Cluster Linking for Failover and Disaster Recovery on Confluent Cloud

56–71 minutes

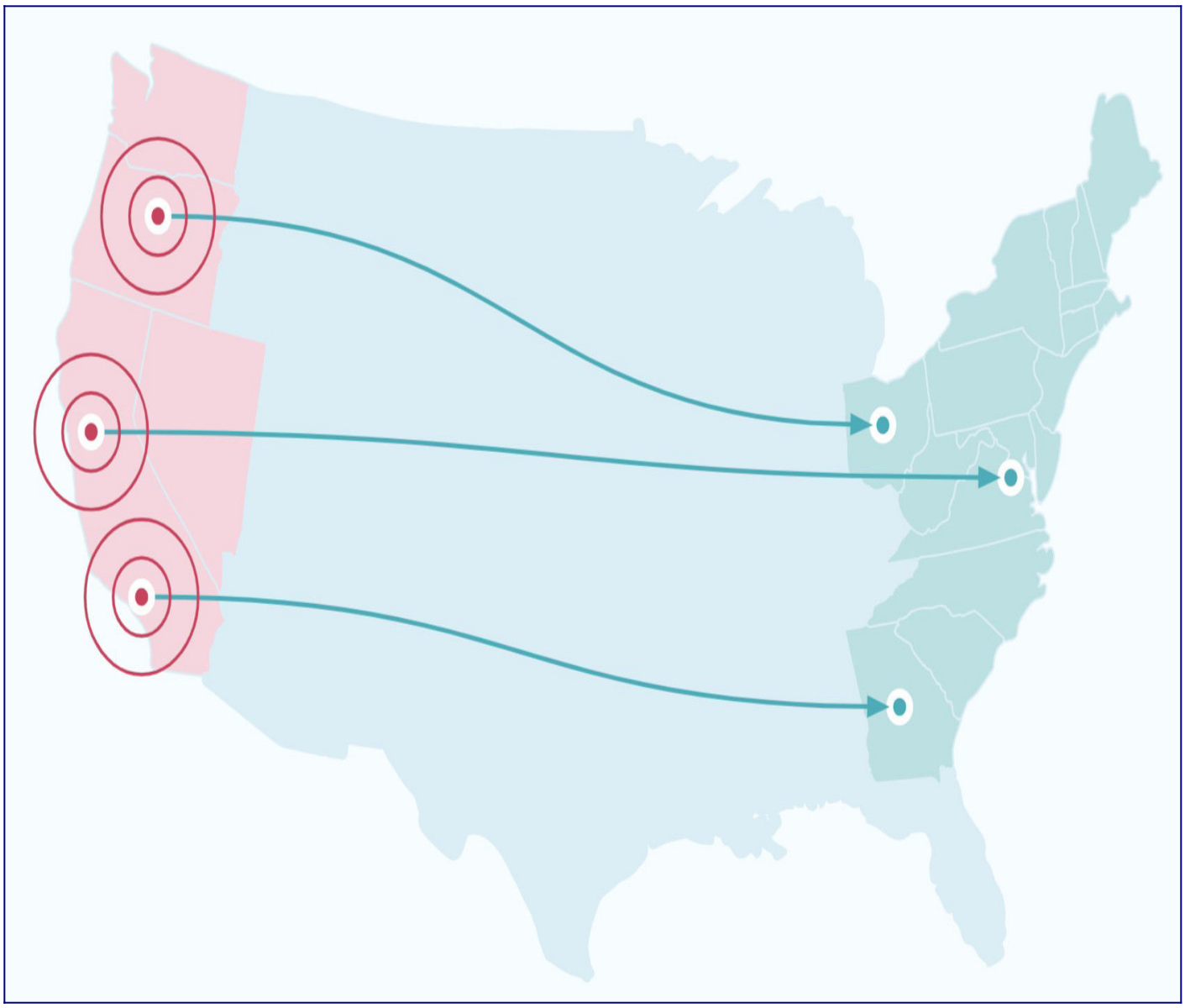
Deploying a disaster recovery strategy with Cluster Linking can increase availability and reliability of your mission-critical applications by minimizing data loss and downtime during unexpected disasters, like public cloud provider outages. This document will explain how to design your disaster recovery strategy and execute a failover.

To see what clusters can use Cluster Linking, see the supported cluster types.

Goal of Disaster Recovery with Cluster Linking

You start with a “primary” cluster that contains data in its topics and metadata used for operations, like consumer offsets and ACLs. Your applications are powered by producers that send data into those topics, and consumers that read data out of those topics.

You can use Cluster Linking to create a disaster recovery (DR) cluster that is in a different region or cloud than the primary cluster. When an outage hits the primary cluster, the DR cluster will have an up-to-date copy of your data and metadata. The producers and consumers can switch over to the DR cluster, allowing them to continue running with low downtime and minimal data loss. Thus, your applications can continue to serve your business and customers during a disaster.



Tip

Have multiple “primary” regions? See the Active/Active Tutorial.

Recovery time objectives (RTOs) and recovery point objectives (RPOs)

A crucial input informing the design of your disaster recovery plan is the Recovery Time Objective(s) (RTO) and Recovery Point Objective(s) (RPO) that you hope to achieve. An RTO is the maximum amount of downtime that your system can have during an outage, measured as the difference between the time the outage occurs and the time your system is back up and running. An RPO is the maximum amount of data you are willing to risk losing because of an outage, measured as the difference between the last message produced to the failed cluster and the last message replicated to the DR cluster from the failed cluster.

Here are some quick reference definitions for relevant terms.

Term	Description
Recovery Point Objective (RPO)	In the event of failure, at which point in the data's history does the failover need to resume from? In other words, how much data can be lost during a failure? In order to have zero RPO, synchronous replication is required.
Recovery Time Objective (RTO)	In the event of failure, how much time can elapse while a failover takes place? In other words, how long can a failover take? In order to have zero RTO, seamless client failover is required.
Region	A synonym for a data center.
Disaster Recovery (DR)	Umbrella term that encompasses architecture, implementation, tooling, policies, and procedures that all allow an application to recover from a disaster or a full region failure.
Event	A single message produced or consumed to/from Confluent Cloud or Confluent Platform.
Millisecond (ms)	1/1,000th of a second

The RTO you can achieve with Cluster Linking depends on your tooling and failover procedures, since failing over your applications is your responsibility. There's no set minimum or maximum RTO.

The RPO you can achieve Cluster Linking is determined by the mirroring lag between the DR cluster and the primary cluster. Mirroring lag is exposed via Metrics API, via REST API, and in the Confluent Cloud Console.

Disaster Recovery requirements for Kafka clients

When implementing a disaster recovery ("DR") plan to failover from a primary cluster to a DR cluster, there are several aspects you must design into your Kafka clients, so that they can failover smoothly:

- Clients must bootstrap to the DR cluster once a failover is triggered
- Consumers must be able to tolerate a small number of duplicate messages ("idempotency")
- Consumers and producers must be tolerant of an RPO ("Recovery-Point Objective")

This section walks through these design requirements.

Clients must bootstrap to the DR cluster once a failover is triggered

When you detect an outage and decide to failover to the DR cluster, your clients must all switch over to the DR cluster to produce and consume data. To do this, your clients must do two things:

- When the clients start, they must use the bootstrap server and security credentials of the DR cluster. It is not best practice to hardcode the bootstrap servers and security credentials of your primary and DR clusters into your clients' code. Instead, you should store the bootstrap server of the active cluster in a Service Discovery tool (like Hashicorp Consul) and the security credentials in a key manager (like Hashicorp Vault or AWS Secret Manager). When a client starts up, it fetches its bootstrap server and security credentials from these tools. To

trigger a failover, you change the active bootstrap server and security credentials in these tools to those of the DR cluster. Then, when your clients restart, they will bootstrap to the DR cluster.

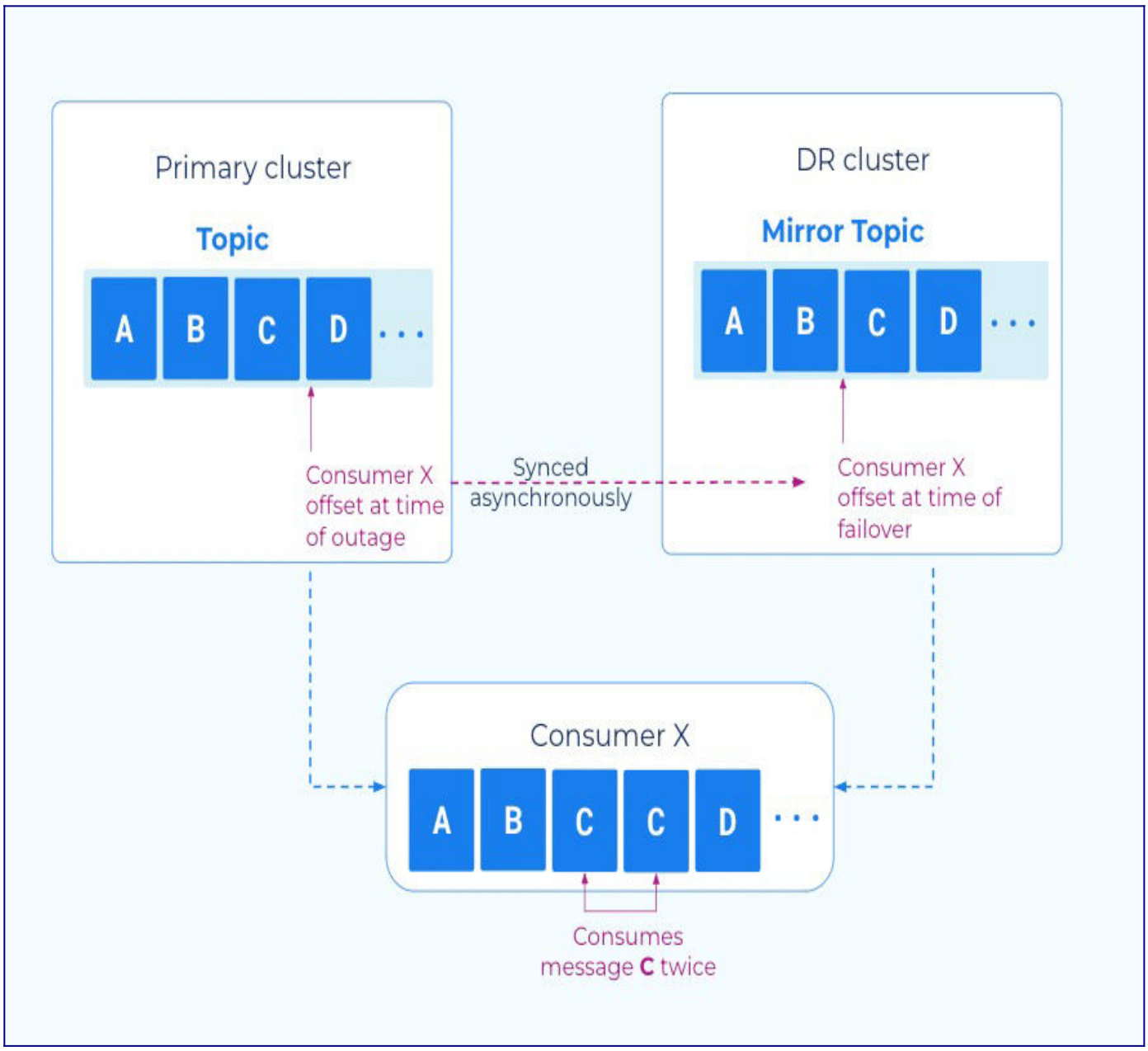
- Any clients that are still running need to stop and restart. If your primary cluster has an outage but not your clients (for example, if you have clients in a different region than the primary cluster that were unaffected by the regional cloud service provider outage), these clients will still be running and attempting to connect to the primary cluster. When you decide to failover, these clients need to stop running and restart, so that they will bootstrap to the DR cluster. There is not a mechanism built into Kafka to do this. There are several approaches you may take to achieve this behavior:
    - If a central Kafka operator manages all clients centrally, such as in a Kubernetes cluster, the operator can order all clients to shut down until the count of running clients is down to 0, and then can scale the clients back up.
    - You can add code wrapping your clients that polls the service discovery tool to check for a change in the bootstrap server. If the bootstrap server changes, the wrapping code restarts the clients.
    - Each team with Kafka clients can be paged and ordered to restart their clients.

How quickly your clients are able to bootstrap to the DR cluster will determine a large part of your recovery time after an outage. It is recommended that you practice the failover process so you can be sure that you can hit your RTOs (“Recovery Time Objectives”).

Clients must be be able to tolerate a small number of duplicate messages (idempotency)

Cluster Linking consumer offset sync gives your applications a low RTO by enabling your consumers to failover and restart very close to the point where they left off, without missing any new messages that were produced on the DR cluster. However, consumer offset sync is an asynchronous process. When consumers commit their offsets to the source cluster, the commit completes without also committing the offsets to the destination cluster. The offsets will are written to the destination cluster every `consumer.offset.sync.ms` milliseconds (a cluster link configuration that you can set as low as 1 second).

Because consumer offset sync is asynchronous, when an outage or disaster occurs, the most recent consumer offsets may not have been committed to the destination cluster, yet. So, when your consumer applications begin consuming on the destination cluster, the first few messages they receive may be messages they have already received. Consumer applications must be able to tolerate these duplicates on a failover.



Producers and consumers must be tolerant of a small RPO

Cluster Linking enables you to have a copy of your topic data on a second cluster, so that you don't lose all of your business-critical event data should a regional outage or disaster occur. However, Cluster Linking is an asynchronous process. When producers produce messages to the source cluster, they get an acknowledgement (also known as an "ack") from the source cluster independent of when, and often before, the cluster link replicates those messages to the DR cluster.

Therefore, when an outage occurs and a failover is triggered, there may be a small number of messages that have not been replicated to the DR cluster. Producer and consumer applications must be tolerant of this.

You can monitor your exposure to this via the mirroring lag in the Metrics API, CLI, REST API, and Confluent Cloud Console.

Setting up a Disaster Recovery cluster

For the Disaster Recovery (DR) cluster to be ready to use when disaster strikes, it will need to have an up-to-date copy of the primary cluster's topic data, consumer group offsets, and ACLs:

- The DR cluster needs up-to-date topic data so that consumers can process messages that they haven't yet consumed. Consumers that are lagging can continue to process topic data while missing as few messages as possible. Any future consumers you create can process historical data without missing any data that was produced before the disaster. This helps you achieve a low Recovery Point Objective (RPO) when a disaster happens.
- The DR cluster needs up-to-date consumer group offsets so that when the consumers switch over to the DR cluster, they can continue processing messages from the point where they left off. This minimizes the number of duplicate messages the consumers read, which helps you minimize application downtime. This helps you achieve a low Recovery Time Objective (RTO).
- The DR cluster needs up-to-date ACLs so that the producers and consumers can already be authorized to connect to it when they switch over. Having these ACLs already set and up-to-date also helps you achieve a low RTO.

Note

When using Schema Linking: To use a mirror topic that has a schema with Confluent Cloud Connect, ksqlDB, broker-side schema ID validation, or the topic viewer, make sure that Schema Linking puts the schema in the default context of the Confluent Cloud Schema Registry. To learn more, see How Schemas work with Mirror Topics.

To set up a Disaster Recovery cluster to use with Cluster Linking:

1. If needed, create a new Dedicated Confluent Cloud cluster with public internet in a different region or cloud provider to use as the DR cluster.
2. Create a cluster link from the primary cluster to the DR cluster. the cluster link should have these configurations:
    1. Enable consumer offset sync. If you plan to failover only a subset of the consumer groups to the DR cluster, then use a filter to only select those consumer group names. Otherwise, sync all consumer group names.
    2. Enable ACL sync. If you plan to failover only a subset of the Kafka clients to the DR cluster, then use a filter to select only those clients. Otherwise, sync all ACLs.
3. Using the cluster link, create a mirror topic on the DR cluster for each of the primary cluster's topics. If you only want DR for a subset of the topics, then only create mirror topics for that subset.

3. Enable auto-create mirror topics on the cluster link, which will automatically create DR mirror topics for the topics that exist on the source cluster. As new topics are created on your source cluster over time, auto-create mirror topics will automatically mirror them to the DR cluster.
    - If you only need DR for a subset of topics, you can scope auto-create mirror topics by topic prefixes or specific topic names.
    - For some use cases, it may be better to create mirror topics from the API call (POST /clusters/{cluster_id}/links/{link_name}/mirrors), CLI command (confluent

kafka mirror create), or Confluent Cloud Console *instead* of enabling auto-create mirror topics. For example, this may be preferable if your architecture has an onboarding process that topics and clients must follow in order to opt-in to DR. If you choose this option, whenever a new topic is created on the primary cluster that needs DR, you must explicitly create a mirror topic on the DR cluster.

With those steps, you create a Disaster Recovery cluster that stays up to date as the primary cluster's data and metadata change.

Whenever you create a new topic on the primary cluster that you want to have DR, create a mirror topic for it on the DR cluster.

Tip

Each Kafka client needs an API key and secret resource-scoped for each cluster that it connects to. To achieve a low RTO, create API keys on the DR cluster ahead of time, and store them in a vault where your Kafka clients can retrieve them when they connect to the DR cluster.

Monitoring a Disaster Recovery cluster

The Disaster Recovery (DR) cluster needs to stay up-to-date with the primary cluster so you can minimize data loss when a disaster hits the primary cluster. Because Cluster Linking is an "asynchronous" process, there may be "lag:" messages that exist on the primary cluster but haven't yet been mirrored to the DR cluster. Lagged data is at risk of being lost when a disaster strikes.

Monitoring lag with the Metrics API

You can monitor the DR cluster's lag using built-in metrics to see how much data any mirror topic risks losing during a disaster. The Metrics API's mirror lag metric reports an estimate of the maximum number of lagged messages on a mirror topic's partitions.

Viewing lag in the CLI

In the Confluent CLI, there are two ways to see a mirror topic's lag at that point in time:

- `confluent kafka mirror list` lists all mirror topics on the destination cluster, and includes a column called `Max Per Partition Mirror Lag`, which shows the maximum lag among each mirror topic's partitions. You can filter on a specific cluster link or mirror topic status with the `--link` and `--mirror-status` flags.
- `confluent kafka mirror describe <mirror-topic> --link <link-name>` shows detailed information about each of that mirror topic's partitions, including a column called `Partition Mirror Lag`, which shows each partition's estimated lag.

Querying for lag in the REST API

The Confluent Community REST API returns a list of a mirror topic's partitions and lag at these endpoints:

- `/kafka/v3/clusters/<destination-cluster-id>/links/<link-name>/mirrors` returns all mirror topics for the cluster link.

- `/kafka/v3/clusters/<destination-cluster-id>/links/<link-name>/mirrors/<mirror-topic>` returns only the specified mirror topic.

The list of partitions and lags takes this format:

```
"mirror_lags": [
  {
    "partition": 0,
    "lag": 24
  },
  {
    "partition": 1,
    "lag": 42
  },
  {
    "partition": 2,
    "lag": 15
  },
  ...
],
```

Failover considerations

When a consumer group first consumes from a mirror topic on your DR cluster, or when you run the `failover` command, note the following considerations.

Order of actions and promoting the Destination as post-failover active cluster¶

You should first stop mirror topics, and then move all of your producers and consumers over to the destination cluster. The destination cluster should become your new active cluster, at least for the duration of the disaster and the recovery. If it works for your use case, a suggested strategy is to make the **Destination cluster** your new, permanent active cluster.

Recover lagged data

There may be lagged data that did not make it to the destination before the disaster occurred. When you move your consumers, if any had not already read that data on the source, then they will not read that data on the destination. If/when the disaster resolves your source cluster, that lagged data will still be there. So, you are free to consume or handle it as fits with your use case.

For example, if the Source was up to offset 105, but the Destination was only up to offset 100, then the source data from offsets 101-105 will not be present on the Destination. The Destination will get new, fresh data from the producers that will go into its offsets 101-105. When the disaster resolves, the Source will still have its data from offsets 101-105 available to consume manually.

Lagged consumer offsets may result in duplicate reads

There may be lagged consumer offsets that did not make it to the destination before the disaster occurred. If this is the case, then when you move your consumers to the destination, they may read duplicate data.

For example, if at the time that you stop your mirroring:

- Consumer A had read up to offset 100 on the Source

- Cluster Linking had mirrored the data through offset 100 to the Destination
- Cluster Linking had last mirrored consumer offsets that showed Consumer A was only at offset 95

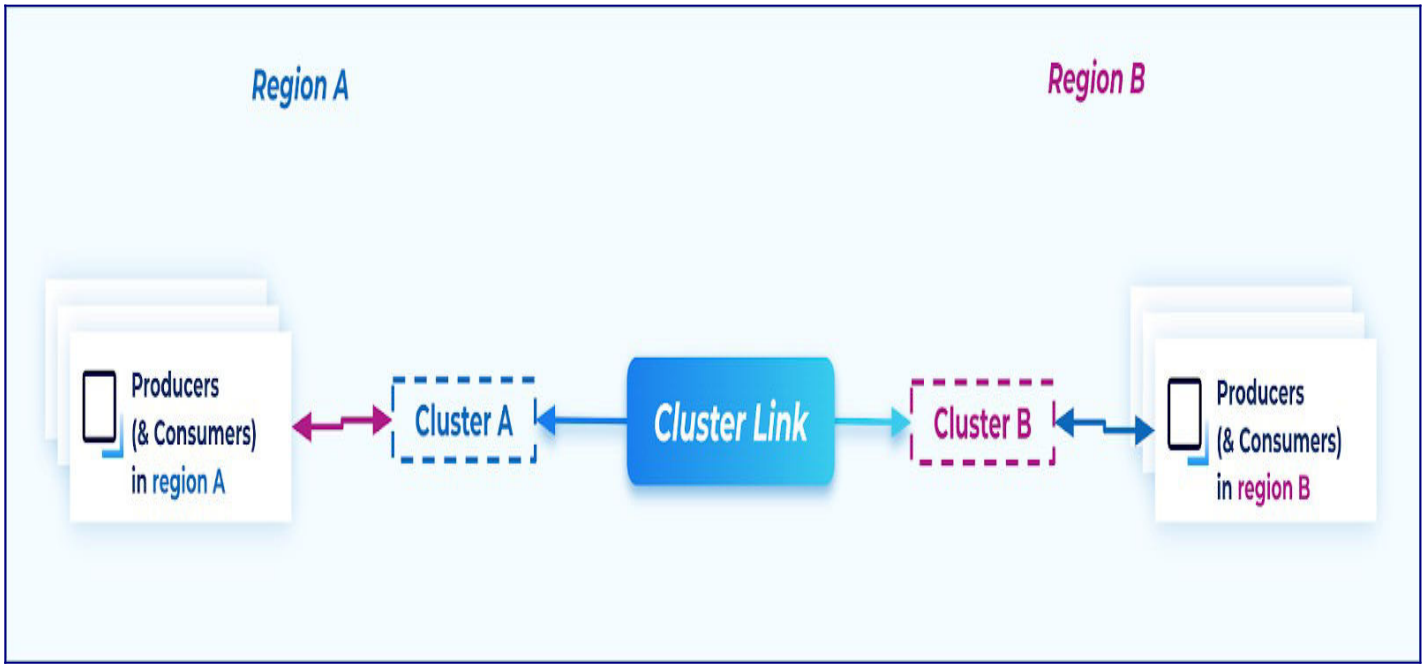
Then when you move Consumer A to the Destination, it may read offsets 96-100 again, resulting in duplicate reads.

Use consumer.offset.sync.ms

Keep in mind that you can configure `consumer.offset.sync.ms` to suit your needs (default is 30 seconds). A more frequent sync might give you a better failover point for your consumer offsets, at the cost of bandwidth and throughput during normal operation.

Active/Active Tutorial

For some advanced use cases, multiple regions must be *active* at the same time. In Kafka, an "active" region is any region that has active producers writing data to it. These architectures are called "active / active." While these architectures typically use two regions, the pattern is straightforward to extend to three or more active regions.



Active/Active: During steady state, producers & consumers are active in multiple regions

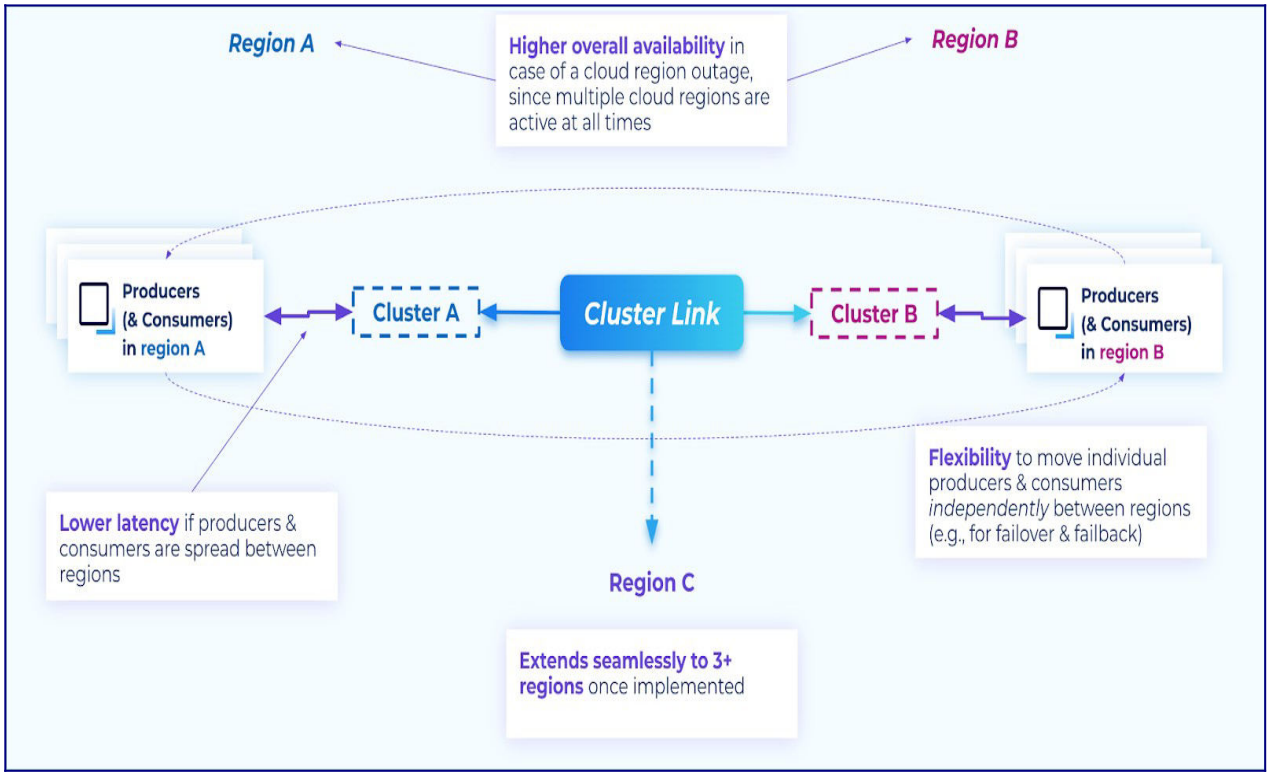
Benefits of active/active architecture¶

Some benefits of an active/active architecture include:

- **Latency** - If client applications are geographically dispersed in order to be closer to the customers they serve, the applications can connect to the nearest cloud region in order to minimize produce and consume latency. This can achieve lower latency reads & writes than having a single active region, which could be far away from clients in other regions.
- **Availability** - Since producers and consumers are active in multiple regions at all times, if any one cloud region has an outage, the other region(s) are unaffected. Thus, a cloud region

outage only affects a subset of traffic, which enables you to achieve even higher availability overall than the active / passive pattern.

- **Flexibility** - Since Kafka clients can produce or consume to any region at a given time, they can be moved between regions independently. Clients do not have to be aware of which region is “active” and which is “passive.” This is different from the active / passive pattern, which requires the producers and consumers to move regions at the same time, coordinated with the *failover* or *promote* command, in order to write to & read from the correct topics. The flexibility of the active / active architecture can make it easier to deploy Kafka clients at scale.
- **Extensibility** - Once an active / active architecture is implemented, new regions can be seamlessly added to the architecture with no changes to existing producers, consumers, or cluster links.



Benefits of an active/active architecture

Note

Active/active is not an extension of active/passive. Active/active has different requirements from active/passive, as explained below, and transitioning from active/passive to active/active is not seamless.

Requirements and constraints¶

Adopting an active/active architecture for Kafka comes with implementation requirements for the following tasks:

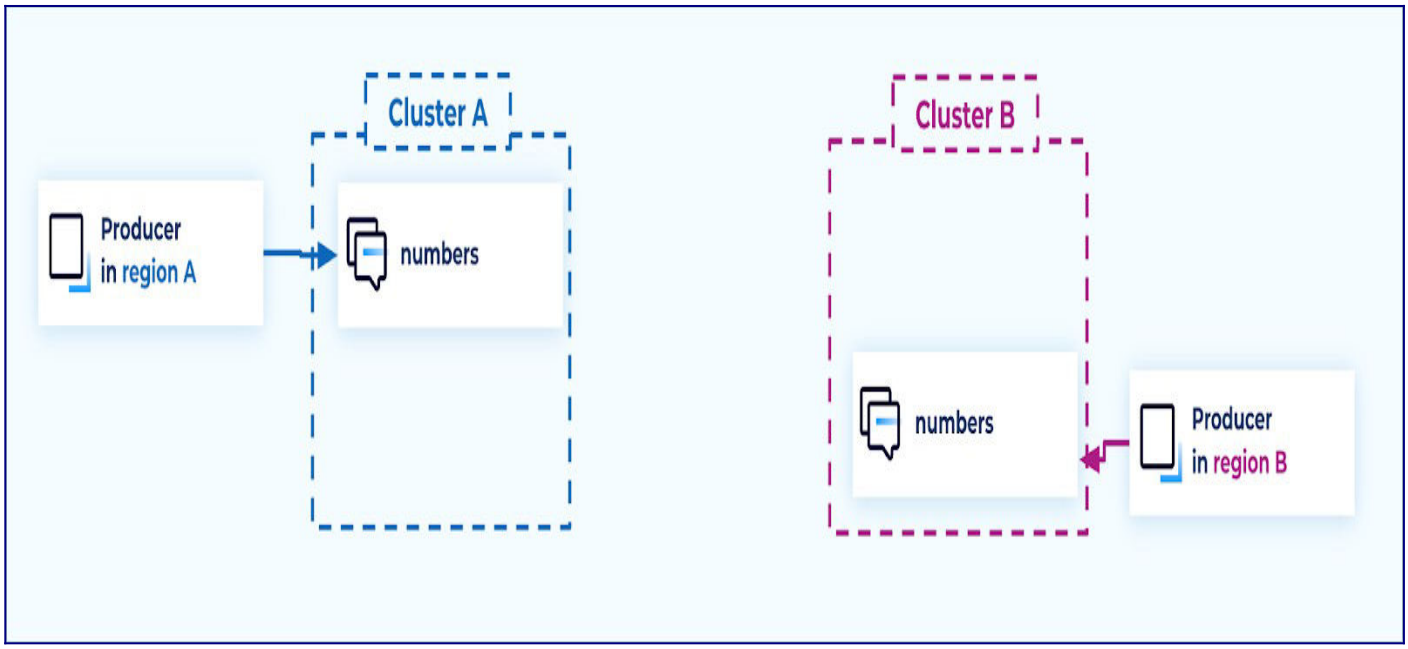
- Create writable topics
- Create cluster links
- Create mirror topics
- Set up consumer groups

- Set up security

Create writable topics¶

For each topic adopting an active/active setup, each region must have one writable (normal) topic. Producers always produce to that topic when connected to that region; meaning that if a producer moves to a different region, it will technically be producing to a different topic

It is recommended that these writable topics use the same name (for example, `numbers` in the image below), so that producers can always use the same topic name in all regions. This can also simplify security rules and schemas.



Create writable topics

Alternatively, topics can be named for the region that they are writable in, for example, `us-east-1.numbers` and `us-west-2.numbers`. This approach requires producers to be aware of the region they are producing to, and use the correct topic name.

Create cluster links¶

A cluster link in bidirectional mode is required between each pair of clusters.

- If there are only two clusters, then one cluster link suffices. If there are three regions and clusters, then three cluster links are required: A to B, B to C, and A to C.
- A cluster link's mode cannot be changed once created. A cluster link must be created with bidirectional mode from the outset.

If the writable topics have the same name on all clusters (as recommended), then the cluster link(s) should be created with `link.prefix`, which adds a prefix to the name of mirror topics created with the cluster link. It is best practice to set the prefix to the name of the region that the topic originated from, for example, `us-west-2`.

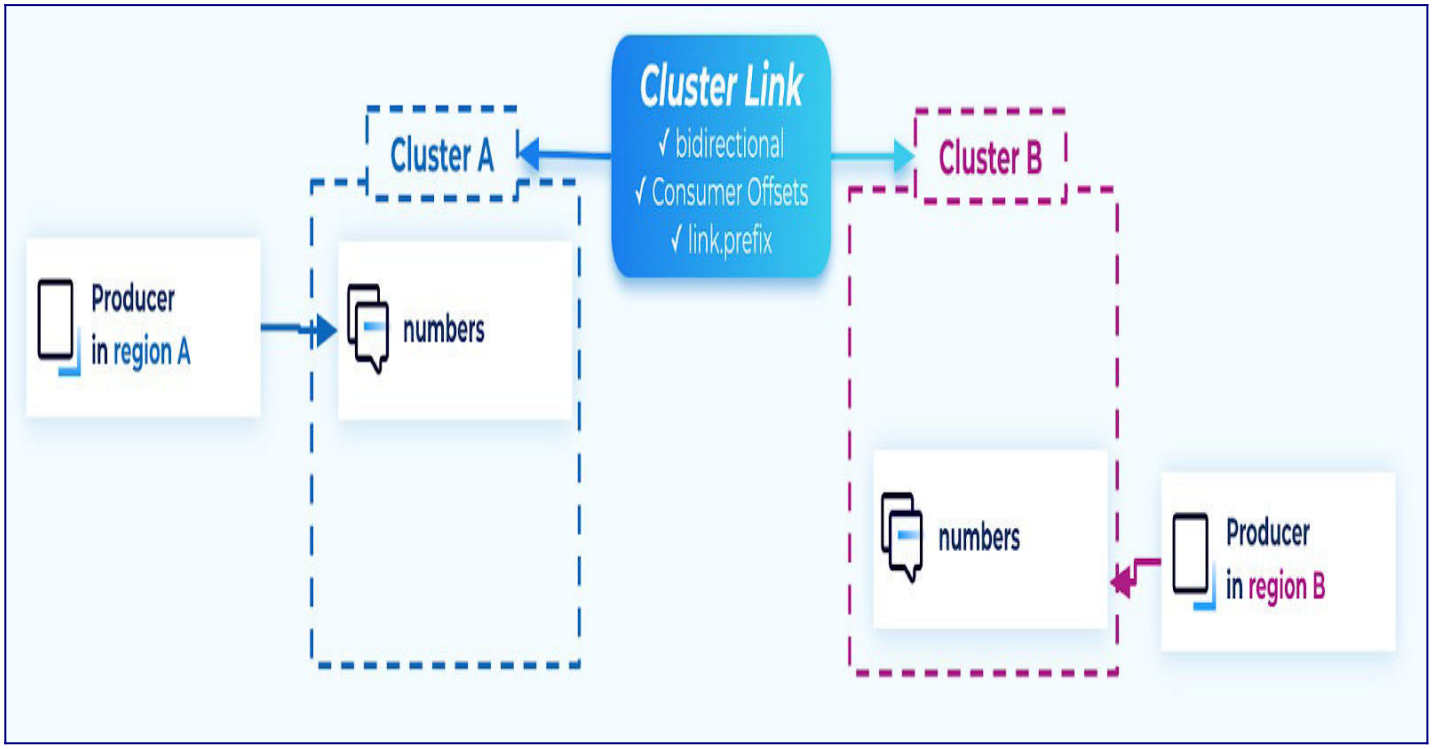
When creating a bidirectional cluster link, if you want the topics to have the same name on both clusters, you must set `link.prefix` on both clusters. Each cluster can have different values for

its `link.prefix`, or the clusters can share the same value of `link.prefix`.

Cluster links should enable consumer offset syncing, so consumer groups will have their offset present in all regions when they choose to failover or failback.

For a bidirectional cluster link, you must enable consumer offset syncing on both clusters when creating each cluster's cluster link object.

If ACL syncing is desired, a dedicated cluster link for that will be required, as explained in the security section.



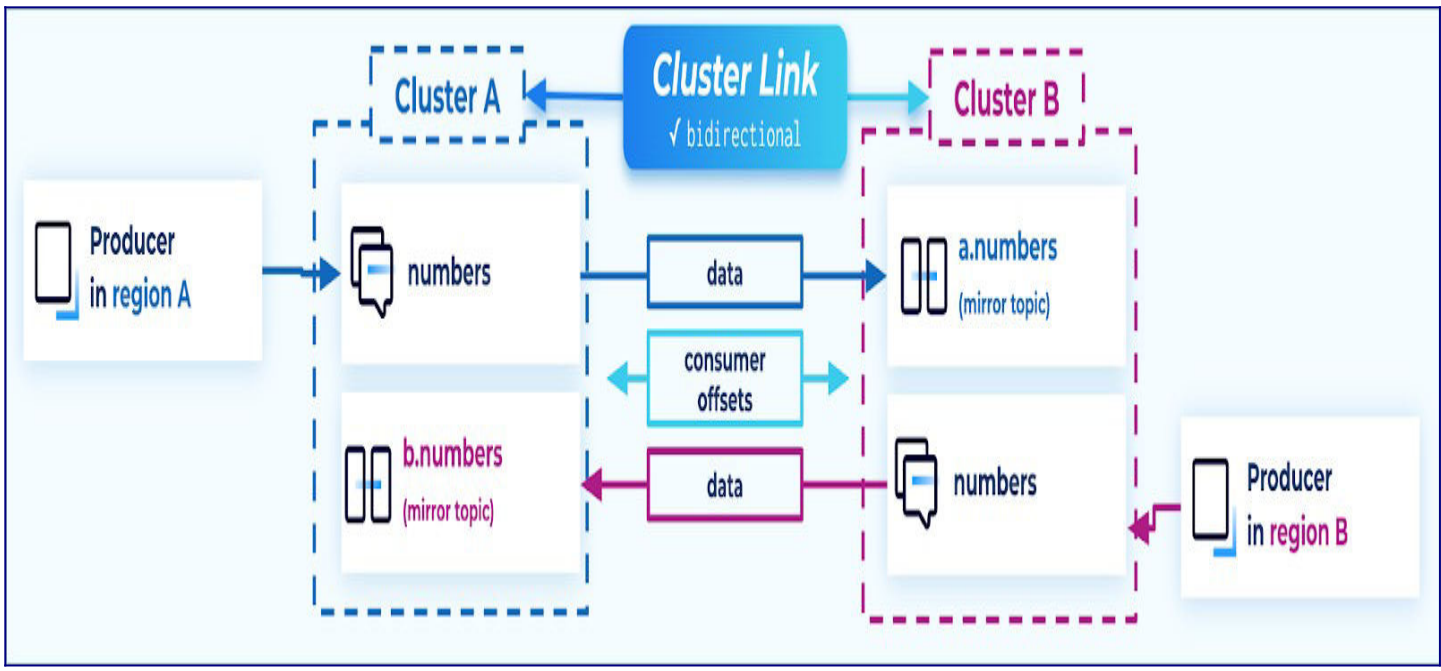
Cluster links

Create mirror topics¶

The cluster links should be used to create mirror topics for every writable topic on each cluster. This way, each cluster in the active/active architecture will have the full set of data.

The cluster link `link.prefix` setting has the effect of creating mirror topics with a prefix, and do not clash with the name of the writable topic.

For each topic, data only flows in one direction (source topic → mirror topic)



Mirror topics

Set up consumer groups¶

In order for a consumer group to receive the full set of data, it should consume across both topics. This can be done natively in Kafka in two ways:

- Specify a list of topic names to consume from.
- Specify a topic name pattern, which will use regex matching to find the topics to consume from.

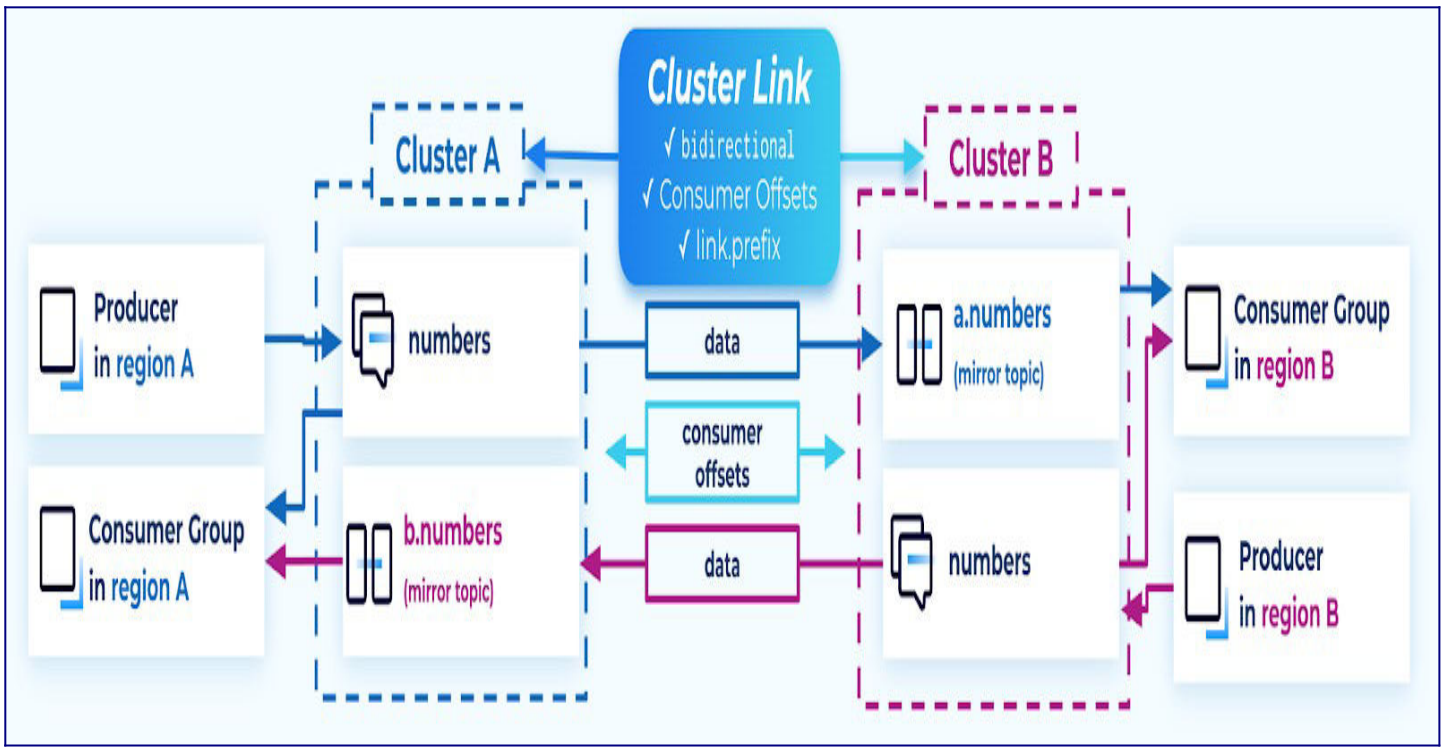
The advantage of using a regex pattern is that if the active/active architecture is extended to 3+ clusters, consumers will pick up the topics from new clusters automatically, without needing a configuration change.

Be careful when constructing the regex to not accidentally craft a regex that could include other, unrelated topics. For example, the regex `^([a-z]*\.)?numbers$` is safer than the regex `.*numbers` because the latter could also include topic names such as `east.private.numbers` or `east.numbers-processed`.

When consumer offset syncing is enabled on a bidirectional cluster link, the consumer offsets flow in both directions: from source topic → mirror topic and from mirror topic → source topic.

Consumer group names must be globally unique, so that their offsets can be synced to other clusters.

- In Kafka, each consumer group consumes from one cluster at a time: a given consumer group can connect to either Cluster A or Cluster B, but not both at the same time.
- If you need to move a consumer group from cluster A to cluster B, make sure it is shut down on cluster A before restarting it on cluster B.



Consumer groups

Important considerations for consumers when using keyed messages:

- If your producers produce messages with keys, and your consumers rely on Kafka to maintain ordering per key, an active/active architecture requires a slightly different strategy.
- Normally in Kafka, when only one topic is in play, all records with a given key will always be in the same partition. A single partition has static (fixed) ordering of messages. This ensures that all consumer groups consume the records for a given key in the same order: the order in which they're stored in that partition in Kafka.
- This is different when multiple topics are in play. Two messages with the same key could end up in different partitions: if message M1 with key K is produced to cluster A, and message M2 with key K is produced to cluster B, then M1 and M2 will be in different topics and thus different partitions.
- Since they are in different partitions, consumers cannot rely on the Kafka partition ordering to keep the messages in order. Some consumer groups may consume M1 and then M2, whereas others may consume M2 and then M1. Kafka does not guarantee ordering across multiple partitions.
- If a consumer group needs to rely on ordering of messages in its processing logic, it is best to use the message timestamp to determine the order in which messages were produced. The message timestamp is assigned by Kafka when the message is originally produced, and is preserved by Cluster Linking.
- For example, the consumer group* could keep a high watermark of the latest timestamp it has seen for each key. If a message comes in with a timestamp before the high watermark for its key, then that message can be either ignored or processed with special out-of-order handling.

In this strategy, ensure the high watermark is across the consumer group–not simply for the individual consumer instance–as partitions for the same key could be assigned to different consumer instances.

Set up security¶

If using RBAC rolebindings, the same role binding needs to be created in each cluster, and modified/deleted in each cluster. Role bindings are not synced between clusters by Cluster Linking.

If using ACLs, there are two possible approaches to sync ACLs between clusters with Cluster Linking:

If the cluster links in the active/active architecture do not use prefixing (not common), then the cluster link ACL sync feature can be enabled on them. When ACLs are created on a cluster for principal X to produce or consume from topics A and B, that ACL will be synced to the other cluster.

If using cluster link prefixing (more common in active/active), then ACL sync cannot be enabled on the cluster links that are used for data replication. (There is no safe, deterministic way to add a prefix to all ACLs, so the combination of these features is disabled.) However, if your ACL scheme fits, an extra cluster link can be created with the sole purpose of syncing ACLs (and no data).

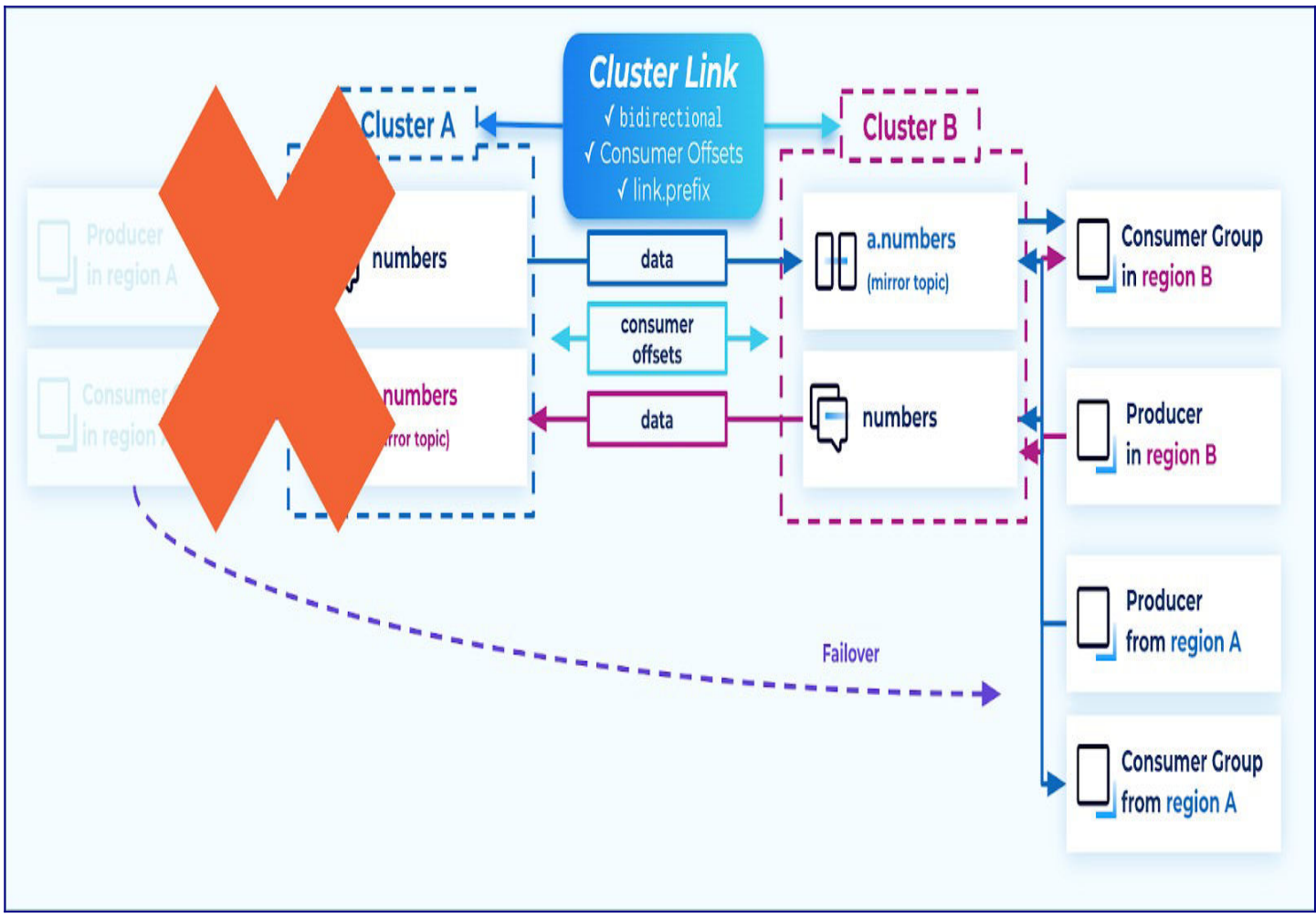
For example, an ACL that allows a principal to read or write from topics `numbers`, `a.numbers`, and `b.numbers` could be synced via this ACL-specific cluster link so that the principal has access to all topics across both clusters.

Failover and failback process¶

The advantage of the active/active pattern shines during failover and failback. Because any producer can produce to either side and any consumer can consume from either side, moving a Kafka client from one cluster to the other is as simple as stopping it on one cluster and starting it on the other. Unlike the active/passive pattern, no mirror topics need to be promoted, and no Cluster Linking APIs need to be called.

Failing over¶

When a disaster strikes, simply shut down any instances of producers and consumers that are still running in the failed region, and restart them in the DR region. In that way, the full topology is restored, from all producers to all consumer groups, and the business has averted an outage.



Failover

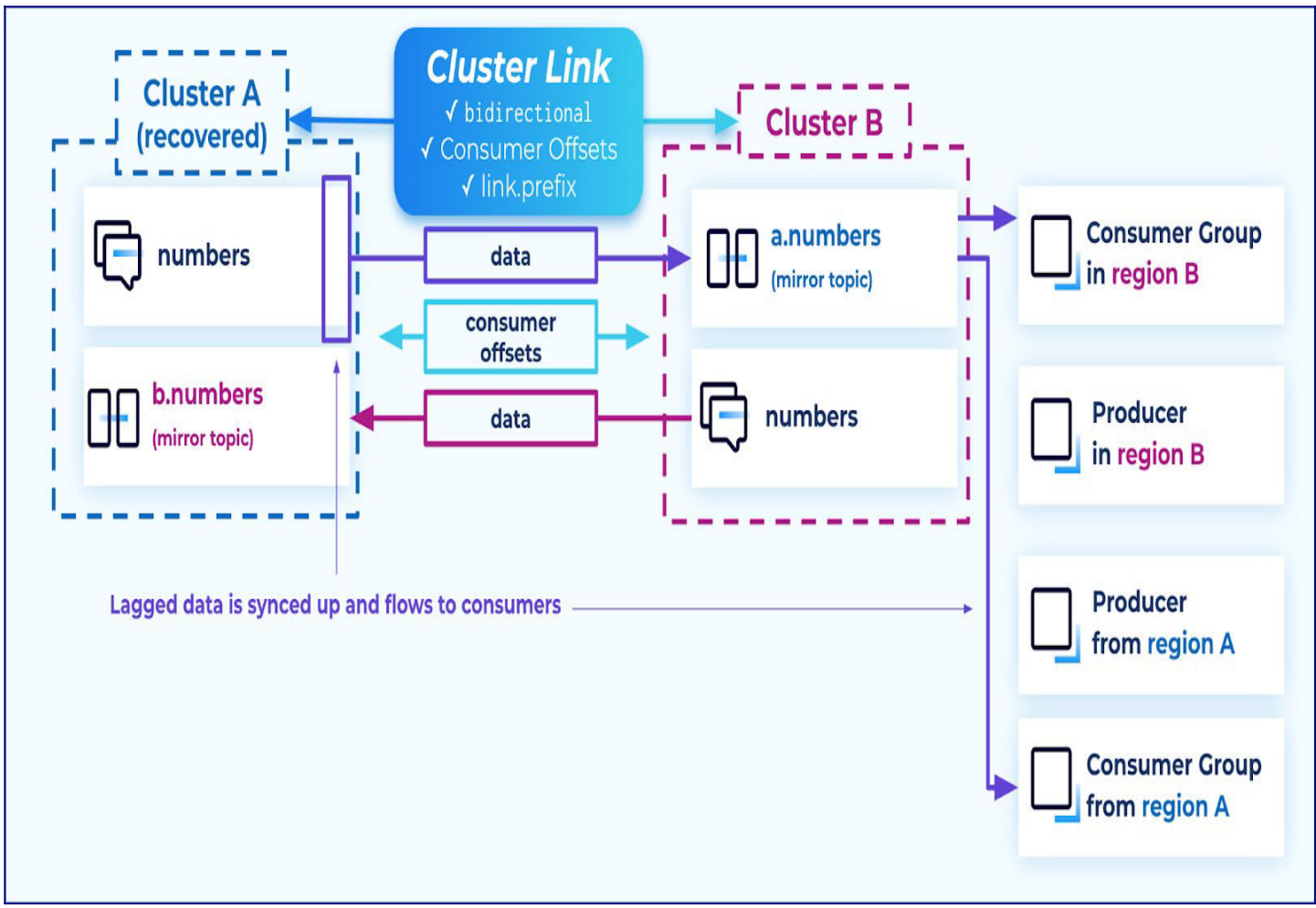
Do not call `failover` on the mirror topics, and do not delete the cluster links. Leaving the mirroring relationship intact allows for easy data recovery and failback.

The requirements for Kafka clients apply to the active/active pattern just as they apply to the active/passive pattern.

Data recovery

Depending on the failure, it's possible that some messages were only on the Confluent cluster in the region that had an outage, and had not yet been mirrored to the other region at the time of failover. For example, if the networking between regions A and B suddenly disappeared, some small number of records may only be on cluster A's *numbers* topic and not yet on cluster B's *a.numbers* topic. While this is uncommon in the cloud–since multi-zone Confluent clusters are spread across three datacenters with redundant networking–it is nonetheless a possibility that enterprises may prepare for.

When the outage is over, we expect the cloud region and its services, along with the Confluent cluster, to recover. We expect the Confluent cluster to still have those records available. At this point, the cluster link will resume functioning and will stream the messages to the other cluster (for example, from numbers to a.numbers). From there, the messages flow to the consumer groups.

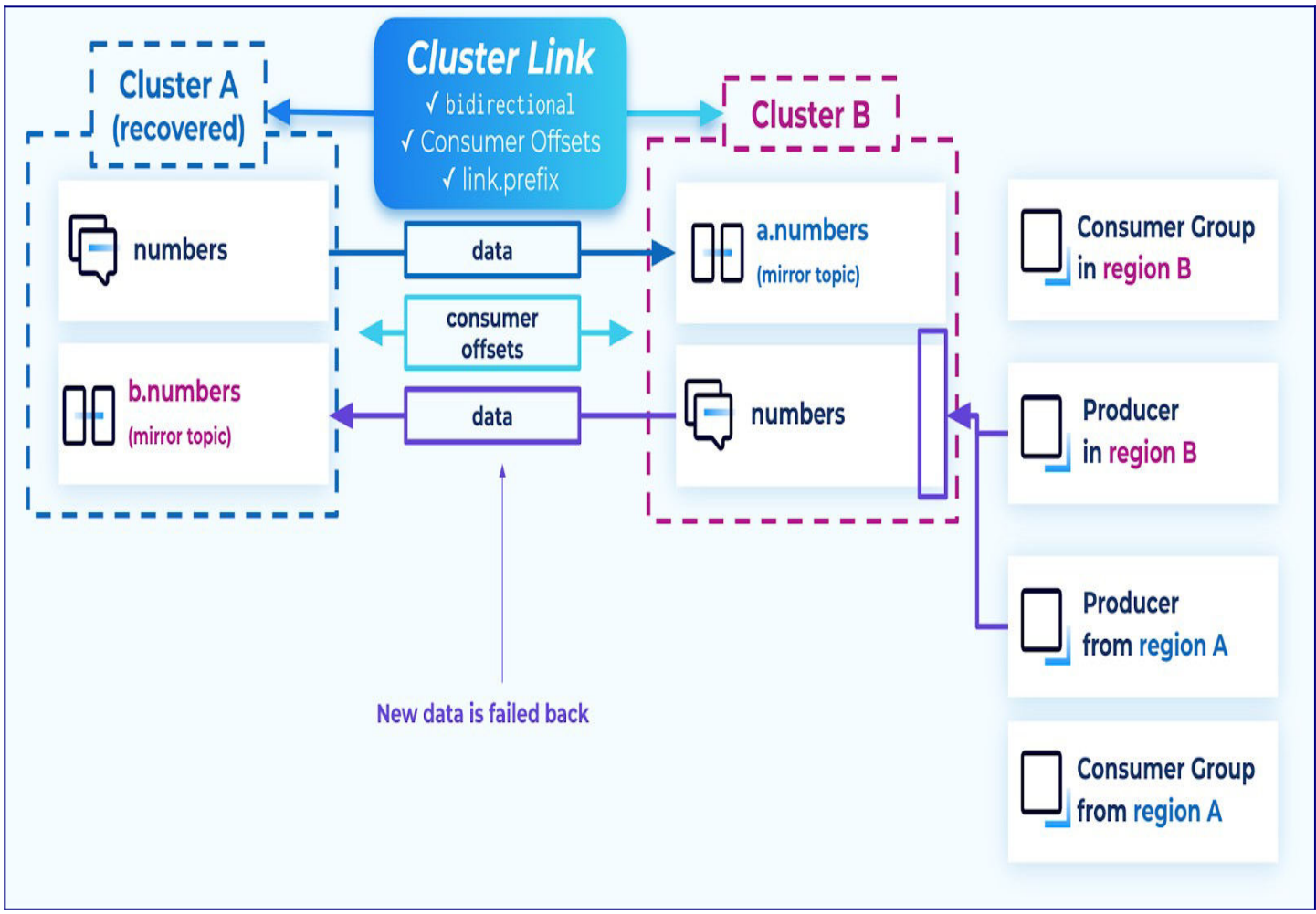


Recovery

If you want to control the timing of when this happens, you can pause the mirror topics on cluster B, and only resume them when you want the data to catch up.

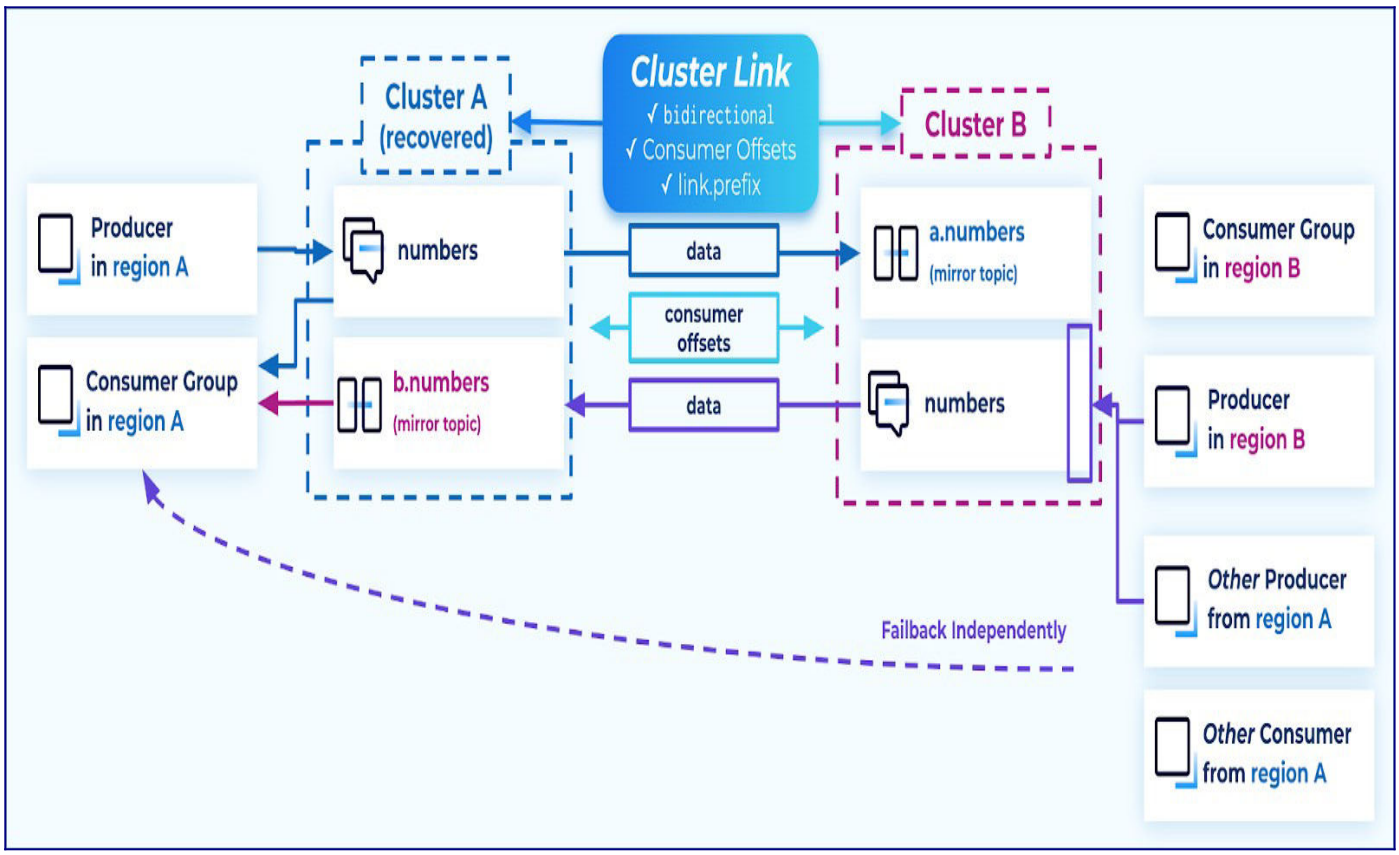
Failing back¶

Similar to the data recovery stage, failing back also takes advantage of the cluster links which were left intact during the outage. All data that had been produced to the surviving cluster during the outage is now synced to the recovered cluster, e.g. from Cluster B's *numbers* topic to Cluster A's *b.numbers* topic.



Failback

Just as with failover, Kafka clients can be failed back simply by stopping them in one region and restarting them connected to the other cluster. An advantage of the active/active pattern is that each producer can be moved at will–unlike the active/passive pattern, where the producers must be moved in a batch, topic by topic, since only one side is writable at a time.



Failback independently

If you need to ensure your consumer groups do not consume duplicate messages during the failback–which can happen if their consumer offsets are "clamped"–then check that:

- Either the mirror topic `lag=0` or the mirror topic `lag < consumer group lag`, and
- Enough time has passed for the consumer group's offsets to be synced. This is configured on the cluster link `consumer.offset.sync.ms` setting, which defaults to 30 seconds. At least one whole time interval needs to have passed since the consumer group last committed its offsets to be sure that the offsets have been synced.

Connectors and sharing data in active/active setups¶

Often, external systems need to consume data from a Confluent cluster, either through a sink connector or through a cluster link to another party's cluster. Sometimes, these data pipelines also need as high availability as possible.

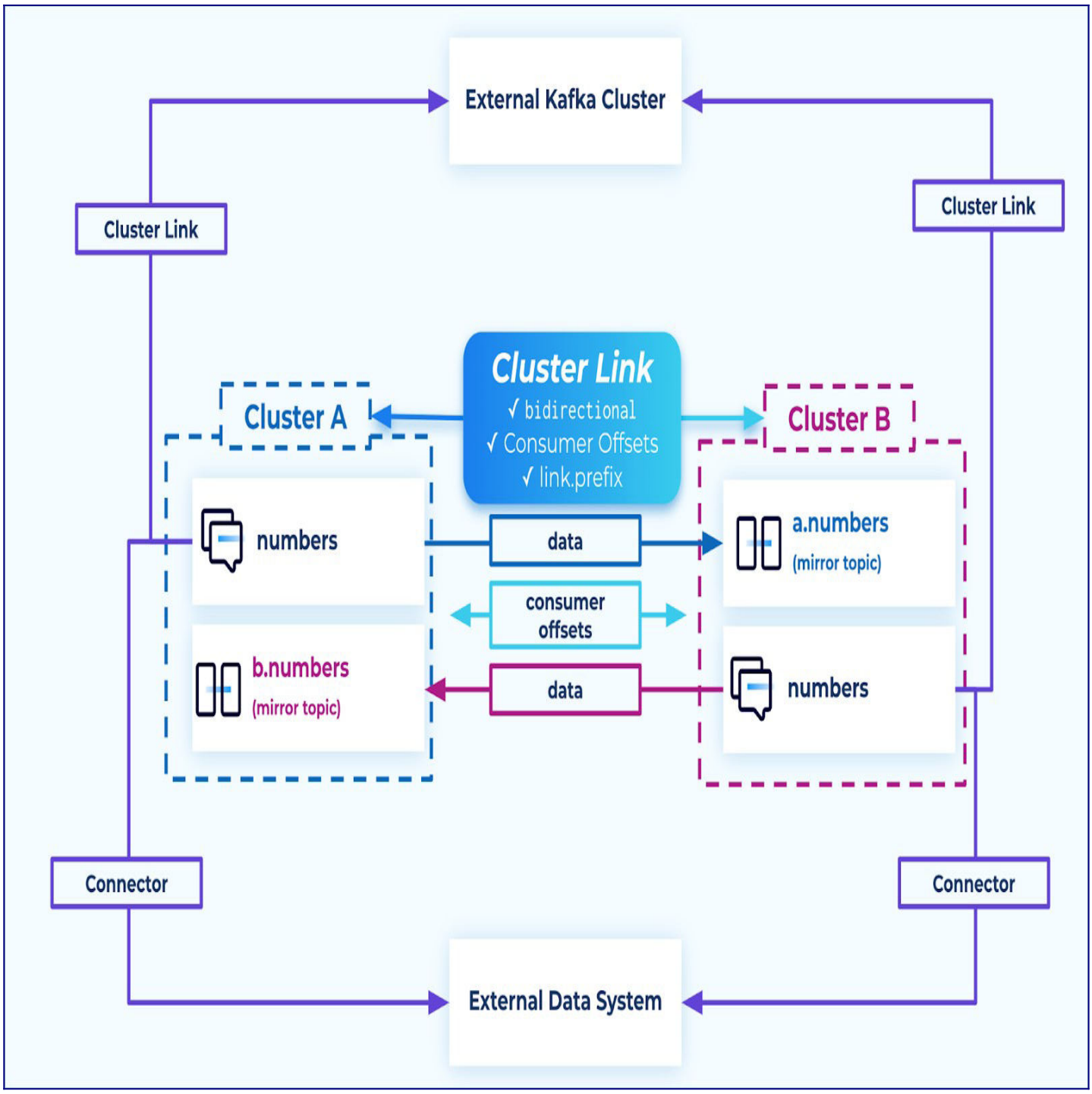
To achieve high availability with external systems in an active/active architecture, it is recommended to stream each of the writable topics to the external systems.

- Connectors: Create a sink connector on each cluster from each writable topic to the external system.
- Confluent clusters: Create a cluster link from each cluster to the external cluster, including only the writable topics from the source clusters.

Sharing the writable topics from each cluster ensures:

- All data across all regions is shared during steady state

- During an outage, the data system continues to receive new data without needing manual intervention, regardless of which region the outage happened in
- After an outage, any lagged data gets caught up automatically



Connectors and data sharing

Active/Passive Tutorial

To explore this use case, you will use Cluster Linking to fail over a topic and a simple command-line consumer and producer from an original cluster to a Disaster Recovery (DR) cluster.

Prerequisites

- **Got Confluent CLI? Make sure it's up-to-date.** If you already have the CLI installed, make sure you have the latest version with new Cluster Linking commands and tools. Run `confluent update` to update to the latest version. For details, see Get the latest version of the Confluent CLI in the quick start prerequisites.
- Make sure you have followed the steps under Commands and prerequisites in the overview. These steps tell you the easiest way to get an up-to-date version of Confluent Cloud if you don't already have it, and provide a quick overview of Cluster Linking commands.
- The DR cluster must be a Dedicated cluster.
- The Original cluster can be a Basic, Standard, or Dedicated. If you do not have these clusters already, you can create them in the Cloud Console or in the Confluent CLI.
- To learn more about supported cluster types and combinations, see Supported cluster types and supported cluster combinations for private networking.

Note

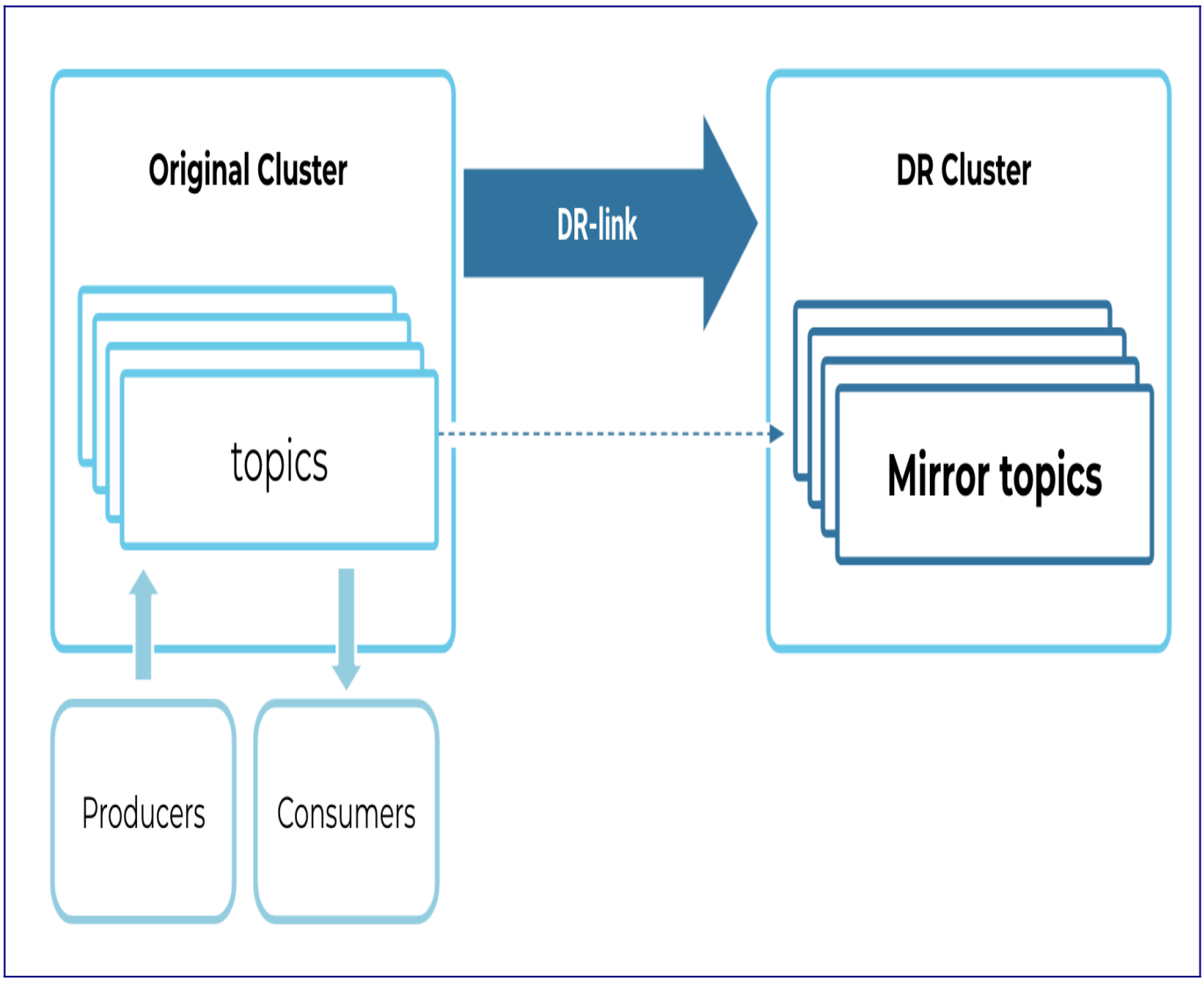
You can use failover between an eligible Confluent Platform cluster and an eligible Confluent Cloud cluster. You will need to use the Confluent CLI for the Confluent Cloud cluster, and for the Confluent Platform cluster. Installation instructions and an overview of CLI are here.

What the tutorial covers

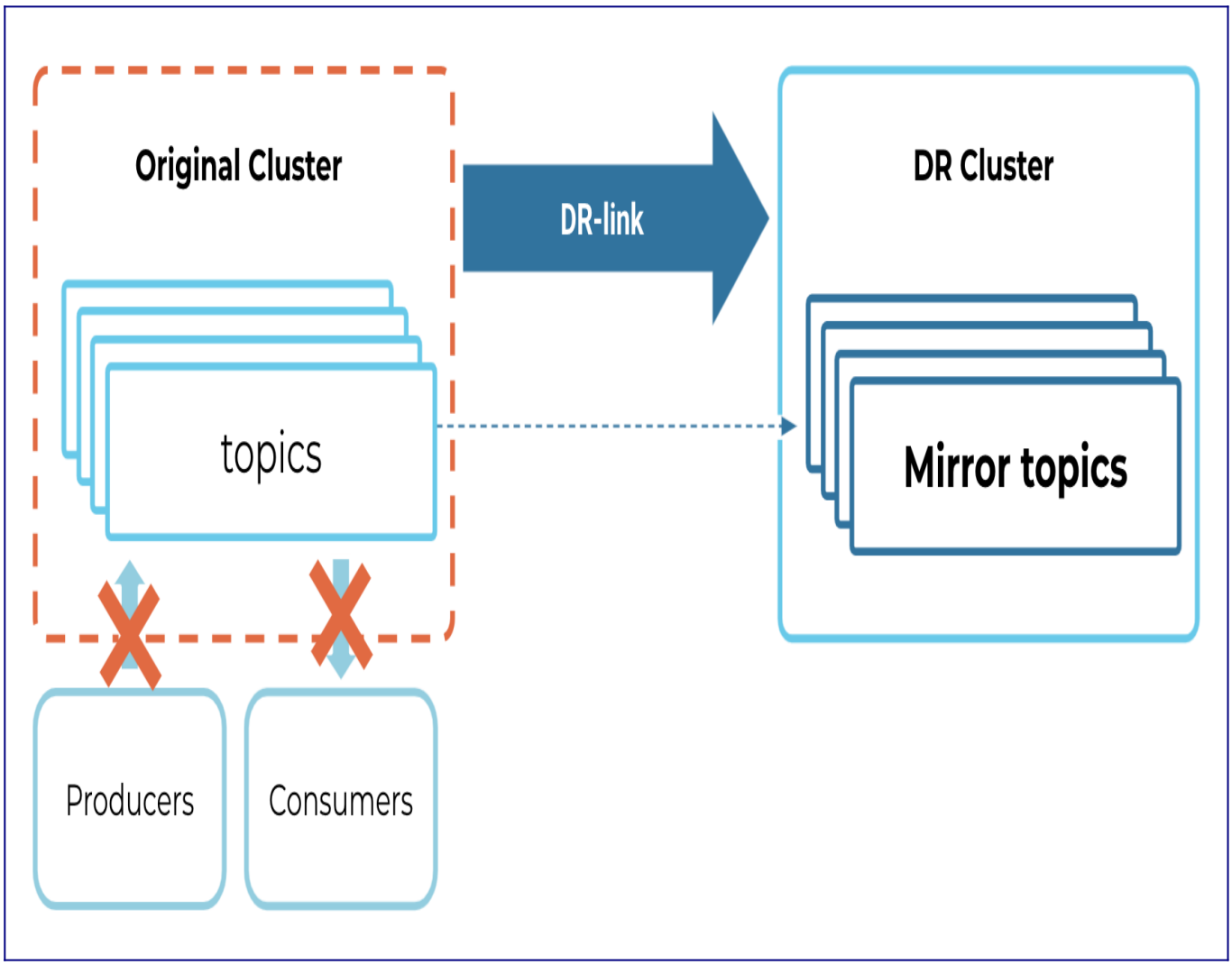
This tutorial demos use of the Confluent CLI Cluster Linking commands to create a DR cluster and failover to it.

A REST API for Cluster Linking commands may be available in future releases.

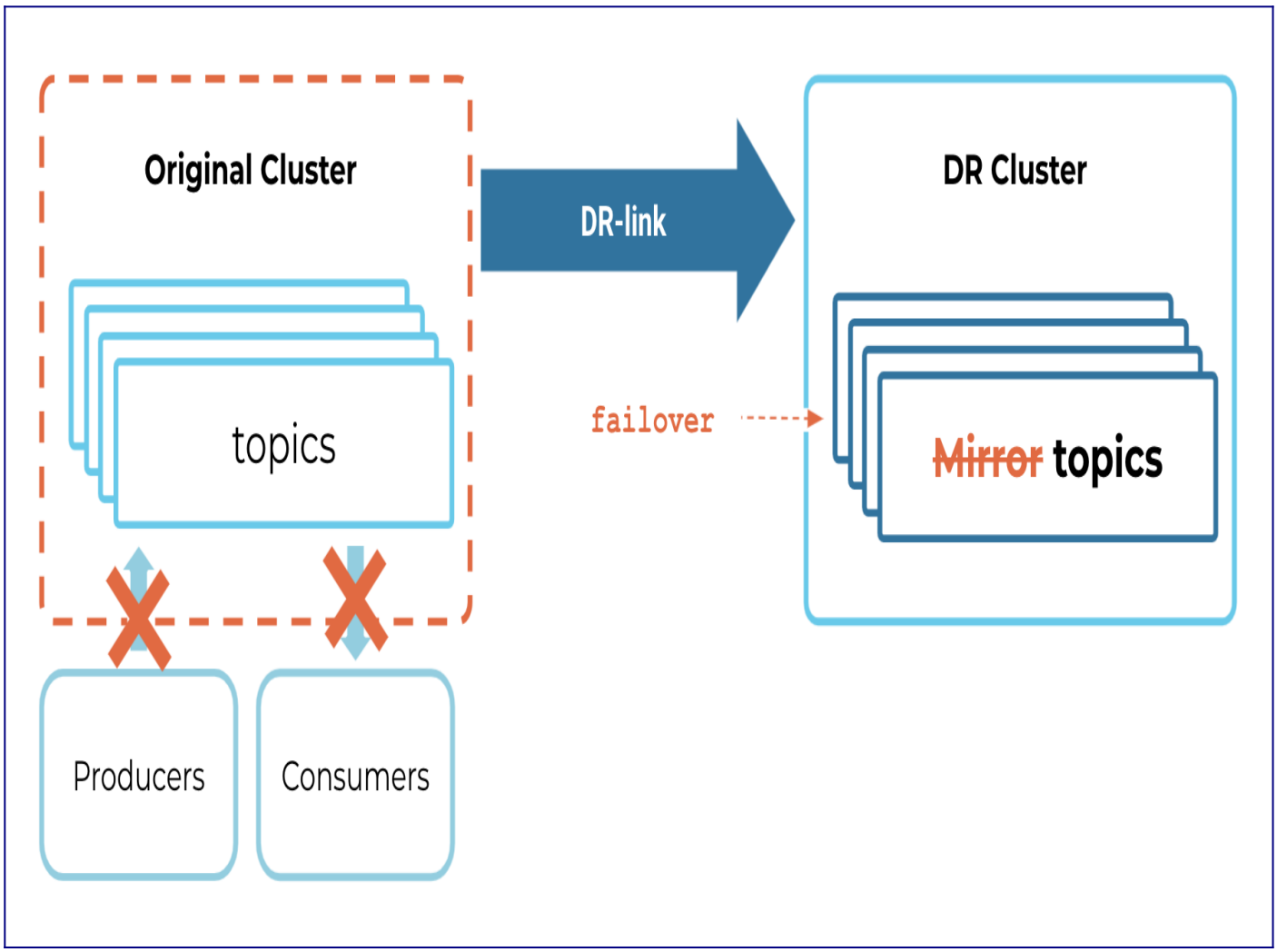
You will start by building a cluster link to the DR cluster (destination) and mirroring all pertinent topics, ACLs, and consumer group offsets. This is the "steady state" setup.



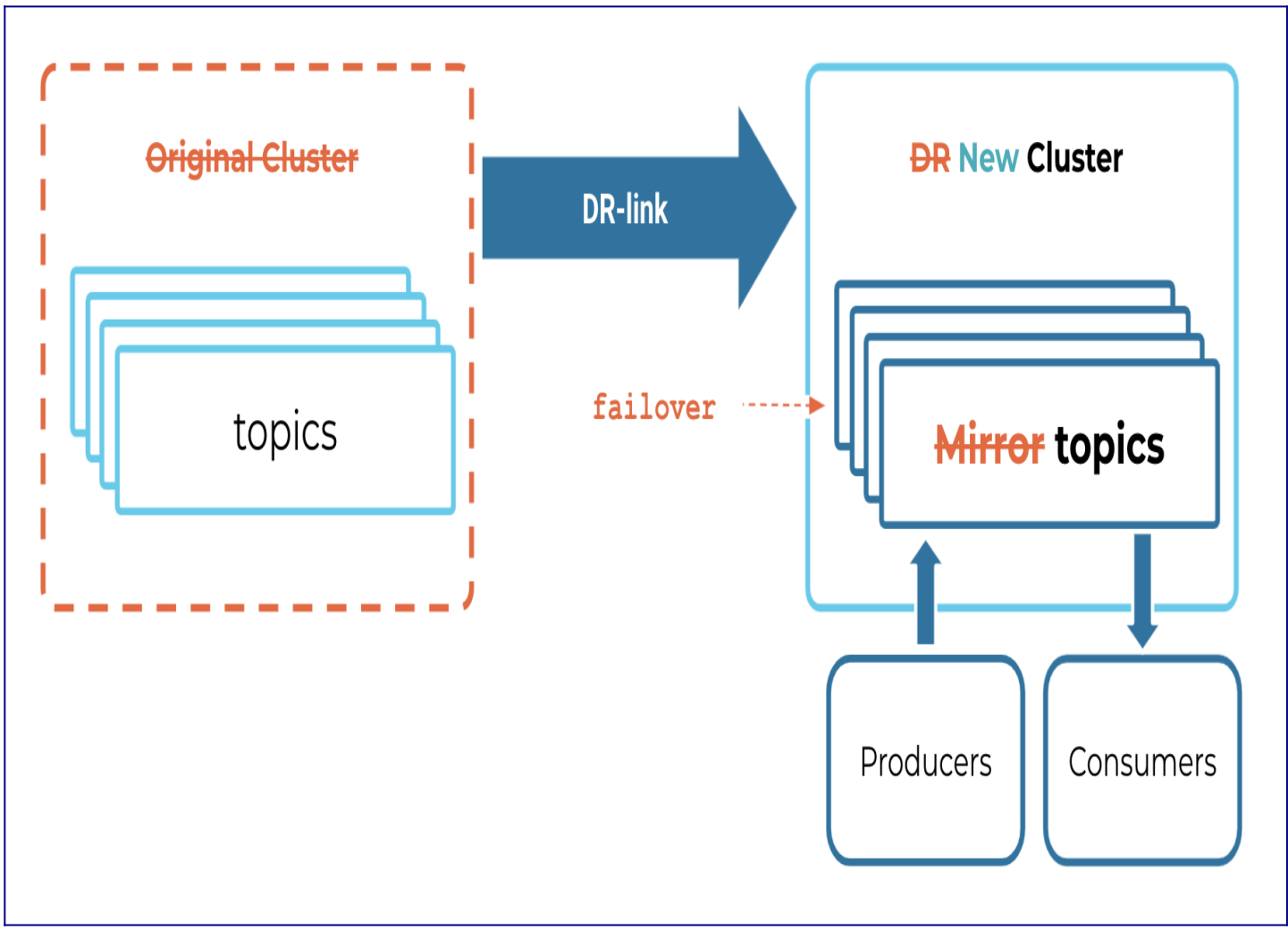
Then you will incur a sample outage on the original (source) cluster. When this happens, the producers, consumers, and cluster link will not be able to interact with the original cluster.



You will then call a failover command that converts the mirror topics on the DR cluster into regular topics.



Finally, you will move producers and consumers over to the DR cluster, and continue operations. The DR cluster has become our new source of truth.



Set up steady state

Create or choose the clusters you want to use

1. Log on to the Confluent Cloud web UI.
2. If you do not already have clusters created, create two clusters in the same environment, as described in Manage Kafka Clusters on Confluent Cloud.

   At least one of these must be a Dedicated cluster, which serves as the DR cluster (the "destination", which will host the mirror topics).

   The original or source cluster can be any type of cluster. To make these quickly distinguishable, you might want to make the source cluster a Basic or Standard cluster, and/or name these ORIGINAL and DR.

Navigate to the environment and clusters

Log on to the CLI (`confluent login`), and make sure you are in the environment you want to use for this tutorial.

To view environments and the clusters they contain, use these commands.

Command	Description
List environments	`confluent environment list`
Select an environment.	`confluent environment use <environment-id>`
List clusters.	`confluent kafka cluster list`
Select a cluster.	`confluent kafka cluster use <cluster-id>`

The tutorial will provide a few reminders about this navigation, but remember you can use this when you use commands without specifying a resource. For example, `confluent kafka link list` or `confluent kafka topic list` gives you the cluster links and topics, respectively, for the currently selected cluster. Otherwise, just specify the resource; for example, `confluent kafka link list --cluster <cluster-id>` or `confluent kafka topic list --cluster <cluster-id>`.

Also, keep in mind, as you walk through these steps, that you can view most cluster and topic information on the Cloud Console at https://confluent.cloud/login. You will do all the work on the CLI, but you may want to use the web UI as a supplemental view.

Keep notes of required information

Tip

To keep track of this information, you may find it easiest to simply save the output of the Confluent CLI commands to a text file and/or use shell environment variables. If you do so, be sure to safeguard API keys and secrets afterwards by deleting files, or moving only the security codes to safer storage. For details on how to do this, and other time-savers, see CLI tips in the overview.

For this walkthrough, you will need the following details accessible through Confluent CLI commands.

- The **cluster ID of your original cluster** (`<original-cluster-id>` for purposes of this tutorial). To get cluster IDs on Confluent Cloud, type the following at the Confluent CLI. In the examples below, the original cluster ID is `lkc-xkd1g`.

  ```
  confluent kafka cluster list
  ```

- The **bootstrap server for the original cluster**. To get this, type the following command, replacing `<original-cluster-id>` with the cluster ID for your source cluster.

  ```
  confluent kafka cluster describe <original-cluster-id>
  ```

  Your output should resemble the following. In the example output below, the bootstrap server is the value: `SASL_SSL://pkc-9kyp5.us-east-1.aws.confluent.cloud:9092`. This value is referred to as `<original-bootstrap-server>` in this tutorial.

  ```
  +--------------+-------------------------------------------------------+
  | Id           | lkc-xkd1g                                             |
  | Name         | AWS US                                                |
  | Type         | DEDICATED                                             |
  ```

```
| Ingress      |                                                          50 |
| Egress       |                                                         150 |
| Storage      | Infinite                                                    |
| Cloud        | aws                                                         |
| Availability | single-zone                                                 |
| Region       | us-east-1                                                   |
| Status       | UP                                                          |
| Endpoint     | SASL_SSL://pkc-9kyp5.us-east-1.aws.confluent.cloud:9092     |
| ApiEndpoint  | https://pkac-rrwjk.us-east-1.aws.confluent.cloud            |
| RestEndpoint | https://pkc-9kyp5.us-east-1.aws.confluent.cloud:443         |
| ClusterSize  |                                                           1 |
+--------------+-------------------------------------------------------+
```

- The **cluster ID of your DR cluster** (`<DR-cluster-id>`). You can get this in the same way as your original cluster ID, with this command `confluent kafka cluster list`.

  ```
  confluent kafka cluster list
  ```

- The **bootstrap server of your DR cluster** (`<DR-bootstrap-server>`). You can get this in the same way as your original cluster bootstrap server, by using the describe command, replacing `<DR-cluster-id>` with your destination cluster ID.

  ```
  confluent kafka cluster describe <DR-cluster-id>
  ```

Create a cluster link between the original and the DR cluster

Start by creating a cluster link that is mirroring topics, ACLs, and consumer group offsets from the source cluster to the destination cluster.

Set up privileges for the cluster link to access topics on the source cluster

Your cluster link needs privileges to read the appropriate topics on your source cluster. To give it these privileges, you create two mechanisms:

- A **service account** for the cluster link. Service accounts are used in Confluent Cloud to group together applications and entities that need access to your Confluent Cloud resources.
- An **API key and secret** that is associated with the cluster link's service account and the source cluster. The link will use this API key to authenticate with the source cluster when it is fetching topic information and messages. A service account can have many API keys, but for this tutorial, you need only one.

To create these resources, do the following:

1. Create a service account for this cluster link.

   ```
   confluent iam service-account create Cluster-Linking-Demo --description
   "For the cluster link created for the DR failover tutorial"
   ```

   Your output should resemble the following.

   ```
   +-------------+-----------+
   | Id          |    254122 |
   | Resource ID | sa-lqxn16 |
   | Name        | ...       |
   | Description | ...       |
   +-------------+-----------+
   ```

Save the ID field (`<service-account-id>` for the purposes of this tutorial).

2. Create the API key and secret.

   ```
   confluent api-key create --resource <original-cluster-id> --service-
   account <service-account-id>
   ```

   Note

   Store this key and secret somewhere safe. When you create the cluster link, you must supply it with this API key and secret, which will be stored on the cluster link itself.

3. Allow the cluster link to read topics on the source cluster. Give the cluster link's service account the ACLs to be able to READ and DESCRIBE-CONFIGS for all topics.

   ```
   confluent kafka acl create --allow --service-account <service-account-id>
   --operations read,describe-configs --topic "*" --cluster <original-
   cluster-id>
   ```

   Tip

   The example above allows read access to all topics by using the asterisk (`--topic "*"`) instead of specifying particular topics. If you wish, you can narrow this down to a specific set of topics to mirror. For example, to allow the cluster link to read all topics that begin with a "clicks" prefix, you can do this:

   ```
   confluent kafka acl create --allow --service-account <service-account-id>
   --operations read,describe-configs --topic clicks --prefix --cluster
   <original-cluster-id>
   ```

4. Provide the capability to sync ACLs from the source to the destination cluster.

   This allows consumers, producers, and other services to continue running, even in the event of a failure.

   To do this, you must give the cluster link's service account an ACL to DESCRIBE the source cluster:

   ```
   confluent kafka acl create --allow --service-account <service-account-id>
   --operations describe --cluster-scope --cluster <original-cluster-id>
   ```

5. Provide the capability to sync consumer group offsets for mirror topics over this cluster link, so that consumers can pick up at the offset at which they left off.

   There are two sets of ACLs your clusters need for this.

   - Give the cluster link's service account the appropriate ACLs to DESCRIBE topics, and to READ and DESCRIBE consumer groups on the source (original) cluster.

     ```
     confluent kafka acl create --allow --service-account <service-
     account-id> --operations describe --topic "*" --cluster <original-
     cluster-id>

     confluent kafka acl create --allow --service-account <service-
     account-id> --operations read,describe --consumer-group "*" --
     cluster <original-cluster-id>
     ```

- Give the cluster link's service account ACLs to READ and ALTER topics on the destination (DR) cluster, and ACLs to READ its consumer groups.

```
confluent kafka acl create --allow --service-account <service-account-id> --operations read,alter --topic "*" --cluster <DR-cluster-id>

confluent kafka acl create --allow --service-account <service-account-id> --operations read --consumer-group "*" --cluster <DR-cluster-id>
```

Create the cluster link

1. Create a configuration file that turns on ACL sync, consumer group offset sync, and includes the security credentials for the link.

   To do this, copy the following lines into a new file called `dr-link.config`, and then replace `<api-key>` and `<api-secret>` with the key and secret you just created.

```
consumer.offset.sync.enable=true
consumer.offset.group.filters={"groupFilters": [{"name":
"*","patternType": "LITERAL","filterType": "INCLUDE"}]}
consumer.offset.sync.ms=1000

acl.sync.enable=true
acl.sync.ms=1000
acl.filters={ "aclFilters": [ { "resourceFilter": { "resourceType": "any",
"patternType": "any" }, "accessFilter": { "operation": "any",
"permissionType": "any" } } ] }

topic.config.sync.ms=1000

security.protocol=SASL_SSL
sasl.mechanism=PLAIN
sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule
required username="<api-key>" password="<api-secret>";
```

   A few notes on this configuration, which does the following:

   - Syncs offsets for all consumer groups (`*`) for these mirror topics. You can filter these down by passing in either exact topic names (`"patternType": "LITERAL"`) or prefixes (`"patternType": "PREFIX"`) to either include (`"filterType": "INCLUDE"`) or exclude (`"filterType": "EXCLUDE"`).
   - Syncs all ACLs that are on the Original cluster (`*`) so that consumers, producers, and other services can access the topics on the DR cluster. Unlike the consumer group offset sync, this is not limited to mirror topics, and may sync ACLs for other topics. You can filter these down by passing in either exact topic names (`"patternType": "LITERAL"`) or prefixes (`"patternType": "PREFIX"`) to either include (`"filterType": "INCLUDE"`) or exclude (`"filterType": "EXCLUDE"`).
   - Syncs consumer offsets, ACLs, and topic configurations every 1000 millseconds. This gives the link more up-to-date values for these. It comes at the expense of higher data throughput over the link, and potentially lower maximum throughput for the topic data. We recommend trying different values to see what works best for your specific cluster link.

- The last line in the file starts with `sasl.jaas.config` and ends with a semicolon (`;`). This must be all on one line, as shown.

2. Create the cluster link as shown below, with the command `confluent kafka link create <flags>`.

   In this example, the cluster link is called `dr-link`.

   ```
   confluent kafka link create dr-link \
     --cluster <dr-cluster-id>  \
     --source-cluster <original-cluster-id> \
     --source-bootstrap-server <original-bootstrap-server>  \
     --config dr-link.config
   ```

   If this is successful, you should get this message: `Created cluster link "dr-link".`

   Also, you can verify that the link was created by listing existing links:

   ```
   confluent kafka link list --cluster <DR-cluster-id>
   ```

Configure the source topic on the original cluster and mirror topic on the DR cluster

1. Create a topic called `dr-topic` on the original cluster.

   For the sake of this demo, create this topic with only one partition (`--partitions 1`). Having only one partition makes it easier to notice how the consumer offsets are synced from original cluster to DR cluster.

   ```
   confluent kafka topic create dr-topic --partitions 1 --cluster <original-cluster-id>
   ```

   You should see the message `Created topic “dr-topic”`, as shown in the example below.

   ```
   > confluent kafka topic create dr-topic --partitions 1 --cluster lkc-xkd1g
   Created topic "dr-topic".
   ```

   You can verify this by listing topics.

   ```
   confluent kafka topic list --cluster <original-cluster-id>
   ```

2. Create a mirror topic of `dr-topic` on the DR cluster.

   ```
   confluent kafka mirror create dr-topic --link <link-name> --cluster <DR-cluster-id>
   ```

   You should see the message `Created mirror topic “dr-topic”`, as shown in the example below.

   ```
   > confluent kafka mirror create dr-topic --link dr-link --cluster lkc-r68yp
   Created mirror topic "dr-topic".
   ```

   Tip

   In the current preview release, you must create each mirror topic one-by-one with a CLI or API command. Future releases may provide a service to Cluster Linking that can

automatically create mirror topics when new topics are created on your source cluster. In this case, you could filter the topics based on a prefix. That feature would be useful for automatically creating DR topics on a DR cluster.

At this point, you have a topic on your original cluster that is mirroring all of its data, ACLs, and consumer group offsets to a mirror topic on your DR cluster.

Produce and consume some data on the original cluster

In this section, you will simulate an application that is producing data to and consuming data from your original cluster. You will use the Confluent CLI to consume and produce functions to do so.

1. Create a service account to represent your CLI based demo clients.

   ```
   confluent iam service-account create CLI --description "From CLI"
   ```

   Your output should resemble:

   ```
   +-------------+-----------+
   | Id          |    254262 |
   | Resource ID | sa-ldr3w1 |
   | Name        | CLI       |
   | Description | From CLI  |
   +-------------+-----------+
   ```

   In the above example, the `<cli-service-account-id>` is 254262.

2. Create an API key and secret for this service account on your original cluster, and save these as `<original-CLI-api-key>` and `<original-CLI-api-secret>`.

   ```
   confluent api-key create --resource <original-cluster-id> --service-
   account <cli-service-account-id>
   ```

3. Create an API key and secret for this service account on your DR cluster, and save these as `<DR-CLI-api-key>` and `<DR-CLI-api-secret>`.

   ```
   confluent api-key create --resource <DR-cluster-id> --service-account
   <cli-service-account-id>
   ```

4. Give your CLI service account enough ACLs to produce and consume messages on your Original cluster.

   ```
   confluent kafka acl create --service-account <cli-service-account-id> --
   allow --operations read,describe,write --topic "*" --cluster <original-
   cluster-id>

   confluent kafka acl create --service-account <cli-service-account-id> --
   allow --operations describe,read --consumer-group "*" --cluster <original-
   cluster-id>
   ```

Now you can produce and consume some data on the original cluster.

1. Tell your CLI to use your original API key on your original cluster.

   ```
   confluent api-key use <original-cluster-api-key> --resource <original-
   cluster-id>
   ```

2. Produce the numbers 1-5 to your topic

```
seq 1 5 | confluent kafka topic produce dr-topic --cluster <original-cluster-id>
```

You should see this output, but the command should complete without needing you to press ^C or ^D.

```
Starting Kafka Producer. ^C or ^D to exit
```

Tip

If you get an error message indicating `unable to connect to Kafka cluster`, wait for a minute or two, then try again. For recently created Kafka clusters and API keys, it may take a few minutes before the resources are ready.

3. Start a CLI consumer to read from the `dr-topic` topic, and give it the name `cli-consumer`.

   As a part of this command, pass in the flag `--from-beginning` to tell the consumer to start from offset `0`.

   ```
   confluent kafka topic consume dr-topic --group cli-consumer --from-beginning
   ```

   After the consumer reads all 5 messages, press Ctrl + C to quit the consumer.

4. In order to observe how consumers pick up from the correct offset on a failover, artificially force some consumer lag on your consumer.

   Produce numbers 6-10 to your topic.

   ```
   seq 6 10 | confluent kafka topic produce dr-topic
   ```

   You should see the following output, but the command should complete without needing you to press ^C or ^D.

   ```
   Starting Kafka Producer. ^C or ^D to exit
   ```

Now, you have produced 10 messages to your topic on your original cluster, but your cli-consumer has only consumed 5.

Monitoring mirroring lag

Because Cluster Linking is an asynchronous process, there may be mirroring lag between the source cluster and the destination cluster.

You can see what your mirroring lag is on a per-partition basis for your DR topic with this command:

```
confluent kafka mirror describe dr-topic --link dr-link --cluster <dr-cluster-id>
```

```
  LinkName  | MirrorTopicName | Partition | PartitionMirrorLag | SourceTopicName
| MirrorStatus | StatusTimeMs
+-----------+-----------------+-----------+--------------------
+----------------+-------------+--------------+
   dr-link |  dr-topic        |         0 |                  0 |  dr-topic
| ACTIVE       | 1624030963587
```

```
   dr-link |  dr-topic        |         1 |                 0 |  dr-topic
| ACTIVE       | 1624030963587
   dr-link |  dr-topic        |         2 |                 0 |  dr-topic
| ACTIVE       | 1624030963587
   dr-link |  dr-topic        |         3 |                 0 |  dr-topic
| ACTIVE       | 1624030963587
   dr-link |  dr-topic        |         4 |                 0 |  dr-topic
| ACTIVE       | 1624030963587
   dr-link |  dr-topic        |         5 |                 0 |  dr-topic
| ACTIVE       | 1624030963587
```

You can also monitor your lag and your mirroring metrics through the Confluent Cloud Metrics. These two metrics are exposed:

- MaxLag shows the maximum lag (in number of messages) among the partitions that are being mirrored. It is available on a per-topic and a per-link basis. This gives you a sense of how much data will be on the Original cluster only at the point of failover.
- Mirroring Throughput shows on a per-link or per-topic basis how much data is being mirrored.

List cluster links and mirror topics

At various points in a workflow, it may be useful to get lists of cluster linking resources such as links or mirror topics. You might want to do this for the purposes of monitoring, or before starting a failover or migration.

To list the cluster links on the active cluster:

```
confluent kafka link list
```

You can get a list mirror topics on a link, or on a cluster.

To list the mirror topics on a specified cluster link:

```
confluent kafka mirror list --link <link-name> --cluster <cluster-id>
```

To list all mirror topics on a particular cluster:

```
confluent kafka mirror list --cluster <cluster-id>
```

Simulate a failover to the DR cluster

In a disaster event, your original cluster is usually unreachable.

In this section, you will go through the steps you follow on the DR cluster in order to resume operations.

1. Perform a dry run of a failover to preview the results without actually executing the command. To do this, simply add the `--dry-run` flag to the end of the command.

   ```
   confluent kafka mirror failover <mirror-topic-name> --link <link-name> --
   cluster <DR-cluster-id> --dry-run
   ```

   For example:

```
confluent kafka mirror failover dr-topic --link dr-link --cluster <DR-cluster-id> --dry-run
```

2. Stop the mirror topic to convert it to a normal, writeable topic.

   ```
   confluent kafka mirror failover <mirror-topic-name> --link <link-name> --cluster <DR-cluster-id>
   ```

   For this example, the mirror topic name and link name will be as follows.

   ```
   confluent kafka mirror failover dr-topic --link dr-link --cluster <DR-cluster-id>
   ```

   Expected output:

   ```
   MirrorTopicName | Partition | PartitionMirrorLag | ErrorMessage | ErrorCode
   ------------------------------------------------------------------------
   dr-topic        |         0 |                  0 |              |
   ```

   The stop command is irreversible. Once you change your mirror topic to a regular topic, you cannot change it back to a mirror topic. If you want it to be a mirror topic once again, you will need to delete it and recreate it as a mirror topic.

3. Now you can produce and consume data on the DR cluster.

   Set your CLI to use the DR cluster's API key:

   ```
   confluent api-key use <DR-CLI-api-key> --resource <DR-cluster-id>
   ```

4. Produce numbers 11-15 on the topic, to show that it is a writeable topic.

   ```
   seq 11 15 | confluent kafka topic produce dr-topic --cluster <DR-cluster-id>
   ```

   You should see this output, but the command should complete without needing you to press ^C or ^D.

   ```
   Starting Kafka Producer. ^C or ^D to exit
   ```

5. "Move" your consumer group to the DR cluster, and consume from `dr-topic` on the DR cluster.

   ```
   confluent kafka topic consume dr-topic --group cli-consumer --cluster <DR-cluster-id>
   ```

   You should expect the consumer to start consuming at number 6, since that's where it left off on the original cluster. If it does, that will show that its consumer offset was correctly synced. It should consume through number 15, which is the last message you produced on the DR cluster.

   After you see number 15, hit Ctrl + C to quit your consumer.

   Expected output:

   ```
   Starting Kafka Consumer. ^C or ^D to exit
   6
   7
   8
   ```

```
9
10
11
12
13
14
15
^CStopping Consumer.
```

You have now failed over your CLI producer and consumer to the DR cluster, where they continued operations smoothly.

Recovery After a Disaster

If your original cluster experienced a temporary failure, like a cloud provider regional outage, then it may come back.

Recover lagged data

Because Cluster Linking is asynchronous, you may have had data on the original cluster that did not make it over to the DR cluster at the time of failure.

If the outage was temporary and your original Confluent Cloud cluster comes back online, there may be some data on it that was never replicated to the DR cluster. You can recover this data if you wish.

1. Find the offsets at which the mirror topics were failed over.

   These offsets are persisted on your DR cluster, as long as your cluster link object exists.

   Note

   If you delete your cluster link on your DR cluster, you will lose these offsets.

   For any topic that you called failover or promote on, you can get the offsets at which the failover occurred with either of the following methods:

   - Use this CLI command, and view the `Last Source Fetch Offset` column in the command output.

     ```
     confluent kafka mirror describe <topic-name>
     ```

   - Use the Confluent Cloud REST API calls to describe mirror topics with either `/links/[link-name]/mirrors` or `/links/[link-name]/mirrors/<topic-name>`, and look at the `mirror_lags` array and the values under `last_source_fetch_offset`.

2. With those offsets in hand, point consumers at your original cluster, and reset their offsets to those you found in. Your consumers can then recover the data and act on it as appropriate for their application.

   Tip

   You can also append these messages to the end of the topics on your DR cluster by using Confluent Replicator.

Move operations back to the original cluster

After failing over your cluster, Kafka clients, and applications to a DR region, the most common strategy is to *fail-forward* (also known as *fail-and-stay*). That is, you keep operations running on the DR region indefinitely, converting it to be your new "primary" region, and build up a new DR region. This is because cloud regions are usually interchangeable for businesses running in the cloud that do not have a physical datacenter tying them to a specific geography.

If you have failed over all applications from the original region, you have little to gain by failing back to that region, which would take effort and introduce risk.

On the other hand, some scenarios may require that you move operations back to your original region, perhaps to minimize latency and cost when interacting with datacenters and nodes in that region. If you want to move operations back to your original cluster, here is the sequence of steps to do so.

1. Recover lagged data.
2. Recover data that was never replicated.
3. Pause the cluster link that was going to the DR cluster.
4. Identify the topics that you wish to move from the DR cluster to the original cluster. If any of these topics still exist on the original cluster, you will need to delete them.
5. Migrate from the DR cluster to your Original cluster. Follow the instructions for data migration.
6. After you have moved all of your topics, consumers, and producers, restore the DR relationship. To do so, delete the topics on the DR cluster, unpause your cluster link, and recreate the appropriate mirror topics (or let the cluster link auto-create them).

Recovery recommendations for ksqlDB and Kafka Streams

In the event of failover, disaster recovery (DR) of ksqlDB and Kafka Streams is not covered by the Cluster Linking DR workflow described above. The following recovery recommendations are provided for these datasets. These recommendations apply to both self-managed Confluent Enterprise and Confluent Cloud ksqlDB and Kafka Streams.

General recommendations

- A best practice is to run a second ksqlDB application in the DR cluster against the mirrored input topic. This will promote a much faster failover, and will ensure that the ksqlDB application in the DR cluster has the same state as the original cluster.
- Only the input topics should be replicated to the DR cluster. The internal topics created by Kafka Streams should not be mirrored. The reasoning behind this recommendation is as follows:
    - Changelogs and output topics may be out of sync with each other since they are replicated asynchronously (race conditions). For windowed processing, some temporary inconsistency may be acceptable, but for other use cases it presents a major problem.
    - Upstream changelogs may lag behind downstream, resulting in an unexpected and altered application state.
- For the DR failover scenario, use one of these strategies:

    - **Recommended:** Have a second application running in the DR site reading from the mirrored input topic.
    - Re-run the application against the DR cluster after failover, and reprocess to rebuild state. (This will result in a slower recovery process because rebuilding state can take time.)

KTables

- You can use Tiered storage to make sure that data is always retained. If you do this, the mirror topic will have all history, and the KTable can be built from history. This should be set up from the very beginning so the input topic retention does not remove needed historical data.
- Another option is to configure compaction on the topic in the original cluster from the very beginning on. You can use the `compaction-lag` to preserve the full X-day history and then compact anything older. In other words, the compaction will only run in the part of the data that is older than the configured lag of X-days.
- Existing applications with a KTable built from an input topic that has already purged data due to retention period can either:
    - **Recommended:** Rebuild from the mirror topic and lose the history in the KTable
    - (Not advised) Mirror the underlying compacted topic (supporting the KTable) to the DR site. If you configure your applications this way, the risk is that the KTable will not accurately represent the current state due to the asynchronous nature of the replication (as noted above), and lagged data will need to be reprocessed (either reproduced, or processed when the original cluster has returned to service). This strategy is not advised but can be tested if absolutely necessary. There is no guarantee of accurate data representation in the DR cluster with this method.

Related content

- Webinar focused on Disaster Recovery: Demo Series: Learn the Confluent Q3 ‘21 Release
- This tutorial covered a specific use case for disaster recovery. Share Data Across Clusters, Regions, and Clouds using Confluent Cloud provides a tutorial on data sharing across topics which may be in the same or different clusters, regions, and clouds. This is another basic use case for Cluster Linking.
- Mirror Topics provides a concept overview of this feature of Cluster Linking.