Strimzi Overview

Table of Contents

Chapter 1. Key features

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

This guide is intended as a starting point for building an understanding of Strimzi. The guide introduces some of the key concepts behind Kafka, which is central to Strimzi, explaining briefly the purpose of Kafka components. Configuration points are outlined, including options to secure and monitor Kafka. A distribution of Strimzi provides the files to deploy and manage a Kafka cluster, as well as example files for configuration and monitoring of your deployment.

A typical Kafka deployment is described, as well as the tools used to deploy and manage Kafka.

1.1. Kafka capabilities

The underlying data stream-processing capabilities and component architecture of Kafka can deliver:

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

1.2. Kafka use cases

Kafka's capabilities make it suitable for:

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

1.3. How Strimzi supports Kafka

Strimzi provides container images and Operators for running Kafka on Kubernetes. Strimzi

Operators are fundamental to the running of Strimzi. The Operators provided with Strimzi are purpose-built with specialist operational knowledge to effectively manage Kafka.

Operators simplify the process of:

- Deploying and running Kafka clusters
- Deploying and running Kafka components
- Configuring access to Kafka
- Securing access to Kafka
- Upgrading Kafka
- Managing brokers
- Creating and managing topics
- Creating and managing users

Chapter 2. Strimzi deployment of Kafka

Apache Kafka components are provided for deployment to Kubernetes with the Strimzi distribution. The Kafka components are generally run as clusters for availability.

A typical deployment incorporating Kafka components might include:

- **Kafka** cluster of broker nodes
- **ZooKeeper** cluster of replicated ZooKeeper instances
- **Kafka Connect** cluster for external data connections
- **Kafka MirrorMaker** cluster to mirror the Kafka cluster in a secondary cluster
- **Kafka Exporter** to extract additional Kafka metrics data for monitoring
- **Kafka Bridge** to make HTTP-based requests to the Kafka cluster

Not all of these components are mandatory, though you need Kafka and ZooKeeper as a minimum. Some components can be deployed without Kafka, such as MirrorMaker or Kafka Connect.

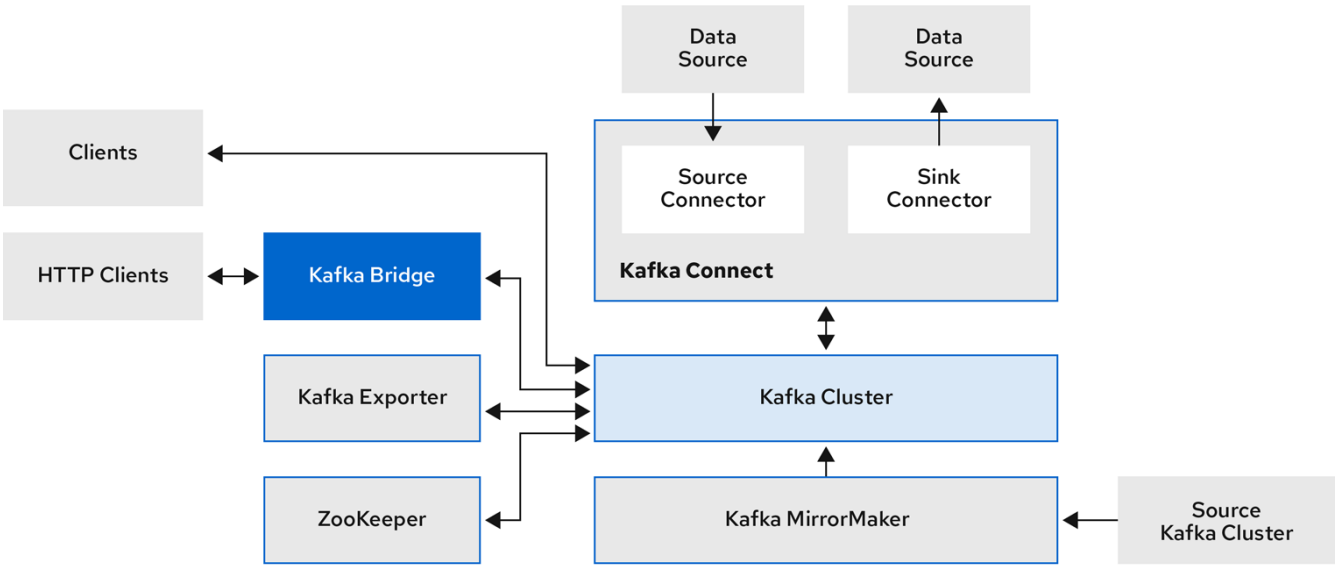
2.1. Kafka component architecture

A cluster of Kafka brokers handles delivery of messages.

A broker uses Apache ZooKeeper for storing configuration data and for cluster coordination. Before running Apache Kafka, an Apache ZooKeeper cluster has to be ready.

Each of the other Kafka components interact with the Kafka cluster to perform specific roles.

Kafka component interaction



Apache ZooKeeper

Apache ZooKeeper is a core dependency for Kafka as it provides a cluster coordination service,

storing and tracking the status of brokers and consumers. ZooKeeper is also used for controller election.

Kafka Connect

Kafka Connect is an integration toolkit for streaming data between Kafka brokers and other systems using *Connector* plugins. Kafka Connect provides a framework for integrating Kafka with an external data source or target, such as a database, for import or export of data using connectors. Connectors are plugins that provide the connection configuration needed.

- A *source* connector pushes external data into Kafka.
- A *sink* connector extracts data out of Kafka

  External data is translated and transformed into the appropriate format.

  You can deploy Kafka Connect with `build` configuration that automatically builds a container image with the connector plugins you require for your data connections.

Kafka MirrorMaker

Kafka MirrorMaker replicates data between two Kafka clusters, within or across data centers.

MirrorMaker takes messages from a source Kafka cluster and writes them to a target Kafka cluster.

Kafka Bridge

Kafka Bridge provides an API for integrating HTTP-based clients with a Kafka cluster.

Kafka Exporter

Kafka Exporter extracts data for analysis as Prometheus metrics, primarily data relating to offsets, consumer groups, consumer lag and topics. Consumer lag is the delay between the last message written to a partition and the message currently being picked up from that partition by a consumer

Chapter 3. About Kafka

Apache Kafka is an open-source distributed publish-subscribe messaging system for fault-tolerant real-time data feeds.

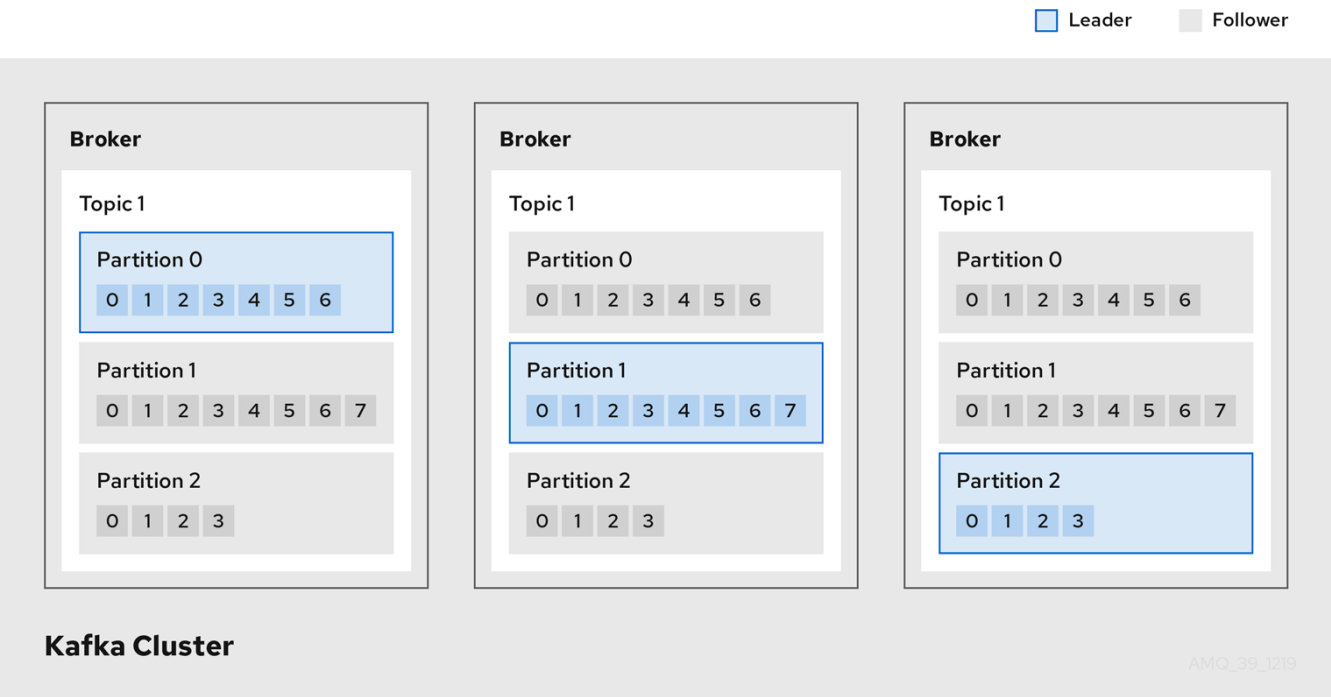
For more information about Apache Kafka, see the Apache Kafka documentation.

3.1. How Kafka operates as a message broker

To maximise your experience of using Strimzi, you need to understand how Kafka operates as a message broker.

A Kafka cluster comprises multiple brokers. Brokers contain topics that receive and store data. Topics are split by partitions, where the data is written. Partitions are replicated across topics for fault tolerance.

Kafka brokers and topics



Broker

A broker, sometimes referred to as a server or node, orchestrates the storage and passing of messages.

Topic

A topic provides a destination for the storage of data. Each topic is split into one or more partitions.

Cluster

A group of broker instances.

Partition

The number of topic partitions is defined by a topic *partition count*.

Partition leader

A partition leader handles all producer requests for a topic.

Partition follower

A partition follower replicates the partition data of a partition leader, optionally handling consumer requests.

Topics use a *replication factor* to configure the number of replicas of each partition within the cluster. A topic comprises at least one partition.

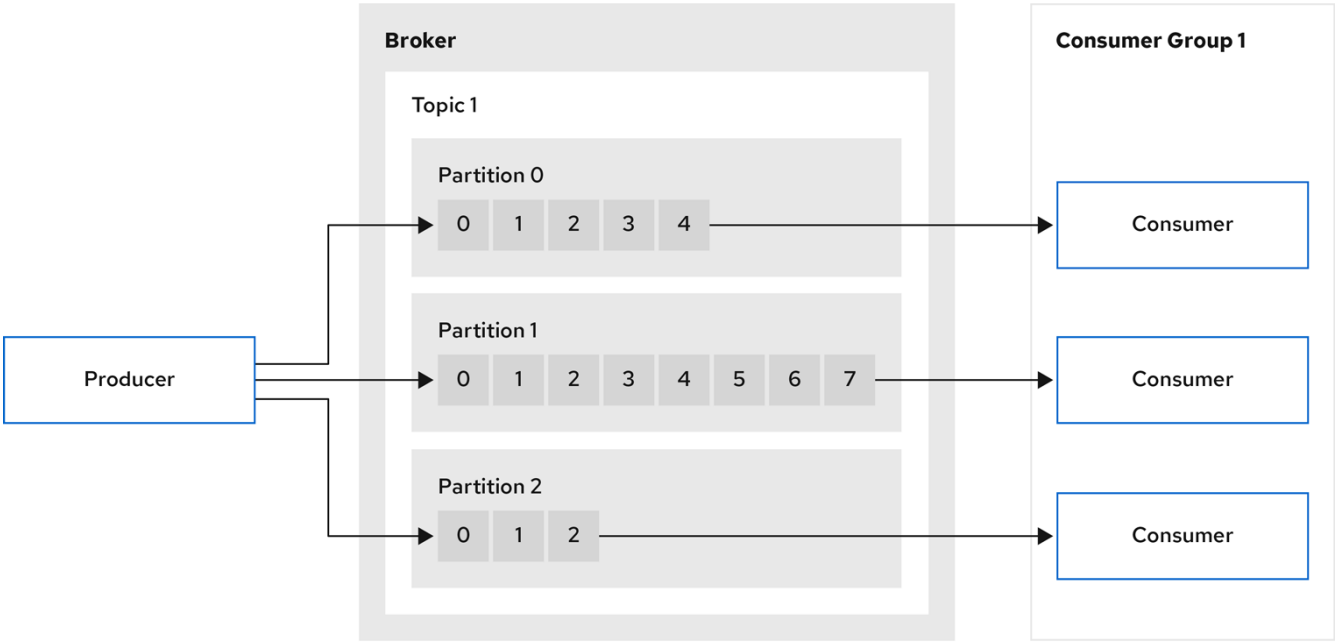
An *in-sync* replica has the same number of messages as the leader. Configuration defines how many replicas must be *in-sync* to be able to produce messages, ensuring that a message is committed only after it has been successfully copied to the replica partition. In this way, if the leader fails the message is not lost.

In the *Kafka brokers and topics* diagram, we can see each numbered partition has a leader and two followers in replicated topics.

3.2. Producers and consumers

Producers and consumers send and receive messages (publish and subscribe) through brokers. Messages comprise an optional *key* and a *value* that contains the message data, plus headers and related metadata. The key is used to identify the subject of the message, or a property of the message. Messages are delivered in batches, and batches and records contain headers and metadata that provide details that are useful for filtering and routing by clients, such as the timestamp and offset position for the record.

Producers and consumers



Producer

A producer sends messages to a broker topic to be written to the end offset of a partition. Messages are written to partitions by a producer on a round robin basis, or to a specific partition based on the message key.

Consumer

A consumer subscribes to a topic and reads messages according to topic, partition and offset.

Consumer group

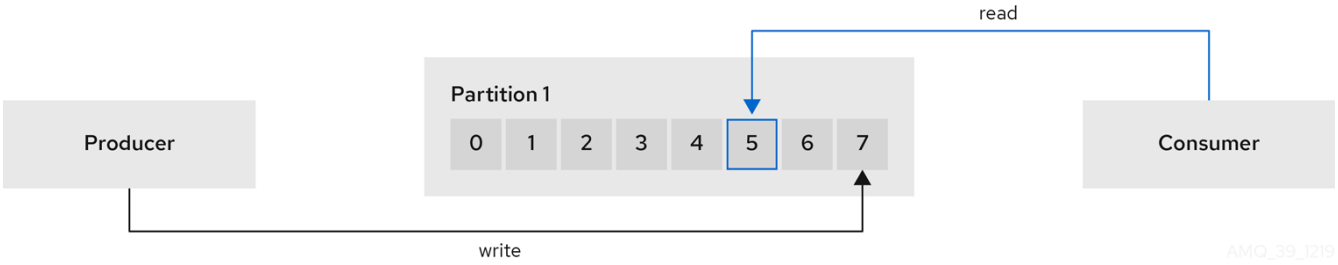
Consumer groups are used to share a typically large data stream generated by multiple producers from a given topic. Consumers are grouped using a `group.id`, allowing messages to be spread across the members. Consumers within a group do not read data from the same partition, but can receive data from one or more partitions.

Offsets

Offsets describe the position of messages within a partition. Each message in a given partition has a unique offset, which helps identify the position of a consumer within the partition to track the number of records that have been consumed.

Committed offsets are written to an offset commit log. A `__consumer_offsets` topic stores information on committed offsets, the position of last and next offset, according to consumer group.

Producing and consuming data


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

Chapter 4. About Kafka Connect

Kafka Connect is an integration toolkit for streaming data between Kafka brokers and other systems. The other system is typically an external data source or target, such as a database.

Kafka Connect uses a plugin architecture to provide the implementation artifacts for connectors. Plugins allow connections to other systems and provide additional configuration to manipulate data. Plugins include connectors and other components, such as data converters and transforms. A connector operates with a specific type of external system. Each connector defines a schema for its configuration. You supply the configuration to Kafka Connect to create a connector instance within Kafka Connect. Connector instances then define a set of tasks for moving data between systems.

Strimzi operates Kafka Connect in *distributed mode*, distributing data streaming tasks across one or more worker pods. A Kafka Connect cluster comprises a group of worker pods. Each connector is instantiated on a single worker. Each connector comprises one or more tasks that are distributed across the group of workers. Distribution across workers permits highly scalable pipelines.

Workers convert data from one format into another format that's suitable for the source or target system. Depending on the configuration of the connector instance, workers might also apply transforms (also known as Single Message Transforms, or SMTs). Transforms adjust messages, such as filtering certain data, before they are converted. Kafka Connect has some built-in transforms, but other transformations can be provided by plugins if necessary.

4.1. How Kafka Connect streams data

Kafka Connect uses connector instances to integrate with other systems to stream data.

Kafka Connect loads existing connector instances on start up and distributes data streaming tasks and connector configuration across worker pods. Workers run the tasks for the connector instances. Each worker runs as a separate pod to make the Kafka Connect cluster more fault tolerant. If there are more tasks than workers, workers are assigned multiple tasks. If a worker fails, its tasks are automatically assigned to active workers in the Kafka Connect cluster.

The main Kafka Connect components used in streaming data are as follows:

- Connectors to create tasks
- Tasks to move data
- Workers to run tasks
- Transforms to manipulate data
- Converters to convert data

4.1.1. Connectors

Connectors can be one of the following type:

- Source connectors that push data into Kafka
- Sink connectors that extract data out of Kafka

Plugins provide the implementation for Kafka Connect to run connector instances. Connector instances create the tasks required to transfer data in and out of Kafka. The Kafka Connect runtime orchestrates the tasks to split the work required between the worker pods.

MirrorMaker 2 also uses the Kafka Connect framework. In this case, the external data system is another Kafka cluster. Specialized connectors for MirrorMaker 2 manage data replication between source and target Kafka clusters.

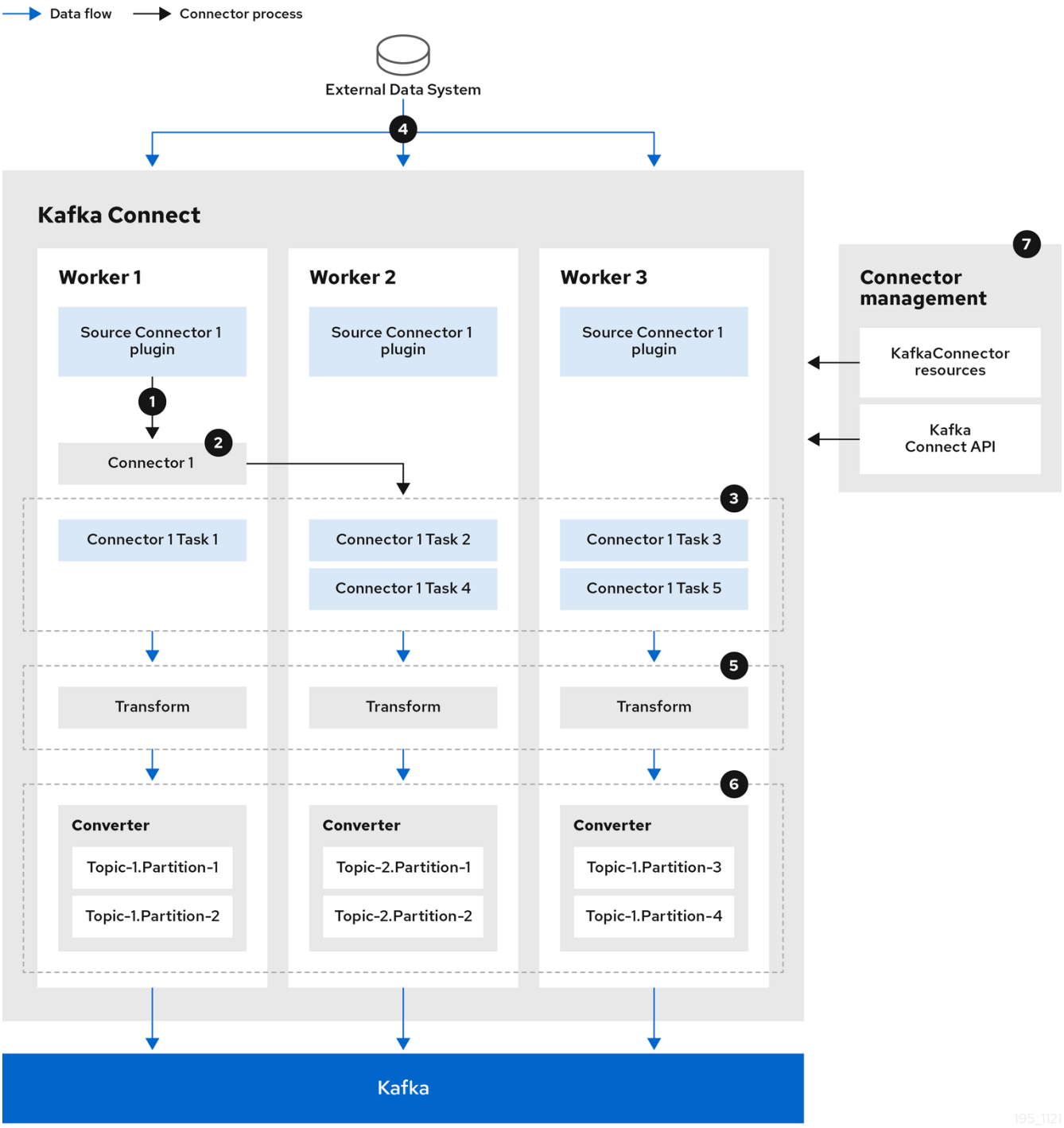
NOTE

In addition to the MirrorMaker 2 connectors, Kafka provides two connectors as examples:

- `FileStreamSourceConnector` streams data from a file on the worker's filesystem to Kafka, reading the input file and sending each line to a given Kafka topic.
- `FileStreamSinkConnector` streams data from Kafka to the worker's filesystem, reading messages from a Kafka topic and writing a line for each in an output file.

The following source connector diagram shows the process flow for a source connector that streams records from an external data system. A Kafka Connect cluster might operate source and sink connectors at the same time. Workers are running in distributed mode in the cluster. Workers can run one or more tasks for more than one connector instance.

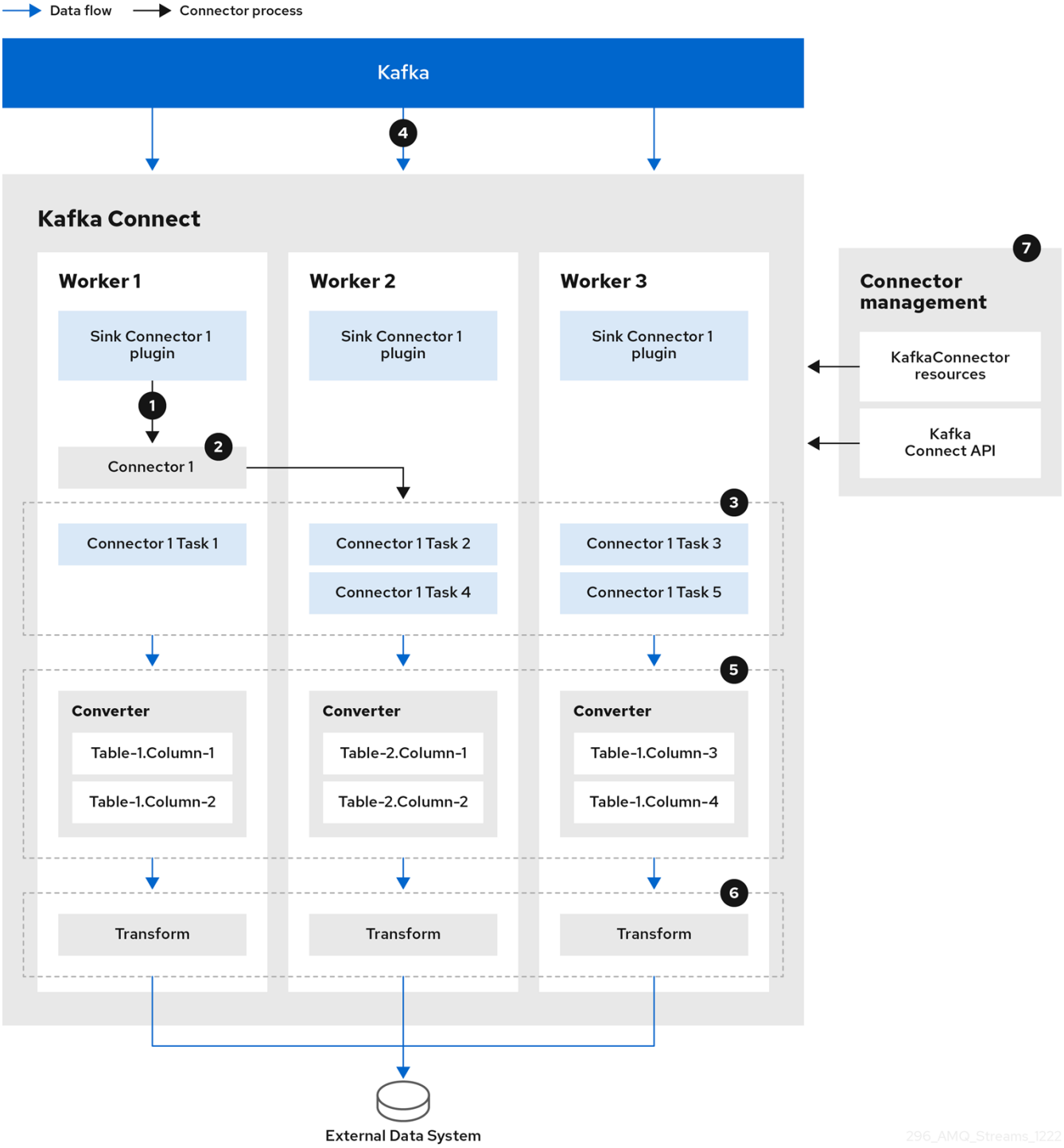
Source connector streaming data to Kafka



1. A plugin provides the implementation artifacts for the source connector
2. A single worker initiates the source connector instance
3. The source connector creates the tasks to stream data
4. Tasks run in parallel to poll the external data system and return records
5. Transforms adjust the records, such as filtering or relabelling them
6. Converters put the records into a format suitable for Kafka
7. The source connector is managed using KafkaConnectors or the Kafka Connect API

The following sink connector diagram shows the process flow when streaming data from Kafka to an external data system.

Sink connector streaming data from Kafka



1. A plugin provides the implementation artifacts for the sink connector
2. A single worker initiates the sink connector instance
3. The sink connector creates the tasks to stream data
4. Tasks run in parallel to poll Kafka and return records
5. Converters put the records into a format suitable for the external data system
6. Transforms adjust the records, such as filtering or relabelling them
7. The sink connector is managed using KafkaConnectors or the Kafka Connect API

4.1.2. Tasks

Data transfer orchestrated by the Kafka Connect runtime is split into tasks that run in parallel. A task is started using the configuration supplied by a connector instance. Kafka Connect distributes the task configurations to workers, which instantiate and execute tasks.

- A source connector task polls the external data system and returns a list of records that a worker sends to the Kafka brokers.
- A sink connector task receives Kafka records from a worker for writing to the external data system.

For sink connectors, the number of tasks created relates to the number of partitions being consumed. For source connectors, how the source data is partitioned is defined by the connector. You can control the maximum number of tasks that can run in parallel by setting `tasksMax` in the connector configuration. The connector might create fewer tasks than the maximum setting. For example, the connector might create fewer tasks if it's not possible to split the source data into that many partitions.

NOTE	In the context of Kafka Connect, a *partition* can mean a topic partition or a *shard of data* in an external system.

4.1.3. Workers

Workers employ the connector configuration deployed to the Kafka Connect cluster. The configuration is stored in an internal Kafka topic used by Kafka Connect. Workers also run connectors and their tasks.

A Kafka Connect cluster contains a group of workers with the same `group.id`. The ID identifies the cluster within Kafka. The ID is assigned in the worker configuration through the `KafkaConnect` resource. Worker configuration also specifies the names of internal Kafka Connect topics. The topics store connector configuration, offset, and status information. The group ID and names of these topics must also be unique to the Kafka Connect cluster.

Workers are assigned one or more connector instances and tasks. The distributed approach to deploying Kafka Connect is fault tolerant and scalable. If a worker pod fails, the tasks it was running are reassigned to active workers. You can add to a group of worker pods through configuration of the `replicas` property in the `KafkaConnect` resource.

4.1.4. Transforms

Kafka Connect translates and transforms external data. Single-message transforms change messages into a format suitable for the target destination. For example, a transform might insert or rename a field. Transforms can also filter and route data. Plugins contain the implementation required for workers to perform one or more transformations.

- Source connectors apply transforms before converting data into a format supported by Kafka.
- Sink connectors apply transforms after converting data into a format suitable for an external data system.

A transform comprises a set of Java class files packaged in a JAR file for inclusion in a connector plugin. Kafka Connect provides a set of standard transforms, but you can also create your own.

4.1.5. Converters

When a worker receives data, it converts the data into an appropriate format using a converter. You specify converters for workers in the worker `config` in the `KafkaConnect` resource.

Kafka Connect can convert data to and from formats supported by Kafka, such as JSON or Avro. It also supports schemas for structuring data. If you are not converting data into a structured format, you don't need to enable schemas.

NOTE	You can also specify converters for specific connectors to override the general Kafka Connect worker configuration that applies to all workers.

Additional resources

- Apache Kafka documentation
- Kafka Connect configuration of workers
- Synchronizing data between Kafka clusters using MirrorMaker 2

Chapter 5. Kafka Bridge interface

The Kafka Bridge provides a RESTful interface that allows HTTP-based clients to interact with a Kafka cluster. It offers the advantages of a web API connection to Strimzi, without the need for client applications to interpret the Kafka protocol.

The API has two main resources — `consumers` and `topics` — that are exposed and made accessible through endpoints to interact with consumers and producers in your Kafka cluster. The resources relate only to the Kafka Bridge, not the consumers and producers connected directly to Kafka.

5.1. HTTP requests

The Kafka Bridge supports HTTP requests to a Kafka cluster, with methods to:

- Send messages to a topic.
- Retrieve messages from topics.
- Retrieve a list of partitions for a topic.
- Create and delete consumers.
- Subscribe consumers to topics, so that they start receiving messages from those topics.
- Retrieve a list of topics that a consumer is subscribed to.
- Unsubscribe consumers from topics.
- Assign partitions to consumers.
- Commit a list of consumer offsets.
- Seek on a partition, so that a consumer starts receiving messages from the first or last offset position, or a given offset position.

The methods provide JSON responses and HTTP response code error handling. Messages can be sent in JSON or binary formats.

Clients can produce and consume messages without the requirement to use the native Kafka protocol.

Additional resources

- To view the API documentation, including example requests and responses, see Using the Strimzi Kafka Bridge.

5.2. Supported clients for the Kafka Bridge

You can use the Kafka Bridge to integrate both *internal* and *external* HTTP client applications with your Kafka cluster.

Internal clients

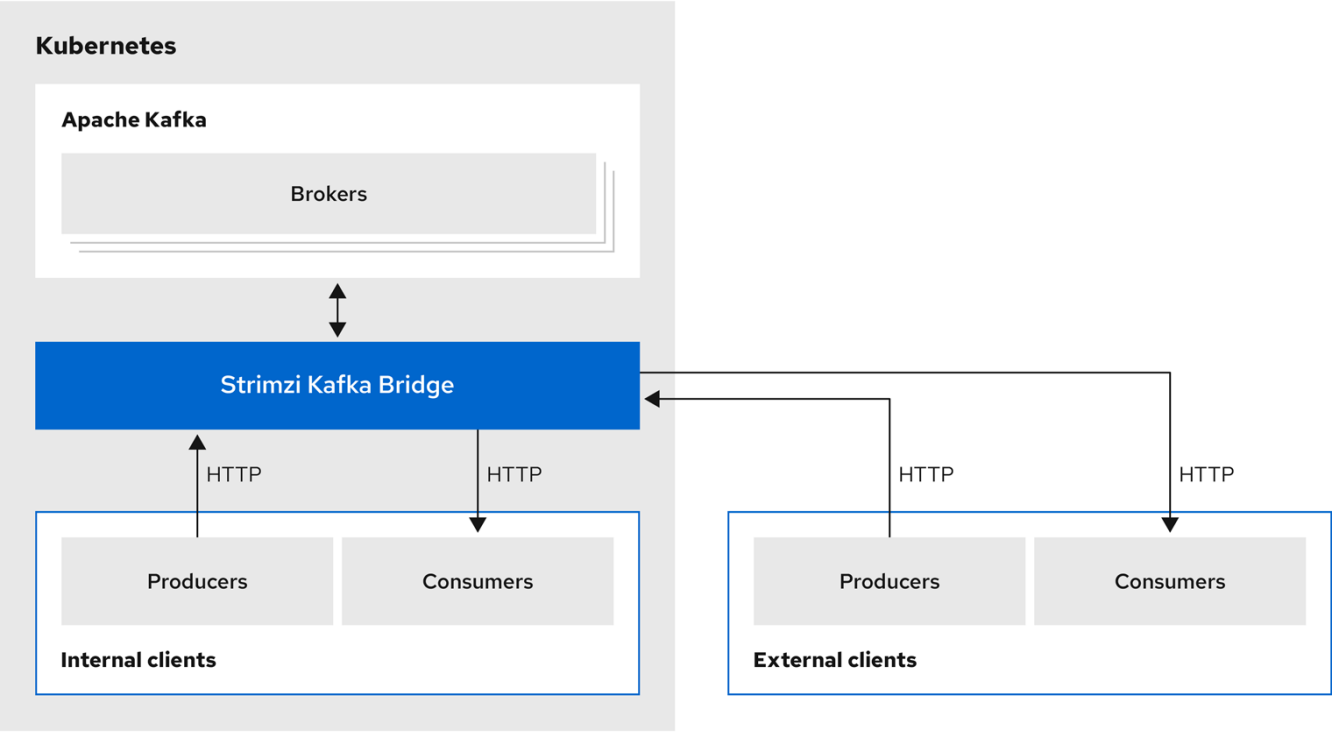
Internal clients are container-based HTTP clients running in *the same* Kubernetes cluster as the Kafka Bridge itself. Internal clients can access the Kafka Bridge on the host and port defined in

the `KafkaBridge` custom resource.

External clients

External clients are HTTP clients running *outside* the Kubernetes cluster in which the Kafka Bridge is deployed and running. External clients can access the Kafka Bridge through an OpenShift Route, a loadbalancer service, or using an Ingress.

HTTP internal and external client integration



Chapter 6. Strimzi Operators

Strimzi supports Kafka using *Operators* to deploy and manage the components and dependencies of Kafka to Kubernetes.

Operators are a method of packaging, deploying, and managing a Kubernetes application. Strimzi Operators extend Kubernetes functionality, automating common and complex tasks related to a Kafka deployment. By implementing knowledge of Kafka operations in code, Kafka administration tasks are simplified and require less manual intervention.

Operators

Strimzi provides operators for managing a Kafka cluster running within a Kubernetes cluster.

Cluster Operator

Deploys and manages Apache Kafka clusters, Kafka Connect, Kafka MirrorMaker, Kafka Bridge, Kafka Exporter, Cruise Control, and the Entity Operator

Entity Operator

Comprises the Topic Operator and User Operator

Topic Operator

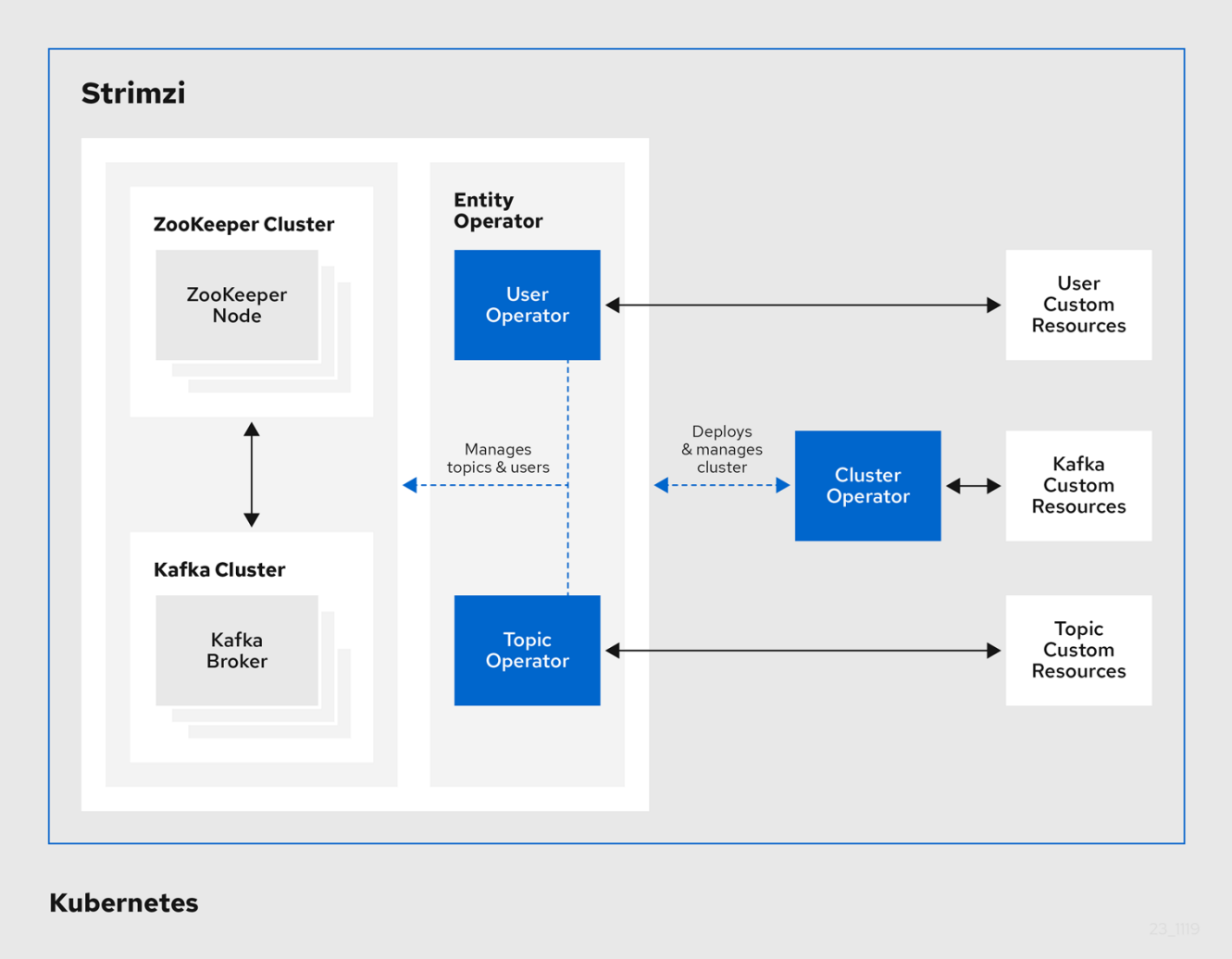
Manages Kafka topics

User Operator

Manages Kafka users

The Cluster Operator can deploy the Topic Operator and User Operator as part of an **Entity Operator** configuration at the same time as a Kafka cluster.

Operators within the Strimzi architecture



6.1. Cluster Operator

Strimzi uses the Cluster Operator to deploy and manage clusters. By default, when you deploy Strimzi a single Cluster Operator replica is deployed. You can add replicas with leader election so that additional Cluster Operators are on standby in case of disruption.

The Cluster Operator manages the clusters of the following Kafka components:

- Kafka (including ZooKeeper, Entity Operator, Kafka Exporter, and Cruise Control)
- Kafka Connect
- Kafka MirrorMaker
- Kafka Bridge

The clusters are deployed using custom resources.

For example, to deploy a Kafka cluster:

- A `Kafka` resource with the cluster configuration is created within the Kubernetes cluster.
- The Cluster Operator deploys a corresponding Kafka cluster, based on what is declared in the `Kafka` resource.

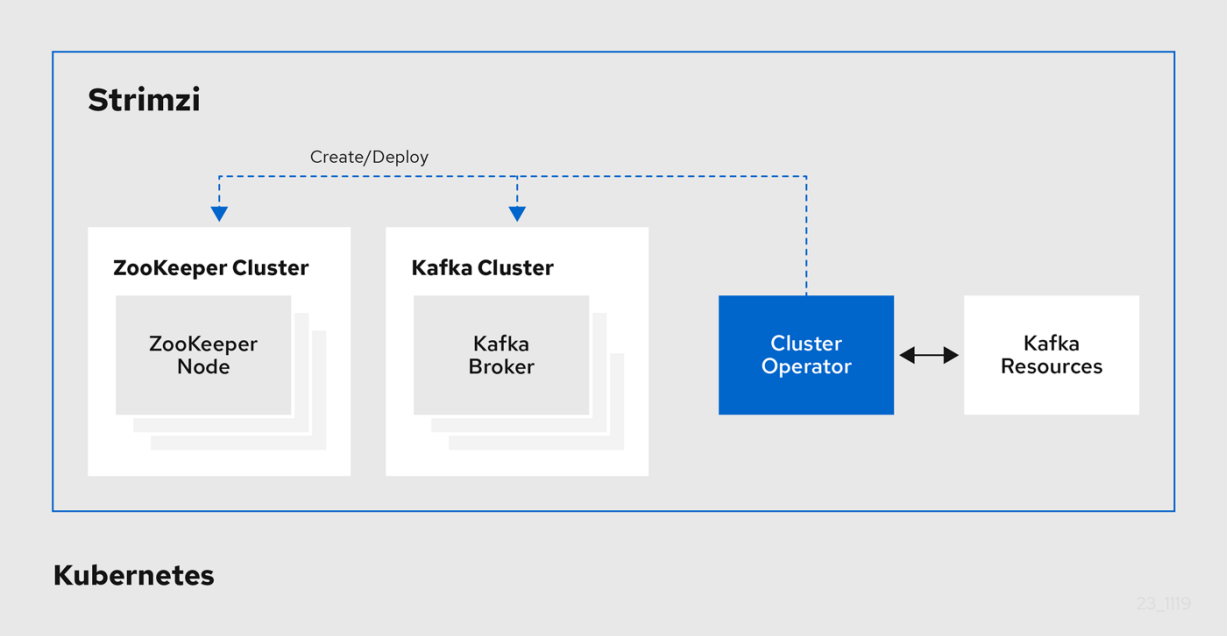
The Cluster Operator can also deploy the following Strimzi operators through configuration of the `Kafka` resource:

- Topic Operator to provide operator-style topic management through `KafkaTopic` custom resources
- User Operator to provide operator-style user management through `KafkaUser` custom resources

The Topic Operator and User Operator function within the Entity Operator on deployment.

You can use the Cluster Operator with a deployment of Strimzi Drain Cleaner to help with pod evictions. By deploying the Strimzi Drain Cleaner, you can use the Cluster Operator to move Kafka pods instead of Kubernetes. Strimzi Drain Cleaner annotates pods being evicted with a rolling update annotation. The annotation informs the Cluster Operator to perform the rolling update.

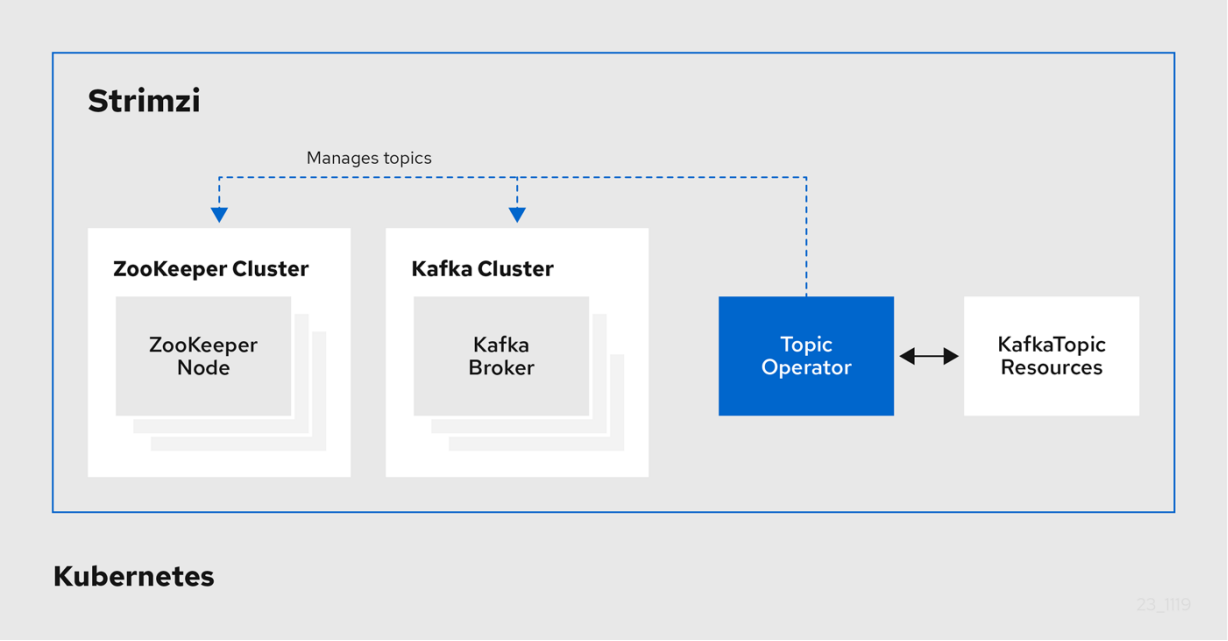
Example architecture for the Cluster Operator



6.2. Topic Operator

The Topic Operator provides a way of managing topics in a Kafka cluster through Kubernetes resources.

Example architecture for the Topic Operator



The role of the Topic Operator is to keep a set of `KafkaTopic` Kubernetes resources describing Kafka topics in-sync with corresponding Kafka topics.

Specifically, if a `KafkaTopic` is:

- Created, the Topic Operator creates the topic
- Deleted, the Topic Operator deletes the topic
- Changed, the Topic Operator updates the topic

Working in the other direction, if a topic is:

- Created within the Kafka cluster, the Operator creates a `KafkaTopic`
- Deleted from the Kafka cluster, the Operator deletes the `KafkaTopic`
- Changed in the Kafka cluster, the Operator updates the `KafkaTopic`

This allows you to declare a `KafkaTopic` as part of your application's deployment and the Topic Operator will take care of creating the topic for you. Your application just needs to deal with producing or consuming from the necessary topics.

The Topic Operator maintains information about each topic in a *topic store*, which is continually synchronized with updates from Kafka topics or Kubernetes `KafkaTopic` custom resources. Updates from operations applied to a local in-memory topic store are persisted to a backup topic store on disk. If a topic is reconfigured or reassigned to other brokers, the `KafkaTopic` will always be up to date.

6.3. User Operator

The User Operator manages Kafka users for a Kafka cluster by watching for `KafkaUser` resources that describe Kafka users, and ensuring that they are configured properly in the Kafka cluster.

For example, if a `KafkaUser` is:

- Created, the User Operator creates the user it describes
- Deleted, the User Operator deletes the user it describes
- Changed, the User Operator updates the user it describes

Unlike the Topic Operator, the User Operator does not sync any changes from the Kafka cluster with the Kubernetes resources. Kafka topics can be created by applications directly in Kafka, but it is not expected that the users will be managed directly in the Kafka cluster in parallel with the User Operator.

The User Operator allows you to declare a `KafkaUser` resource as part of your application's deployment. You can specify the authentication and authorization mechanism for the user. You can also configure *user quotas* that control usage of Kafka resources to ensure, for example, that a user does not monopolize access to a broker.

When the user is created, the user credentials are created in a `Secret`. Your application needs to use the user and its credentials for authentication and to produce or consume messages.

In addition to managing credentials for authentication, the User Operator also manages authorization rules by including a description of the user's access rights in the `KafkaUser` declaration.

6.4. Feature gates in Strimzi Operators

You can enable and disable some features of operators using *feature gates*.

Feature gates are set in the operator configuration and have three stages of maturity: alpha, beta, or General Availability (GA).

For more information, see Feature gates.

Chapter 7. Kafka configuration

A deployment of Kafka components to a Kubernetes cluster using Strimzi is highly configurable through the application of custom resources. Custom resources are created as instances of APIs added by Custom resource definitions (CRDs) to extend Kubernetes resources.

CRDs act as configuration instructions to describe the custom resources in a Kubernetes cluster, and are provided with Strimzi for each Kafka component used in a deployment, as well as users and topics. CRDs and custom resources are defined as YAML files. Example YAML files are provided with the Strimzi distribution.

CRDs also allow Strimzi resources to benefit from native Kubernetes features like CLI accessibility and configuration validation.

In this section we look at how Kafka components are configured through custom resources, starting with common configuration points and then important configuration considerations specific to components.

Strimzi provides example configuration files, which can serve as a starting point when building your own Kafka component configuration for deployment.

7.1. Custom resources

After a new custom resource type is added to your cluster by installing a CRD, you can create instances of the resource based on its specification.

The custom resources for Strimzi components have common configuration properties, which are defined under `spec`.

In this fragment from a Kafka topic custom resource, the `apiVersion` and `kind` properties identify the associated CRD. The `spec` property shows configuration that defines the number of partitions and replicas for the topic.

Kafka topic custom resource

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaTopic
metadata:
  name: my-topic
  labels:
    strimzi.io/cluster: my-cluster
spec:
  partitions: 1
  replicas: 1
  # ...
```

There are many additional configuration options that can be incorporated into a YAML definition, some common and some specific to a particular component.

Additional resources

- Extend the Kubernetes API with CustomResourceDefinitions

7.2. Common configuration

Some of the configuration options common to resources are described here. Security and metrics collection might also be adopted where applicable.

Bootstrap servers

Bootstrap servers are used for host/port connection to a Kafka cluster for:

- Kafka Connect
- Kafka Bridge
- Kafka MirrorMaker producers and consumers

CPU and memory resources

You request CPU and memory resources for components. Limits specify the maximum resources that can be consumed by a given container.

Resource requests and limits for the Topic Operator and User Operator are set in the `Kafka` resource.

Logging

You define the logging level for the component. Logging can be defined directly (inline) or externally using a config map.

Healthchecks

Healthcheck configuration introduces *liveness* and *readiness* probes to know when to restart a container (liveness) and when a container can accept traffic (readiness).

JVM options

JVM options provide maximum and minimum memory allocation to optimize the performance of the component according to the platform it is running on.

Pod scheduling

Pod schedules use *affinity/anti-affinity* rules to determine under what circumstances a pod is scheduled onto a node.

Example YAML showing common configuration

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: my-cluster
spec:
  # ...
  bootstrapServers: my-cluster-kafka-bootstrap:9092
```

```
  resources:
    requests:
      cpu: 12
      memory: 64Gi
    limits:
      cpu: 12
      memory: 64Gi
  logging:
    type: inline
    loggers:
      connect.root.logger.level: "INFO"
  readinessProbe:
    initialDelaySeconds: 15
    timeoutSeconds: 5
  livenessProbe:
    initialDelaySeconds: 15
    timeoutSeconds: 5
  jvmOptions:
    "-Xmx": "2g"
    "-Xms": "2g"
  template:
    pod:
      affinity:
        nodeAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
              - matchExpressions:
                  - key: node-type
                    operator: In
                    values:
                      - fast-network
  # ...
```

7.3. Kafka cluster configuration

A kafka cluster comprises one or more brokers. For producers and consumers to be able to access topics within the brokers, Kafka configuration must define how data is stored in the cluster, and how the data is accessed. You can configure a Kafka cluster to run with multiple broker nodes across racks.

Storage

Kafka and ZooKeeper store data on disks.

Strimzi requires block storage provisioned through `StorageClass`. The file system format for storage must be *XFS* or *EXT4*. Three types of data storage are supported:

Ephemeral (Recommended for development only)

Ephemeral storage stores data for the lifetime of an instance. Data is lost when the instance is restarted.

Persistent

Persistent storage relates to long-term data storage independent of the lifecycle of the instance.

JBOD (Just a Bunch of Disks, suitable for Kafka only)

JBOD allows you to use multiple disks to store commit logs in each broker.

The disk capacity used by an existing Kafka cluster can be increased if supported by the infrastructure.

Listeners

Listeners configure how clients connect to a Kafka cluster.

By specifying a unique name and port for each listener within a Kafka cluster, you can configure multiple listeners.

The following types of listener are supported:

- **Internal listeners** for access within Kubernetes
- **External listeners** for access outside of Kubernetes

You can enable TLS encryption for listeners, and configure authentication.

Internal listeners expose Kafka by specifying an `internal` type:

- `internal` to connect within the same Kubernetes cluster
- `cluster-ip` to expose Kafka using per-broker `ClusterIP` services

External listeners expose Kafka by specifying an external `type`:

- `route` to use OpenShift routes and the default HAProxy router
- `loadbalancer` to use loadbalancer services
- `nodeport` to use ports on Kubernetes nodes
- `ingress` to use Kubernetes *Ingress* and the Ingress NGINX Controller for Kubernetes

NOTE	With the `cluster-ip` type can add your own access mechanism. For example, you can use the listener with a custom Ingress controller or the Kubernetes Gateway API.

If you are using OAuth 2.0 for token-based authentication, you can configure listeners to use the authorization server.

Rack awareness

Racks represent data centers, or racks in data centers, or availability zones. Configure rack awareness to distribute Kafka broker pods and topic replicas across racks. Enable rack awareness using the `rack` property to specify a `topologyKey`. The `topologyKey` is the name of the label assigned to Kubernetes worker nodes, which identifies the rack. Strimzi assigns a rack ID to each Kafka broker. Kafka brokers use the IDs to spread partition replicas across racks. You can

also specify the `RackAwareReplicaSelector` selector plugin to use with rack awareness. The plugin matches the rack IDs of brokers and consumers, so that messages are consumed from the closest replica. To use the plugin, consumers must also have rack awareness enabled. You can enable rack awareness in Kafka Connect, MirrorMaker 2, and the Kafka Bridge.

Example YAML showing Kafka configuration

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    listeners:
      - name: tls
        port: 9093
        type: internal
        tls: true
        authentication:
          type: tls
      - name: external1
        port: 9094
        type: route
        tls: true
        authentication:
          type: tls
    # ...
    storage:
      type: persistent-claim
      size: 10000Gi
    # ...
    rack:
      topologyKey: topology.kubernetes.io/zone
    config:
      replica.selector.class: org.apache.kafka.common.replica.RackAwareReplicaSelector
    # ...
```

7.4. Kafka MirrorMaker configuration

Kafka MirrorMaker replicates data between two or more active Kafka clusters, within or across data centers. To set up MirrorMaker, a source and target (destination) Kafka cluster must be running.

7.4.1. MirrorMaker 2 configuration

MirrorMaker 2 consumes messages from a source Kafka cluster and writes them to a target Kafka cluster.

MirrorMaker 2 uses:

- Source cluster configuration to consume data from the source cluster
- Target cluster configuration to output data to the target cluster

MirrorMaker 2 is based on the Kafka Connect framework, *connectors* managing the transfer of data between clusters.

You configure MirrorMaker 2 to define the Kafka Connect deployment, including the connection details of the source and target clusters, and then run a set of MirrorMaker 2 connectors to make the connection.

MirrorMaker 2 consists of the following connectors:

`MirrorSourceConnector`

The source connector replicates topics from a source cluster to a target cluster. It also replicates ACLs and is necessary for the `MirrorCheckpointConnector` to run.

`MirrorCheckpointConnector`

The checkpoint connector periodically tracks offsets. If enabled, it also synchronizes consumer group offsets between the source and target cluster.

`MirrorHeartbeatConnector`

The heartbeat connector periodically checks connectivity between the source and target cluster.

NOTE If you are using the User Operator to manage ACLs, ACL replication through the connector is not possible.

The process of *mirroring* data from a source cluster to a target cluster is asynchronous. Each MirrorMaker 2 instance mirrors data from one source cluster to one target cluster. You can use more than one MirrorMaker 2 instance to mirror data between any number of clusters.

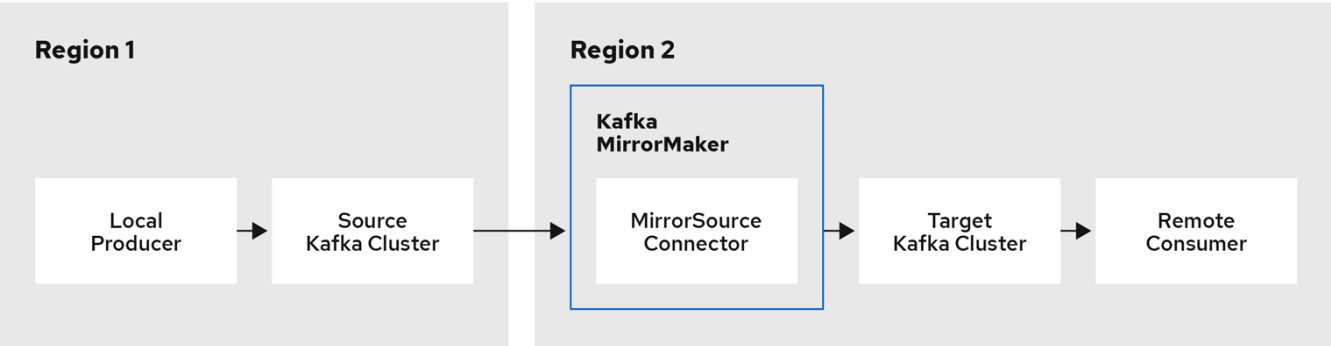


Figure 1. Replication across two clusters

By default, a check for new topics in the source cluster is made every 10 minutes. You can change the frequency by adding `refresh.topics.interval.seconds` to the source connector configuration.

Cluster configuration

You can use MirrorMaker 2 in *active/passive* or *active/active* cluster configurations.

active/active cluster configuration

An active/active configuration has two active clusters replicating data bidirectionally. Applications can use either cluster. Each cluster can provide the same data. In this way, you can make the same data available in different geographical locations. As consumer groups are active in both clusters, consumer offsets for replicated topics are not synchronized back to the source cluster.

active/passive cluster configuration

An active/passive configuration has an active cluster replicating data to a passive cluster. The passive cluster remains on standby. You might use the passive cluster for data recovery in the event of system failure.

The expectation is that producers and consumers connect to active clusters only. A MirrorMaker 2 cluster is required at each target destination.

Bidirectional replication (active/active)

The MirrorMaker 2 architecture supports bidirectional replication in an *active/active* cluster configuration.

Each cluster replicates the data of the other cluster using the concept of *source* and *remote* topics. As the same topics are stored in each cluster, remote topics are automatically renamed by MirrorMaker 2 to represent the source cluster. The name of the originating cluster is prepended to the name of the topic.

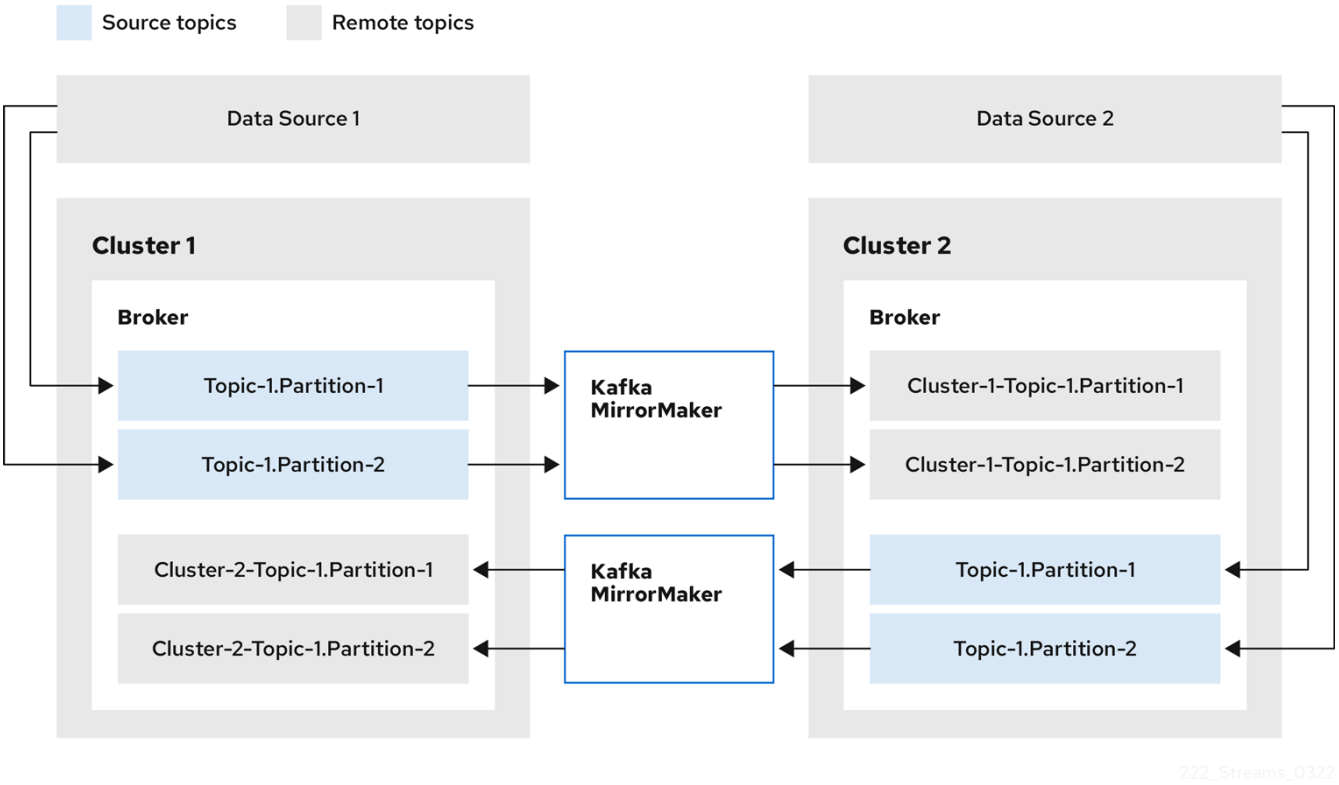


Figure 2. Topic renaming

By flagging the originating cluster, topics are not replicated back to that cluster.

The concept of replication through *remote* topics is useful when configuring an architecture that requires data aggregation. Consumers can subscribe to source and remote topics within the same cluster, without the need for a separate aggregation cluster.

Unidirectional replication (active/passive)

The MirrorMaker 2 architecture supports unidirectional replication in an *active/passive* cluster configuration.

You can use an *active/passive* cluster configuration to make backups or migrate data to another cluster. In this situation, you might not want automatic renaming of remote topics.

You can override automatic renaming by adding `IdentityReplicationPolicy` to the source connector configuration. With this configuration applied, topics retain their original names.

Example YAML showing MirrorMaker 2 configuration

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaMirrorMaker2
metadata:
  name: my-mirror-maker2
  spec:
    version: 3.4.0
    connectCluster: "my-cluster-target"
    clusters:
```

```
    - alias: "my-cluster-source"
      bootstrapServers: my-cluster-source-kafka-bootstrap:9092
    - alias: "my-cluster-target"
      bootstrapServers: my-cluster-target-kafka-bootstrap:9092
    mirrors:
    - sourceCluster: "my-cluster-source"
      targetCluster: "my-cluster-target"
      sourceConnector: {}
    topicsPattern: ".*"
    groupsPattern: "group1|group2|group3"
```

7.4.2. MirrorMaker configuration

Kafka MirrorMaker (also referred to as MirrorMaker 1) uses producers and consumers to replicate data across clusters as follows:

- Consumers consume data from the source cluster
- Producers output data to the target cluster

Consumer and producer configuration includes any required authentication and encryption settings. An `include` property defines the topics to mirror from the source to the target cluster.

NOTE	MirrorMaker was deprecated in Kafka 3.0.0 and will be removed in Kafka 4.0.0. As a consequence, the Strimzi `KafkaMirrorMaker` custom resource which is used to deploy MirrorMaker has been deprecated. The `KafkaMirrorMaker` resource will be removed from Strimzi when Kafka 4.0.0 is adopted.

Key Consumer configuration

Consumer group identifier

The consumer group ID for a MirrorMaker consumer so that messages consumed are assigned to a consumer group.

Number of consumer streams

A value to determine the number of consumers in a consumer group that consume a message in parallel.

Offset commit interval

An offset commit interval to set the time between consuming and committing a message.

Key Producer configuration

Cancel option for send failure

You can define whether a message send failure is ignored or MirrorMaker is terminated and recreated.

Example YAML showing MirrorMaker configuration

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaMirrorMaker
metadata:
  name: my-mirror-maker
spec:
  # ...
  consumer:
    bootstrapServers: my-source-cluster-kafka-bootstrap:9092
    groupId: "my-group"
    numStreams: 2
    offsetCommitInterval: 120000
    # ...
  producer:
    # ...
    abortOnSendFailure: false
    # ...
  include: "my-topic|other-topic"
  # ...
```

7.5. Kafka Connect configuration

Use Strimzi's `KafkaConnect` resource to quickly and easily create new Kafka Connect clusters.

When you deploy Kafka Connect using the `KafkaConnect` resource, you specify bootstrap server addresses (in `spec.bootstrapServers`) for connecting to a Kafka cluster. You can specify more than one address in case a server goes down. You also specify the authentication credentials and TLS encryption certificates to make a secure connection.

NOTE	The Kafka cluster doesn't need to be managed by Strimzi or deployed to a Kubernetes cluster.

You can also use the `KafkaConnect` resource to specify the following:

- Plugin configuration to build a container image that includes the plugins to make connections
- Configuration for the worker pods that belong to the Kafka Connect cluster
- An annotation to enable use of the `KafkaConnector` resource to manage plugins

The Cluster Operator manages Kafka Connect clusters deployed using the `KafkaConnect` resource and connectors created using the `KafkaConnector` resource.

Plugin configuration

Plugins provide the implementation for creating connector instances. When a plugin is instantiated, configuration is provided for connection to a specific type of external data system. Plugins provide a set of one or more JAR files that define a connector and task implementation for connecting to a given kind of data source. Plugins for many external systems are available for use

with Kafka Connect. You can also create your own plugins.

The configuration describes the source input data and target output data to feed into and out of Kafka Connect. For a source connector, external source data must reference specific topics that will store the messages. The plugins might also contain the libraries and files needed to transform the data.

A Kafka Connect deployment can have one or more plugins, but only one version of each plugin.

You can create a custom Kafka Connect image that includes your choice of plugins. You can create the image in two ways:

- Automatically using Kafka Connect configuration
- Manually using a Dockerfile and a Kafka container image as a base image

To create the container image automatically, you specify the plugins to add to your Kafka Connect cluster using the `build` property of the `KafkaConnect` resource. Strimzi automatically downloads and adds the plugin artifacts to a new container image.

Example plugin configuration

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: my-connect-cluster
  annotations:
    strimzi.io/use-connector-resources: "true"
spec:
  # ...
  build: ①
    output: ②
      type: docker
      image: my-registry.io/my-org/my-connect-cluster:latest
      pushSecret: my-registry-credentials
    plugins: ③
      - name: my-connector
        artifacts:
          - type: tgz
            url: https://<plugin_download_location>.tgz
            sha512sum: <checksum_to_verify_the_plugin>
      # ...
  # ...
```

① Build configuration properties for building a container image with plugins automatically.

② Configuration of the container registry where new images are pushed. The `output` properties describe the type and name of the image, and optionally the name of the secret containing the credentials needed to access the container registry.

③ List of plugins and their artifacts to add to the new container image. The `plugins` properties describe the type of artifact and the URL from which the artifact is downloaded. Each plugin

must be configured with at least one artifact. Additionally, you can specify a SHA-512 checksum to verify the artifact before unpacking it.

If you are using a Dockerfile to build an image, you can use Strimzi's latest container image as a base image to add your plugin configuration file.

Example showing manual addition of plugin configuration

```
FROM quay.io/strimzi/kafka:0.35.1-kafka-3.4.0
USER root:root
COPY ./my-plugins/ /opt/kafka/plugins/
USER 1001
```

Kafka Connect cluster configuration for workers

You specify the configuration for workers in the `config` property of the `KafkaConnect` resource.

A distributed Kafka Connect cluster has a group ID and a set of internal configuration topics.

- `group.id`
- `offset.storage.topic`
- `config.storage.topic`
- `status.storage.topic`

Kafka Connect clusters are configured by default with the same values for these properties. Kafka Connect clusters cannot share the group ID or topic names as it will create errors. If multiple different Kafka Connect clusters are used, these settings must be unique for the workers of each Kafka Connect cluster created.

The names of the connectors used by each Kafka Connect cluster must also be unique.

In the following example worker configuration, JSON converters are specified. A replication factor is set for the internal Kafka topics used by Kafka Connect. This should be at least 3 for a production environment. Changing the replication factor after the topics have been created will have no effect.

Example worker configuration

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
# ...
spec:
  config:
    # ...
    group.id: my-connect-cluster ①
    offset.storage.topic: my-connect-cluster-offsets ②
    config.storage.topic: my-connect-cluster-configs ③
    status.storage.topic: my-connect-cluster-status ④
    key.converter: org.apache.kafka.connect.json.JsonConverter ⑤
    value.converter: org.apache.kafka.connect.json.JsonConverter ⑥
```

```
    key.converter.schemas.enable: true ⑦
    value.converter.schemas.enable: true ⑧
    config.storage.replication.factor: 3 ⑨
    offset.storage.replication.factor: 3 ⑩
    status.storage.replication.factor: 3 ⑪
  # ...
```

① The Kafka Connect cluster ID within Kafka. Must be unique for each Kafka Connect cluster.

② Kafka topic that stores connector offsets. Must be unique for each Kafka Connect cluster.

③ Kafka topic that stores connector and task status configurations. Must be unique for each Kafka Connect cluster.

④ Kafka topic that stores connector and task status updates. Must be unique for each Kafka Connect cluster.

⑤ Converter to transform message keys into JSON format for storage in Kafka.

⑥ Converter to transform message values into JSON format for storage in Kafka.

⑦ Schema enabled for converting message keys into structured JSON format.

⑧ Schema enabled for converting message values into structured JSON format.

⑨ Replication factor for the Kafka topic that stores connector offsets.

⑩ Replication factor for the Kafka topic that stores connector and task status configurations.

⑪ Replication factor for the Kafka topic that stores connector and task status updates.

`KafkaConnector` management of connectors

After plugins have been added to the container image used for the worker pods in a deployment, you can use Strimzi's `KafkaConnector` custom resource or the Kafka Connect API to manage connector instances. You can also create new connector instances using these options.

The `KafkaConnector` resource offers a Kubernetes-native approach to management of connectors by the Cluster Operator. To manage connectors with `KafkaConnector` resources, you must specify an annotation in your `KafkaConnect` custom resource.

Annotation to enable KafkaConnectors

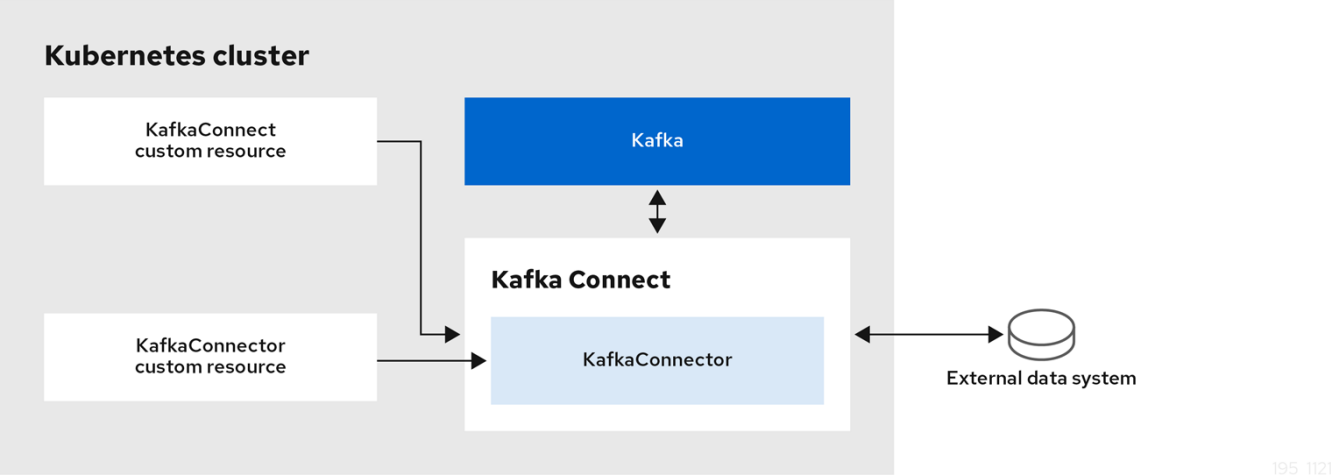
```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: my-connect-cluster
  annotations:
    strimzi.io/use-connector-resources: "true"
  # ...
```

Setting `use-connector-resources` to `true` enables KafkaConnectors to create, delete, and reconfigure connectors.

If `use-connector-resources` is enabled in your `KafkaConnect` configuration, you must use the

`KafkaConnector` resource to define and manage connectors. `KafkaConnector` resources are configured to connect to external systems. They are deployed to the same Kubernetes cluster as the Kafka Connect cluster and Kafka cluster interacting with the external data system.

Kafka components are contained in the same Kubernetes cluster



The configuration specifies how connector instances connect to an external data system, including any authentication. You also need to state what data to watch. For a source connector, you might provide a database name in the configuration. You can also specify where the data should sit in Kafka by specifying a target topic name.

Use `tasksMax` to specify the maximum number of tasks. For example, a source connector with `tasksMax: 2` might split the import of source data into two tasks.

Example KafkaConnector source connector configuration

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnector
metadata:
  name: my-source-connector  ①
  labels:
    strimzi.io/cluster: my-connect-cluster ②
spec:
  class: org.apache.kafka.connect.file.FileStreamSourceConnector ③
  tasksMax: 2 ④
  config: ⑤
    file: "/opt/kafka/LICENSE" ⑥
    topic: my-topic ⑦
    # ...
```

① Name of the `KafkaConnector` resource, which is used as the name of the connector. Use any name that is valid for a Kubernetes resource.

② Name of the Kafka Connect cluster to create the connector instance in. Connectors must be deployed to the same namespace as the Kafka Connect cluster they link to.

③ Full name of the connector class. This should be present in the image being used by the Kafka Connect cluster.

④ Maximum number of Kafka Connect tasks that the connector can create.

⑤ Connector configuration as key-value pairs.

⑥ Location of the external data file. In this example, we're configuring the `FileStreamSourceConnector` to read from the `/opt/kafka/LICENSE` file.

⑦ Kafka topic to publish the source data to.

NOTE	You can load confidential configuration values for a connector from Kubernetes Secrets or ConfigMaps.

Kafka Connect API

Use the Kafka Connect REST API as an alternative to using `KafkaConnector` resources to manage connectors. The Kafka Connect REST API is available as a service running on `<connect_cluster_name>-connect-api:8083`, where *<connect_cluster_name>* is the name of your Kafka Connect cluster.

You add the connector configuration as a JSON object.

Example curl request to add connector configuration

```
curl -X POST \
  http://my-connect-cluster-connect-api:8083/connectors \
  -H 'Content-Type: application/json' \
  -d '{ "name": "my-source-connector",
    "config":
    {
      "connector.class":"org.apache.kafka.connect.file.FileStreamSourceConnector",
      "file": "/opt/kafka/LICENSE",
      "topic":"my-topic",
      "tasksMax": "4",
      "type": "source"
    }
}'
```

If KafkaConnectors are enabled, manual changes made directly using the Kafka Connect REST API are reverted by the Cluster Operator.

The operations supported by the REST API are described in the Apache Kafka Connect API documentation.

NOTE	You can expose the Kafka Connect API service outside Kubernetes. You do this by creating a service that uses a connection mechanism that provides the access, such as an ingress or route. Use advisedly as the connection is insecure.

Additional resources

- Kafka Connect configuration options
- Kafka Connect configuration for multiple instances

- Extending Kafka Connect with plugins
- Creating a new container image automatically using Strimzi
- Creating a Docker image from the Kafka Connect base image
- Build schema reference
- Source and sink connector configuration options
- Loading configuration values from external sources

7.6. Kafka Bridge configuration

A Kafka Bridge configuration requires a bootstrap server specification for the Kafka cluster it connects to, as well as any encryption and authentication options required.

Kafka Bridge consumer and producer configuration is standard, as described in the Apache Kafka configuration documentation for consumers and Apache Kafka configuration documentation for producers.

HTTP-related configuration options set the port connection which the server listens on.

CORS

The Kafka Bridge supports the use of Cross-Origin Resource Sharing (CORS). CORS is a HTTP mechanism that allows browser access to selected resources from more than one origin, for example, resources on different domains. If you choose to use CORS, you can define a list of allowed resource origins and HTTP methods for interaction with the Kafka cluster through the Kafka Bridge. The lists are defined in the `http` specification of the Kafka Bridge configuration.

CORS allows for *simple* and *preflighted* requests between origin sources on different domains.

- A simple request is a HTTP request that must have an allowed origin defined in its header.
- A preflighted request sends an initial OPTIONS HTTP request before the actual request to check that the origin and the method are allowed.

Example YAML showing Kafka Bridge configuration

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaBridge
metadata:
  name: my-bridge
spec:
  # ...
  bootstrapServers: my-cluster-kafka:9092
  http:
    port: 8080
    cors:
      allowedOrigins: "https://strimzi.io"
      allowedMethods: "GET,POST,PUT,DELETE,OPTIONS,PATCH"
```

```
  consumer:
    config:
      auto.offset.reset: earliest
  producer:
    config:
      delivery.timeout.ms: 300000
  # ...
```

Additional resources

- Fetch CORS specification

Chapter 8. Securing Kafka

A secure deployment of Strimzi might encompass one or more of the following security measures:

- Encryption for data exchange
- Authentication to prove identity
- Authorization to allow or decline actions executed by users
- Running Strimzi on FIPS-enabled Kubernetes clusters to ensure data security and system interoperability

8.1. Encryption

Strimzi supports Transport Layer Security (TLS), a protocol for encrypted communication.

Communication is always encrypted for communication between:

- Kafka brokers
- ZooKeeper nodes
- Operators and Kafka brokers
- Operators and ZooKeeper nodes
- Kafka Exporter

You can also configure TLS encryption between Kafka brokers and clients. TLS is specified for external clients when configuring an external listener for the Kafka broker.

Strimzi components and Kafka clients use digital certificates for encryption. The Cluster Operator sets up certificates to enable encryption within the Kafka cluster. You can provide your own server certificates, referred to as *Kafka listener certificates*, for communication between Kafka clients and Kafka brokers, and inter-cluster communication.

Strimzi uses *Secrets* to store the certificates and private keys required for mTLS in PEM and PKCS #12 format.

A TLS CA (certificate authority) issues certificates to authenticate the identity of a component. Strimzi verifies the certificates for the components against the CA certificate.

- Strimzi components are verified against the *cluster CA* CA
- Kafka clients are verified against the *clients CA* CA

8.2. Authentication

Kafka listeners use authentication to ensure a secure client connection to the Kafka cluster.

Supported authentication mechanisms:

- mTLS authentication (on listeners with TLS-enabled encryption)

- SASL SCRAM-SHA-512
- OAuth 2.0 token based authentication
- Custom authentication

The User Operator manages user credentials for mTLS and SCRAM authentication, but not OAuth 2.0. For example, through the User Operator you can create a user representing a client that requires access to the Kafka cluster, and specify `tls` as the authentication type.

Using OAuth 2.0 token-based authentication, application clients can access Kafka brokers without exposing account credentials. An authorization server handles the granting of access and inquiries about access.

Custom authentication allows for any type of kafka-supported authentication. It can provide more flexibility, but also adds complexity.

8.3. Authorization

Kafka clusters use authorization to control the operations that are permitted on Kafka brokers by specific clients or users. If applied to a Kafka cluster, authorization is enabled for all listeners used for client connection.

If a user is added to a list of *super users* in a Kafka broker configuration, the user is allowed unlimited access to the cluster regardless of any authorization constraints implemented through authorization mechanisms.

Supported authorization mechanisms:

- Simple authorization
- OAuth 2.0 authorization (if you are using OAuth 2.0 token-based authentication)
- Open Policy Agent (OPA) authorization
- Custom authorization

Simple authorization uses `AclAuthorizer`, the default Kafka authorization plugin. `AclAuthorizer` uses Access Control Lists (ACLs) to define which users have access to which resources. For custom authorization, you configure your own `Authorizer` plugin to enforce ACL rules.

OAuth 2.0 and OPA provide policy-based control from an authorization server. Security policies and permissions used to grant access to resources on Kafka brokers are defined in the authorization server.

URLs are used to connect to the authorization server and verify that an operation requested by a client or user is allowed or denied. Users and clients are matched against the policies created in the authorization server that permit access to perform specific actions on Kafka brokers.

8.4. Federal Information Processing Standards (FIPS)

Federal Information Processing Standards (FIPS) are a set of security standards established by the

US government to ensure the confidentiality, integrity, and availability of sensitive data and information that is processed or transmitted by information systems. The OpenJDK used in Strimzi container images automatically enables FIPS mode when running on a FIPS-enabled Kubernetes cluster.

NOTE If you don't want to use FIPS, you can disable it in the deployment configuration of the Cluster Operator using the `FIPS_MODE` environment variable.

Chapter 9. Monitoring

Monitoring data allows you to monitor the performance and health of Strimzi. You can configure your deployment to capture metrics data for analysis and notifications.

Metrics data is useful when investigating issues with connectivity and data delivery. For example, metrics data can identify under-replicated partitions or the rate at which messages are consumed. Alerting rules can provide time-critical notifications on such metrics through a specified communications channel. Monitoring visualizations present real-time metrics data to help determine when and how to update the configuration of your deployment. Example metrics configuration files are provided with Strimzi.

Distributed tracing complements the gathering of metrics data by providing a facility for end-to-end tracking of messages through Strimzi.

Cruise Control provides support for rebalancing of Kafka clusters, based on workload data.

Metrics and monitoring tools

Strimzi can employ the following tools for metrics and monitoring:

Prometheus

Prometheus pulls metrics from Kafka, ZooKeeper and Kafka Connect clusters. The Prometheus **Alertmanager** plugin handles alerts and routes them to a notification service.

Kafka Exporter

Kafka Exporter adds additional Prometheus metrics.

Grafana

Grafana Labs provides dashboard visualizations of Prometheus metrics.

Jaeger

Jaeger documentation provides distributed tracing support to track transactions between applications.

Cruise Control

Cruise Control monitors data distribution and performs data rebalances across a Kafka cluster.

9.1. Prometheus

Prometheus can extract metrics data from Kafka components and the Strimzi Operators.

To use Prometheus to obtain metrics data and provide alerts, Prometheus and the Prometheus Alertmanager plugin must be deployed. Kafka resources must also be deployed or redeployed with metrics configuration to expose the metrics data.

Prometheus scrapes the exposed metrics data for monitoring. Alertmanager issues alerts when conditions indicate potential problems, based on pre-defined alerting rules.

Sample metrics and alerting rules configuration files are provided with Strimzi. The sample alerting mechanism provided with Strimzi is configured to send notifications to a Slack channel.

9.2. Grafana

Grafana uses the metrics data exposed by Prometheus to present dashboard visualizations for monitoring.

A deployment of Grafana is required, with Prometheus added as a data source. Example dashboards, supplied with Strimzi as JSON files, are imported through the Grafana interface to present monitoring data.

9.3. Kafka Exporter

Kafka Exporter is an open source project to enhance monitoring of Apache Kafka brokers and clients. Kafka Exporter is deployed with a Kafka cluster to extract additional Prometheus metrics data from Kafka brokers related to offsets, consumer groups, consumer lag, and topics. You can use the Grafana dashboard provided to visualize the data collected by Prometheus from Kafka Exporter.

A sample configuration file, alerting rules and Grafana dashboard for Kafka Exporter are provided with Strimzi.

9.4. Distributed tracing

Distributed tracing tracks the progress of transactions between applications in a distributed system. In a microservices architecture, tracing tracks the progress of transactions between services. Trace data is useful for monitoring application performance and investigating issues with target systems and end-user applications.

In Strimzi, tracing facilitates the end-to-end tracking of messages: from source systems to Kafka, and then from Kafka to target systems and applications. Distributed tracing complements the monitoring of metrics in Grafana dashboards, as well as the component loggers.

Support for tracing is built in to the following Kafka components:

- MirrorMaker to trace messages from a source cluster to a target cluster
- Kafka Connect to trace messages consumed and produced by Kafka Connect
- Kafka Bridge to trace messages between Kafka and HTTP client applications

Tracing is not supported for Kafka brokers.

You enable and configure tracing for these components through their custom resources. You add tracing configuration using `spec.template` properties.

You enable tracing by specifying a tracing type using the `spec.tracing.type` property:

`opentelemetry`

Specify `type: opentelemetry` to use OpenTelemetry. By Default, OpenTelemetry uses the OTLP (OpenTelemetry Protocol) exporter and endpoint to get trace data. You can specify other tracing systems supported by OpenTelemetry, including Jaeger tracing. To do this, you change the OpenTelemetry exporter and endpoint in the tracing configuration.

`jaeger`

Specify `type:jaeger` to use OpenTracing and the Jaeger client to get trace data.

NOTE	Support for `type: jaeger` tracing is deprecated. The Jaeger clients are now retired and the OpenTracing project archived. As such, we cannot guarantee their support for future Kafka versions. If possible, we will maintain the support for `type: jaeger` tracing until June 2023 and remove it afterwards. Please migrate to OpenTelemetry as soon as possible.

Tracing for Kafka clients

Client applications, such as Kafka producers and consumers, can also be set up so that transactions are monitored. Clients are configured with a tracing profile, and a tracer is initialized for the client application to use.

9.5. Cruise Control

Cruise Control is an open source system that supports the following Kafka operations:

- Monitoring cluster workload
- Rebalancing a cluster based on predefined constraints

The operations help with running a more balanced Kafka cluster that uses broker pods more efficiently.

A typical cluster can become unevenly loaded over time. Partitions that handle large amounts of message traffic might not be evenly distributed across the available brokers. To rebalance the cluster, administrators must monitor the load on brokers and manually reassign busy partitions to brokers with spare capacity.

Cruise Control automates the cluster rebalancing process. It constructs a *workload model* of resource utilization for the cluster—based on CPU, disk, and network load—and generates optimization proposals (that you can approve or reject) for more balanced partition assignments. A set of configurable optimization goals is used to calculate these proposals.

You can generate optimization proposals in specific modes. The default `full` mode rebalances partitions across all brokers. You can also use the `add-brokers` and `remove-brokers` modes to accommodate changes when scaling a cluster up or down.

When you approve an optimization proposal, Cruise Control applies it to your Kafka cluster. You configure and generate optimization proposals using a `KafkaRebalance` resource. You can configure the resource using an annotation so that optimization proposals are approved automatically or

manually.

NOTE	Prometheus can extract Cruise Control metrics data, including data related to optimization proposals and rebalancing operations. A sample configuration file and Grafana dashboard for Cruise Control are provided with Strimzi.