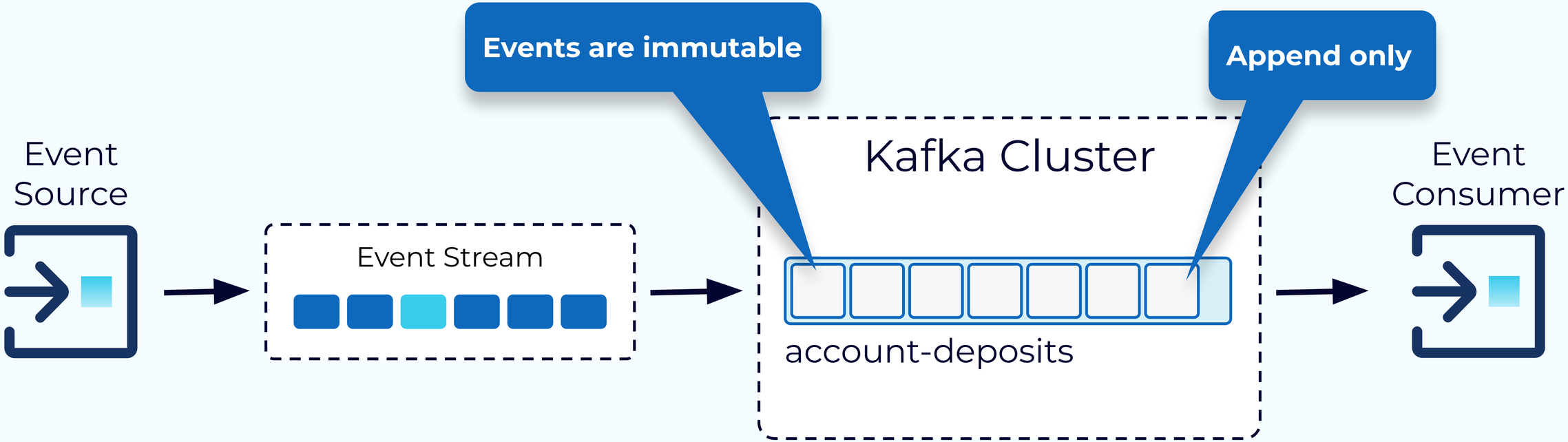
CONFLUENT Developer



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.

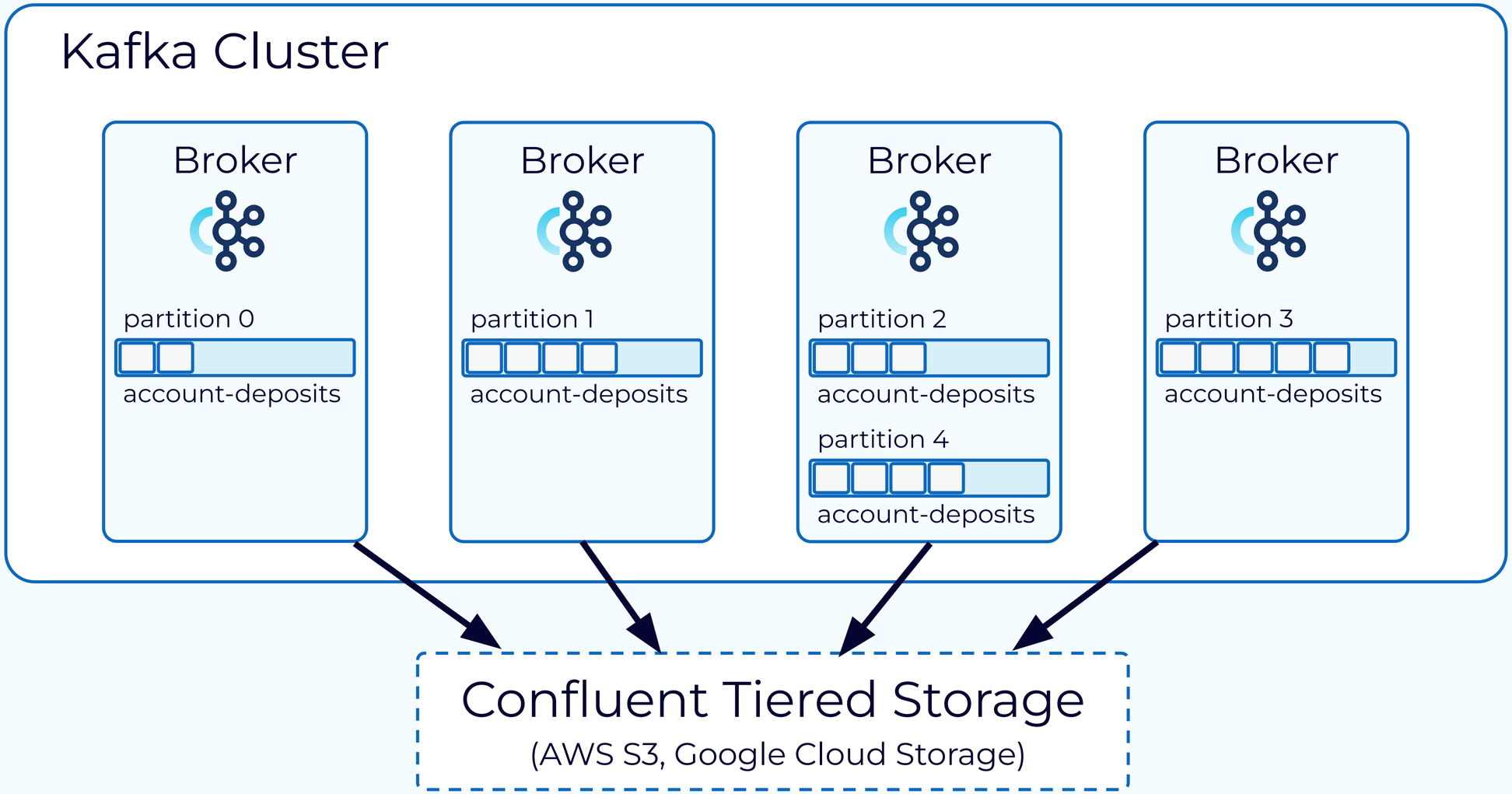
Kafka Topics

CONFLUENT Developer



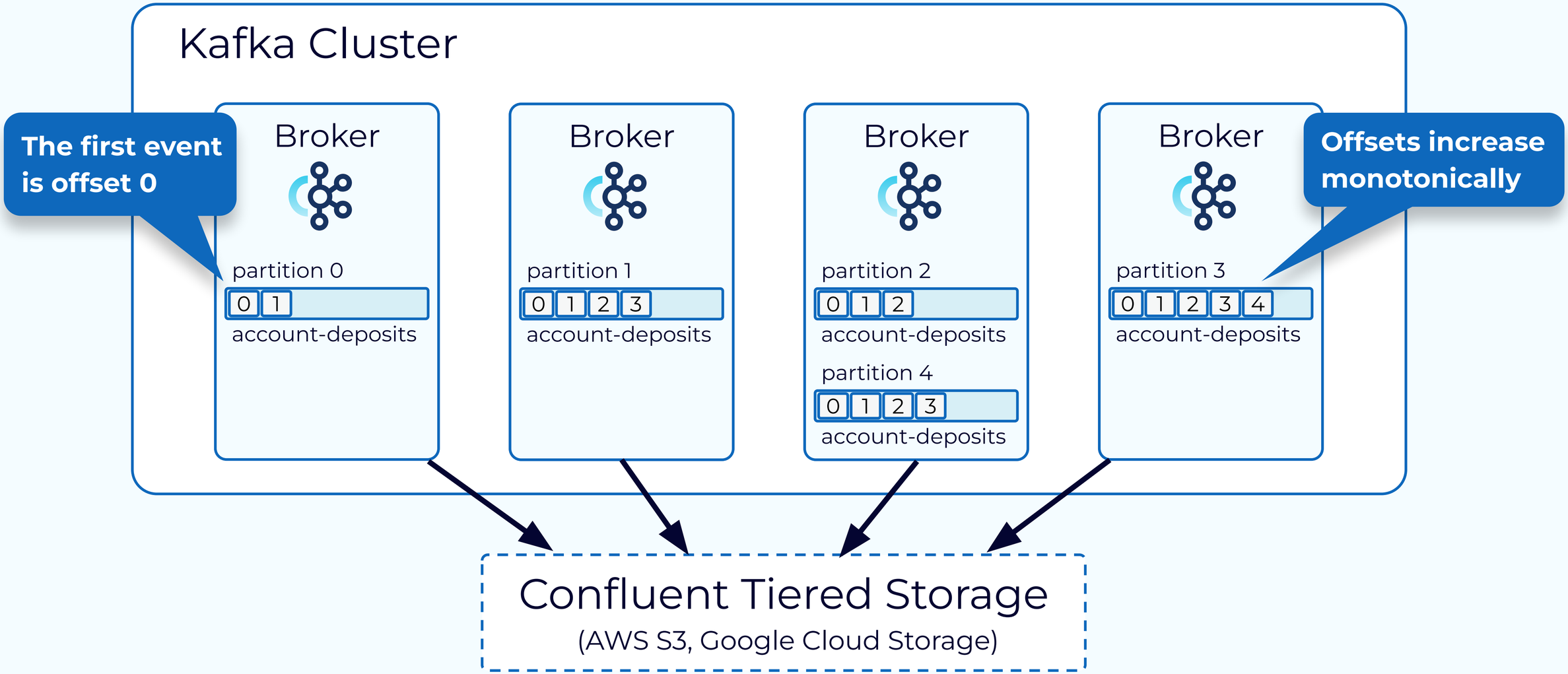
Topic Partitions

CONFLUENT Developer



Partition Offsets

CONFLUENT Developer

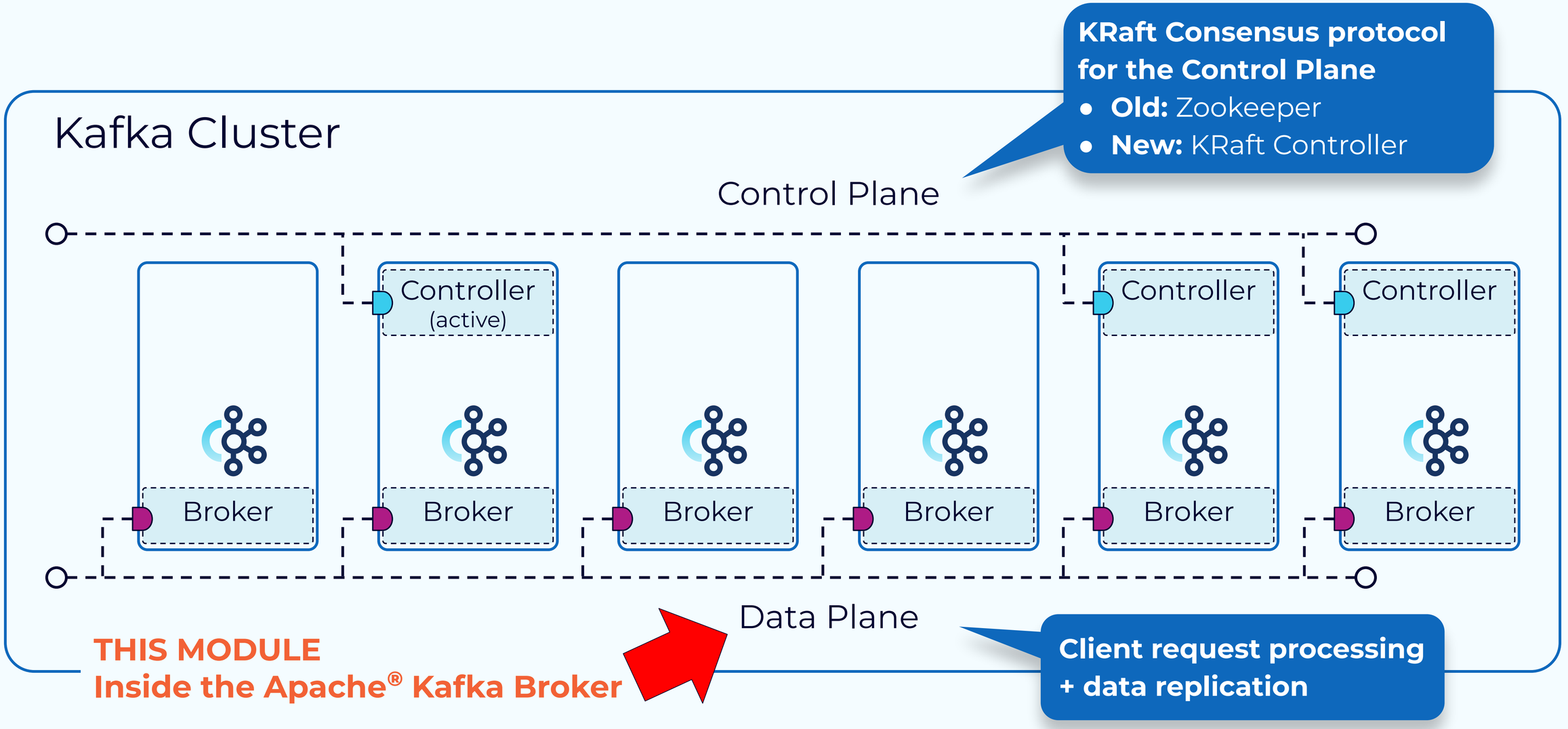


CONFLUENT
Developer

Inside the Apache Kafka® Broker

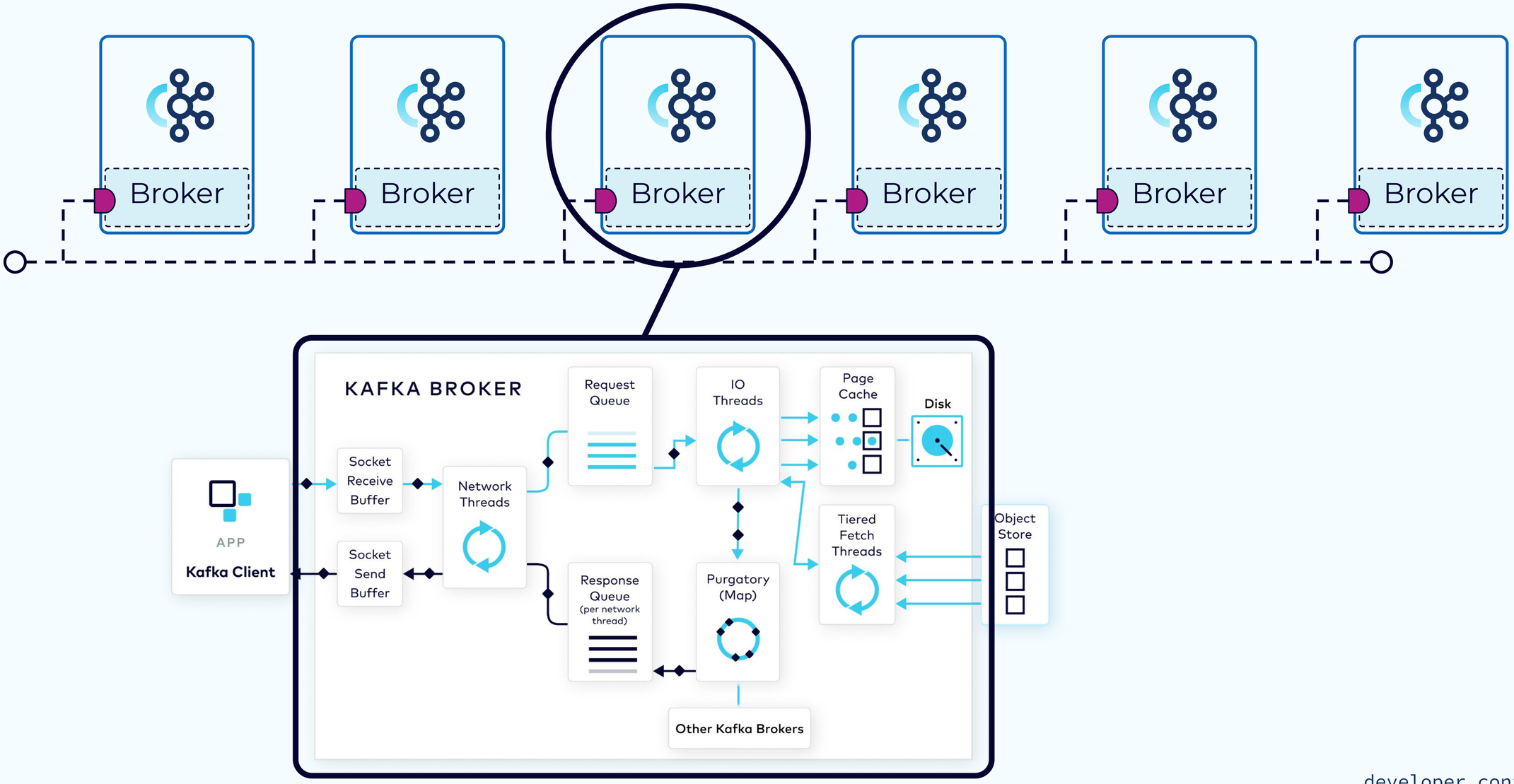
Kafka Manages Data & Metadata Separately

CONFLUENT Developer



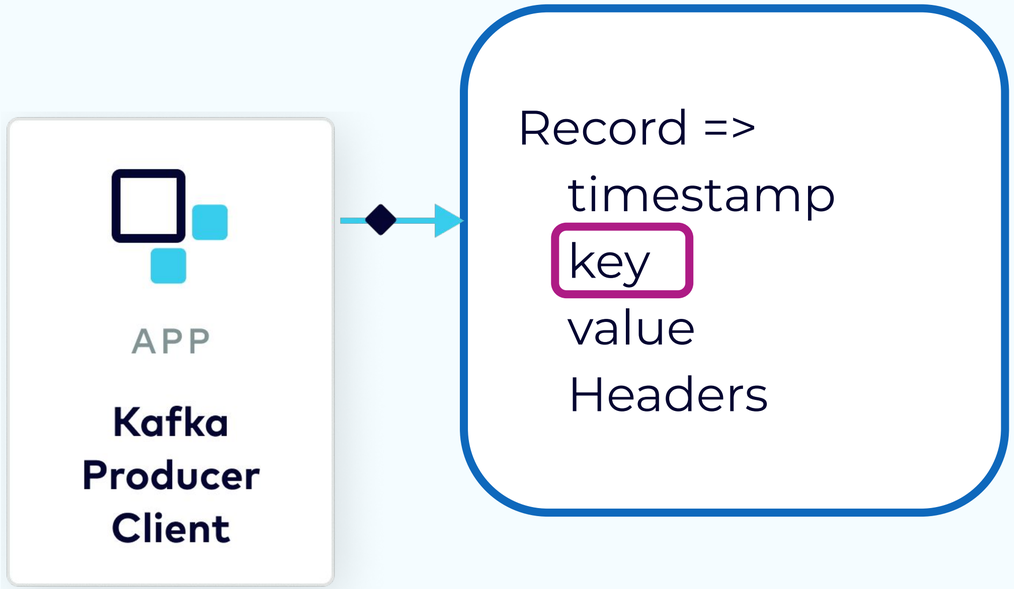
Inside The Apache Kafka Broker

CONFLUENT Developer

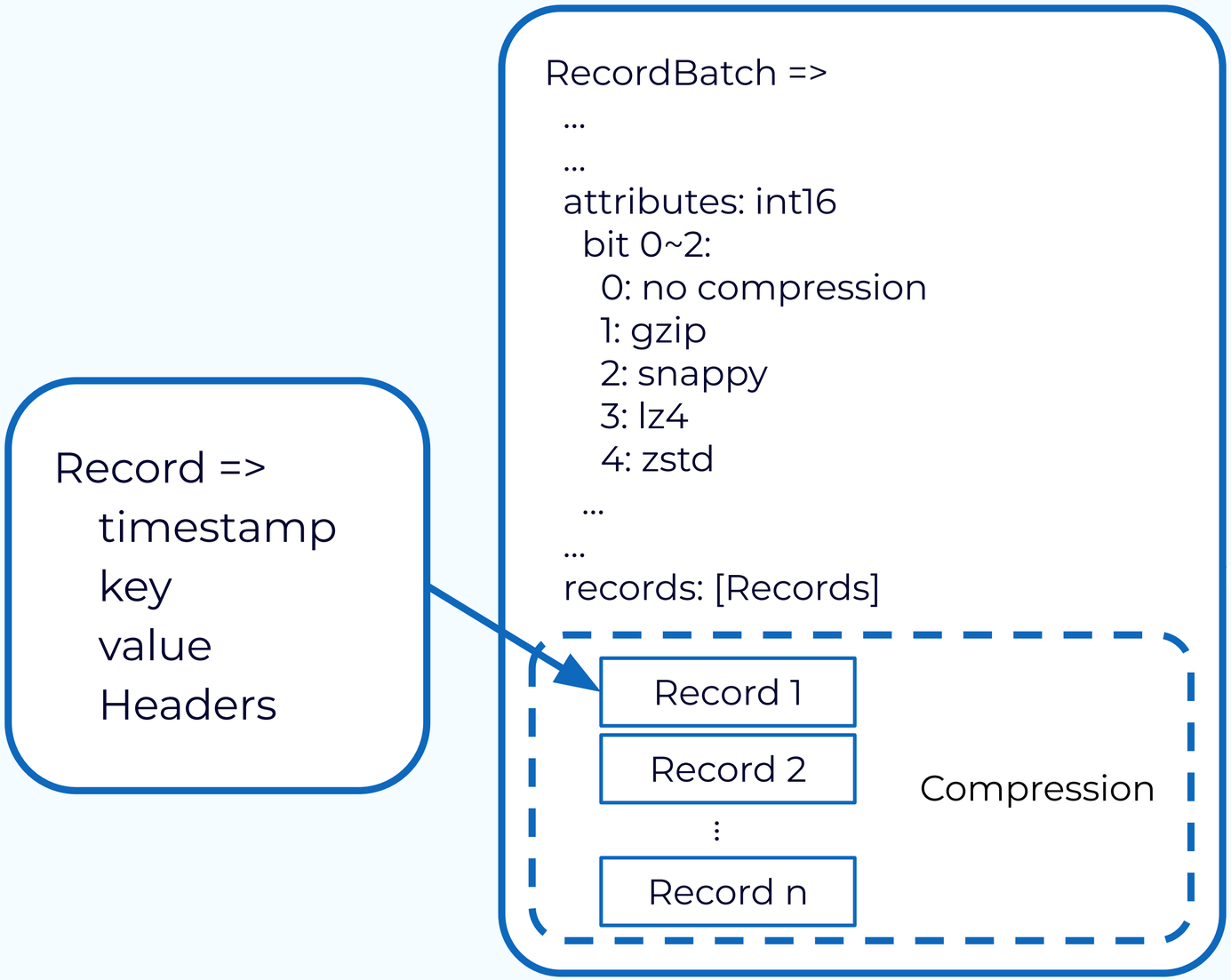


Assign Record to Topic Partition

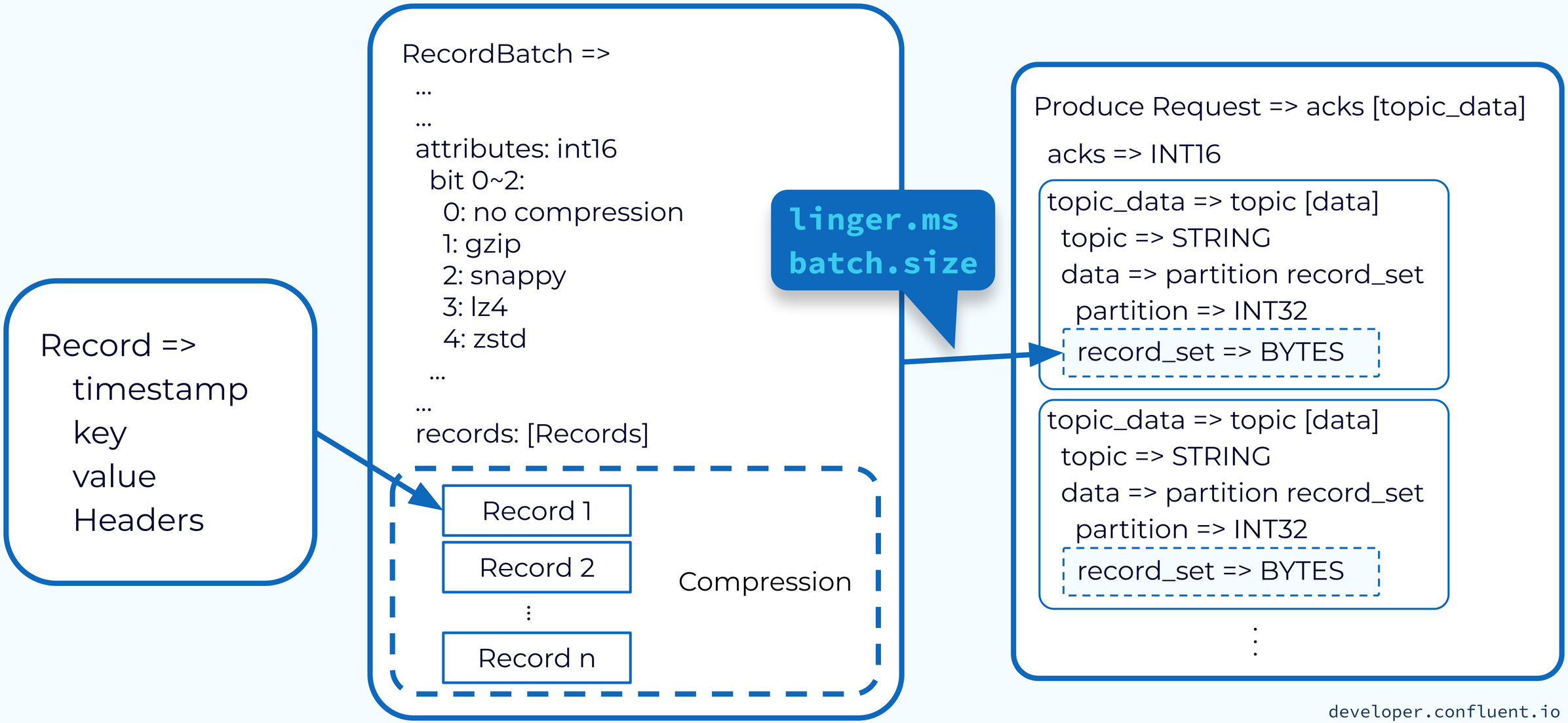
CONFLUENT Developer



Records Accumulated into Record Batches

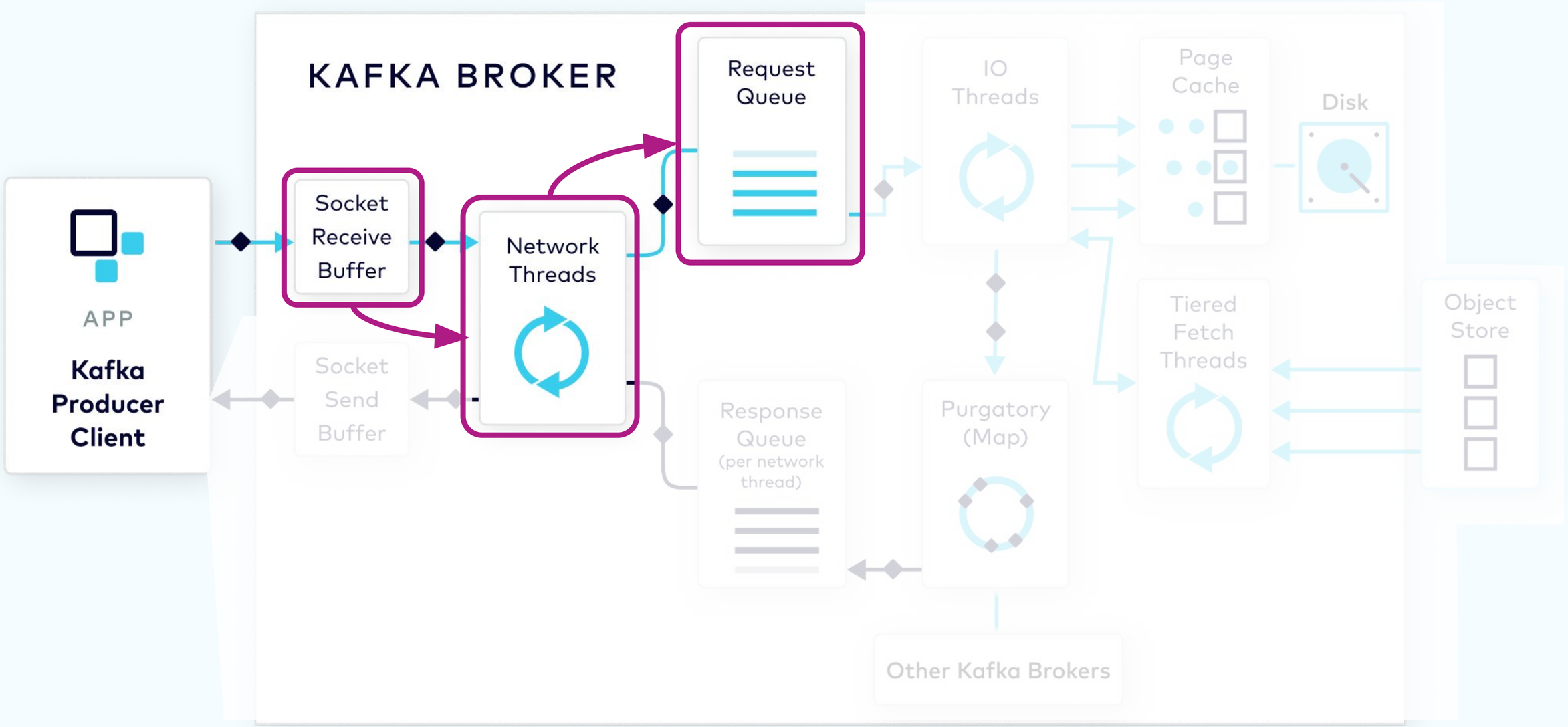


Record Batches Drained into Produce Requests



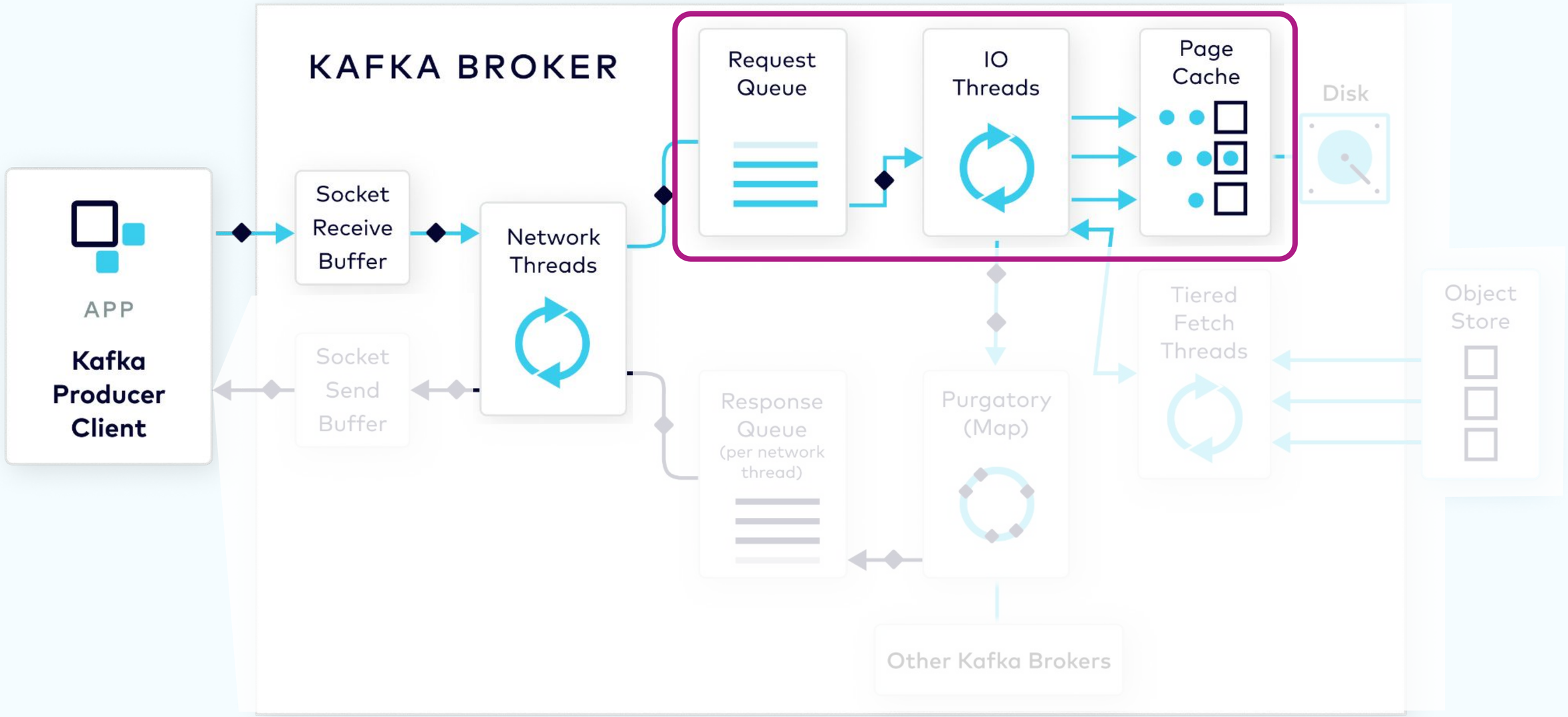
Network Thread Adds Request to Queue

CONFLUENT Developer



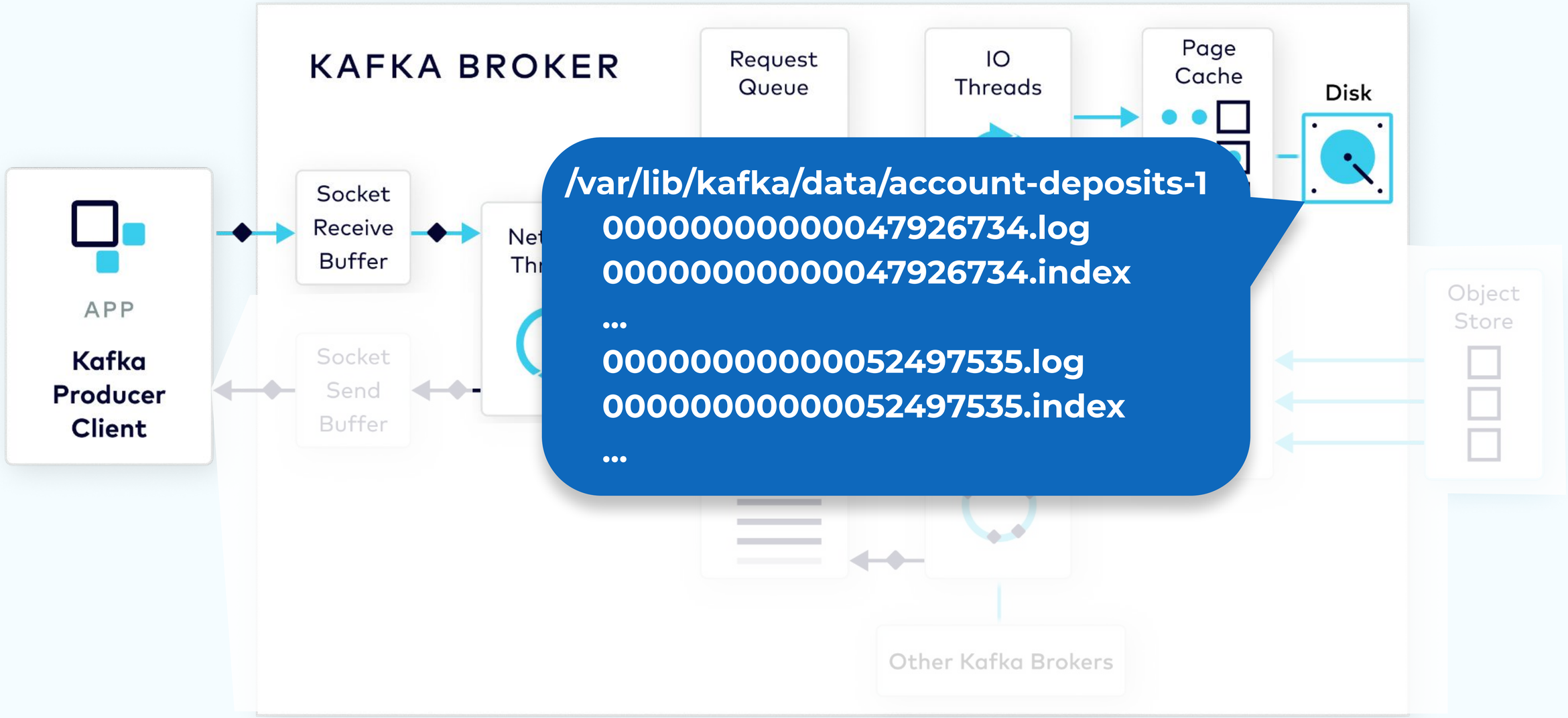
IO Thread Verifies Record Batch And Stores

CONFLUENT Developer



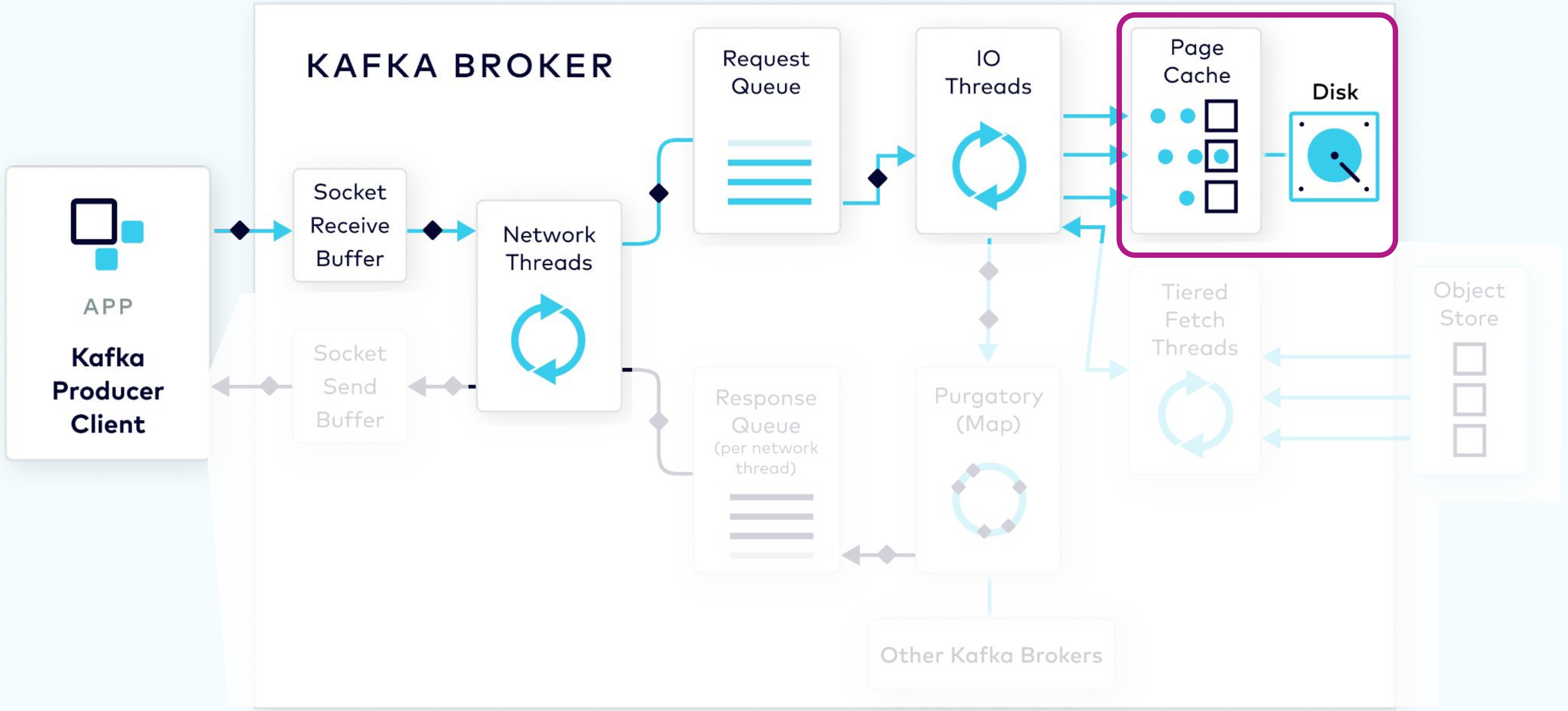
Kafka Physical Storage

CONFLUENT Developer



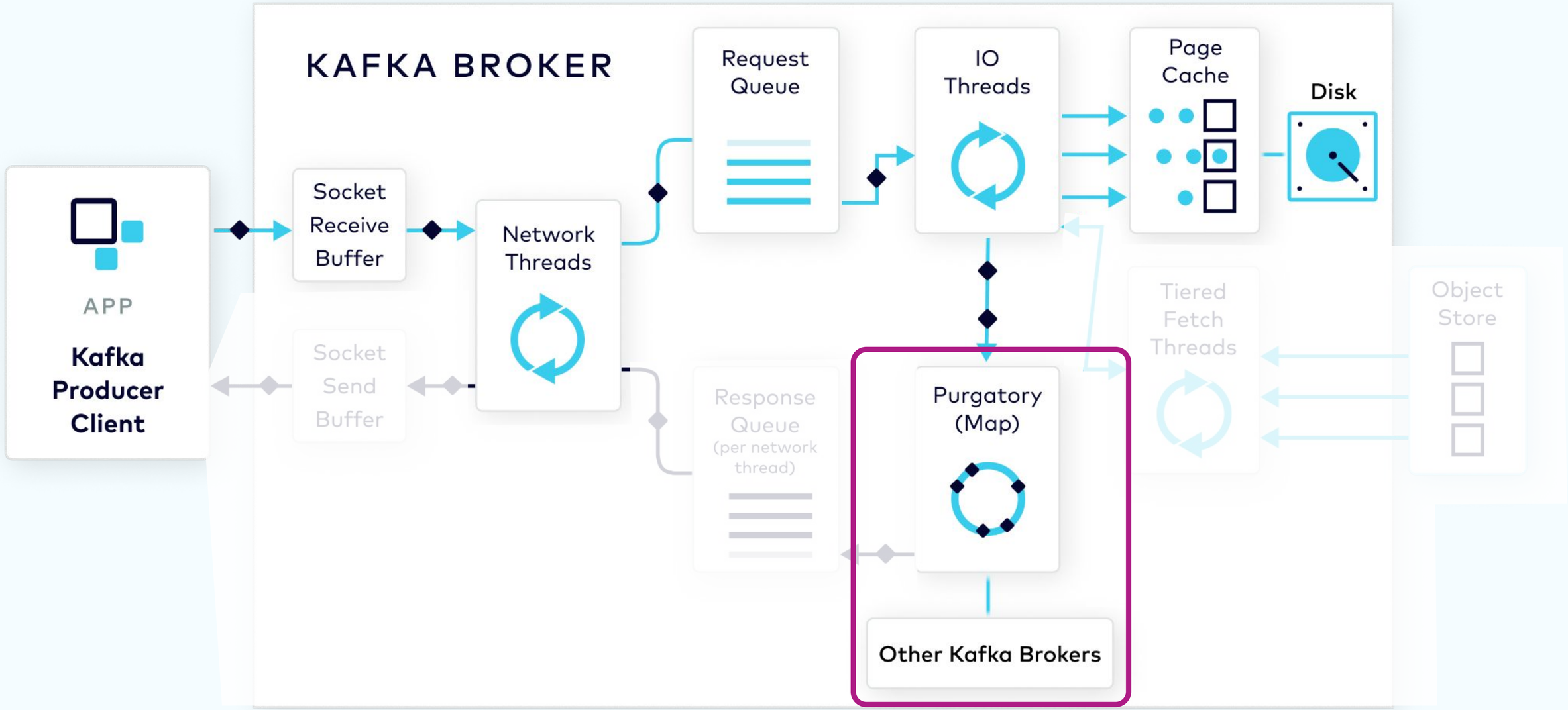
By Default, Data Is Written Asynch to Disk

CONFLUENT Developer



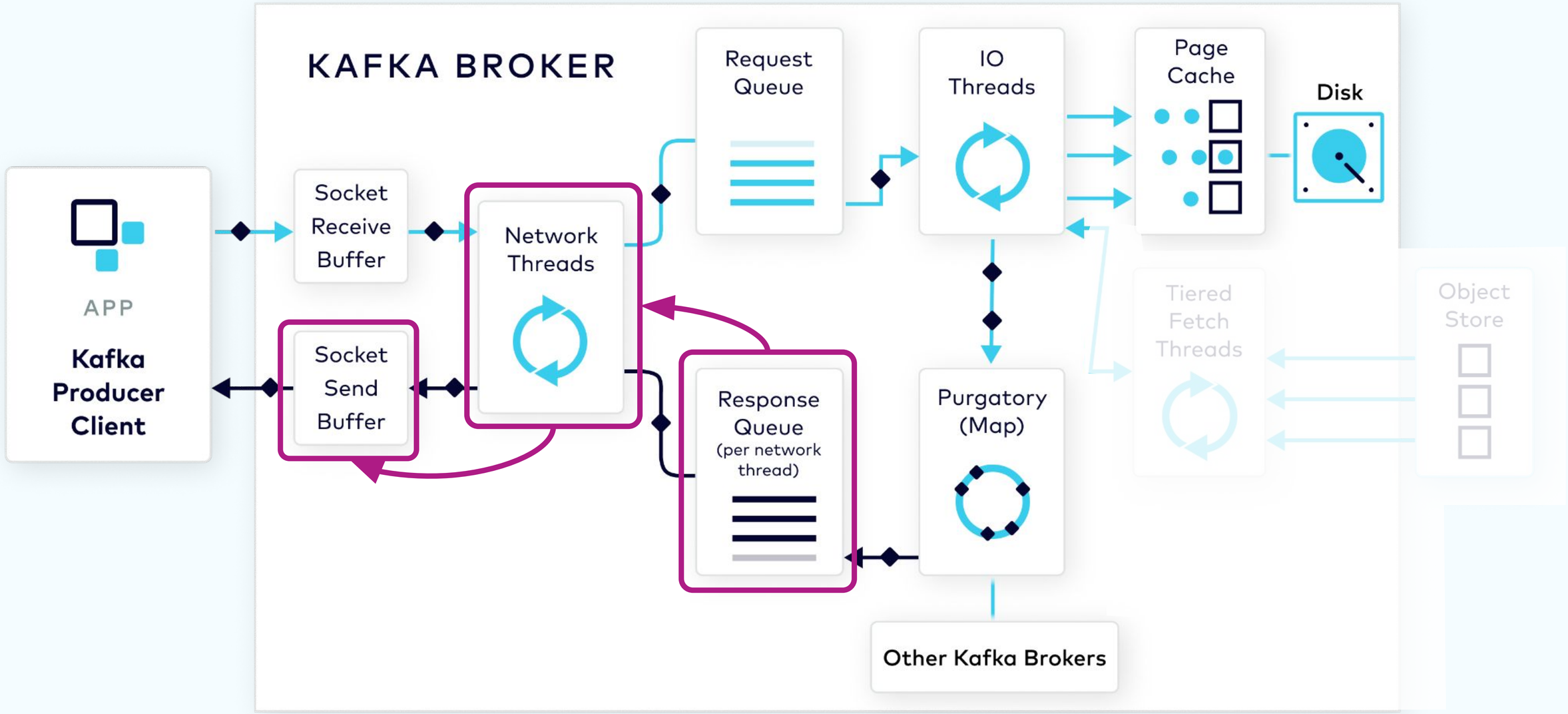
Purgatory Holds Requests Being Replicated

CONFLUENT Developer



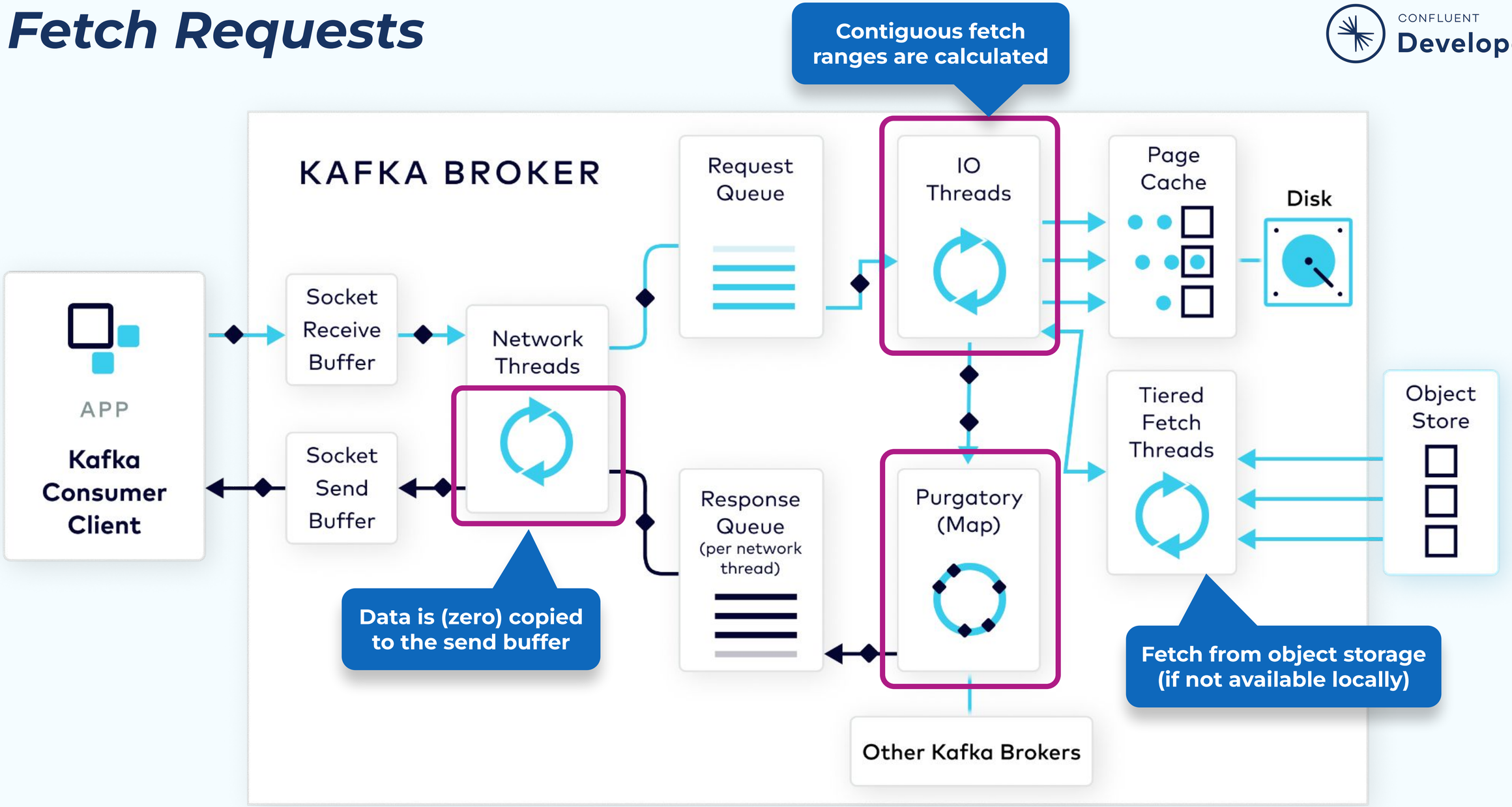
Response Added to Socket Send Buffer

CONFLUENT Developer



Fetch Requests

CONFLUENT Developer

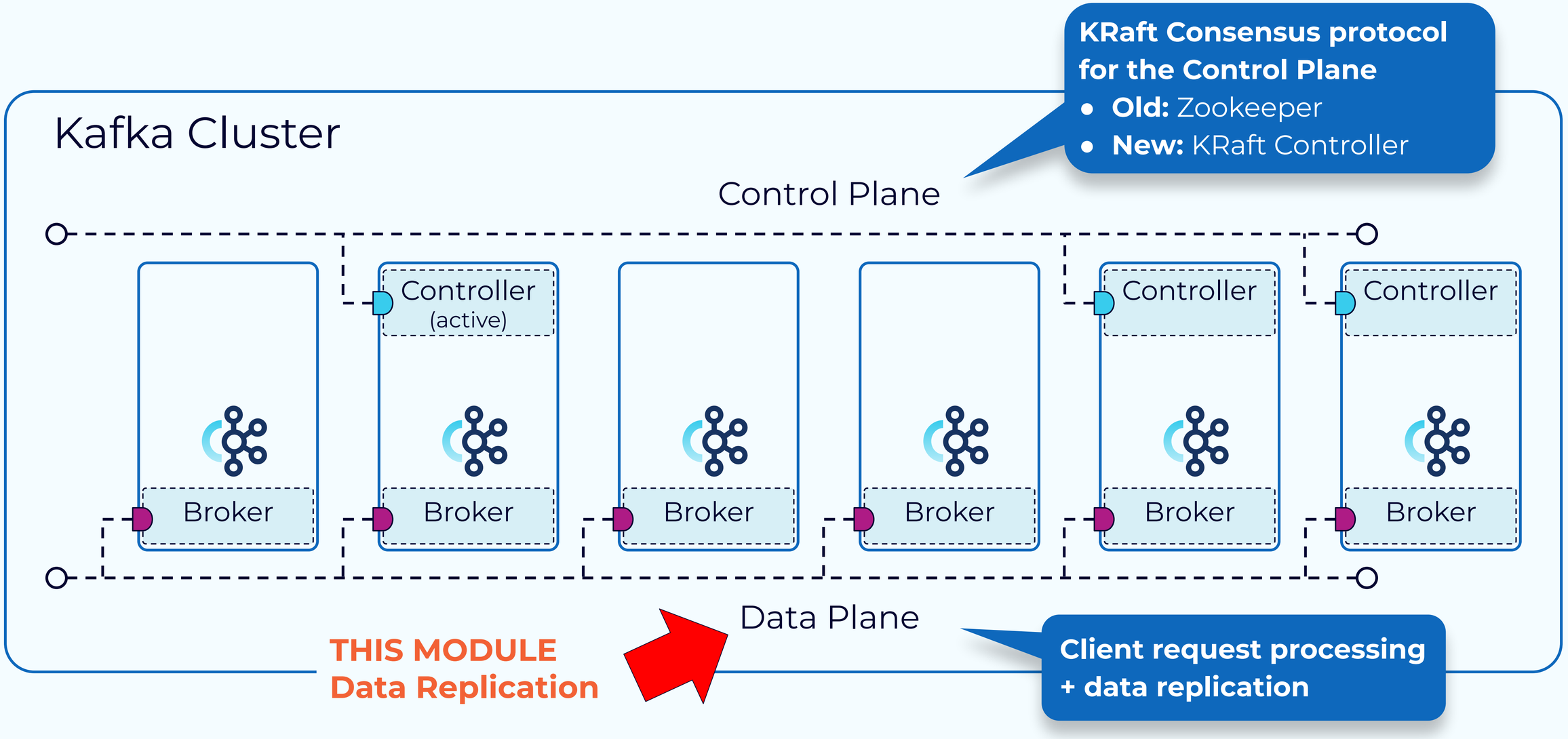


CONFLUENT
Developer

Data Plane: Replication Protocol

Kafka Manages Data & Metadata Separately

CONFLUENT Developer

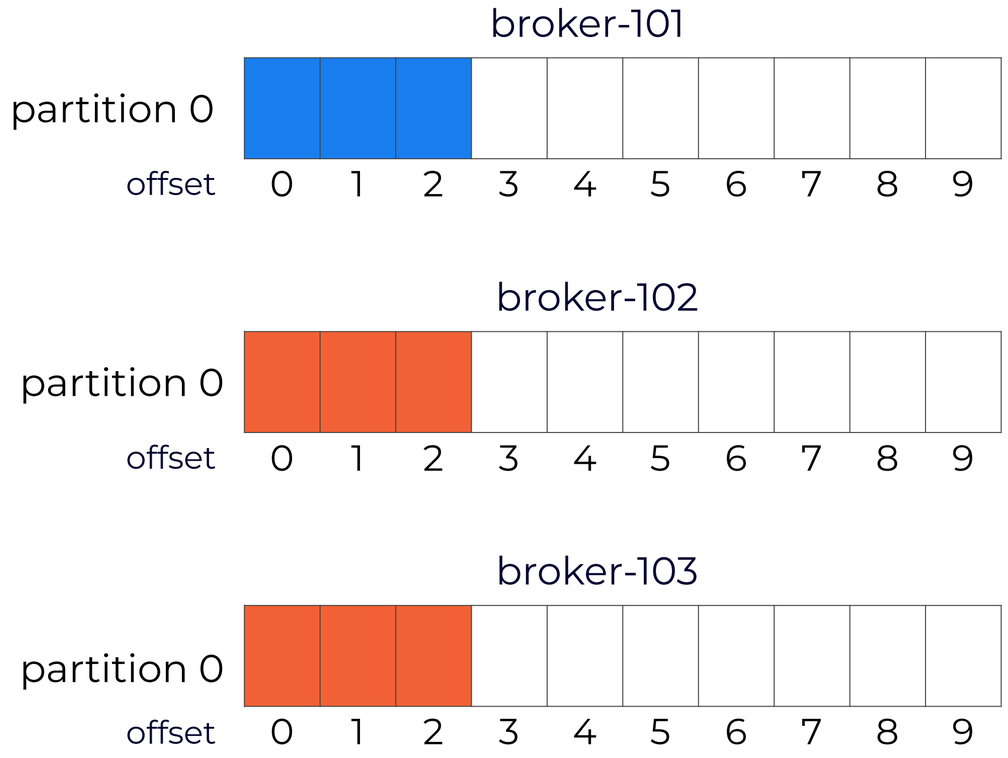


Kafka Data Replication

CONFLUENT Developer

Each topic partition is replicated to multiple brokers

Given n replicas, no data loss with n-1 replica failure



producer

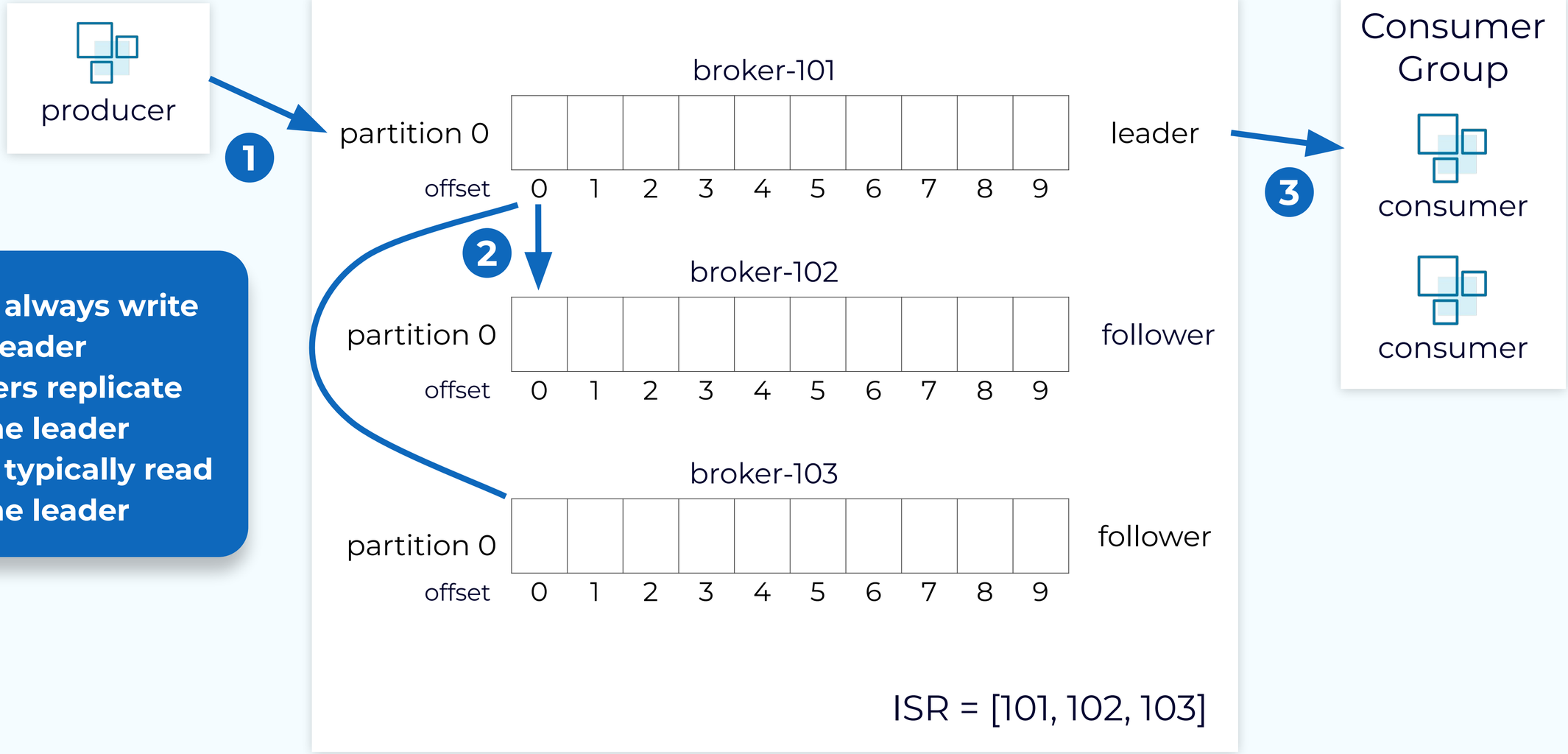
Consumer Group

consumer

consumer

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

CONFLUENT Developer

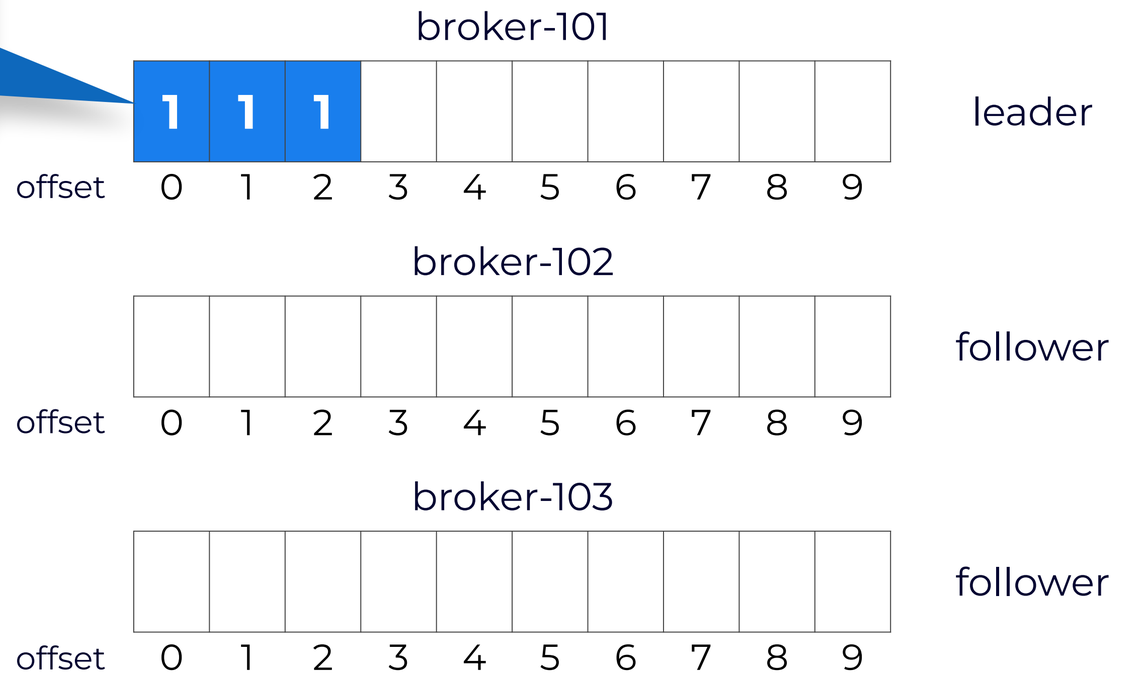


1) Clients always write to the leader
2) Followers replicate from the leader
3) Clients typically read from the leader

Leader Epoch

CONFLUENT Developer

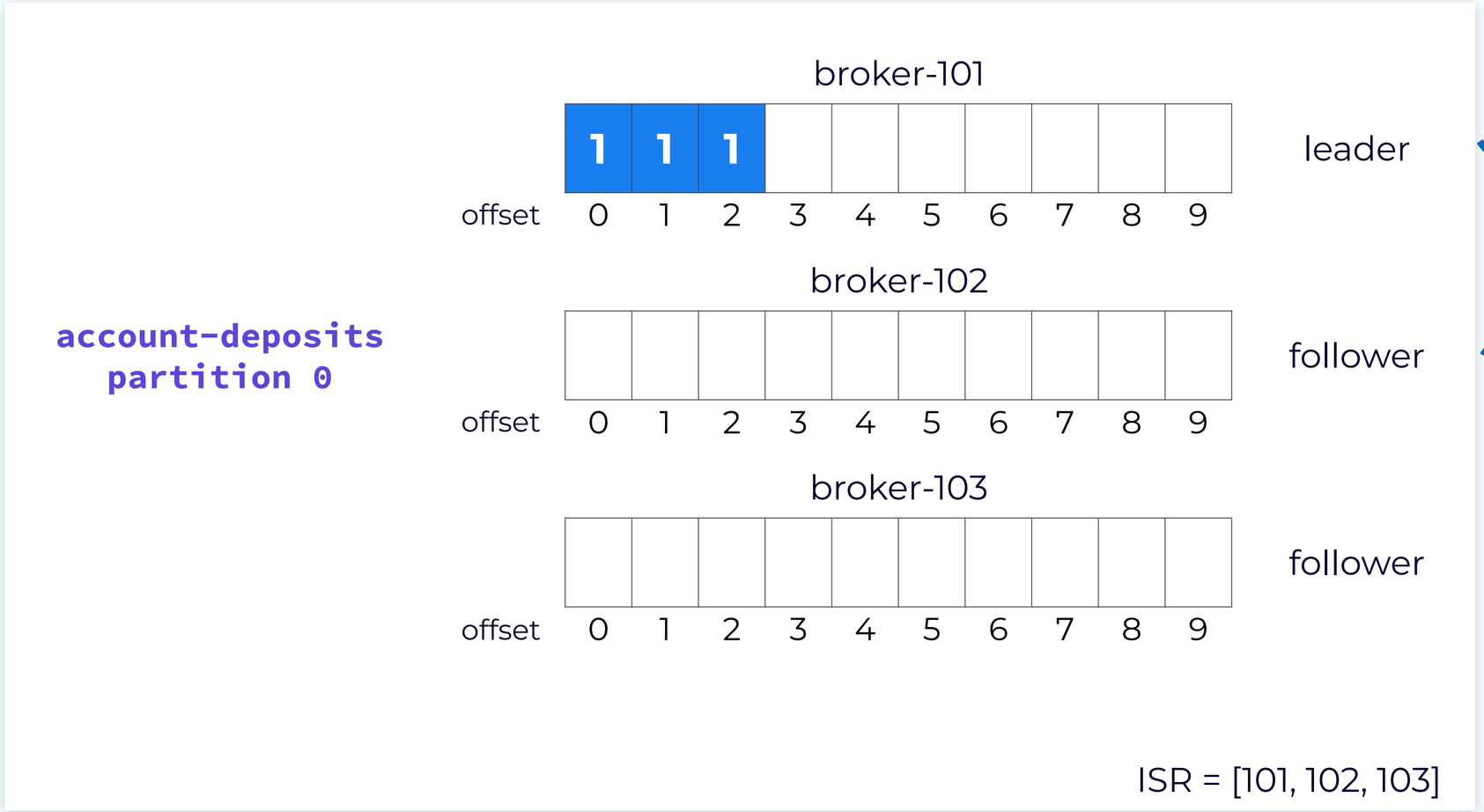
Leader epoch is included for a batch of records



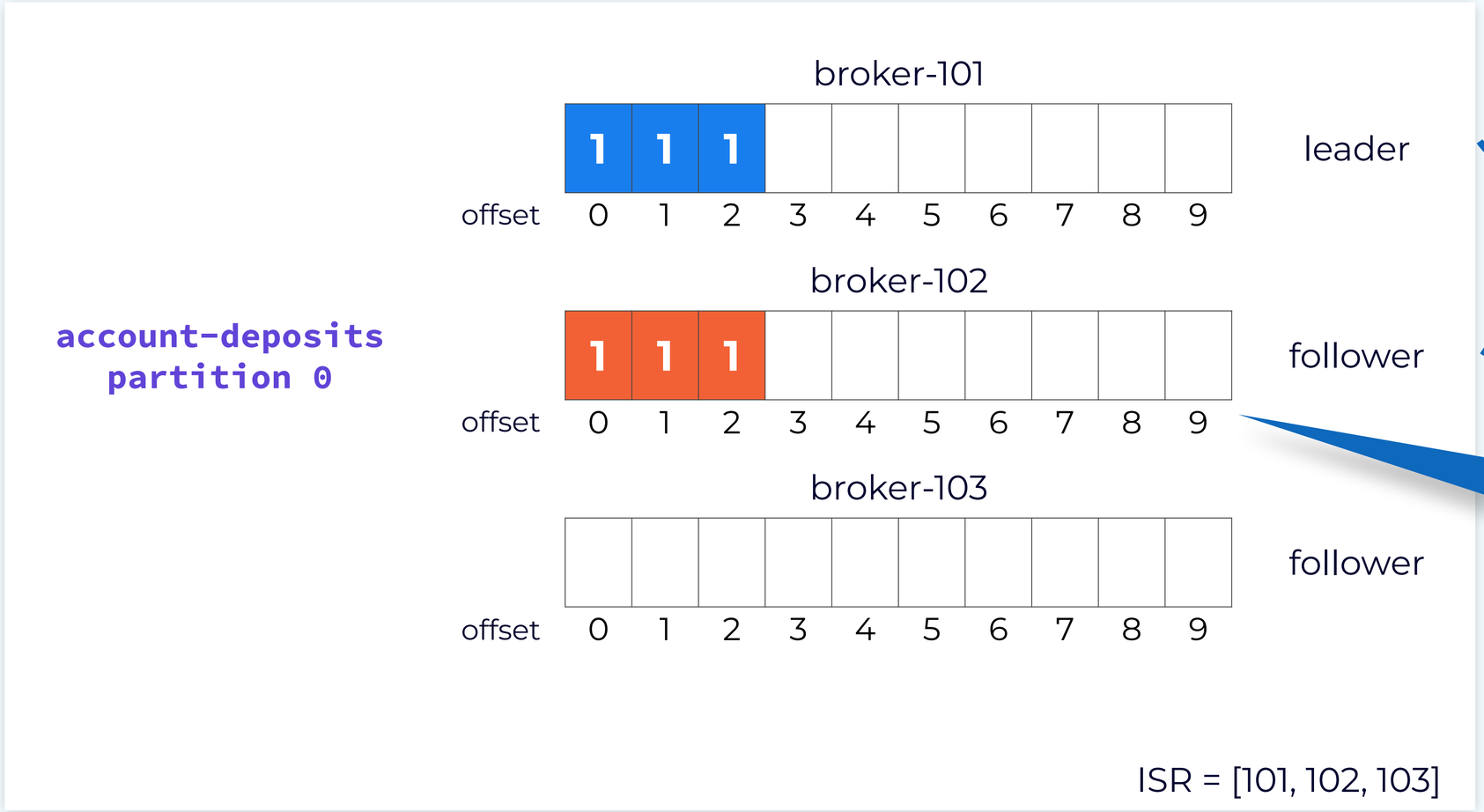
`account-deposits`
`partition 0`

ISR = [101, 102, 103]

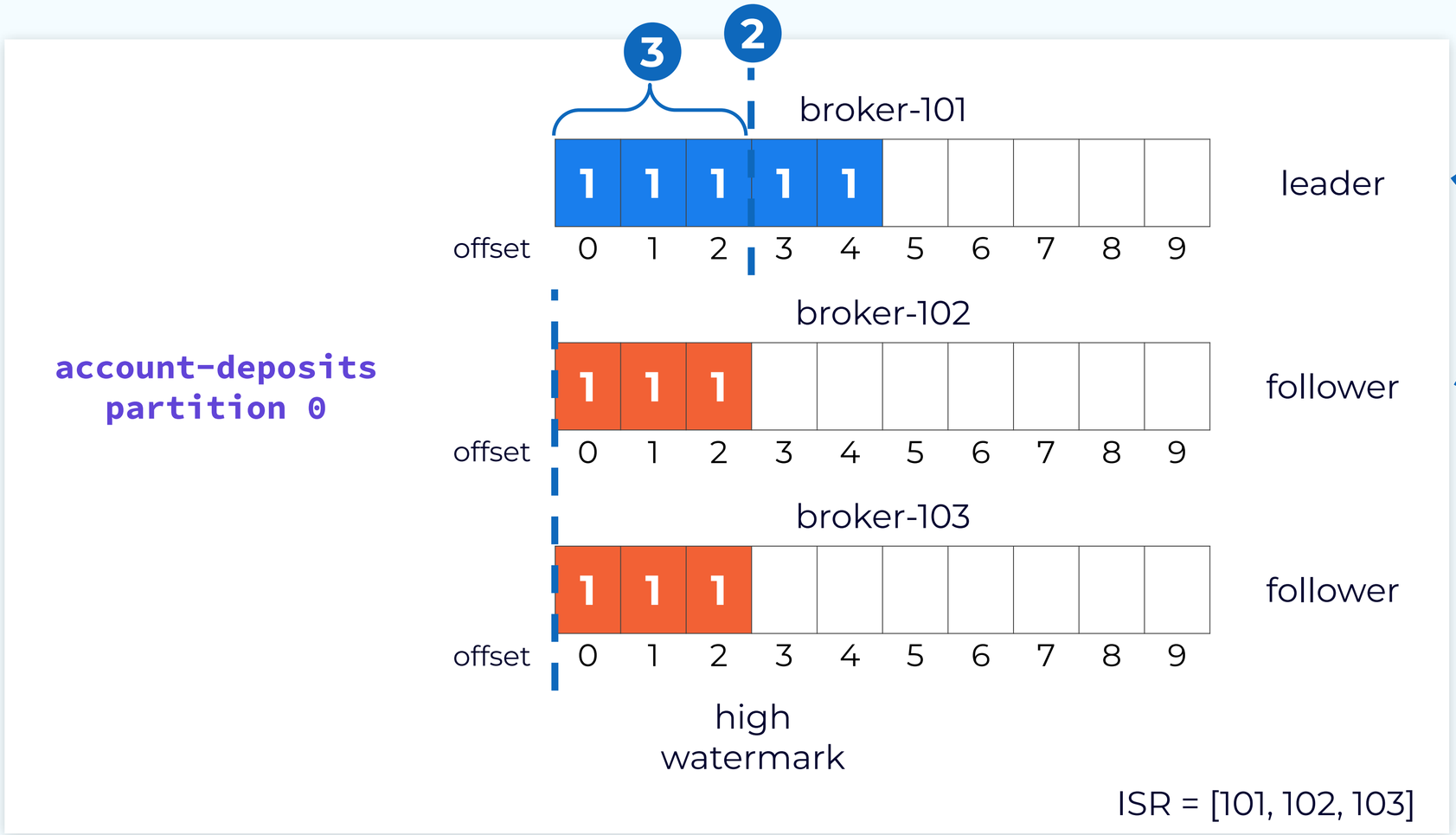
Follower Fetch Request



Follower Fetch Response



Committing Partition Offsets



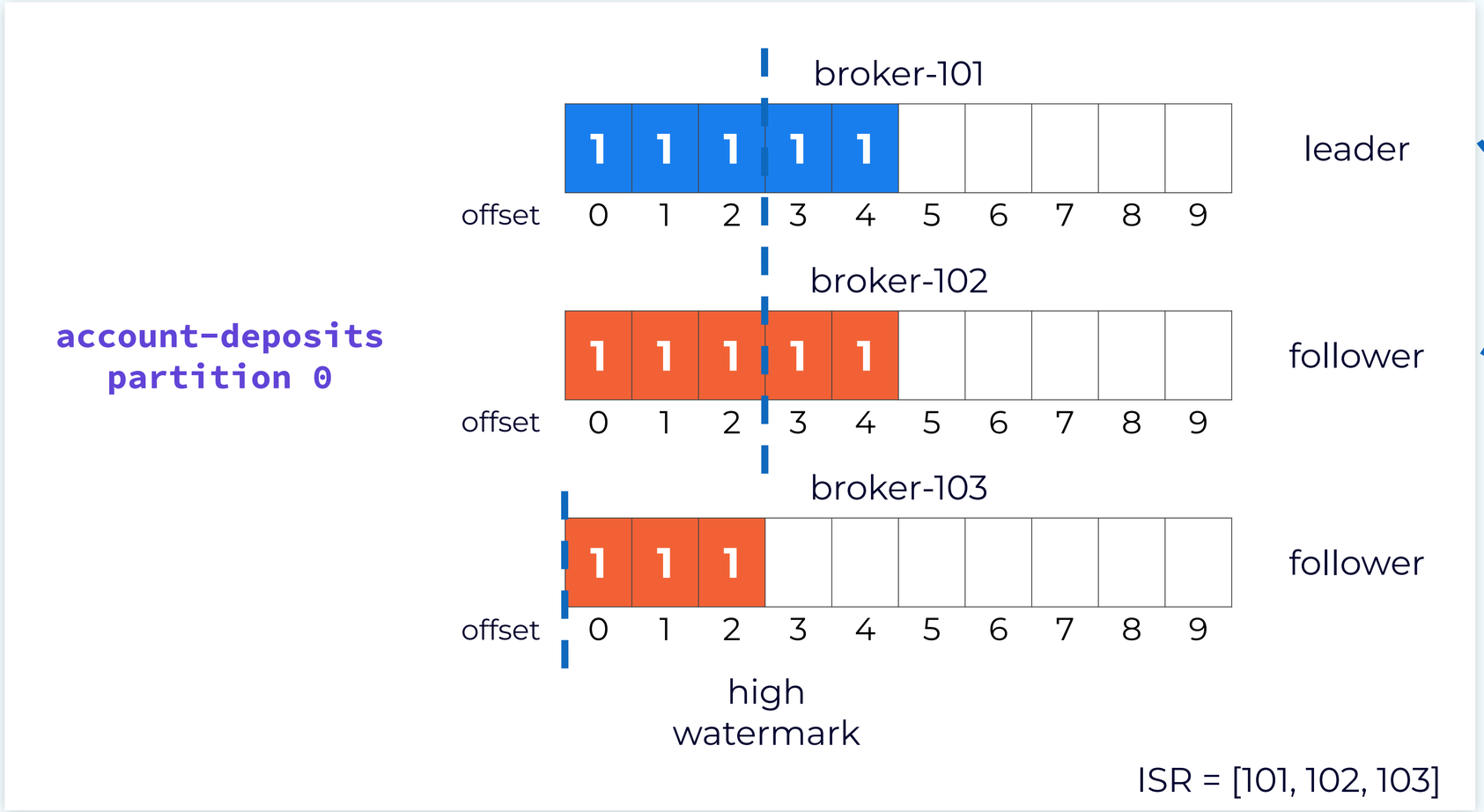
1

`FetchRequest`
`offset: 3`

1) Subsequent fetch request implicitly confirms receipt of previously fetched records
2) Leader commits records once all followers in ISR have confirmed receipt
3) Only committed records are exposed to consumers

Advancing the Follower High Watermark

CONFLUENT Developer

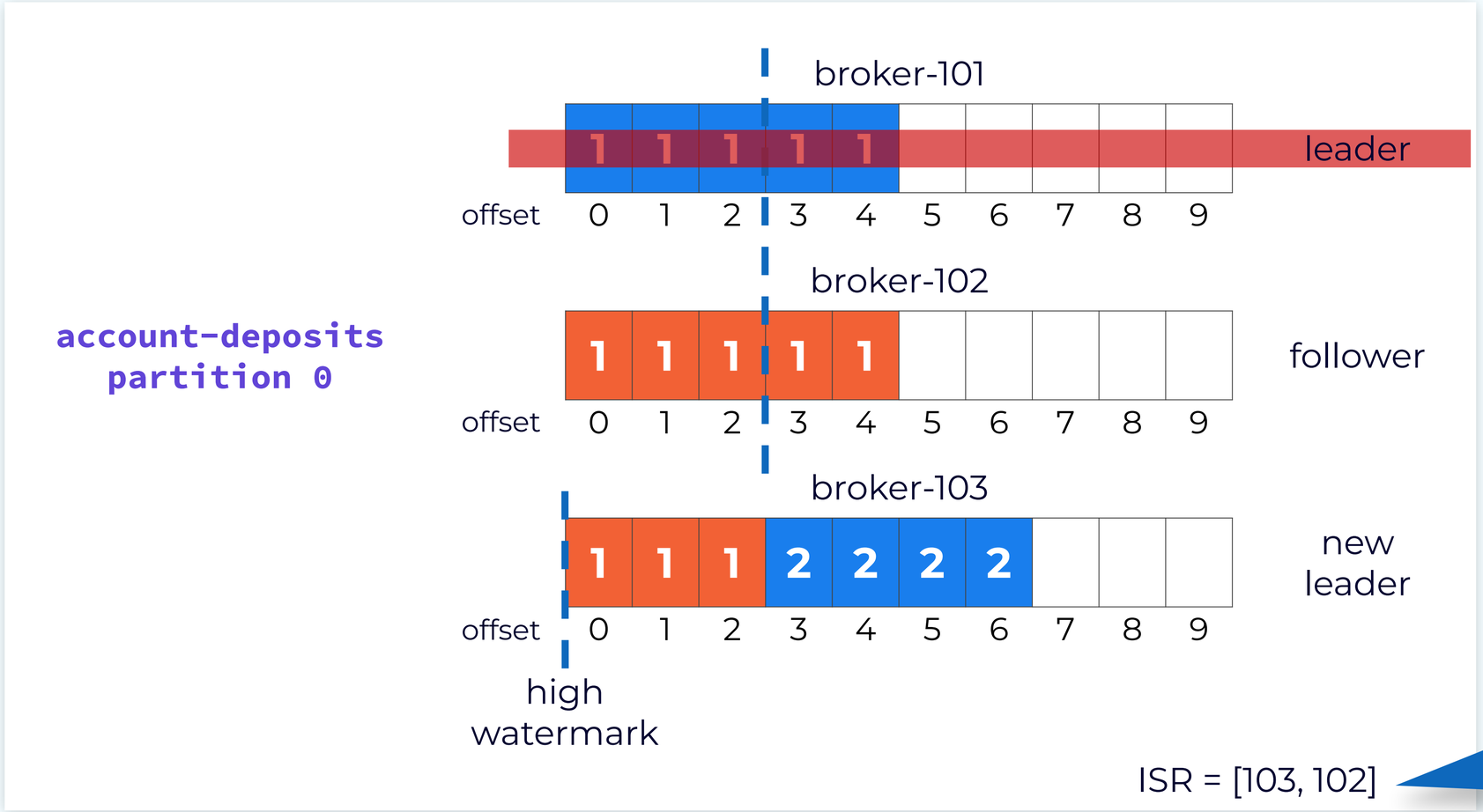



```
FetchResponse
 HighwaterMarkOffset: 3
 records: [
  (offset: 3, epoch: 1),
  (offset: 4, epoch: 1)
 ]
```

Subsequent fetch responses include new high watermark

Handling Leader Failure

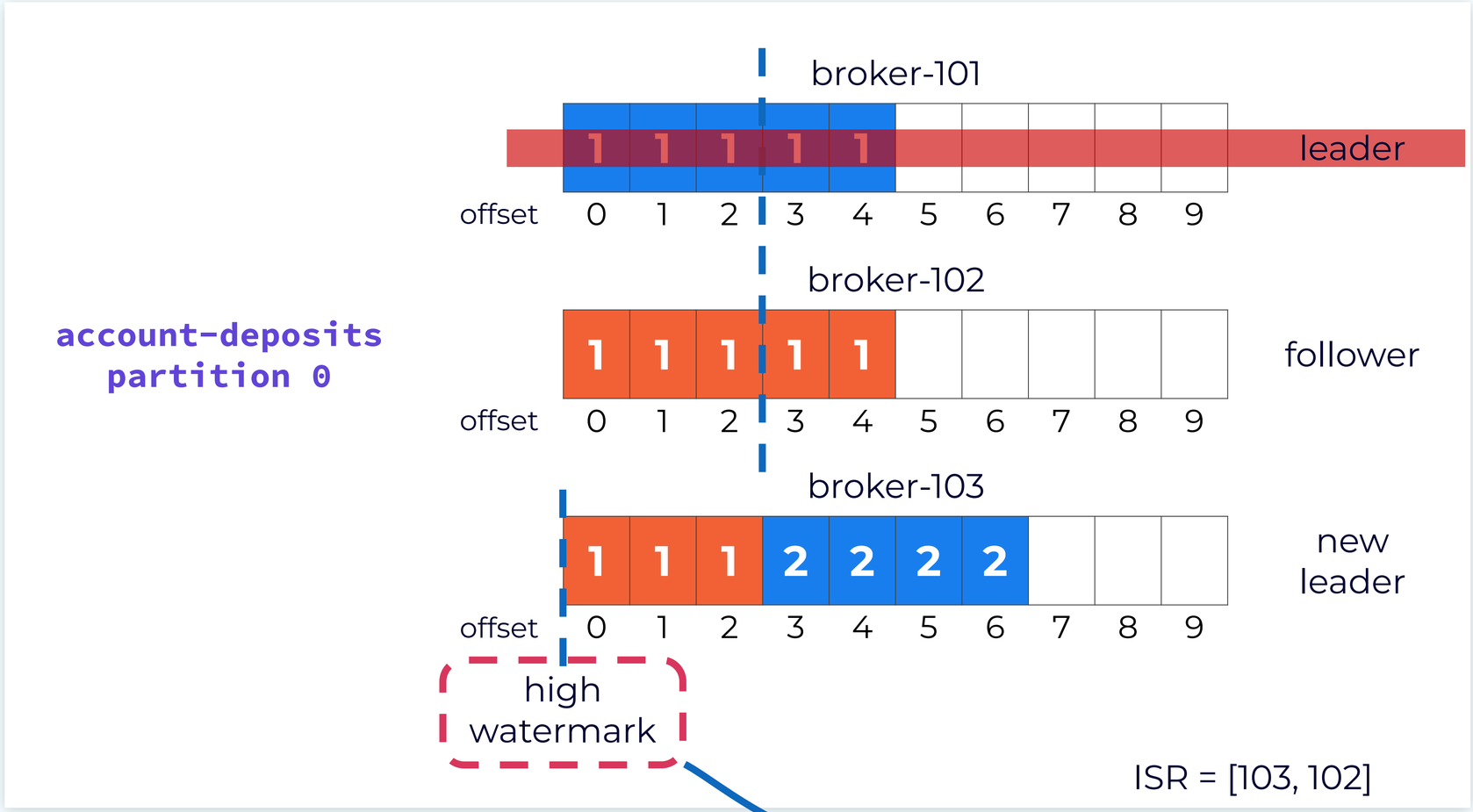
CONFLUENT Developer



New leader elected from ISR and propagated through control plane

Temporary Decreased High Watermark

CONFLUENT Developer

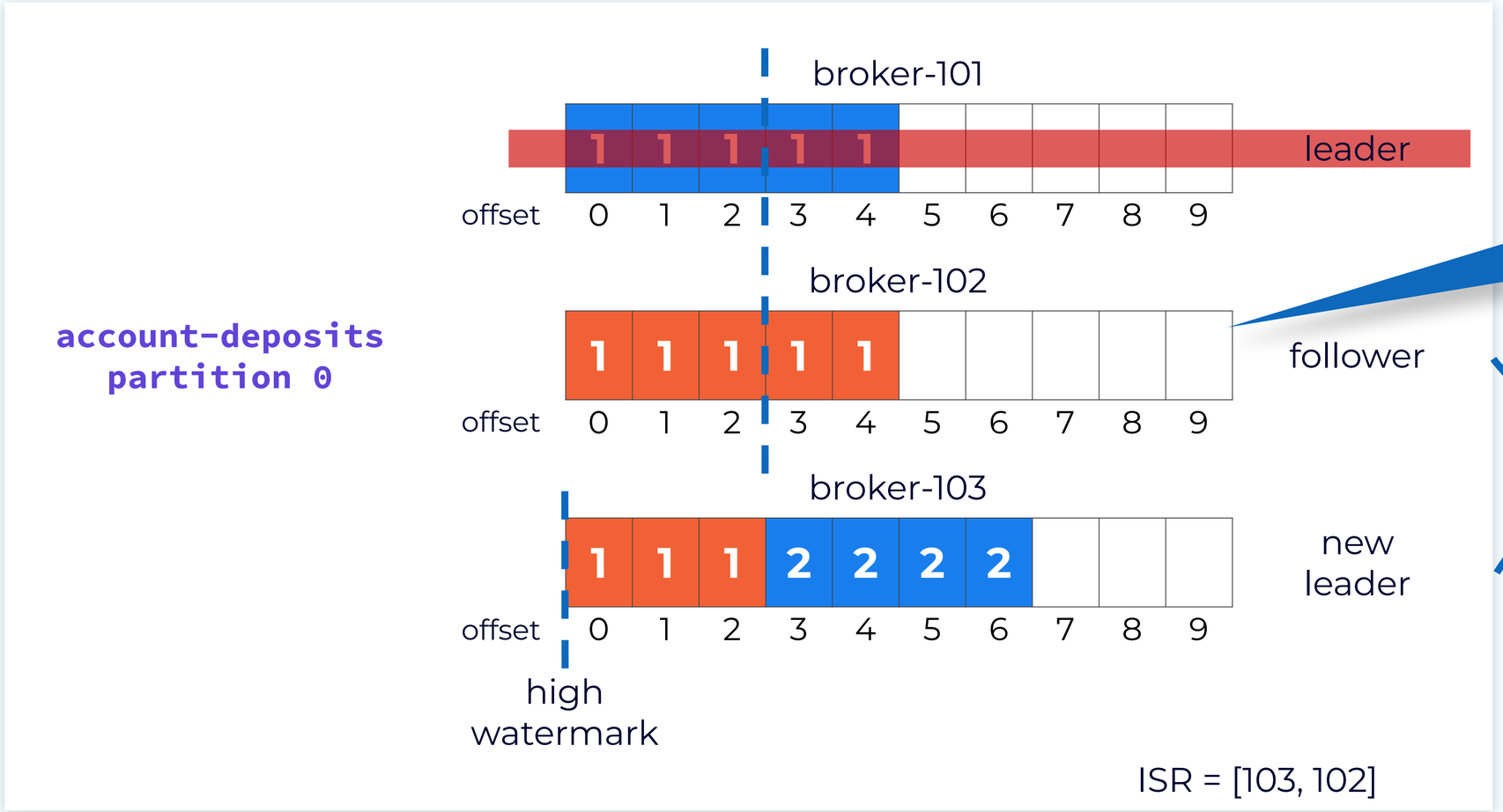


New leader high watermark could be less than true high watermark

- **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

CONFLUENT Developer



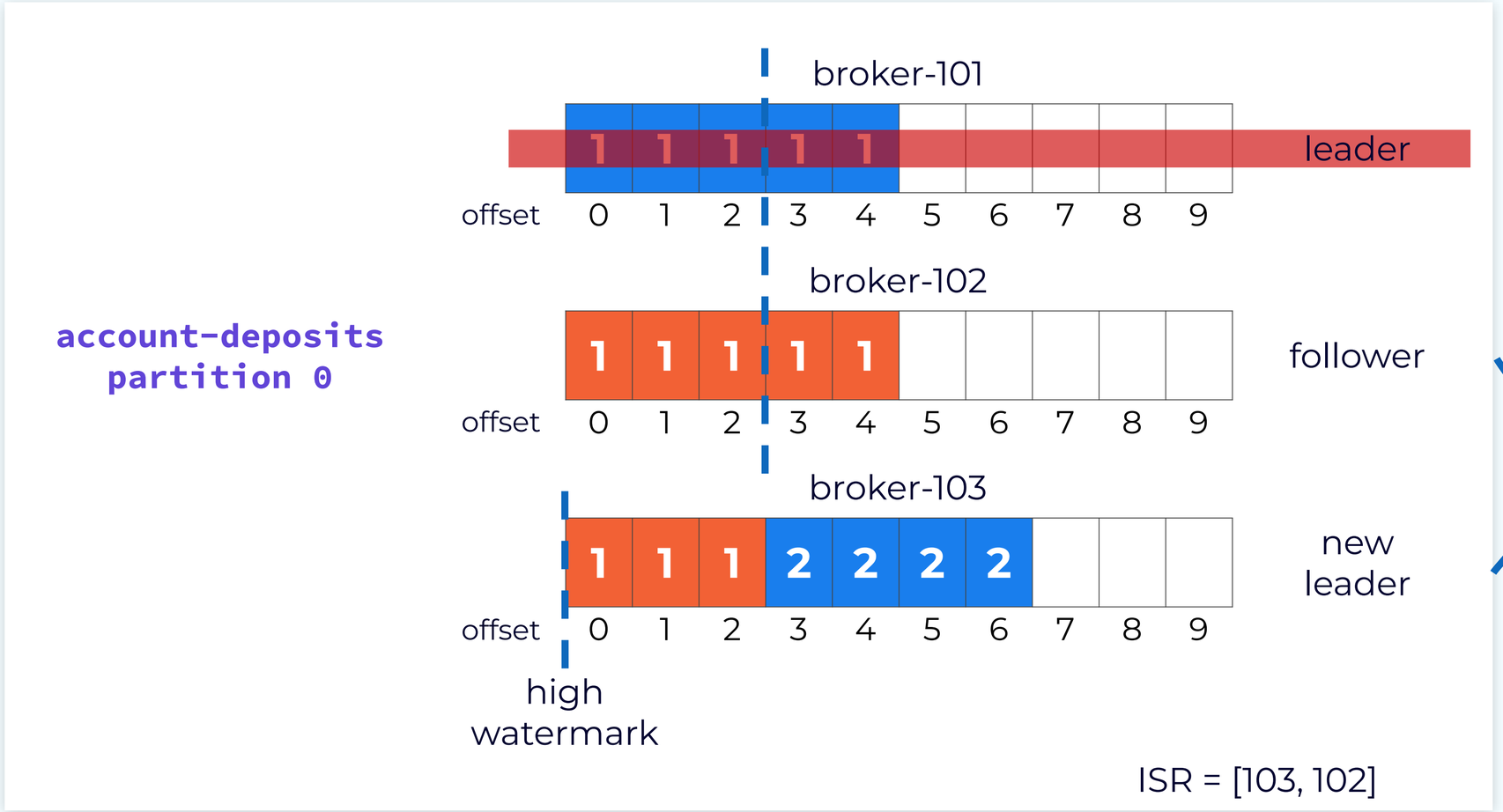
`account-deposits partition 0`

Follower replica out of sync with the new leader

```
FetchRequest
 offset: 5
 lastFetchEpoch: 1
```

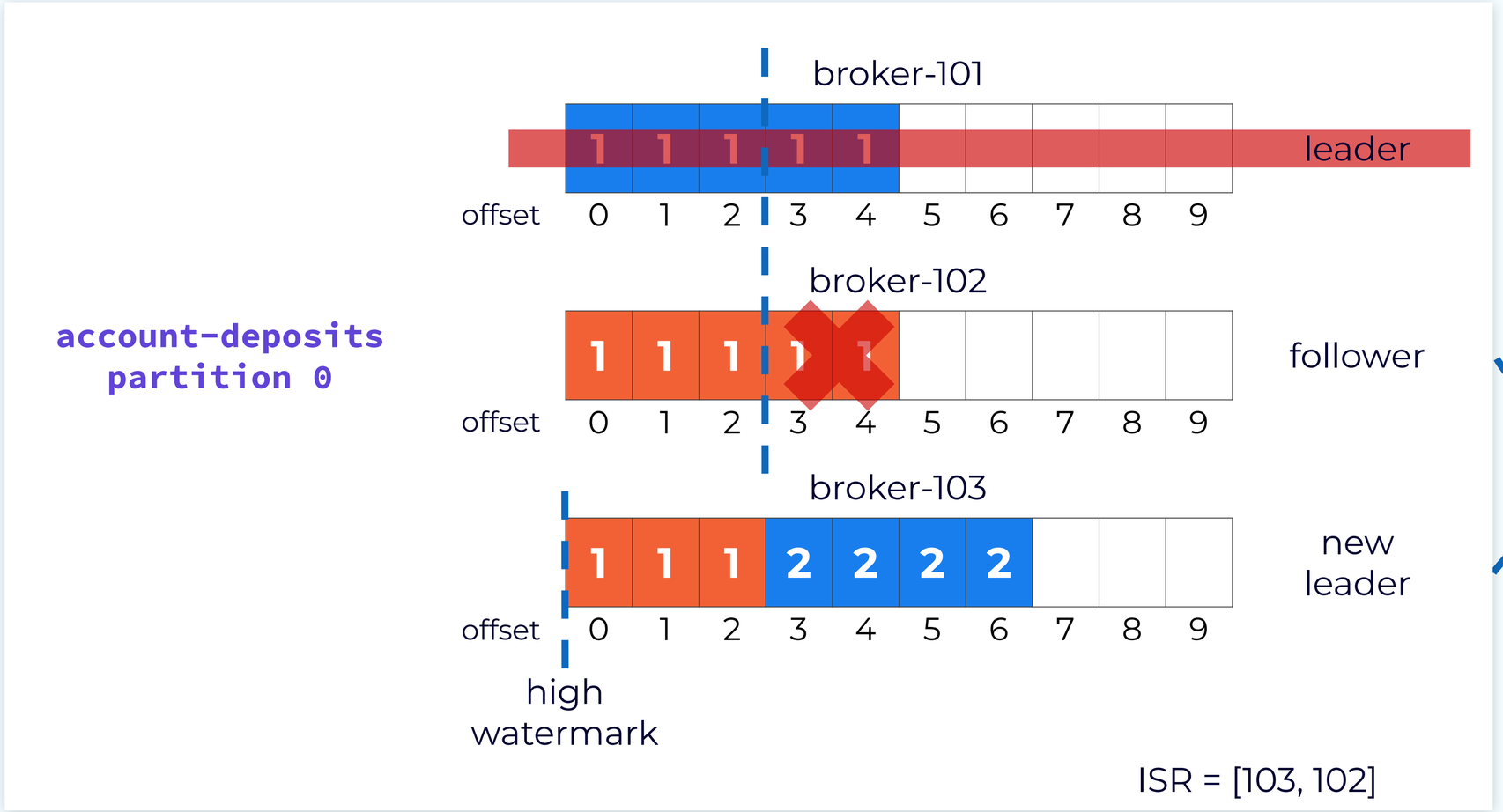
Follower fetch request identifies divergence

Fetch Response Informs Follower of Divergence



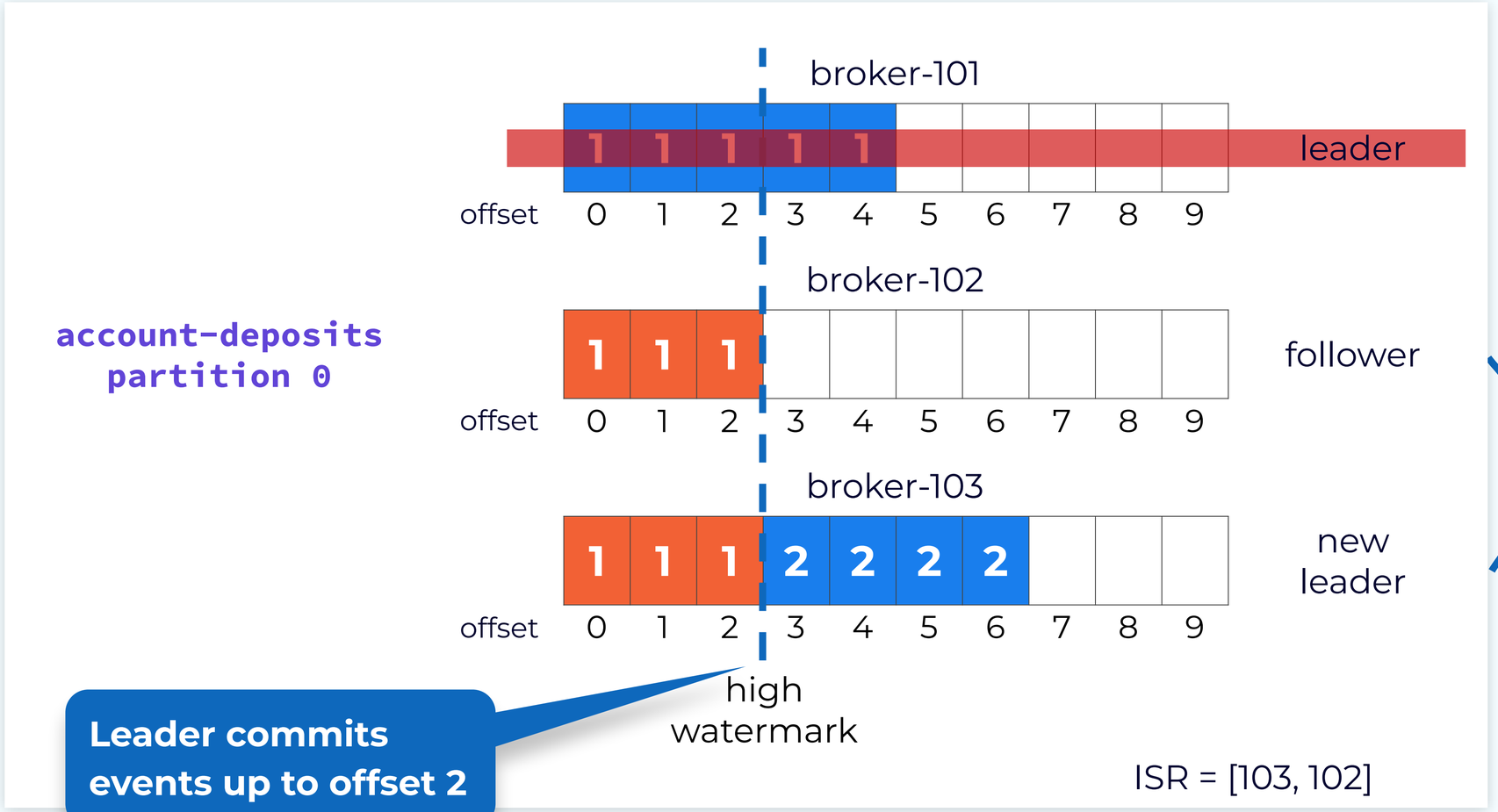

```
FetchResponse
 HighwaterMarkOffset: 0
 divergingEpoch:
  epoch: 1
  endOffset: 3
```

Follower Truncates Log To Match Leader Log

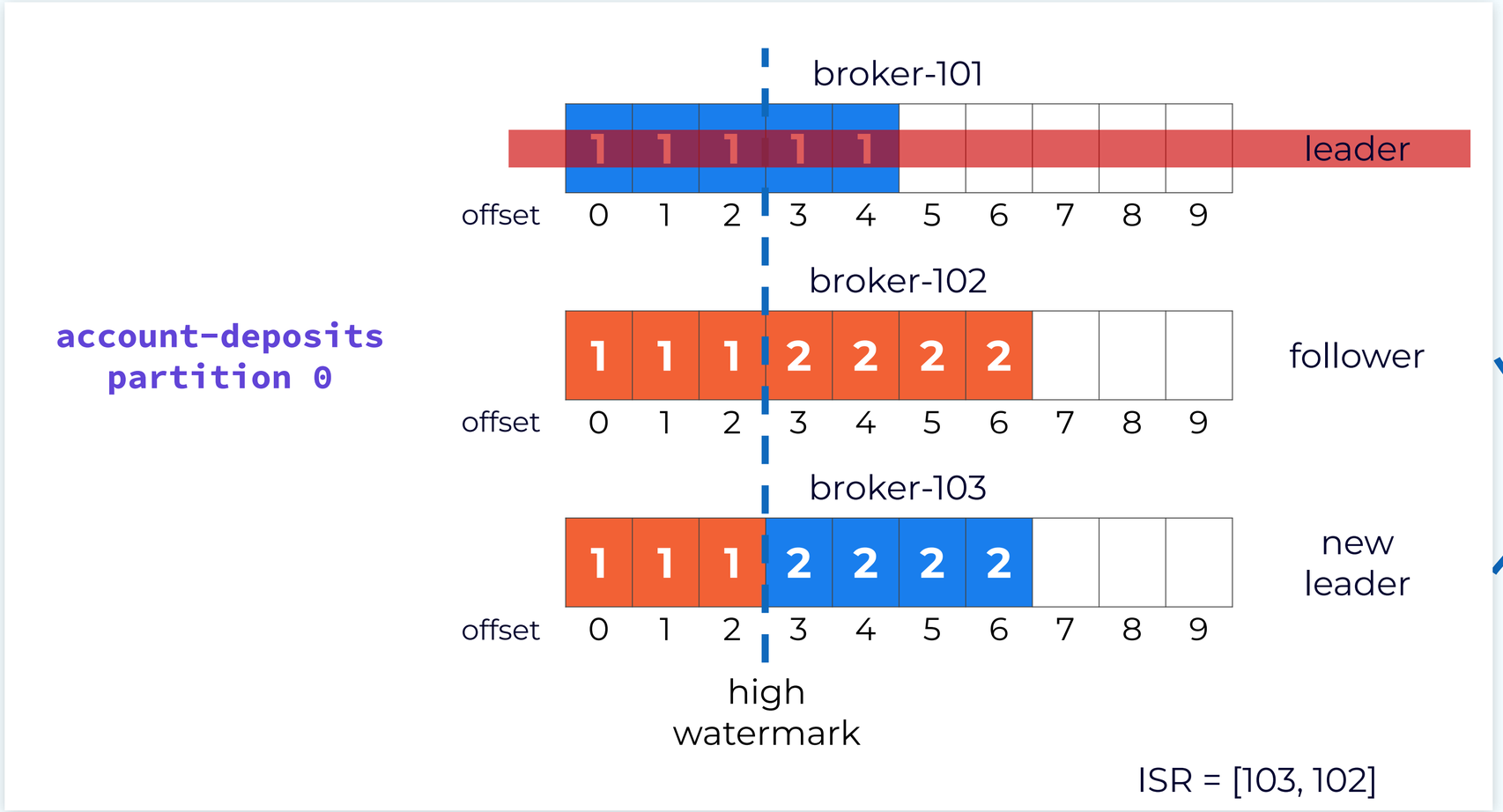


Subsequent Fetch with Updated Offset & Epoch

CONFLUENT Developer



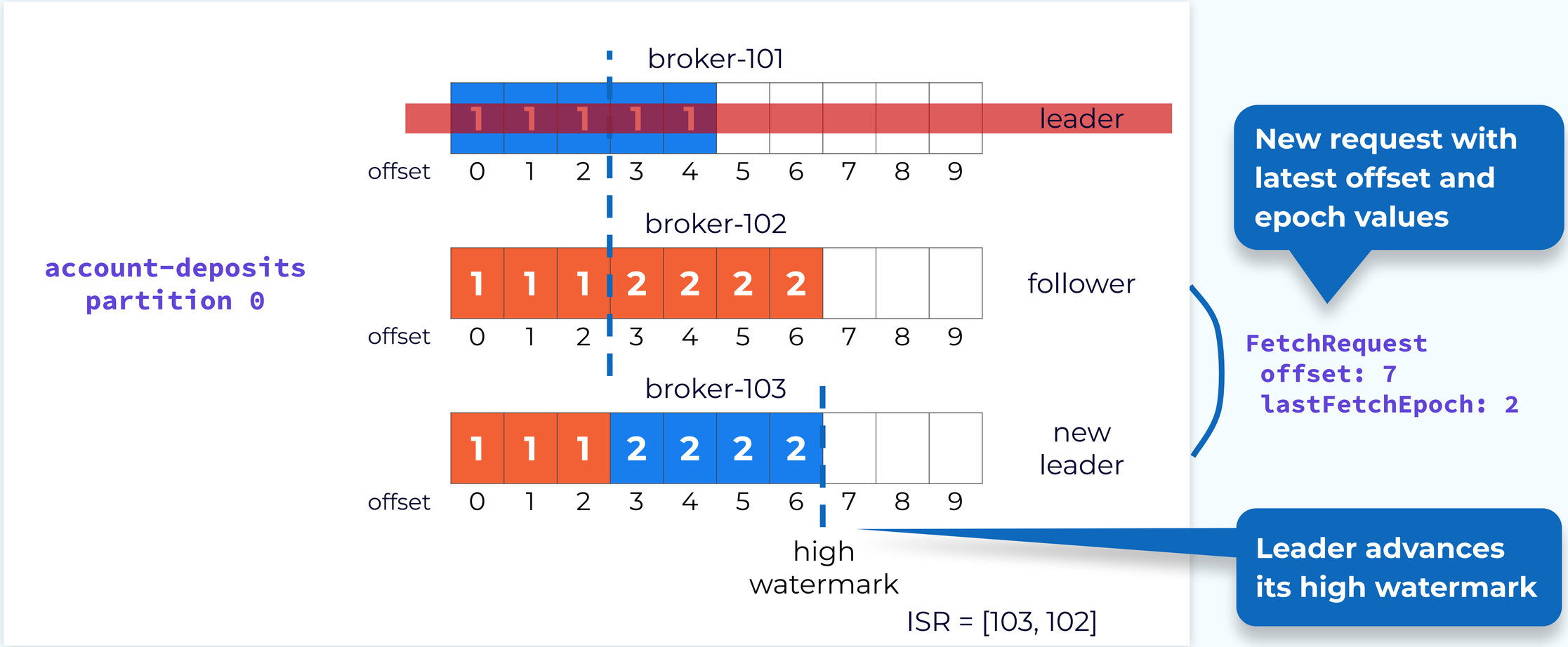
Follower 102 Reconciled



Leader sends requested offsets

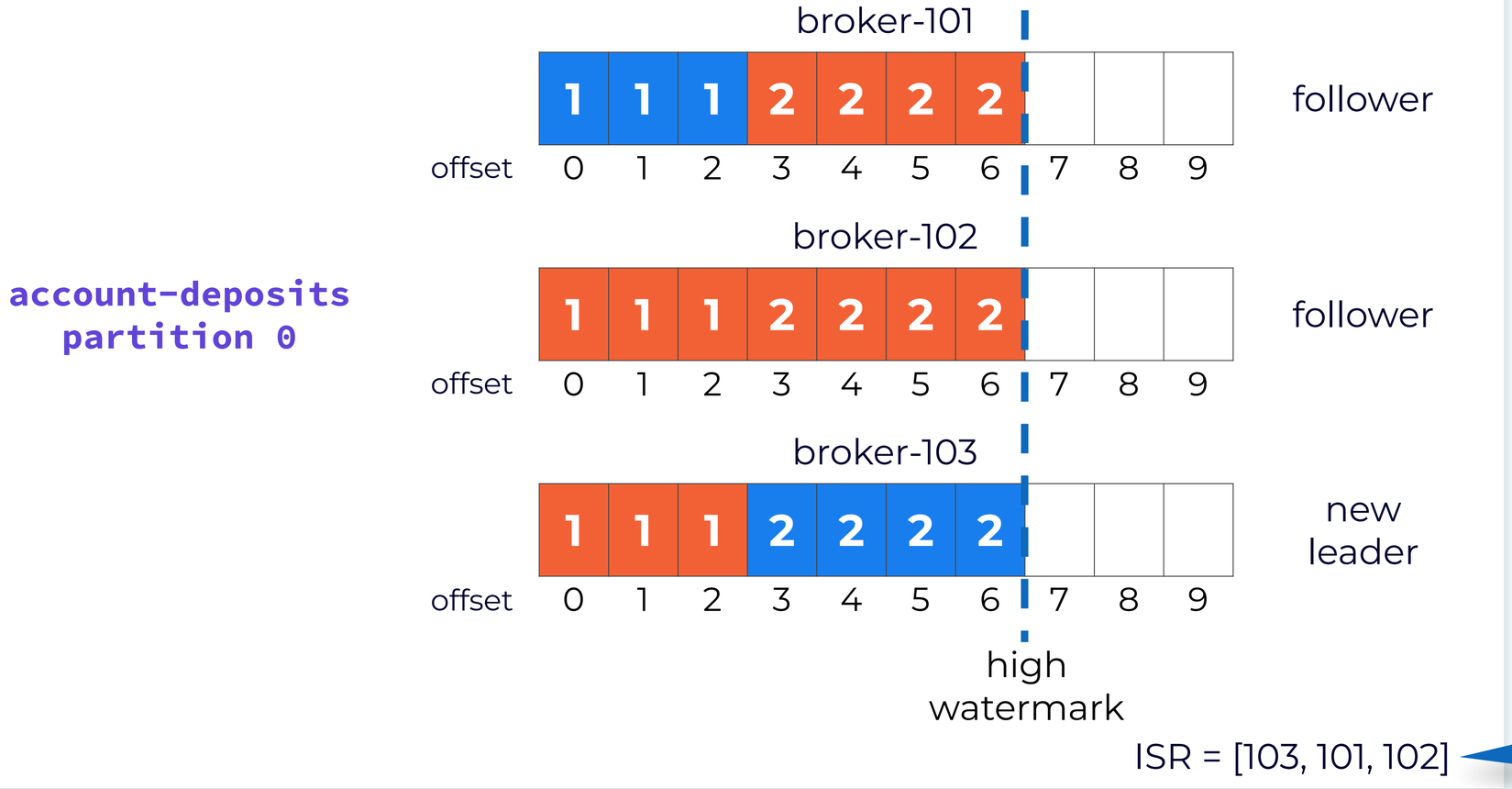
```
FetchResponse
 HighwaterMarkOffset: 3
 records: [
  (offset: 3, epoch: 2),
  (offset: 4, epoch: 2),
  (offset: 5, epoch: 2),
  (offset: 6, epoch: 2)
 ]
```

Follower 102 Acknowledges New Records

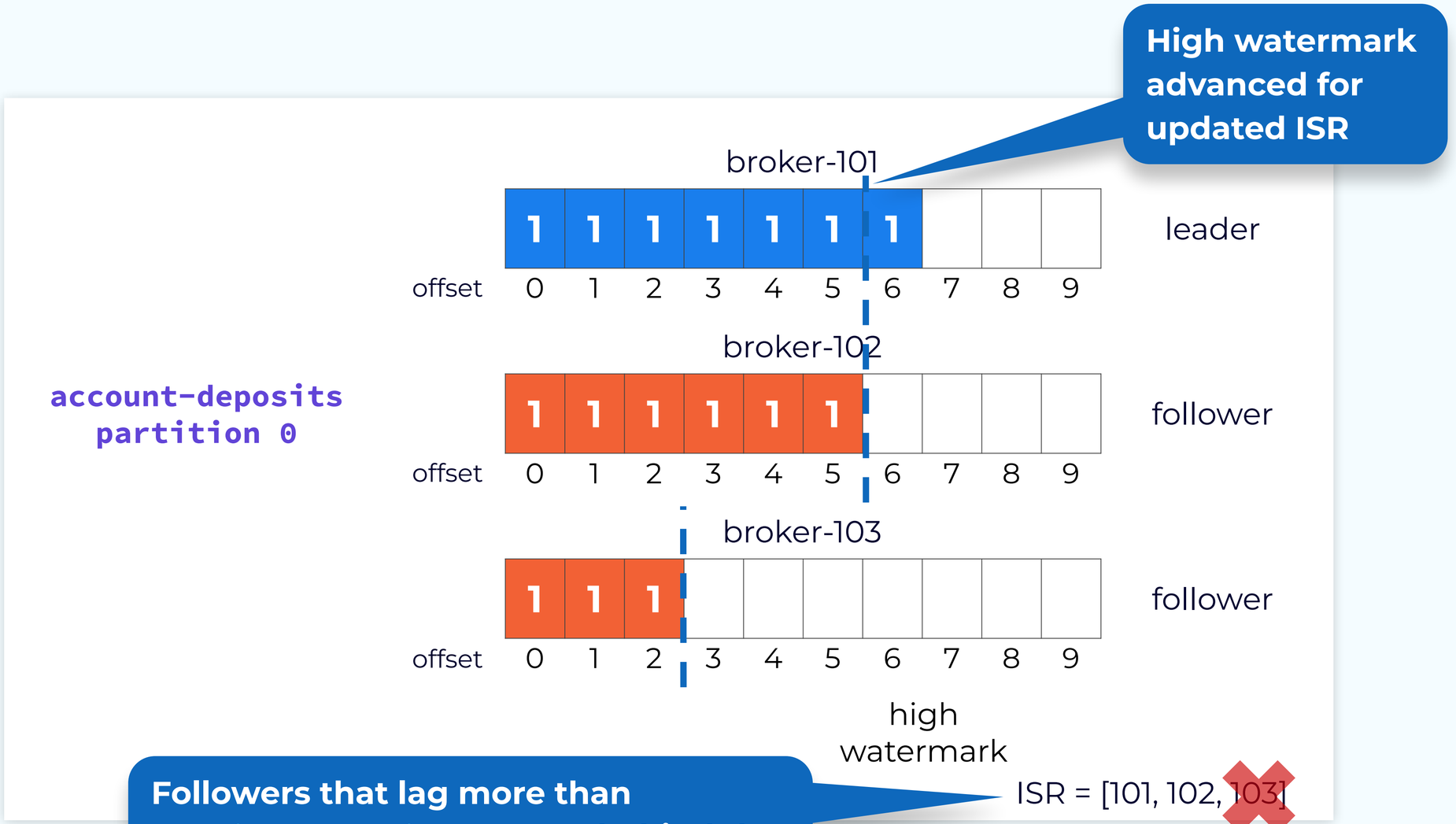


Follower 101 Rejoins the Cluster

CONFLUENT Developer



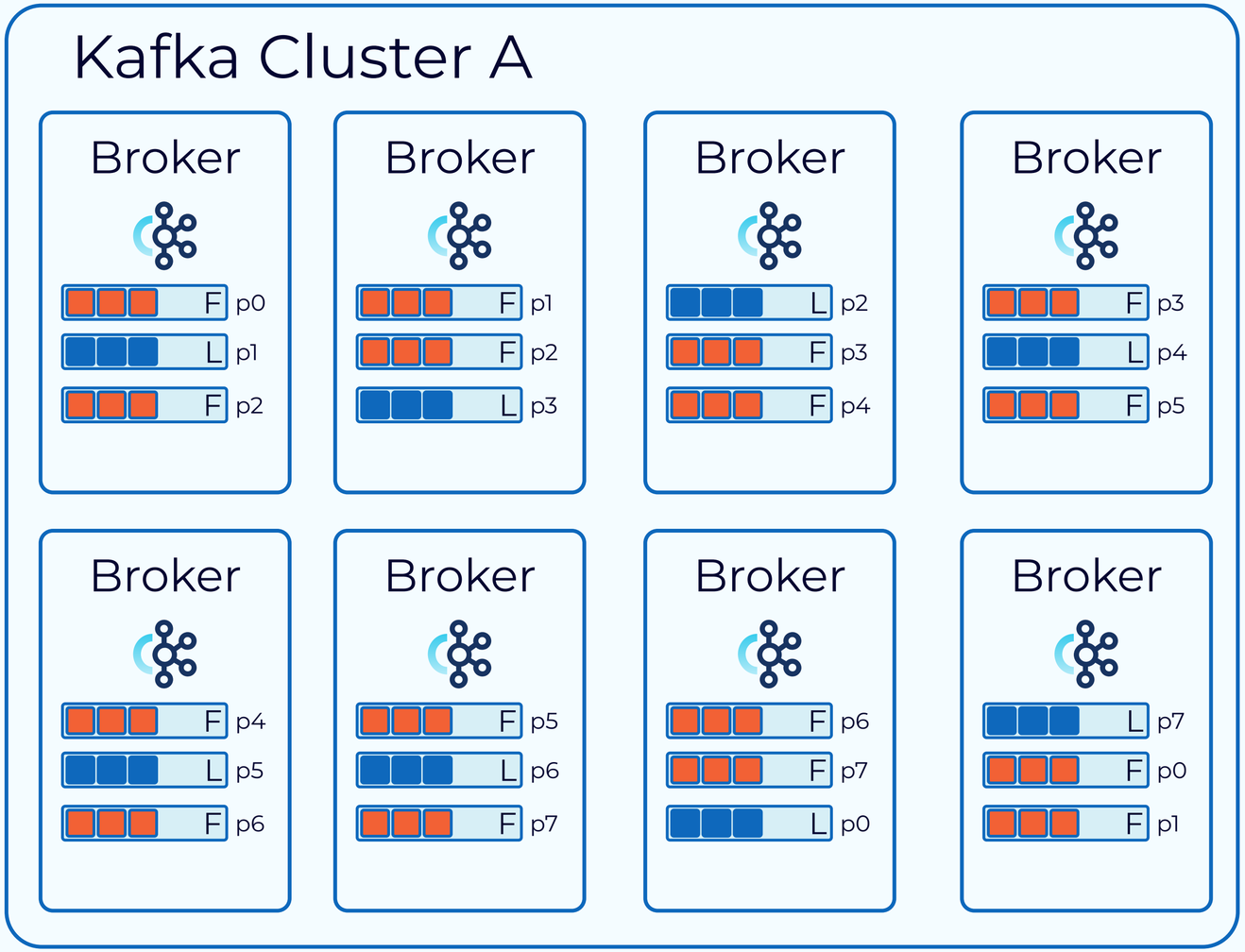
Handling Failed or Slow Followers



Partition Leader Balancing

CONFLUENT Developer

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

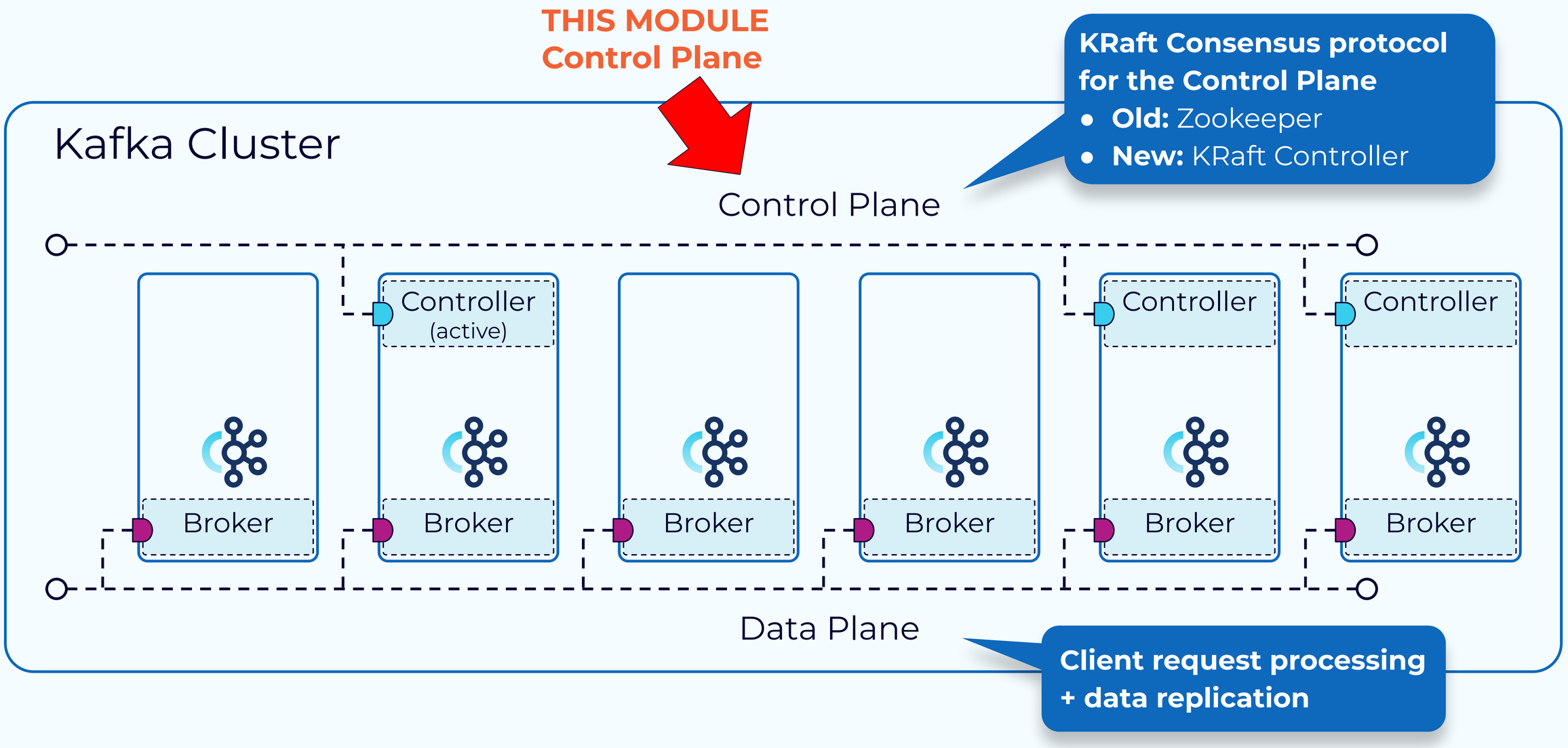


CONFLUENT
Developer

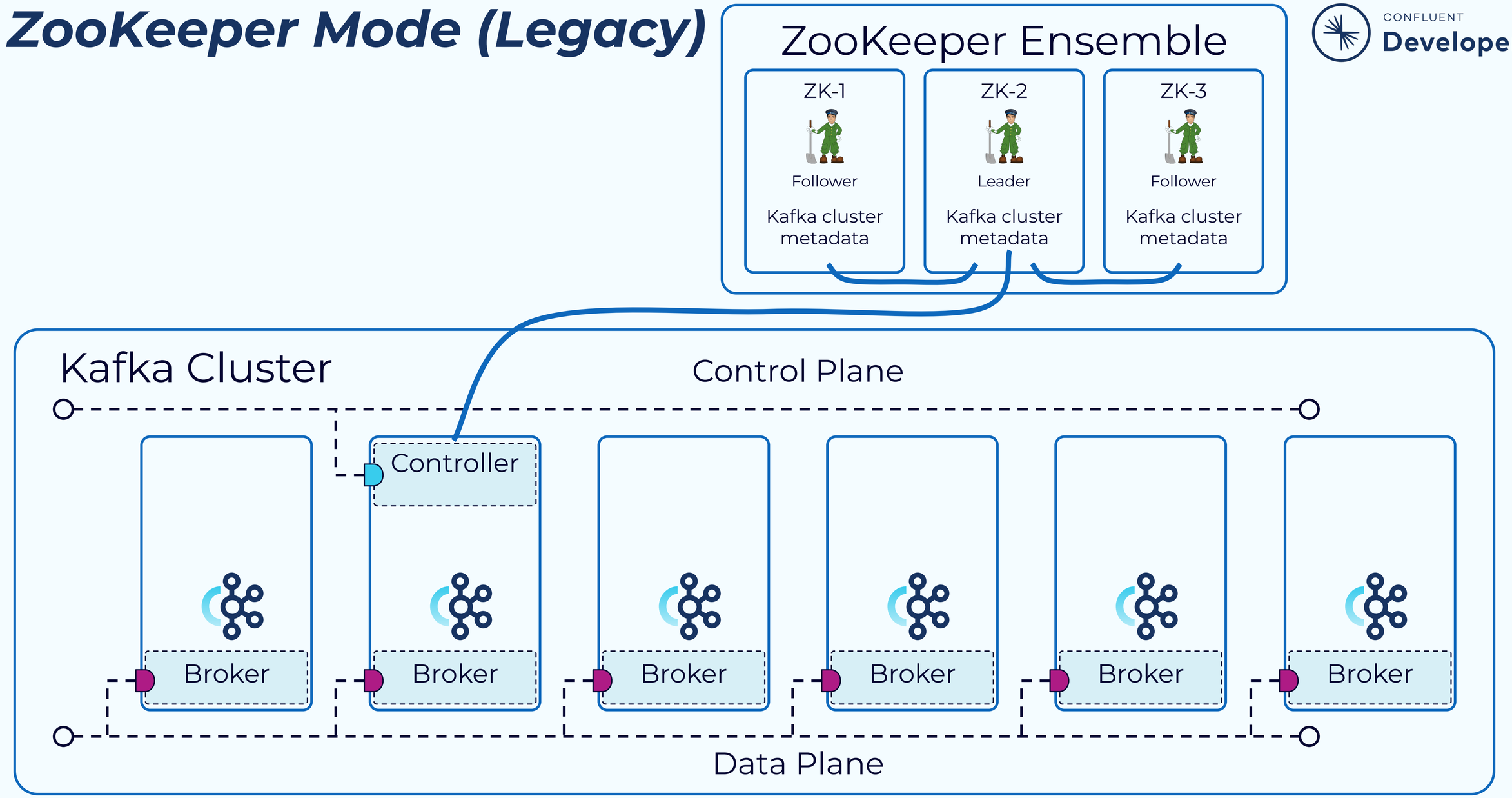
The Apache Kafka® Control Plane

Kafka Manages Data & Metadata Separately

CONFLUENT Developer

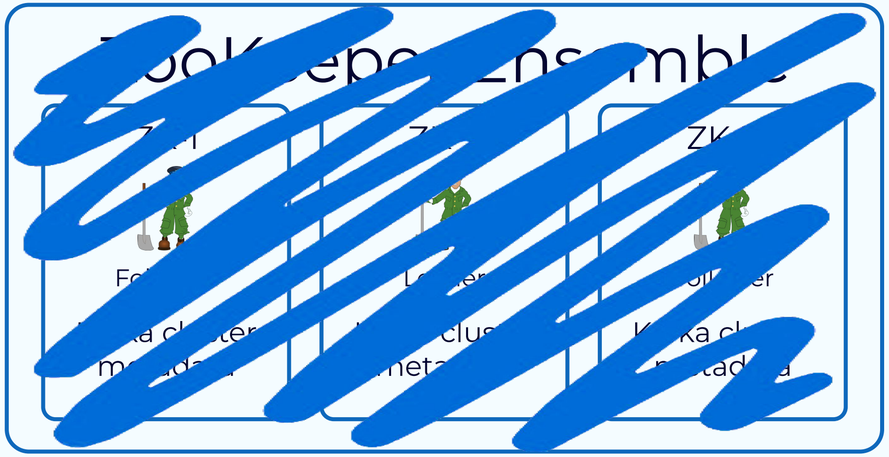


ZooKeeper Mode (Legacy)



CONFLUENT Developer

KRaft Mode (Nascent)



CONFLUENT Developer

Controller pool

Kafka Cluster

Control Plane

Controller (active)

Controller

Controller

Broker

Broker

Broker

Broker

Broker

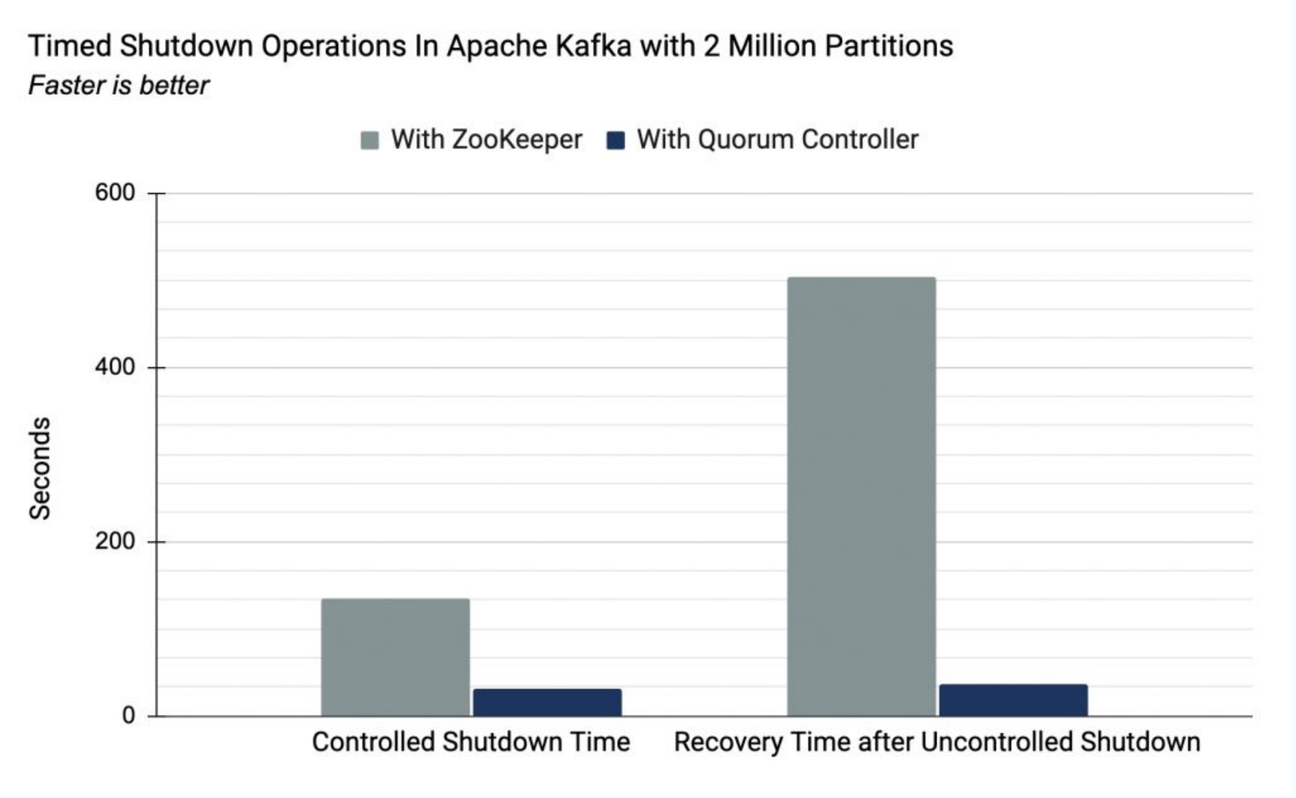
Broker

Data Plane

KRaft Mode Advantages

CONFLUENT Developer

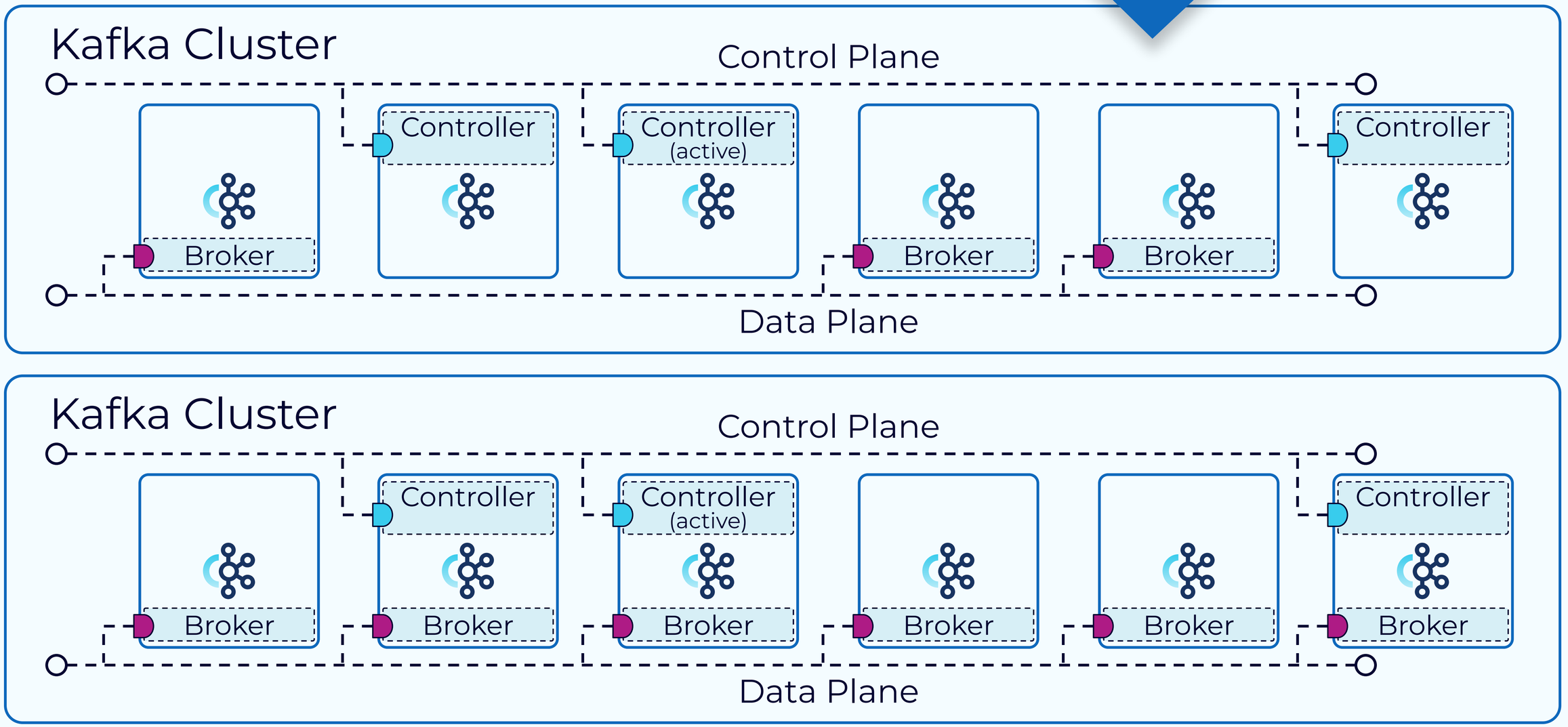
- Fewer moving parts make simpler operation
- Improved cluster scalability due to faster recovery (see chart)
- More efficient, incremental propagation of metadata to brokers



KRaft Cluster Node Roles

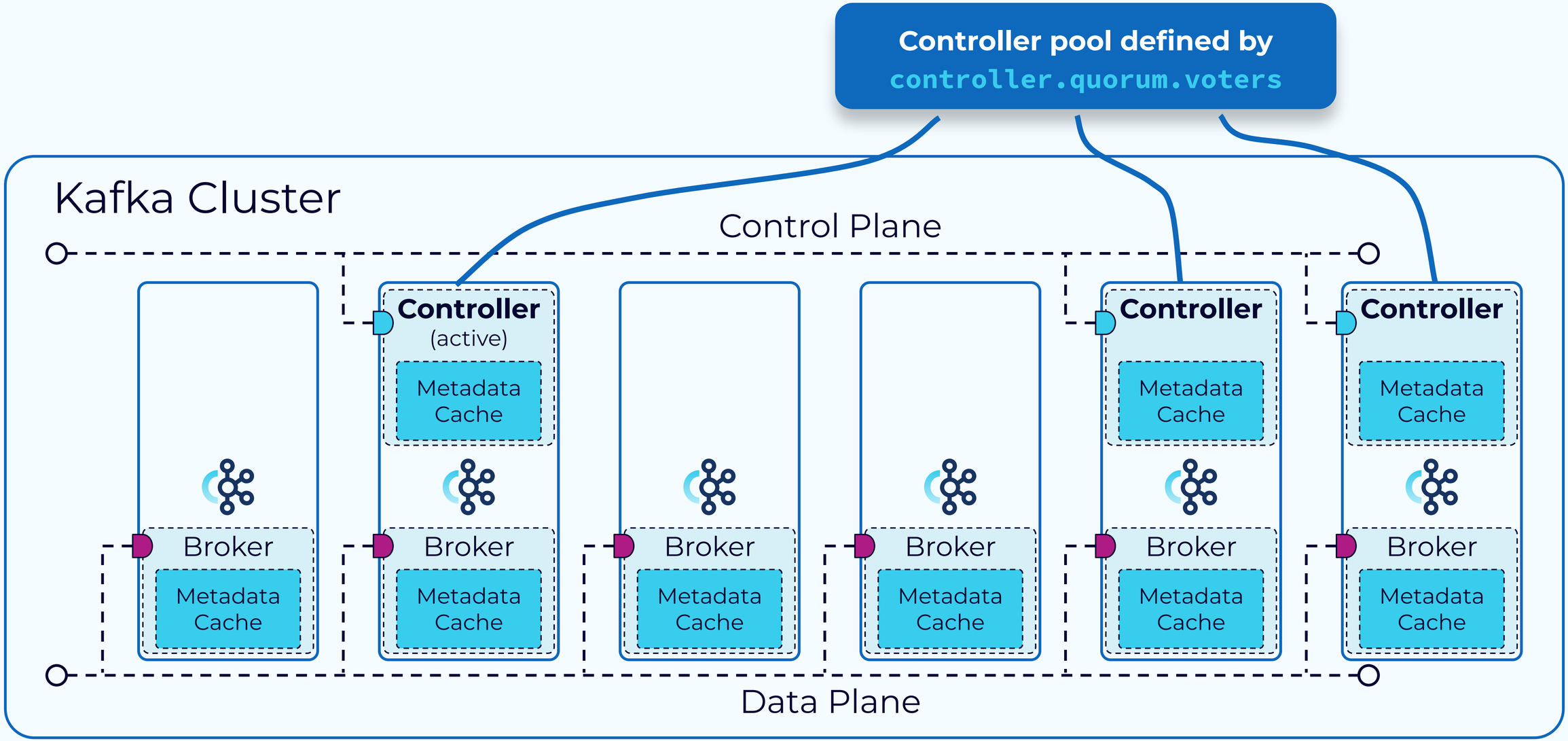
Cluster node role can be `controller` or `broker` or `controller, broker`

CONFLUENT Developer



KRaft Mode Controller

CONFLUENT Developer

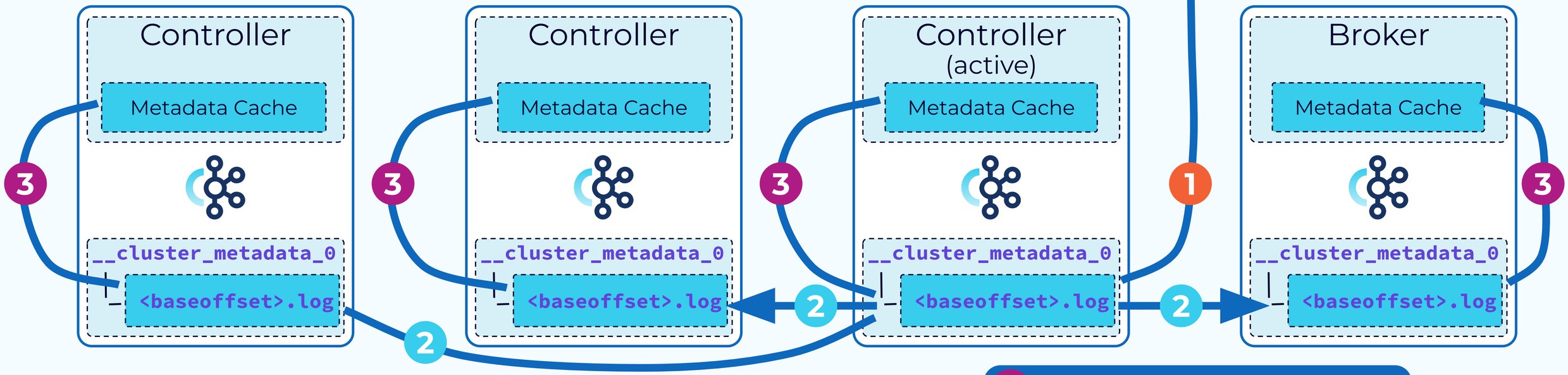


KRaft Cluster Metadata

CONFLUENT Developer

1 Active controller writes the metadata change records to the `__cluster_metadata` topic, e.g.

```
{"topicId":"JJmWDIT9Q9eXVDvCaBIJhg", "partitionId":0, "leader":1, "isr": [1,2,3]}
```



2 Controllers (followers) and brokers (observers) replicate `__cluster_metadata` topic events.

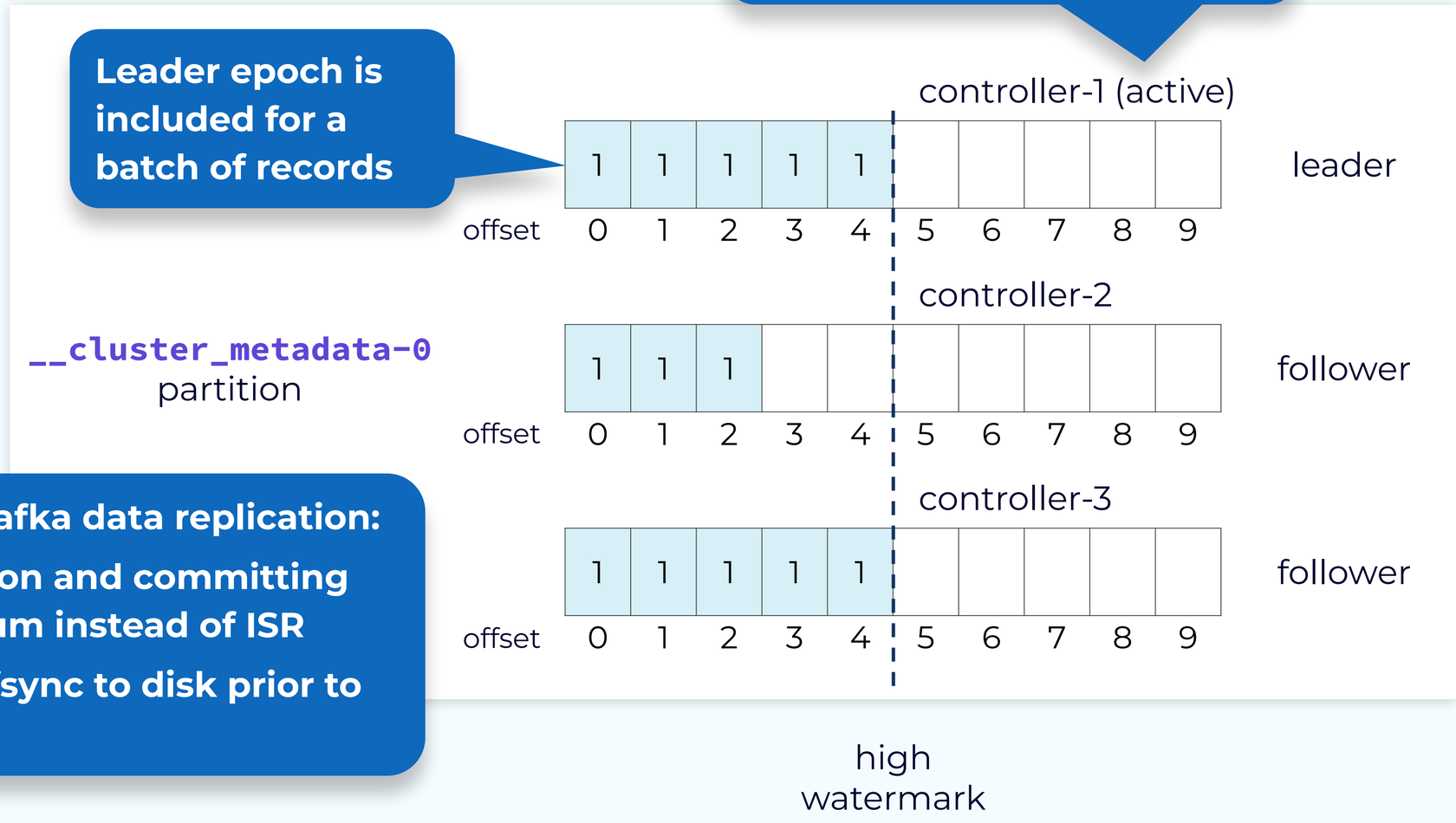
3 Controllers and brokers update metadata cache from committed log records

KRaft Metadata Replication

CONFLUENT Developer

The active controller is the `__cluster_metadata-0` partition leader

Leader epoch is included for a batch of records

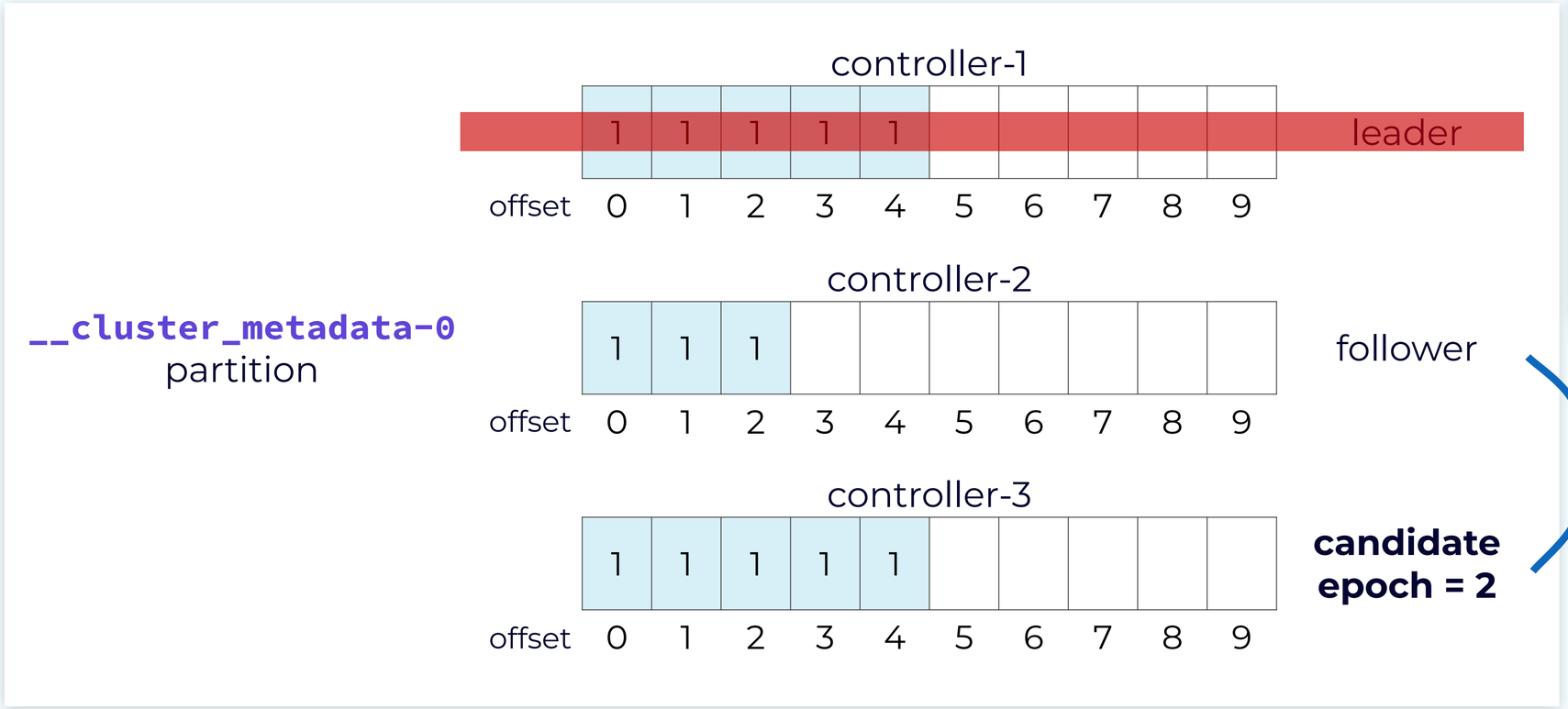


Difference to Kafka data replication:

- **Leader election and committing record: quorum instead of ISR**
- **Records are fsync to disk prior to commit**

Leader Election Step 1: Vote Request

CONFLUENT Developer



In response to an absent leader, a controller:

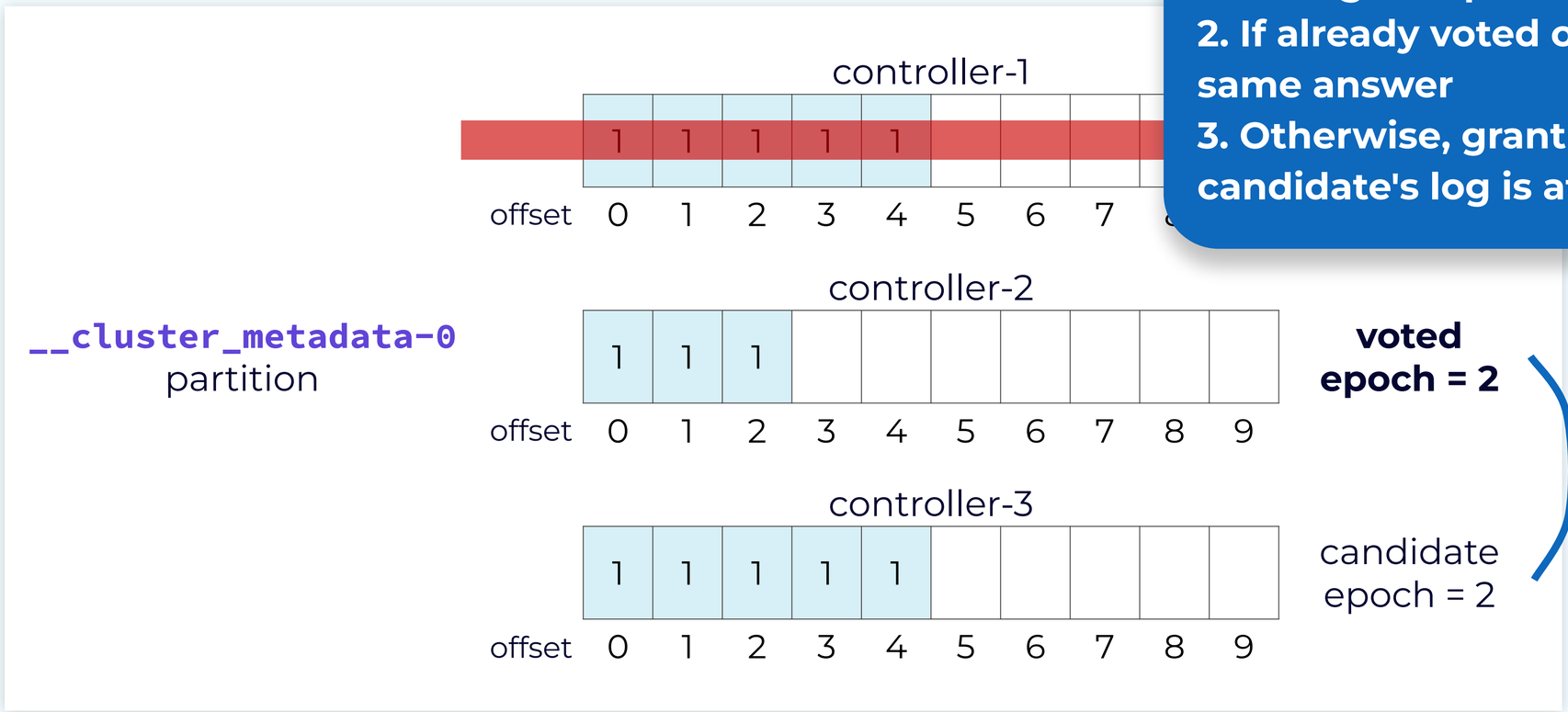
- **Increases its epoch**
- **Transitions to candidate**
- **Votes for itself**
- **Requests votes from other controllers**

```
VoteRequest
 candidateEpoch: 2
 lastOffset: 4
 lastOffsetEpoch: 1
```

Leader Election Step 2: Vote Response

CONFLUENT Developer

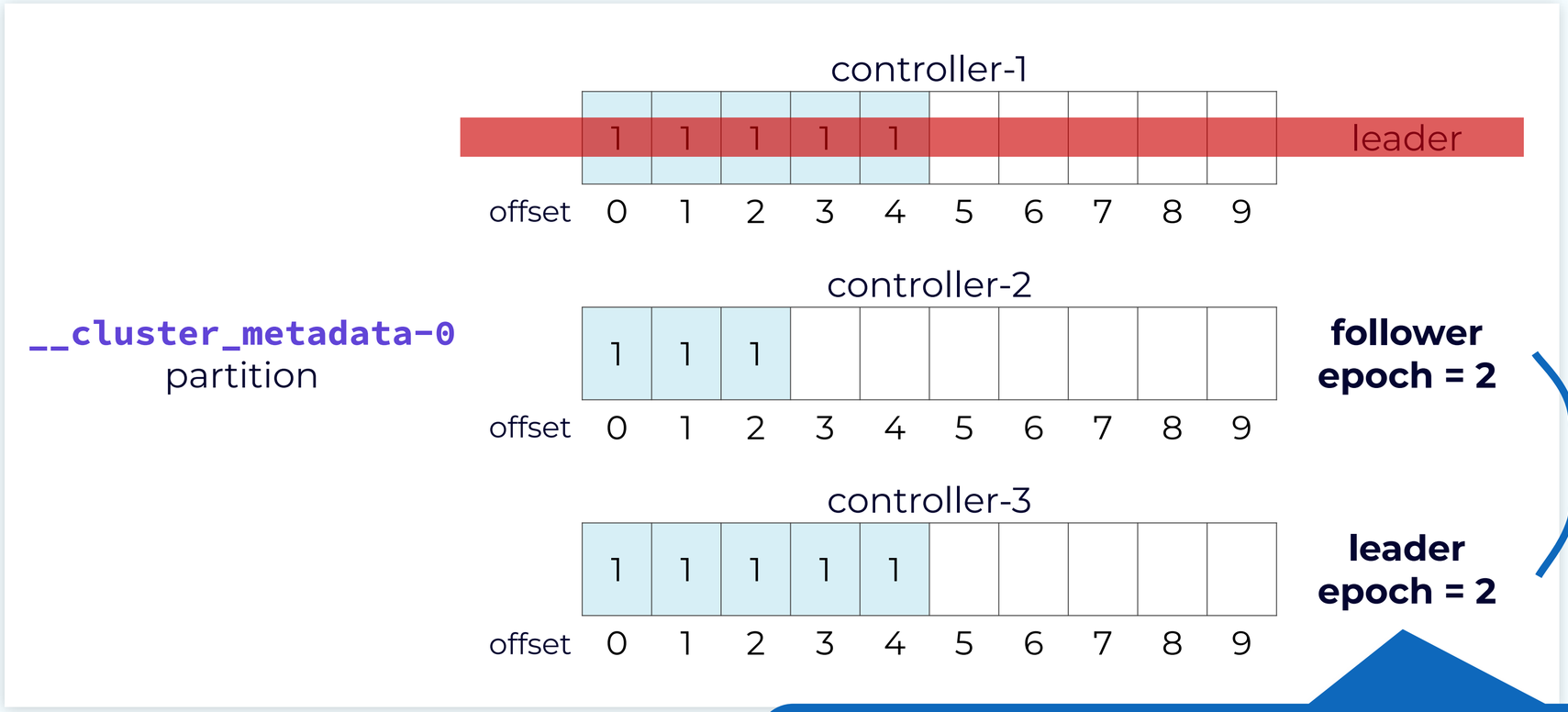
1. If a higher epoch is seen, reject
2. If already voted on epoch, send same answer
3. Otherwise, grant vote only if the candidate's log is at least as "long"



3 `controller-3 lastOffset: 4 lastOffsetEpoch: 1 ≥ controller-2 lastOffset: 2 lastOffsetEpoch: 1`

Leader Election Step 3: Completion

CONFLUENT Developer

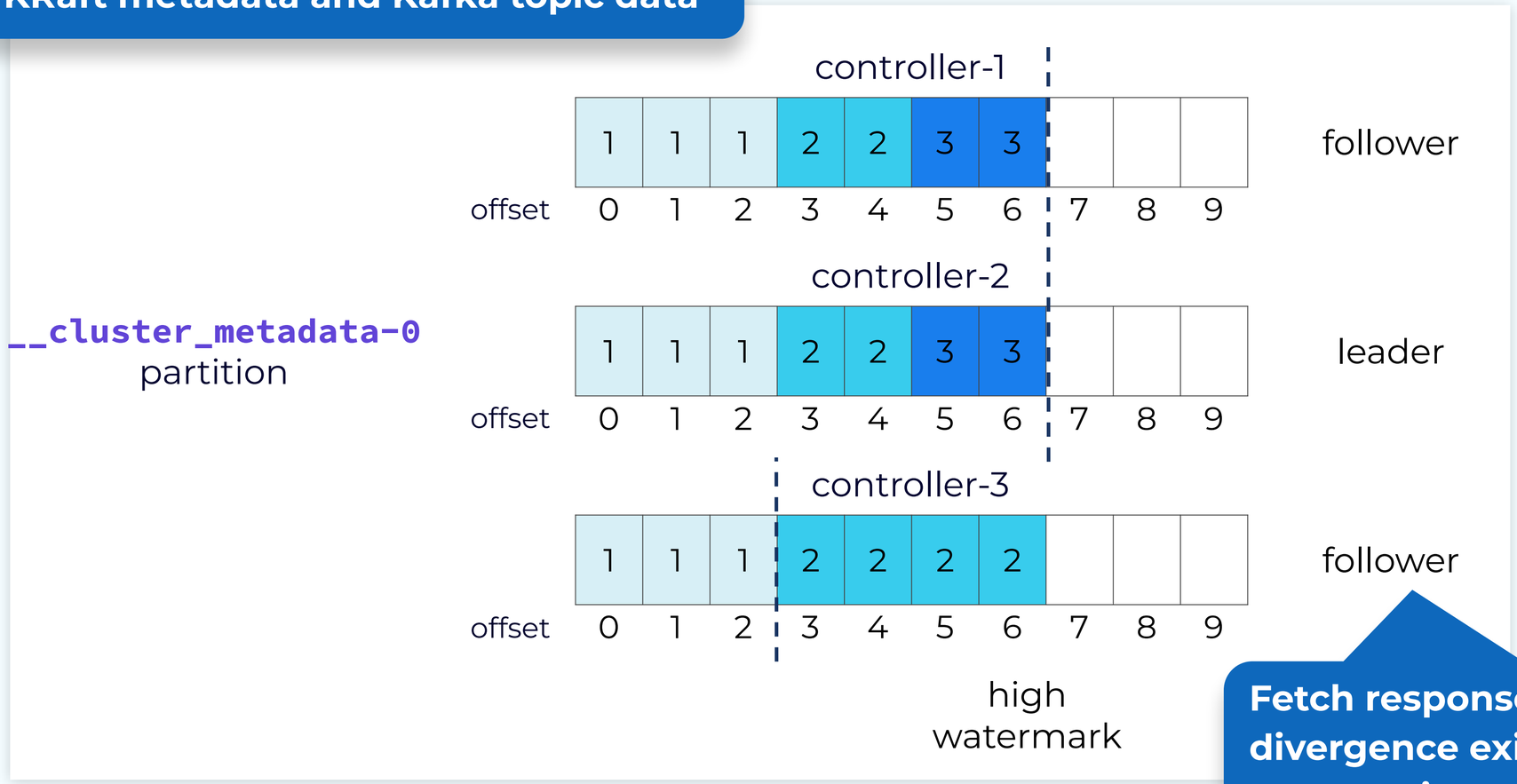


When a candidate receives the majority votes, it becomes the new leader and notifies other controllers the election is complete

Metadata Replica Reconciliation

CONFLUENT Developer

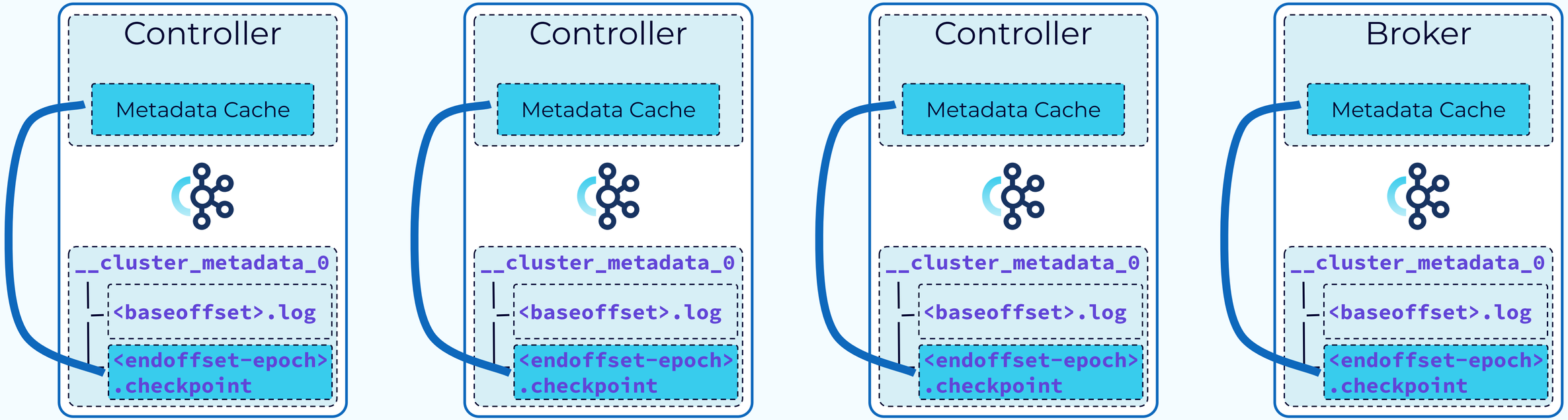
The same reconciliation process is used for both KRaft metadata and Kafka topic data



Fetch response will tell controller-3 divergence exists and it will truncate its replica to offset 5

KRaft Cluster Metadata Snapshot

CONFLUENT Developer



`00000000000000000128-0000000001.checkpoint`

Periodic point-in-time snapshot of in-memory metadata cache

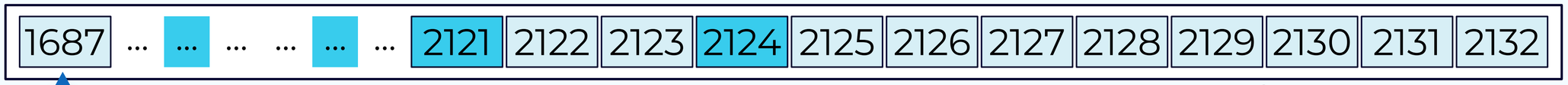
- **Uniquely identified by its `EndOffset` and `Epoch`**
- **Stores only one entry per key**
- **A bit like a “compacted topic”**

Metadata Log Truncation with Snapshot

CONFLUENT Developer

Log of events, i.e. multiple events, potentially for the same key, in order

`__cluster_metadata-0/00000000000000001687.log`



`LogStartOffset`

Snapshot → 2129

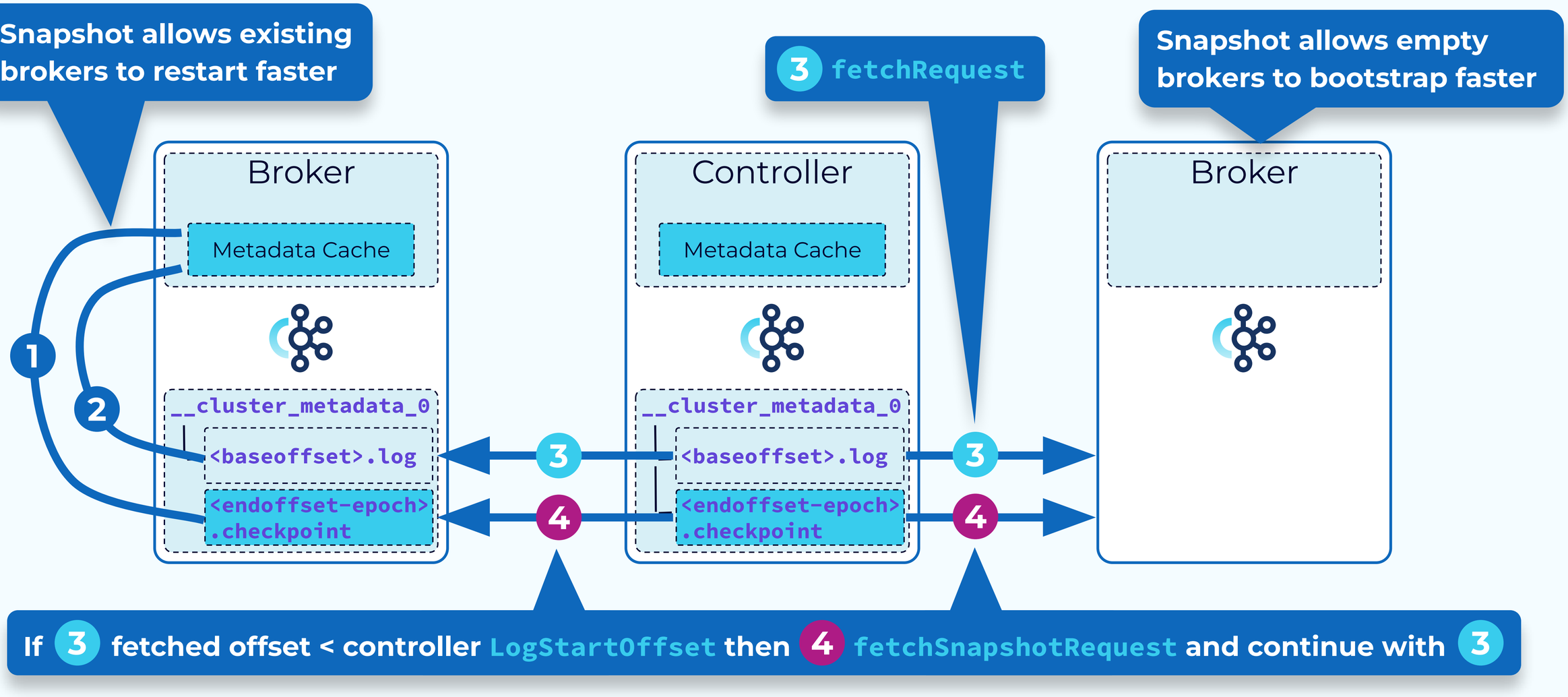
Log growth is limited since it is safe to truncate the beginning of the log up to the end of the latest snapshot

`00000000000000002130-0000000003.checkpoint`

- redundant record for its key not included in snapshot

When a Snapshot Is Read

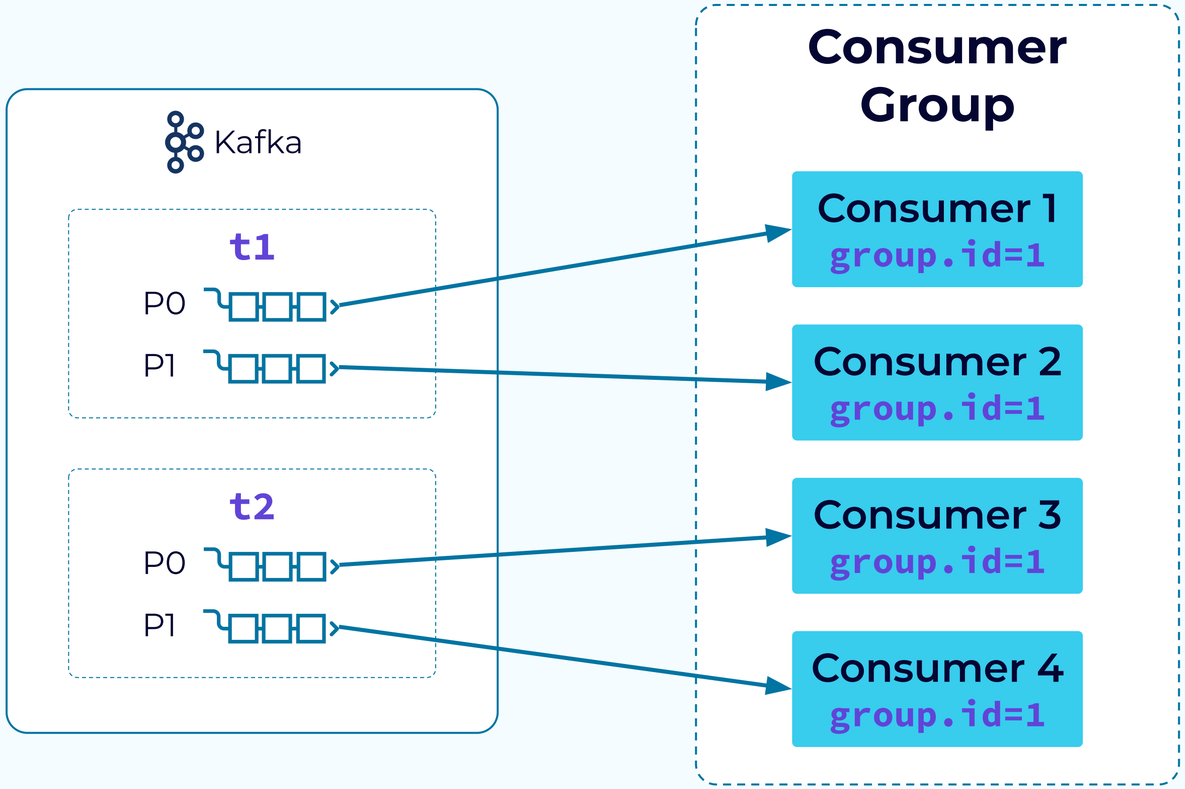
CONFLUENT Developer



CONFLUENT
Developer

Consumer Group Protocol

Kafka Consumer Group



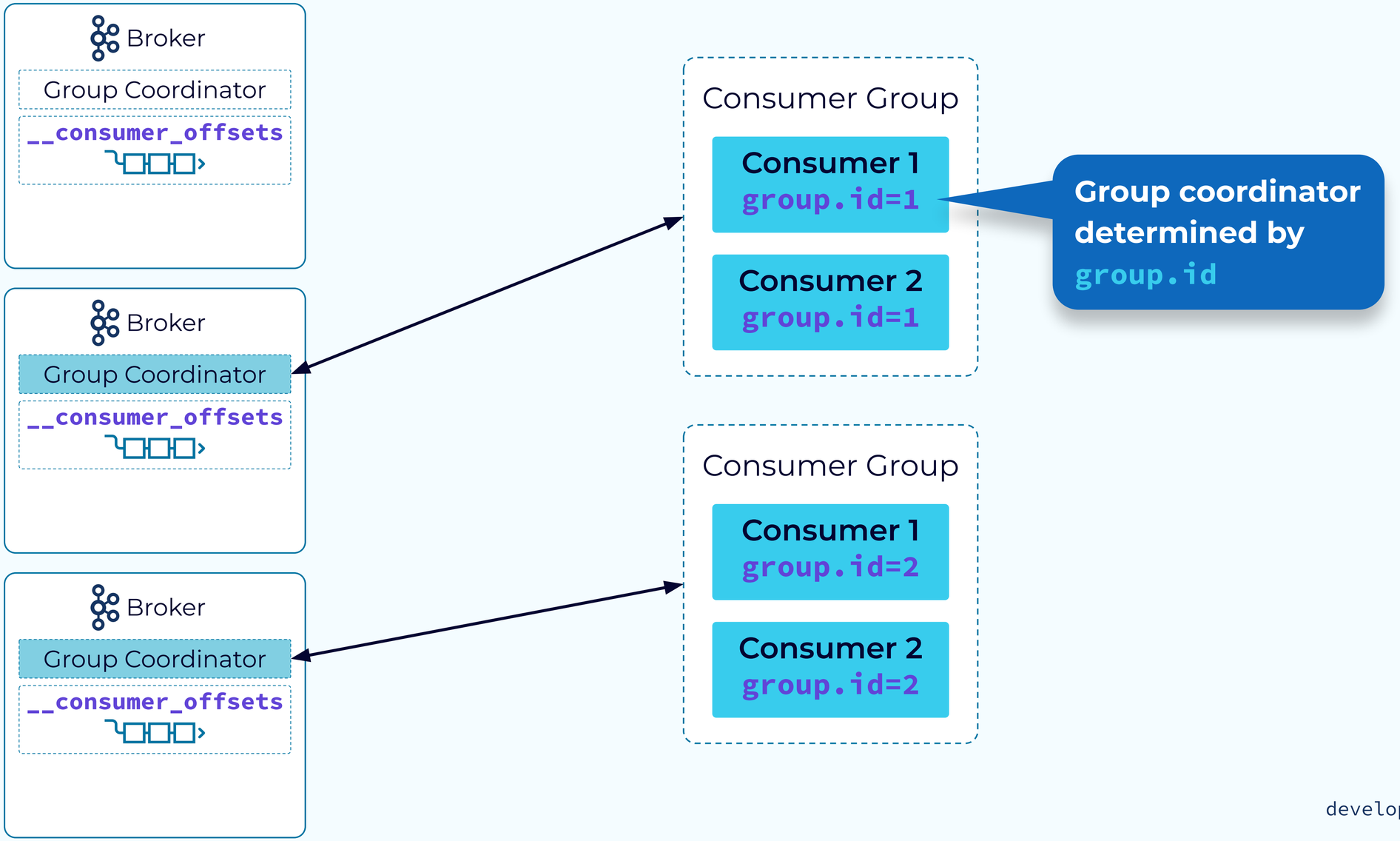
To enable group consumption:

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

Benefits:

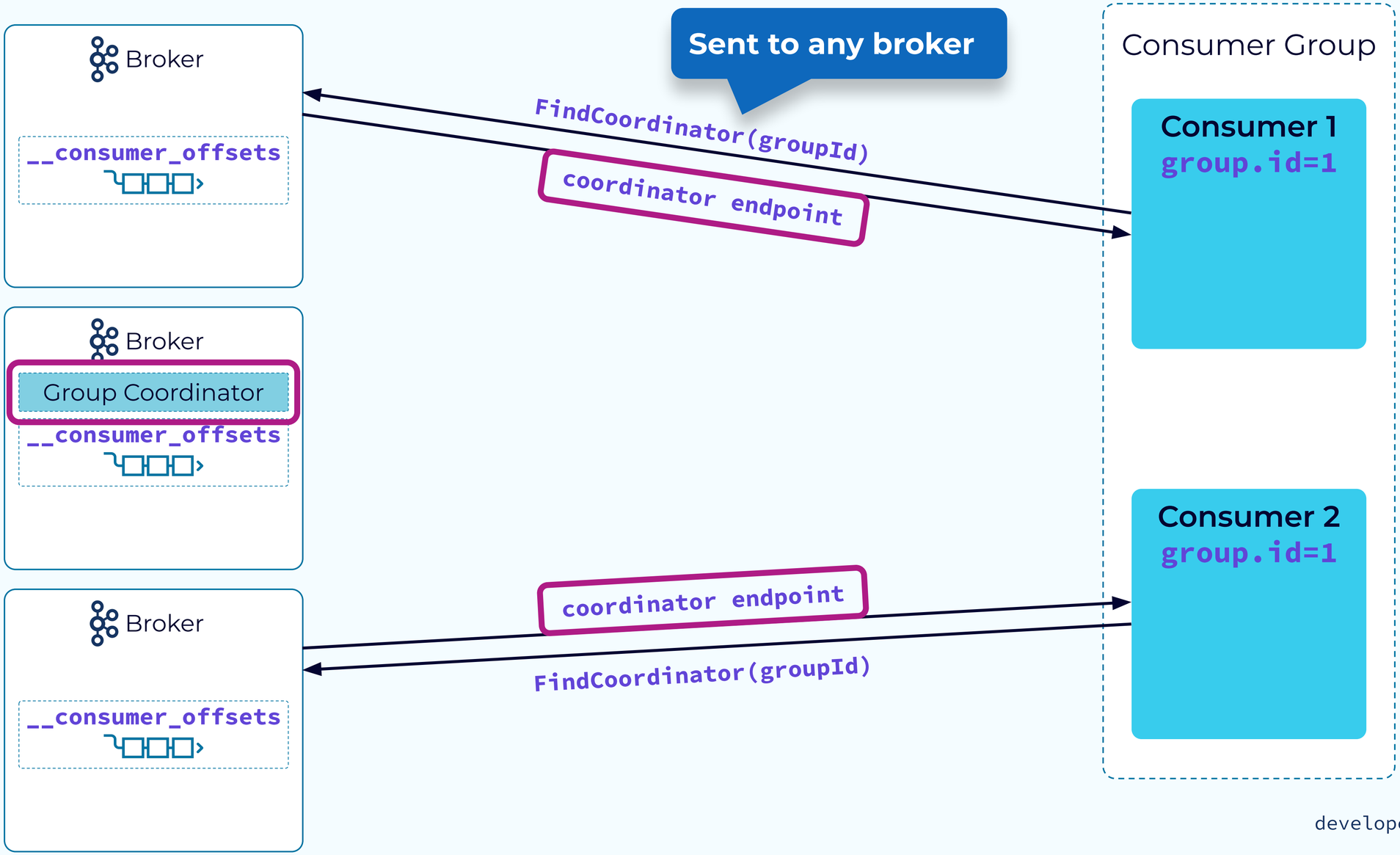
1) Scalability
2) Elasticity
3) Fault tolerance

Group Coordinator



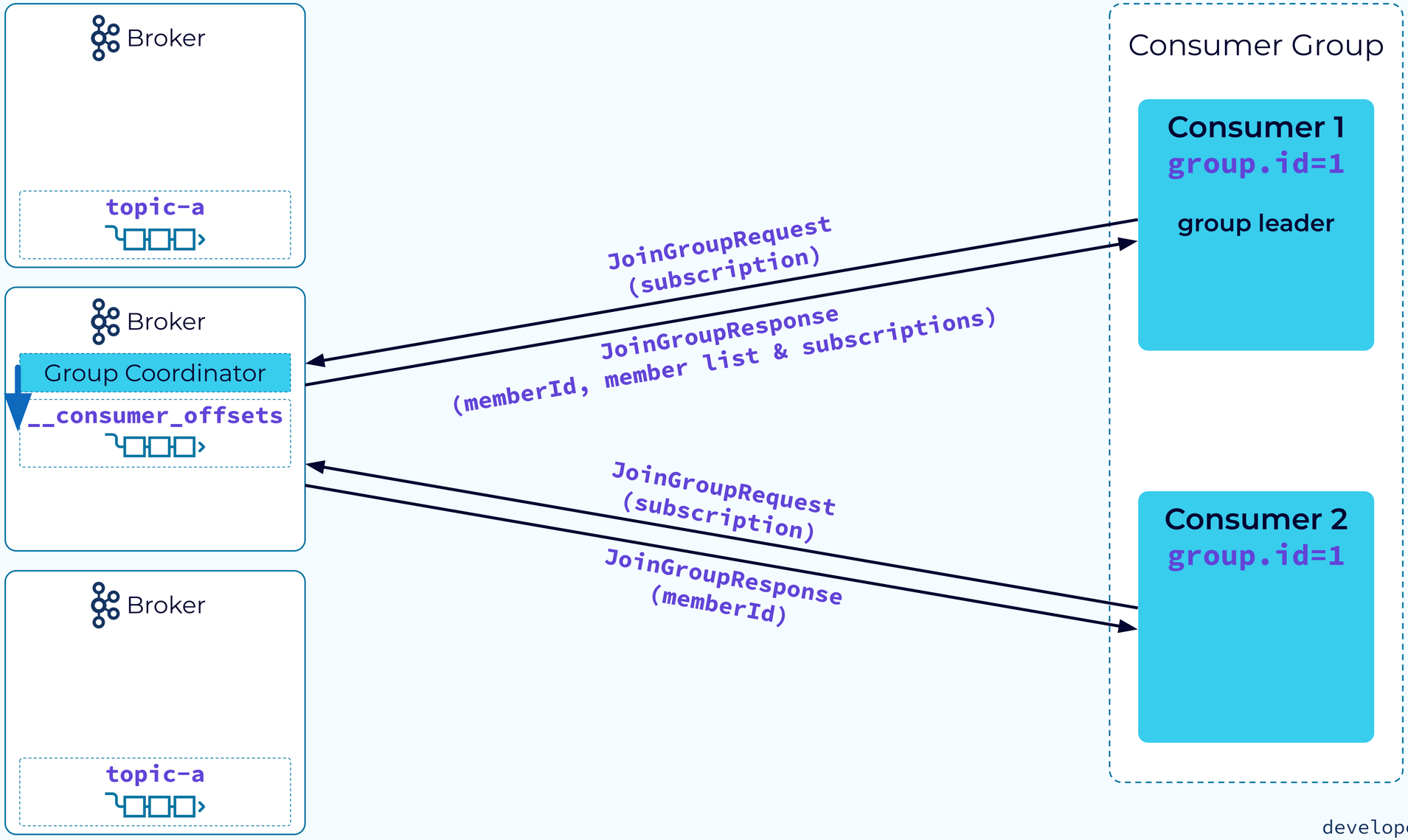
Group Startup: Step 1 - Find Group Coordinator

CONFLUENT Developer



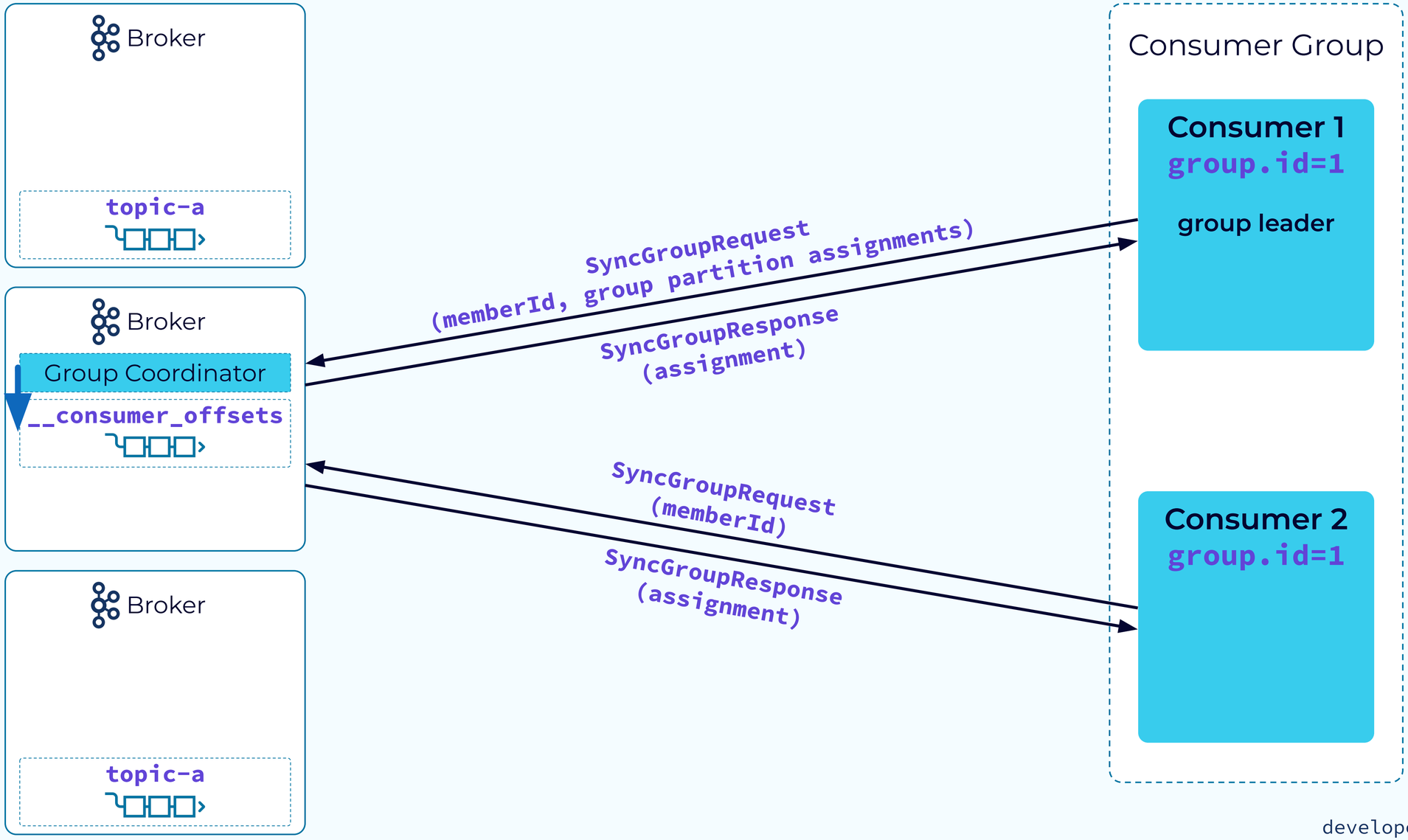
Group Startup: Step 2 - Members Join

CONFLUENT Developer



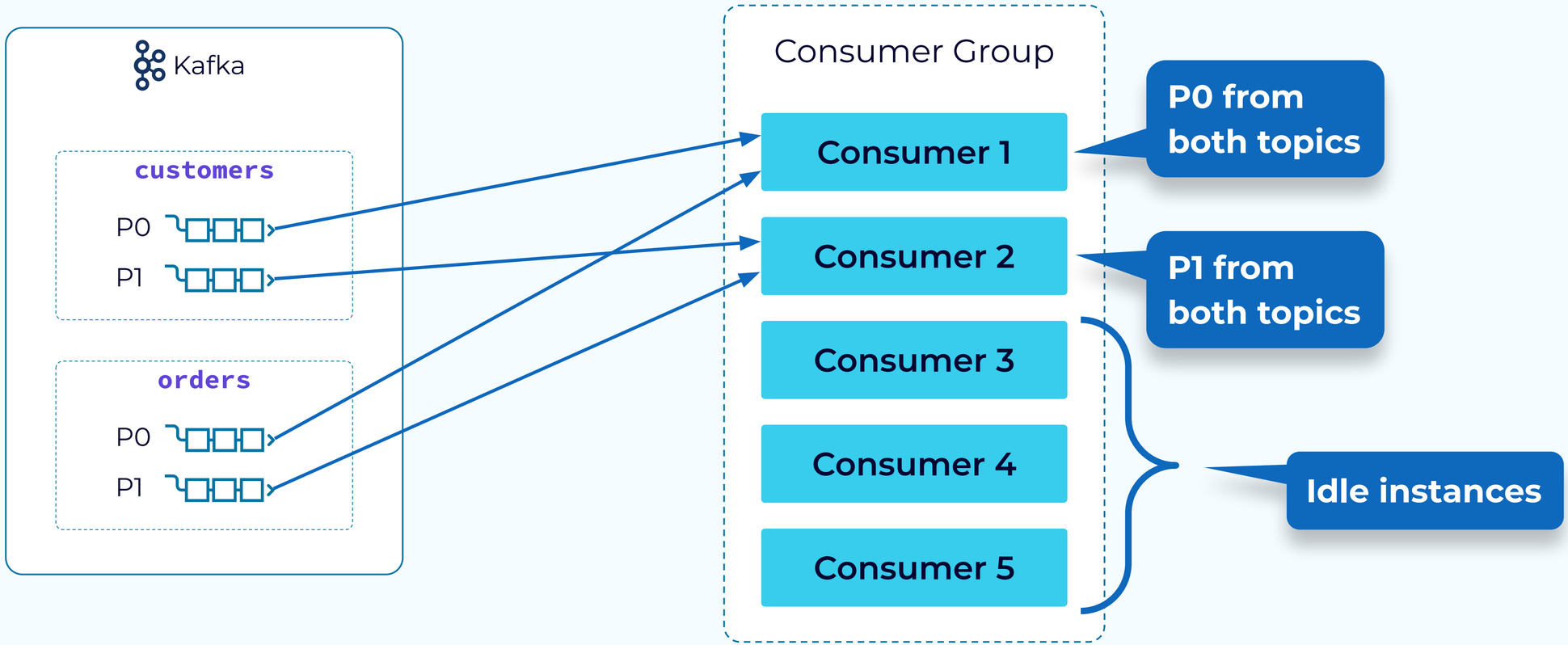
Group Startup: Step 3 - Partitions Assigned

CONFLUENT Developer



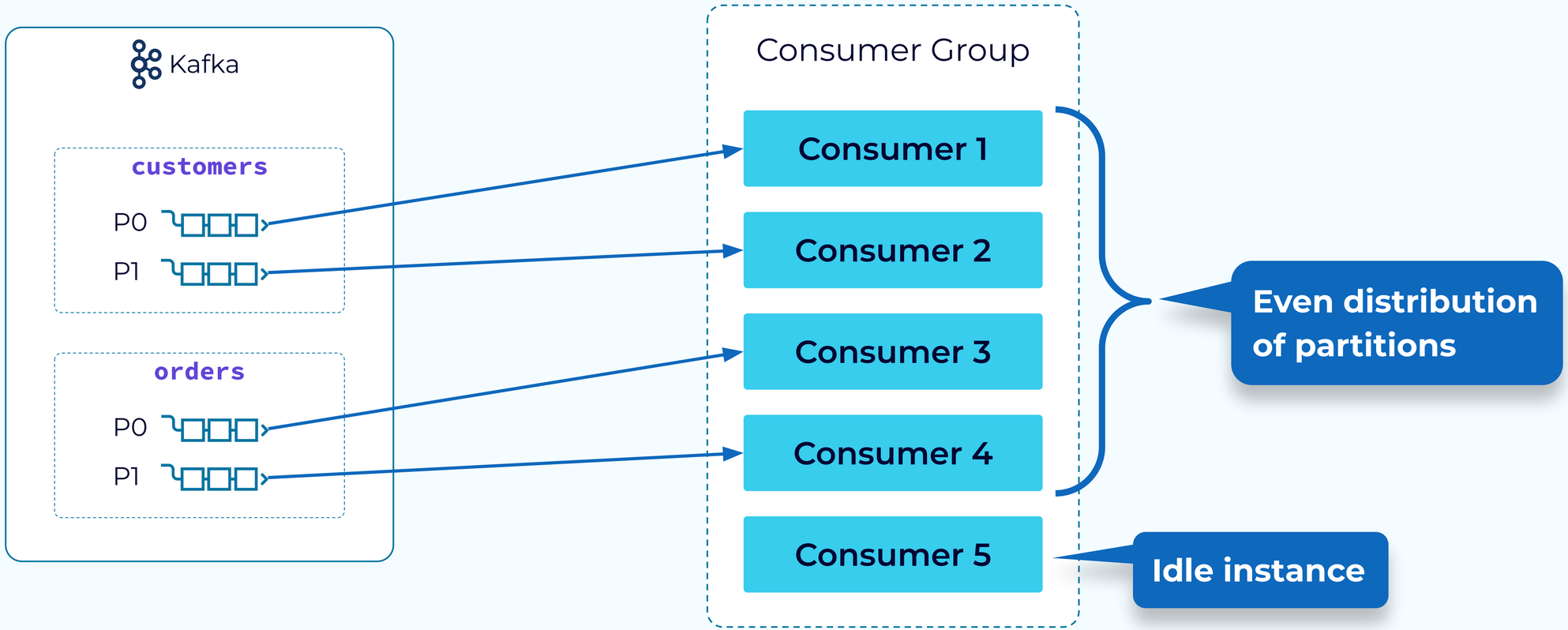
Range Partition Assignment Strategies

CONFLUENT Developer

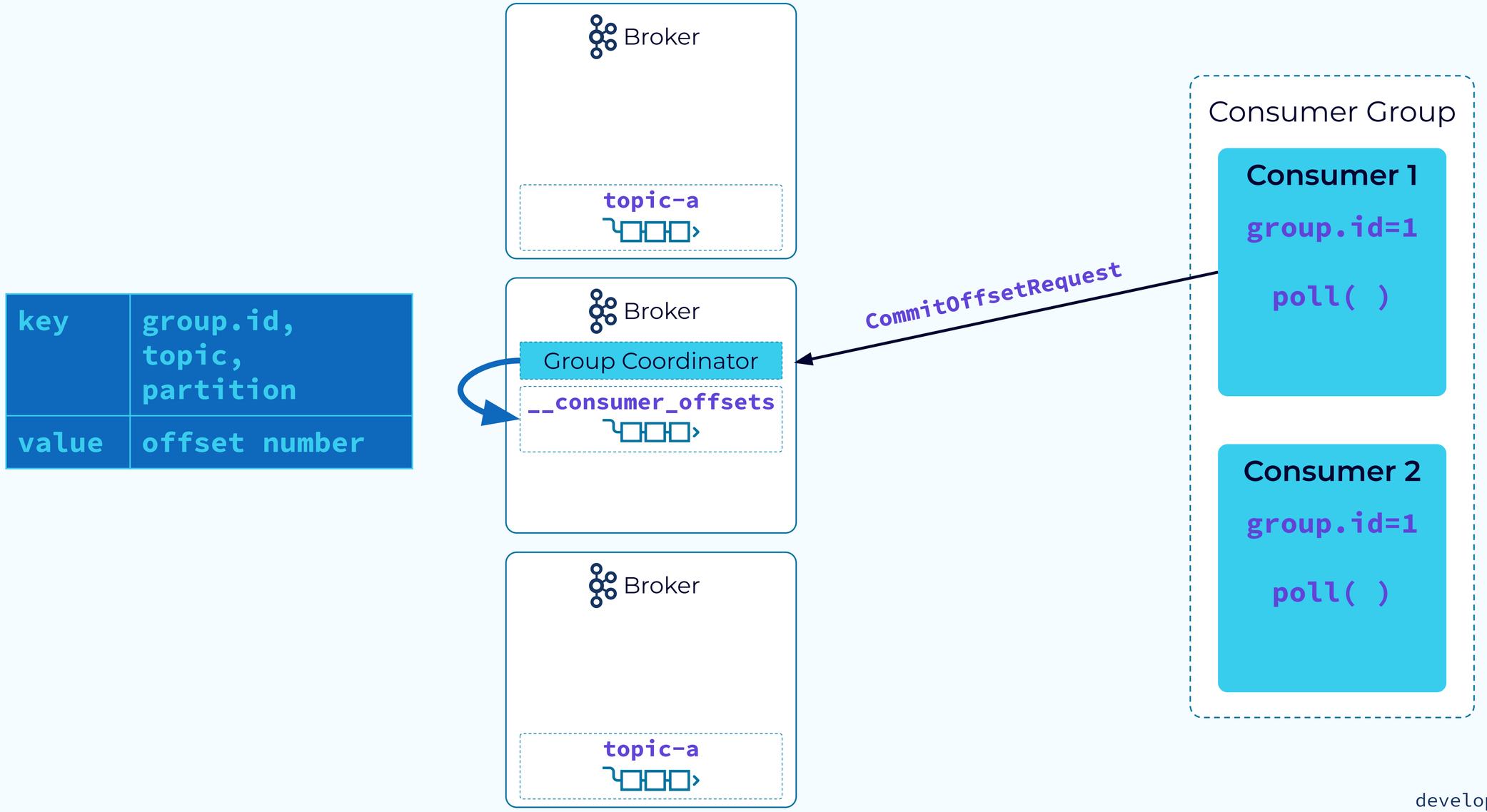


Round Robin and Sticky Partition Assignment Strategies

CONFLUENT Developer

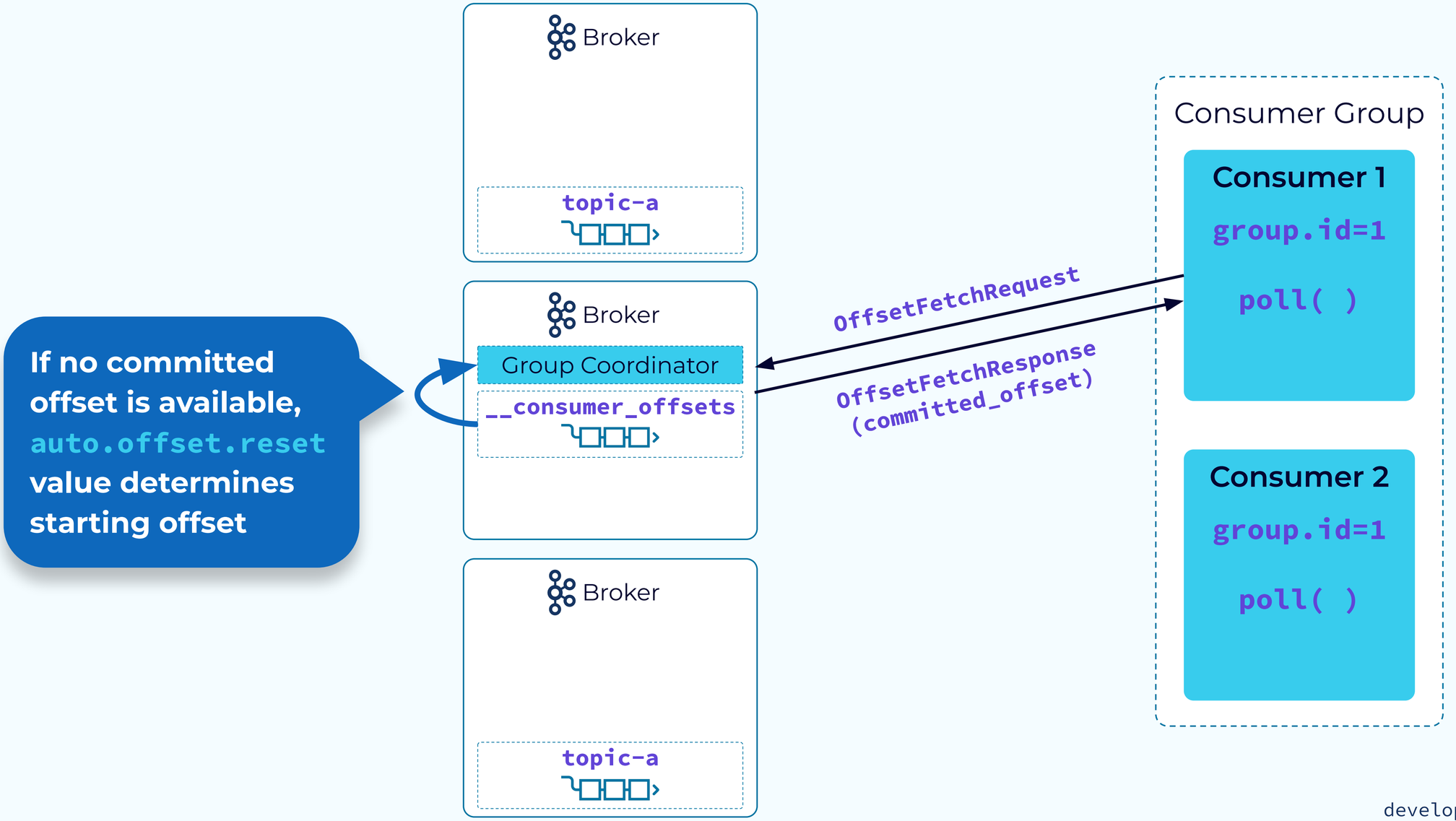


Tracking Partition Consumption



Determining Starting Offset to Consume

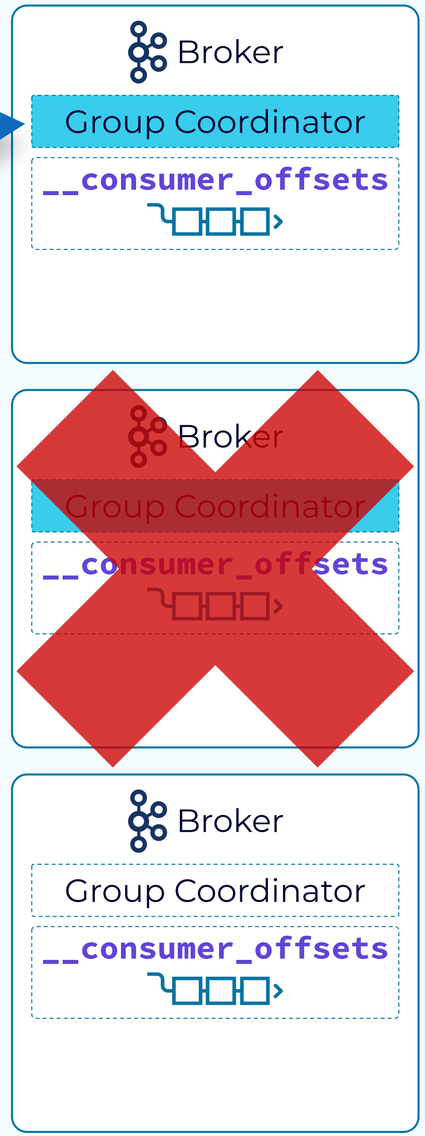
CONFLUENT Developer

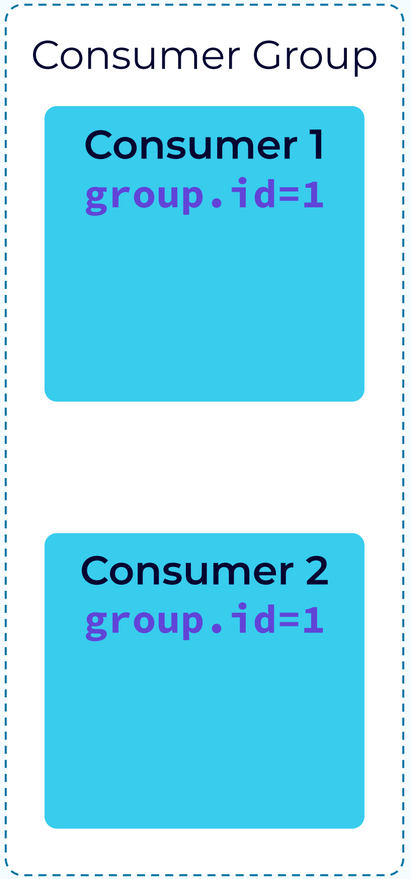


Group Coordinator Failover

CONFLUENT Developer

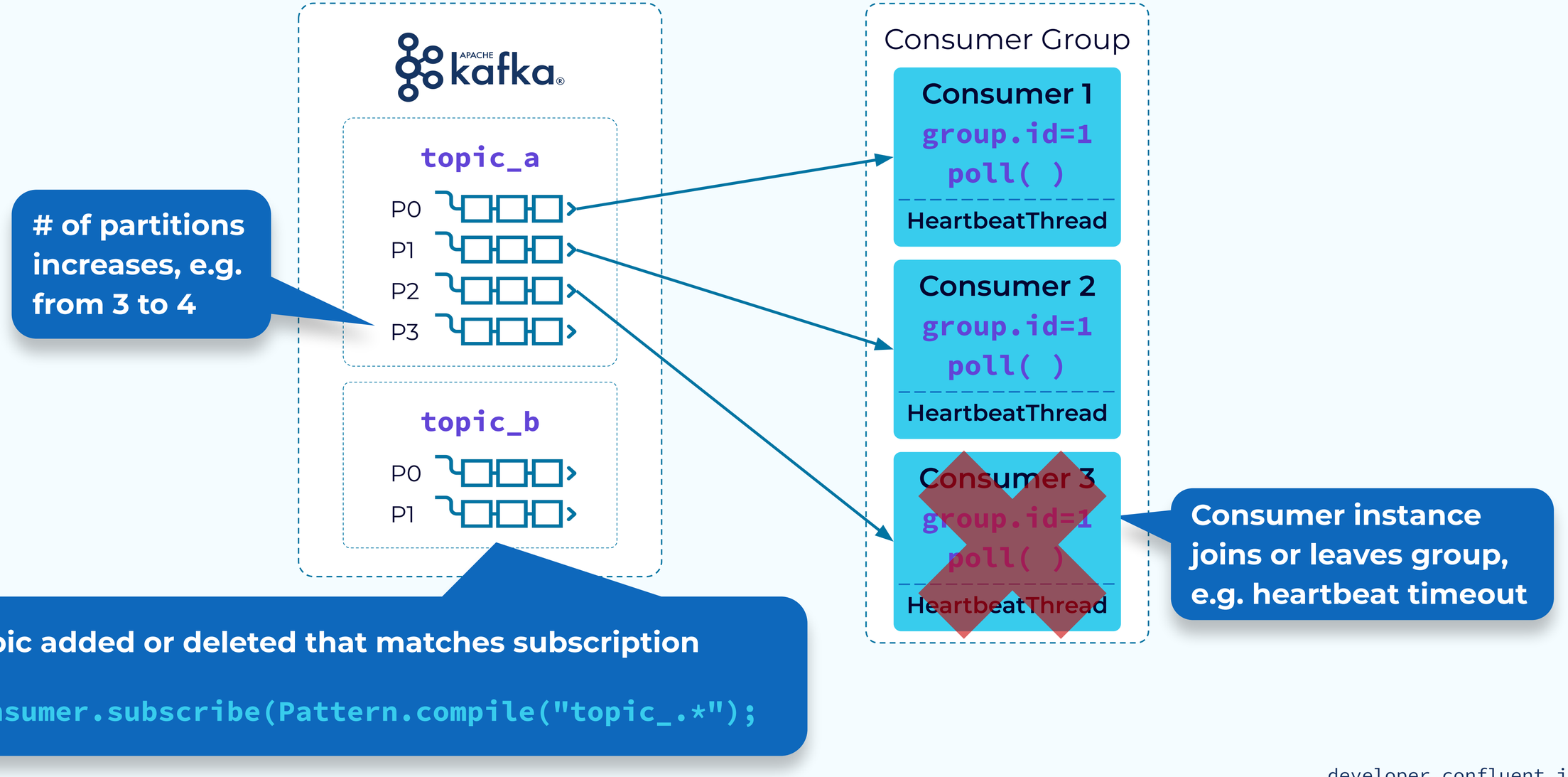
Group coordinator fails over to `__consumer_offsets` new partition leader





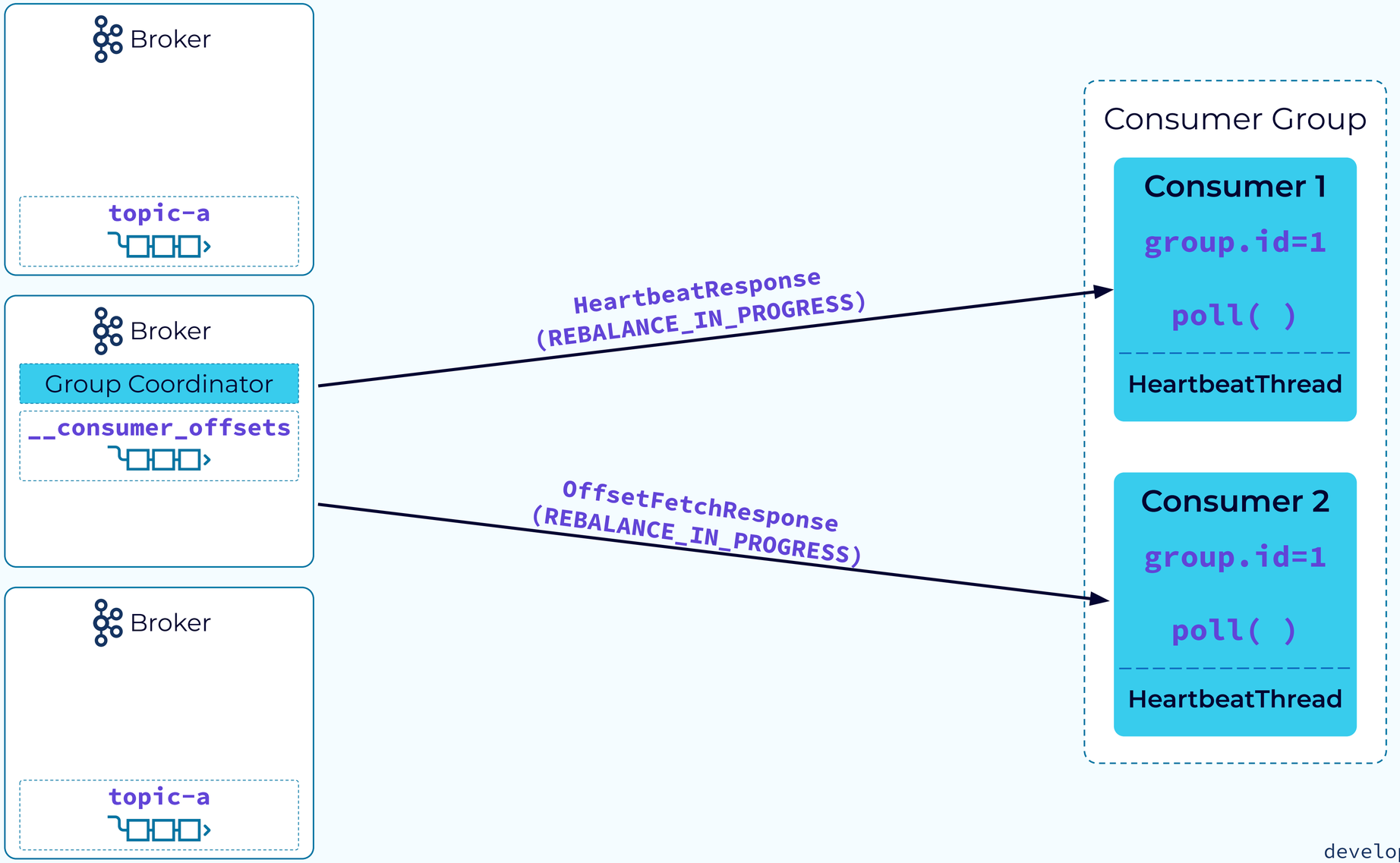
Consumer Group Rebalance Triggers

CONFLUENT Developer



Consumer Group Rebalance Notification

CONFLUENT Developer

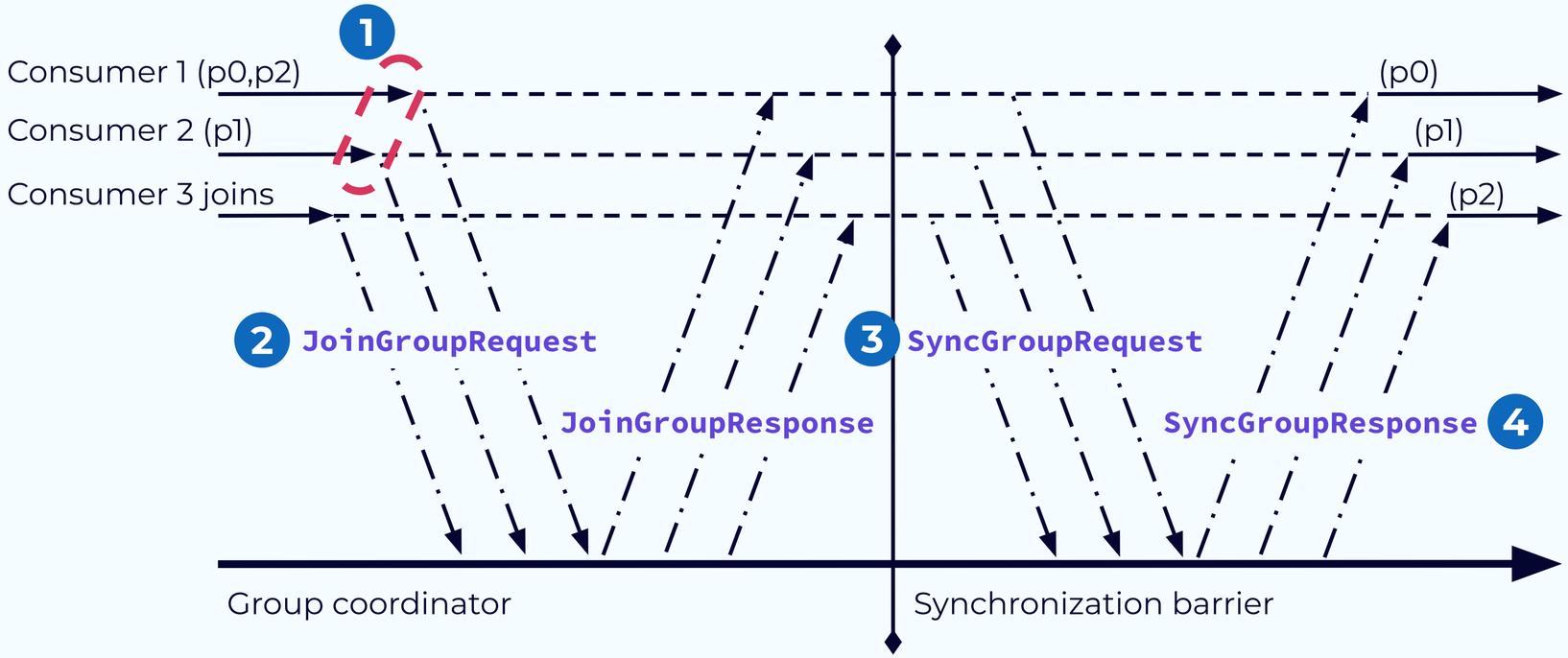


Stop-the-world Rebalance

CONFLUENT Developer

Consumers:

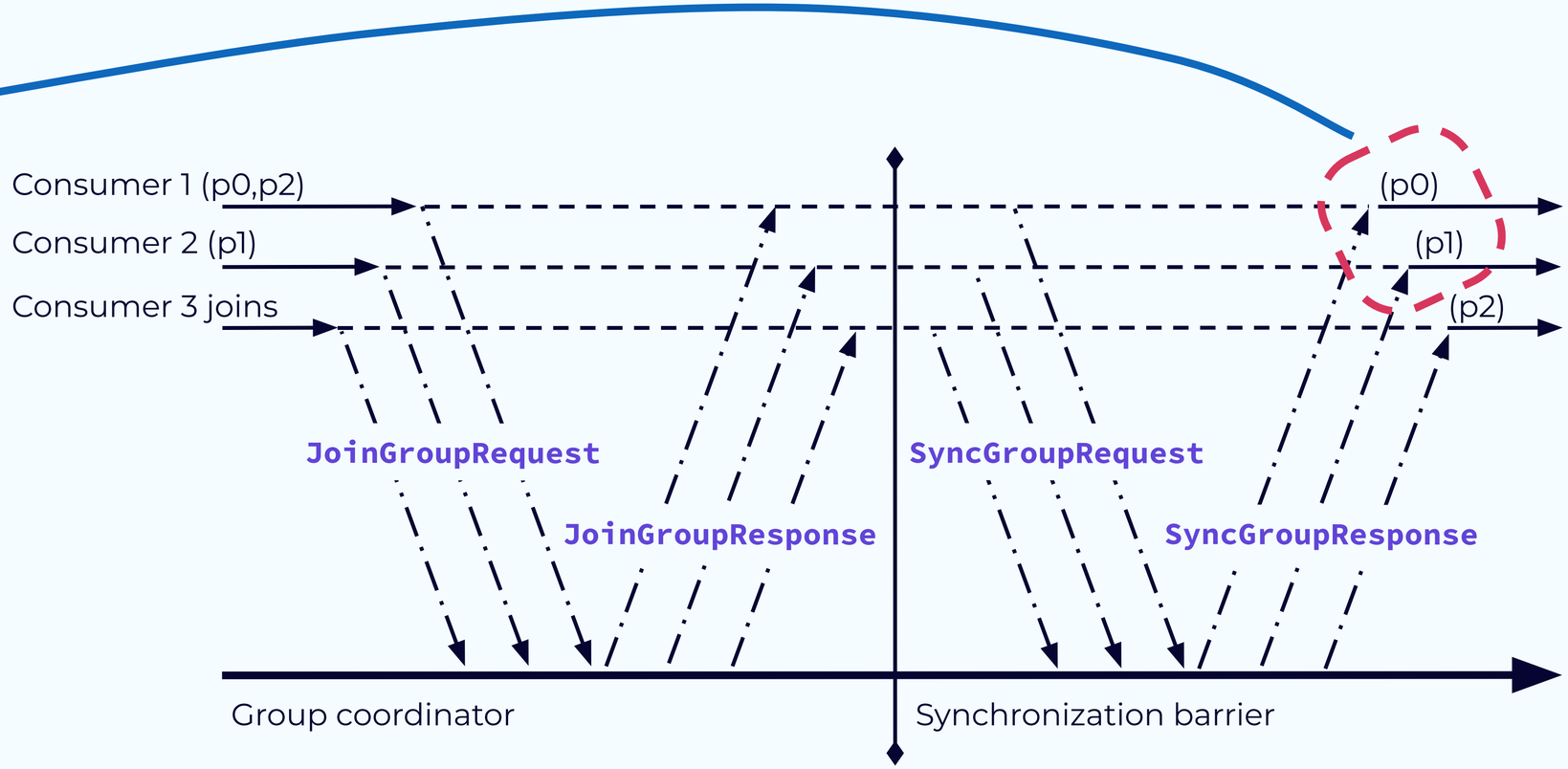
1) Revoke current partition assignment and clean up the partition states
2) Join the group
3) Sync with the group
4) Receive new partition assignments
   a) Build the partition state
   b) Resume consumption



Stop-the-world Problem 1 - Rebuilding the State

CONFLUENT Developer

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

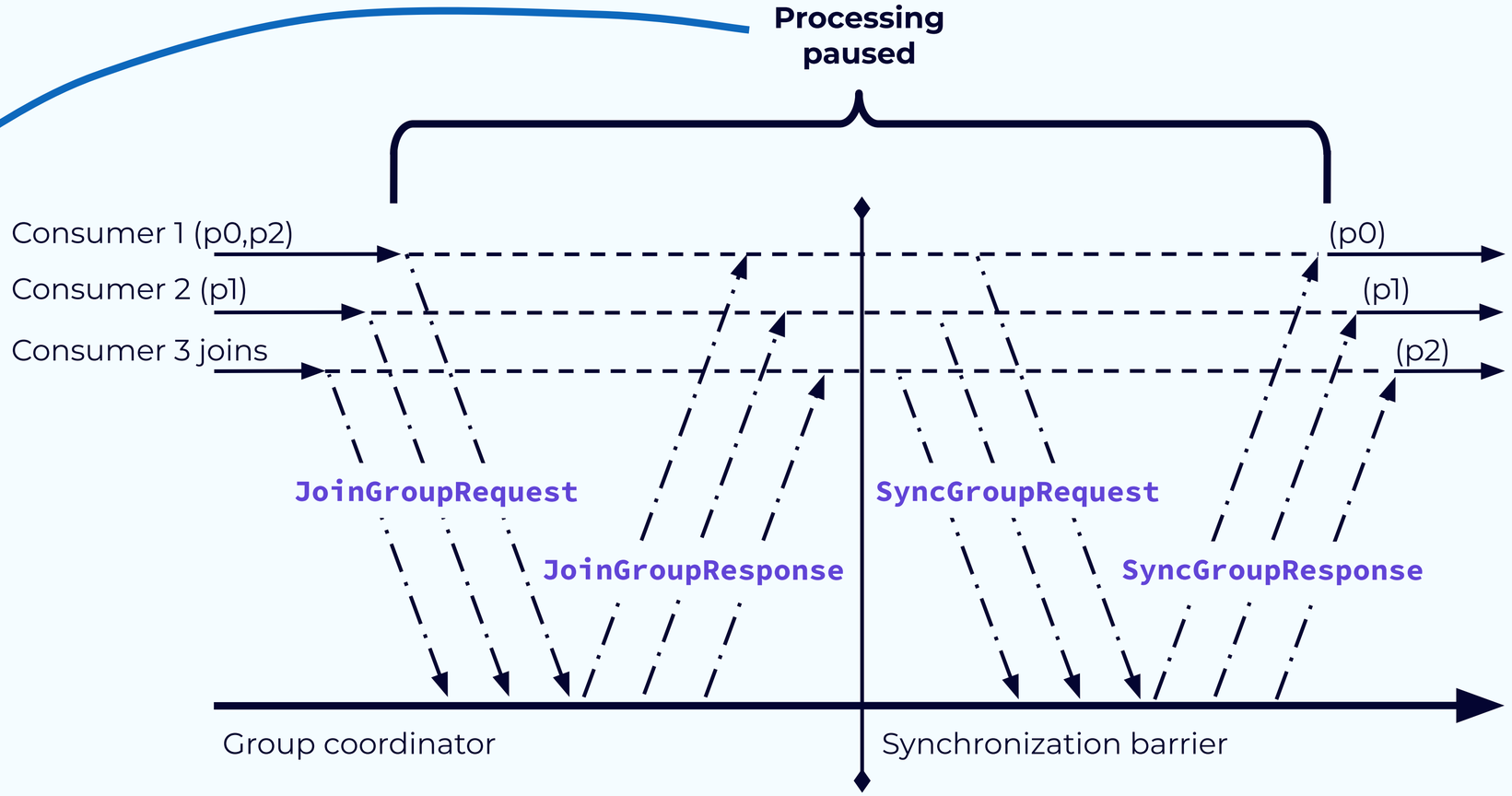


Stop-the-world Problem 2 - Paused Processing

CONFLUENT Developer

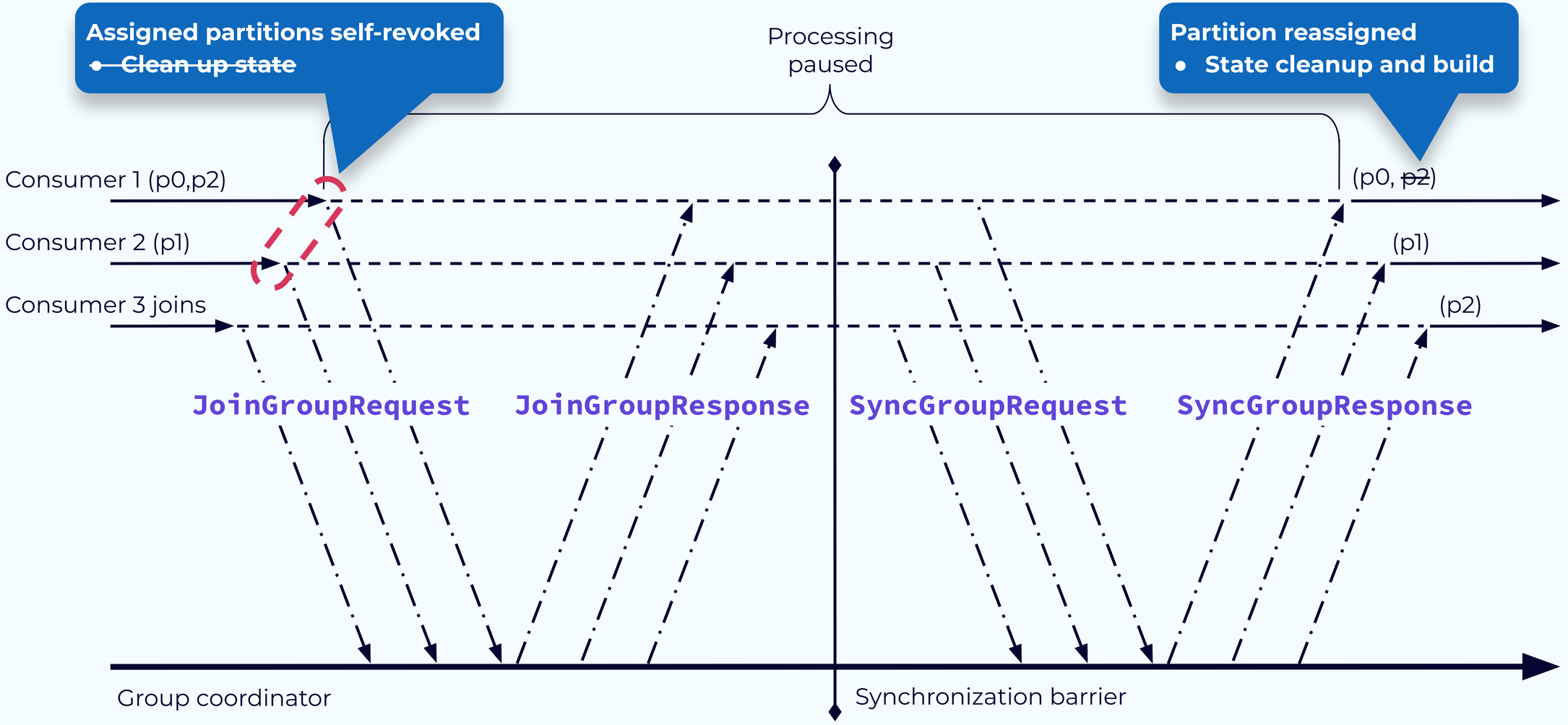
Processing pauses for all subscribed partitions for the duration of the rebalance

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



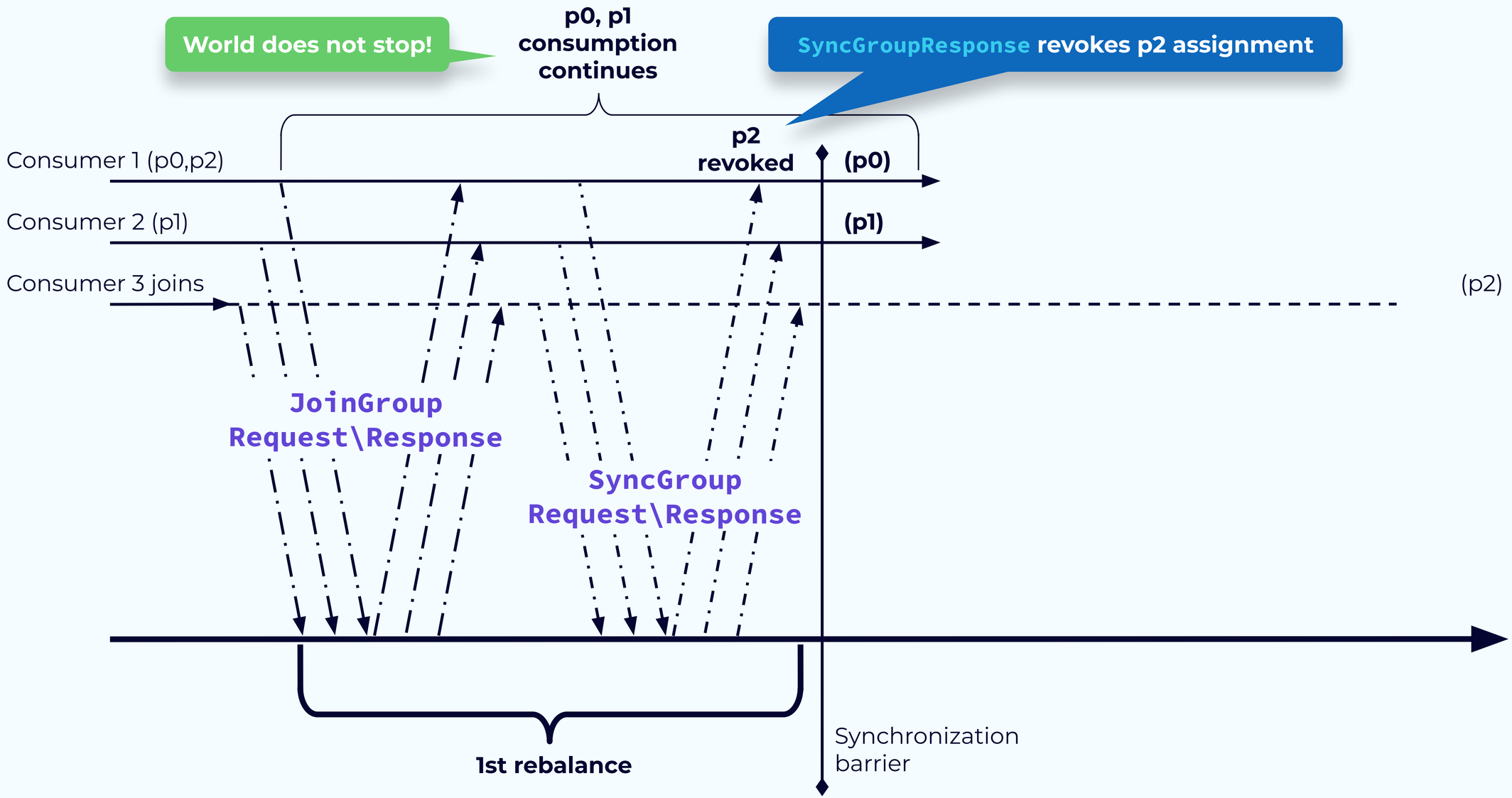
Avoid Needless State Rebuild with `StickyAssignor`

CONFLUENT Developer



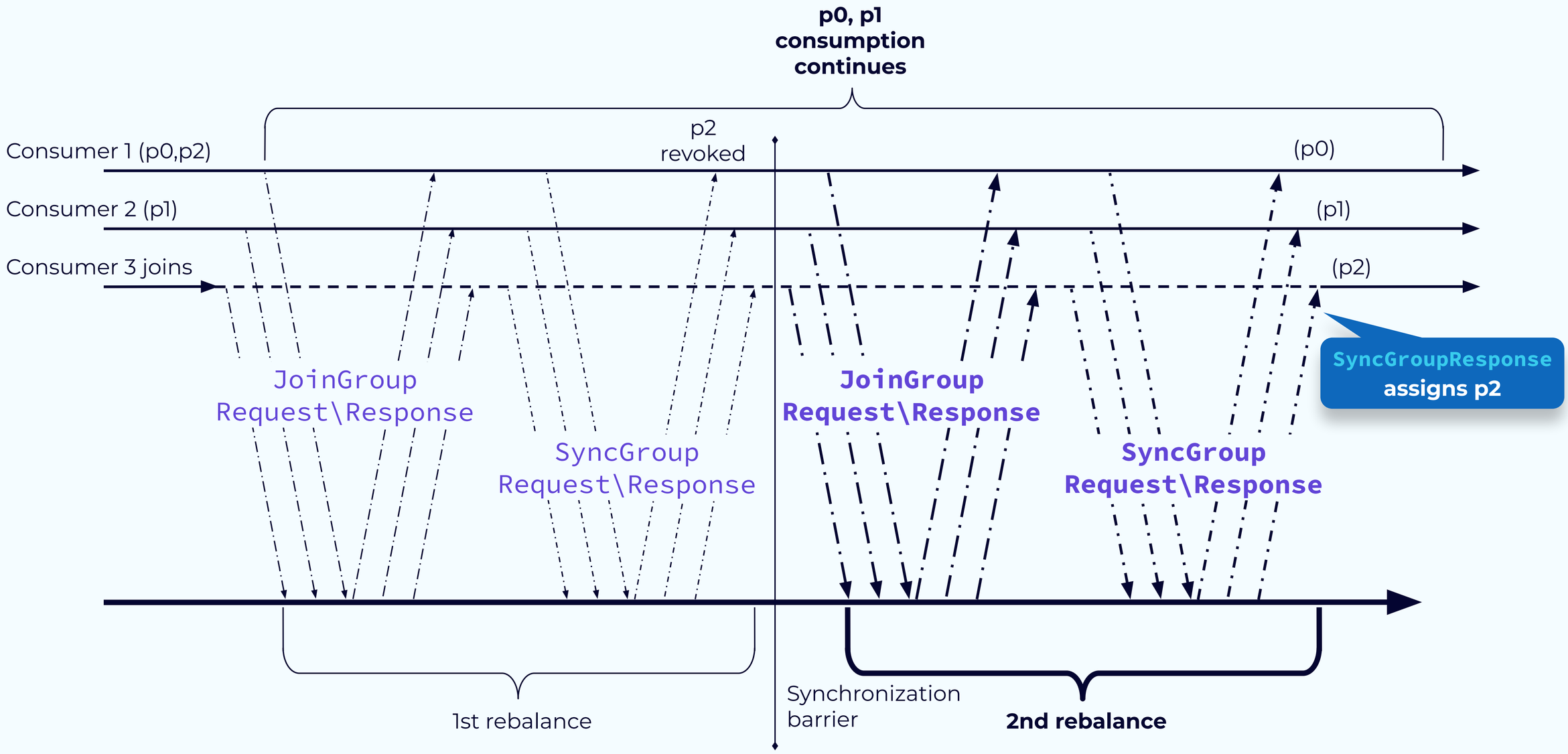
Avoid Processing Pause with `CooperativeStickyAssignor`

CONFLUENT Developer

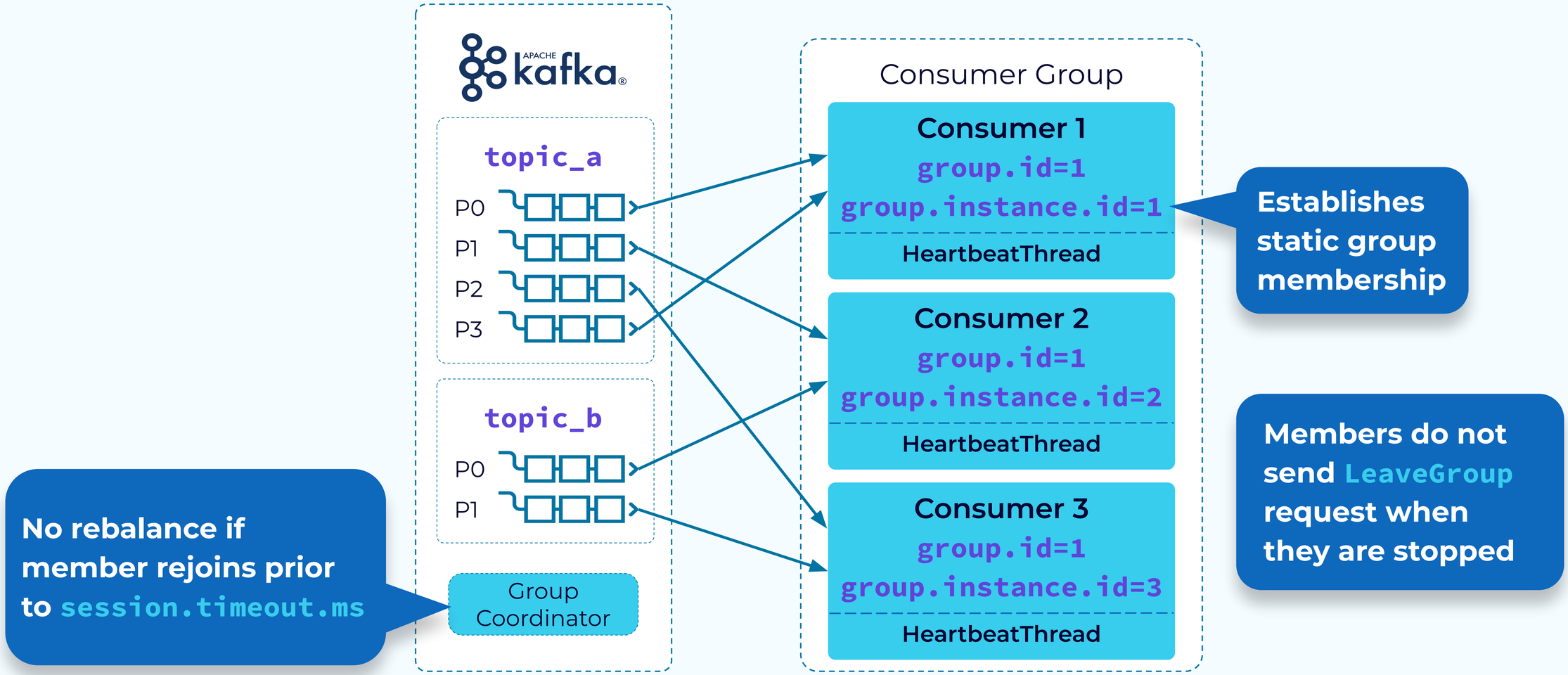


Avoid Processing Pause with `CooperativeStickyAssignor`

CONFLUENT Developer



Avoid Rebalance with Static Group Membership

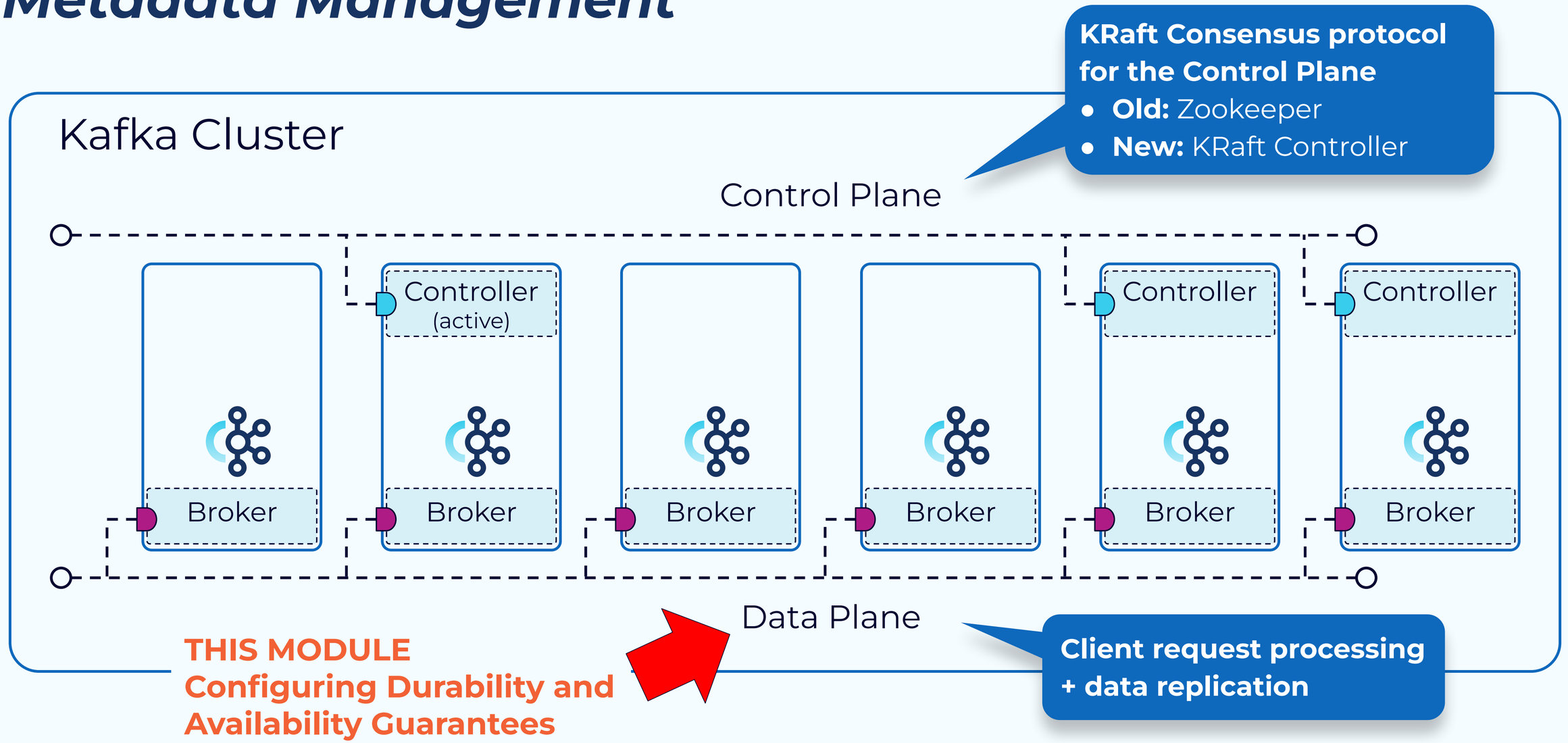


CONFLUENT
Developer

Configuring Durability, Availability and Ordering Guarantees

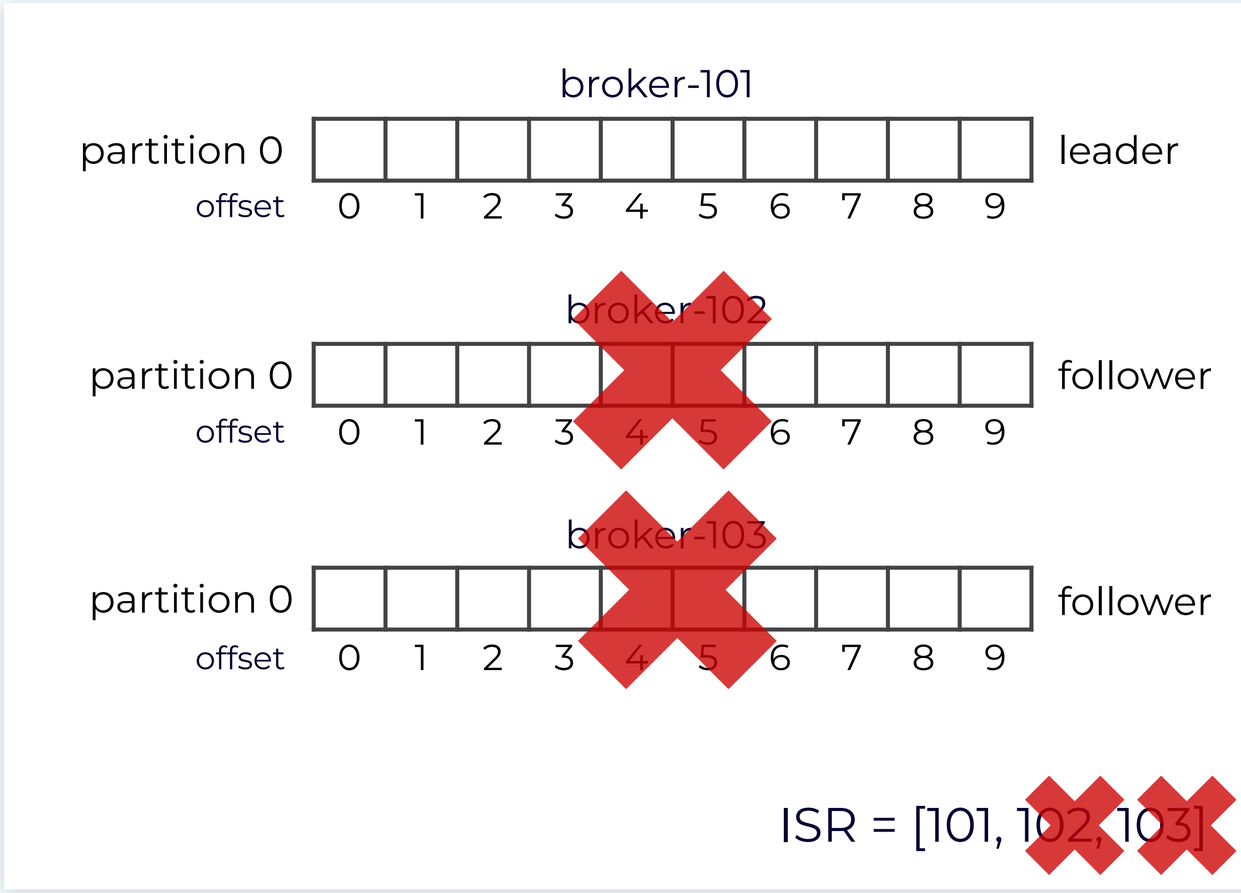
Kafka Separates Data & Metadata Management

CONFLUENT Developer



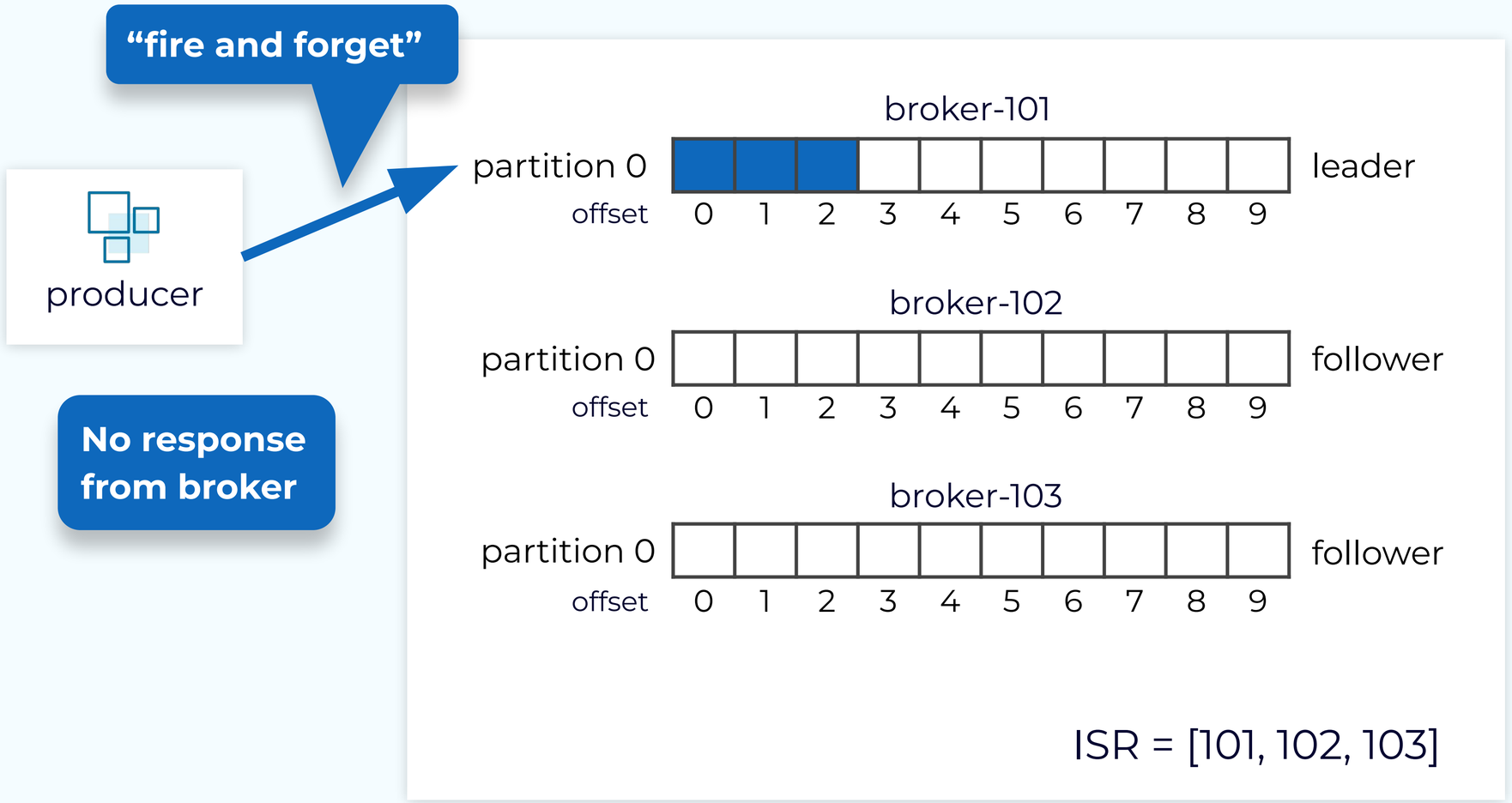
Data Durability and Availability Guarantees

CONFLUENT Developer



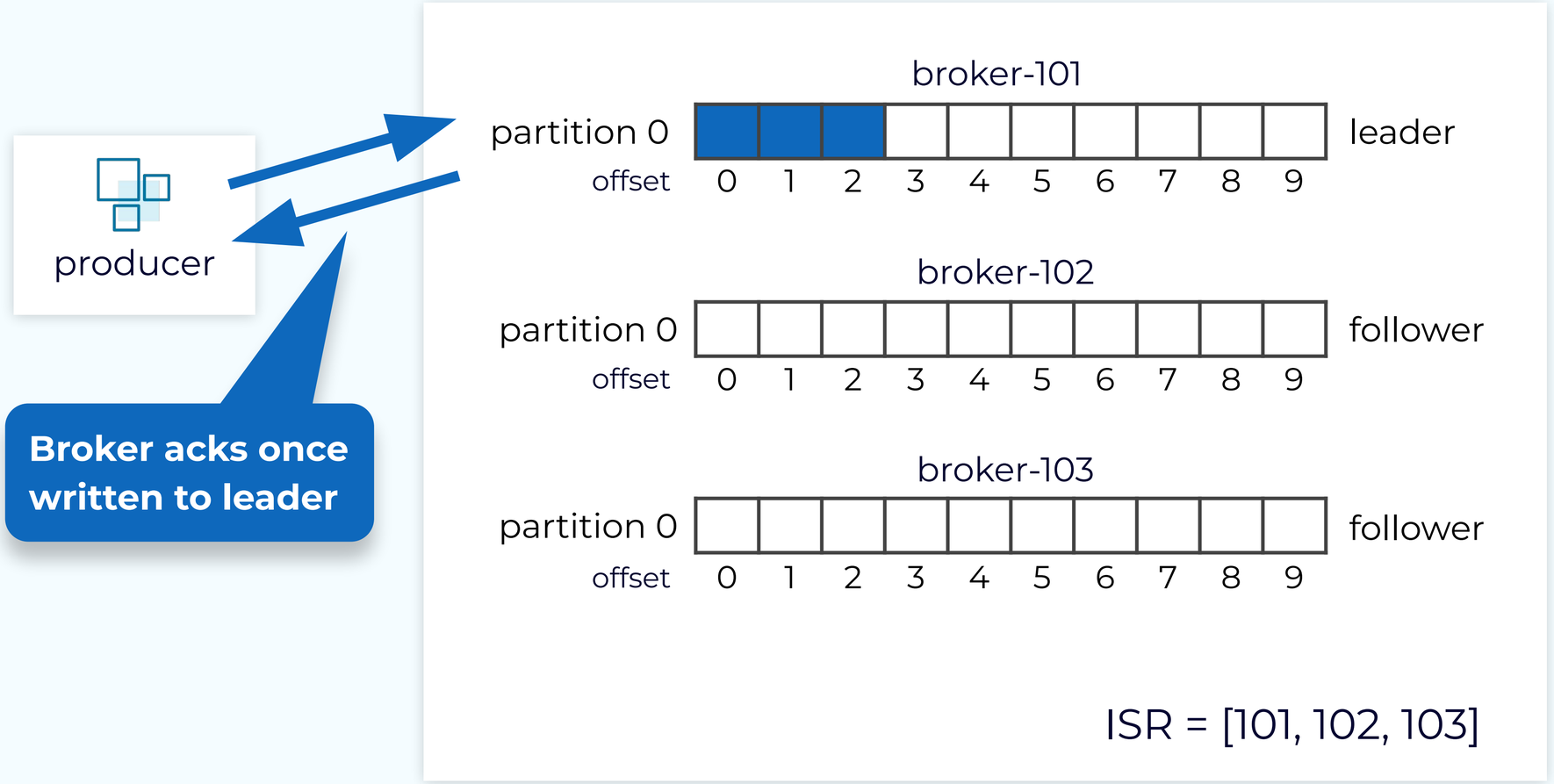
Producer `acks=0`

CONFLUENT Developer



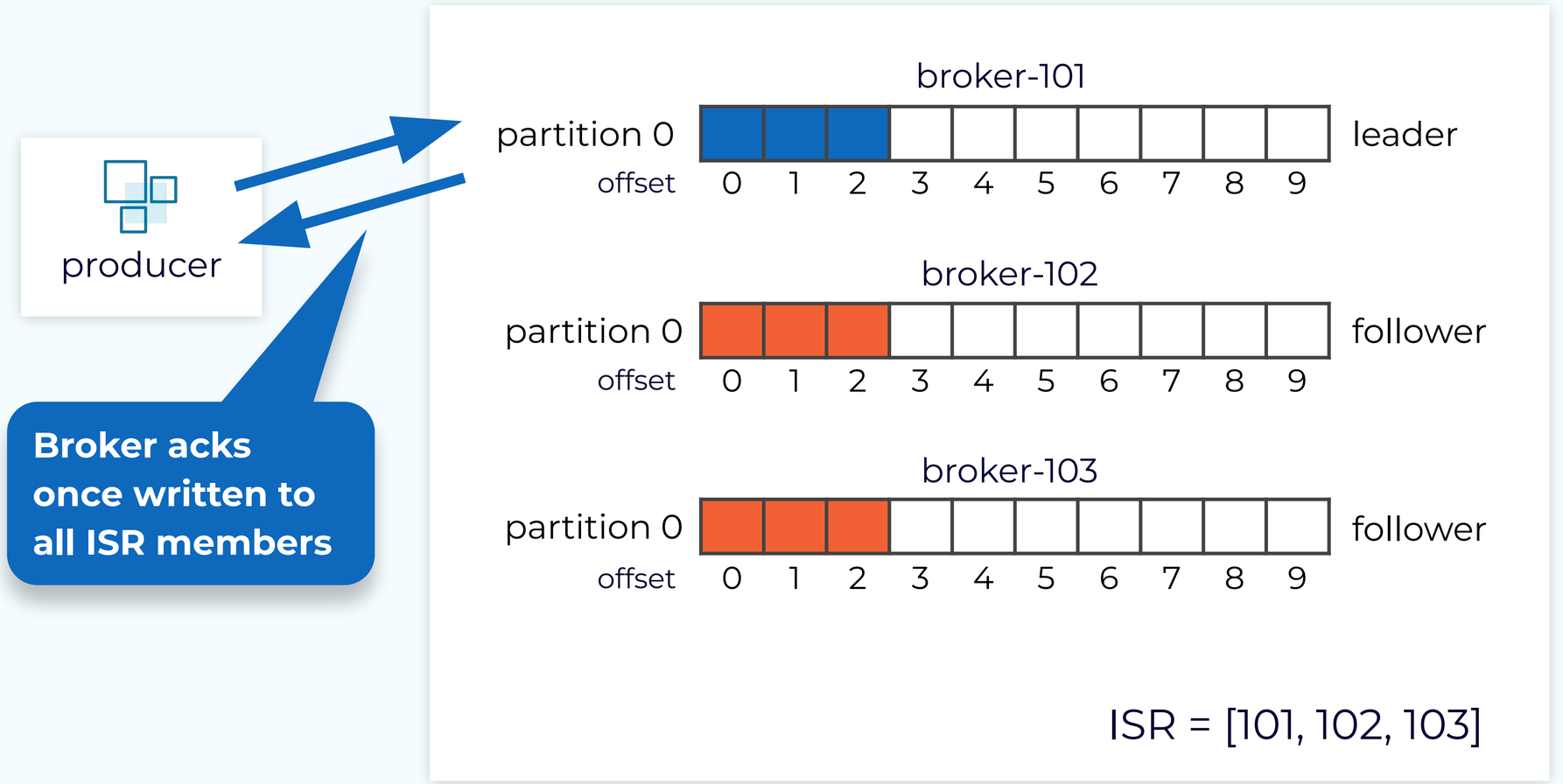
Producer acks=1

CONFLUENT Developer



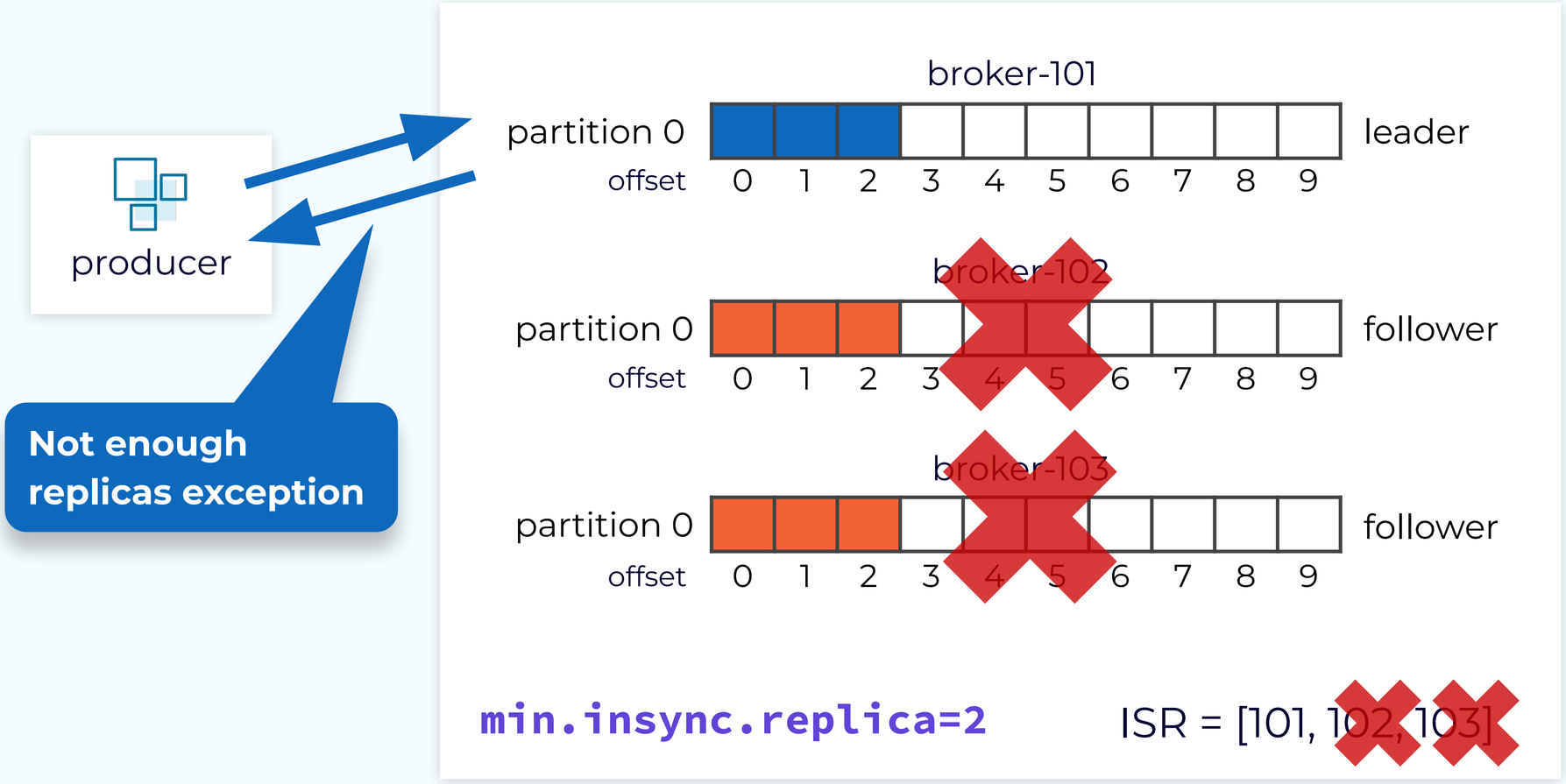
Producer acks=all

CONFLUENT Developer



Topic `min.insync.replicas`

CONFLUENT Developer

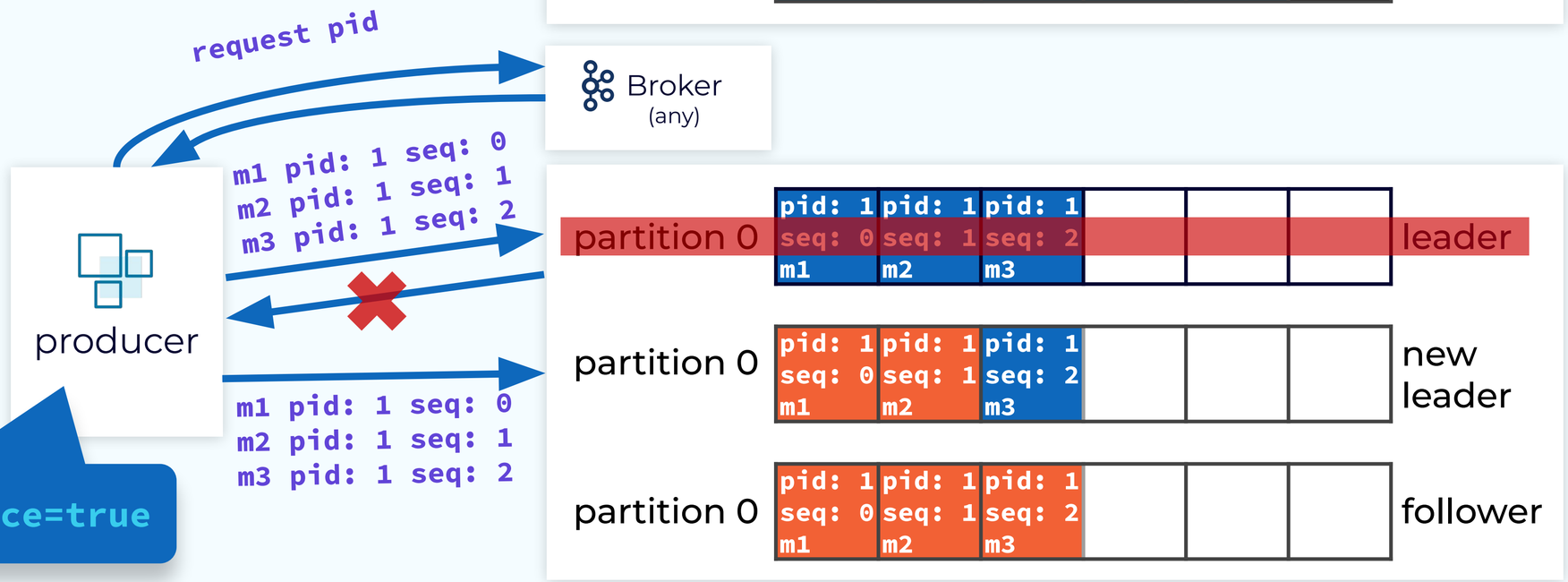


Producer Idempotency

CONFLUENT Developer

Without idempotency, duplicate and out of order records can occur

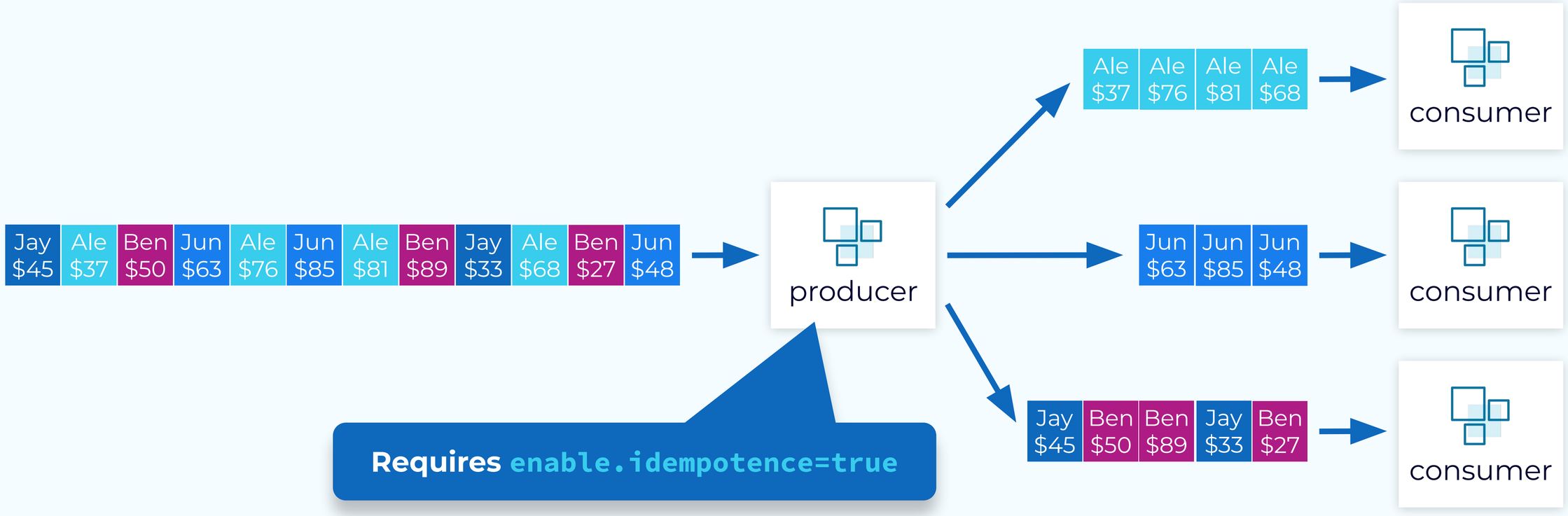
Idempotency's seq #'s prevent both of these issues



`enable.idempotence=true`

End-to-End Ordering Guarantee

CONFLUENT Developer



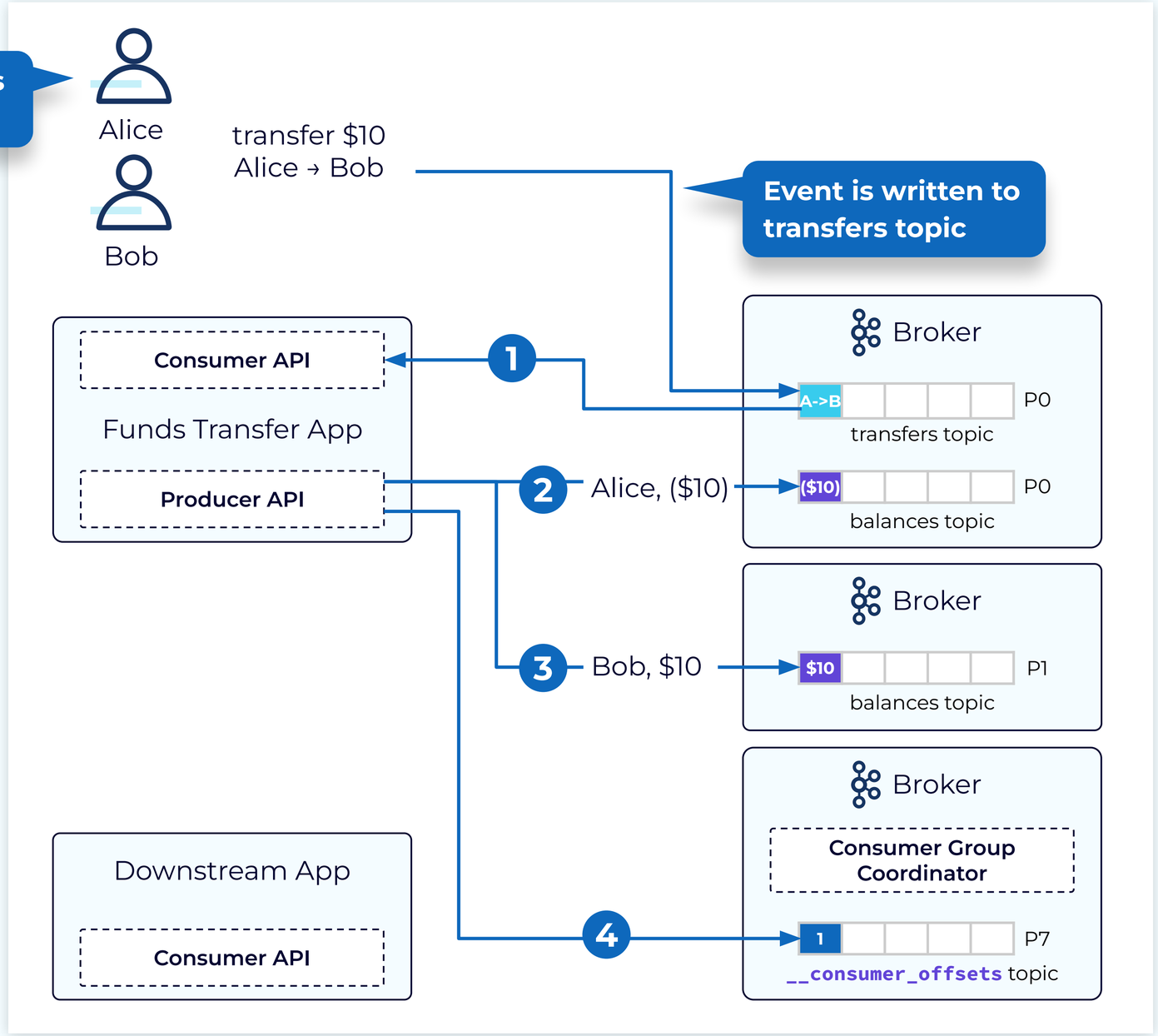
CONFLUENT
Developer

Transactions

Why Are Transactions Needed?

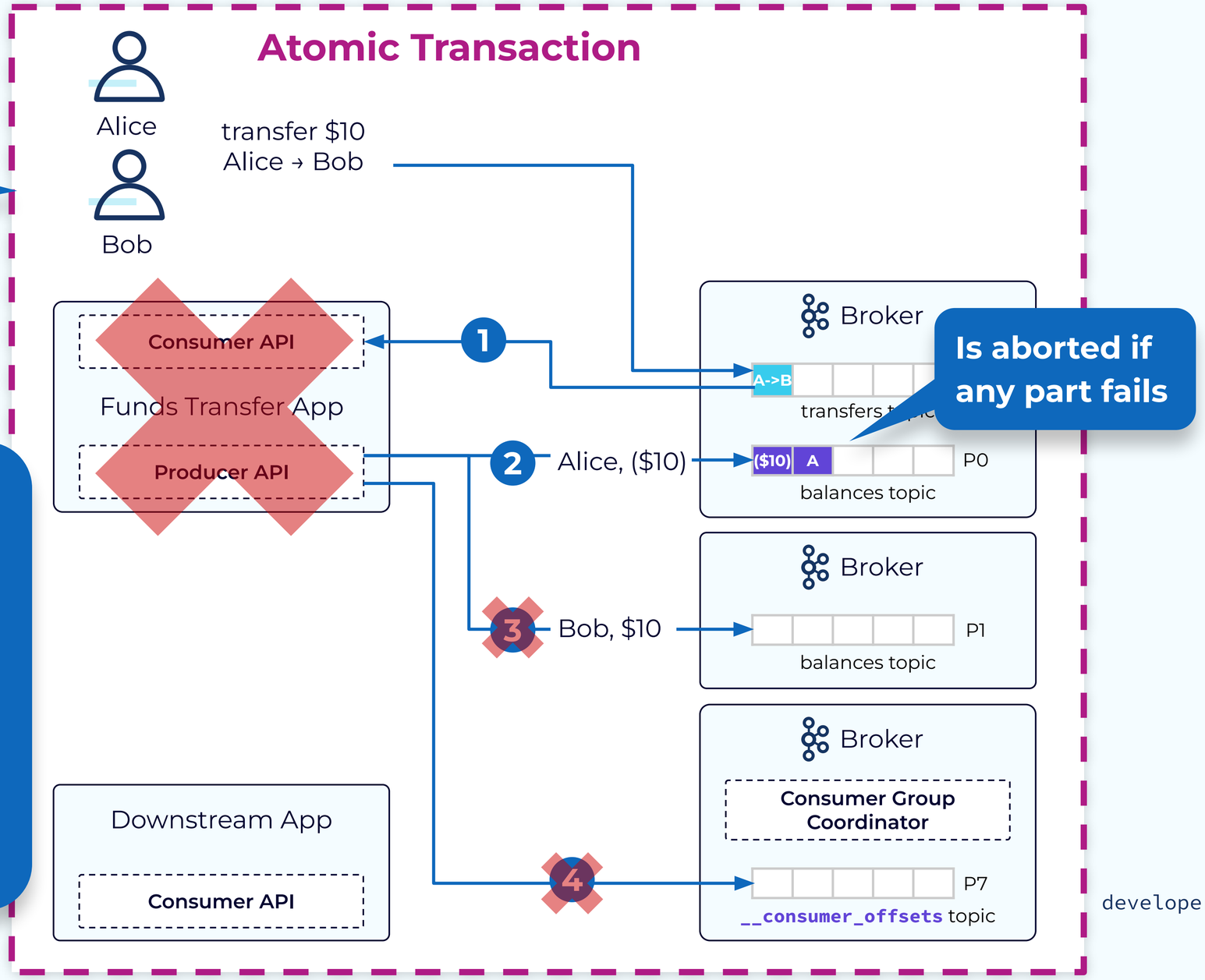
CONFLUENT Developer

1. Event is fetched by the consumer
2. Debit event is written
3. Credit event is written
4. Transfer event offset is committed



Kafka Transactions Deliver Exactly Once

CONFLUENT Developer



Transaction is only committed if all parts succeed

Is aborted if any part fails

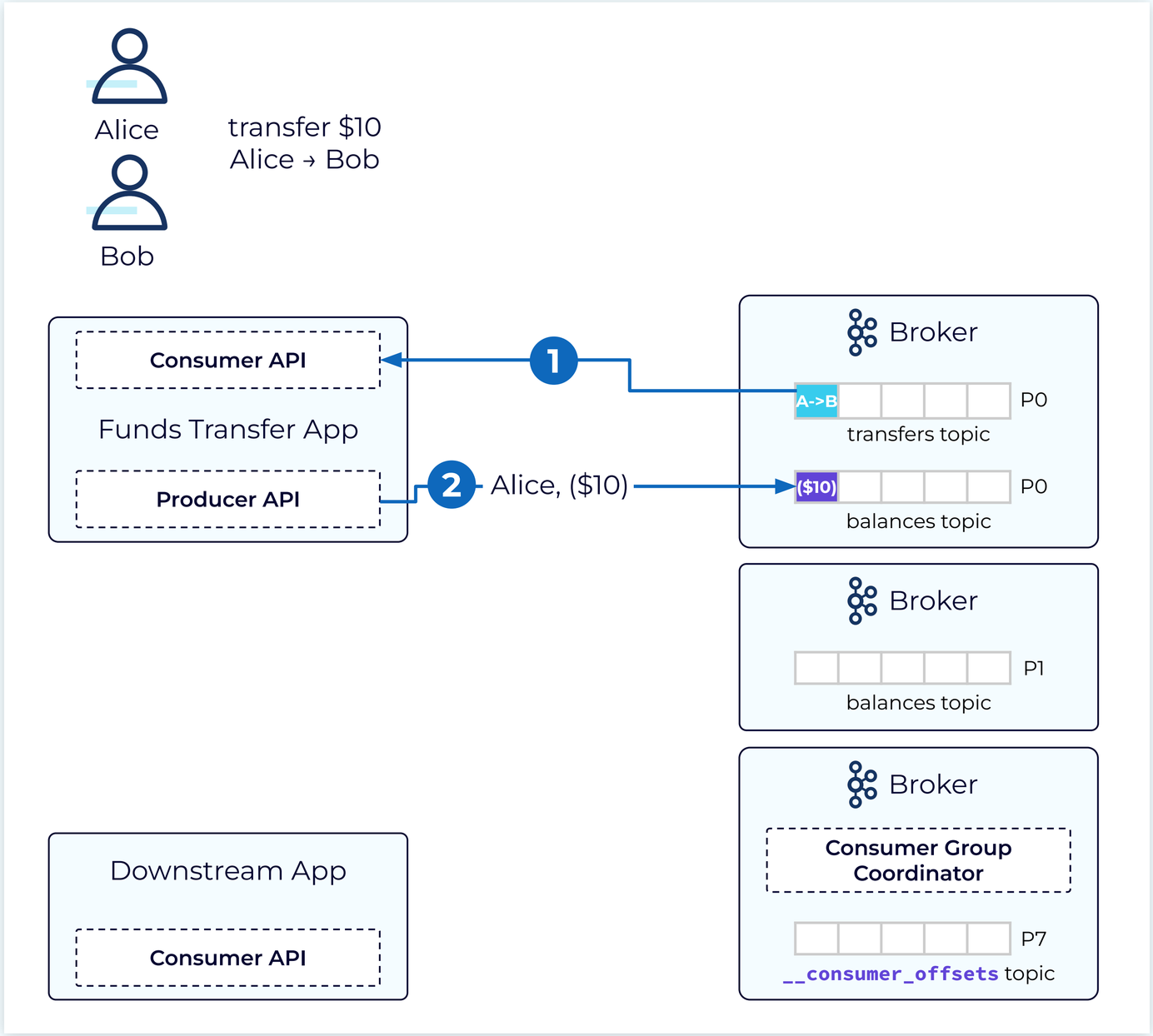
Using transactions with Kafka Streams is quite simple:

1) Set `processing.guarantee` to `exactly_once_v2` in `StreamsConfig`
2) Set `isolation.level` to `read_committed` in the Consumer configuration

System Failure Without Transactions

CONFLUENT Developer

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



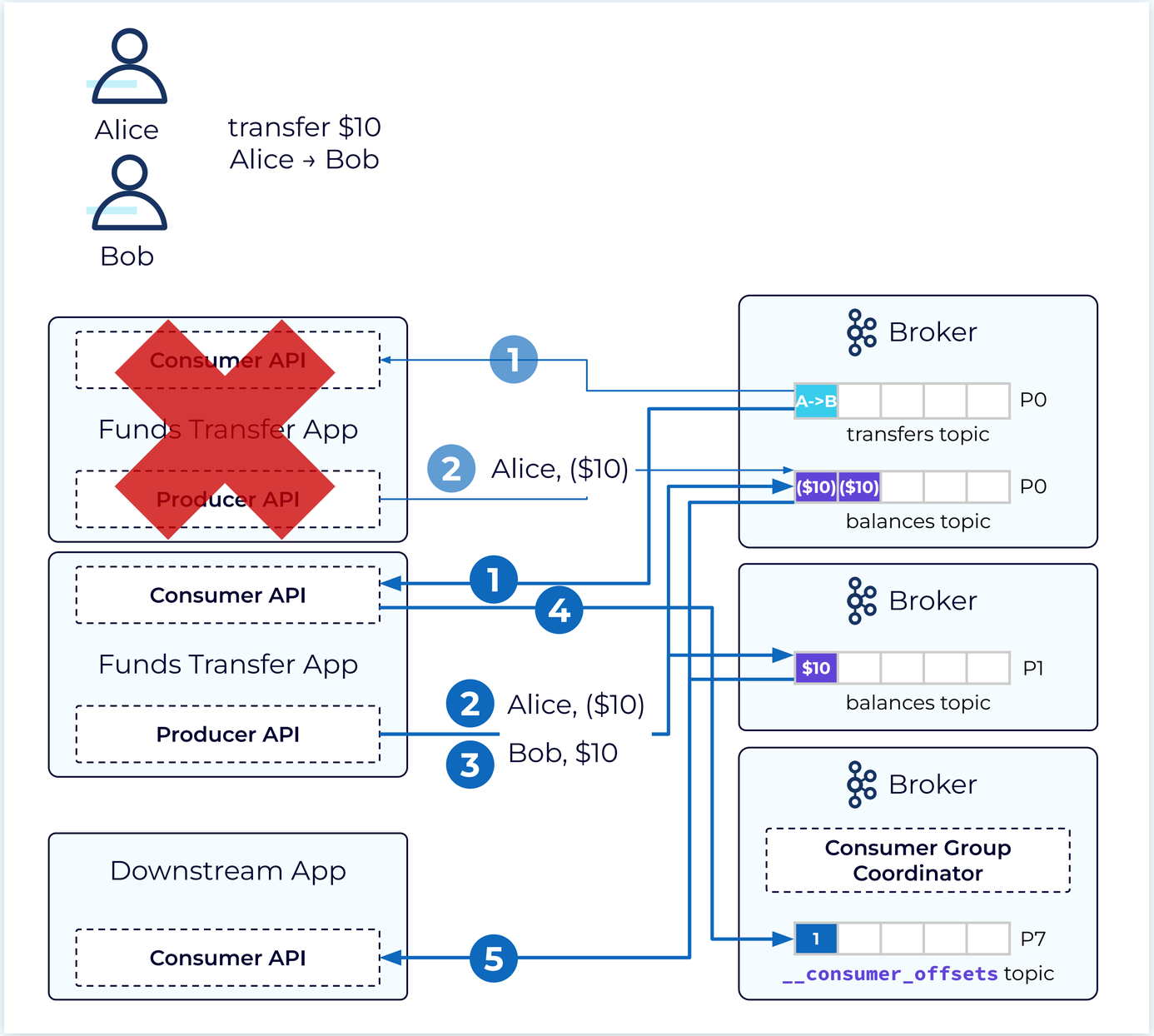
System Failure Without Transactions

CONFLUENT Developer

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

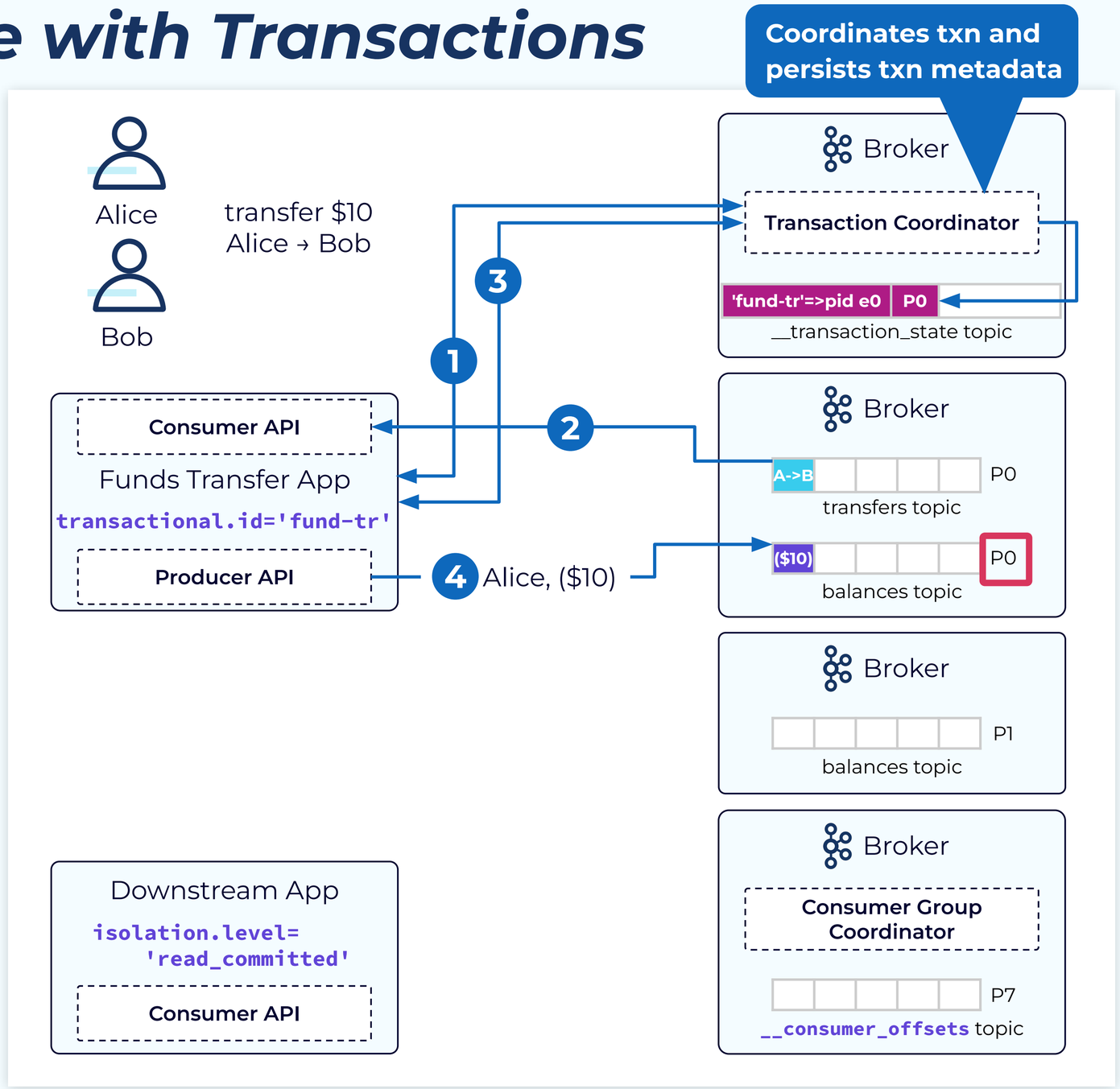
Application instance fails without committing offset and new application instance starts

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



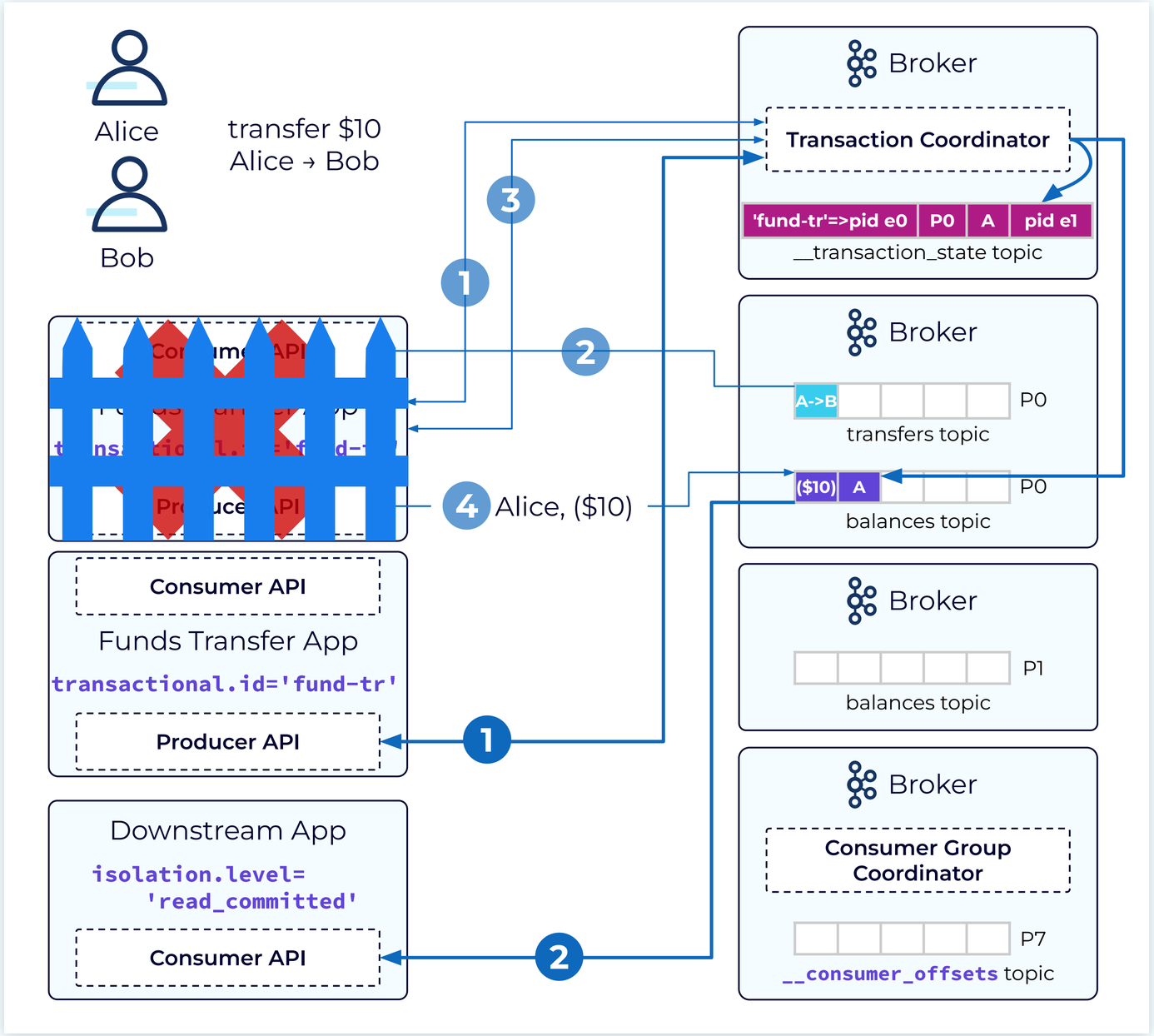
System Failure with Transactions

CONFLUENT Developer

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

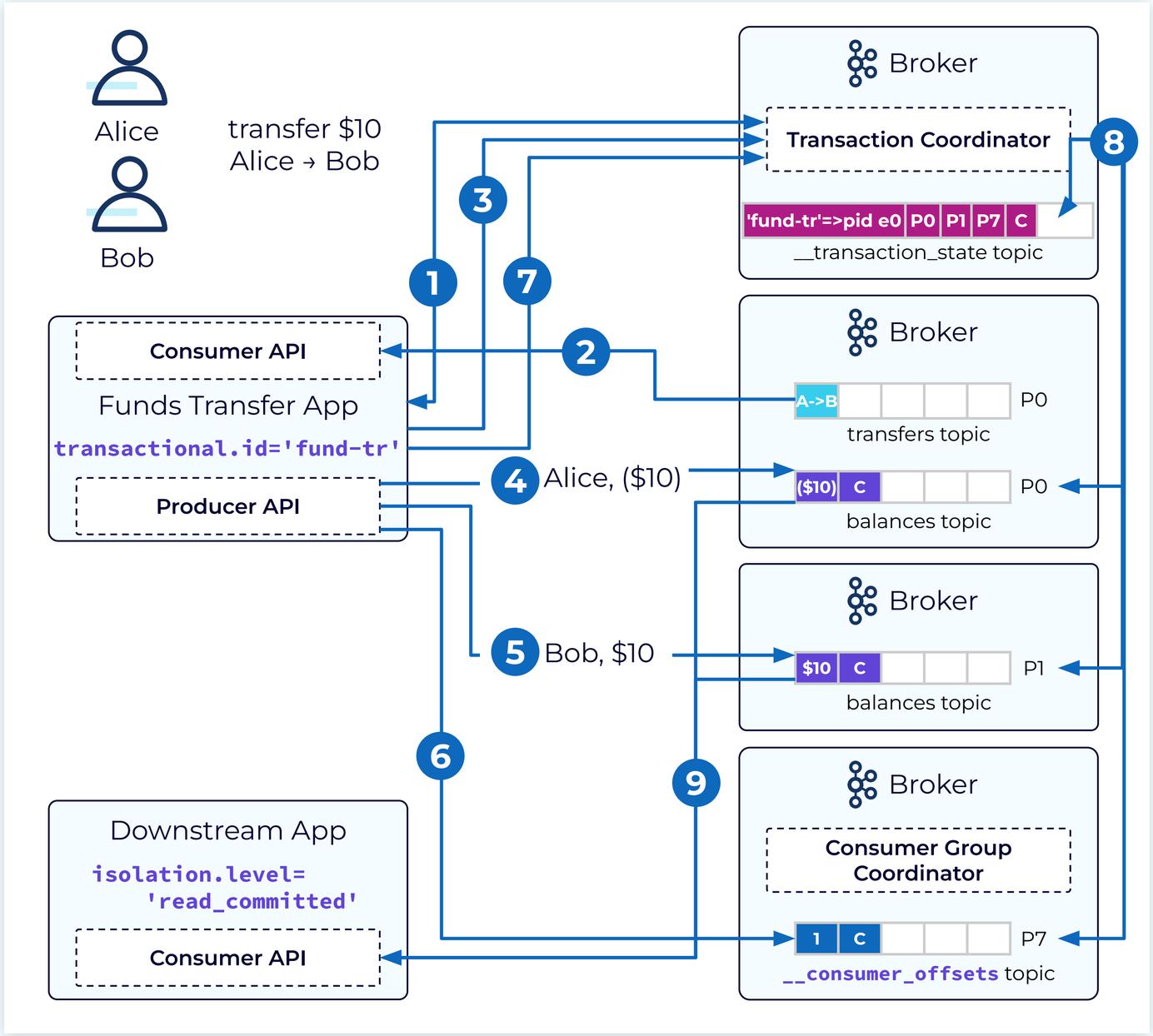
1. **New instance requests txn ID**
   a. **Coordinator fences previous instance by aborting pending txn and bumping up epoch**
2. **Downstream consumer with `read_committed` discards aborted events**



System with Successful Committed Transaction

CONFLUENT Developer

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

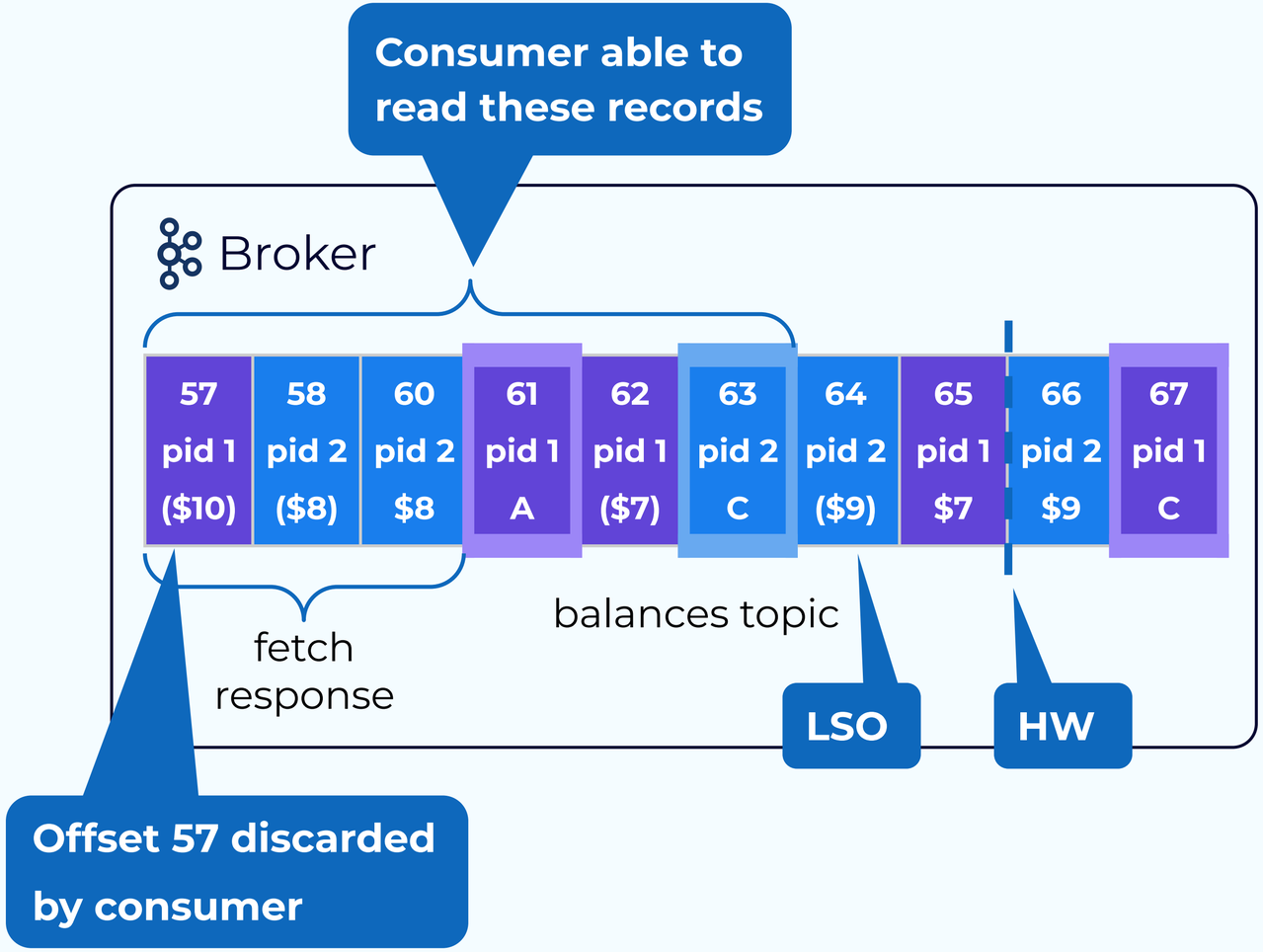


developer.confluent.io

Consuming Transactions with `read_committed`

CONFLUENT Developer

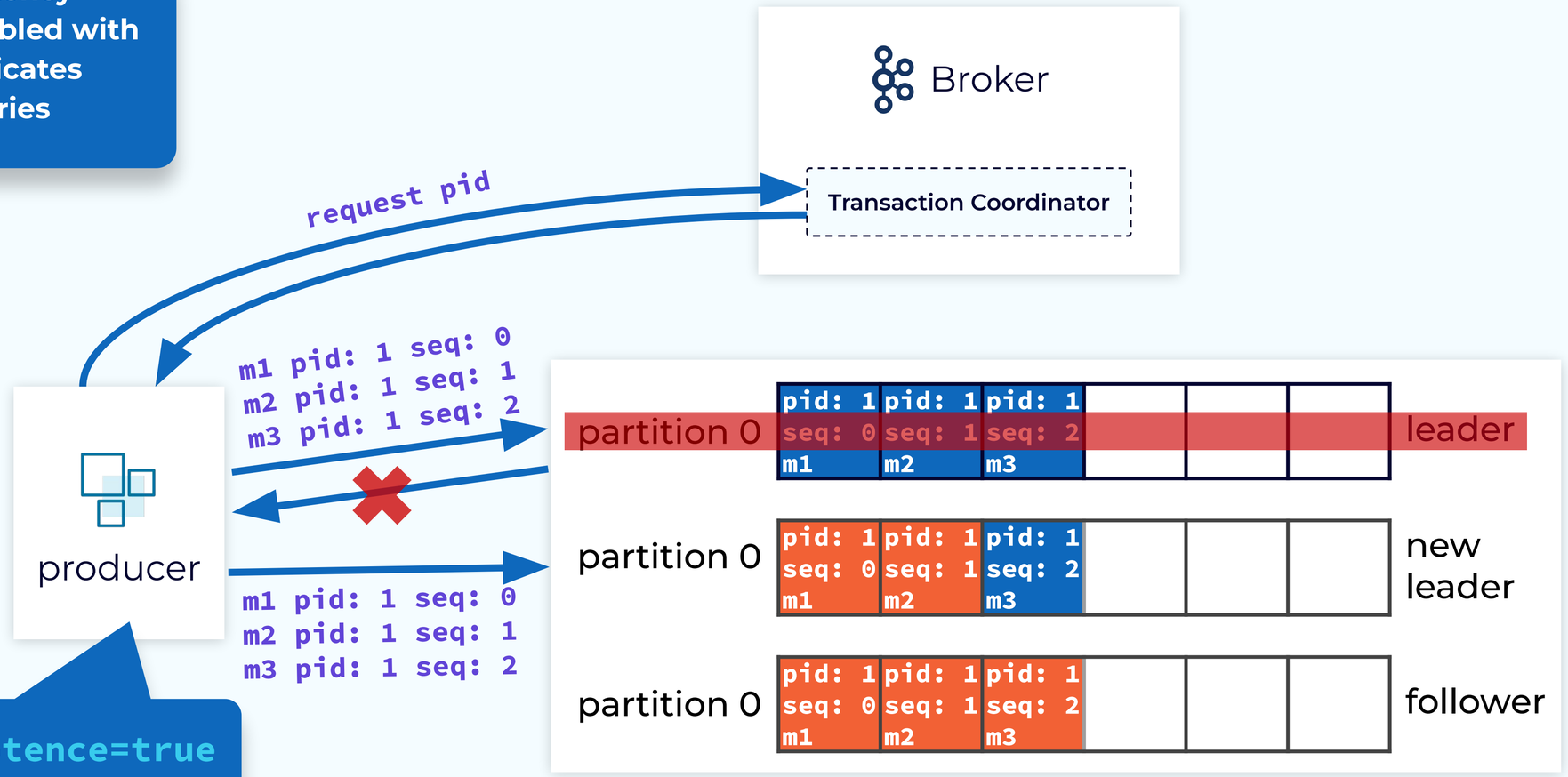
- Leader maintains last stable offset (LSO), the smallest offset of any open transaction
- Fetch response includes
  - only records up to LSO
  - metadata for skipping aborted records



Transactions: Producer Idempotency

CONFLUENT Developer

Producer idempotency automatically enabled with EOS to avoid duplicates from producer retries



Transactions: Balance Overhead with Latency

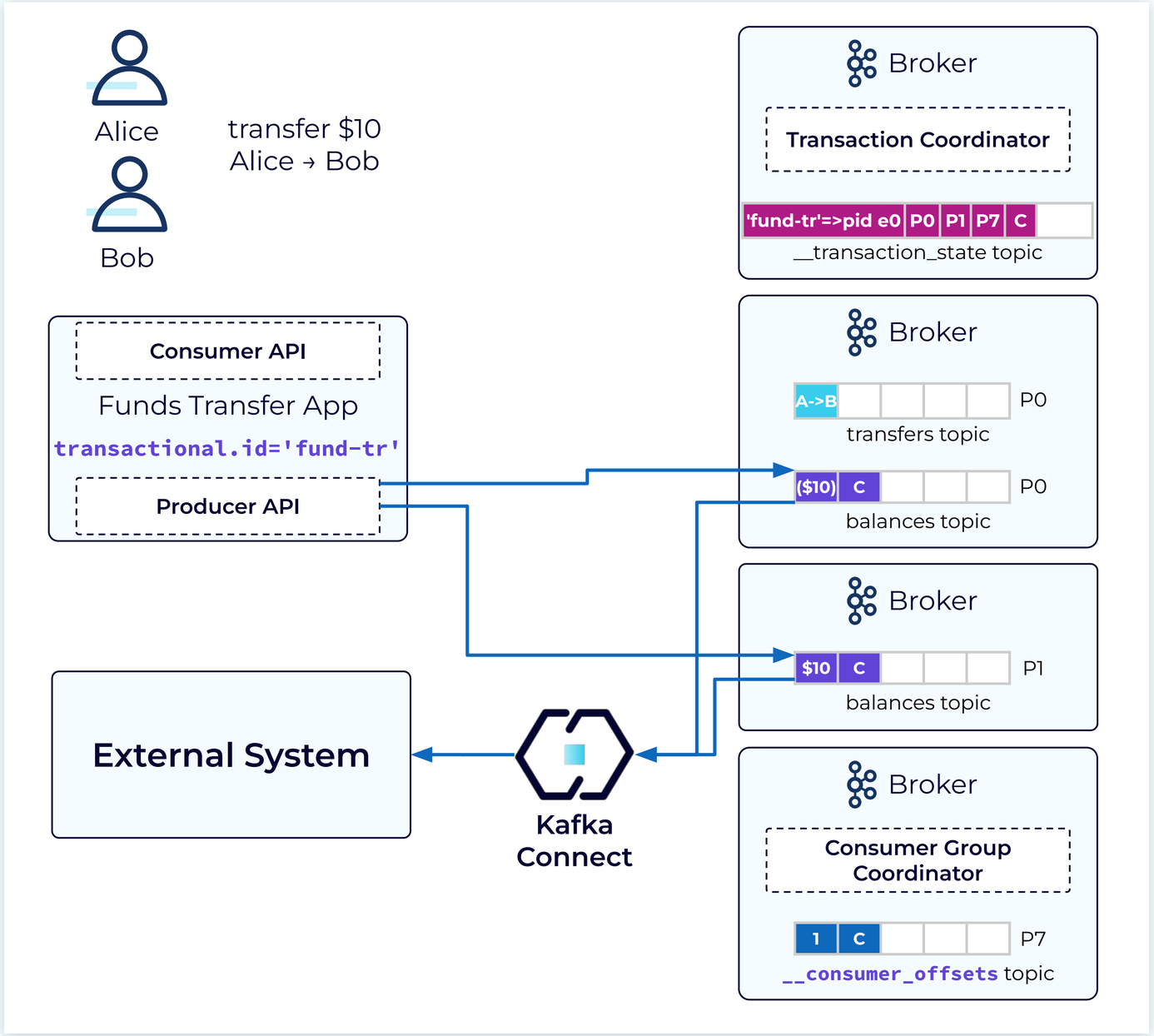
- Too few records per txn adds noticeable overhead
- Too many records per txn delays the exposure of output
- Controllable through `commit.interval.ms` in KStreams

Interacting with External Systems

CONFLUENT Developer

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**

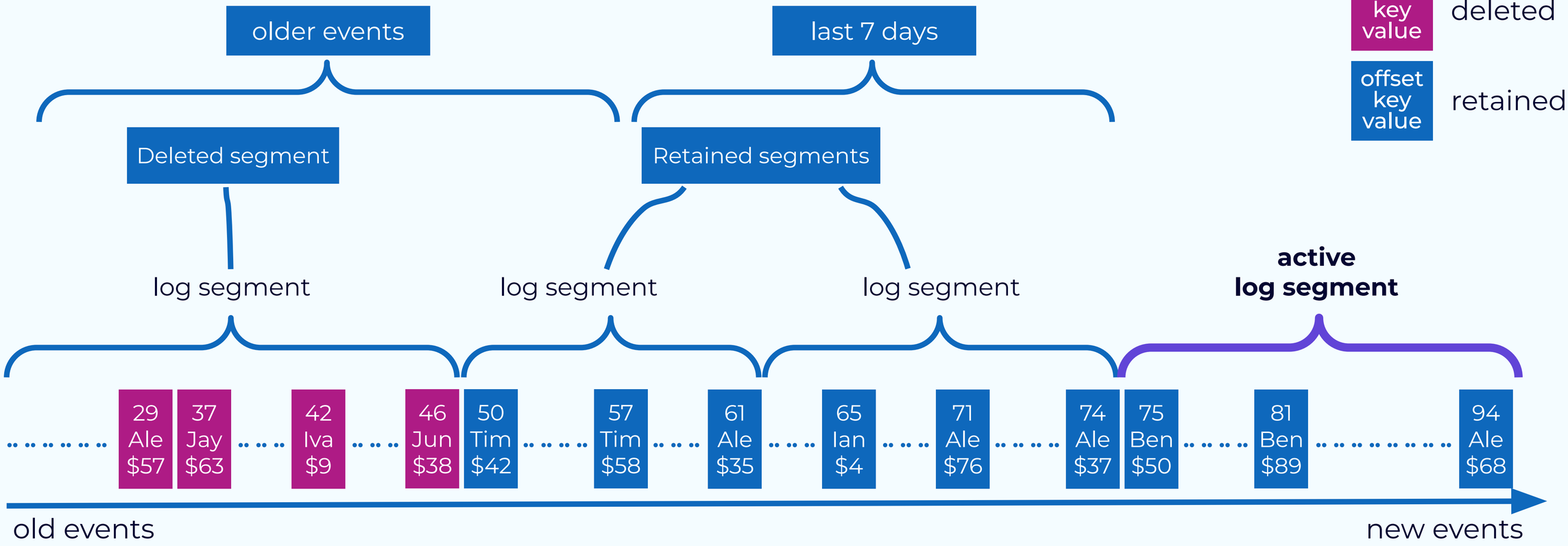


CONFLUENT
Developer

Topic Compaction

Time-Based Retention

CONFLUENT Developer



Time based retention applied at the log segment level of granularity

```
cleanup.policy=delete
retention.ms=60480000
(7 days default)
```

Topic Compaction: Key-Based Retention

CONFLUENT Developer

- Most recent occurrence per key is kept
- All other occurrences marked for deletion over time

Null value indicates deletion of key

Retention policy `cleanup.policy=compact`

offset key value deleted

offset key value retained

Usage and Benefits of Topic Compaction

CONFLUENT Developer

Usage:

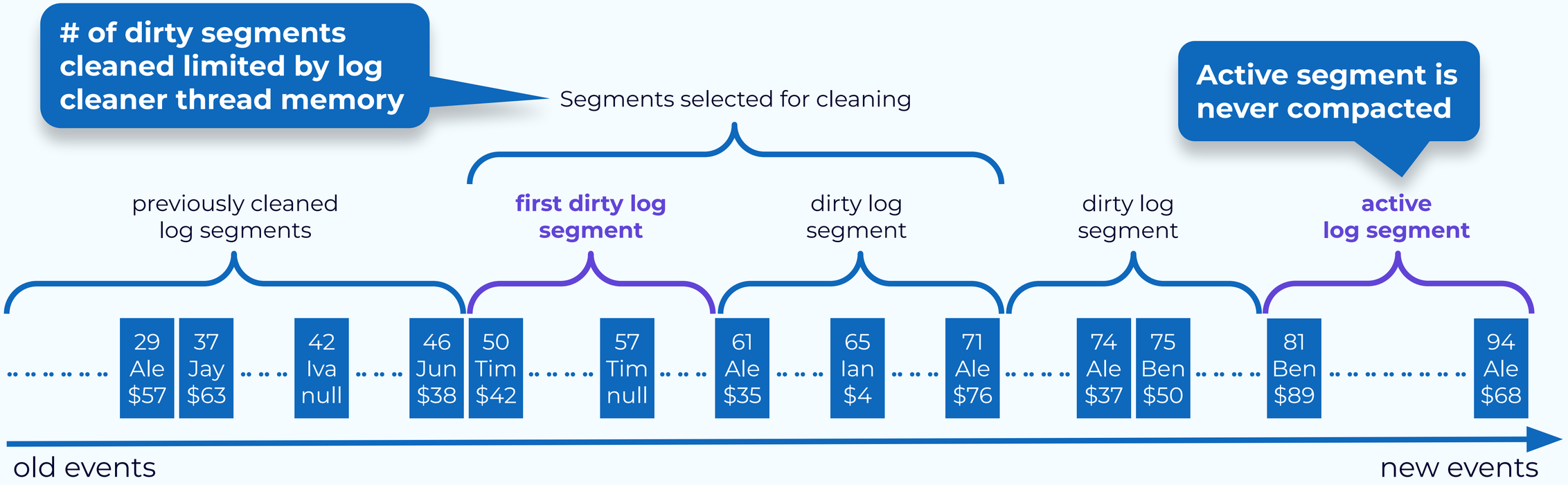
- Streaming updatable data sets (e.g. profiles, catalogue)
- ksqlDB state and tables

Benefits:

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

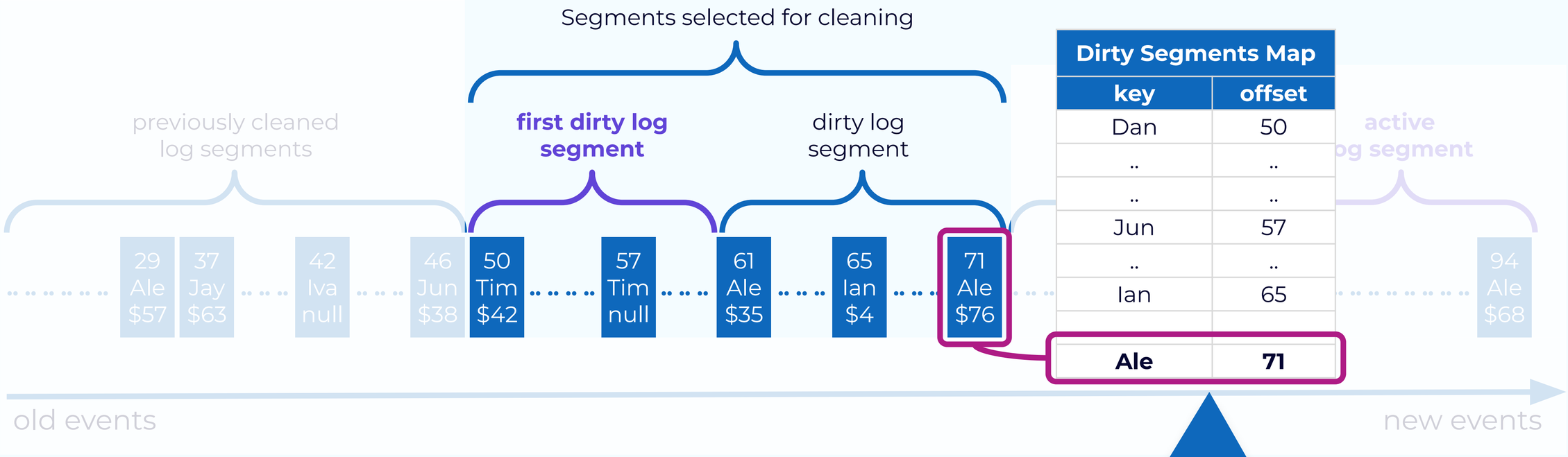
Compaction Process - Segments to Clean

CONFLUENT Developer



Compaction Process - Build Dirty Segment Map

CONFLUENT Developer



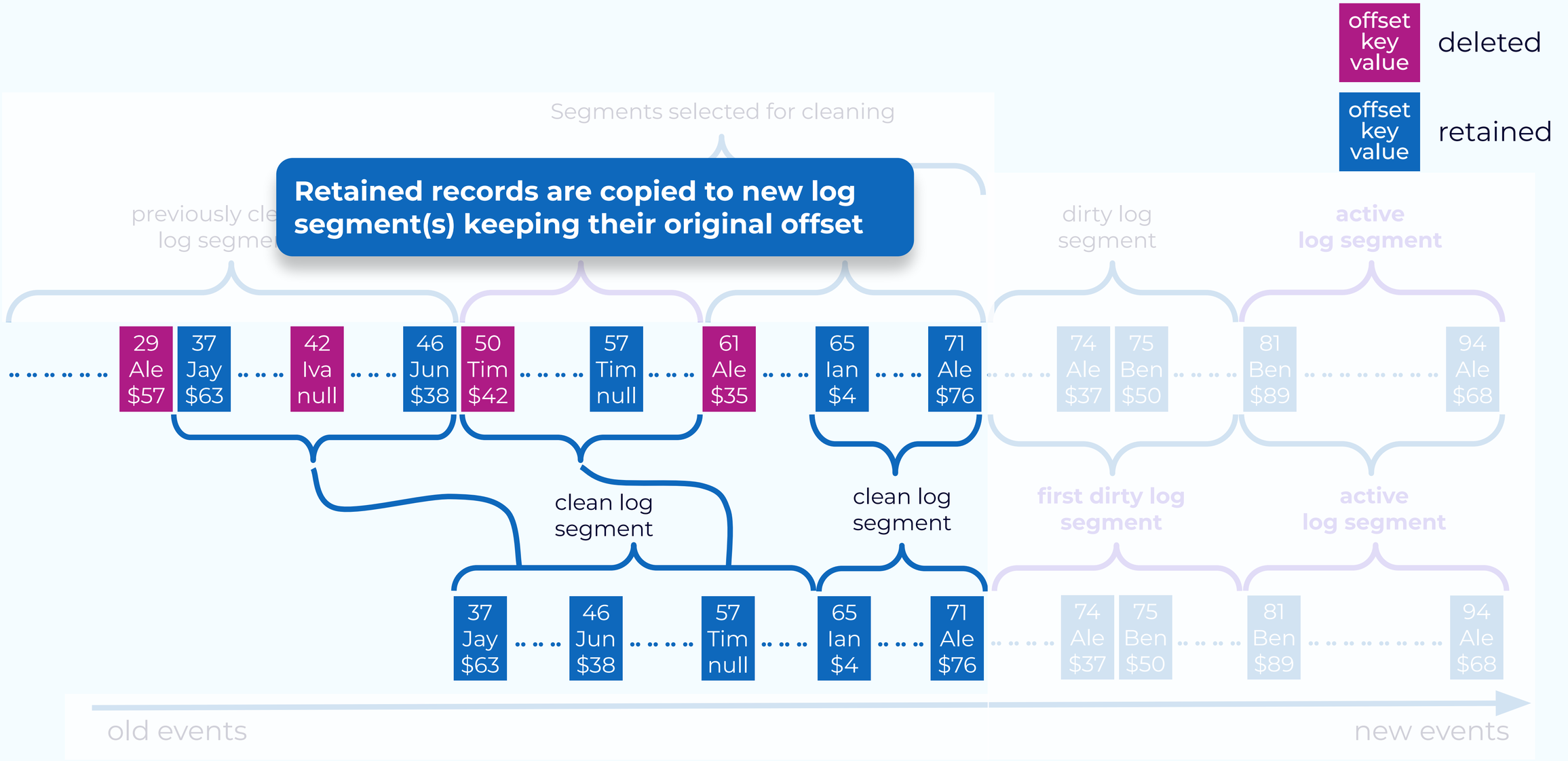
In-memory map of latest offset for each key in selected dirty segments

Compaction Process - Deleting Events

CONFLUENT Developer

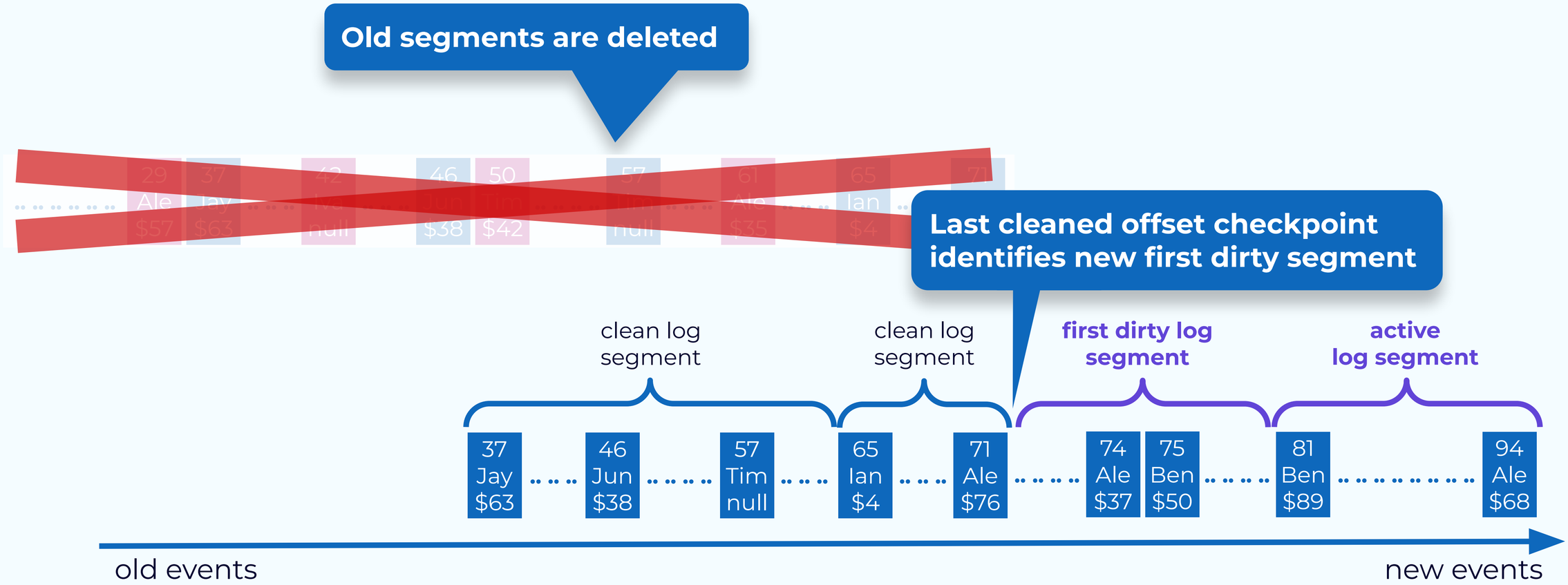
Compaction Process - Retaining Events

CONFLUENT Developer



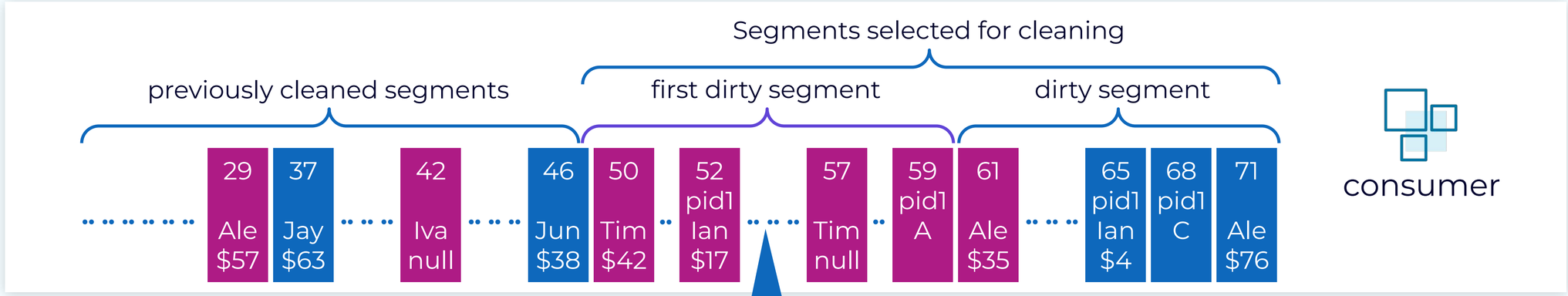
Compaction Process - Replace Old Segments

CONFLUENT Developer



Cleaning Tombstone and Transaction Markers

CONFLUENT Developer



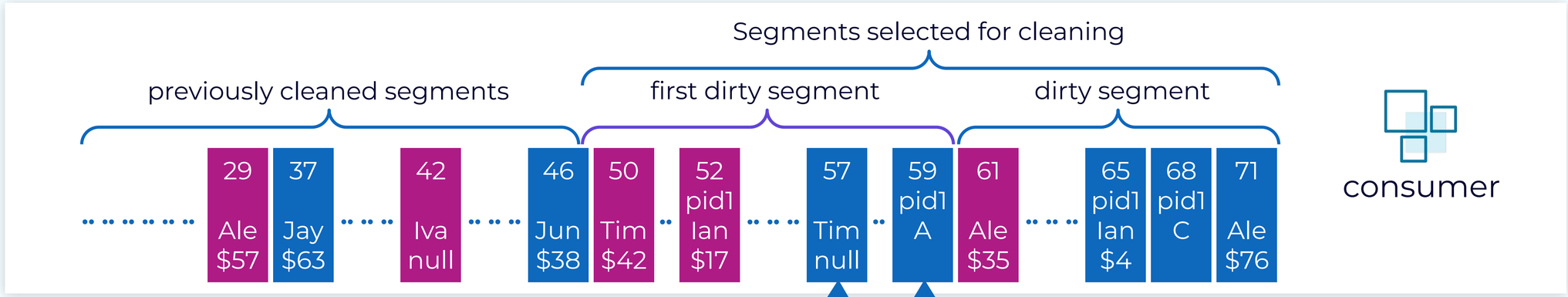
offset key value

deleted

offset key value

retained

Paused consumer with committed offset = 54 misses offset 57 null and 59 txn commit records



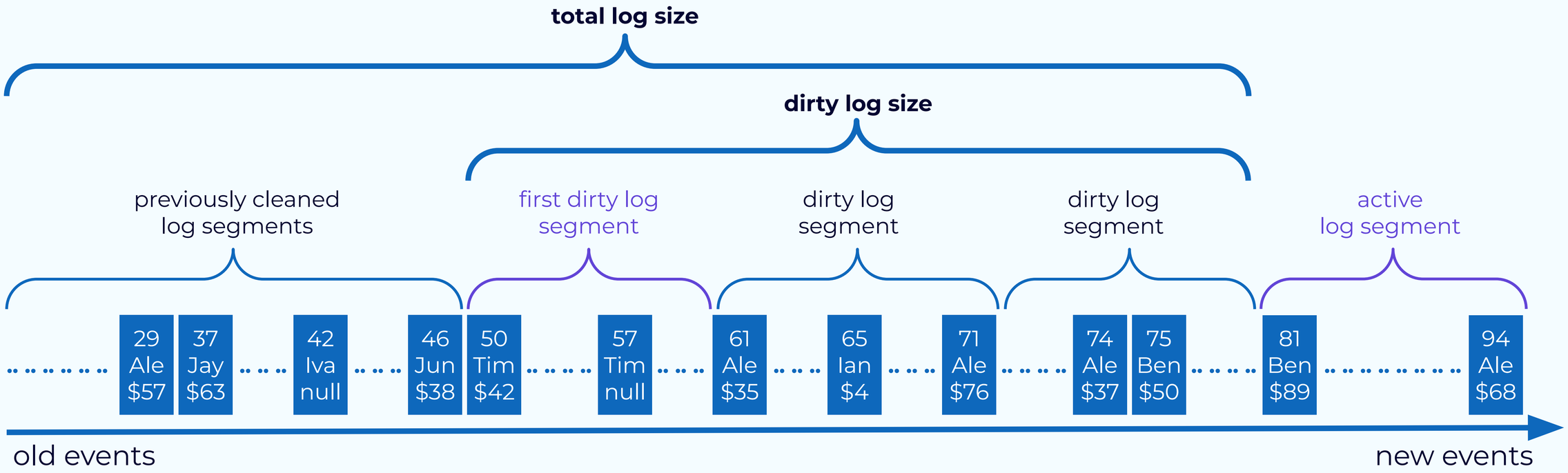
Solution: delay removal of tombstone and txn markers by `first-clean-time + delete.retention.ms`

When Compaction Is Triggered

CONFLUENT Developer

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

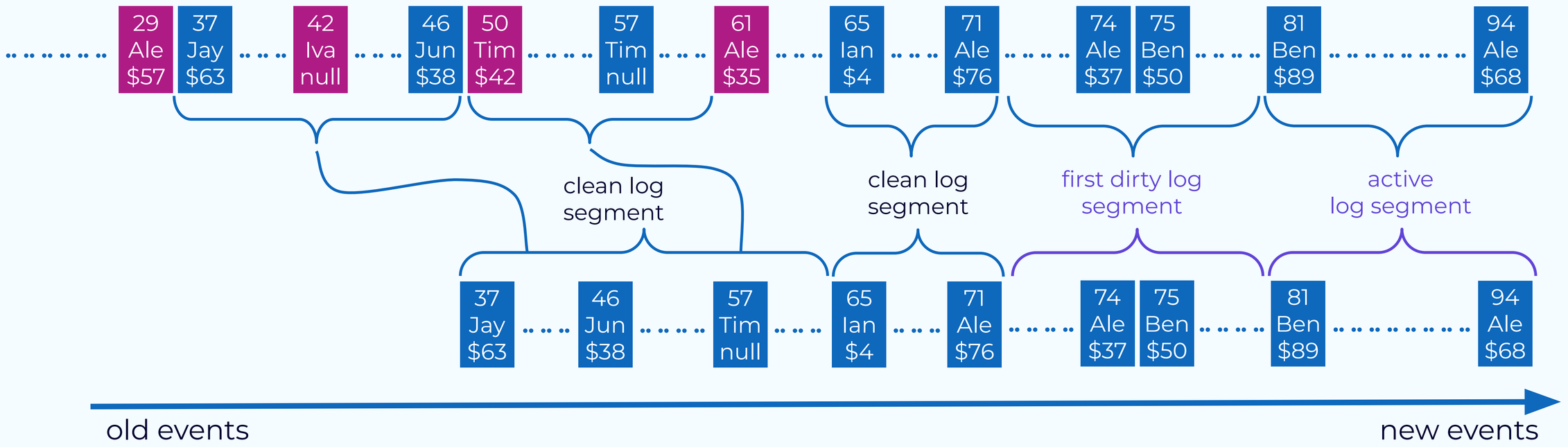
CONFLUENT Developer

offset key value — deleted

offset key value — retained

A consumer:

1) **is not guaranteed to see every record**
2) **is guaranteed to see the latest record for a key**



Topic Compaction

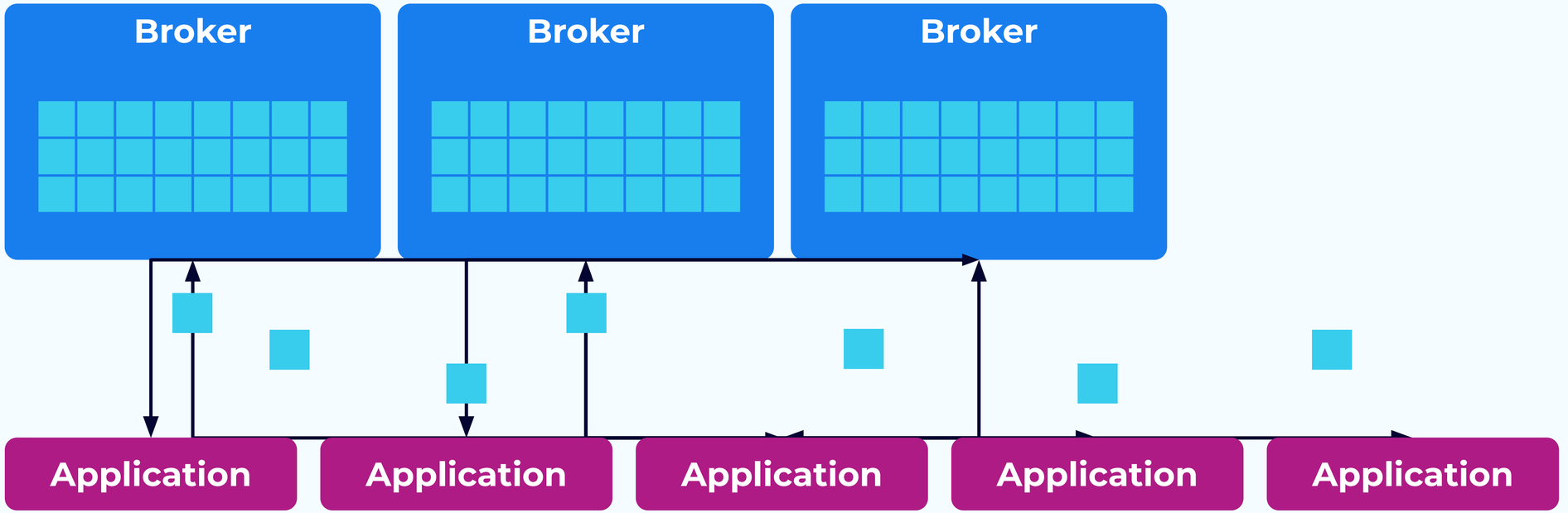
CONFLUENT
Developer

CONFLUENT
Developer

Tiered Storage

Current Kafka Storage and Its Shortcomings

CONFLUENT Developer



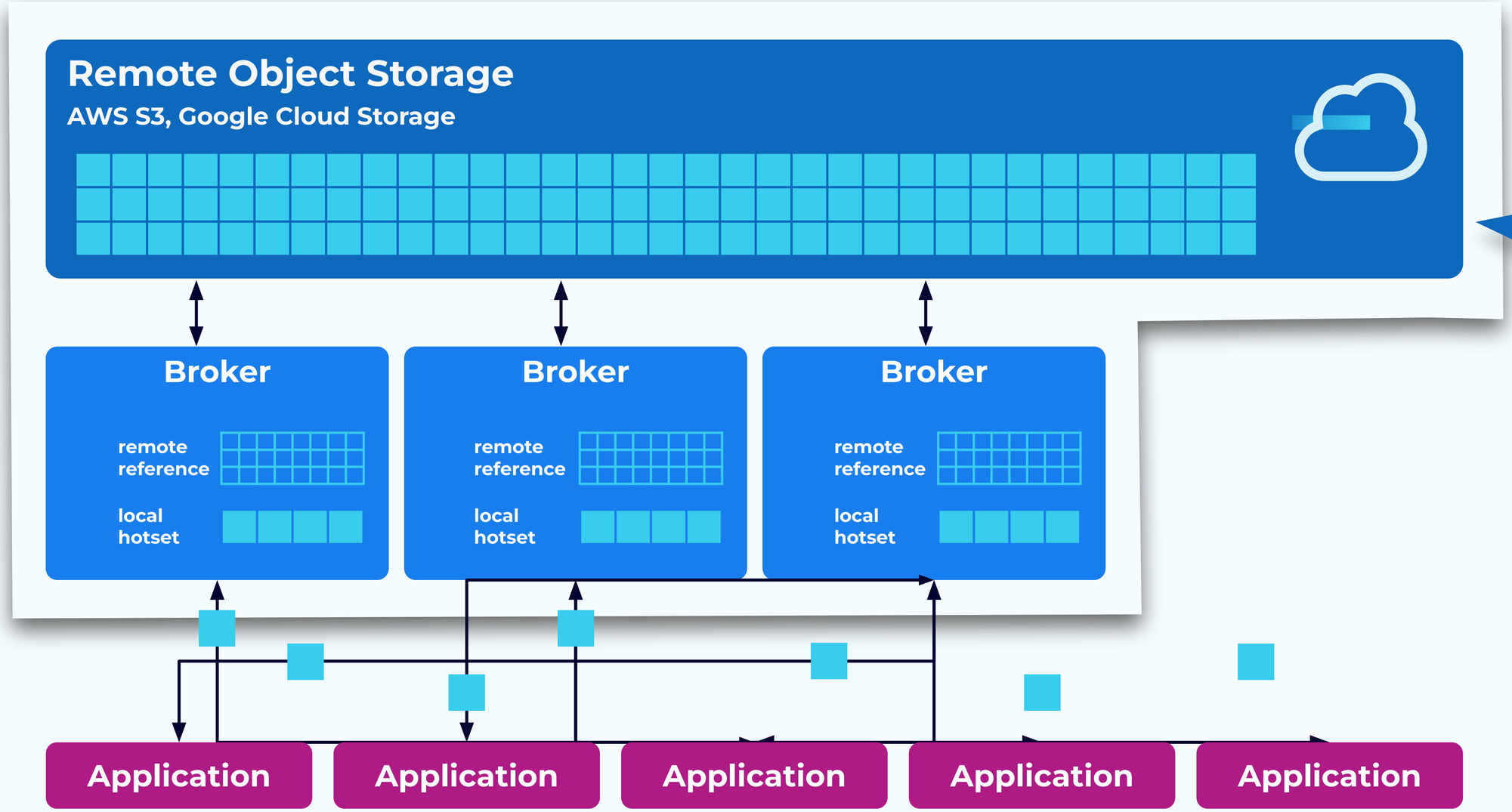
Cost

Elasticity

Isolation

Tiered Storage

CONFLUENT Developer



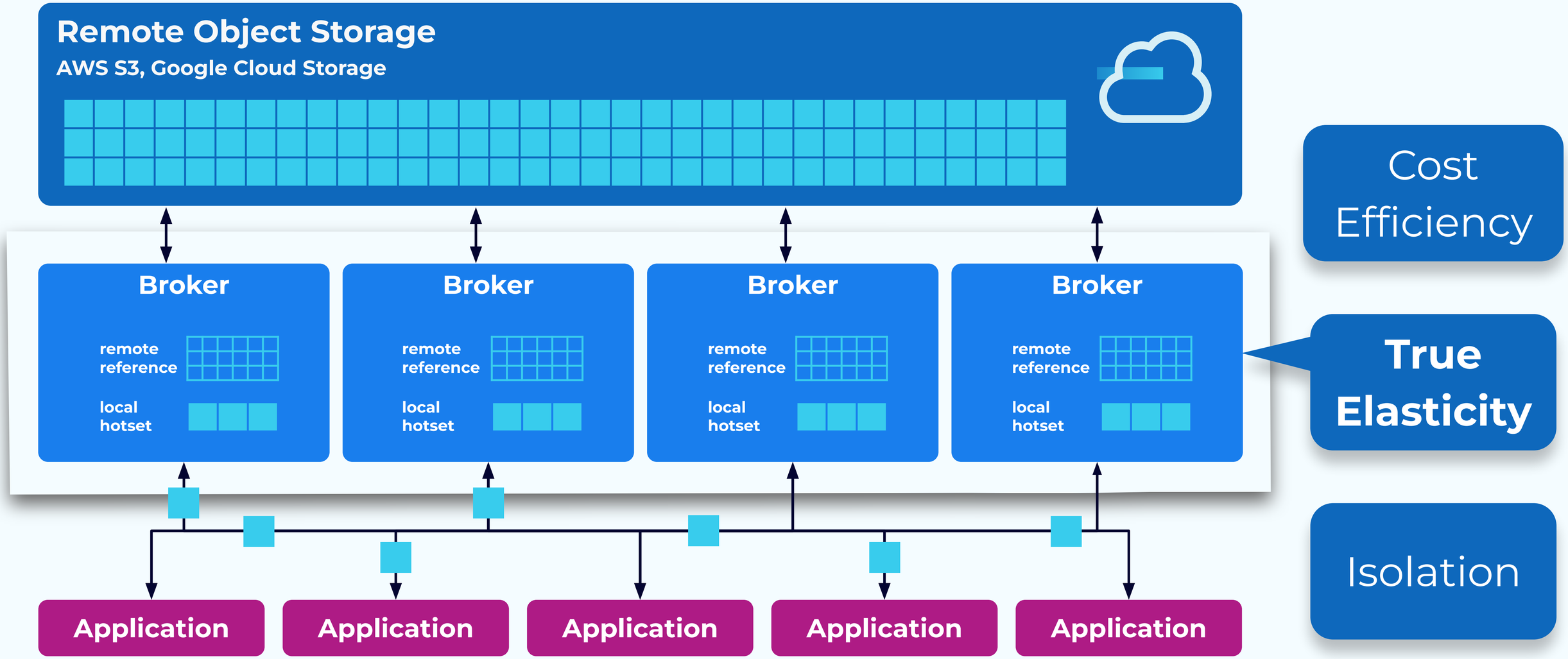
Cost Efficiency

Elasticity

Isolation

Tiered Storage

CONFLUENT Developer



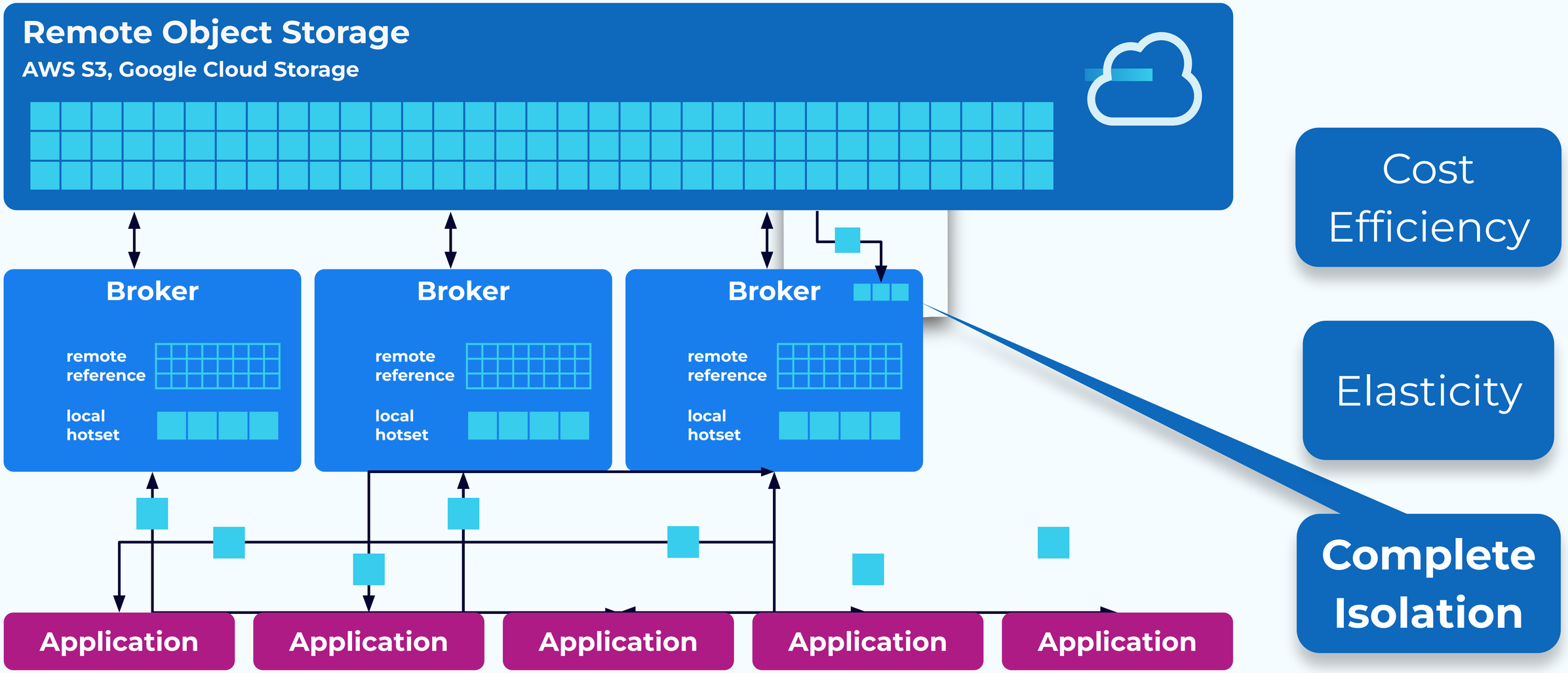
Cost Efficiency

True Elasticity

Isolation

Tiered Storage

CONFLUENT Developer



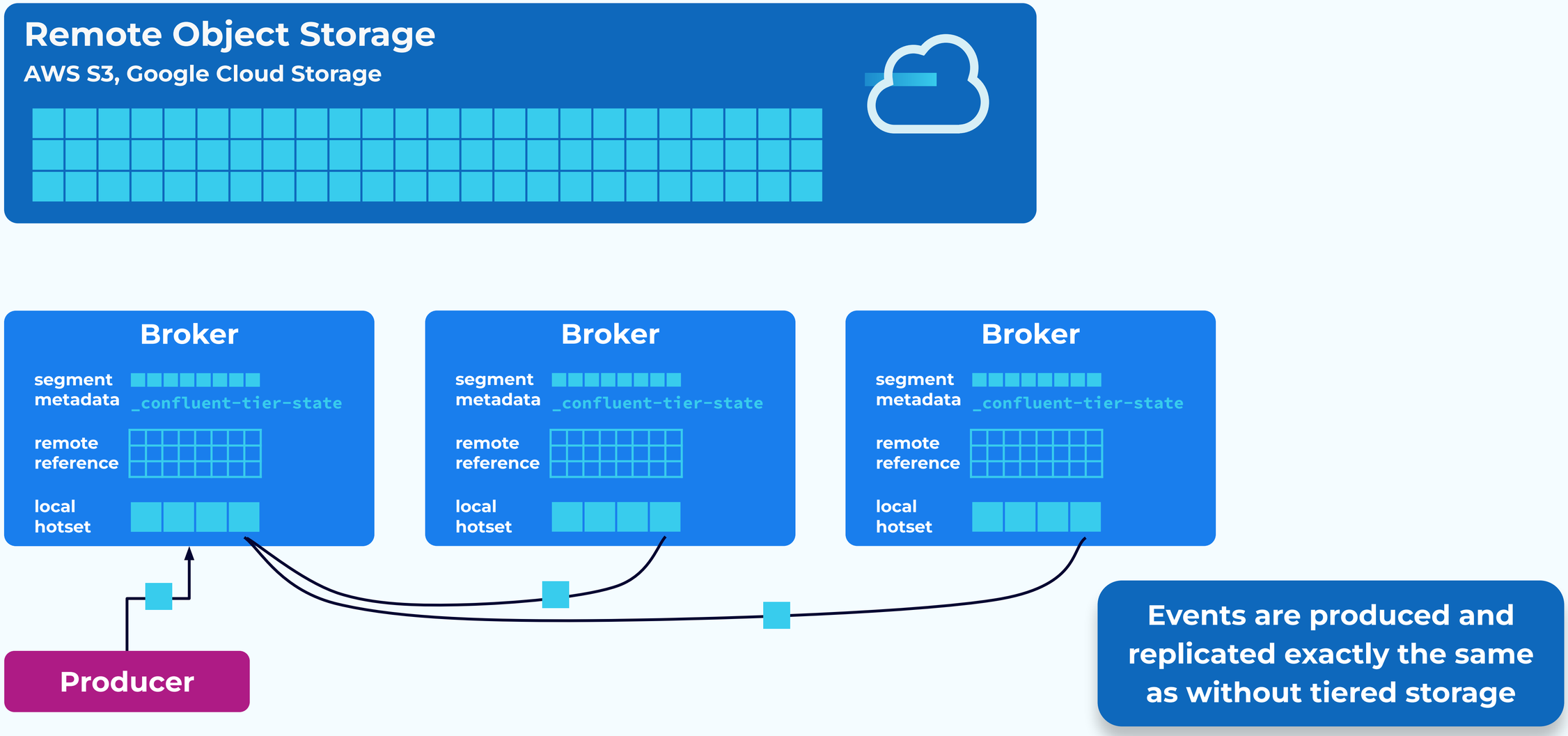
Cost Efficiency

Elasticity

Complete Isolation

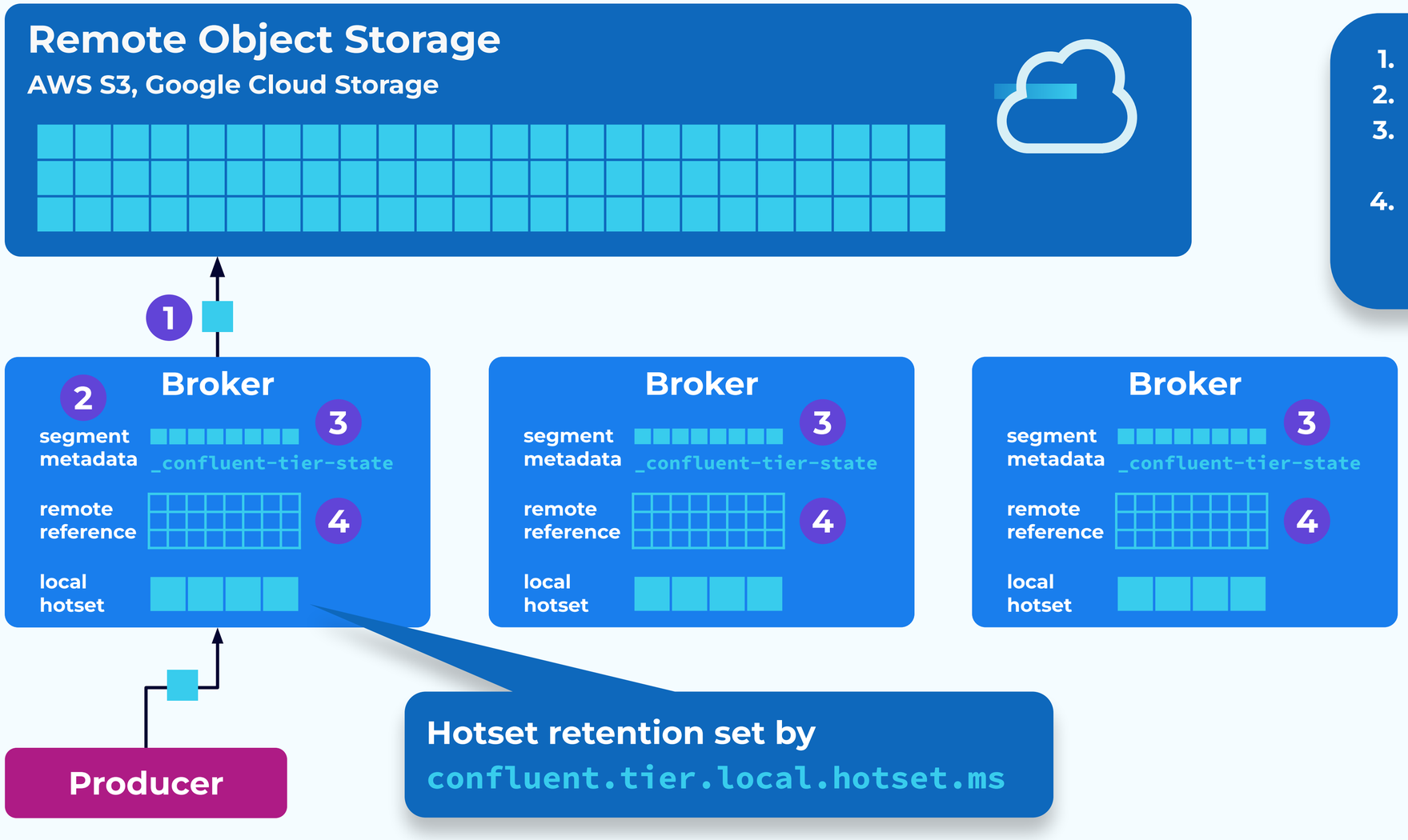
Writing Events to a Tiered Topic

CONFLUENT Developer



Tiering Events to the Remote Object Store

CONFLUENT Developer

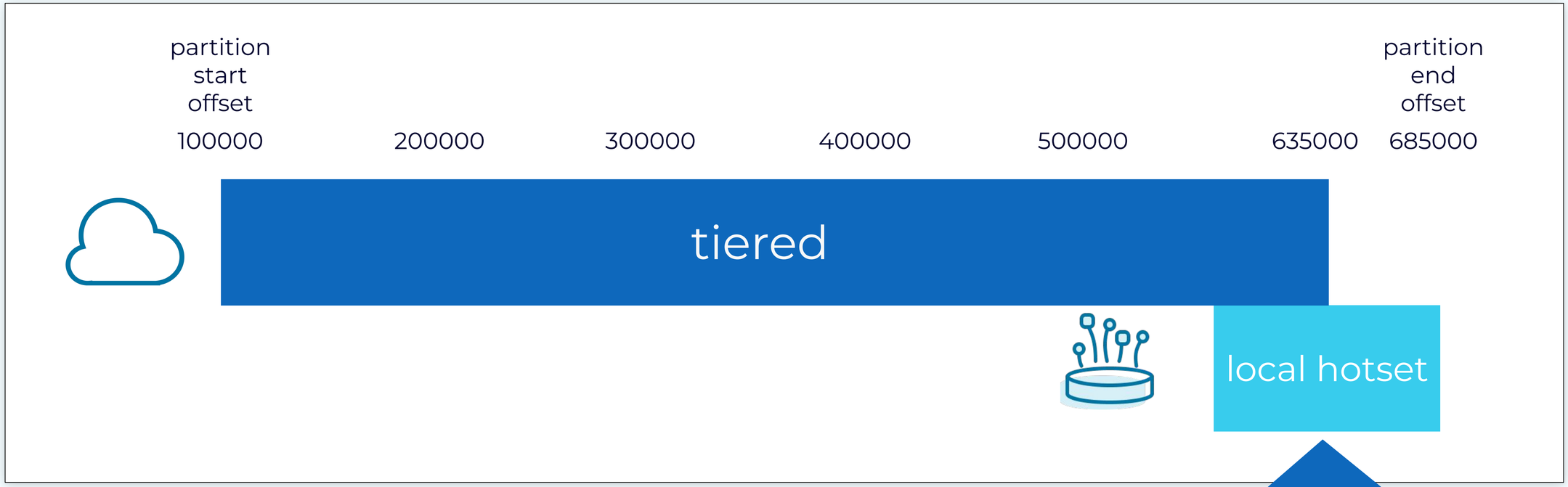


1. Archive segment
2. Write segment metadata
3. Segment metadata is fetched by brokers
4. Brokers materialize segment metadata and build remote log view

Broker Logical View of Tiered Partition

CONFLUENT Developer

Logical view of log based upon metadata contained in `_confluent-tier-state`

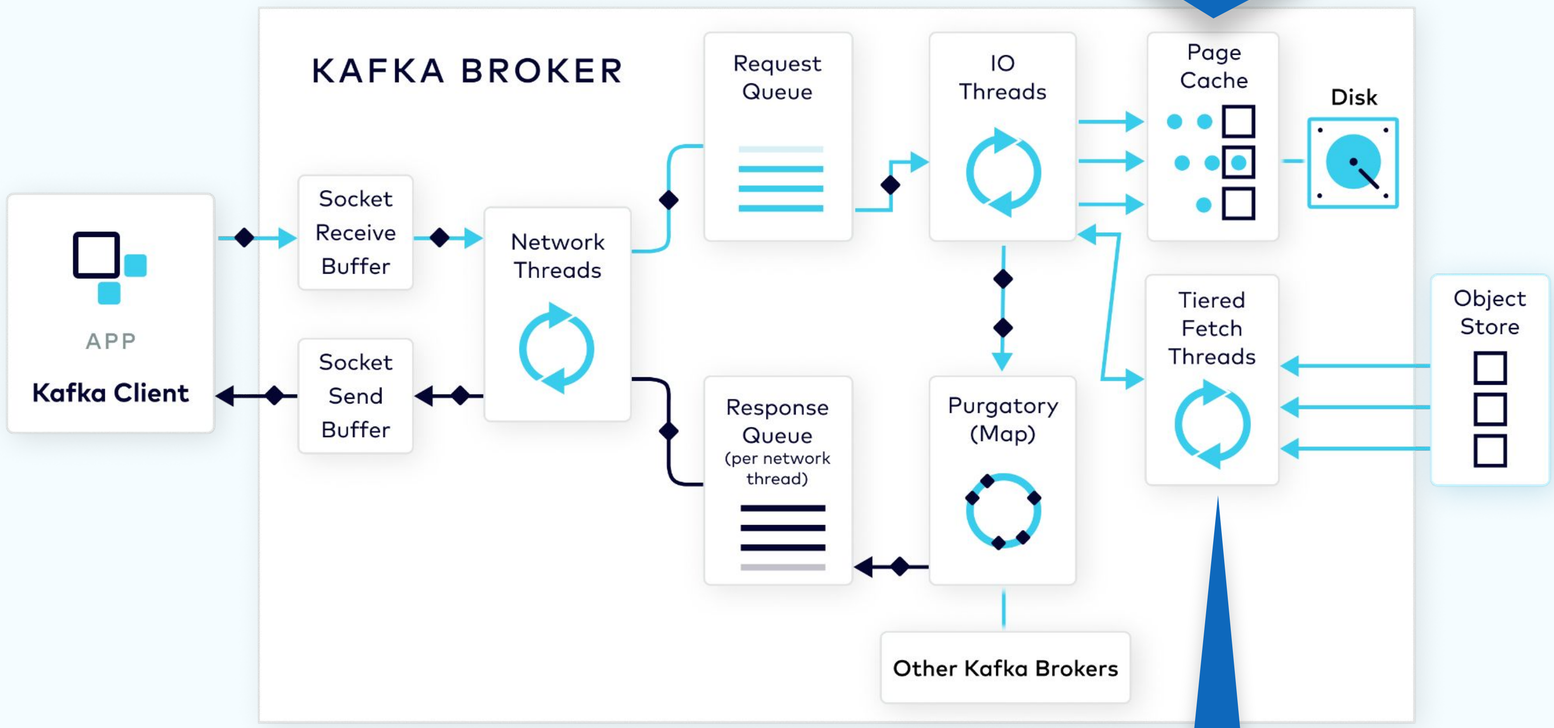


Recent events can exist in both the hotset and object storage

Fetching Tiered Data

CONFLUENT Developer

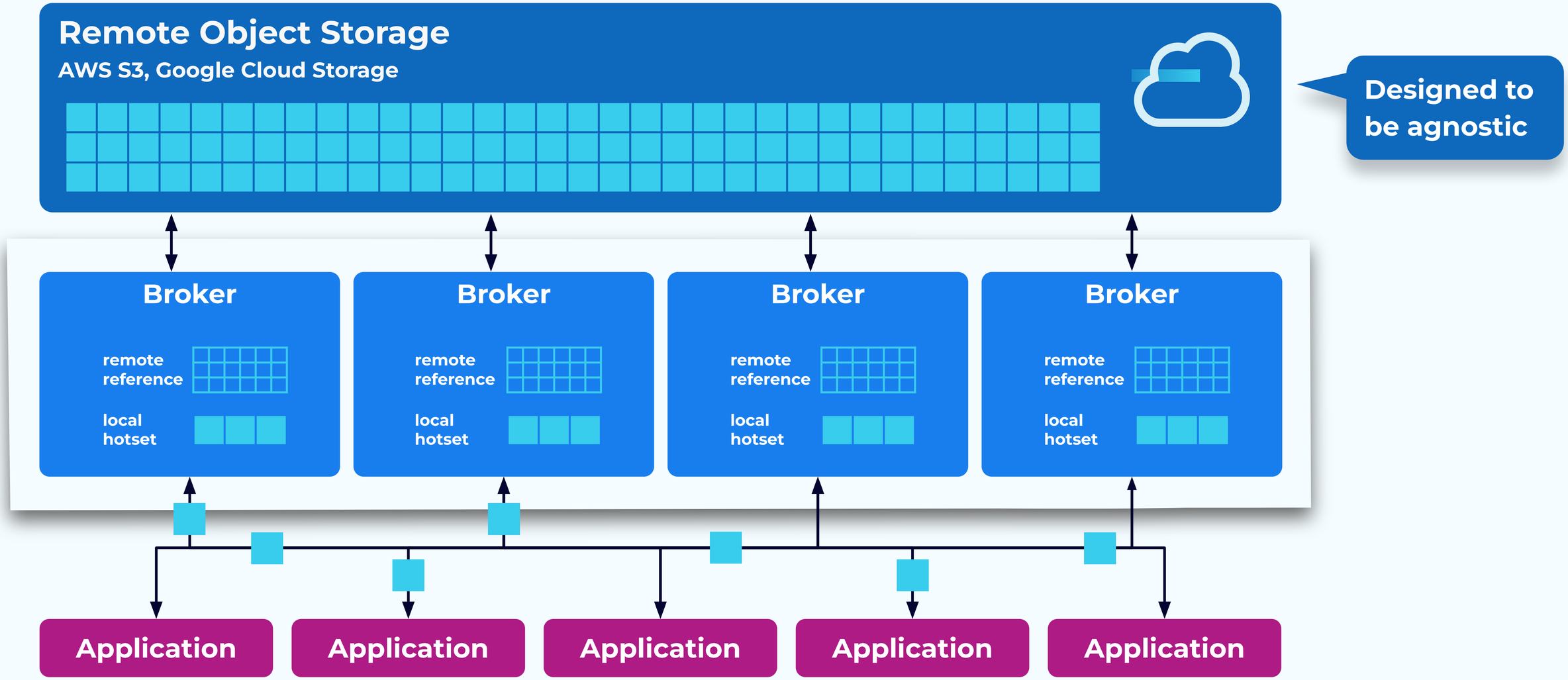
If data is in the hotset, it is returned using standard Kafka methods



If data is not in the hotset, broker retrieves it async from remote storage and serves it to the client

Tiered Storage Portability

CONFLUENT Developer

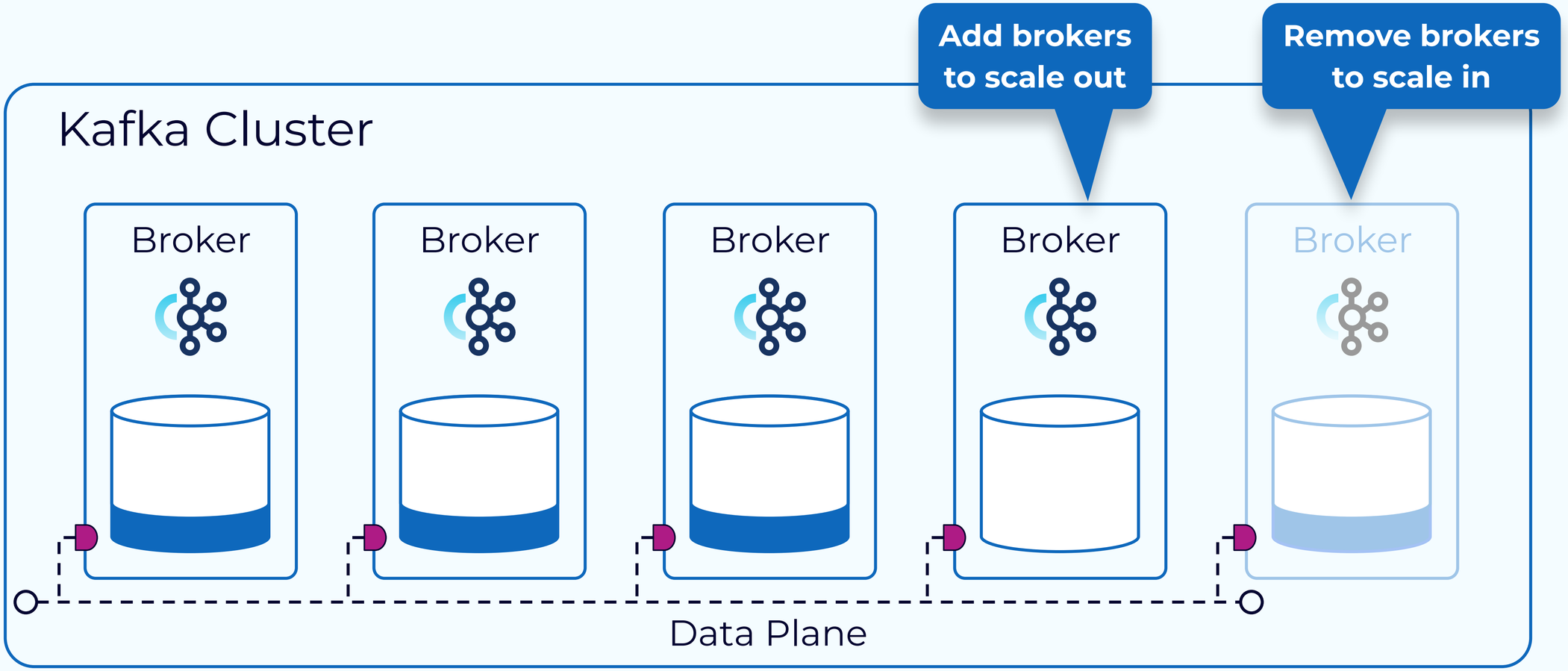


CONFLUENT
Developer

Cluster Elasticity

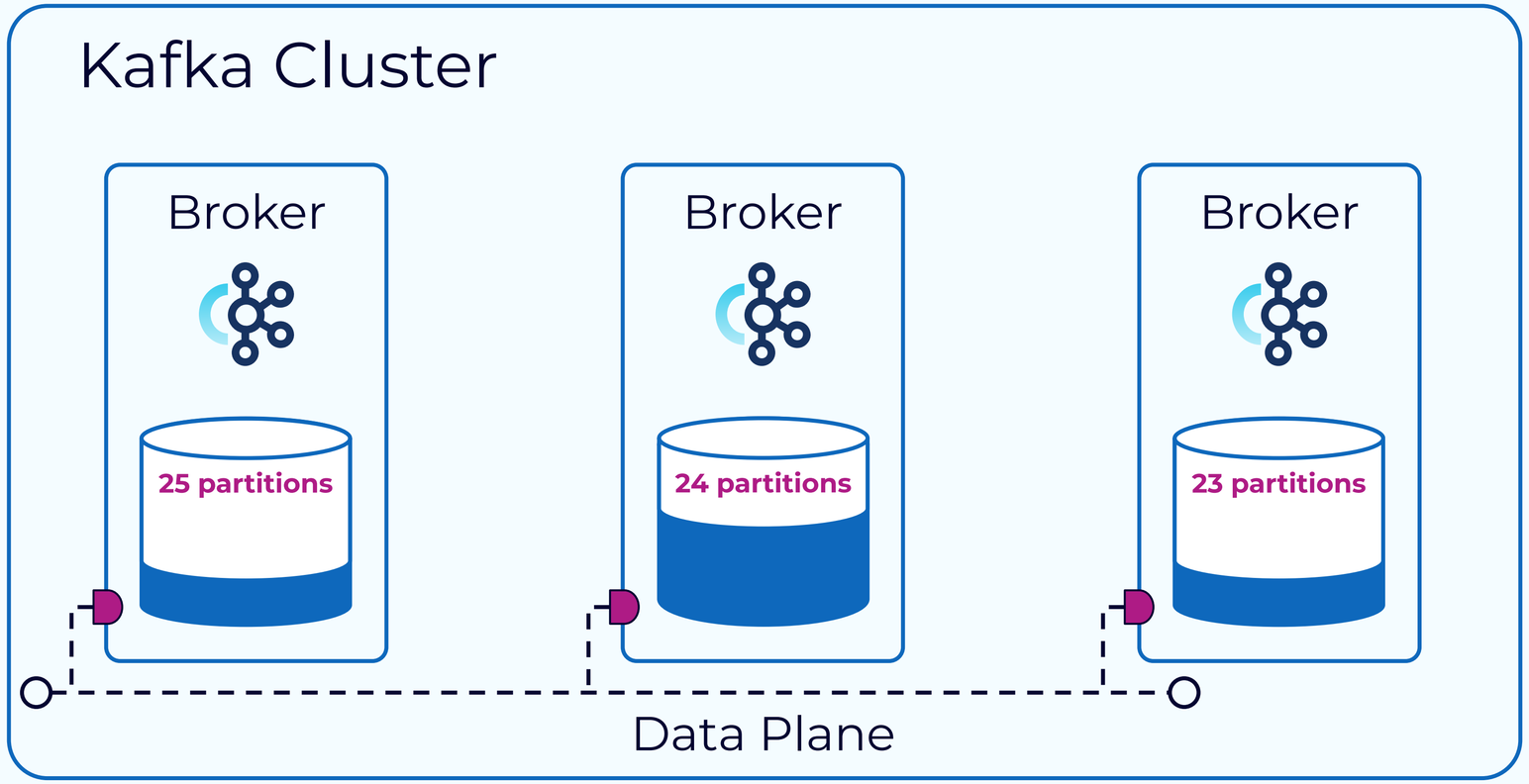
Cluster Scaling

CONFLUENT Developer



Unbalanced Data Distribution

CONFLUENT
Developer



Automatic Cluster Scaling with Confluent Cloud

CONFLUENT
Developer

Confluent Cloud

kafka-reassign-partitions.sh

CONFLUENT Developer

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

CONFLUENT Developer

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
```

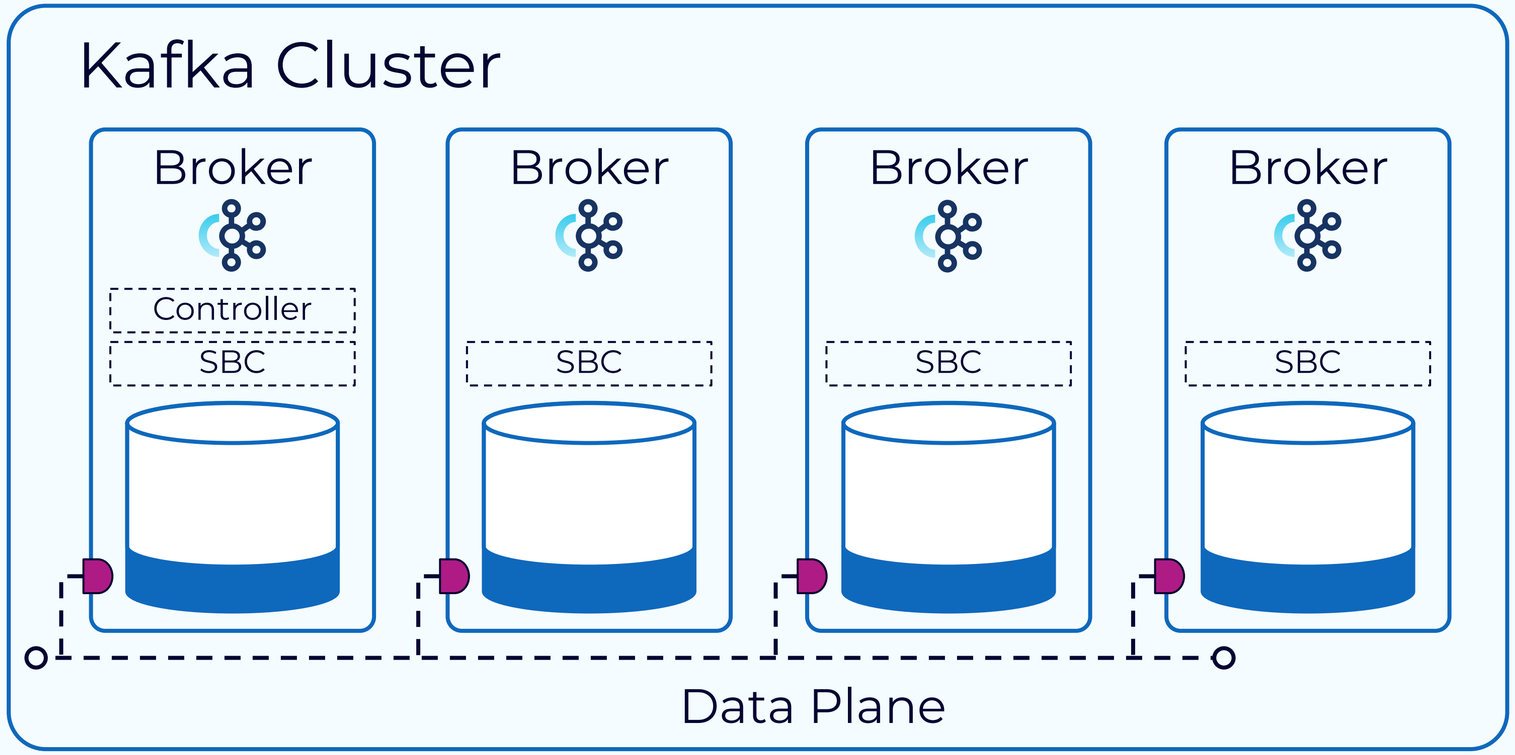
```
Computing the rebalance plan (this may take a while) ...
You are about to move 512 replica(s) for 512 partitions to 1 broker(s) with total size 233.9 MB.
The preferred leader for 192 partition(s) will be changed.
In total, the assignment for 515 partitions will be changed.
The minimum free volume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

CONFLUENT Developer

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 and Processing

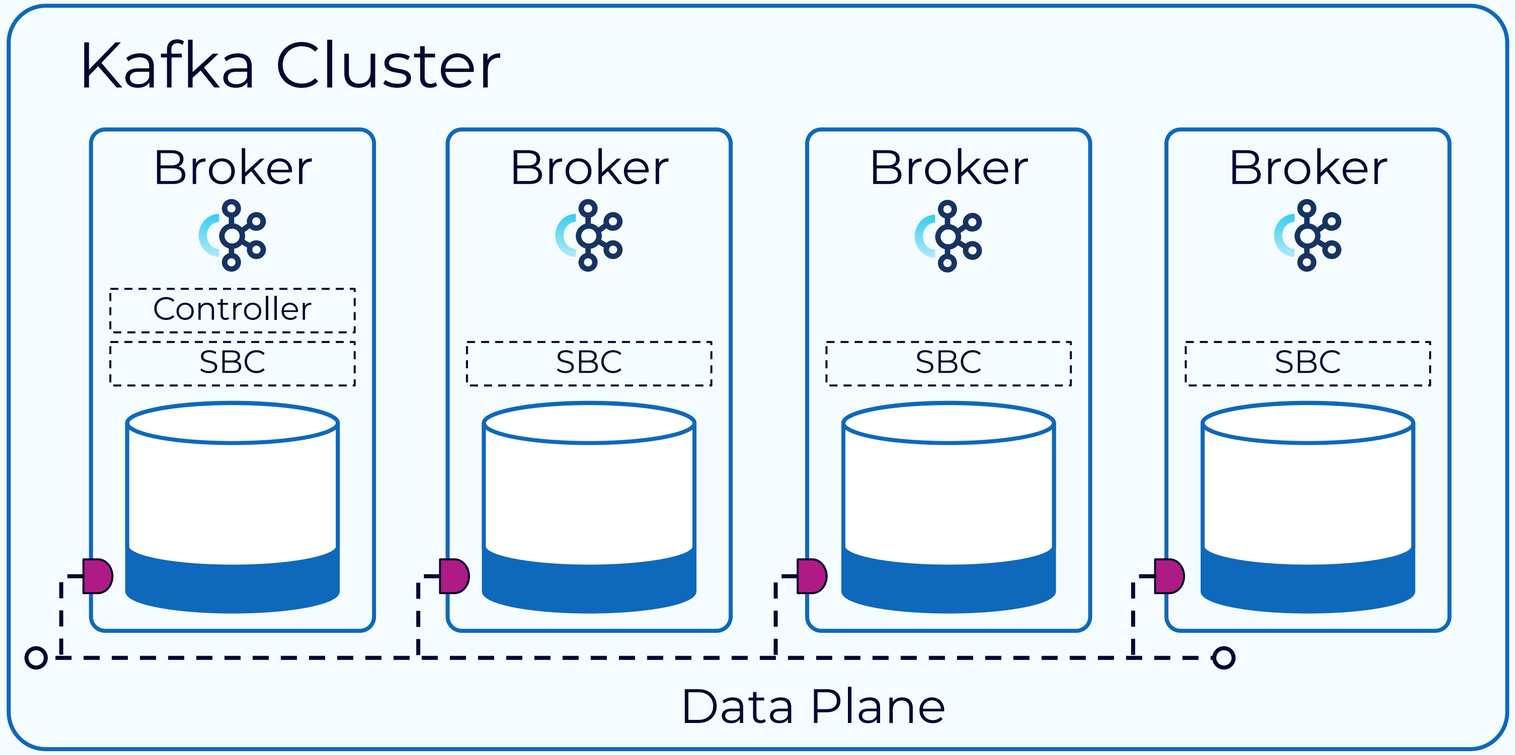
CONFLUENT Developer

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

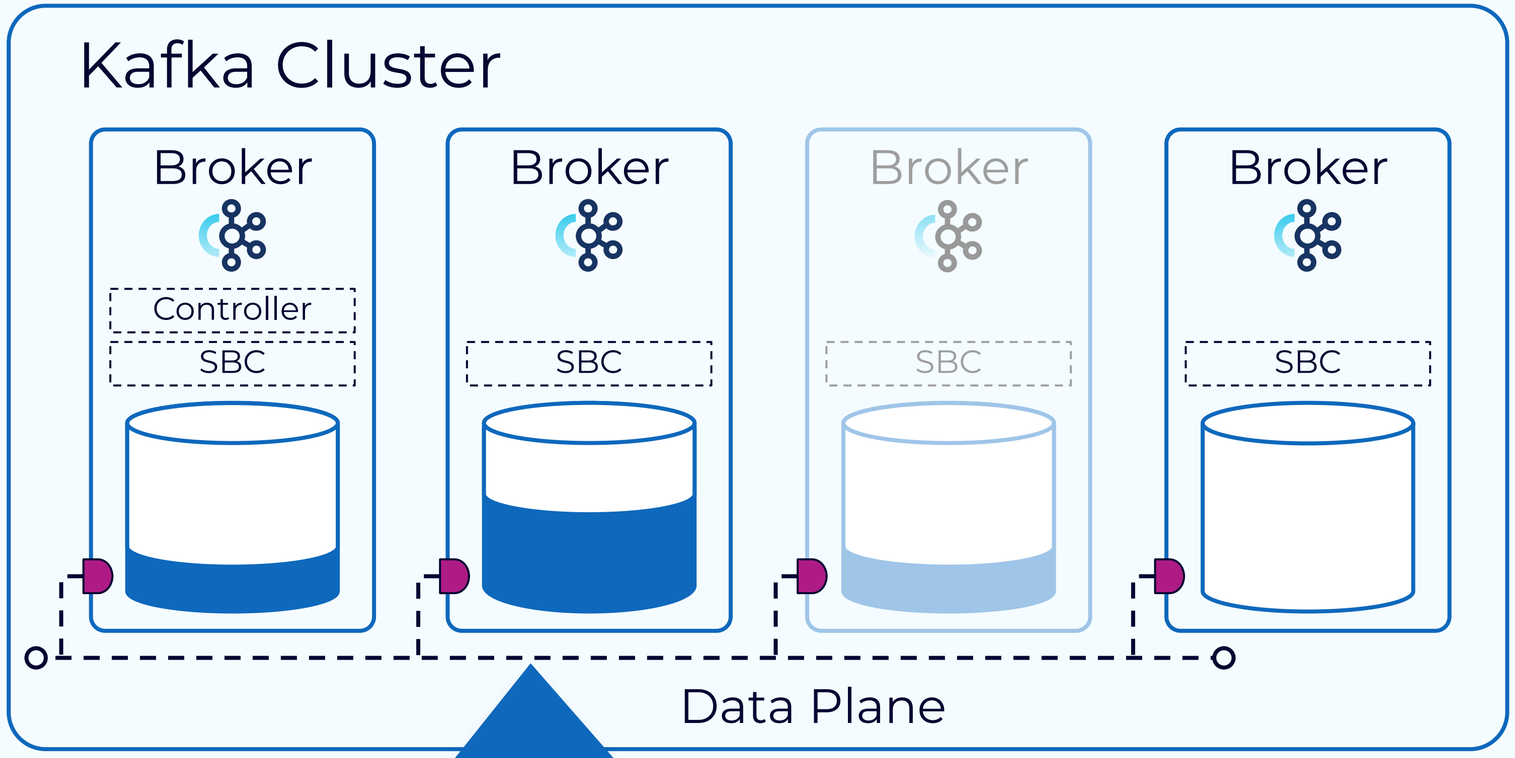
CONFLUENT Developer

Auto rebalance trigger options:

1) Added and removed brokers
2) Any uneven load

Uneven load condition is based upon:

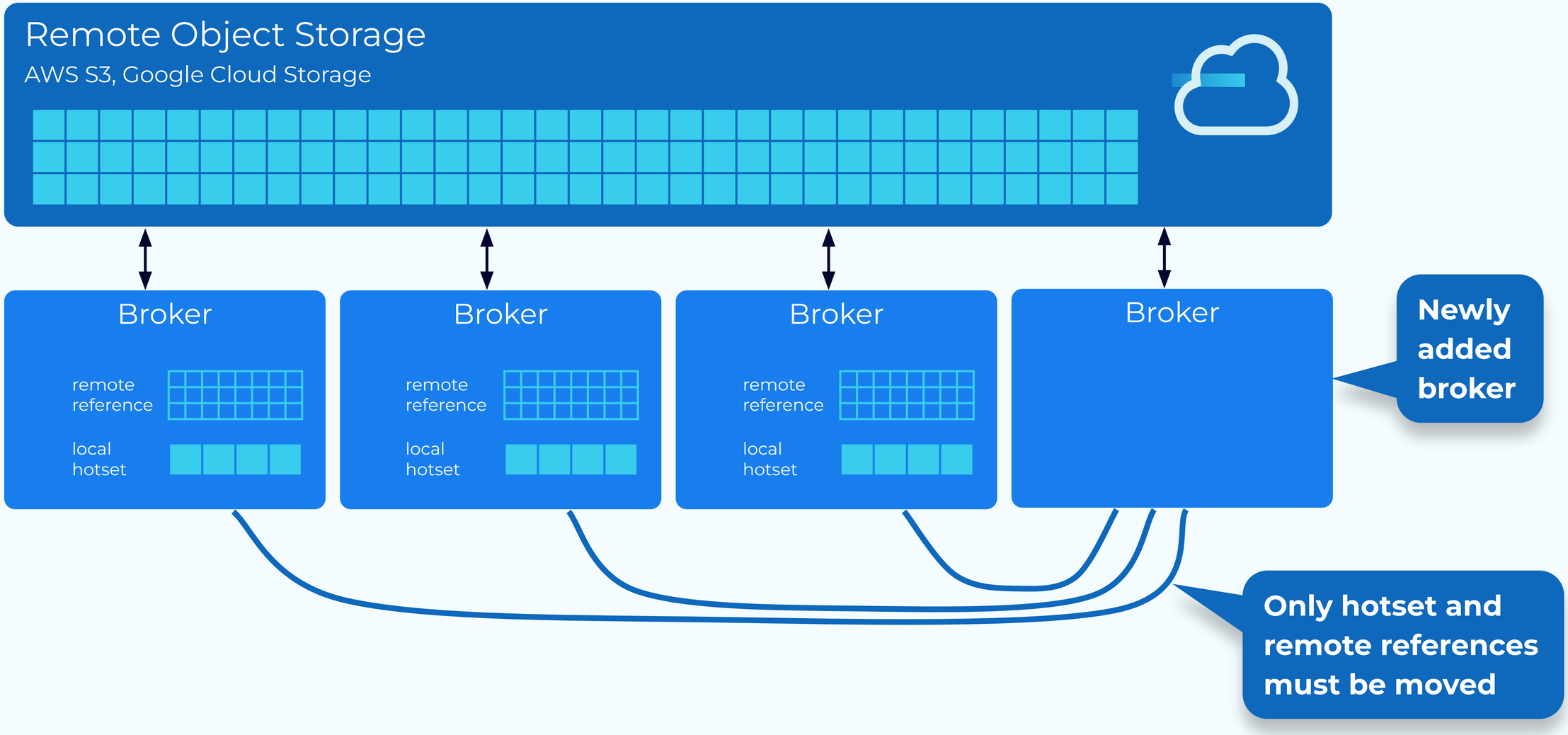
1) Disk usage and network usage
2) Number of partitions/replicas per broker
3) Leadership and rack awareness



SBC throttles replication during rebalance to prevent negative impact to clients

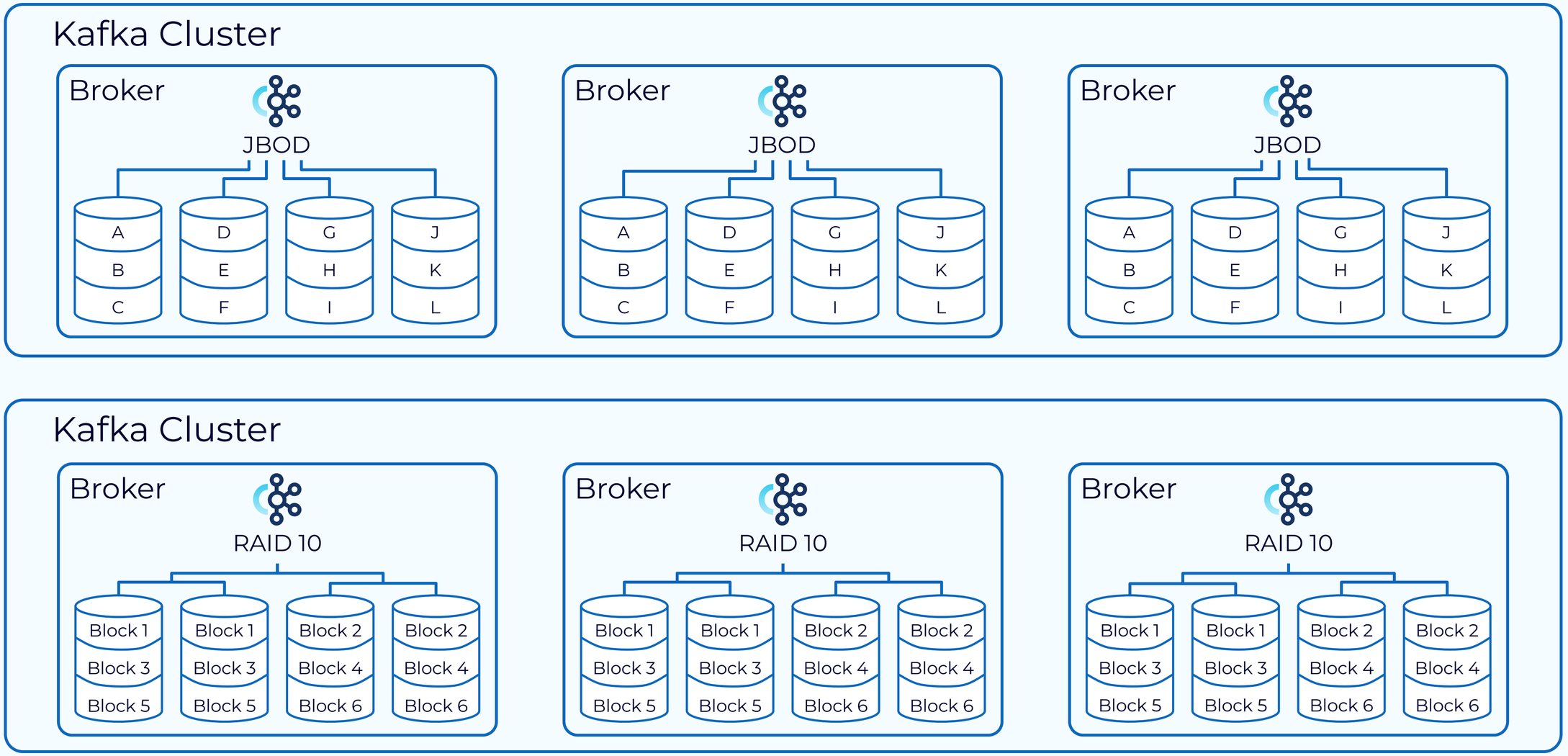
Fast Rebalancing with Tiered Storage

CONFLUENT Developer



JBOD vs RAID

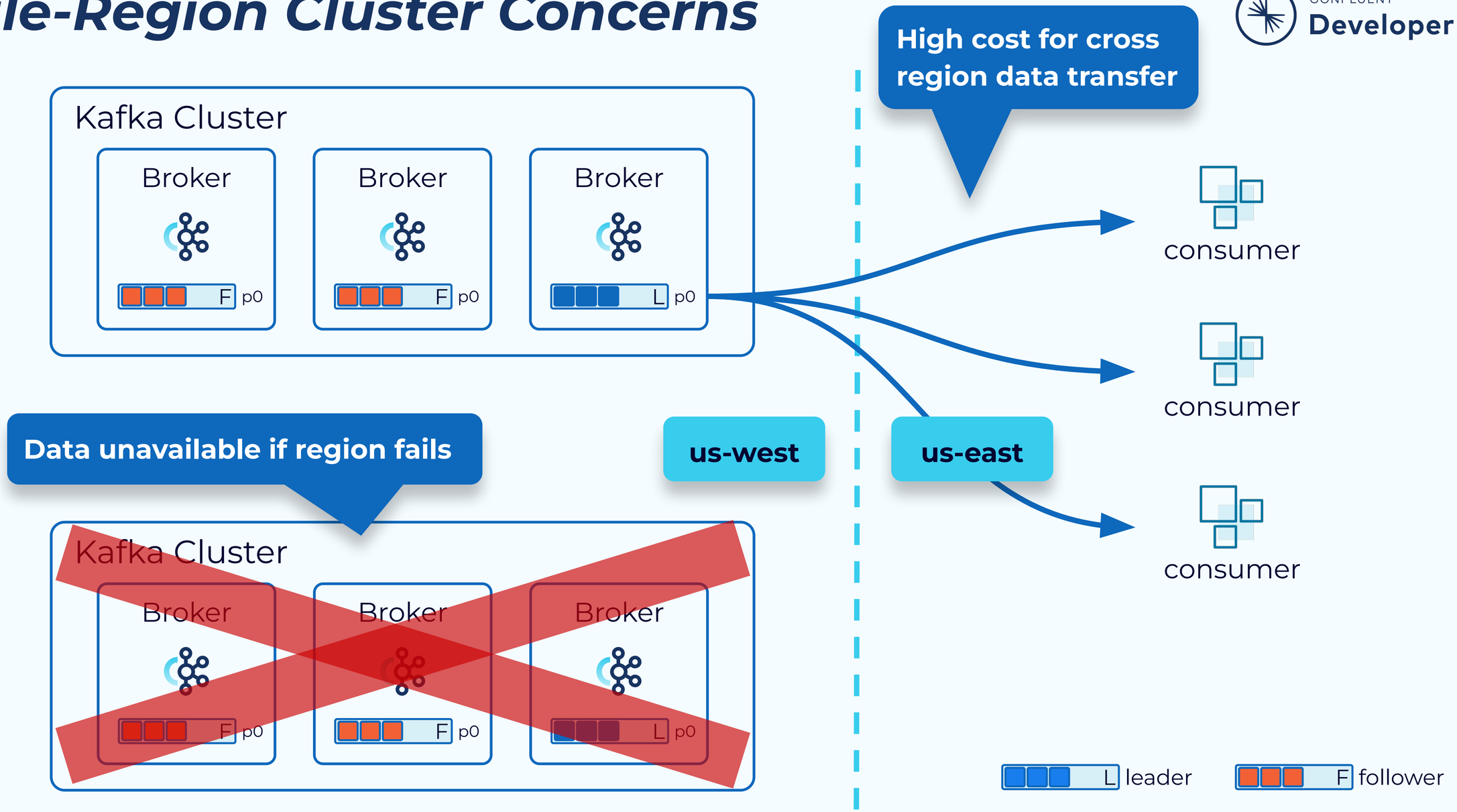
CONFLUENT Developer



CONFLUENT
Developer

Geo-Replication

Single-Region Cluster Concerns



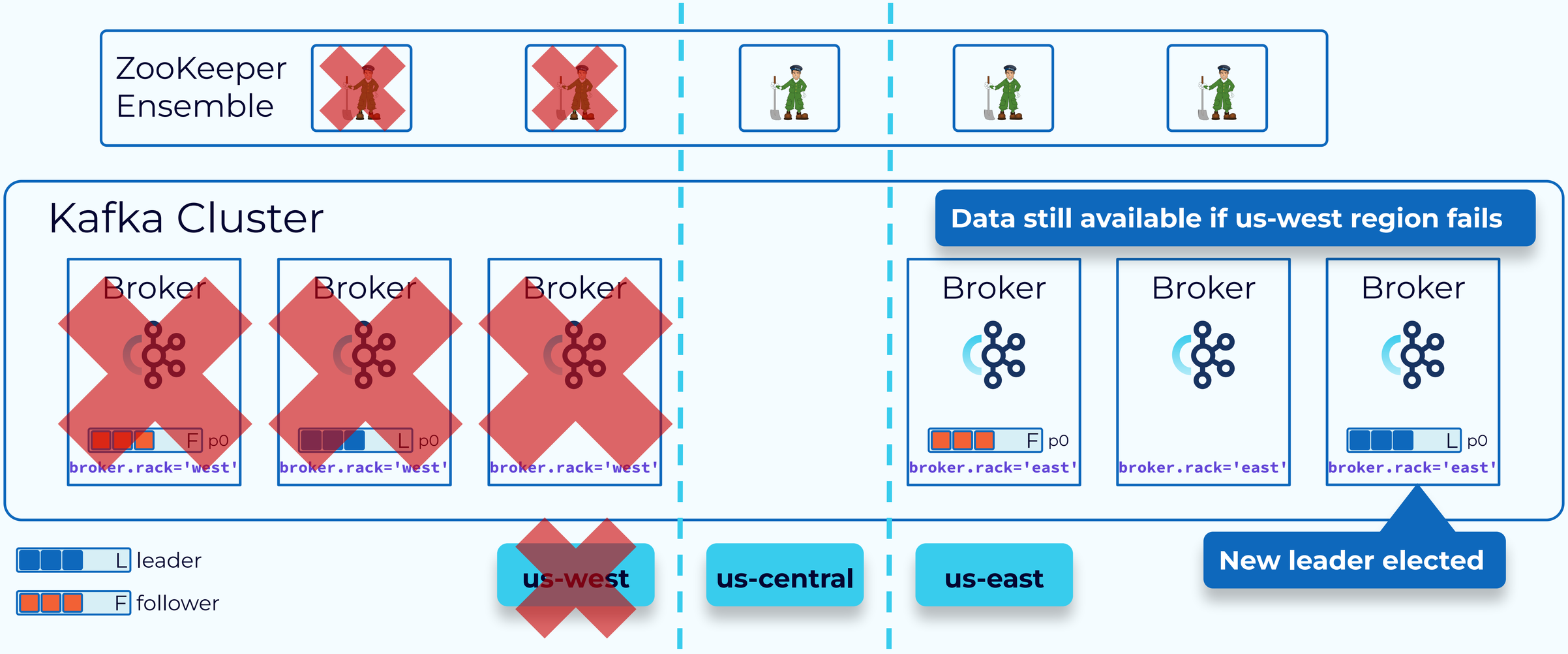
Geo-Replication

CONFLUENT Developer

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

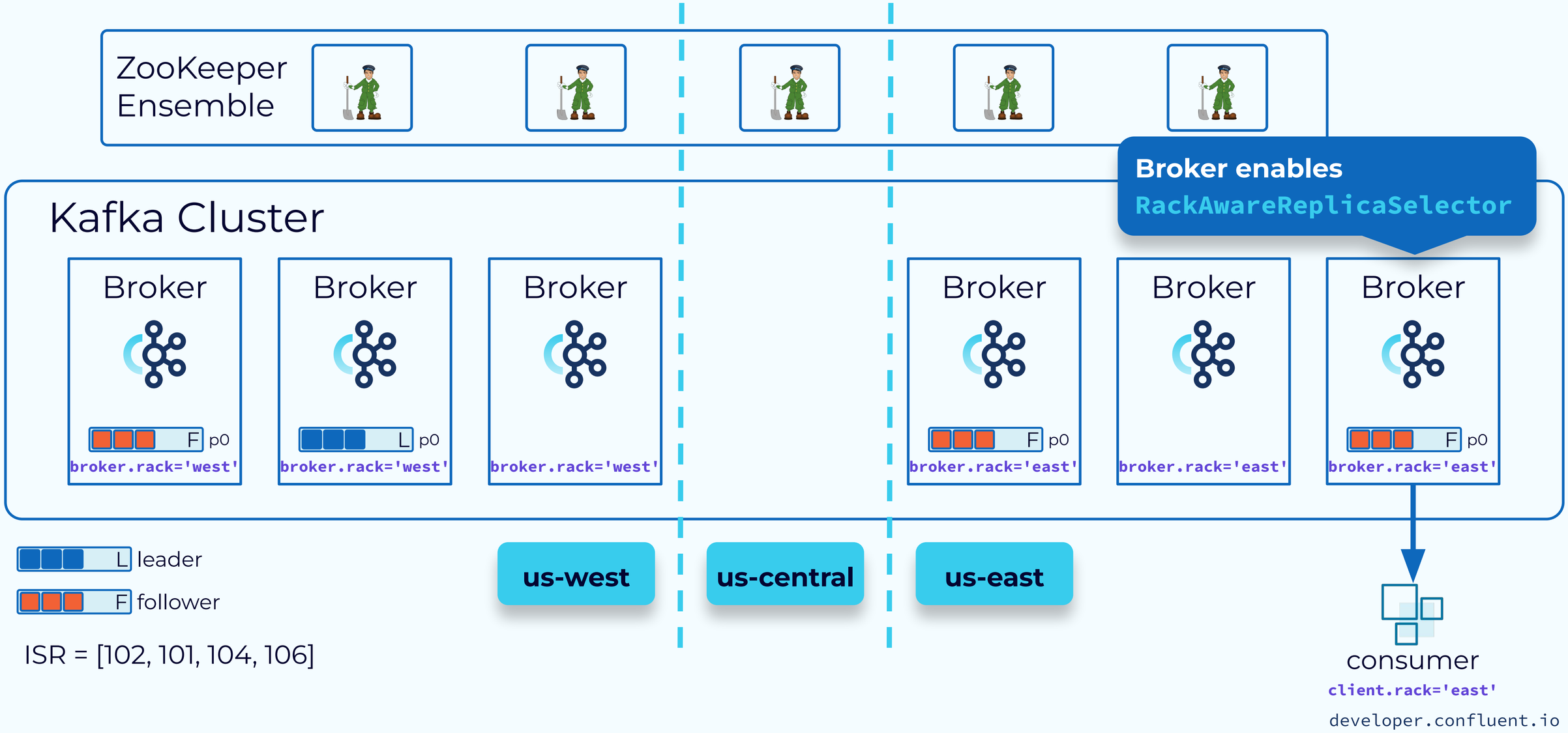
Confluent Multi-Region Cluster

CONFLUENT Developer



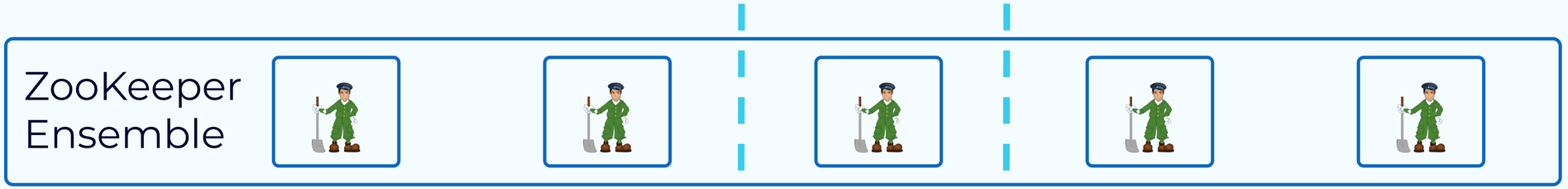
Better Locality with Fetch From Follower

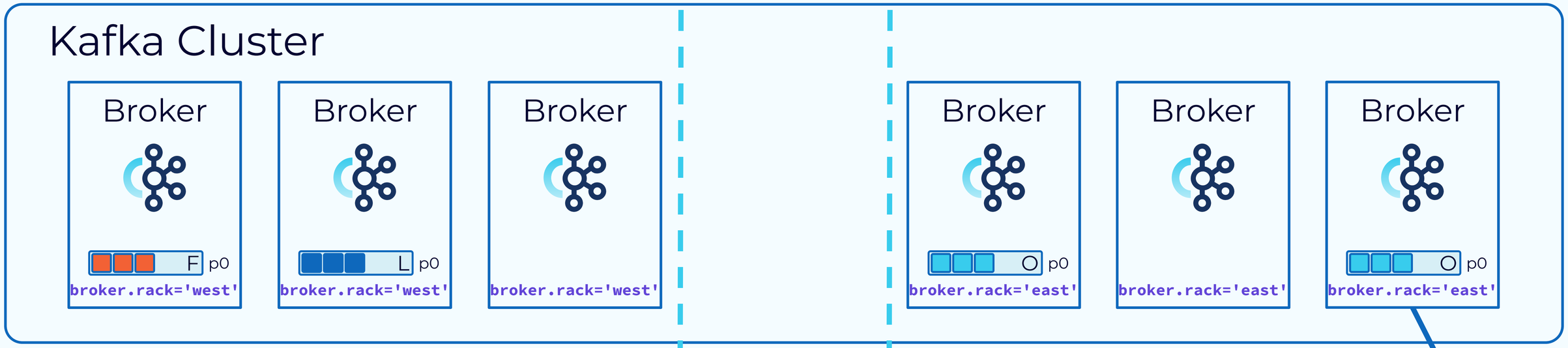
CONFLUENT Developer



Async Replication with Observers

CONFLUENT Developer





L leader

F follower

O observer

ISR = [102, 101]

us-west **us-central** **us-east**

Observer replicas are not included in the ISR

- `observerPromotionPolicy` **sets conditions where observers join the ISR**

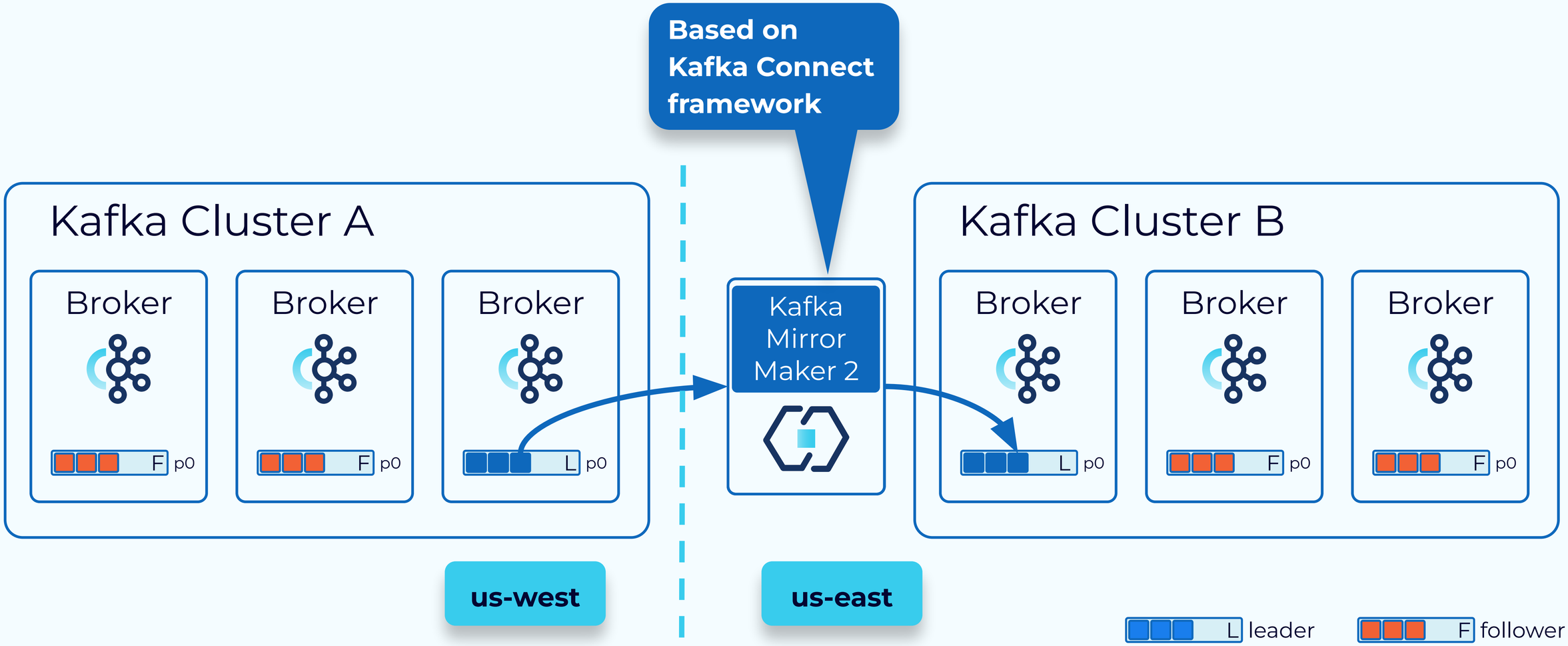
consumer

`client.rack='east'`

developer.confluent.io

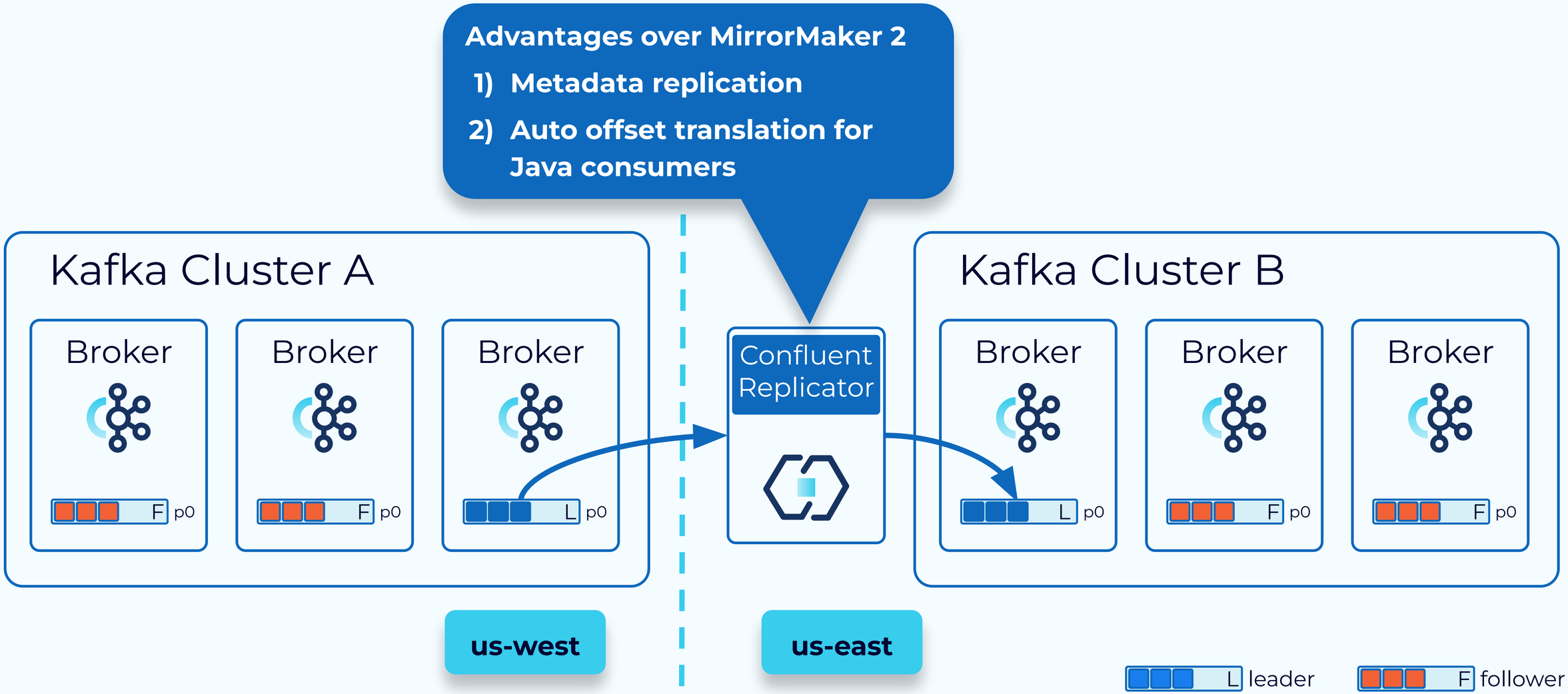
Kafka MirrorMaker 2

CONFLUENT Developer



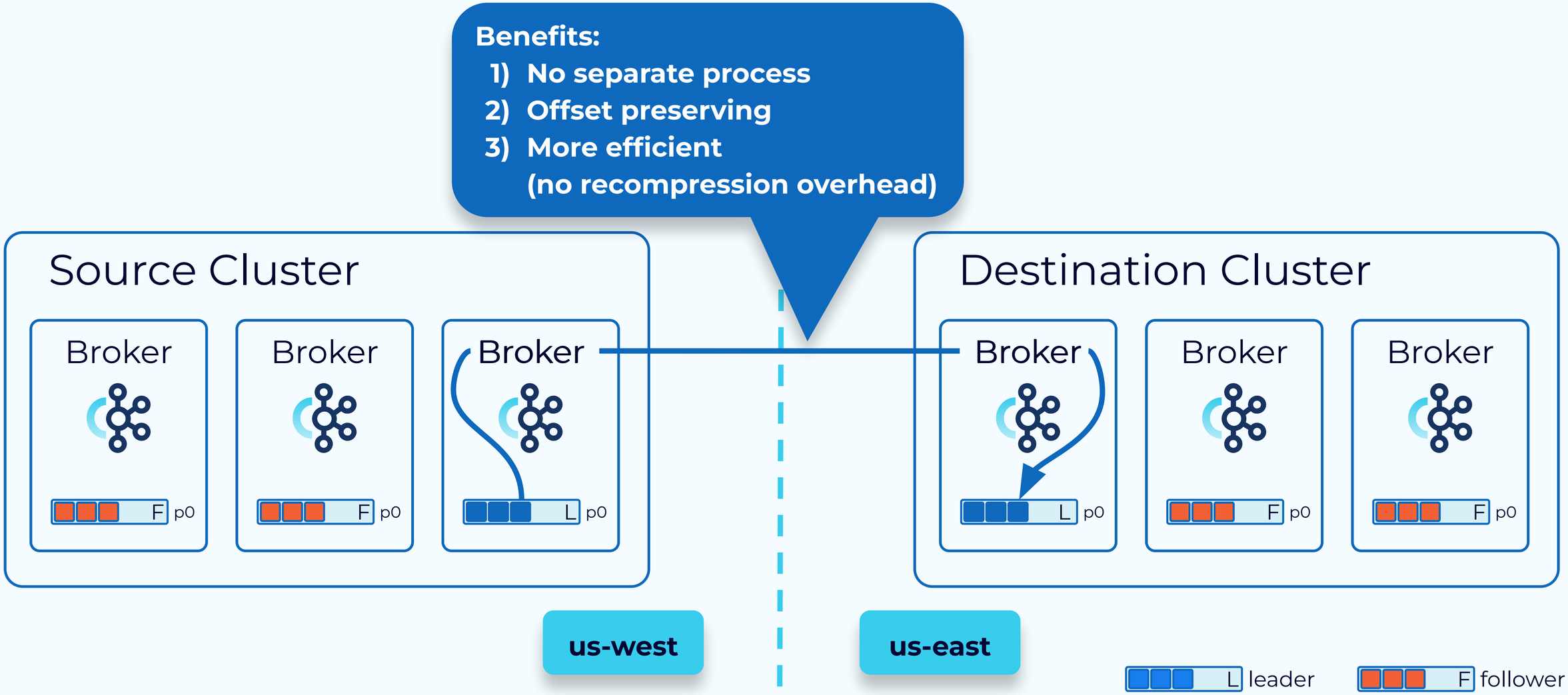
Confluent Replicator

CONFLUENT Developer



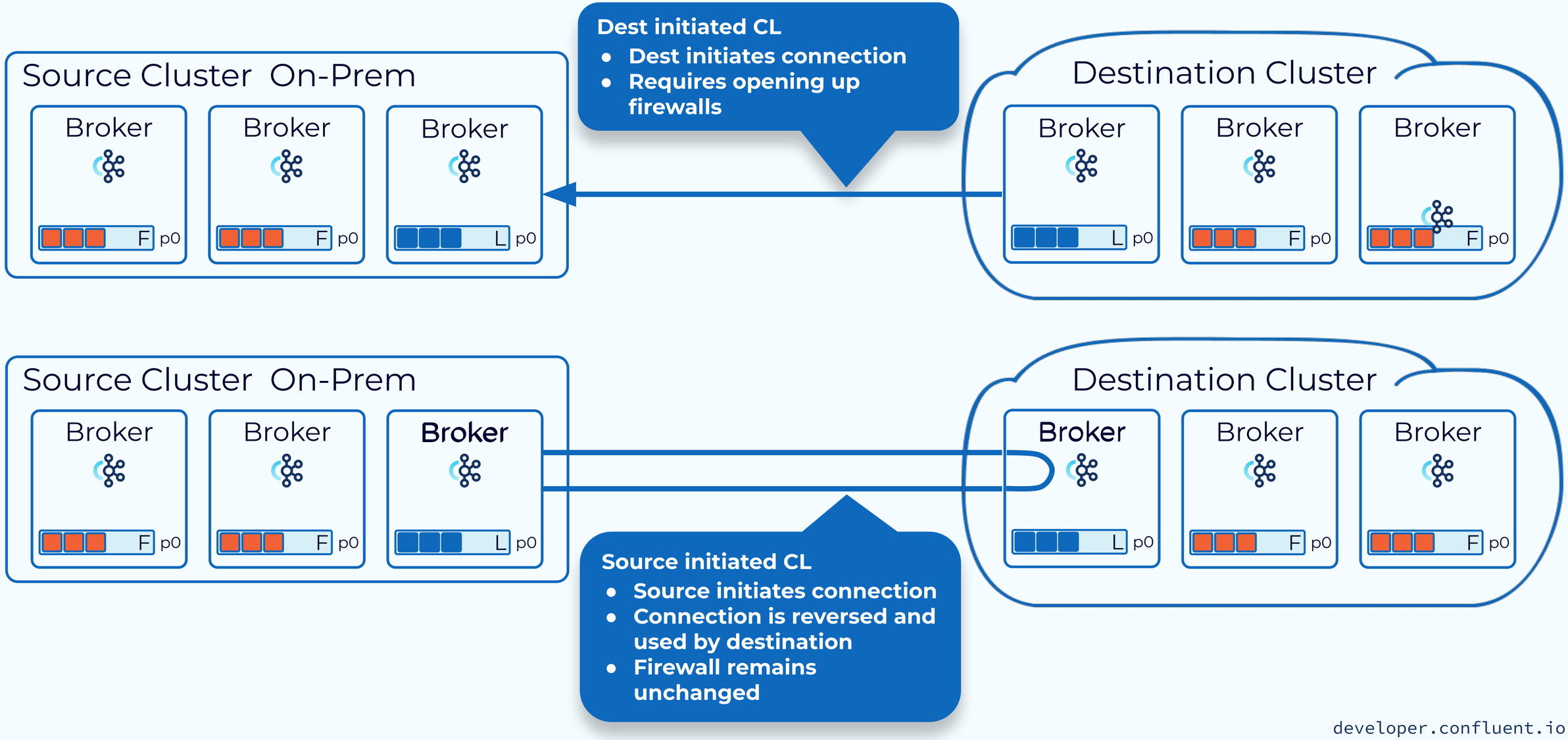
Confluent Cluster Linking

CONFLUENT Developer



Destination vs Source Initiated Cluster Linking

CONFLUENT Developer



Geo-Replication Recap

CONFLUENT Developer

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

CONFLUENT
Developer

Hands On: Cluster Linking

CONFLUENT
Developer

Your Apache Kafka journey begins here

developer.confluent.io