CONFLUENT

Mojaloop Architecture Workshop

Engagement Report & Runbook

Neeraj Upreti

Date	Version	Comments
11/30/2023	v0.1	First version for high availability and optimization recommendations
12/04/2023	v0.2	Added high availability guidelines for deployment architecture and trade-offs for recommendations
12/08/2023	v0.3	VM section and deployment architecture update

ABSTRACT

This document describes the recommendations provided by Confluent from the recent professional services engagement.

Table of Contents

[Mojaloop Architecture Workshop 1]
- Table of Contents 2
- Engagement Report 3
  - Engagement Goals & Confluent Approach 3
  - Executive Summary 4
- Recommendations for High Availability 5
  - Definition of High Availability 5
  - Managing Garbage Collection for High Availability 5
  - Kubernetes - Setup Pod Disruptor Budget 6
  - Configuring Broker for High Availability 6
  - Configuring Consumers for High Availability 8
  - Tuning Virtual Memory for High Availability 8
- Kafka Kubernetes Deployment Architecture 10
  - High Level Deployment Architecture for Availability 10
  - Guidelines for high availability and reliability 10
  - Broker Deployment 10
  - Zookeeper Deployment 11
  - Miscellaneous deployment related points to consider 12
- Recommendations for Optimizing Kafka Platform 14
  - Higher Throughput 14
  - Minimal Latency 15
  - Monitoring io-ratio and io-wait-ratio For Optimization 15
  - Monitoring batch-size-avg and compression-rate-avg 16
  - Producer Settings for Durability and Ordering 16
  - Hardware and Infrastructure Optimizations 17
- Confluent Recommendations to Optimize and Scale 19
  - About Confluent 19
  - Schema Management 20
  - Kafka Error Handling Patterns 22
  - Disaster Recovery 23

Engagement Report

Confluent Professional Services were engaged with Mojaloop in remote on November 2023. This report summarizes the activities and recommendations made during that engagement.

Engagement Goals & Confluent Approach

As a part of our engagement, Mojaloop and Confluent set out to define ways to::

- Optimally configure an on-prem deployment utilizing hardware plant across two data centers for national scale within East Africa and SouthEast Asia, and
- Achieve high availability, low latency, and resiliency to failure in this on-prem deployment.

Confluent took an interactive and input-based approach to cover all the aspects of the Central Ledger platform and the priority objectives of the engagement by:

- Facilitating a two-day workshop in Mural to map current state architecture and define future state designs based on business needs, service level agreements ("SLAs"), and disaster recovery,
- Reviewing architecture to ensure the proper utilization of Kafka in application,
- Hosting sessions to improve knowledge of Kafka with Best practices and limitations such as how to organize Kafka within a K8s cluster to achieve high availability and resiliency to failure in an on-premises deployment.

The result is this Engagement Report & Runbook to structure OSK for new adopters to implement the Customer solution with recommendations such as architecture design, hardware planning considerations, setup and installation, configuration, monitoring and alerting, Kafka and Confluent best practices, troubleshooting, disaster recovery, and future coding.

Executive Summary

Mojaloop is building the Central Ledger Open Source Kafka Platform to serve the financial clients in multiple geographies.

As part of this they have made use of the Open source Kafka (OSK) for real time payment processing for financial entities within a region.

During our workshop we collaborated with Mojaloop SMEs to take a deep dive into Central Ledger application architecture and its workflow. We also reviewed the infrastructure used for deployments.

After covering the current state of application architecture we reviewed and discussed future requirements and objectives for the Central Ledger platform.

Please refer to the remaining sections of the report for the details of high availability and optimization of the Kafka platform.

Recommendations for High Availability

Definition of High Availability

High Availability is defined as the availability of the Kafka Platform to serve the client requests and to recover from failure scenarios as quickly as possible.

Managing Garbage Collection for High Availability

- **Enable GC logging in Broker JVM:** set environment variable GC_LOG_ENABLED=”true” and restart broker.
- **Monitor zookeeper timeouts with JMX Metric for > 0:** kafka.server:type=SessionExpireListener,name=ZooKeeperExpiresPerSec

Garbage collection can lead to kafka soft failures and those should be avoided. Garbage collection is a background Java thread that pauses processes to reclaim unused memory. Long Garbage Collection periods in the Broker should be avoided. If GC exceeds the Broker parameter **zookeeper.session.timeout.ms** (6 seconds, by default), the Broker will appear to be offline and the leaders hosted by that Broker will be re-elected unnecessarily. Also, if a Consumer has a session timeout with the Consumer Group Coordinator (a Broker), it will leave and rejoin its Consumer Group. The Broker setting zookeeper.session.timeout.ms determines how long a Broker can be offline before it is considered dead (default 6 seconds). Monitor ZooKeeperExpiresPerSec for any value greater than 0.

Garbage Collection Recommendations:

- Use the G1 Garbage Collector (Since Java 7) or Z Garbage Collector (since Java 11).
- Enable GC Logging. If Brokers are losing sessions from ZooKeeper (shown as session expiration errors in the server.log),there is likely a ZooKeeper connection problem. Enabling GC logging will help determine if long GC times are the cause. Long GC times are one of the most common reasons for session expiration.
- Check that the connection between Brokers and ZooKeeper is good. Otherwise, ZooKeeper may falsely detect a Broker as dead. Reference - https://docs.confluent.io/current/kafka/deployment.html#jvm

Kubernetes - Setup Pod Disruptor Budget

Configure PodDisruptionBugdet to ensure a minimum number of zookeeper servers available. Eg. For a five pod Zookeeper ensemble, set minAvailable = 3.

Trade-off: Do not reduce zookeeper instances below quorum otherwise kafka writes will fail.

Example setting -

```
$ kubectl get pdb zookeeper -oyaml | c -l yaml
apiVersion: policy/v1beta1
kind: PodDisruptionBudget
...
spec:
        minAvailable: 3
                selector:
                        matchLabels:
                                namespace: operator
                                type: zookeeper
...
```

Configuring Broker for High Availability

- Set **num.recovery.threads.per.data.dir =** number of CPU cores **OR** directory count in **log.dir** broker configuration property **OR** the number of disks in the RAID set.

This helps in faster recovery if a Broker had a hard failure. Broker will attempt to do log recovery. The Broker doesn't know when it failed or what state the log is in. For each Partition, the Broker will read the last flushed offset from the recovery-point-offset-checkpoint file and read every message from that offset to the end of the Commit Log to verify the CRC. At the first offset where the CRC doesn't match, the Broker will truncate the log from that point on and start replicating from the Leader.

Log recovery can be I/O intensive because of the amount of reads and writes. Recovery may also be CPU intensive if the logs are compressed. Recovery time will be improved by using multiple threads, one per Partition. By default, the Broker starts one recovery thread per Partition directory, which is sufficient for deployments where there is one disk per log directory.

- **Set min.insync.replicas = 1 for high availability.**

When a producer sets acks=all (or acks=-1), min.insync.replicas specifies the minimum number of replicas that must acknowledge a write for the write to be considered successful. If this minimum cannot be met, then the producer will raise an exception.

By setting this value low (e.g.,min.insync.replicas=1), the system will tolerate more replica failures. As long as the minimum number of replicas is met, the producer requests will continue to succeed, which increases availability for the partition.

Trade-off: Lower value for **min.insync.replicas** can provide less durability in case there are additional out-of sync replicas, and producers will mark messages committed by brokers when only one replica committed and not others. For durability, keep this value high enough (eg. 3), so that the producer marks the message committed only when more replicas mark it as committed.

- **Avoid Higher partition Count**

Higher partition counts may increase parallelism, but having more partitions can also increase recovery time in the event of a broker failure. All produce and consume requests will pause until the leader election completes, and these leader elections happen per partition. So take this recovery time into consideration when choosing partition counts.

Trade-off: Very low partition count will lower the overall throughput.

- **Set unclean.leader.election.enable=true**

Broker failures will result in partition leader elections. To optimize for high availability, new leaders can be allowed to be elected even if they were removed from the ISR list. To make this happen, set the configuration parameter unclean.leader.election.enable=true. This makes leader election happen more quickly, which increases overall availability.

Trade-off: Broker election happens automatically, you do have control over which brokers are eligible to become leaders. To optimize for durability, new leaders should be elected from just the ISR list to prevent the chance of losing messages that were committed but not replicated. So settings for high availability will decrease durability probability.

Configuring Consumers for High Availability

- **Set session.timeout.ms = 30000** milliseconds (default 45000)

Helps to detect hard failure of consumers (eg. SIGKILL) quickly and initiate recovery.

This timeout value is used to detect client failures when using Kafka's group management facility. The consumer sends periodic heartbeats to indicate its liveness to the broker. If no heartbeats are received by the broker before the expiration of this session timeout, then the broker will remove this client from the group and initiate a rebalance of partitions. The value must be in the allowable range as configured in the broker configuration by **group.min.session.timeout.ms** and **group.max.session.timeout.ms**.

Consumers can share processing load by being a part of a consumer group. If a consumer unexpectedly fails, Kafka can detect the failure and rebalance the partitions amongst the remaining consumers in the consumer group. The consumer failures can be hard failures (e.g SIGKILL) or soft failures (e.g. expired session timeouts), and they can be detected either when consumers fail to send heartbeats or when they fail to send poll() calls.

Trade-off: session.timeout.ms should not be set too low so as to trigger soft failure of consumers.

Tuning Virtual Memory for High Availability

- Set **vm.swappiness = 1** (Default: 60) to minimize memory swapping.
- Set **vm.dirty_ratio=80** (Default: 20) to decrease the frequency of blocking flushes (synchronous)
- Set **vm.dirty_background_ratio=5** (Default: 10) to increase the frequency of non-blocking background flushes (asynchronous)

The native Linux mechanism of swapping processes from memory to disk can cause serious performance limitations in Kafka. Setting vm.swappiness=1 prevents the system from swapping processes too frequently but still allows for emergency swapping instead of killing processes.

Trade-off: It is not recommended to use a value of 0, because it would never allow a swap under any circumstances, thus forfeiting the safety net afforded when using this parameter.

vm.dirty_ratio and vm.dirty_background_ratio: For good performance of Kafka service it is recommended to have more frequent background flushes and less frequent blocking flushes of unflushed (i.e., "dirty") memory.

The **vm.dirty_ratio** is the highest percentage of memory that can remain unflushed before Linux blocks I/O. If the ratio is set low, these blocking flushes happen more frequently, which degrades Kafka's performance and prevents Consumers from benefiting from zero copy transfer. High ratios cause less frequent flushes, so we set the value to 80.

Trade-off: Be aware that if a Broker has a sudden failure, all unflushed data is lost on that Broker. However, since production data is typically being replicated to other Brokers, this should not usually be a concern.

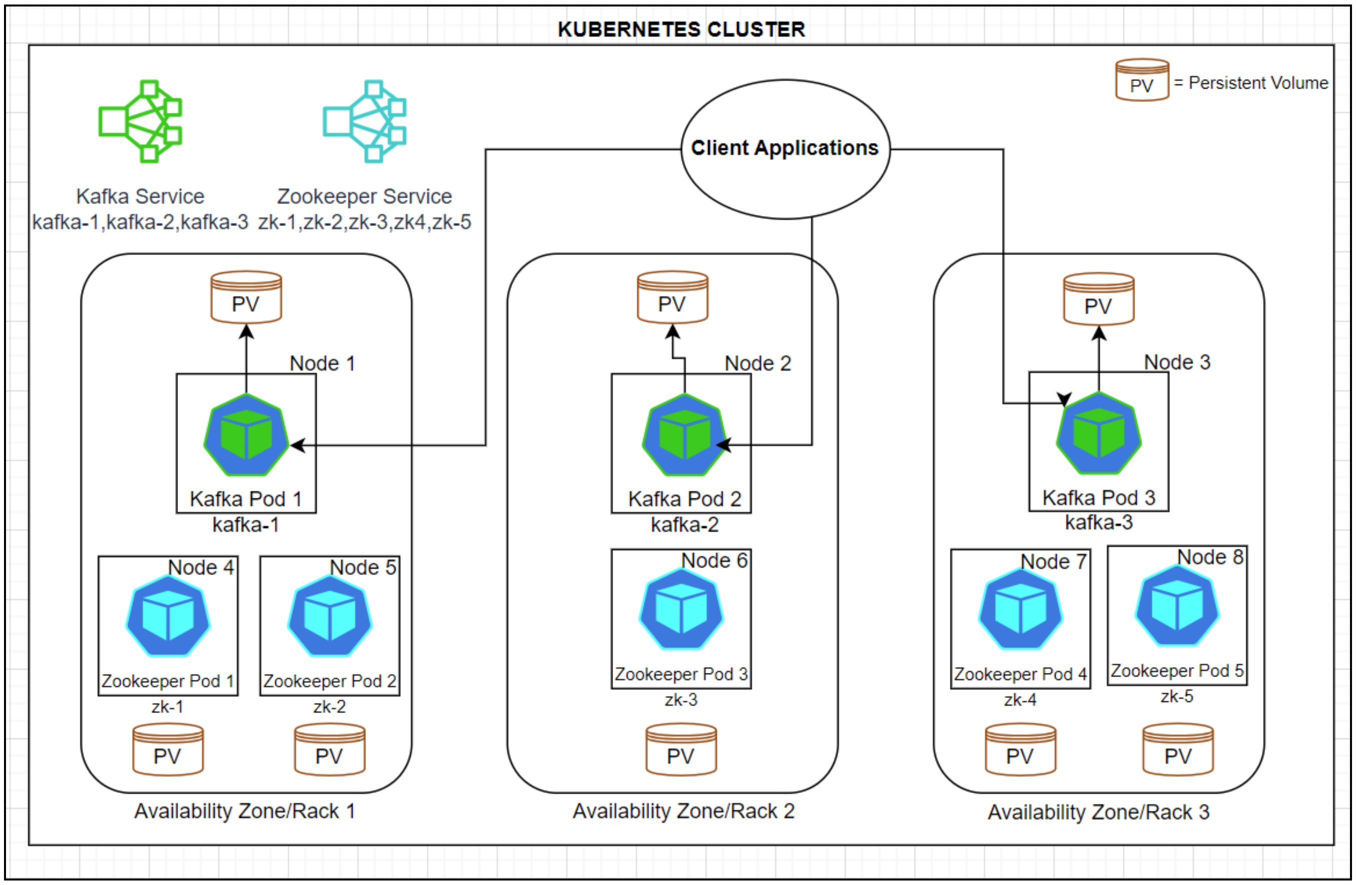
The **vm.dirty_background_ratio** is the percentage of system memory that can be dirty before the system can start writing data to disks in the background without blocking I/O. By decreasing this setting, we increase the frequency of non-blocking background flushes.

Trade-off: On the one hand, we want to hold a large page cache to take advantage of zero copy transfer, but on the other hand, we also want to avoid building up too much dirty memory and forcing a blocking flush.

Note: update /etc/sysctl.conf file or use command sysctl -w <VM property>=<VM property value> to set the three VM configs.

Kafka Kubernetes Deployment Architecture

High Level Deployment Architecture for Availability



Guidelines for high availability and reliability

Broker Deployment

1. Settings for High Availability in case of one planned and one unplanned failure -
   - A minimum in-sync replicas of 2 for topic partitions.
   - A replication factor of 3 for topics.
   - At least 3 Kafka brokers, each running on different nodes.
   - Nodes spread across three kubernetes availability zones.

2. High Availability - Never use a single instance of a kafka service containing a single pod.

3. High Availability - If N is the number of Pods required to serve peak traffic then setup N + 2 Pods in the Kafka cluster.

4. High Availability - Setup a PodDisruptionBudget, otherwise there is nothing that would prevent all replicas of your service to get evicted at once during node upgrades of the underlying cluster.

5. High Availability - Deploy Kafka Cluster as Kubernetes StatefulSets. This gives them persistent volumes for storage and it also gives each Pod in the cluster its own DNS entry via Headless services.

6. High Availability - Local Storage is recommended for on-premise deployment and shared storage for cloud deployments of Kafka service.

7. High Availability - Define headless service for each Kafka Broker. This gives a unique service name for each broker and is configured in advertised.listeners for that broker.

8. Reliability - Use internal Kubernetes IPs for communication between brokers and with clients on the same Kubernetes cluster.

9. High Availability - Configure Kubernetes liveness, readiness and startup probes to trigger an automatic reschedule of a Kafka pod that is not responding anymore.

10. High Availability - Kafka heavily utilizes the open source page cache to buffer data. Kernel manages page cache, which all Pods use. The KAFKA_HEAP_OPTS environment variable controls the JVM heap size of the Kafka brokers. -Xms=2G -Xmx=2G is sufficient for most deployments to begin.

11. High Availability - Whether or not the JVM process running the broker is still alive determines the liveness of the broker. A readiness check to determine if the application can accept requests ultimately decides readiness. Example -
```
readinessProbe:
exec:
command:
- sh
- -c
- "/opt/kafka/bin/kafka-broker-api-versions.sh --bootstrap -server=localhost:9093"
```

Zookeeper Deployment

1. High Availability - Never use a single instance of zookeeper service containing a single pod.

2. High Availability - If N is the number of Pods required to serve peak traffic then setup N + 2 Pods in the Zookeeper cluster.

3. High Availability - Setup a PodDisruptionBudget, otherwise there is nothing that would prevent all replicas of your service to get evicted at once during node upgrades of the underlying cluster.

4. High Availability - Deploy Zookeeper cluster as Kubernetes StatefulSets. This gives them persistent volumes for storage and it also gives each Pod in the cluster its own DNS entry via Headless services.

5. High Availability - ZooKeeper uses the JVM heap. initial allocation 512M and increase up to 4 GB RAM. ZooKeeper is not a CPU intensive application. For a typical deployment, allocate 2 CPUs and adjust as necessary. Zookeeper custom resource configuration -
   ```
   zookeeper:
     name: zookeeper
     replicas: 5
     resources:
       requests:
       cpu: 200m
       memory: 4G
   ```

6. High Availability - Configure Kubernetes liveness, readiness and startup probes to trigger an automatic reschedule of a Zookeeper pod that is not responding anymore.

Miscellaneous deployment related points to consider

1. High Availability - Use an incremental deployment strategy that rolls out any update to a subset (the canary set) of kafka or zookeeper service first, and only continue the rollout after verification of correct operation on the canary set.

2. High availability and security - Avoid using hostPath volumes for any storage requirement in Kafka Cluster as they present security risk.

3. Reliability - Continuously upgrade clusters to catch up with the fast pace of Kubernetes releases.

4. High availability - Set appropriate values for the following Kubernetes Deployment parameters to avoid any latency due to JVM heap during restarts - **minReadySeconds** (the minimum number of seconds for which a newly created Pod should be ready without any of its containers crashing, for it to be considered available. Set to 2), **maxUnavailable** (maximum number of Pods that can be unavailable during the update process, set to 0), **maxSurge** (maximum number of Pods that can be created over the desired number of Pods. Set to 1). Tune the settings based on observations and metrics.

5. Reliability - Use the following for stateful services like Kafka and Zookeeper -
   - Kubernetes - Primitives PersistentVolumes and PersistentVolumeClaims
   - Google Kubernetes Engine (GKE) - StorageClass gce-pd
   - Azure Kubernetes Service (AKS) - StorageClass azure-disk
   - Amazon Elastic Kubernetes Service (Amazon EKS) - StorageClass aws-ebs

6. High availability - Cloud Compute Instances - Apache Kafka is memory (RAM) intensive. Following instance types for high availability are recommended -
    - GKE - n1-highmem-2–n1-highmem-64
    - AKS - G-Series
    - EKS - r3.large, r3.8xlarge, x1.32xlarge

Recommendations for Optimizing Kafka Platform

Throughput: Amount of data moving through Kafka per second. Measured in Megabytes per second.

Latency: Delay from the time data is written to the time it is read. Measured in milliseconds.

Higher Throughput

Make the following configuration updates on producer for higher throughput -

batch.size = 1000000 (kB) default 16 kB

linger.ms = 100 milliseconds , default = 0

buffer.memory = Set at least equal to Number of partitions multiplied by batch.size value. If this calculated value comes less than 32 MB which is the default value, then set the default value only.

Exactly when a message is pushed from the queue on producer end to the Brokers is determined by either how long the messages have been in the queue (linger.ms) or the amount of data (batch.size). The default behavior is for the Producer to push messages in real-time so linger.ms defaults to 0; i.e., send messages as soon as they arrive.

Compression of messages for higher throughput

On producer set **compression.type = gzip OR snappy OR lz4 OR zstd**

On broker set **compression.type= producer**

Batch size refers to compressed size if compression is enabled. The compression type used by a Producer is noted as an attribute in the messages that it produces. This allows multiple Producers writing to the same Topic to use different compression types. Consumers will decompress messages according to the compression type denoted in the header of each message.

Trade-off: The compression.type can be set on Brokers, Topics, or Producers. On Brokers and Topics, the default is compression.type=producer, which means the compression codec of the Producer is respected. This can result in a single Topic containing messages of various compression types. If the Broker or Topic has its own compression type set, then all messages will be compressed with the specified codec. This puts extra work on the Broker, but will save space. Disk space is usually relatively cheap compared to compute time, so usually it is best to leave Broker and Topic compression.type settings to the producer.

Minimal Latency

Many of the Kafka configuration parameters discussed in the section on throughput have default settings that optimize for latency. Thus, those configuration parameters generally don't need to be adjusted. For example **batch.size** = 16 kB, **linger.ms** = 0 milliseconds and **compression.type=none**.

Trade-off: Keeping the values as defaults will not let throughput improve, so tune these values periodically to strike a balance between latency and throughput. Observe JMX metrics and then make a decision.

Monitoring io-ratio and io-wait-ratio For Optimization

- **io-ratio:** Defined as the fraction of time the client spent on producing/retrieving data to/from the broker.
- **io-wait-ratio:** Fraction of the time client is idle.

These are two JMX Metrics in Kafka producer and consumer. Helpful to resolve "slowness" of producer or consumer.

· If io-wait-ratio is close to 1, it indicates that the client is mostly idle and the bottleneck is likely on the Broker.

· If io-ratio is close to 1, it indicates that the client is mostly busy interacting with the Brokers. If the client is a Producer, it may be appropriate to do more batching or compression to increase throughput. If the client is a Consumer, it may be appropriate to increase max.partition.fetch.bytes or increase the size of the Consumer Group to achieve higher throughput. Remember that the most Consumers that one can run in a Consumer Group is limited by the number of Partitions in the consumed Topics.

• If io-ratio and io-wait-ratio are both close to 0, it indicates that the client is the bottleneck. For Producers, make sure that the Producer callback is not doing expensive operations (e.g., writing to a log4j file). For Consumers, make sure that there is no expensive step in processing each returned record.

Monitoring batch-size-avg and compression-rate-avg

batch-size-avg: The average number of bytes sent per partition per-request. A batch-size-avg much smaller than batch.size indicates inefficient batching, so consider increasing linger.ms.

compression-rate-avg: The average compression rate of record batches, defined as the average ratio of the compressed batch size over the uncompressed size. A low compression-rate-avg would indicate that compression is not creating much space savings and so may not be worth the system resources to run the compression.

Producer Settings for Durability and Ordering

Set **acks=all** and **enable.idempotence = true** and at topic level set **min.insync.replicas = 2**

For durability: the topic level configuration, min.insync.replicas, works along with the acks configuration. This setting tells the broker to not allow an event to be written to a topic unless there are N replicas in the ISR. Combined with acks=all, this ensures that any events that are received onto the topic will be stored in N replicas before the event send is acknowledged.

As an example, if we have a replication factor of 3 and min.insync.replicas set to 2 then we can tolerate one failure and still receive new events. If we lose two nodes, then the producer sending requests would receive an exception informing the producer that there were not enough replicas. The producer could retry until there are enough replicas, or bubble the exception up. In either case, no data is lost.

Event Ordering guarantees: Are handled mainly by Kafka's partitioning and the fact that partitions are append-only immutable logs. Events are written to a particular partition in the order they were sent, and consumers read those events in the same order. But failures can cause duplicate events to be written to the partition which will throw off the ordering.

To prevent this, we can use the Idempotent Producer, which guarantees that duplicate events will not be sent in the case of a failure.

To enable idempotence, set enable.idempotence = true on the producer which is the default value as of Kafka 3.0. With this set, the producer tags each event with a producer ID and a sequence number. These values will be sent with the events and stored in the log. If the events are sent again because of a failure, those same identifiers will be included. If duplicate events are sent, the broker will see that the producer ID and sequence number already exist and will reject those events and return a DUP response to the client.

End-to-end ordering guarantee: Combine acks=all, producer idempotence, and keyed events. Events with a specific key will always land in a specific partition in the order they are sent, and consumers will always read them from that specific partition in that exact order.

Trade-off: min.insync.replicas should not be lowered to 0 or else the durability will be affected, producers will mark messages as committed in case of out of sync replicas also.

Hardware and Infrastructure Optimizations

Increase File Descriptor Limit

Kafka uses a very large number of files and a large number of sockets to communicate with the clients. All of this requires a relatively high number of available file descriptors. Most modern Linux distributions have only 1,024 file descriptors allowed per process. File descriptor count should be increased to at least 100,000, and possibly more. This process is dependent on a particular OS and distribution. For example Red Hat -

To see the current file descriptor for a user su - <linux user> $ ulimit -n To set file descriptor limit via command su - <linux user> $ ulimit -n 100000. To permanently set file descriptor limit Add or update below lines to file /etc/security/limits.conf

```
<linux process>      soft    nofile      100000

<linux process>     hard    nofile      100000
```

Soft limit in the first line defines the number of file handles or open files that the user will have after they log in. Hard Limit defines the number up to which the limit can be increased for the user via ulimit command. Keeping them the same will incur the maximum availability of open file descriptors as recommended for kafka processes.

To calculate the current mmap number, count the .index files in the Kafka data directory. The .index files represent the majority of the memory mapped files - find . -name '*index' | wc -l

Set the vm.max_map_count for the session. This will calculate the current number of memory mapped files. The minimum value for mmap limit (vm.max_map_count) is the number of open files ulimit.

Trade-off: Set vm.max_map_count sufficiently higher than the number of .index files to account for broker segment growth.

```
sysctl -w vm.max_map_count=262144
```

Set the vm.max_map_count so that it will survive a reboot use this command:

```
echo 'vm.max_map_count=262144' >> /etc/sysctl.conf
```

```
sysctl -p
```

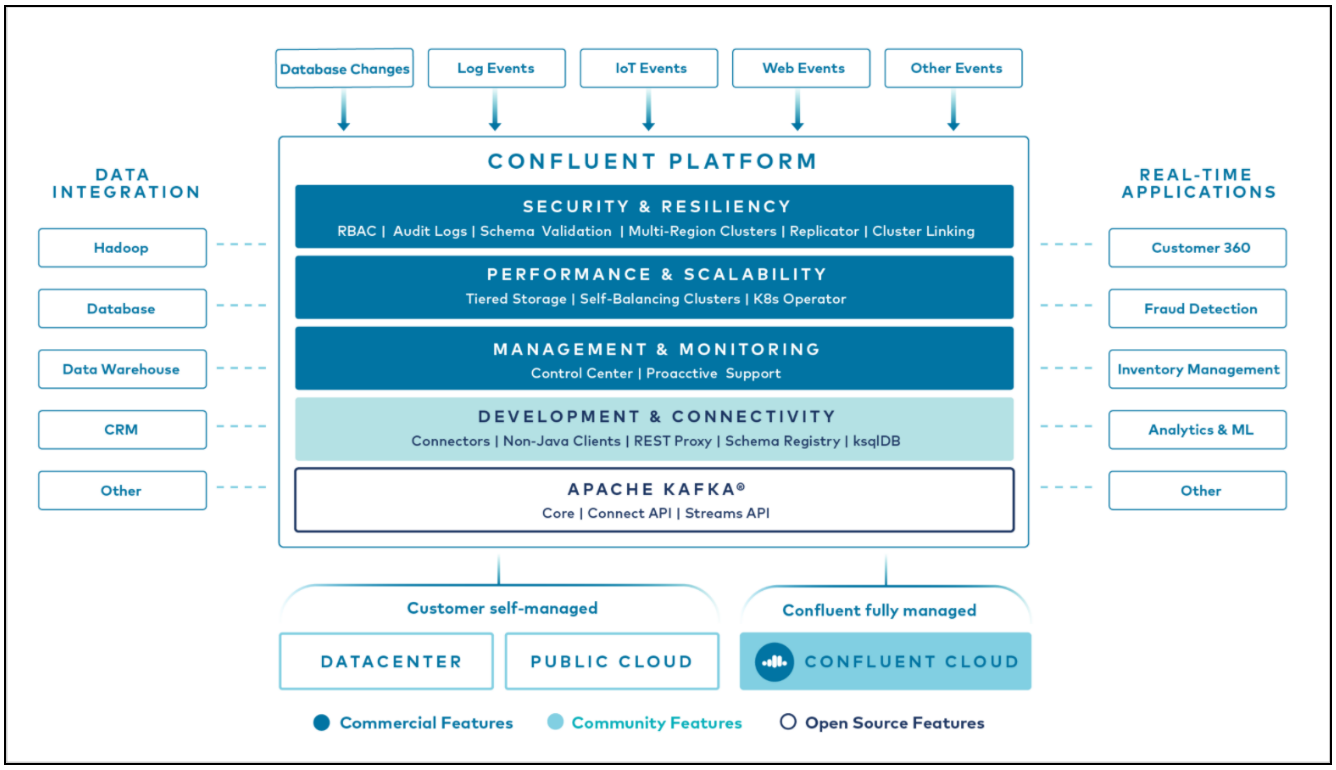
Confluent Recommendations to Optimize and Scale

About Confluent

Confluent provides a full-scale data streaming platform that enables easy access, storage and management of data as continuous, real-time streams. Built by the original creators of Apache Kafka®, Confluent expands the benefits of Kafka with enterprise-grade features while removing the burden of Kafka management or monitoring.

Confluent Platform lets customers focus on how to derive business value from your data rather than worrying about the underlying mechanics, such as how data is being transported or integrated between disparate systems. Specifically, Confluent Platform simplifies connecting data sources to Kafka, building streaming applications, as well as securing, monitoring, and managing your Kafka infrastructure.

CONFLUENT PLATFORM COMPONENTS



Confluent is cloud native, offering 10x Apache Kafka service powered by the Kora Engine, that enables elastic scalability, guaranteed resiliency, and low latency. The solution is also complete, meaning it has 120+ pre-built connectors across all your apps and data systems to create an enterprise-grade data streaming solution with security and governance. It is also everywhere, meaning it is available as a fully managed service on AWS, Azure, and Google Cloud with flexibility for on-prem and private cloud workloads.

Common benefits customers see when leveraging Confluent Cloud include:

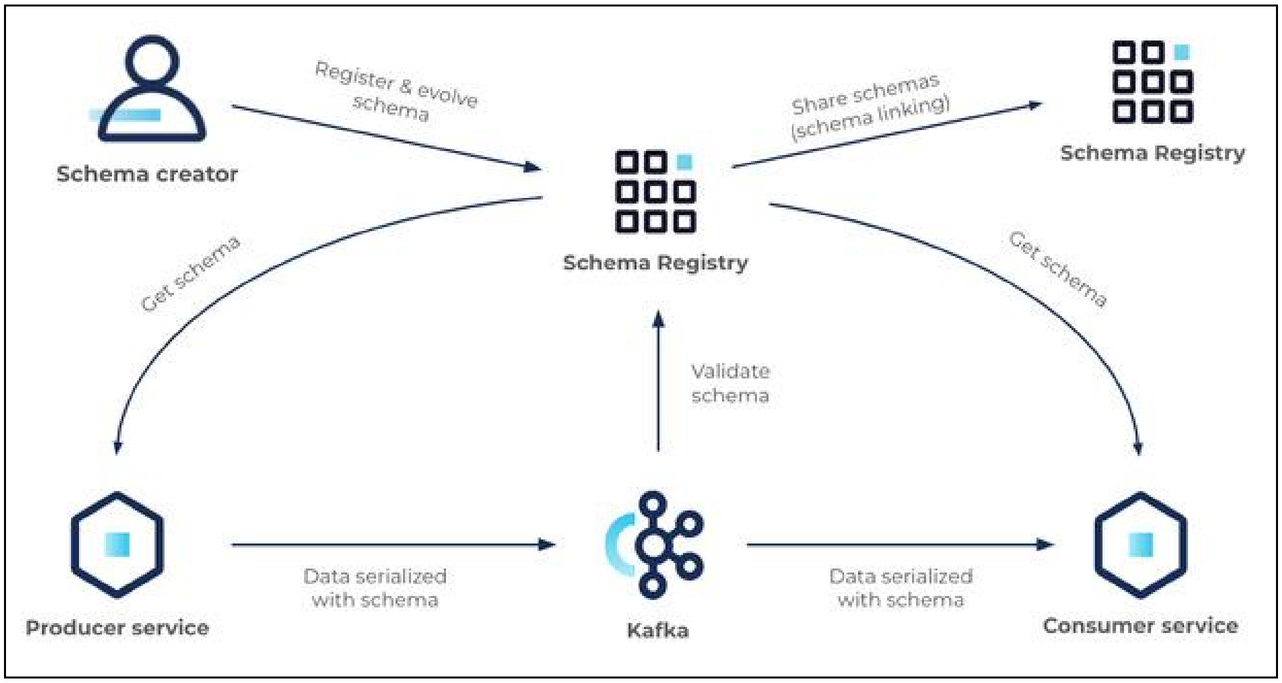
- Serverless solution that is automated and fully managed with zero ops
- Elastic scaling to avoid over-provisioning infrastructure
- Infinite and tiered storage to retain data at any scale without growing compute
- High availability guaranteed 99.99% uptime SLA with built-in failover and multi-AZ replication
- No ZooKeeper management that is completely abstracted away
- No-touch patching and upgrades resulting in zero downtime for maintenance

Learn more at **confluen.io »**

Schema Management

Schema Registry provides a centralized repository for managing and validating schemas for topic message data, and for serialization and deserialization of the data over the network. Producers and consumers to Kafka topics can use schemas to ensure data consistency and compatibility as schemas evolve. Schema Registry is a key component for data governance, helping to ensure data quality, adherence to standards, visibility into data lineage, audit capabilities, collaboration across teams, efficient application development protocols, and system performance.

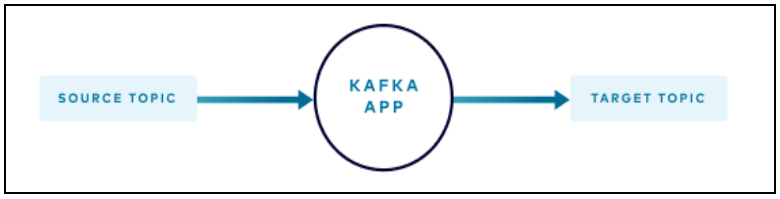
SCHEMA REGISTRY AND BROKER WORKFLOW



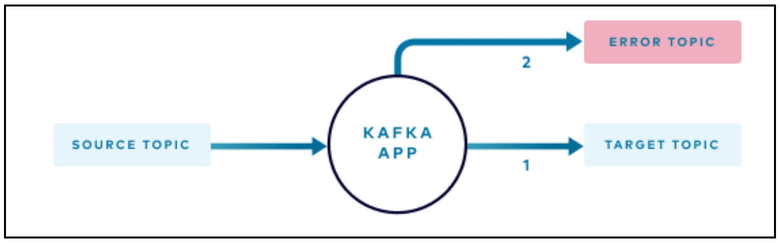
Kafka Error Handling Patterns

Following are some of the patterns to handle errors and retries in Kafka event streaming application layer -

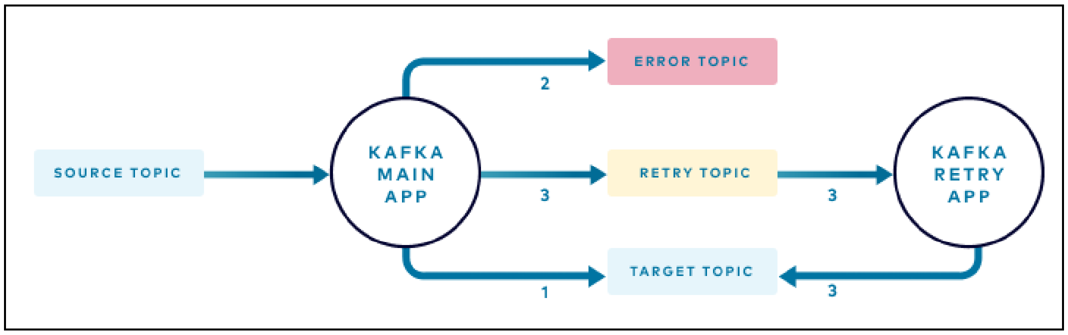
Stop on Error: Applicable when all input events must be processed in order without exceptions. When an exception occurs in input processing then the application stops. This prevents further processing and incorrect ordering of events. After manual intervention to fix the problematic input application is started to process new input events.



Dead Letter Queue: Events that cannot be processed by the application are routed to an error topic while the main stream continues. It's important to note that in this approach there are no conditions that require or support a retry process. In other words, an event can be processed successfully, or it is routed to an error topic in case of unexpected message format or missing required attributes.



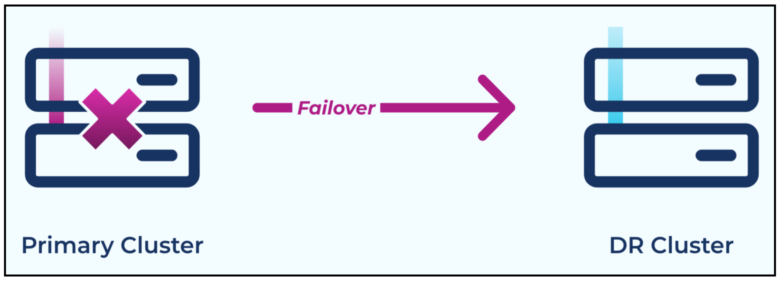
Add Retry Topic and Retry Application: Add a retry topic and retry application logic so that the system has the ability to process most events right away while delaying the processing of other events until the required conditions are met to process them. Under normal circumstances, the application processes each event in the source topic and publishes the result to the target topic events that cannot be processed, for example, those that don't have the expected format or are missing required attributes, are routed to the error topic. Events for which dependent data is not available are routed to a retry topic where a retry instance of your application periodically attempts to process the events.



Disaster Recovery

A Disaster Recovery (DR) cluster is implemented that is available to failover should your primary cluster experience an outage or disaster. Confluent Cluster Linking keeps DR cluster in sync with data, metadata, topic structure, topic configurations, and consumer offsets so that you can achieve low recovery point objectives (RPOs) and recovery time objectives (RTOs), often measured in minutes.

Cluster Linking for DR does not require an expensive network, complicated management, or extra software components, and because Cluster Linking preserves offsets and syncs consumer offsets, consumer applications of all languages can failover and pick up near the point where they left off, achieving low downtime without custom code or interceptors. Alternatively Kafka MirrorMaker can be used to replicate the data and have a secondary kafka cluster ready in case of disaster.


Failover
Primary Cluster
DR Cluster