STUDY GUIDE

Confluent Certified Developer for Apache Kafka (CCDAK)

PREPARED BY

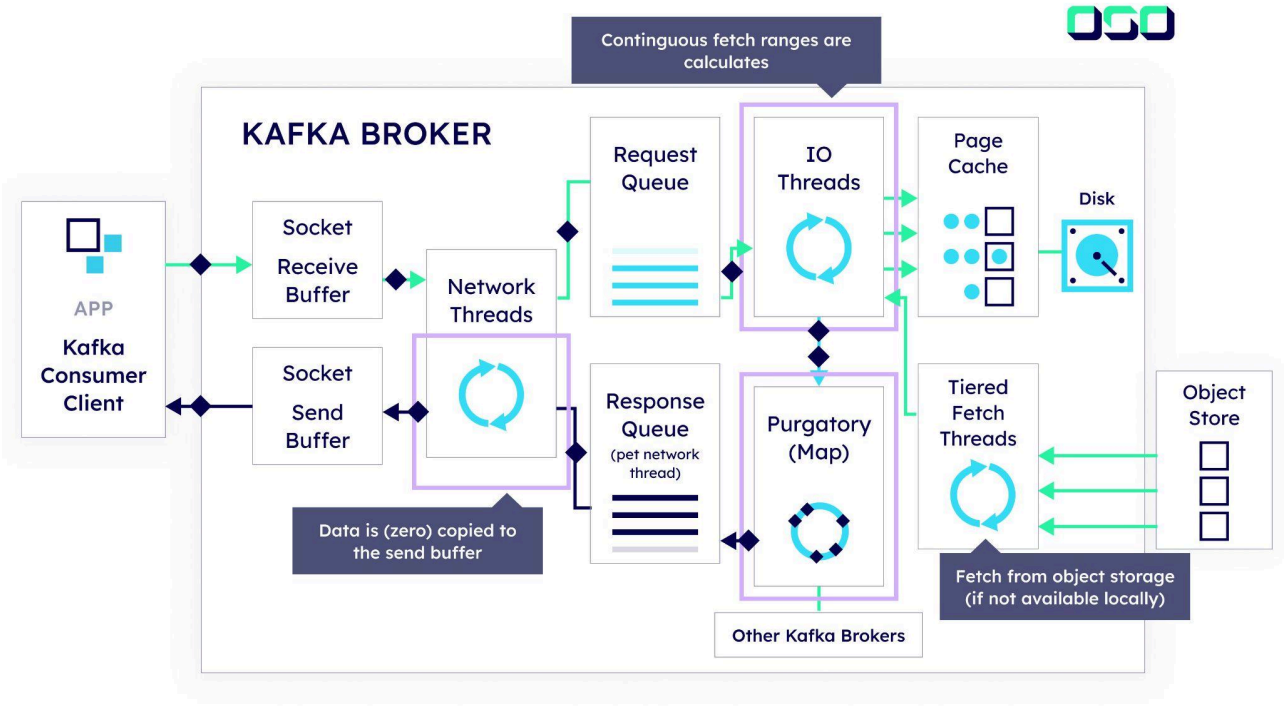
Sion Smith

PUBLISHED
October 2023

1. Introduction

We have all been there, trying to study for a subject we know nothing about, wondering if what we are learning or reading is actually going to turn up in the test. This guide outlines everything you will need to know to pass your Confluent Kafka Developer Certification (CCDAK).

We cover Apache Kafka's core architecture, delving into server-side mechanics including operational pathways, partition-based message persistence, reliability safeguards, and sequence preservation protocols. Enhanced features encompassing consumer group coordination, atomic operations, hierarchical storage management, scalable deployment, and multi-region synchronisation are thoroughly examined. The platform's fundamental structure is emphasised, showcasing its implementation as a real-time event distribution framework built on decentralised storage foundations. The Publication and Subscription interfaces enable message transmission and consumption, while Integration and Transformation APIs (Stream Processing and KSQL) support system connectivity and data manipulation.



1.1 Key concepts

- **Events:** Modelled as records with a timestamp, key, value, and optional headers.
- **Topics:** Organise events of the same type, similar to database tables.
- **Partitions:** Distribute data within a topic across Kafka cluster nodes, serving as units of data distribution and parallelism.
- **Offsets:** Unique IDs for events in a partition, aiding in event order tracking.

2. The Kafka Broker

In a Kafka cluster, the control plane handles metadata, while the data plane manages client requests. Two main request types are handled by brokers: produce requests from producers and fetch requests from consumers.

2.1 Producer requests

- The Producers send records which are assigned to partitions, either by key hashing or round-robin.
- Records are batched for efficiency and compression (depending on the configuration)
- Batches form a producer request sent to the broker.
- Network threads receive and queue the request for I/O threads.
- I/O threads validate data, append it to the commit log, and manage replication.
- Pending requests are held in purgatory until replication completes, then responses are generated and sent back to clients.

2.2 Fetch / consumer requests

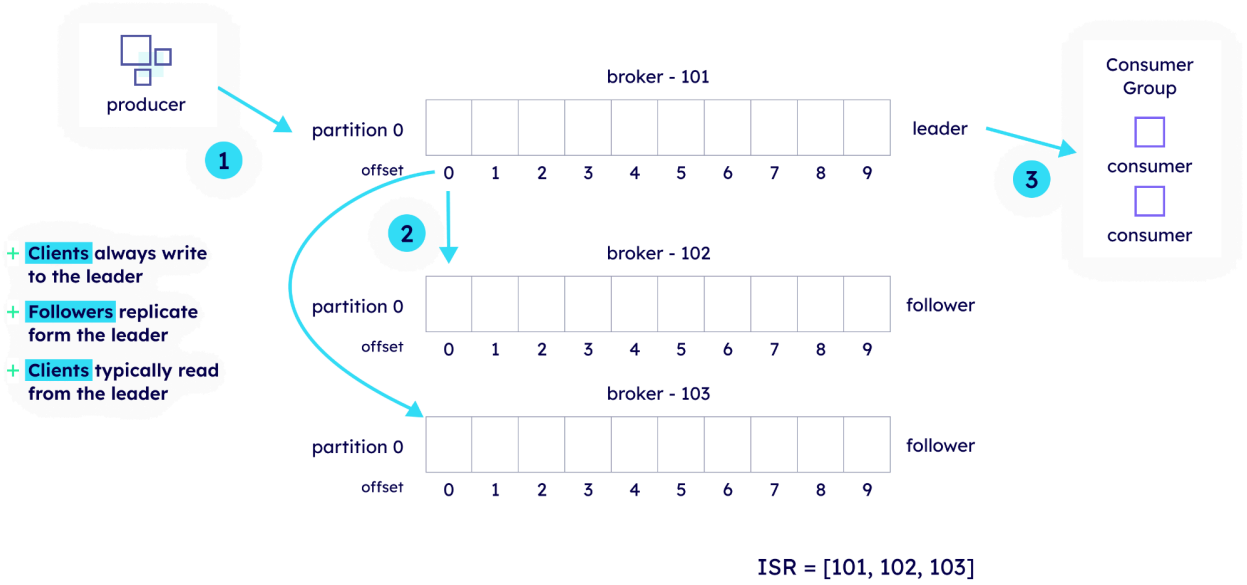
- Consumers must specify topic, partition, and starting offset in fetch requests.
- Similar to produce requests, fetch requests are queued and processed by I/O threads.
- I/O threads use the index to locate data and return it to consumers.
- Fetch requests can wait in purgatory for sufficient data accumulation.
- Responses are sent efficiently using zero-copy transfer, though disk access may cause delays.

3. Broker replication

A Kafka broker data replication protocol is crucial for durability and availability in mission-critical applications. When creating a topic, users specify the number of replicas. A leader replica handles data from producers, while followers fetch and replicate this data.

3.1 Key concepts

- In-Sync Replica (ISR): Set of replicas fully caught up with the leader.
- Leader and Followers: The leader manages data writes, while followers replicate it.
- Leader Epoch: Monotonically increasing number marking a leader's generation.
- High Watermark: Marks committed records available to consumers.



3.2 Replication process

1. Leader handles the producer requests and appends data to its log.
2. Followers fetch data from the leader, appending it to their logs.
3. Records are committed when present in all ISR replicas.
4. High watermark advances as records are confirmed by followers.

When a partition leader fails, the KRaft controller promotes new leader elections from ISR replicas. Data reconciliation ensures followers' logs match the new leader's log by truncating uncommitted data and fetching updates.

4. Kafka control plane

Kafka's control plane manages cluster metadata. Initially, Kafka relied on ZooKeeper, but the new KRaft (Kafka Raft) mode eliminates this dependency, integrating a Raft-based consensus service within Kafka.

4.1 KRaft advantages

- Simplifies operations by removing the need to manage ZooKeeper.
- Enhances efficiency and scalability of metadata handling.

4.2 KRaft configuration

- Supports non-overlapping and shared mode setups for controllers and brokers.
- Uses an in-memory metadata cache for quick controller failover.

4.3 Metadata management

- Metadata changes are logged in a special single-partition topic called "cluster metadata."
- Metadata replication mimics data replication but uses a quorum-based system without ISR.

4.4 Leader election

- Follows a quorum-based approach with vote requests and responses among controllers.
- New leader coordinates log reconciliation to ensure consistency.

4.5 Snapshot mechanism

- Periodic snapshots prevent indefinite growth of metadata logs.
- Snapshots help rebuild in-memory metadata caches on broker restarts and during metadata fetches.

The KRaft migration is a well documented process which can be found on the Confluent website. Kafka 4.0 will be fully saying goodbye to ZooKeeper. There will be no support for running in ZK mode, or migrating from ZK mode.

5. Consumer Group Protocol

To enable the scalability and parallel data processing, Kafka has a concept of consumer groups. A consumer group divides the topic data across multiple consumer instances, each assigned to process a subset of partitions. The group coordinator manages this process by assigning individual partitions to consumers, monitors them and rebalances when changes like group membership is triggered.

5.1 Group initialization

- New consumers identify the group coordinator for each topic.
- A group leader, which is selected from all the online consumers, assigns partitions to individual consumers using a pluggable strategy.

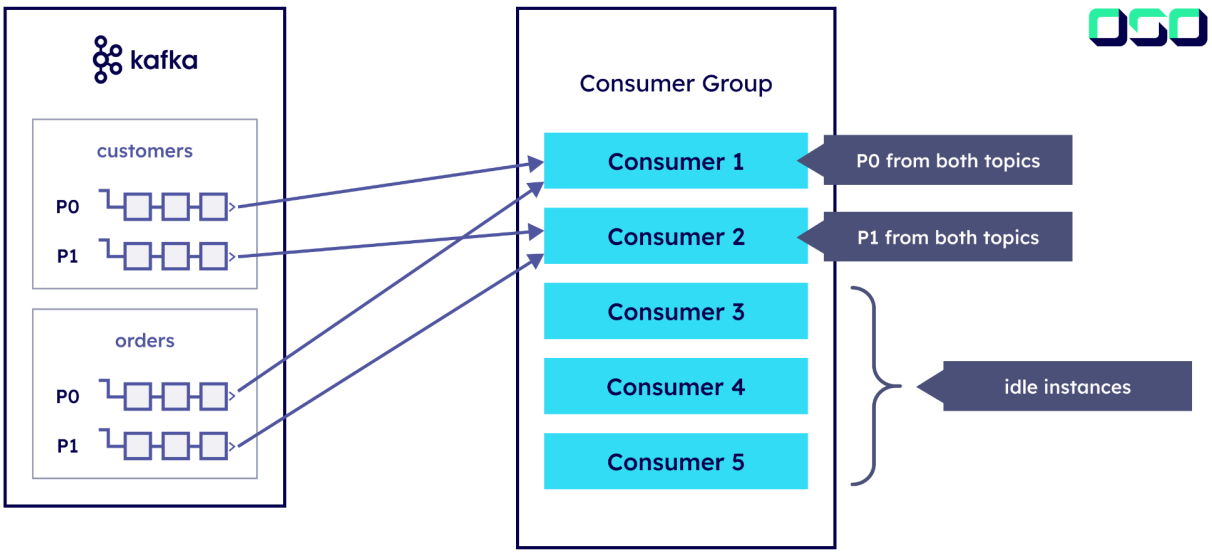
5.2 Group coordinator

- Responsible for managing group membership and partition assignment.
- The state of this component is stored in the **consumer_offsets** internal topic.
- Manages all consumer groups for all topics.

5.3 Partition assignment strategies

Range:

- Assigning a continuous range of partitions to each consumer
- Useful when processing related data together



Round Robin:

- Distributes partitions equally among consumers
- Ideal for high-throughput scenarios requiring parallel processing

Sticky:

- Maintains partition assignments during rebalances when possible
- Reduces overhead by minimising partition movements

Cooperative Sticky

- Supports cooperative rebalances
- Allows uninterrupted consumption from partitions that aren't being reassigned
- Enables smoother rebalancing operations

There are several important aspects to consider when selecting a partition assignment strategy**:**

- **Message Ordering:** Kafka only guarantees message ordering within a single partition
- **Scalability:** The number of consumers in a group cannot exceed the number of partitions
- **Load Balancing**: The assignment strategy impacts how evenly the workload is distributed across consumers
- For stateful services or applications where partition assignments are important to the application logic, using the StickyAssignor is recommended as it attempts to maintain partition assignments across rebalances while still ensuring even distribution

5.3 Offset tracking

Consumer offset is a mechanism that tracks the sequential order in which messages are consumed from Kafka topics. It serves as a bookmark indicating where a consumer should resume reading if there's a pause or failure in processing.

- Each consumer group maintains its own offset position for every partition it consumes
- Offsets are stored in an internal Kafka topic called **__consumer_offsets**
- The offset is an integer that uniquely identifies each message within a partition

Tracking Process:

- As consumers read messages, they advance their offset linearly
- Consumers periodically checkpoint their position to the **__consumer_offsets** topic

- The offset is controlled by the consumer, allowing it to consume messages in any order if needed

Triggering rebalancing:

- When a consumer fails, restarts or a new one joins, a rebalance is triggered.
- If you add a new partition to a topic, a rebalance is triggered.
- Rebalancing pauses all processing on the topic until the state has been rebuilt.

6. Kafka configuration

Apache Kafka ensures high availability and reliance through replication, where specifying N replicas on a topic will allow tolerance for N-1 failures. When a producer writes an event to a topic, the acknowledgment of that write is configurable. A configuration property known as **ACKS,** modes vary from **0** (no durability, low latency) to **all** (high durability, higher latency) which ensures a copy of the data has been written to all replicas before proceeding. These are set when creating the KAfka producer:

- **acks=0 (Fire and Forget)** Producer doesn't wait for any acknowledgement
- **acks=1 (Leader Acknowledgement)** Producer waits for acknowledgement from partition leader only
- **acks=all (All Replicas)** Producer waits for acknowledgement from all in-sync replicas

In applications where it is critical applications do not lose data, setting the property **min.insync.replicas** to more than 1 ensures records are only acknowledged if replicated to the specific number of replicas. Ordering guarantees are typically maintained but can be disrupted by network failures which could potentially lead to duplication and reordering.

In order to mitigate this, the Kafka protocol supports idempotent producers which assign a unique ID and sequence number to each message, preventing duplicates and maintaining order even during retries.

If you want to guarantee end-to-end ordering of messages with a topic with the same key, enable idempotency and set **acks=all**

7. Transactions

Apache Kafka transactions provide atomic write capabilities across multiple topics and partitions, ensuring data consistency and exactly-once processing semantics. Transactions are necessary to simplify application logic and act like a database when failure occurs with a complete rollback.

Kafka transactions are designed specifically to enable exactly once semantics, allowing the application process events and **only** write once to an output topic. You enable this on each transactional producer by setting a unique **transactional.id.** By setting this you guarantee the consumers only see committed data.

Behaviour matrix

Scenario	Description
Failure without Transactions	Results in duplicates and inconsistencies when application restarts.
Failure with Transactions	Producer ID and transaction state is managed by the transaction coordinator. Prevents uncommitted data from being exported to consumers.
Successful transaction	Coorindiator commits transaction log and the data is exposed to consumers. Kafka brokers will maintain the last stable offset to ensure read consistency.
Consuming transaction with read_commited	Consumers with **read_commited** ensures that only committed records are read, avoiding / filtering aborted transactions. In read_committed mode, the consumer uses a mechanism called Last Stable Offset (LSO)
Idempotent producers	If enabled, this feature ensures exactly-once message delivery semantics by tagging records with a unique producer ID and sequence number. Duplicates are prevented and order is maintained during failure.
Transaction commit intervals	To balance the overhead of data consistency with latency, we can adjust **commit.interval.ms** and **transaction.timeout.ms**
External systems	Kafka transactions are only bounded within the internals of Kafka. If you need to wrap things like †database calls consider a two-phase strategy or an outbox pattern.

8. Topic compaction

Topic compaction retains the latest record for each key while removing older records with the same key, regardless of retention period. This feature is particularly useful for maintaining state-based data or when only the latest updates are relevant.

8.1 How does compaction work?

Structure: Consists of two main parts

1. The **head** is where new records are appended and may contain duplicate keys
2. The **tail** contains only unique keys after compaction has occurred

Compaction Process

1. All messages in a compacted topic must have explicit keys
2. Kafka periodically scans the log segments to identify messages with duplicate keys
3. When duplicates are found, only the most recent message for each key is retained
4. Older messages with the same key are removed during the compaction process

Logs are divided into cleaned (contains no duplicates) and dirty (contains duplicates) segments. Older segments are swapped out with clean ones consumers read from these updated segments.

Tombstones and transaction markers

A tombstone is a special message with a key and a null payload that signals deletion of all messages with that key. A two-phase approach is taken to cleaning these segments to prevent prematurely deleting null records ensuring the applications have time to read deletions.

Compacted topics which contain transactional control records also require special handling. Only messages from committed transactions after the transactional high watermark can participate in compaction. Two main scenarios are handled:

1. When the transactional high watermark is in the active segment, no compaction occurs
2. When it's in a rolled-over segment, messages from pending transactions must be preserved

8.2 Configuration

Topic compaction is enabled at a topic level using **cleanup.policy=compact,** other notable configurations are:

- **log.cleaner.enable:** Must be set to ‘true
- **log.cleaner.min.cleanable.ratio:** Controls when compaction is triggered
- **min.compaction.lag.ms:** Defines minimum time before messages become eligible for compaction

9. Tiered storage

Tiered Storage, which was initially introduced as an early access feature in Kafka 3.6, has now been promoted to production-ready status in the 3.9 release. This has been introduced to cater for a more cloud native deployment and mitigate three limitations with the original storage design:

- **Cost:** Local block based storage is very expensive compared to object storage
- **Elasticity:** Adding storage would require operational complexities across all brokers.
- **Isolation:** Can cause performance issues when touching local disk.

Tiered Storage in Kafka operates across two distinct storage tiers:

- **Local Tier:** Uses fast, expensive storage (like SSDs/NVMe) for recent, frequently accessed data.
- **Remote Tier:** Uses slower, cost-effective storage (like S3/HDFS) for historical data.

9.1 How does tiered work?

Data Storage Process

1. New data is initially written to the local storage tier on Kafka brokers
2. All segments except the active (open) segment are asynchronously transferred to remote storage
3. Data moves between tiers based on retention settings, not copied but moved to optimise storage
4. Only one copy of each segment is uploaded to remote storage, regardless of replication factor

Data Retrieval

1. Recent data is served from local storage using standard Kafka mechanisms
2. When consumers request historical data from remote storage:
3. Brokers download and cache the records locally
4. A small portion of disk space (2-16GB or 5% of broker disk) is allocated for caching remote data
5. Subsequent retrievals of the same data benefit from the cache

Limitations

- Not supported for compacted topics
- Multiple log directories (JBOD features) are not supported
- Once enabled, tiered storage cannot be deactivated without support intervention
- Increasing local retention doesn't move remote segments back to local storage

10. Cluster elasticity

Kafka can be scaled up and down by adding and removing brokers from the cluster. This should NOT be done dynamically by some metric driven approach - no matter who tells you. Adding and removing brokers requires redistributing data to maintain balance. Over time, data can become unevenly distributed. This takes time and resources from each broker and should be a planned activity.

10.1 Cluster tooling

- **The Kafka Reassignment Tool** (kafka-reassign-partitions.sh) is a tool primarily for managing partition distribution and replication by allowing administrators to move partitions between brokers and adjust replication factors. It uses a JSON format file to specify new replica assignments.
- **Kafka Auto Data Balancer** Confluent only tool that automatically balance data across kafka brooks and racks to ensure even distribution of:
    - Leader partitions
    - Disk usage
    - Resource utilisation
- **Self-Balancing Clusters** Agents monitor and continuously adjust cluster balance ensuring even data distribution with minimal manual intervention.

10.2 Storage options

If using local storage, you must make the decision on JBOD vs RAID. This comes down to your specific use case:

RAID 10 (Preferred)

- Offers the best balance of performance and redundancy
- Provides better load balancing among disks
- Higher cost due to reduced usable storage capacity
- Good protection against disk failures
- Recommended if budget allows

RAID 1

- Second preferred option after RAID 10
- Provides basic mirroring
- Good for smaller deployments
- Offers redundancy with less complexity

RAID 0

- Third preferred option
- Offers good performance
- No redundancy
- Higher risk due to lack of fault tolerance

RAID 5/6 (not recommended)

- Not recommended for Kafka
- Significant negative impact on write throughput
- High I/O cost during array rebuilding
- Poor performance during disk failures

If you are planning to use JBOD it is important to take these considerations into account:

Limitations

- JBOD does not automatically balance load across disks like RAID-10
- Manual monitoring and management of disk space is required
- Kafka won't automatically redistribute partitions if one disk gets full
- A partition stays on its assigned disk and won't move to other disks automatically

Best Practices

- Monitor disk usage actively across all volumes
- Implement manual load balancing between disks
- Use separate I/O paths for each disk for better performance
- Avoid network-attached storage (NAS)
- Configure proper replication factor (minimum 3) for data safety

11. Disaster recovery

Apache Kafka's disaster recovery (DR) involves a range of strategies and architectures to ensure business continuity in case of catastrophic failures. Replication to different geo-located regions allows for data availability across multiple regions and can lower the latency in remote regions through Confluent's backbone network. We should know all the different options for the types of replication:

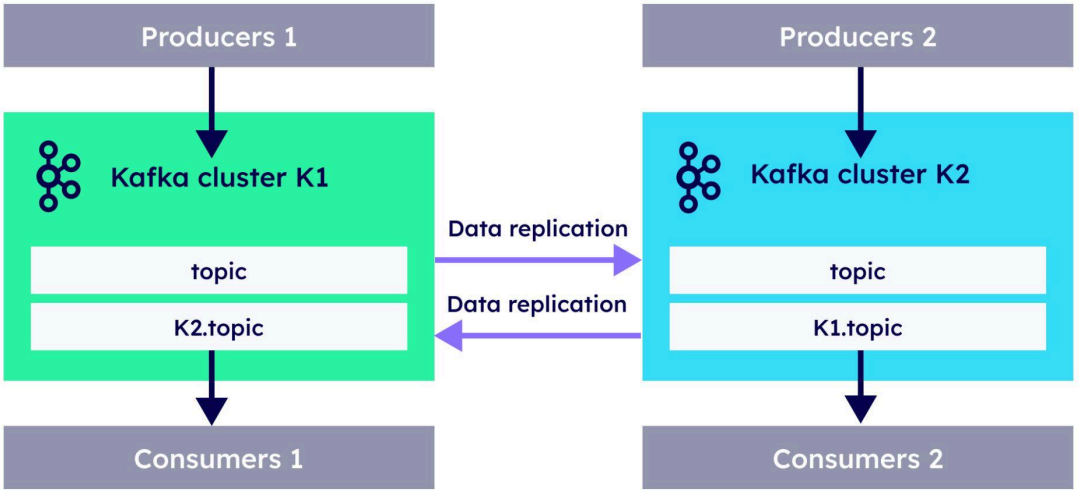
11.1 Cluster linking

Cluster Linking creates a persistent bridge between Kafka clusters, enabling:

- Direct cluster-to-cluster connections without additional components.
- Byte-for-byte replication of topics.
- Preservation of message partitions and offsets.
- Brokers act as servers with an embedded process with each.

Some of the advantages to note with this approach are:

- Tolerates network latency and unpredictable networking availability
- Handles compressed messages more efficiently by avoiding decompression-recompression cycles / Automatically recovers from reconnections



11.2 Confluent replicator

Built as an enterprise-grade solution for cross-datacenter replication is has some useful features compared to the open source equivalent:

- Integrates with Kafka Connect and Confluent Control Center
- Provides centralised management and monitoring capabilities

- Handles both topic configuration and data replication

Some advantages using Confluent replicator over other open source alternatives:

- Offers centralised configuration management
- Integrates with Confluent Control Center for GUI-based monitoring
- Provides easier deployment and configuration options

11.3 MirrorMaker 2

MirrorMaker 2 is built on top of the Kafka Connect framework and offers significant improvements over its predecessor. Some important features you need to remember:

- Offset preservation is not configured out of the box, it requires a manual process for translation.
- Dynamic configuration changes support with bidirectional replication capabilities.
- Topic renaming and filtering capabilities much like Confluent replicator.

11.4 Feature comparison

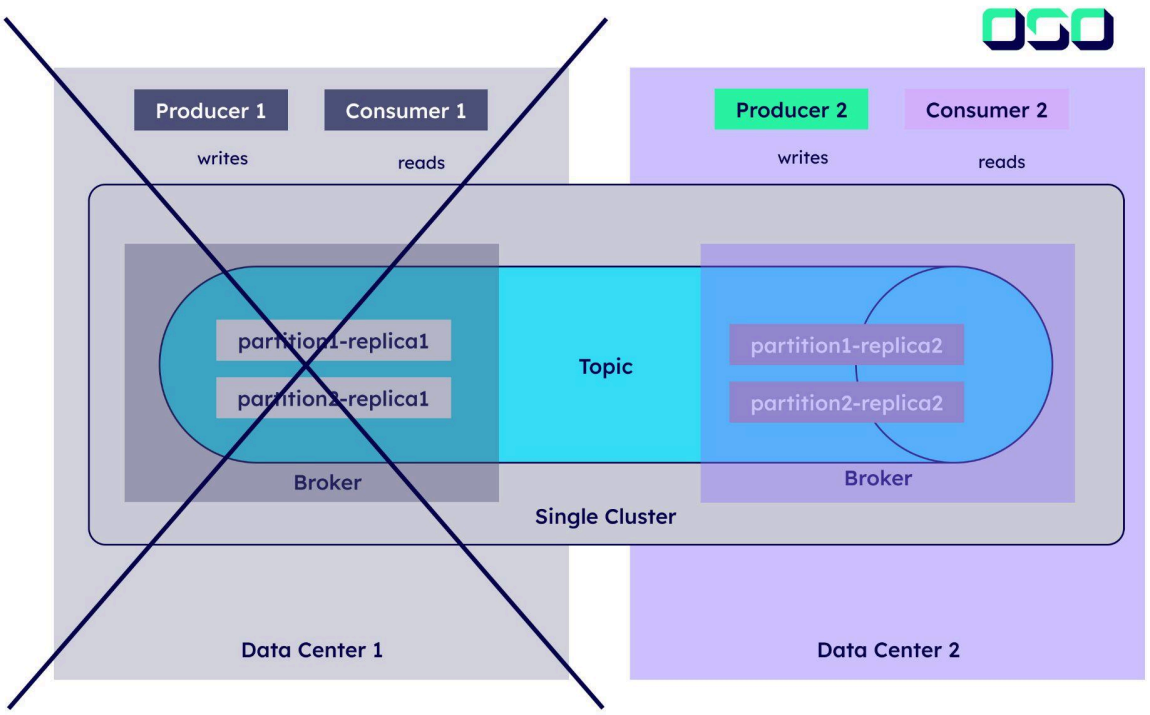
Feature	MirrorMaker 2	Confluent replicator
Topic auto-detection	Yes	Yes
Configuration sync	Yes	Yes
ACL management	Yes	Yes
Website Gui monitoring	No	Yes
Enterprise support	Community	Commercial

11.5 Stretch cluster

A stretched Kafka cluster is a single logical cluster deployed across multiple datacenters or availability zones, while maintaining synchronous replication between sites. The cluster operates as one unified system, with replicas distributed evenly across different physical locations using Kafka's rack awareness feature

- Demands low latency (recommended maximum 50ms) between datacenters.
- Requires stable, high-bandwidth connections between sites
- Provides strong data durability through synchronous replication.
- Enables automatic failover when a datacenter fails.
- Client applications remain unaware of multiple datacenters.

- Uses Kafka's rack awareness feature to distribute replicas evenly across datacenters.
- Requires careful configuration of min.insync.replicas.
- Typically configured with acks=all for maximum durability.



12. Sample Exam Questions

Thank you to Daniel Sobrado for putting together a great repository full of Confluent Kafka questions to practise. These are scenario based questions, to help you prepare your understanding on the kind of questions you may face. All of this great information can be found here:

https://github.com/danielsobrado/CCDAK-Exam-Questions

13. Next steps

If you would like to like us to support you deploy, secure Apache Kafka on your infrastructure please reach out to us via email enquires@oso.sh