

# Deploying and Managing Strimzi

# Table of Contents

1. Deployment overview .....	1
1.1. Strimzi custom resources .....	1
1.2. Strimzi operators .....	10
1.3. Using the HTTP Bridge to connect with a Kafka cluster .....	16
1.4. Seamless FIPS support .....	16
1.5. Document conventions .....	17
1.6. Additional resources .....	17
2. Using Kafka in KRaft mode .....	18
2.1. KRaft limitations .....	19
2.2. Migrating ZooKeeper-based Kafka clusters .....	20
3. Deployment methods .....	22
4. Deployment path .....	23
5. Downloading deployment files .....	24
6. Preparing for your deployment .....	25
6.1. Deployment prerequisites .....	25
6.2. Planning your Cluster Operator deployment .....	26
6.3. Pushing container images to your own registry .....	27
6.4. Designating Strimzi administrators .....	28
7. Deploying Strimzi using installation files .....	30
7.1. Deploying the Cluster Operator .....	30
7.2. Deploying Kafka .....	35
7.3. Deploying Kafka Connect .....	44
7.4. Adding Kafka Connect connectors .....	47
7.5. Deploying Kafka MirrorMaker .....	61
7.6. Deploying HTTP Bridge .....	63
7.7. Alternative standalone deployment options for Strimzi operators .....	66
8. Deploying Strimzi from OperatorHub.io .....	76
9. Deploying Strimzi using Helm .....	77
10. Feature gates .....	78
10.1. Feature gate releases .....	78
10.2. Graduated feature gates (GA) .....	79
10.3. Stable feature gates (Beta) .....	79
10.4. Early access feature gates (Alpha) .....	79
10.5. Enabling feature gates .....	80
11. Configuring a deployment .....	81
11.1. Using example configuration files .....	81
11.2. Configuring Kafka .....	83
11.3. Configuring node pools .....	95

11.4. Configuring Kafka storage .....	111
11.5. Configuring the Entity Operator .....	124
11.6. Configuring the Cluster Operator .....	127
11.7. Configuring Kafka Connect .....	141
11.8. Configuring Kafka Connect connectors .....	148
11.9. Configuring MirrorMaker 2 .....	158
11.10. Configuring MirrorMaker 2 connectors .....	176
11.11. Configuring the HTTP Bridge .....	183
11.12. Applying optional common configuration .....	186
11.13. Configuring logging .....	215
11.14. Restrictions on Kubernetes labels .....	223
12. Using the Topic Operator to manage Kafka topics .....	225
12.1. Topic management .....	225
12.2. Topic naming conventions .....	225
12.3. Handling changes to topics .....	226
12.4. Configuring Kafka topics .....	227
12.5. Configuring topics for replication and number of partitions .....	230
12.6. Managing KafkaTopic resources without impacting Kafka topics .....	230
12.7. Enabling topic management for existing Kafka topics .....	232
12.8. Deleting managed topics .....	233
12.9. Removing finalizers on topics .....	234
12.10. Considerations when disabling topic deletion .....	235
12.11. Tuning request batches for topic operations .....	235
13. Using the User Operator to manage Kafka users .....	237
13.1. Configuring Kafka users .....	237
13.2. Ignoring specific users .....	240
14. Using the Access Operator to manage client connections .....	242
14.1. Deploying the Access Operator .....	242
14.2. Using the Access Operator .....	243
15. Setting up client access to a Kafka cluster .....	246
15.1. Deploying example clients .....	246
15.2. Configuring listeners to connect to Kafka .....	247
15.3. Listener naming conventions .....	248
15.4. Accessing Kafka using node ports .....	249
15.5. Accessing Kafka using loadbalancers .....	252
15.6. Accessing Kafka using an Ingress NGINX Controller for Kubernetes .....	255
15.7. Accessing Kafka using OpenShift routes .....	258
15.8. Discovering connection details for clients .....	261
16. Securing access to a Kafka cluster .....	264
16.1. Configuring client authentication on listeners .....	264
16.2. Configuring authorized access to Kafka .....	273

16.3. Configuring user (client-side) security mechanisms .....	275
16.4. Example: Setting up secure client access .....	282
16.5. Troubleshooting TLS hostname verification with node ports .....	288
17. Configuring OAuth 2.0 token-based security .....	289
17.1. Migrating to the <b>custom</b> authentication type .....	289
17.2. Before you begin configuring OAuth 2.0 .....	289
17.3. Server-side configuration .....	290
17.4. Client-side configuration .....	294
17.5. Enabling authorization on Kafka brokers .....	297
17.6. Using Keycloak as an OAuth 2.0 provider .....	300
18. Managing TLS certificates .....	301
18.1. Internal cluster CA and clients CA .....	303
18.2. Secrets generated by the operators .....	303
18.3. Certificate renewal and validity periods .....	310
18.4. Configuring internal clients to trust the cluster CA .....	319
18.5. Configuring external clients to trust the cluster CA .....	321
18.6. Using your own CA certificates and private keys .....	322
19. Applying security context to Strimzi pods and containers .....	334
19.1. How to configure security context .....	334
19.2. Enabling the Restricted Provider for the Cluster Operator .....	337
19.3. Implementing a custom pod security provider .....	338
19.4. Handling of security context by Kubernetes platform .....	339
20. Scaling clusters by adding or removing brokers .....	340
20.1. Triggering auto-rebalances when scaling clusters .....	341
20.2. Skipping checks on scale-down operations .....	345
21. Using Cruise Control for cluster rebalancing .....	346
21.1. Cruise Control components and features .....	346
21.2. Deploying Cruise Control with Kafka .....	362
21.3. Generating optimization proposals .....	365
21.4. Approving optimization proposals .....	370
21.5. Tracking rebalances .....	371
21.6. Stopping rebalances .....	372
21.7. Troubleshooting and refreshing rebalances .....	373
22. Using Cruise Control to modify topic replication factor .....	375
23. Using Cruise Control to reassign partitions on JBOD disks .....	378
24. Using the partition reassignment tool .....	386
24.1. Partition reassignment tool overview .....	386
24.2. Generating a reassignment JSON file to reassign partitions .....	390
24.3. Using the partition reassignment tool to reassign partitions after adding brokers .....	395
24.4. Using the partition reassignment tool to reassign partitions before removing brokers .....	397
24.5. Using the partition reassignment tool to modify topic replication factor .....	400

25. Introducing metrics .....	404
25.1. Example metrics files .....	404
25.2. Using Prometheus with Strimzi .....	409
25.3. Enabling the example Grafana dashboards .....	417
25.4. Monitoring custom resources with kube-state-metrics .....	418
25.5. Consumer lag monitoring .....	419
25.6. Cruise Control operations monitoring .....	423
26. Introducing distributed tracing .....	425
26.1. Tracing options .....	425
26.2. Environment variables for tracing .....	426
26.3. Setting up distributed tracing .....	427
27. Evicting pods with the Strimzi Drain Cleaner .....	438
27.1. Default webhook configuration .....	438
27.2. Deploying the Strimzi Drain Cleaner using installation files .....	439
27.3. Deploying the Strimzi Drain Cleaner using Helm .....	441
27.4. Using the Strimzi Drain Cleaner .....	443
27.5. Adding or renewing the TLS certificates used by the Strimzi Drain Cleaner .....	444
27.6. Watching the TLS certificates used by the Strimzi Drain Cleaner .....	447
28. Managing rolling updates .....	449
28.1. Performing a rolling update using a pod management annotation .....	449
28.2. Performing a rolling update using a pod annotation .....	450
29. Finding information on Kafka restarts .....	452
29.1. Reasons for a restart event .....	452
29.2. Restart event filters .....	453
29.3. Checking Kafka restarts .....	454
30. Converting Strimzi custom resources to the v1 API .....	456
30.1. Conversion tool commands .....	456
30.2. Preparing for conversion .....	457
30.3. Converting custom resource YAML files .....	457
30.4. Converting custom resources in a Kubernetes cluster .....	458
30.5. Running the conversion tool as a Kubernetes job .....	459
30.6. Upgrading CRD storage version to v1 .....	462
30.7. Updating custom resources for the v1 API .....	463
31. Upgrading Strimzi .....	470
31.1. Required upgrade sequence .....	470
31.2. Upgrading Kubernetes with minimal downtime .....	471
31.3. Migrating to KRaft from versions earlier than 0.39 .....	472
31.4. Converting to v1 API from versions earlier than 0.49 .....	473
31.5. Upgrading the Cluster Operator .....	473
31.6. Upgrading Kafka clusters .....	477
31.7. Checking the status of an upgrade .....	479

31.8. Strimzi upgrade paths . . . . .	480
31.9. Strategies for upgrading clients . . . . .	481
32. Downgrading Strimzi . . . . .	482
32.1. Downgrading Kafka clusters and client applications . . . . .	482
32.2. Strimzi downgrade paths . . . . .	484
32.3. Downgrading the Cluster Operator . . . . .	485
33. Uninstalling Strimzi . . . . .	488
33.1. Uninstalling Strimzi using the CLI . . . . .	488
33.2. Uninstalling Strimzi from OperatorHub.io . . . . .	489
34. Cluster recovery from persistent volumes . . . . .	491
34.1. Cluster recovery scenarios . . . . .	491
34.2. Recovering a deleted Kafka cluster . . . . .	492
35. Tuning Kafka configuration . . . . .	497
35.1. Tools that help with tuning . . . . .	497
35.2. Managed broker configurations . . . . .	497
35.3. Kafka broker configuration tuning . . . . .	498
35.4. Kafka producer configuration tuning . . . . .	509
35.5. Kafka consumer configuration tuning . . . . .	515
35.6. Handling high volumes of messages . . . . .	524
35.7. Handling large message sizes . . . . .	529

# Chapter 1. Deployment overview

Strimzi simplifies the process of running [Apache Kafka](#) within a Kubernetes cluster.

This guide provides instructions for deploying and managing Strimzi. Deployment options and steps are covered using the example installation files included with Strimzi. While the guide highlights important configuration considerations, it does not cover all available options. For a deeper understanding of the Kafka component configuration options, refer to the [Strimzi Custom Resource API Reference](#).

In addition to deployment instructions, the guide offers pre- and post-deployment guidance. It covers setting up and securing client access to your Kafka cluster. Furthermore, it explores additional deployment options such as metrics integration, distributed tracing, and cluster management tools like Cruise Control and the Strimzi Drain Cleaner. You'll also find recommendations on managing Strimzi and fine-tuning Kafka configuration for optimal performance.

Upgrade instructions are provided for both Strimzi and Kafka, to help keep your deployment up to date.

Strimzi is designed to be compatible with all types of Kubernetes clusters, irrespective of their distribution. Whether your deployment involves public or private clouds, or if you are setting up a local development environment, the instructions in this guide are applicable in all cases.

## 1.1. Strimzi custom resources

The deployment of Kafka components onto a Kubernetes cluster using Strimzi is highly configurable through the use of custom resources. These resources are created as instances of APIs introduced by Custom Resource Definitions (CRDs), which 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.

### 1.1.1. Strimzi custom resource example

CRDs require a one-time installation in a cluster to define the schemas used to instantiate and manage Strimzi-specific 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.

Depending on the cluster setup, installation typically requires cluster admin privileges.

**NOTE** Access to manage custom resources is limited to Strimzi administrators. For more

information, see [Designating Strimzi administrators](#).

A CRD defines a new `kind` of resource, such as `kind: Kafka`, within a Kubernetes cluster.

The Kubernetes API server allows custom resources to be created based on the `kind` and understands from the CRD how to validate and store the custom resource when it is added to the Kubernetes cluster.

Each Strimzi-specific custom resource conforms to the schema defined by the CRD for the resource's `kind`. The custom resources for Strimzi components have common configuration properties, which are defined under `spec`.

To understand the relationship between a CRD and a custom resource, let's look at a sample of the CRD for a Kafka topic.

#### *Kafka topic CRD*

```
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata: ①
  name: kafkatopics.kafka.strimzi.io
  labels:
    app: strimzi
spec: ②
  group: kafka.strimzi.io
  versions:
    v1
  scope: Namespaced
  names:
    # ...
    singular: kafkatopic
    plural: kafkatopics
    shortNames:
      - kt ③
  additionalPrinterColumns: ④
    # ...
  subresources:
    status: {} ⑤
  validation: ⑥
  openAPIV3Schema:
    properties:
      spec:
        type: object
        properties:
          partitions:
            type: integer
            minimum: 1
          replicas:
            type: integer
            minimum: 1
            maximum: 32767
```

```
# ...
```

- ① The metadata for the topic CRD, its name and a label to identify the CRD.
- ② The specification for this CRD, including the group (domain) name, the plural name and the supported schema version, which are used in the URL to access the API of the topic. The other names are used to identify instance resources in the CLI. For example, `kubectl get kafkatopic my-topic` or `kubectl get kafkatopics`.
- ③ The shortname can be used in CLI commands. For example, `kubectl get kt` can be used as an abbreviation instead of `kubectl get kafkatopic`.
- ④ The information presented when using a `get` command on the custom resource.
- ⑤ The current status of the CRD as described in the [schema reference](#) for the resource.
- ⑥ openAPI3Schema validation provides validation for the creation of topic custom resources. For example, a topic requires at least one partition and one replica.

**NOTE**

You can identify the CRD YAML files supplied with the Strimzi installation files, because the file names contain an index number followed by 'Crd'.

Here is a corresponding example of a `KafkaTopic` custom resource.

*Kafka topic custom resource*

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaTopic ①
metadata:
  name: my-topic
  labels:
    strimzi.io/cluster: my-cluster ②
spec: ③
  partitions: 1
  replicas: 1
  config:
    retention.ms: 7200000
    segment.bytes: 1073741824
status:
  conditions: ④
    lastTransitionTime: "2019-08-20T11:37:00.706Z"
    status: "True"
    type: Ready
  observedGeneration: 1
/ ...
```

- ① The `kind` and `apiVersion` identify the CRD of which the custom resource is an instance.
- ② A label, applicable only to `KafkaTopic` and `KafkaUser` resources, that defines the name of the Kafka cluster (which is same as the name of the `Kafka` resource) to which a topic or user belongs.
- ③ The spec shows the number of partitions and replicas for the topic as well as the configuration parameters for the topic itself. In this example, the retention period for a message to remain in

the topic and the segment file size for the log are specified.

- ④ Status conditions for the `KafkaTopic` resource. The `type` condition changed to `Ready` at the `lastTransitionTime`.

Custom resources can be applied to a cluster through the platform CLI. When the custom resource is created, it uses the same validation as the built-in resources of the Kubernetes API.

After a `KafkaTopic` custom resource is created, the Topic Operator is notified and corresponding Kafka topics are created in Strimzi.

#### *Additional resources*

- [Extend the Kubernetes API with CustomResourceDefinitions](#)
- [Example configuration files provided with Strimzi](#)

### 1.1.2. Performing `kubectl` operations on custom resources

You can use `kubectl` commands to retrieve information and perform other operations on Strimzi custom resources. Use `kubectl` commands, such as `get`, `describe`, `edit`, or `delete`, to perform operations on resource types. For example, `kubectl get kafkatopics` retrieves a list of all Kafka topics and `kubectl get kafkas` retrieves all deployed Kafka clusters.

When referencing resource types, you can use both singular and plural names: `kubectl get kafkas` gets the same results as `kubectl get kafka`.

You can also use the *short name* of the resource. Learning short names can save you time when managing Strimzi. The short name for `Kafka` is `k`, so you can also run `kubectl get k` to list all Kafka clusters.

#### *Listing Kafka clusters*

```
kubectl get k

NAME      READY   METADATA STATE   WARNINGS
my-cluster  True    KRaft
```

Table 1. Long and short names for each Strimzi resource

Strimzi resource	Long name	Short name
Kafka	kafka	k
Kafka Node Pool	kafkanodepool	knp
Kafka Topic	kafkatopic	kt
Kafka User	kafkauser	ku
Kafka Connect	kafkaconnect	kc
Kafka Connector	kafkaconnector	kctr
Kafka MirrorMaker 2	kafkamirrormaker2	kmm2

Strimzi resource	Long name	Short name
HTTP Bridge	kafkabridge	kb
Kafka Rebalance	kafkarebalance	kr
Strimzi Pod Set	strimzipodset	sps

## Resource categories

Categories of custom resources can also be used in `kubectl` commands.

All Strimzi custom resources belong to the category `strimzi`, so you can use `strimzi` to get all the Strimzi resources with one command.

For example, running `kubectl get strimzi` lists all Strimzi custom resources in a given namespace.

### *Listing all custom resources*

```
kubectl get strimzi

NAME                                     PODS   READY   PODS   CURRENT
PODS   AGE
strimzipodset.core.strimzi.io/my-cluster-brokers    3      3      3
6h11m
strimzipodset.core.strimzi.io/my-cluster-controllers 3      3      3
6h11m

NAME                                     DESIRED   REPLICAS   ROLES
NODEIDS
kafkanodepool.kafka.strimzi.io/brokers      3          ["broker"]
[3,4,5]
kafkanodepool.kafka.strimzi.io/controllers   3          ["controller"]
[0,1,2]

NAME                                     READY   METADATA STATE   WARNINGS
kafka.kafka.strimzi.io/my-cluster        True    KRaft

NAME                                     PARTITIONS   REPLICATION FACTOR
kafkatopic.kafka.strimzi.io/kafka-apps  3            3

NAME                                     AUTHENTICATION   AUTHORIZATION
kafkauser.kafka.strimzi.io/my-user       tls           simple
```

The `kubectl get strimzi -o name` command returns all resource types and resource names. The `-o name` option fetches the output in the *type/name* format

### *Listing all resource types and names*

```
kubectl get strimzi -o name

strimzipodset.core.strimzi.io/my-cluster-brokers
```

```
strimzipodset.core.strimzi.io/my-cluster-controllers
kafkanodepool.kafka.strimzi.io/brokers
kafkanodepool.kafka.strimzi.io/controllers
kafka.kafka.strimzi.io/my-cluster
kafkatopic.kafka.strimzi.io/kafka-apps
kafkauser.kafka.strimzi.io/my-user
```

You can combine this `strimzi` command with other commands. For example, you can pass it into a `kubectl delete` command to delete all resources in a single command.

#### *Deleting all custom resources*

```
kubectl delete $(kubectl get strimzi -o name)

strimzipodset.core.strimzi.io "my-cluster-brokers" deleted
strimzipodset.core.strimzi.io "my-cluster-controllers" deleted
kafkanodepool.kafka.strimzi.io "brokers" deleted
kafkanodepool.kafka.strimzi.io "controllers" deleted
kafka.kafka.strimzi.io "my-cluster" deleted
kafkatopic.kafka.strimzi.io "kafka-apps" deleted
kafkauser.kafka.strimzi.io "my-user" deleted
```

Deleting all resources in a single operation might be useful, for example, when you are testing new Strimzi features.

#### **Querying the status of sub-resources**

There are other values you can pass to the `-o` option. For example, by using `-o yaml` you get the output in YAML format. Using `-o json` will return it as JSON.

You can see all the options in `kubectl get --help`.

One of the most useful options is the [JSONPath support](#), which allows you to pass JSONPath expressions to query the Kubernetes API. A JSONPath expression can extract or navigate specific parts of any resource.

For example, you can use the JSONPath expression `{.status.listeners[?(@.name=="tls")].bootstrapServers}` to get the bootstrap address from the status of the Kafka custom resource and use it in your Kafka clients.

Here, the command retrieves the `bootstrapServers` value of the listener named `tls`:

#### *Retrieving the bootstrap address*

```
kubectl get kafka my-cluster
-o=jsonpath='{.status.listeners[?(@.name=="tls")].bootstrapServers}{\n}'  
my-cluster-kafka-bootstrap.myproject.svc:9093
```

By changing the name condition you can also get the address of the other Kafka listeners.

You can use [jsonpath](#) to extract any other property or group of properties from any custom resource.

### 1.1.3. Strimzi custom resource status information

Status properties provide status information for certain custom resources.

The following table lists the custom resources that provide status information (when deployed) and the schemas that define the status properties.

For more information on the schemas, see the [Strimzi Custom Resource API Reference](#).

*Table 2. Custom resources that provide status information*

Strimzi resource	Schema reference	Publishes status information on...
Kafka	KafkaStatus ListenerStatus UsedNodePoolStatus KafkaAutoRebalanceStatus	The Kafka cluster, its listeners, node pools, and any auto-rebalances on scaling
KafkaNodePool	KafkaNodePoolStatus	The nodes in the node pool, their roles, and the associated Kafka cluster
KafkaTopic	KafkaTopicStatus	Kafka topics in the Kafka cluster
KafkaUser	KafkaUserStatus	Kafka users in the Kafka cluster
KafkaConnect	KafkaConnectStatus	The Kafka Connect cluster and connector plugins
KafkaConnector	KafkaConnectorStatus	KafkaConnector resources
KafkaMirrorMaker2	KafkaMirrorMaker2Status	The Kafka MirrorMaker 2 cluster and internal connectors
KafkaBridge	KafkaBridgeStatus	The HTTP Bridge
KafkaRebalance	KafkaRebalanceStatus	The status and results of a rebalance
StrimziPodSet	StrimziPodSetStatus	The number of pods: being managed, using the current version, and in a ready state

The `status` property of a resource provides information on the state of the resource. The `status.conditions` and `status.observedGeneration` properties are common to all resources.

#### `status.conditions`

Status conditions describe the *current state* of a resource. Status condition properties are useful for tracking progress related to the resource achieving its *desired state*, as defined by the

configuration specified in its `spec`. Status condition properties provide the time and reason the state of the resource changed, and details of events preventing or delaying the operator from realizing the desired state.

### `status.observedGeneration`

Last observed generation denotes the latest reconciliation of the resource by the Cluster Operator. If the value of `observedGeneration` is different from the value of `metadata.generation` (the current version of the deployment), the operator has not yet processed the latest update to the resource. If these values are the same, the status information reflects the most recent changes to the resource.

The `status` properties also provide resource-specific information. For example, `KafkaStatus` provides information on listener addresses, and the ID of the Kafka cluster.

`KafkaStatus` also provides information on the Kafka and Strimzi versions being used. You can check the values of `operatorLastSuccessfulVersion` and `kafkaVersion` to determine whether an upgrade of Strimzi or Kafka has completed

Strimzi creates and maintains the status of custom resources, periodically evaluating the current state of the custom resource and updating its status accordingly. When performing an update on a custom resource using `kubectl edit`, for example, its `status` is not editable. Moreover, changing the `status` would not affect the configuration of the Kafka cluster.

Here we see the `status` properties for a `Kafka` custom resource.

#### *Kafka custom resource status*

```
apiVersion: kafka.strimzi.io/v1
kind: Kafka
metadata:
spec:
  # ...
status:
  clusterId: XP9FP2P-RByvEy0W4cOEUA ①
  conditions: ②
    - lastTransitionTime: '2023-01-20T17:56:29.396588Z'
      status: 'True'
      type: Ready ③
  kafkaMetadataState: KRaft ④
  kafkaVersion: 4.1.1 ⑤
  kafkaNodePools: ⑥
    - name: broker
    - name: controller
  listeners: ⑦
    - addresses:
        - host: my-cluster-kafka-bootstrap.prm-project.svc
          port: 9092
  bootstrapServers: 'my-cluster-kafka-bootstrap.prm-project.svc:9092'
  name: plain
  - addresses:
      - host: my-cluster-kafka-bootstrap.prm-project.svc
```

```

    port: 9093
  bootstrapServers: 'my-cluster-kafka-bootstrap.prm-project.svc:9093'
  certificates:
    - |
      -----BEGIN CERTIFICATE-----
      -----END CERTIFICATE-----
    name: tls
    - addresses:
      - host: >-
        2054284155.us-east-2.elb.amazonaws.com
        port: 9095
      bootstrapServers: >-
        2054284155.us-east-2.elb.amazonaws.com:9095
      certificates:
        - |
          -----BEGIN CERTIFICATE-----
          -----END CERTIFICATE-----
        name: external3
        - addresses:
          - host: ip-10-0-172-202.us-east-2.compute.internal
            port: 31644
        bootstrapServers: 'ip-10-0-172-202.us-east-2.compute.internal:31644'
        certificates:
          - |
            -----BEGIN CERTIFICATE-----
            -----END CERTIFICATE-----
        name: external4
      observedGeneration: 3 ⑧
      operatorLastSuccessfulVersion: 0.50.0 ⑨

```

- ① The Kafka cluster ID.
- ② Status **conditions** describe the current state of the Kafka cluster.
- ③ The **Ready** condition indicates that the Cluster Operator considers the Kafka cluster able to handle traffic.
- ④ Kafka metadata state that shows KRaft is managing Kafka metadata and coordinating operations.
- ⑤ The version of Kafka being used by the Kafka cluster.
- ⑥ The node pools belonging to the Kafka cluster.
- ⑦ The **listeners** describe Kafka bootstrap addresses by type.
- ⑧ The **observedGeneration** value indicates the last reconciliation of the **Kafka** custom resource by the Cluster Operator.
- ⑨ The version of the operator that successfully completed the last reconciliation.

**NOTE** The Kafka bootstrap addresses listed in the status do not signify that those

endpoints or the Kafka cluster is in a [Ready](#) state.

#### 1.1.4. Finding the status of a custom resource

Use `kubectl` with the `status` subresource of a custom resource to retrieve information about the resource.

##### *Prerequisites*

- A Kubernetes cluster.
- The Cluster Operator is running.

##### *Procedure*

- Specify the custom resource and use the `-o jsonpath` option to apply a standard JSONPath expression to select the `status` property:

```
kubectl get kafka <kafka_resource_name> -o jsonpath='{.status}' | jq
```

This expression returns all the status information for the specified custom resource. You can use dot notation, such as `status.listeners` or `status.observedGeneration`, to fine-tune the status information you wish to see.

Using the [jq command line JSON parser tool](#) makes it easier to read the output.

##### *Additional resources*

- For more information about using JSONPath, see [JSONPath support](#).

## 1.2. Strimzi operators

Strimzi uses operators to deploy and manage Kafka components. The operators continuously monitor Strimzi custom resources (like `Kafka`, `KafkaTopic`, and `KafkaUser`) and reconcile the state of Kafka components to match their configuration.

This reconciliation process involves three main operations:

### **Creation**

When you create a Strimzi custom resource, the responsible operator detects it and takes the necessary actions to create the component. This might involve creating Kubernetes resources, such as `Deployment`, `Pod`, `Service`, and `ConfigMap`, or configuring items inside the Kafka cluster itself, such as topics and users.

### **Update**

Each time you update a custom resource, the operator detects the change and applies a corresponding update if the changes are valid. This could trigger a rolling update of pods or reconfigure a resource within Kafka. Rolling updates maintain the availability of the Kafka cluster, but can lead to service disruption in the Kafka clients.

## Deletion

When you delete a custom resource, the operator detects the deletion and acts to remove the component. Most dependent resources are deleted automatically by Kubernetes garbage collection. The exact behavior depends on the resource type. For a Kafka cluster, PVCs are retained by default to prevent data loss. For a Kafka topic, the topic is fully deleted from the Apache Kafka cluster.

Strimzi provides the following operators, each responsible for different aspects of a Kafka deployment:

### Cluster Operator (required)

The Cluster Operator is the core operator and must be deployed first. It handles the deployment and lifecycle of Apache Kafka clusters on Kubernetes, automating the setup of Kafka nodes and related resources.

Additionally, Strimzi provides **Drain Cleaner**, which is deployed separately. Drain Cleaner supports the Cluster Operator in managing pod evictions for Kafka clusters.

### Entity Operator (recommended)

The Entity Operator can be deployed by the Cluster Operator. It runs in a single pod and includes one or both of the following operators:

- **Topic Operator** to manage Kafka topics.
- **User Operator** to manage Kafka users.

Each operator runs in a separate container within the Entity Operator pod.

### Access Operator (optional)

Manages and shares Kafka cluster connection details. It is deployed independently of the Cluster Operator.

**NOTE** The Topic Operator and User Operator can also be deployed standalone (without the Entity Operator) to manage topics and users for a Kafka cluster that is not managed by Strimzi.

### 1.2.1. Operator-watched Kafka resources

Operators watch and manage Kafka resources within defined Kubernetes namespaces. The namespace scope of where each operator can watch these resources differs.

You can choose the namespace scope for the Cluster Operator. The Topic Operator and the User Operator can only watch a single Kafka cluster in a namespace. And they can only be connected to a single Kafka cluster.

**WARNING** While the operator can be configured to watch multiple namespaces, each watched namespace should contain only one instance of a specific component type, such as one Kafka cluster, to avoid conflicts.

Table 3. Operator resource and scope

Operator	Watched resources	Namespace scope
<b>Cluster Operator</b>	<code>Kafka</code> <code>KafkaNodePool</code> <code>KafkaConnect</code> <code>KafkaConnector</code> <code>KafkaMirrorMaker2</code> <code>KafkaBridge</code> <code>KafkaRebalance</code>	Single, multiple, or all
<b>Topic Operator</b>	<code>KafkaTopic</code>	Single namespace (one Kafka cluster only)
<b>User Operator</b>	<code>KafkaUser</code>	Single namespace (one Kafka cluster only)
<b>Access Operator</b>	<code>KafkaAccess</code>	Single or all

**NOTE**

For a standalone deployment of the Topic Operator or User Operator, you specify a namespace and connection to the Kafka cluster to watch in the configuration.

## 1.2.2. Managing RBAC resources

The Cluster Operator creates and manages role-based access control (RBAC) resources for Strimzi components that need access to Kubernetes resources.

For the Cluster Operator to function, it needs permission within the Kubernetes cluster to interact with Kafka resources, such as `Kafka` and `KafkaConnect`, as well as managed resources like `ConfigMap`, `Pod`, `Deployment`, and `Service`.

Permission is specified through the following Kubernetes RBAC resources:

- `ServiceAccount`
- `Role` and `ClusterRole`
- `RoleBinding` and `ClusterRoleBinding`

### Delegating privileges to Strimzi components

The Cluster Operator runs under a service account called `strimzi-cluster-operator`, which is assigned cluster roles that give it permission to create the necessary RBAC resources for Strimzi components. Role bindings associate the cluster roles with the service account.

Kubernetes enforces [privilege escalation prevention](#), meaning the Cluster Operator cannot grant privileges it does not possess, nor can it grant such privileges in a namespace it cannot access. Consequently, the Cluster Operator must have the necessary privileges for all the components it orchestrates.

The Cluster Operator must be able to do the following:

- Enable the Topic Operator to manage `KafkaTopic` resources by creating `Role` and `RoleBinding` resources in the relevant namespace.

- Enable the User Operator to manage `KafkaUser` resources by creating `Role` and `RoleBinding` resources in the relevant namespace.
- Allow Strimzi to discover the failure domain of a `Node` by creating a `ClusterRoleBinding`.

When using rack-aware partition assignment, broker pods need to access information about the `Node` they are running on, such as the Availability Zone in Amazon AWS. Similarly, when using `NodePort` type listeners, broker pods need to advertise the address of the `Node` they are running on. Since a `Node` is a cluster-scoped resource, this access must be granted through a `ClusterRoleBinding`, not a namespace-scoped `RoleBinding`.

The following sections describe the RBAC resources required by the Cluster Operator.

### `ClusterRole` resources

The Cluster Operator uses `ClusterRole` resources to provide the necessary access to resources. Depending on the Kubernetes cluster setup, a cluster administrator might be needed to create the cluster roles.

**NOTE**

Cluster administrator rights are only needed for the creation of `ClusterRole` resources. The Cluster Operator will not run under a cluster admin account.

The RBAC resources follow the *principle of least privilege* and contain only those privileges needed by the Cluster Operator to operate the cluster of the Kafka component.

All cluster roles are required by the Cluster Operator in order to delegate privileges.

*Table 4. ClusterRole resources*

Name	Description
<code>strimzi-cluster-operator-namespaced</code>	Access rights for namespace-scoped resources used by the Cluster Operator to deploy and manage the operands.
<code>strimzi-cluster-operator-global</code>	Access rights for cluster-scoped resources used by the Cluster Operator to deploy and manage the operands.
<code>strimzi-cluster-operator-leader-election</code>	Access rights used by the Cluster Operator for leader election.
<code>strimzi-cluster-operator-watched</code>	Access rights used by the Cluster Operator to watch and manage the Strimzi custom resources.
<code>strimzi-kafka-broker</code>	Access rights to allow Kafka brokers to get the topology labels from Kubernetes worker nodes when rack-awareness is used.
<code>strimzi-entity-operator</code>	Access rights used by the Topic and User Operators to manage Kafka users and topics.

Name	Description
<code>strimzi-kafka-client</code>	Access rights to allow Kafka Connect, MirrorMaker (1 and 2), and HTTP Bridge to get the topology labels from Kubernetes worker nodes when rack-awareness is used.

### ClusterRoleBinding resources

The Cluster Operator uses `ClusterRoleBinding` and `RoleBinding` resources to associate its `ClusterRole` with its `ServiceAccount`. Cluster role bindings are required by cluster roles containing cluster-scoped resources.

*Table 5. ClusterRoleBinding resources*

Name	Description
<code>strimzi-cluster-operator</code>	Grants the Cluster Operator the rights from the <code>strimzi-cluster-operator-global</code> cluster role.
<code>strimzi-cluster-operator-kafka-broker-delegation</code>	Grants the Cluster Operator the rights from the <code>strimzi-entity-operator</code> cluster role.
<code>strimzi-cluster-operator-kafka-client-delegation</code>	Grants the Cluster Operator the rights from the <code>strimzi-kafka-client</code> cluster role.

*Table 6. RoleBinding resources*

Name	Description
<code>strimzi-cluster-operator</code>	Grants the Cluster Operator the rights from the <code>strimzi-cluster-operator-namespaced</code> cluster role.
<code>strimzi-cluster-operator-leader-election</code>	Grants the Cluster Operator the rights from the <code>strimzi-cluster-operator-leader-election</code> cluster role.
<code>strimzi-cluster-operator-watched</code>	Grants the Cluster Operator the rights from the <code>strimzi-cluster-operator-watched</code> cluster role.
<code>strimzi-cluster-operator-entity-operator-delegation</code>	Grants the Cluster Operator the rights from the <code>strimzi-cluster-operator-entity-operator-delegation</code> cluster role.

### ServiceAccount resources

The Cluster Operator runs using the `strimzi-cluster-operator ServiceAccount`. This service account grants it the privileges it requires to manage the operands. The Cluster Operator creates additional `ClusterRoleBinding` and `RoleBinding` resources to delegate some of these RBAC rights to the operands.

Each of the operands uses its own service account created by the Cluster Operator. This allows the Cluster Operator to follow the principle of least privilege and give the operands only the access rights that are really needed.

Table 7. ServiceAccount resources

Name	Used by
<cluster_name>-kafka	Kafka broker pods
<cluster_name>-entity-operator	Entity Operator
<cluster_name>-cruise-control	Cruise Control pods
<cluster_name>-kafka-exporter	Kafka Exporter pods
<cluster_name>-connect	Kafka Connect pods
<cluster_name>-mirror-maker	MirrorMaker pods
<cluster_name>-mirrormaker2	MirrorMaker 2 pods
<cluster_name>-bridge	HTTP Bridge pods

### 1.2.3. Managing pod resources

The `StrimziPodSet` custom resource is used by Strimzi to create and manage Kafka, Kafka Connect, and MirrorMaker 2 pods.

You must not create, update, or delete `StrimziPodSet` resources. The `StrimziPodSet` custom resource is used internally and resources are managed solely by the Cluster Operator. As a consequence, the Cluster Operator must be running properly to avoid the possibility of pods not starting and Kafka clusters not being available.

**NOTE**

Kubernetes `Deployment` resources are used for creating and managing the pods of other components.

### 1.2.4. Lock acquisition warnings for cluster operations

The Cluster Operator ensures that only one operation runs at a time for each cluster by using locks. If another operation attempts to start while a lock is held, it waits until the current operation completes.

Operations such as cluster creation, rolling updates, scaling down, and scaling up are managed by the Cluster Operator.

If acquiring a lock takes longer than the configured timeout (`STRIMZI_OPERATION_TIMEOUT_MS`), a DEBUG message is logged:

*Example DEBUG message for lock acquisition*

```
DEBUG AbstractOperator:406 - Reconciliation #55(timer) Kafka(myproject/my-cluster):
Failed to acquire lock lock::myproject::Kafka::my-cluster within 10000ms.
```

Timed-out operations are retried during the next periodic reconciliation in intervals defined by `STRIMZI_FULL_RECONCILIATION_INTERVAL_MS` (by default 120 seconds).

If an INFO message continues to appear with the same reconciliation number, it might

indicate a lock release error:

*Example INFO message for reconciliation*

```
INFO AbstractOperator:399 - Reconciliation #1(watch) Kafka(myproject/my-cluster):  
Reconciliation is in progress
```

Restarting the Cluster Operator can resolve such issues.

## 1.3. Using the HTTP Bridge to connect with a Kafka cluster

You can use the HTTP Bridge API to create and manage consumers and send and receive records over HTTP rather than the native Kafka protocol.

When you set up the HTTP Bridge you configure HTTP access to the Kafka cluster. You can then use the HTTP Bridge to produce and consume messages from the cluster, as well as performing other operations through its REST interface.

*Additional resources*

- For information on installing and using the HTTP Bridge, see [Using the HTTP Bridge](#).

## 1.4. Seamless FIPS support

Federal Information Processing Standards (FIPS) are standards for computer security and interoperability. When running Strimzi on a FIPS-enabled Kubernetes cluster, the OpenJDK used in Strimzi container images automatically switches to FIPS mode. From version 0.33, Strimzi can run on FIPS-enabled Kubernetes clusters without any changes or special configuration. It uses only the FIPS-compliant security libraries from the OpenJDK.

**IMPORTANT**

If you are using FIPS-enabled Kubernetes clusters, you may experience higher memory consumption compared to regular Kubernetes clusters. To avoid any issues, we suggest increasing the memory request to at least 512Mi.

### 1.4.1. NIST validation

Strimzi is designed to use FIPS-validated cryptographic libraries for secure communication in a FIPS-enabled Kubernetes cluster. However, it's important to note that while Strimzi can leverage these libraries in a FIPS environment, the underlying Universal Base Images (UBI) used in Strimzi deployments may not inherently include NIST-validated binaries. This means that while Strimzi can leverage cryptographic libraries for FIPS, the specific binaries within the Strimzi container images might not have undergone NIST validation.

For more information about the NIST validation program and validated modules, see [Cryptographic Module Validation Program](#) on the NIST website.

## 1.4.2. Minimum password length

When running in the FIPS mode, SCRAM-SHA-512 passwords need to be at least 32 characters long. From Strimzi 0.33, the default password length in Strimzi User Operator is set to 32 characters as well. If you have a Kafka cluster with custom configuration that uses a password length that is less than 32 characters, you need to update your configuration. If you have any users with passwords shorter than 32 characters, you need to regenerate a password with the required length. You can do that, for example, by deleting the user secret and waiting for the User Operator to create a new password with the appropriate length.

### Additional resources

- [Disabling FIPS mode using Cluster Operator configuration](#)
- [What are Federal Information Processing Standards \(FIPS\)](#)

## 1.5. Document conventions

User-replaced values, also known as *replaceables*, are shown in with angle brackets (< >). Underscores ( \_ ) are used for multi-word values. If the value refers to code or commands, `monospace` is also used.

For example, the following code shows that `<my_namespace>` must be replaced by the correct namespace name:

```
sed -i 's/namespace: .*/namespace: <my_namespace>/' install/cluster-operator/*RoleBinding*.yaml
```

## 1.6. Additional resources

- [Strimzi Overview](#)
- [Strimzi Custom Resource API Reference](#)
- [Using the HTTP Bridge](#)

# Chapter 2. Using Kafka in KRaft mode

KRaft (Kafka Raft metadata) mode replaces Kafka's dependency on ZooKeeper for cluster management. KRaft mode simplifies the deployment and management of Kafka clusters by bringing metadata management and coordination of clusters into Kafka.

Kafka in KRaft mode is designed to offer enhanced reliability, scalability, and throughput. Metadata operations become more efficient as they are directly integrated. And by removing the need to maintain a ZooKeeper cluster, there's also a reduction in the operational and security overhead.

To deploy a Kafka cluster in KRaft mode, you must use [Kafka](#) and [KafkaNodePool](#) custom resources. For more details and examples, see [Deploying a Kafka cluster](#).

Through [node pool configuration using KafkaNodePool resources](#), nodes are assigned the role of broker, controller, or both:

- **Controller** nodes operate in the control plane to manage cluster metadata and the state of the cluster using a Raft-based consensus protocol.
- **Broker** nodes operate in the data plane to manage the streaming of messages, receiving and storing data in topic partitions.
- **Dual-role** nodes fulfill the responsibilities of controllers and brokers.

Controllers use a metadata log, stored as a single-partition topic (`__cluster_metadata`) on every node, which records the state of the cluster. When requests are made to change the cluster configuration, an active (lead) controller manages updates to the metadata log, and follower controllers replicate these updates. The metadata log stores information on brokers, replicas, topics, and partitions, including the state of in-sync replicas and partition leadership. Kafka uses this metadata to coordinate changes and manage the cluster effectively.

Broker nodes act as observers, storing the metadata log passively to stay up-to-date with the cluster's state. Each node fetches updates to the log independently. If you are using JBOD storage, you can [change the volume that stores the metadata log](#).

**NOTE** The KRaft metadata version used in the Kafka cluster must be supported by the Kafka version in use. Both versions are managed through the [Kafka](#) resource configuration. For more information, see [Configuring Kafka](#).

In the following example, a Kafka cluster comprises a quorum of controller and broker nodes for fault tolerance and high availability.

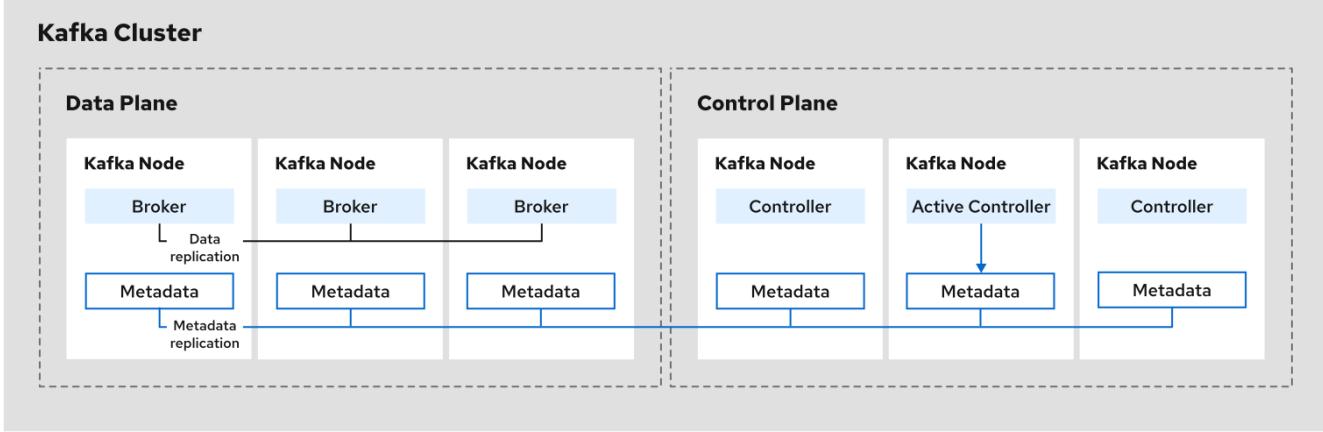


Figure 1. Example cluster with separate broker and controller nodes

In a typical production environment, use dedicated broker and controller nodes. However, you might want to use nodes in a dual-role configuration for development or testing.

You can use a combination of nodes that combine roles with nodes that perform a single role. In the following example, three nodes perform a dual role and two nodes act only as brokers.

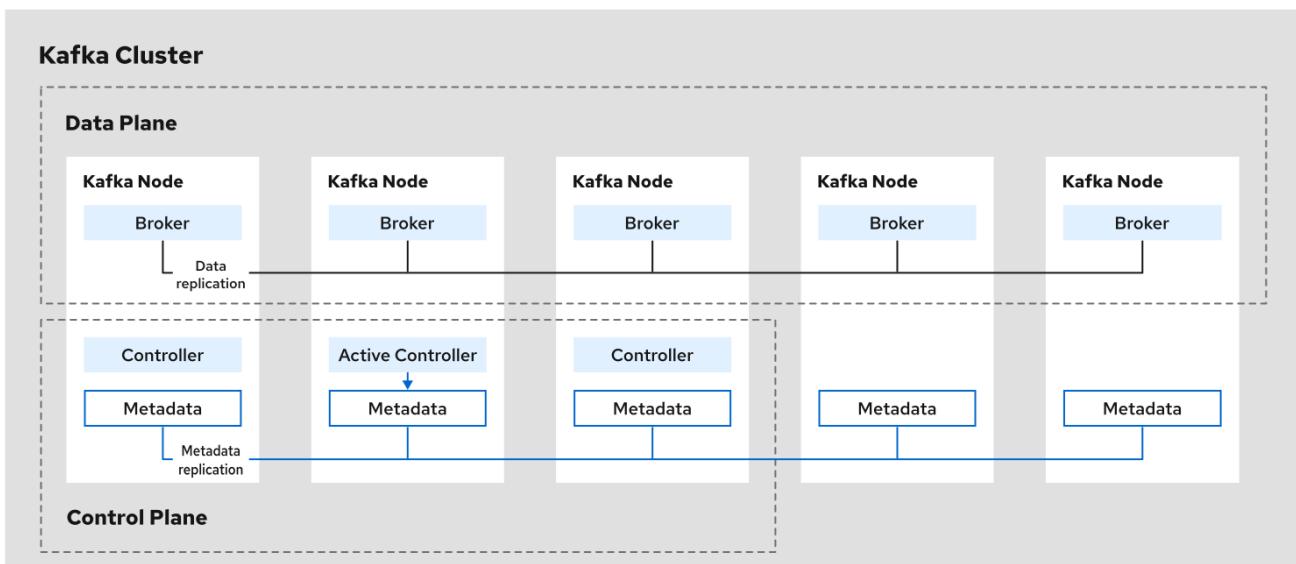


Figure 2. Example cluster with dual-role nodes and dedicated broker nodes

## 2.1. KRaft limitations

KRaft limitations primarily relate to controller scaling, which impacts cluster operations.

### 2.1.1. Controller scaling

KRaft mode supports two types of controller quorums:

- **Static controller quorums**

In this mode, the number of controllers is fixed, and scaling requires downtime.

- **Dynamic controller quorums**

This mode enables dynamic scaling of controllers without downtime. New controllers join as

observers, replicate the metadata log, and eventually become eligible to join the quorum. If a controller being removed is the active controller, it will step down from the quorum only after the new quorum is confirmed.

Scaling is useful not only for adding or removing controllers, but supports the following operations:

- Renaming a node pool, which involves adding a new node pool with the desired name and deleting the old one.
- Changing non-JBOD storage, which requires creating a new node pool with the updated storage configuration and removing the old one.

Dynamic controller quorums provide the flexibility to make these operations significantly easier to perform.

### 2.1.2. Limitations with static controller quorums

Migration between static and dynamic controller quorums is not currently supported by Apache Kafka, though it is expected to be introduced in a future release. As a result, Strimzi uses static controller quorums for all deployments, including new installations. All pre-existing KRaft-based Apache Kafka clusters that use static controller quorums must continue using them. To ensure compatibility with existing KRaft-based clusters, Strimzi continues to use static controller quorums as well.

This limitation means dynamic scaling of controller quorums cannot be used to support the following:

- Adding or removing node pools with controller roles
- Adding the controller role to an existing node pool
- Removing the controller role from an existing node pool
- Scaling a node pool with the controller role
- Renaming a node pool with the controller role

Static controller quorums also limit operations that require scaling. For example, changing the storage type for a node pool with a controller role is not possible because it involves scaling the controller quorum. For non-JBOD storage, creating a new node pool with the desired storage type, adding it to the cluster, and removing the old one would require scaling operations, which are not supported. In some cases, workarounds are possible. For instance, when modifying node pool roles to combine controller and broker functions, you can [add broker roles to controller nodes](#) instead of adding controller roles to broker nodes to avoid controller scaling. However, this approach would require reassigning more data, which may temporarily affect cluster performance.

Once migration is possible, Strimzi plans to assess introducing support for dynamic quorums.

## 2.2. Migrating ZooKeeper-based Kafka clusters

Kafka 4.0 runs exclusively in KRaft mode, with no ZooKeeper integration. As a result of this change, Strimzi removed support for ZooKeeper-based Kafka clusters starting with version 0.46.

To upgrade to Strimzi 0.46 or later, first migrate any ZooKeeper-based Kafka clusters to KRaft mode.

**NOTE:** To perform the migration before upgrading, follow the procedure outlined in the Strimzi 0.45.x documentation. For more information, see [Migrating to KRaft mode](#).

# Chapter 3. Deployment methods

You can deploy Strimzi on Kubernetes 1.27 and later using one of the following methods:

Installation method	Description
<a href="#">Deployment files (YAML files)</a>	<a href="#">Download the deployment files</a> to manually deploy Strimzi components. The installation files bundle all the necessary Kubernetes resources, including the Custom Resource Definitions (CRDs) that define resources like <a href="#">Kafka</a> and <a href="#">KafkaTopic</a> , the RBAC roles for permissions, and the <a href="#">Deployment</a> for the Cluster Operator itself. This method offers the greatest flexibility and control.
<a href="#">OperatorHub.io</a>	Install Strimzi using the Operator Lifecycle Manager (OLM) from OperatorHub.io. Once the Cluster Operator is running, you can deploy Strimzi components using custom resources. This method provides a standard configuration and supports automatic updates for streamlined lifecycle management.
<a href="#">Helm chart</a>	Use a Helm chart to deploy the Cluster Operator, then deploy Strimzi components using custom resources. Helm charts offer a convenient and repeatable way to manage installations, especially in environments already using Helm.

**NOTE**

All deployment methods assume that you have access to a running Kubernetes cluster with appropriate permissions. Some methods may also require additional setup, such as access to container registries.

# Chapter 4. Deployment path

You can configure a deployment where Strimzi manages a single Kafka cluster in the same namespace, suitable for development or testing. Alternatively, Strimzi can manage multiple Kafka clusters across different namespaces in a production environment.

The basic deployment path includes the following steps:

1. Create a Kubernetes namespace for the Cluster Operator.
2. Deploy the Cluster Operator based on your chosen deployment method.
3. Deploy the Kafka cluster, including the Topic Operator and User Operator if desired.
4. Optionally, deploy additional components:
  - The Topic Operator and User Operator as standalone components, if not deployed with the Kafka cluster
  - Kafka Connect
  - Kafka MirrorMaker
  - HTTP Bridge
  - Metrics monitoring components

The Cluster Operator creates Kubernetes resources such as [Deployment](#), [Service](#), and [Pod](#) for each component. The resource names are appended with the name of the deployed component. For example, a Kafka cluster named [my-kafka-cluster](#) will have a service named [my-kafka-cluster-kafka](#).

# Chapter 5. Downloading deployment files

To deploy Strimzi components using YAML files, download and extract the latest release archive ([strimzi-0.50.0.\\*](#)) from the [GitHub releases page](#).

The release archive contains sample YAML files for deploying Strimzi components to Kubernetes using `kubectl`.

Begin by deploying the Cluster Operator from the `install/cluster-operator` directory to watch a single namespace, multiple namespaces, or all namespaces.

In the `install` folder, you can also deploy other Strimzi components, including:

- Strimzi administrator roles (`strimzi-admin`)
- Standalone Topic Operator (`topic-operator`)
- Standalone User Operator (`user-operator`)
- Strimzi Drain Cleaner (`drain-cleaner`)

The `examples` folder [provides examples of Strimzi custom resources](#) to help you develop your own Kafka configurations.

**NOTE**

Strimzi container images are available through the [Container Registry](#), but we recommend using the provided YAML files for deployment.

# Chapter 6. Preparing for your deployment

Prepare for a deployment of Strimzi by completing any necessary pre-deployment tasks. Take the necessary preparatory steps according to your specific requirements, such as the following:

- Ensuring you have the necessary prerequisites before deploying Strimzi
- Considering operator deployment best practices
- Pushing the Strimzi container images into your own registry (if required)
- Setting up admin roles to enable configuration of custom resources used in the deployment

**NOTE**

To run the commands in this guide, your cluster user must have the rights to manage role-based access control (RBAC) and CRDs.

## 6.1. Deployment prerequisites

To deploy Strimzi, you will need the following:

- A Kubernetes 1.27 and later cluster.
- The `kubectl` command-line tool is installed and configured to connect to the running cluster.

For more information on the tools available for running Kubernetes, see [Install Tools](#) in the Kubernetes documentation.

**NOTE**

Strimzi supports some features that are specific to OpenShift, where such integration benefits OpenShift users and there is no equivalent implementation using standard Kubernetes.

### oc and kubectl commands

The `oc` command functions as an alternative to `kubectl`. In almost all cases the example `kubectl` commands used in this guide can be done using `oc` simply by replacing the command name (options and arguments remain the same).

In other words, instead of using:

```
kubectl apply -f <your_file>
```

when using OpenShift you can use:

```
oc apply -f <your_file>
```

**NOTE**

As an exception to this general rule, `oc` uses `oc adm` subcommands for *cluster management* functionality, whereas `kubectl` does not make this distinction. For example, the `oc` equivalent of `kubectl taint` is `oc adm taint`.

## 6.2. Planning your Cluster Operator deployment

To support a stable and reliable Strimzi deployment, follow the best practices in this section. Run a single Cluster Operator per Kubernetes cluster, choose an appropriate watch strategy, and isolate components within watched namespaces to reduce the risk of conflicts and unexpected behavior.

### 6.2.1. Avoiding deployment conflicts

A single operator is capable of managing multiple Kafka clusters across different namespaces. Deploying multiple instances of the Cluster Operator, particularly with different versions, introduces the following risks:

#### Resource conflicts

Conflicts over cluster-scoped resources like Custom Resource Definitions (CRDs) and ClusterRoles, leading to unpredictable behavior. This conflict occurs even when the operators are deployed in separate namespaces.

#### Version incompatibility

Different operator versions can create compatibility issues with the Kafka clusters they manage. New Strimzi releases may introduce features, bug fixes, or other changes that are not backward-compatible.

#### *Approach to avoid risks*

To avoid these risks, the recommended approach to deploying the Cluster Operator is as follows:

#### Run a single Cluster Operator

Deploy only one Cluster Operator per Kubernetes cluster.

#### Consider a dedicated namespace

Install the Cluster Operator in its own namespace, separate from the Kafka components it manages. This separation is most useful when the operator is configured to watch multiple namespaces, but it can also help prevent uncontrolled growth of resources in a single namespace.

#### Keep everything updated

Regularly update Strimzi and the version of Kafka it manages so that you have the latest features, bug fixes, and enhancements.

### 6.2.2. Choosing namespace watch options

You configure the Cluster Operator to watch for changes to [Kafka resources](#) in specific namespaces.

You can configure the operator to watch:

- [A single selected namespace](#)
- [A specific list of multiple namespaces](#)
- [All namespaces in the cluster](#)

Choosing to watch a specific list of multiple namespaces can have the biggest impact on performance due to increased processing overhead. To optimize performance, the recommended modes are to either watch a single namespace for focused monitoring or all namespaces for a comprehensive view of the entire cluster.

### 6.2.3. Isolating components in watched namespaces

After deploying the Cluster Operator, it begins watching specified namespaces for changes to Kafka resources. To reduce risks and maintain reliability, isolate component types within each watched namespace. Each namespace should contain only one instance of a given component type, such as one Kafka cluster, to avoid the following types of issues:

- Conflicting resource names
- Ambiguity in access management
- Topic and user name collisions
- Unpredictable behavior during upgrades or recovery

## 6.3. Pushing container images to your own registry

Container images for Strimzi are available in the [Container Registry](#). The installation YAML files provided by Strimzi will pull the images directly from the [Container Registry](#).

If you do not have access to the [Container Registry](#) or want to use your own container repository:

1. Pull **all** container images listed here
2. Push them into your own registry
3. Update the image names in the YAML files used in deployment

**NOTE**    Each Kafka version supported for the release has a separate image.

Container image	Namespace/Repository	Description
Kafka	<ul style="list-style-type: none"><li>• quay.io/strimzi/kafka:0.50.0-kafka-4.0.0</li><li>• quay.io/strimzi/kafka:0.50.0-kafka-4.0.1</li><li>• quay.io/strimzi/kafka:0.50.0-kafka-4.1.0</li><li>• quay.io/strimzi/kafka:0.50.0-kafka-4.1.1</li></ul>	Strimzi image for running Kafka, including: <ul style="list-style-type: none"><li>• Kafka Broker</li><li>• Kafka Connect</li><li>• Kafka MirrorMaker 2</li><li>• Cruise Control</li></ul>

Container image	Namespace/Repository	Description
Operator	<ul style="list-style-type: none"> <li>quay.io/stimzi/operator:0.5.0</li> </ul>	Stimzi image for running the operators: <ul style="list-style-type: none"> <li>Cluster Operator</li> <li>Topic Operator</li> <li>User Operator</li> <li>Kafka Initializer</li> </ul>
HTTP Bridge	<ul style="list-style-type: none"> <li>quay.io/stimzi/kafka-bridge:0.33.1</li> </ul>	Stimzi image for running the HTTP Bridge
Stimzi Drain Cleaner	<ul style="list-style-type: none"> <li>quay.io/stimzi/drain-cleaner:1.5.0</li> </ul>	Stimzi image for running the Stimzi Drain Cleaner

## 6.4. Designating Stimzi administrators

Stimzi provides custom resources for configuration of your deployment. By default, permission to view, create, edit, and delete these resources is limited to Kubernetes cluster administrators. Stimzi provides two cluster roles that you can use to assign these rights to other users:

- `stimzi-view` allows users to view and list Stimzi resources.
- `stimzi-admin` allows users to also create, edit or delete Stimzi resources.

When you install these roles, they will automatically aggregate (add) these rights to the default Kubernetes cluster roles. `stimzi-view` aggregates to the `view` role, and `stimzi-admin` aggregates to the `edit` and `admin` roles. Because of the aggregation, you might not need to assign these roles to users who already have similar rights.

The following procedure shows how to assign a `stimzi-admin` role that allows non-cluster administrators to manage Stimzi resources.

A system administrator can designate Stimzi administrators after the Cluster Operator is deployed.

### Prerequisites

- The Stimzi admin deployment files, which are included in the Stimzi [deployment files](#).
- The Stimzi Custom Resource Definitions (CRDs) and role-based access control (RBAC) resources to manage the CRDs have been [deployed with the Cluster Operator](#).

### Procedure

1. Create the `stimzi-view` and `stimzi-admin` cluster roles in Kubernetes.

```
kubectl create -f install/stimzi-admin
```

2. If needed, assign the roles that provide access rights to users that require them.

```
kubectl create clusterrolebinding strimzi-admin --clusterrole=strimzi-admin --  
user=user1 --user=user2
```

# Chapter 7. Deploying Strimzi using installation files

Download and use the Strimzi [deployment files](#) to deploy Strimzi components to a Kubernetes cluster.

You can deploy Strimzi 0.50.0 on Kubernetes 1.27 and later.

The steps to deploy Strimzi using the installation files are as follows:

1. [Deploy the Cluster Operator](#)
2. Use the Cluster Operator to deploy the following:
  - a. [Kafka cluster](#)
  - b. [Topic Operator](#)
  - c. [User Operator](#)
3. Optionally, deploy the following Kafka components according to your requirements:
  - [Kafka Connect](#)
  - [Kafka MirrorMaker](#)
  - [HTTP Bridge](#)

**NOTE**

To run the commands in this guide, a Kubernetes user must have the rights to manage role-based access control (RBAC) and CRDs.

## 7.1. Deploying the Cluster Operator

The first step for any deployment of Strimzi is to install the Cluster Operator, which is responsible for deploying and managing Kafka clusters within a Kubernetes cluster.

A single command applies all the installation files in the `install/cluster-operator` folder: `kubectl apply -f ./install/cluster-operator`.

The command sets up everything you need to be able to create and manage a Kafka deployment, including the following resources:

- Cluster Operator ([Deployment](#), [ConfigMap](#))
- Strimzi CRDs ([CustomResourceDefinition](#))
- RBAC resources ([ClusterRole](#), [ClusterRoleBinding](#), [RoleBinding](#))
- Service account ([ServiceAccount](#))

Cluster-scoped resources like [CustomResourceDefinition](#), [ClusterRole](#), and [ClusterRoleBinding](#) require administrator privileges for installation. Prior to installation, it's advisable to review the [ClusterRole](#) specifications to ensure they do not grant unnecessary privileges.

After installation, the Cluster Operator runs as a regular [Deployment](#) to watch for updates of Kafka resources. For more information, see [Operator-watched resources](#).

Any standard (non-admin) Kubernetes user with privileges to access the [Deployment](#) can configure it. A cluster administrator can also grant standard users the [privileges necessary to manage Strimzi custom resources](#).

By default, a single replica of the Cluster Operator is deployed. You can add replicas with leader election so that additional Cluster Operators are on standby in case of disruption. For more information, see [Running multiple Cluster Operator replicas with leader election](#).

**WARNING**

While the operator can be configured to watch multiple namespaces, each watched namespace should contain only one instance of a specific component type, such as one Kafka cluster, to avoid conflicts.

### 7.1.1. Deploying the Cluster Operator to watch a single namespace

This procedure shows how to deploy the Cluster Operator to watch Strimzi resources in a single namespace in your Kubernetes cluster.

#### *Prerequisites*

- You need an account with permission to create and manage [CustomResourceDefinition](#) and RBAC ([ClusterRole](#), and [RoleBinding](#)) resources.

#### *Procedure*

1. Edit the Strimzi installation files to use the namespace the Cluster Operator is going to be installed into.

For example, in this procedure the Cluster Operator is installed into the namespace `my-cluster-operator-namespace`.

On Linux, use:

```
sed -i 's/namespace: .*/namespace: my-cluster-operator-namespace/' install/cluster-operator/*RoleBinding*.yaml
```

On MacOS, use:

```
sed -i '' 's/namespace: .*/namespace: my-cluster-operator-namespace/' install/cluster-operator/*RoleBinding*.yaml
```

2. Deploy the Cluster Operator:

```
kubectl create -f install/cluster-operator -n my-cluster-operator-namespace
```

3. Check the status of the deployment:

```
kubectl get deployments -n my-cluster-operator-namespace
```

*Output shows the deployment name and readiness*

NAME	READY	UP-TO-DATE	AVAILABLE
strimzi-cluster-operator	1/1	1	1

**READY** shows the number of replicas that are ready/expected. The deployment is successful when the **AVAILABLE** output shows **1**.

### 7.1.2. Deploying the Cluster Operator to watch multiple namespaces

This procedure shows how to deploy the Cluster Operator to watch Strimzi resources across multiple namespaces in your Kubernetes cluster.

#### Prerequisites

- You need an account with permission to create and manage [CustomResourceDefinition](#) and RBAC ([ClusterRole](#), and [RoleBinding](#)) resources.

#### Procedure

1. Edit the Strimzi installation files to use the namespace the Cluster Operator is going to be installed into.

For example, in this procedure the Cluster Operator is installed into the namespace [my-cluster-operator-namespace](#).

On Linux, use:

```
sed -i 's/namespace: .*/namespace: my-cluster-operator-namespace/' install/cluster-operator/*RoleBinding*.yaml
```

On MacOS, use:

```
sed -i '' 's/namespace: .*/namespace: my-cluster-operator-namespace/' install/cluster-operator/*RoleBinding*.yaml
```

2. Edit the [install/cluster-operator/060-Deployment-strimzi-cluster-operator.yaml](#) file to add a list of all the namespaces the Cluster Operator will watch to the [STRIMZI\\_NAMESPACE](#) environment variable.

For example, in this procedure the Cluster Operator will watch the namespaces [watched-namespace-1](#), [watched-namespace-2](#), [watched-namespace-3](#).

```
apiVersion: apps/v1
kind: Deployment
```

```

spec:
  # ...
  template:
    spec:
      serviceAccountName: strimzi-cluster-operator
      containers:
        - name: strimzi-cluster-operator
          image: quay.io/strimzi/operator:0.50.0
          imagePullPolicy: IfNotPresent
          env:
            - name: STRIMZI_NAMESPACE
              value: watched-namespace-1,watched-namespace-2,watched-namespace-3

```

### 3. For each namespace listed, install the [RoleBindings](#).

In this example, we replace `watched-namespace` in these commands with the namespaces listed in the previous step, repeating them for `watched-namespace-1`, `watched-namespace-2`, `watched-namespace-3`:

```

kubectl create -f install/cluster-operator/020-RoleBinding-strimzi-cluster-
operator.yaml -n <watched_namespace>
kubectl create -f install/cluster-operator/023-RoleBinding-strimzi-cluster-
operator.yaml -n <watched_namespace>
kubectl create -f install/cluster-operator/031-RoleBinding-strimzi-cluster-
operator-entity-operator-delegation.yaml -n <watched_namespace>

```

### 4. Deploy the Cluster Operator:

```
kubectl create -f install/cluster-operator -n my-cluster-operator-namespace
```

### 5. Check the status of the deployment:

```
kubectl get deployments -n my-cluster-operator-namespace
```

*Output shows the deployment name and readiness*

NAME	READY	UP-TO-DATE	AVAILABLE
strimzi-cluster-operator	1/1	1	1

**READY** shows the number of replicas that are ready/expected. The deployment is successful when the **AVAILABLE** output shows **1**.

## 7.1.3. Deploying the Cluster Operator to watch all namespaces

This procedure shows how to deploy the Cluster Operator to watch Strimzi resources across all namespaces in your Kubernetes cluster.

When running in this mode, the Cluster Operator automatically manages clusters in any new namespaces that are created.

#### Prerequisites

- You need an account with permission to create and manage [CustomResourceDefinition](#) and RBAC ([ClusterRole](#), and [RoleBinding](#)) resources.

#### Procedure

1. Edit the Strimzi installation files to use the namespace the Cluster Operator is going to be installed into.

For example, in this procedure the Cluster Operator is installed into the namespace `my-cluster-operator-namespace`.

On Linux, use:

```
sed -i 's/namespace: .*/namespace: my-cluster-operator-namespace/' install/cluster-operator/*RoleBinding*.yaml
```

On MacOS, use:

```
sed -i '' 's/namespace: .*/namespace: my-cluster-operator-namespace/' install/cluster-operator/*RoleBinding*.yaml
```

2. Edit the `install/cluster-operator/060-Deployment-stimzi-cluster-operator.yaml` file to set the value of the `STRIMZI_NAMESPACE` environment variable to `*`.

```
apiVersion: apps/v1
kind: Deployment
spec:
  # ...
  template:
    spec:
      # ...
      serviceAccountName: strimzi-cluster-operator
      containers:
        - name: strimzi-cluster-operator
          image: quay.io/stimzi/operator:0.50.0
          imagePullPolicy: IfNotPresent
          env:
            - name: STRIMZI_NAMESPACE
              value: "*"
            # ...
```

3. Create [ClusterRoleBindings](#) that grant cluster-wide access for all namespaces to the Cluster Operator.

```
kubectl create clusterrolebinding strimzi-cluster-operator-namespaced  
--clusterrole=strimzi-cluster-operator-namespaced --serviceaccount my-cluster-  
operator-namespace:strimzi-cluster-operator  
kubectl create clusterrolebinding strimzi-cluster-operator-watched  
--clusterrole=strimzi-cluster-operator-watched --serviceaccount my-cluster-  
operator-namespace:strimzi-cluster-operator  
kubectl create clusterrolebinding strimzi-cluster-operator-entity-operator-  
delegation --clusterrole=strimzi-entity-operator --serviceaccount my-cluster-  
operator-namespace:strimzi-cluster-operator
```

4. Deploy the Cluster Operator to your Kubernetes cluster.

```
kubectl create -f install/cluster-operator -n my-cluster-operator-namespace
```

5. Check the status of the deployment:

```
kubectl get deployments -n my-cluster-operator-namespace
```

*Output shows the deployment name and readiness*

NAME	READY	UP-TO-DATE	AVAILABLE
strimzi-cluster-operator	1/1	1	1

**READY** shows the number of replicas that are ready/expected. The deployment is successful when the **AVAILABLE** output shows **1**.

## 7.2. Deploying Kafka

To be able to manage a Kafka cluster with the Cluster Operator, you must deploy it as a **Kafka** resource. Strimzi provides example deployment files to do this. You can use these files to deploy the Topic Operator and User Operator at the same time.

After you have deployed the Cluster Operator, use a **Kafka** resource to deploy the following components:

- [Kafka cluster](#)
- [Topic Operator](#)
- [User Operator](#)

Node pools are used in the deployment of a Kafka cluster. Node pools represent a distinct group of Kafka nodes within the Kafka cluster that share the same configuration. For each Kafka node in the node pool, any configuration not defined in node pool is inherited from the cluster configuration in the **Kafka** resource.

If you haven't deployed a Kafka cluster as a **Kafka** resource, you can't use the Cluster Operator to

manage it. This applies, for example, to a Kafka cluster running outside of Kubernetes. However, you can use the Topic Operator and User Operator with a Kafka cluster that is **not managed** by Strimzi, by [deploying them as standalone components](#). You can also deploy and use other Kafka components with a Kafka cluster not managed by Strimzi.

### 7.2.1. Deploying a Kafka cluster

This procedure shows how to deploy a Kafka cluster and associated node pools using the Cluster Operator.

The deployment uses a YAML file to provide the specification to create a [Kafka](#) resource and [KafkaNodePool](#) resources.

Strimzi provides the following [example deployment files](#) that you can use to create a Kafka cluster that uses node pools:

#### [kafka/kafka-with-dual-role-nodes.yaml](#)

Deploys a Kafka cluster with one pool of nodes that share the broker and controller roles.

#### [kafka/kafka-persistent.yaml](#)

Deploys a persistent Kafka cluster with one pool of controller nodes and one pool of broker nodes.

#### [kafka/kafka-ephemeral.yaml](#)

Deploys an ephemeral Kafka cluster with one pool of controller nodes and one pool of broker nodes.

#### [kafka/kafka-single-node.yaml](#)

Deploys a Kafka cluster with a single node.

#### [kafka/kafka-jbod.yaml](#)

Deploys a Kafka cluster with multiple volumes in each broker node.

In this procedure, we use the example deployment file that deploys a Kafka cluster with one pool of nodes that share the broker and controller roles.

The example YAML files specify the latest supported Kafka version and KRaft metadata version used by the Kafka cluster.

#### **WARNING**

When deploying multiple Kafka clusters managed by the Cluster Operator, deploy each cluster into a separate namespace. Deploying multiple clusters in the same namespace can lead to naming conflicts and resource collisions.

#### *Prerequisites*

- [The Cluster Operator must be deployed.](#)

#### *Before you begin*

By default, the example deployment files specify [my-cluster](#) as the Kafka cluster name. The name cannot be changed after the cluster has been deployed. To change the cluster name before you

deploy the cluster, edit the `Kafka.metadata.name` property of the `Kafka` resource in the relevant YAML file.

#### Procedure

1. Deploy a Kafka cluster.

To deploy a Kafka cluster with a single node pool that uses dual-role nodes:

```
kubectl apply -f examples/kafka/kafka-with-dual-role-nodes.yaml
```

2. Check the status of the deployment:

```
kubectl get pods -n <my_cluster_operator_namespace>
```

*Output shows the node pool names and readiness*

NAME	READY	STATUS	RESTARTS
my-cluster-entity-operator	3/3	Running	0
my-cluster-pool-a-0	1/1	Running	0
my-cluster-pool-a-1	1/1	Running	0
my-cluster-pool-a-4	1/1	Running	0

- `my-cluster` is the name of the Kafka cluster.
- `pool-a` is the name of the node pool.

A sequential index number starting with `0` identifies each Kafka pod created.

`READY` shows the number of replicas that are ready/expected. The deployment is successful when the `STATUS` displays as `Running`.

Information on the deployment is also shown in the status of the `KafkaNodePool` resource, including a list of IDs for nodes in the pool.

#### NOTE

Node IDs are assigned sequentially starting at 0 (zero) across all node pools within a cluster. This means that node IDs might not run sequentially within a specific node pool. If there are gaps in the sequence of node IDs across the cluster, the next node to be added is assigned an ID that fills the gap. When scaling down, the node with the highest node ID within a pool is removed.

#### Additional resources

- [Kafka cluster configuration](#)
- [Node pool configuration](#)

### 7.2.2. Deploying the Topic Operator using the Cluster Operator

This procedure describes how to deploy the Topic Operator using the Cluster Operator.

You configure the `entityOperator` property of the `Kafka` resource to include the `topicOperator`. By default, the Topic Operator watches for `KafkaTopic` resources in the namespace of the Kafka cluster deployed by the Cluster Operator. You can also specify a namespace using `watchedNamespace` in the Topic Operator `spec`. A single Topic Operator can watch a single namespace. One namespace should be watched by only one Topic Operator.

**WARNING**

Do not deploy more than one Kafka cluster into the same namespace. This causes the Topic Operator connected to each cluster to compete for the same topic resources using the same names, leading to name collisions and unpredictable behavior.

If you want to use the Topic Operator with a Kafka cluster that is not managed by Strimzi, you must [deploy the Topic Operator as a standalone component](#).

For more information about configuring the `entityOperator` and `topicOperator` properties, see [Configuring the Entity Operator](#).

*Prerequisites*

- The Cluster Operator must be deployed.

*Procedure*

1. Edit the `entityOperator` properties of the `Kafka` resource to include `topicOperator`:

```
apiVersion: kafka.strimzi.io/v1
kind: Kafka
metadata:
  name: my-cluster
spec:
  ...
  entityOperator:
    topicOperator: {}
    userOperator: {}
```

2. Configure the Topic Operator `spec` using the properties described in the [EntityTopicOperatorSpec schema reference](#).

Use an empty object (`{}`) if you want all properties to use their default values.

3. Create or update the resource:

```
kubectl apply -f <kafka_configuration_file>
```

4. Check the status of the deployment:

```
kubectl get pods -n <my_cluster_operator_namespace>
```

*Output shows the pod name and readiness*

NAME	READY	STATUS	RESTARTS
my-cluster-entity-operator	3/3	Running	0
# ...			

`my-cluster` is the name of the Kafka cluster.

`READY` shows the number of replicas that are ready/expected. The deployment is successful when the `STATUS` displays as `Running`.

### 7.2.3. Deploying the User Operator using the Cluster Operator

This procedure describes how to deploy the User Operator using the Cluster Operator.

You configure the `entityOperator` property of the `Kafka` resource to include the `userOperator`. By default, the User Operator watches for `KafkaUser` resources in the namespace of the Kafka cluster deployment. You can also specify a namespace using `watchedNamespace` in the User Operator `spec`. A single User Operator can watch a single namespace. One namespace should be watched by only one User Operator.

**WARNING** Do not deploy more than one Kafka cluster into the same namespace. This can cause the User Operator deployed with each cluster to compete for the same `KafkaUser` resources, leading to name collisions and unpredictable behavior.

If you want to use the User Operator with a Kafka cluster that is not managed by Strimzi, you must [deploy the User Operator as a standalone component](#).

For more information about configuring the `entityOperator` and `userOperator` properties, see [Configuring the Entity Operator](#).

#### Prerequisites

- The Cluster Operator must be deployed.

#### Procedure

1. Edit the `entityOperator` properties of the `Kafka` resource to include `userOperator`:

```
apiVersion: kafka.strimzi.io/v1
kind: Kafka
metadata:
  name: my-cluster
spec:
  #...
  entityOperator:
    topicOperator: {}
    userOperator: {}
```

2. Configure the User Operator `spec` using the properties described in [EntityUserOperatorSpec](#)

## [schema reference](#).

Use an empty object ({} ) if you want all properties to use their default values.

3. Create or update the resource:

```
kubectl apply -f <kafka_configuration_file>
```

4. Check the status of the deployment:

```
kubectl get pods -n <my_cluster_operator_namespace>
```

*Output shows the pod name and readiness*

NAME	READY	STATUS	RESTARTS
my-cluster-entity-operator	3/3	Running	0
# ...			

**my-cluster** is the name of the Kafka cluster.

**READY** shows the number of replicas that are ready/expected. The deployment is successful when the **STATUS** displays as **Running**.

### 7.2.4. List of Kafka cluster resources

The following resources are created by the Cluster Operator in the Kubernetes cluster.

*Shared resources*

#### **<kafka\_cluster\_name>-cluster-ca**

Secret with the Cluster CA private key used to encrypt the cluster communication.

#### **<kafka\_cluster\_name>-cluster-ca-cert**

Secret with the Cluster CA public key. This key can be used to verify the identity of the Kafka brokers.

#### **<kafka\_cluster\_name>-clients-ca**

Secret with the Clients CA private key used to sign user certificates

#### **<kafka\_cluster\_name>-clients-ca-cert**

Secret with the Clients CA public key. This key can be used to verify the identity of the Kafka users.

#### **<kafka\_cluster\_name>-cluster-operator-certs**

Secret with Cluster operators keys for communication with Kafka.

*Kafka brokers and controllers*

## **<kafka\_cluster\_name>-kafka**

Name given to the following Kafka resources:

- Service account used by the Kafka pods.
- PodDisruptionBudget that applies to all Kafka cluster node pool pods.
- Role granting the Kafka brokers and controllers access to read their certificates and credentials.

## **<kafka\_cluster\_name>-kafka-brokers**

Service needed to have DNS resolve the Kafka broker pods IP addresses directly.

## **<kafka\_cluster\_name>-kafka-bootstrap**

Service can be used as bootstrap servers for Kafka clients connecting from within the Kubernetes cluster.

## **<kafka\_cluster\_name>-kafka-external-bootstrap**

Bootstrap service for clients connecting from outside the Kubernetes cluster. This resource is created only when an external listener is enabled. The old service name will be used for backwards compatibility when the listener name is `external` and port is `9094`.

## **<kafka\_cluster\_name>-kafka-external-bootstrap**

Bootstrap route for clients connecting from outside the Kubernetes cluster. This resource is created only when an external listener is enabled and set to type `route`. The old route name will be used for backwards compatibility when the listener name is `external` and port is `9094`.

## **<kafka\_cluster\_name>-kafka-<listener\_name>-bootstrap**

Bootstrap service for clients connecting from outside the Kubernetes cluster. This resource is created only when an external listener is enabled. The new service name will be used for all other external listeners.

## **<kafka\_cluster\_name>-kafka-<listener\_name>-bootstrap**

Bootstrap route for clients connecting from outside the Kubernetes cluster. This resource is created only when an external listener is enabled and set to type `route`. The new route name will be used for all other external listeners.

## **<kafka\_cluster\_name>-kafka-<listener\_name>-bootstrap**

Bootstrap ingress for clients connecting from outside the Kubernetes cluster. This resource is created only when an external listener is enabled and set to type `ingress`. The new route name will be used for all other external listeners.

## **<kafka\_cluster\_name>-network-policy-kafka**

Network policy managing access to the Kafka services.

## **<kafka\_cluster\_name>-kafka-role**

Role binding granting the Kafka brokers and controllers access to read their certificates and credentials.

## **strimzi-namespace-name-<kafka\_cluster\_name>-kafka-init**

Cluster role binding used by the Kafka brokers.

## **<kafka\_cluster\_name>-jmx**

Secret with JMX username and password used to secure the Kafka broker port. This resource is created only when JMX is enabled in Kafka.

### *Kafka node pools*

The resources that are created per node pool. The naming convention includes the name of the Kafka cluster and the node pool: **<kafka\_cluster\_name>-<pool\_name>**.

## **<kafka\_cluster\_name>-<pool\_name>**

Name given to the StrimziPodSet for managing the Kafka node pool.

## **<kafka\_cluster\_name>-<pool\_name>-<pod\_id>**

Name given to the following Kafka node pool resources:

- Pods created by the StrimziPodSet.
- Secret with Kafka node public and private keys.
- ConfigMaps with Kafka node configuration.
- Service used to route traffic from outside the Kubernetes cluster to individual pods. This resource is created only when an external listener is enabled. The old service name will be used for backwards compatibility when the listener name is **external** and port is **9094**.
- Route for traffic from outside the Kubernetes cluster to individual pods. This resource is created only when an external listener is enabled and set to type **route**. The old route name will be used for backwards compatibility when the listener name is **external** and port is **9094**.
- Ingress for traffic from outside the Kubernetes cluster to individual pods. This resource is created only when an external listener is enabled and set to type **ingress**. The old ingress name will be used for backwards compatibility when the listener name is **external** and port is **9094**.

## **<kafka\_cluster\_name>-<pool\_name>-<listener\_name>-<pod\_id>**

Service used to route traffic from outside the Kubernetes cluster to individual pods, or when using the type **cluster-ip** listener. This resource is created only when an external listener is enabled. The new service name will be used for all other external listeners.

## **<kafka\_cluster\_name>-<pool\_name>-<listener\_name>-<pod\_id>**

Route for traffic from outside the Kubernetes cluster to individual pods. This resource is created only when an external listener is enabled and set to type **route**. The new route name will be used for all other external listeners.

## **<kafka\_cluster\_name>-<pool\_name>-<listener\_name>-<pod\_id>**

Ingress for traffic from outside the Kubernetes cluster to individual pods. This resource is created only when an external listener is enabled and set to type **ingress**. The new ingress name will be used for all other external listeners.

### **data-<kafka\_cluster\_name>-<pool\_name>-<pod\_id>**

Persistent Volume Claim for the volume used for storing data for a specific node. This resource is created only if persistent storage is selected for provisioning persistent volumes to store data.

### **data-<id>-<kafka\_cluster\_name>-<pool\_name>-<pod\_id>**

Persistent Volume Claim for the volume **id** used for storing data for a specific node. This resource is created only if persistent storage is selected for JBOD volumes when provisioning persistent volumes to store data.

### *Entity Operator*

These resources are only created if the Entity Operator is deployed using the Cluster Operator.

### **<kafka\_cluster\_name>-entity-operator**

Name given to the following Entity Operator resources:

- Deployment with Topic and User Operators.
- Service account used by the Entity Operator.
- Network policy managing access to the Entity Operator metrics.
- Role granting the Entity Operator the rights to manage topics and users.

### **<kafka\_cluster\_name>-entity-operator-<random\_string>**

Pod created by the Entity Operator deployment.

### **<kafka\_cluster\_name>-entity-topic-operator-config**

ConfigMap with ancillary configuration for Topic Operators.

### **<kafka\_cluster\_name>-entity-user-operator-config**

ConfigMap with ancillary configuration for User Operators.

### **<kafka\_cluster\_name>-entity-topic-operator-certs**

Secret with Topic Operator keys for communication with Kafka.

### **<kafka\_cluster\_name>-entity-user-operator-certs**

Secret with User Operator keys for communication with Kafka.

### **<kafka\_cluster\_name>-entity-topic-operator**

Role binding used by the Entity Topic Operator.

### **<kafka\_cluster\_name>-entity-user-operator**

Role binding used by the Entity User Operator.

### *Kafka Exporter*

These resources are only created if the Kafka Exporter is deployed using the Cluster Operator.

### **<kafka\_cluster\_name>-kafka-exporter**

Name given to the following Kafka Exporter resources:

- Deployment with Kafka Exporter.

- Service used to collect consumer lag metrics.
- Service account used by the Kafka Exporter.
- Network policy managing access to the Kafka Exporter metrics.

#### <kafka\_cluster\_name>-kafka-exporter-<random\_string>

Pod created by the Kafka Exporter deployment.

#### *Cruise Control*

These resources are only created if Cruise Control was deployed using the Cluster Operator.

#### <kafka\_cluster\_name>-cruise-control

Name given to the following Cruise Control resources:

- Deployment with Cruise Control.
- Service used to communicate with Cruise Control.
- Service account used by the Cruise Control.

#### <kafka\_cluster\_name>-cruise-control-<random\_string>

Pod created by the Cruise Control deployment.

#### <kafka\_cluster\_name>-cruise-control-config

ConfigMap that contains the Cruise Control ancillary configuration, and is mounted as a volume by the Cruise Control pods.

#### <kafka\_cluster\_name>-cruise-control-certs

Secret with Cruise Control keys for communication with Kafka.

#### <kafka\_cluster\_name>-network-policy-cruise-control

Network policy managing access to the Cruise Control service.

## 7.3. Deploying 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 or messaging system, for import or export of data using connectors. Connectors are plugins that provide the connection configuration needed.

In Strimzi, Kafka Connect is deployed in distributed mode. Kafka Connect can also work in standalone mode, but this is not supported by Strimzi.

Using the concept of *connectors*, Kafka Connect provides a framework for moving large amounts of data into and out of your Kafka cluster while maintaining scalability and reliability.

The Cluster Operator manages Kafka Connect clusters deployed using the [KafkaConnect](#) resource and connectors created using the [KafkaConnector](#) resource.

In order to use Kafka Connect, you need to do the following.

- Deploy a Kafka Connect cluster
- Add connectors to integrate with other systems

**NOTE** The term *connector* is used interchangeably to mean a connector instance running within a Kafka Connect cluster, or a connector class. In this guide, the term *connector* is used when the meaning is clear from the context.

### 7.3.1. Deploying Kafka Connect

This procedure shows how to deploy a Kafka Connect cluster to your Kubernetes cluster using the Cluster Operator.

A Kafka Connect cluster deployment is implemented with a configurable number of nodes (also called *workers*) that distribute the workload of connectors as *tasks*, ensuring a scalable and reliable message flow.

The deployment uses a YAML file to provide the specification to create a [KafkaConnect](#) resource.

Strimzi provides [example configuration files](#). In this procedure, we use the following example file:

- `examples/connect/kafka-connect.yaml`

**WARNING** When deploying multiple Kafka Connect clusters managed by the Cluster Operator, deploy each cluster into a separate namespace. Deploying multiple clusters in the same namespace can lead to naming conflicts and resource collisions.

#### Prerequisites

- Cluster Operator is deployed.
- Kafka cluster is running.

This procedure assumes that the Kafka cluster was deployed using Strimzi.

#### Procedure

1. Edit the deployment file to configure connection details (if required).

In `examples/connect/kafka-connect.yaml`, add or update the following properties as needed:

- `spec.bootstrapServers` to specify the Kafka bootstrap address.
- `spec.authentication` to specify the authentication type as `tls`, `scram-sha-256`, `scram-sha-512`, `plain`, or `custom`. See the [KafkaConnectSpec schema properties](#) for configuration details.
- `spec.tls.trustedCertificates` to configure the TLS certificate.  
Use `[]` (an empty array) to trust the default Java CAs, or specify secrets containing trusted certificates. See the [trustedCertificates properties](#) for configuration details.

2. Configure the deployment for multiple Kafka Connect clusters (if required).

If you plan to run more than one Kafka Connect cluster in the same environment, each instance

must use unique internal topic names and a unique group ID.

Update the `spec.config` properties in `kafka-connect.yaml` to replace the default values.

See [Configuring multiple Kafka Connect clusters](#) for details.

3. Deploy Kafka Connect to your Kubernetes cluster. Use the `examples/connect/kafka-connect.yaml` file to deploy Kafka Connect.

```
kubectl apply -f examples/connect/kafka-connect.yaml
```

4. Check the status of the deployment:

```
kubectl get pods -n <my_cluster_operator_namespace>
```

*Output shows the deployment name and readiness*

NAME	READY	STATUS	RESTARTS
my-connect-cluster-connect-<pod_id>	1/1	Running	0

In this example, `my-connect-cluster` is the name of the Kafka Connect cluster. A pod ID identifies each created pod. By default, the deployment creates a single Kafka Connect pod. `READY` shows the number of ready versus expected replicas. The deployment is successful when the `STATUS` is `Running`.

*Additional resources*

[Kafka Connect cluster configuration](#)

### 7.3.2. List of Kafka Connect cluster resources

The following resources are created by the Cluster Operator in the Kubernetes cluster:

#### `<connect_cluster_name>-connect`

Name given to the following Kafka Connect resources:

- StrimziPodSet that creates the Kafka Connect worker node pods.
- Headless service that provides stable DNS names to the Kafka Connect pods.
- Service account used by the Kafka Connect pods.
- Pod disruption budget configured for the Kafka Connect worker nodes.
- Network policy managing access to the Kafka Connect REST API.
- Role granting the Kafka Connect worker nodes access to read their certificates and credentials.

#### `<connect_cluster_name>-connect-<pod_id>`

Pods created by the Kafka Connect StrimziPodSet.

## <connect\_cluster\_name>-connect-tls-trusted-certs

Secret with TLS certificates.

## <connect\_cluster\_name>-connect-api

Service which exposes the REST interface for managing the Kafka Connect cluster.

## <connect\_cluster\_name>-connect-config

ConfigMap which contains the Kafka Connect ancillary configuration and is mounted as a volume by the Kafka Connect pods.

## <connect\_cluster\_name>-connect-role

Role binding granting the Kafka Connect worker nodes access to read their certificates and credentials.

## strimzi-<namespace-name>-<connect\_cluster\_name>-connect-init

Cluster role binding used by the Kafka Connect cluster.

## <connect\_cluster\_name>-connect-build

Pod used to build a new container image with additional connector plugins (only when Kafka Connect Build feature is used).

## <connect\_cluster\_name>-connect-dockerfile

ConfigMap with the Dockerfile generated to build the new container image with additional connector plugins (only when the Kafka Connect build feature is used).

## 7.4. Adding Kafka Connect connectors

Kafka Connect uses connectors to integrate with other systems to stream data. A connector is an instance of a Kafka [Connector](#) class, which can be one of the following type:

### Source connector

A source connector is a runtime entity that fetches data from an external system and feeds it to Kafka as messages.

### Sink connector

A sink connector is a runtime entity that fetches messages from Kafka topics and feeds them to an external system.

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.

Plugins provide a set of one or more artifacts that define a connector and task implementation for connecting to a given kind of data source. The configuration describes the source input data and target output data to feed into and out of Kafka Connect. 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. Plugins for many external systems are available for use with Kafka Connect. You can also create your own plugins.

Add connector plugins to Kafka Connect in one of the following ways:

- [Configure Kafka Connect to build a new container image with plugins automatically](#)
- [Mount connector plugins as Kubernetes Image volumes](#)
- [Create a Docker image from the base Kafka Connect image](#) (manually or using continuous integration)

After plugins have been added to the container image, you can start, stop, and manage connector instances in the following ways:

- [Using Strimzi's KafkaConnector custom resource](#)
- [Using the Kafka Connect API](#)

You can also create new connector instances using these options.

#### 7.4.1. Building new container images with connector plugins automatically

Configure Kafka Connect so that Strimzi automatically builds a new container image with additional connectors. You define the connector plugins using the `.spec.build.plugins` property of the `KafkaConnect` custom resource.

Strimzi automatically downloads and adds the connector plugins into a new container image. The container is pushed into the container repository specified in `.spec.build.output` and automatically used in the Kafka Connect deployment.

##### Prerequisites

- [The Cluster Operator must be deployed.](#)
- A container registry.

You need to provide your own container registry where images can be pushed to, stored, and pulled from. Strimzi supports private container registries as well as public registries such as [Quay](#) or [Docker Hub](#).

##### Procedure

1. Configure the `KafkaConnect` custom resource by specifying the container registry in `.spec.build.output`, and additional connectors in `.spec.build.plugins`:

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaConnect
metadata:
  name: my-connect-cluster
spec: ①
```

```

#...
build:
  output: ②
    type: docker
    image: my-registry.io/my-org/my-connect-cluster:latest
    pushSecret: my-registry-credentials
  plugins: ③
    - name: connector-1
      artifacts:
        - type: tgz
          url: <url_to_download_connector_1_artifact>
          sha512sum: <SHA-512_checksum_of_connector_1_artifact>
    - name: connector-2
      artifacts:
        - type: jar
          url: <url_to_download_connector_2_artifact>
          sha512sum: <SHA-512_checksum_of_connector_2_artifact>
#...

```

① The specification for the Kafka Connect cluster.

② (Required) Configuration of the container registry where new images are pushed.

③ (Required) List of connector plugins and their artifacts to add to the new container image. Each plugin must be configured with at least one **artifact**.

## 2. Create or update the resource:

```
$ kubectl apply -f <kafka_connect_configuration_file>
```

3. Wait for the new container image to build, and for the Kafka Connect cluster to be deployed.

4. Use the Kafka Connect REST API or **KafkaConnector** custom resources to use the connector plugins you added.

### *Rebuilding the container image with new artifacts*

A new container image is built automatically when you change the base image (**.spec.image**) or change the connector plugin artifacts configuration (**.spec.build.plugins**).

To pull an upgraded base image or to download the latest connector plugin artifacts without changing the **KafkaConnect** resource, you can trigger a rebuild of the container image associated with the Kafka Connect cluster by applying the annotation **strimzi.io/force-rebuild=true** to the Kafka Connect **StrimziPodSet** resource.

The annotation triggers the rebuilding process, fetching any new artifacts for plugins specified in the **KafkaConnect** custom resource and incorporating them into the container image. The rebuild includes downloads of new plugin artifacts without versions.

### *Additional resources*

- [Kafka Connect Build schema reference](#)

## 7.4.2. Using Kubernetes Image Volumes to add connector plugins

Configure Kafka Connect so that connectors are automatically mounted as [Kubernetes Image Volumes](#). You define the connector plugins using the `.spec.plugins` property of the [KafkaConnect](#) custom resource. Strimzi automatically mounts them into the Kafka Connect deployment.

**IMPORTANT**

This feature requires that the Kubernetes Image Volume feature is enabled and supported by your Kubernetes cluster.

*Prerequisites*

- [The Cluster Operator must be deployed.](#)
- Your connector plugins must be available in your container registry as container images (OCI artifacts).

*Procedure*

1. Configure the [KafkaConnect](#) custom resource by specifying the additional connector plugins in `.spec.plugins`:

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaConnect
metadata:
  name: my-connect-cluster
spec: ①
  #...
  plugins: ②
    - name: connector-1
      artifacts:
        - type: image
          reference: <reference_to_container_image_or_OCI_artifact>
          pullPolicy: <pull_policy>
    - name: connector-2
      artifacts:
        - type: image
          reference: <reference_to_container_image_or_OCI_artifact>
          pullPolicy: <pull_policy>
  #...
```

① The specification for the Kafka Connect cluster.

② (Required) List of connector plugins and their artifacts to mount in the Kafka Connect deployment. Each plugin must be configured with at least one [artifact](#).

2. Create or update the resource:

```
$ kubectl apply -f <kafka_connect_configuration_file>
```

3. Wait for the Kafka Connect cluster to be deployed.

4. Use the Kafka Connect REST API or [KafkaConnector](#) custom resources to configure and run connectors based on the plugins you added.

#### *Additional resources*

- [Kafka Connect plugins schema reference](#)

### 7.4.3. Building custom Kafka Connect images with connector plugins

Create a custom Docker image with connector plugins from the Kafka Connect base image. Add the custom image to the [/opt/kafka/plugins](#) directory.

You can use the Kafka container image on [Container Registry](#) as a base image for creating your own custom image with additional connector plugins. At startup, the Strimzi version of Kafka Connect loads any third-party connector plugins contained in [/opt/kafka/plugins](#).

Use the [KafkaConnect.spec.image](#) property to scope the custom image to a single Kafka Connect cluster. Editing the [STRIMZI\\_KAFKA\\_CONNECT\\_IMAGES](#) environment variable in the Cluster Operator instead applies the image globally to all Connect clusters.

This procedure shows how to do the following:

- Build a custom image from the Kafka Connect base.
- Push the image to your registry.
- Reference the image by setting [KafkaConnect.spec.image](#).

**NOTE**

When you specify a custom image, coordinate upgrades to keep the image and the Kafka version aligned.

#### *Prerequisites*

- [The Cluster Operator must be deployed.](#)

#### *Procedure*

1. Create a new [Dockerfile](#) using [quay.io/strimzi/kafka:0.50.0-kafka-4.1.1](#) as the base image:

```
FROM quay.io/strimzi/kafka:0.50.0-kafka-4.1.1
USER root:root
COPY ./my-plugins/ /opt/kafka/plugins/
USER 1001
```

#### *Example plugins file*

```
$ tree ./my-plugins/
./my-plugins/
├── debezium-connector-mongodb
│   ├── bson-<version>.jar
│   ├── CHANGELOG.md
│   ├── CONTRIBUTE.md
│   └── COPYRIGHT.txt
```

```

    ├── debezium-connector-mongodb-<version>.jar
    ├── debezium-core-<version>.jar
    ├── LICENSE.txt
    ├── mongodb-driver-core-<version>.jar
    ├── README.md
    └── # ...
  └── debezium-connector-mysql
    ├── CHANGELOG.md
    ├── CONTRIBUTE.md
    ├── COPYRIGHT.txt
    ├── debezium-connector-mysql-<version>.jar
    ├── debezium-core-<version>.jar
    ├── LICENSE.txt
    ├── mysql-binlog-connector-java-<version>.jar
    ├── mysql-connector-java-<version>.jar
    ├── README.md
    └── # ...
  └── debezium-connector-postgres
    ├── CHANGELOG.md
    ├── CONTRIBUTE.md
    ├── COPYRIGHT.txt
    ├── debezium-connector-postgres-<version>.jar
    ├── debezium-core-<version>.jar
    ├── LICENSE.txt
    ├── postgresql-<version>.jar
    ├── protobuf-java-<version>.jar
    ├── README.md
    └── # ...

```

The COPY command points to the plugin files to copy to the container image.

This example adds plugins for Debezium connectors (MongoDB, MySQL, and PostgreSQL), though not all files are listed for brevity. Debezium running in Kafka Connect looks the same as any other Kafka Connect task.

2. Build the container image.
3. Push your custom image to your container registry.
4. Point the `KafkaConnect` custom resource to the new container image.

Edit the `KafkaConnect.spec.image` property of the KafkaConnect custom resource to specify your custom image.

```

apiVersion: kafka.strimzi.io/v1
kind: KafkaConnect
metadata:
  name: my-connect-cluster
spec: ①
  version: 4.1.1 ②
  image: my-new-container-image ③

```

```
#...
```

- ① The specification for the Kafka Connect cluster.
- ② The Kafka version. Always specify the Kafka version that matches your Kafka cluster to avoid version conflicts.
- ③ The custom docker image for Kafka Connect pods.

**WARNING**

Check the Kafka version you need for your custom image. If `spec.version` is unspecified, Strimzi defaults to Kafka 4.1.1, which may cause compatibility issues if your Kafka cluster runs an earlier version. Version mismatches may lead to connector failures and runtime errors.

*Additional resources*

- Container image configuration and the `KafkaConnect.spec.image` property
- Cluster Operator configuration and the `STRIMZI_KAFKA_CONNECT_IMAGES` variable

#### 7.4.4. Deploying KafkaConnector resources

Deploy `KafkaConnector` resources to manage connectors. The `KafkaConnector` custom resource offers a Kubernetes-native approach to management of connectors by the Cluster Operator. You don't need to send HTTP requests to manage connectors, as with the Kafka Connect REST API. You manage a running connector instance by updating its corresponding `KafkaConnector` resource, and then applying the updates. The Cluster Operator updates the configurations of the running connector instances. You remove a connector by deleting its corresponding `KafkaConnector`.

`KafkaConnector` resources must be deployed to the same namespace as the Kafka Connect cluster they link to.

In the configuration shown in this procedure, the `autoRestart` feature is enabled (`enabled: true`) for automatic restarts of failed connectors and tasks. You can also annotate the `KafkaConnector` resource to `restart a connector` or `restart a connector task` manually.

*Example connectors*

You can use your own connectors or try the examples provided by Strimzi. Up until Apache Kafka 3.1.0, example file connector plugins were included with Apache Kafka. Starting from the 3.1.1 and 3.2.0 releases of Apache Kafka, the examples need to be `added to the plugin path as any other connector`.

Strimzi provides an `example KafkaConnector configuration file` (`examples/connect/source-connector.yaml`) for the example file connector plugins, which creates the following connector instances as `KafkaConnector` resources:

- A `FileStreamSourceConnector` instance that reads each line from the Kafka license file (the source) and writes the data as messages to a single Kafka topic.
- A `FileStreamSinkConnector` instance that reads messages from the Kafka topic and writes the messages to a temporary file (the sink).

We use the example file to create connectors in this procedure.

**NOTE** The example connectors are not intended for use in a production environment.

#### Prerequisites

- A Kafka Connect deployment
- The Cluster Operator is running

#### Procedure

1. Add the [FileStreamSourceConnector](#) and [FileStreamSinkConnector](#) plugins to Kafka Connect in one of the following ways:
  - [Configure Kafka Connect to build a new container image with plugins automatically](#)
  - [Create a Docker image from the base Kafka Connect image](#) (manually or using continuous integration)
2. Set the `strimzi.io/use-connector-resources` annotation to `true` in the Kafka Connect configuration.

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaConnect
metadata:
  name: my-connect-cluster
  annotations:
    strimzi.io/use-connector-resources: "true"
spec:
  # ...
```

With the [KafkaConnector](#) resources enabled, the Cluster Operator watches for them.

3. Edit the [examples/connect/source-connector.yaml](#) file:

#### Example source connector configuration

```
apiVersion: kafka.strimzi.io/v1
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
  autoRestart: ⑤
  enabled: true
  config: ⑥
    file: "/opt/kafka/LICENSE" ⑦
    topic: my-topic ⑧
```

```
# ...
```

- 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 Enables automatic restarts of failed connectors and tasks. By default, the number of restarts is indefinite, but you can set a maximum on the number of automatic restarts using the `maxRestarts` property.
- ⑥ [Connector configuration](#) as key-value pairs.
- ⑦ Location of the external data file. In this example, we're configuring the `FileStreamSourceConnector` to read from the `/opt/kafka/LICENSE` file.
- ⑧ Kafka topic to publish the source data to.

#### 4. Create the source `KafkaConnector` in your Kubernetes cluster:

```
kubectl apply -f examples/connect/source-connector.yaml
```

#### 5. Create an `examples/connect/sink-connector.yaml` file:

```
touch examples/connect/sink-connector.yaml
```

#### 6. Paste the following YAML into the `sink-connector.yaml` file:

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaConnector
metadata:
  name: my-sink-connector
  labels:
    strimzi.io/cluster: my-connect
spec:
  class: org.apache.kafka.connect.file.FileStreamSinkConnector ①
  tasksMax: 2
  config: ②
    file: "/tmp/my-file" ③
    topics: my-topic ④
```

- ① Full name or alias of the connector class. This should be present in the image being used by the Kafka Connect cluster.
- ② [Connector configuration](#) as key-value pairs.

③ Temporary file to publish the source data to.

④ Kafka topic to read the source data from.

## 7. Create the sink [KafkaConnector](#) in your Kubernetes cluster:

```
kubectl apply -f examples/connect/sink-connector.yaml
```

## 8. Check that the connector resources were created:

```
kubectl get kctr --selector strimzi.io/cluster=<my_connect_cluster> -o name  
my-source-connector  
my-sink-connector
```

Replace <my\_connect\_cluster> with the name of your Kafka Connect cluster.

## 9. In the container, execute [kafka-console-consumer.sh](#) to read the messages that were written to the topic by the source connector:

```
kubectl exec <my_kafka_cluster>-kafka-0 -i -t -- bin/kafka-console-consumer.sh  
--bootstrap-server <my_kafka_cluster>-kafka-bootstrap.NAMESPACE.svc:9092 --topic  
my-topic --from-beginning
```

Replace <my\_kafka\_cluster> with the name of your Kafka cluster.

## Source and sink connector configuration options

The connector configuration is defined in the `spec.config` property of the [KafkaConnector](#) resource.

The [FileStreamSourceConnector](#) and [FileStreamSinkConnector](#) classes support the same configuration options as the Kafka Connect REST API. Other connectors support different configuration options.

*Table 8. Configuration options for the [FileStreamSource](#) connector class*

Name	Type	Default value	Description
<code>file</code>	String	Null	Source file to write messages to. If not specified, the standard input is used.
<code>topic</code>	List	Null	The Kafka topic to publish data to.

*Table 9. Configuration options for [FileStreamSinkConnector](#) class*

Name	Type	Default value	Description
file	String	Null	Destination file to write messages to. If not specified, the standard output is used.
topics	List	Null	One or more Kafka topics to read data from.
topics.regex	String	Null	A regular expression matching one or more Kafka topics to read data from.

#### 7.4.5. Exposing the 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. The service is created when you create a Kafka Connect instance.

The operations supported by the Kafka Connect REST API are described in the [Apache Kafka Connect API documentation](#).

**NOTE**

The `strimzi.io/use-connector-resources` annotation enables KafkaConnectors. If you applied the annotation to your [KafkaConnect](#) resource configuration, you need to remove it to use the Kafka Connect API. Otherwise, manual changes made directly using the Kafka Connect REST API are reverted by the Cluster Operator.

You can 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"
  }
}'
```

The API is only accessible within the Kubernetes cluster. If you want to make the Kafka Connect API

accessible to applications running outside of the Kubernetes cluster, you can expose it manually by creating one of the following features:

- [LoadBalancer](#) or [NodePort](#) type services
- [Ingress](#) resources (Kubernetes only)
- OpenShift routes (OpenShift only)

**NOTE** The connection is insecure, so allow external access advisedly.

If you decide to create services, use the labels from the [selector](#) of the `<connect_cluster_name>-connect-api` service to configure the pods to which the service will route the traffic:

*Selector configuration for the service*

```
# ...
selector:
  strimzi.io/cluster: my-connect-cluster ①
  strimzi.io/kind: KafkaConnect
  strimzi.io/name: my-connect-cluster-connect ②
#...
```

① Name of the Kafka Connect custom resource in your Kubernetes cluster.

② Name of the Kafka Connect deployment created by the Cluster Operator.

You must also create a [NetworkPolicy](#) that allows HTTP requests from external clients.

*Example NetworkPolicy to allow requests to the Kafka Connect API*

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: my-custom-connect-network-policy
spec:
  ingress:
    - from:
      - podSelector: ①
        matchLabels:
          app: my-connector-manager
    ports:
      - port: 8083
        protocol: TCP
  podSelector:
    matchLabels:
      strimzi.io/cluster: my-connect-cluster
      strimzi.io/kind: KafkaConnect
      strimzi.io/name: my-connect-cluster-connect
  policyTypes:
    - Ingress
```

① The label of the pod that is allowed to connect to the API.

To add the connector configuration outside the cluster, use the URL of the resource that exposes the API in the curl command.

#### 7.4.6. Limiting access to the Kafka Connect API

It is crucial to restrict access to the Kafka Connect API only to trusted users to prevent unauthorized actions and potential security issues. The Kafka Connect API provides extensive capabilities for altering connector configurations, which makes it all the more important to take security precautions. Someone with access to the Kafka Connect API could potentially obtain sensitive information that an administrator may assume is secure.

The Kafka Connect REST API can be accessed by anyone who has authenticated access to the Kubernetes cluster and knows the endpoint URL, which includes the hostname/IP address and port number.

For example, suppose an organization uses a Kafka Connect cluster and connectors to stream sensitive data from a customer database to a central database. The administrator uses a configuration provider plugin to store sensitive information related to connecting to the customer database and the central database, such as database connection details and authentication credentials. The configuration provider protects this sensitive information from being exposed to unauthorized users. However, someone who has access to the Kafka Connect API can still obtain access to the customer database without the consent of the administrator. They can do this by setting up a fake database and configuring a connector to connect to it. They then modify the connector configuration to point to the customer database, but instead of sending the data to the central database, they send it to the fake database. By configuring the connector to connect to the fake database, the login details and credentials for connecting to the customer database are intercepted, even though they are stored securely in the configuration provider.

If you are using the [KafkaConnector](#) custom resources, then by default the Kubernetes RBAC rules permit only Kubernetes cluster administrators to make changes to connectors. You can also [designate non-cluster administrators to manage Strimzi resources](#). With [KafkaConnector](#) resources enabled in your Kafka Connect configuration, changes made directly using the Kafka Connect REST API are reverted by the Cluster Operator. If you are not using the [KafkaConnector](#) resource, the default RBAC rules do not limit access to the Kafka Connect API. If you want to limit direct access to the Kafka Connect REST API using Kubernetes RBAC, you need to enable and use the [KafkaConnector](#) resources.

For improved security, we recommend configuring the following properties for the Kafka Connect API:

##### `org.apache.kafka.disallowed.login.modules`

(Kafka 3.4 or later) Set the `org.apache.kafka.disallowed.login.modules` Java system property to prevent the use of insecure login modules. For example, specifying `com.sun.security.auth.module.JndiLoginModule` prevents the use of the Kafka `JndiLoginModule`.

#### *Example configuration for disallowing login modules*

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaConnect
metadata:
  name: my-connect-cluster
  annotations:
    strimzi.io/use-connector-resources: "true"
spec:
  # ...
  jvmOptions:
    javaSystemProperties:
      - name: org.apache.kafka.disallowed.login.modules
        value: com.sun.security.auth.module.JndiLoginModule,
          org.apache.kafka.common.security.kerberos.KerberosLoginModule
  # ...
```

Only allow trusted login modules and follow the latest advice from Kafka for the version you are using. As a best practice, you should explicitly disallow insecure login modules in your Kafka Connect configuration by using the `org.apache.kafka.disallowed.login.modules` system property.

#### **connector.client.config.override.policy**

Set the `connector.client.config.override.policy` property to `None` to prevent connector configurations from overriding the Kafka Connect configuration and the consumers and producers it uses.

#### *Example configuration to specify connector override policy*

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaConnect
metadata:
  name: my-connect-cluster
  annotations:
    strimzi.io/use-connector-resources: "true"
spec:
  # ...
  config:
    connector.client.config.override.policy: None
  # ...
```

### **7.4.7. Switching to using KafkaConnector custom resources**

You can switch from using the Kafka Connect API to using `KafkaConnector` custom resources to manage your connectors. To make the switch, do the following in the order shown:

1. Deploy `KafkaConnector` resources with the configuration to create your connector instances.
2. Enable `KafkaConnector` resources in your Kafka Connect configuration by setting the `strimzi.io/use-connector-resources` annotation to `true`.

**WARNING**

If you enable `KafkaConnector` resources before creating them, you delete all connectors.

To switch from using `KafkaConnector` resources to using the Kafka Connect API, first remove the annotation that enables the `KafkaConnector` resources from your Kafka Connect configuration. Otherwise, manual changes made directly using the Kafka Connect REST API are reverted by the Cluster Operator.

When making the switch, [check the status of the `KafkaConnect` resource](#). The value of `metadata.generation` (the current version of the deployment) must match `status.observedGeneration` (the latest reconciliation of the resource). When the Kafka Connect cluster is `Ready`, you can delete the `KafkaConnector` resources.

## 7.5. Deploying Kafka MirrorMaker

Kafka MirrorMaker replicates data between two or more Kafka clusters, within or across data centers. This process is called mirroring to avoid confusion with the concept of Kafka partition replication. MirrorMaker consumes messages from a source cluster and republishes those messages to a target cluster.

Data replication across clusters supports scenarios that require the following:

- Recovery of data in the event of a system failure
- Consolidation of data from multiple source clusters for centralized analysis
- Restriction of data access to a specific cluster
- Provision of data at a specific location to improve latency

### 7.5.1. Deploying Kafka MirrorMaker

This procedure shows how to deploy a Kafka MirrorMaker 2 cluster to your Kubernetes cluster using the Cluster Operator.

The deployment uses a YAML file to provide the specification to create a `KafkaMirrorMaker2` resource. MirrorMaker 2 is based on Kafka Connect and uses its configuration properties.

Strimzi provides [example configuration files](#). In this procedure, we use the following example file:

- `examples/mirror-maker/kafka-mirror-maker-2.yaml`

**WARNING**

When deploying multiple Kafka MirrorMaker 2 clusters managed by the Cluster Operator, deploy each cluster into a separate namespace. Deploying multiple clusters in the same namespace can lead to naming conflicts and resource collisions.

#### *Prerequisites*

- Cluster Operator is deployed.
- Kafka cluster is running.

This procedure assumes that the Kafka cluster was deployed using Strimzi.

#### Procedure

1. Edit the deployment file to configure connection details (if required).

In `examples/mirror-maker/kafka-mirror-maker-2.yaml`, add or update the following properties as needed:

- `spec.target.bootstrapServers` and `.spec.mirrors[].source.bootstrapServers` to specify the Kafka bootstrap address for the source and target clusters.
- `spec.target.alias` and `.spec.mirrors[].source.alias` to specify a unique identifier for each cluster.
- `spec.target.authentication` and `.spec.mirrors[].source.authentication` to specify the authentication type for each cluster as `tls`, `scram-sha-256`, `scram-sha-512`, `plain`, or `custom`. See the [KafkaMirrorMaker2Spec schema properties](#) for configuration details.
- `spec.target.tls.trustedCertificates` and `.spec.mirrors[].source.tls.trustedCertificates` to configure the TLS certificate for each cluster. Use `[]` (an empty array) to trust the default Java CAs, or specify secrets containing trusted certificates. See the [trustedCertificates properties](#) for configuration details.

2. Configure the deployment for multiple MirrorMaker 2 clusters (if required).

If you plan to run more than one MirrorMaker 2 cluster in the same environment, each instance must use unique internal topic names and a unique group ID.

Update the `spec.target` properties in `kafka-mirror-maker-2.yaml` to replace the default values.

See [Configuring multiple MirrorMaker 2 clusters](#) for details.

3. Deploy Kafka MirrorMaker to your Kubernetes cluster:

```
kubectl apply -f examples/mirror-maker/kafka-mirror-maker-2.yaml
```

4. Check the status of the deployment:

```
kubectl get pods -n <my_cluster_operator_namespace>
```

*Output shows the deployment name and readiness*

NAME	READY	STATUS	RESTARTS
my-mm2-cluster-mirrormaker2-<pod_id>	1/1	Running	1

In this example, `my-mm2-cluster` is the name of the Kafka MirrorMaker 2 cluster. A pod ID identifies each created pod. By default, the deployment creates a single MirrorMaker 2 pod. `READY` shows the number of ready versus expected replicas. The deployment is successful when the `STATUS` is `Running`.

## *Additional resources*

- [Kafka MirrorMaker 2 cluster configuration](#)

### **7.5.2. List of Kafka MirrorMaker 2 cluster resources**

The following resources are created by the Cluster Operator in the Kubernetes cluster:

**<mirrormaker2\_cluster\_name>-mirrormaker2**

Name given to the following MirrorMaker 2 resources:

- StrimziPodSet that creates the MirrorMaker 2 worker node pods.
- Headless service that provides stable DNS names to the MirrorMaker 2 pods.
- Service account used by the MirrorMaker 2 pods.
- Pod disruption budget configured for the MirrorMaker 2 worker nodes.
- Network Policy managing access to the MirrorMaker 2 REST API.
- Role granting the MirrorMaker 2 worker nodes access to read their certificates and credentials.

**<mirrormaker2\_cluster\_name>-mirrormaker2-<pod\_id>**

Pods created by the MirrorMaker 2 StrimziPodSet.

**<mirrormaker2\_cluster\_name>-mirrormaker2-tls-trusted-certs**

Secret with TLS certificates.

**<mirrormaker2\_cluster\_name>-mirrormaker2-api**

Service which exposes the REST interface for managing the MirrorMaker 2 cluster.

**<mirrormaker2\_cluster\_name>-mirrormaker2-config**

ConfigMap which contains the MirrorMaker 2 ancillary configuration and is mounted as a volume by the MirrorMaker 2 pods.

**<mirrormaker2\_cluster\_name>-mirrormaker2-role**

Role binding granting the MirrorMaker 2 worker nodes access to read their certificates and credentials.

**strimzi-<namespace-name>-<mirrormaker2\_cluster\_name>-mirrormaker2-init**

Cluster role binding used by the MirrorMaker 2 cluster.

## **7.6. Deploying HTTP Bridge**

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

### **7.6.1. Deploying HTTP Bridge**

This procedure shows how to deploy a HTTP Bridge cluster to your Kubernetes cluster using the Cluster Operator.

The deployment uses a YAML file to provide the specification to create a [KafkaBridge](#) resource.

Strimzi provides [example configuration files](#). In this procedure, we use the following example file:

- [examples/bridge/kafka-bridge.yaml](#)

#### *Prerequisites*

- Cluster Operator is deployed.
- Kafka cluster is running.

This procedure assumes that the Kafka cluster was deployed using Strimzi.

#### *Procedure*

1. Edit the deployment file to configure connection details (if required).

In [examples/bridge/kafka-bridge.yaml](#), add or update the following properties as needed:

- `spec.bootstrapServers` to specify the Kafka bootstrap address.
- `spec.authentication` to specify the authentication type as `tls`, `scram-sha-256`, `scram-sha-512`, `plain`, or `custom`. See the [KafkaBridgeSpec schema properties](#) for configuration details.
- `spec.tls.trustedCertificates` to configure the TLS certificate.  
Use `[]` (an empty array) to trust the default Java CAs, or specify secrets containing trusted certificates. See the [trustedCertificates properties](#) for configuration details.

2. Deploy HTTP Bridge to your Kubernetes cluster:

```
kubectl apply -f examples/bridge/kafka-bridge.yaml
```

3. Check the status of the deployment:

```
kubectl get pods -n <my_cluster_operator_namespace>
```

*Output shows the deployment name and readiness*

NAME	READY	STATUS	RESTARTS
my-bridge-bridge-<pod_id>	1/1	Running	0

In this example, `my-bridge` is the name of the HTTP Bridge cluster. A pod ID identifies each created pod. By default, the deployment creates a single HTTP Bridge pod. `READY` shows the number of ready versus expected replicas. The deployment is successful when the `STATUS` is `Running`.

#### *Additional resources*

- [HTTP Bridge cluster configuration](#)
- [Using the HTTP Bridge](#)

## 7.6.2. Exposing the HTTP Bridge service to your local machine

Use port forwarding to expose the HTTP Bridge service to your local machine on <http://localhost:8080>.

**NOTE** Port forwarding is only suitable for development and testing purposes.

### Procedure

1. List the names of the pods in your Kubernetes cluster:

```
kubectl get pods -o name  
  
pod/kafka-consumer  
# ...  
pod/my-bridge-bridge-<pod_id>
```

2. Connect to the HTTP Bridge pod on port **8080**:

```
kubectl port-forward pod/my-bridge-bridge-<pod_id> 8080:8080 &
```

**NOTE** If port 8080 on your local machine is already in use, use an alternative HTTP port, such as **8008**.

API requests are now forwarded from port 8080 on your local machine to port 8080 in the HTTP Bridge pod.

## 7.6.3. Accessing the HTTP Bridge outside of Kubernetes

After deployment, the HTTP Bridge can only be accessed by applications running in the same Kubernetes cluster. These applications use the **<kafka\_bridge\_name>-bridge-service** service to access the API.

If you want to make the HTTP Bridge accessible to applications running outside of the Kubernetes cluster, you can expose it manually by creating one of the following features:

- **LoadBalancer** or **NodePort** type services
- **Ingress** resources (Kubernetes only)
- OpenShift routes (OpenShift only)

If you decide to create Services, use the labels from the **selector** of the **<kafka\_bridge\_name>-bridge-service** service to configure the pods to which the service will route the traffic:

```
# ...  
selector:  
  strimzi.io/cluster: kafka-bridge-name ①  
  strimzi.io/kind: KafkaBridge
```

```
#...
```

- ① Name of the HTTP Bridge custom resource in your Kubernetes cluster.

#### 7.6.4. List of HTTP Bridge cluster resources

The following resources are created by the Cluster Operator in the Kubernetes cluster:

**<bridge\_cluster\_name>-bridge**

Deployment which is in charge to create the HTTP Bridge worker node pods.

**<bridge\_cluster\_name>-bridge-service**

Service which exposes the HTTP Bridge REST interface.

**<bridge\_cluster\_name>-bridge-config**

ConfigMap which contains the HTTP Bridge ancillary configuration and is mounted as a volume by the Kafka broker pods.

**<bridge\_cluster\_name>-bridge**

Pod Disruption Budget configured for the HTTP Bridge worker nodes.

## 7.7. Alternative standalone deployment options for Strimzi operators

You can perform a standalone deployment of the Topic Operator and User Operator. Consider a standalone deployment of these operators if you are using a Kafka cluster that is not managed by the Cluster Operator.

You deploy the operators to Kubernetes, Kafka can be running outside of Kubernetes. For example, you might be using a managed Kafka service.

To connect the standalone operators to your Kafka cluster, you must set environment variables that specify the cluster address and authentication details. These variables are automatically configured when deploying the operators with the Cluster Operator.

### 7.7.1. Deploying the standalone Topic Operator

This procedure shows how to deploy the Topic Operator as a standalone component for topic management. You can use a standalone Topic Operator with a Kafka cluster that is not managed by the Cluster Operator.

Standalone deployment files are provided with Strimzi. Use the [05-Deployment-strimzi-topic-operator.yaml](#) deployment file to deploy the Topic Operator. Add or set the environment variables needed to make a connection to a Kafka cluster.

The Topic Operator watches for **KafkaTopic** resources in a single namespace. You specify the namespace to watch, and the connection to the Kafka cluster, in the Topic Operator configuration. A single Topic Operator can watch a single namespace. One namespace should be watched by only

one Topic Operator. If you want to use more than one Topic Operator, configure each of them to watch different namespaces. In this way, you can use Topic Operators with multiple Kafka clusters.

**WARNING**

Do not deploy more than one Kafka cluster into the same namespace. This causes the Topic Operator connected to each cluster to compete for the same topic resources using the same names, leading to name collisions and unpredictable behavior.

*Prerequisites*

- The standalone Topic Operator deployment files, which are included in the Strimzi [deployment files](#).
- You are running a Kafka cluster for the Topic Operator to connect to.

As long as the standalone Topic Operator is correctly configured for connection, the Kafka cluster can be running on a bare-metal environment, a virtual machine, or as a managed cloud application service.

*Procedure*

1. Edit the `env` properties in the [install/topic-operator/05-Deployment-strimzi-topic-operator.yaml](#) standalone deployment file.

*Example standalone Topic Operator deployment configuration*

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: strimzi-topic-operator
  labels:
    app: strimzi
spec:
  # ...
  template:
    # ...
    spec:
      # ...
      containers:
        - name: strimzi-topic-operator
          # ...
          env:
            - name: STRIMZI_NAMESPACE ①
              valueFrom:
                fieldRef:
                  fieldPath: metadata.namespace
            - name: STRIMZI_KAFKA_BOOTSTRAP_SERVERS ②
              value: my-kafka-bootstrap-address:9092
            - name: STRIMZI_RESOURCE_LABELS ③
              value: "strimzi.io/cluster=my-cluster"
            - name: STRIMZI_FULL_RECONCILIATION_INTERVAL_MS ④
              value: "120000"
```

```

- name: STRIMZI_LOG_LEVEL ⑤
  value: INFO
- name: STRIMZI_TLS_ENABLED ⑥
  value: "false"
- name: STRIMZI_JAVA_OPTS ⑦
  value: "-Xmx=512M -Xms=256M"
- name: STRIMZI_JAVA_SYSTEM_PROPERTIES ⑧
  value: "-Djavax.net.debug=verbose -DpropertyName=value"
- name: STRIMZI_PUBLIC_CA ⑨
  value: "false"
- name: STRIMZI_TLS_AUTH_ENABLED ⑩
  value: "false"
- name: STRIMZI_SASL_ENABLED ⑪
  value: "false"
- name: STRIMZI_SASL_USERNAME ⑫
  value: "admin"
- name: STRIMZI_SASL_PASSWORD ⑬
  value: "password"
- name: STRIMZI_SASL_MECHANISM ⑭
  value: "scram-sha-512"
- name: STRIMZI_SECURITY_PROTOCOL ⑮
  value: "SSL"
- name: STRIMZI_USE_FINALIZERS
  value: "false" ⑯

```

- ① The Kubernetes namespace for the Topic Operator to watch for [KafkaTopic](#) resources. Specify the namespace of the Kafka cluster.
- ② The host and port pair of the bootstrap broker address to discover and connect to all brokers in the Kafka cluster. Use a comma-separated list to specify two or three broker addresses in case a server is down.
- ③ The label to identify the [KafkaTopic](#) resources managed by the Topic Operator. This does not have to be the name of the Kafka cluster. It can be the label assigned to the [KafkaTopic](#) resource. If you deploy more than one Topic Operator, the labels must be unique for each. That is, the operators cannot manage the same resources.
- ④ The interval between periodic reconciliations, in milliseconds. The default is [120000](#) (2 minutes).
- ⑤ The level for printing logging messages. You can set the level to [ERROR](#), [WARNING](#), [INFO](#), [DEBUG](#), or [TRACE](#).
- ⑥ Enables TLS support for encrypted communication with the Kafka brokers.
- ⑦ (Optional) The Java options used by the JVM running the Topic Operator.
- ⑧ (Optional) The debugging ([-D](#)) options set for the Topic Operator.
- ⑨ (Optional) Skips the generation of trust store certificates if TLS is enabled through [STRIMZI\\_TLS\\_ENABLED](#). If this environment variable is enabled, the brokers must use a public trusted certificate authority for their TLS certificates. The default is [false](#).
- ⑩ (Optional) Generates key store certificates for mTLS authentication. Setting this to [false](#) disables client authentication with mTLS to the Kafka brokers. The default is [true](#).

- ⑪ (Optional) Enables SASL support for client authentication when connecting to Kafka brokers. The default is `false`.
  - ⑫ (Optional) The SASL username for client authentication. Mandatory only if SASL is enabled through `STRIMZI_SASL_ENABLED`.
  - ⑬ (Optional) The SASL password for client authentication. Mandatory only if SASL is enabled through `STRIMZI_SASL_ENABLED`.
  - ⑭ (Optional) The SASL mechanism for client authentication. Mandatory only if SASL is enabled through `STRIMZI_SASL_ENABLED`. You can set the value to `plain`, `scram-sha-256`, or `scram-sha-512`.
  - ⑮ (Optional) The security protocol used for communication with Kafka brokers. The default value is "PLAINTEXT". You can set the value to `PLAINTEXT`, `SSL`, `SASL_PLAINTEXT`, or `SASL_SSL`.
  - ⑯ Set `STRIMZI_USE_FINALIZERS` to `false` if you do not want to use finalizers to control [topic deletion](#).
2. If you want to connect to Kafka brokers that are using certificates from a public certificate authority, set `STRIMZI_PUBLIC_CA` to `true`. Set this property to `true`, for example, if you are using Amazon AWS MSK service.
  3. If you enabled mTLS with the `STRIMZI_TLS_ENABLED` environment variable, specify the keystore and truststore used to authenticate connection to the Kafka cluster.

*Example mTLS configuration*

```
# ....
env:
  - name: STRIMZI_TRUSTSTORE_LOCATION ①
    value: "/path/to/truststore.p12"
  - name: STRIMZI_TRUSTSTORE_PASSWORD ②
    value: "TRUSTSTORE-PASSWORD"
  - name: STRIMZI_KEYSTORE_LOCATION ③
    value: "/path/to/keystore.p12"
  - name: STRIMZI_KEYSTORE_PASSWORD ④
    value: "KEYSTORE-PASSWORD"
# ...
```

- ① The truststore contains the public keys of the Certificate Authorities used to sign the Kafka server certificates.
  - ② The password for accessing the truststore.
  - ③ The keystore contains the private key for mTLS authentication.
  - ④ The password for accessing the keystore.
4. If you need to configure custom SASL authentication, you can define the necessary authentication properties using the `STRIMZI_SASL_CUSTOM_CONFIG_JSON` environment variable for the standalone operator. For example, this configuration may be used for accessing a Kafka cluster in a cloud provider with a custom login module like the [Amazon MSK Library for AWS Identity and Access Management \(aws-msk iam-auth\)](#).

The property `STRIMZI_ALTERABLE_TOPIC_CONFIG` defaults to `ALL`, allowing all `.spec.config`

properties to be set in the **KafkaTopic** resource. If this setting is not suitable for a managed Kafka service, do as follows:

- If only a subset of properties is configurable, list them as comma-separated values.
- If no properties are to be configured, use **NONE**, which is equivalent to an empty property list.

**NOTE**

Only Kafka configuration properties starting with **sasl.** can be set with the **STRIMZI\_SASL\_CUSTOM\_CONFIG\_JSON** environment variable.

*Example custom SASL configuration*

```
# ....
env:
  - name: STRIMZI_SASL_ENABLED
    value: "true"
  - name: STRIMZI_SECURITY_PROTOCOL
    value: SASL_SSL
  - name: STRIMZI_SKIP_CLUSTER_CONFIG REVIEW ①
    value: "true"
  - name: STRIMZI_ALTERABLE_TOPIC_CONFIG ②
    value: compression.type, max.message.bytes,
message.timestamp.difference.max.ms, message.timestamp.type, retention.bytes,
retention.ms
  - name: STRIMZI_SASL_CUSTOM_CONFIG_JSON ③
    value: |
      {
        "sasl.mechanism": "AWS_MSK_IAM",
        "sasl.jaas.config": "software.amazon.msk.auth.iam.IAMLoginModule
required;",
        "sasl.client.callback.handler.class":
"software.amazon.msk.auth.iam.IAMClientCallbackHandler"
      }
  - name: STRIMZI_PUBLIC_CA
    value: "true"
  - name: STRIMZI_TRUSTSTORE_LOCATION
    value: /etc/pki/java/cacerts
  - name: STRIMZI_TRUSTSTORE_PASSWORD
    value: changeit
  - name: STRIMZI_KAFKA_BOOTSTRAP_SERVERS
    value: my-kafka-cluster-.kafka-serverless.us-east-1.amazonaws.com:9098
# ...
```

- ① Disables cluster configuration lookup for managed Kafka services that don't allow topic configuration changes.
- ② Defines the topic configuration properties that can be updated based on the limitations set by managed Kafka services.
- ③ Specifies the SASL properties to be set in JSON format. Only properties starting with **sasl.** are allowed.

### *Example Dockerfile with external jars*

```
FROM quay.io/stimzi/operator:0.50.0

USER root

RUN mkdir -p ${STRIMZI_HOME}/external-libs
RUN chmod +rx ${STRIMZI_HOME}/external-libs

COPY ./aws-msk-iam-auth-and-dependencies/* ${STRIMZI_HOME}/external-libs/
ENV JAVA_CLASSPATH=${STRIMZI_HOME}/external-libs/*

USER 1001
```

5. Apply the changes to the [Deployment](#) configuration to deploy the Topic Operator.
6. Check the status of the deployment:

```
kubectl get deployments
```

*Output shows the deployment name and readiness*

NAME	READY	UP-TO-DATE	AVAILABLE
stimzi-topic-operator	1/1	1	1

**READY** shows the number of replicas that are ready/expected. The deployment is successful when the **AVAILABLE** output shows **1**.

## 7.7.2. Deploying the standalone User Operator

This procedure shows how to deploy the User Operator as a standalone component for user management. You can use a standalone User Operator with a Kafka cluster that is not managed by the Cluster Operator.

A standalone deployment can operate with any Kafka cluster.

Standalone deployment files are provided with Strimzi. Use the [05-Deployment-stimzi-user-operator.yaml](#) deployment file to deploy the User Operator. Add or set the environment variables needed to make a connection to a Kafka cluster.

The User Operator watches for [KafkaUser](#) resources in a single namespace. You specify the namespace to watch, and the connection to the Kafka cluster, in the User Operator configuration. A single User Operator can watch a single namespace. One namespace should be watched by only one User Operator. If you want to use more than one User Operator, configure each of them to watch different namespaces. In this way, you can use the User Operator with multiple Kafka clusters.

### **WARNING**

Do not configure standalone User Operators that connect to different Kafka clusters to watch the same namespace. This can cause the operators to compete

for the same `KafkaUser` resources, leading to name collisions and unpredictable behavior.

#### Prerequisites

- The standalone User Operator deployment files, which are included in the Strimzi [deployment files](#).
- You are running a Kafka cluster for the User Operator to connect to.

As long as the standalone User Operator is correctly configured for connection, the Kafka cluster can be running on a bare-metal environment, a virtual machine, or as a managed cloud application service.

#### Procedure

1. Edit the following `env` properties in the `install/user-operator/05-Deployment-stimzi-user-operator.yaml` standalone deployment file.

#### Example standalone User Operator deployment configuration

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: stimzi-user-operator
  labels:
    app: stimzi
spec:
  # ...
  template:
    # ...
    spec:
      # ...
      containers:
        - name: stimzi-user-operator
          # ...
          env:
            - name: STRIMZI_NAMESPACE ①
              valueFrom:
                fieldRef:
                  fieldPath: metadata.namespace
            - name: STRIMZI_KAFKA_BOOTSTRAP_SERVERS ②
              value: my-kafka-bootstrap-address:9092
            - name: STRIMZI_CA_CERT_NAME ③
              value: my-cluster-clients-ca-cert
            - name: STRIMZI_CA_KEY_NAME ④
              value: my-cluster-clients-ca
            - name: STRIMZI_LABELS ⑤
              value: "stimzi.io/cluster=my-cluster"
            - name: STRIMZI_FULL_RECONCILIATION_INTERVAL_MS ⑥
              value: "120000"
            - name: STRIMZI_WORK_QUEUE_SIZE ⑦
              value: 10000
```

```

- name: STRIMZI_CONTROLLER_THREAD_POOL_SIZE ⑧
  value: 10
- name: STRIMZI_USER_OPERATIONS_THREAD_POOL_SIZE ⑨
  value: 4
- name: STRIMZI_LOG_LEVEL ⑩
  value: INFO
- name: STRIMZI_GC_LOG_ENABLED ⑪
  value: "true"
- name: STRIMZI_CA_VALIDITY ⑫
  value: "365"
- name: STRIMZI_CA_RENEWAL ⑬
  value: "30"
- name: STRIMZI_JAVA_OPTS ⑭
  value: "-Xmx=512M -Xms=256M"
- name: STRIMZI_JAVA_SYSTEM_PROPERTIES ⑮
  value: "-Djavax.net.debug=verbose -DpropertyName=value"
- name: STRIMZI_SECRET_PREFIX ⑯
  value: "kafka-"
- name: STRIMZI_ACLS_ADMIN_API_SUPPORTED ⑰
  value: "true"
- name: STRIMZI_MAINTENANCE_TIME_WINDOWS ⑱
  value: '* * 8-10 * * ?;* * 14-15 * * ?'
- name: STRIMZI_KAFKA_ADMIN_CLIENT_CONFIGURATION ⑲
  value: |
    default.api.timeout.ms=120000
    request.timeout.ms=60000
- name: STRIMZI_IGNORED_USERS_PATTERN ⑳
  value: "^ANONYMOUS$"

```

- ① The Kubernetes namespace for the User Operator to watch for `KafkaUser` resources. Only one namespace can be specified.
- ② The host and port pair of the bootstrap broker address to discover and connect to all brokers in the Kafka cluster. Use a comma-separated list to specify two or three broker addresses in case a server is down.
- ③ The Kubernetes `Secret` that contains the public key (`ca.crt`) value of the CA (certificate authority) that signs new user certificates for mTLS authentication.
- ④ The Kubernetes `Secret` that contains the private key (`ca.key`) value of the CA that signs new user certificates for mTLS authentication.
- ⑤ The label to identify the `KafkaUser` resources managed by the User Operator. This does not have to be the name of the Kafka cluster. It can be the label assigned to the `KafkaUser` resource. If you deploy more than one User Operator, the labels must be unique for each. That is, the operators cannot manage the same resources.
- ⑥ The interval between periodic reconciliations, in milliseconds. The default is `120000` (2 minutes).
- ⑦ The size of the controller event queue. The size of the queue should be at least as big as the maximal amount of users you expect the User Operator to operate. The default is `1024`.

- ⑧ The size of the worker pool for reconciling the users. Bigger pool might require more resources, but it will also handle more **KafkaUser** resources The default is **50**.
- ⑨ The size of the worker pool for Kafka Admin API and Kubernetes operations. Bigger pool might require more resources, but it will also handle more **KafkaUser** resources The default is **4**.
- ⑩ The level for printing logging messages. You can set the level to **ERROR, WARNING, INFO, DEBUG**, or **TRACE**.
- ⑪ Enables garbage collection (GC) logging. The default is **true**.
- ⑫ The validity period for the CA. The default is **365** days.
- ⑬ The renewal period for the CA. The renewal period is measured backwards from the expiry date of the current certificate. The default is **30** days to initiate certificate renewal before the old certificates expire.
- ⑭ (Optional) The Java options used by the JVM running the User Operator
- ⑮ (Optional) The debugging (**-D**) options set for the User Operator
- ⑯ (Optional) Prefix for the names of Kubernetes secrets created by the User Operator.
- ⑰ (Optional) Indicates whether the Kafka cluster supports management of authorization ACL rules using the Kafka Admin API. When set to **false**, the User Operator will reject all resources with **simple** authorization ACL rules. This helps to avoid unnecessary exceptions in the Kafka cluster logs. The default is **true**.
- ⑱ (Optional) Semi-colon separated list of Cron Expressions defining the maintenance time windows during which the expiring user certificates will be renewed.
- ⑲ (Optional) Configuration options for configuring the Kafka Admin client used by the User Operator in the properties format.
- ⑳ (Optional) Regular expression to configure users for which any existing ACLs, Quotas and SCRAM-SHa credentials will be ignored. This option is useful when you want to configure the User Operator to ignore users that are managed through another mechanism.

2. If you are using mTLS to connect to the Kafka cluster, specify the secrets used to authenticate connection. Otherwise, go to the next step.

#### *Example mTLS configuration*

```
# ....
env:
  - name: STRIMZI_CLUSTER_CA_CERT_SECRET_NAME ①
    value: my-cluster-cluster-ca-cert
  - name: STRIMZI_EO_KEY_SECRET_NAME ②
    value: my-cluster-entity-operator-certs
  - name: STRIMZI_EO_KEY_NAME ③
    value: entity-operator.key
  - name: STRIMZI_EO_CERT_NAME ④
    value: entity-operator.crt
# ..."
```

- ① The Kubernetes **Secret** that contains the public key (**ca.crt**) value of the CA that signs Kafka broker certificates.
- ② The Kubernetes **Secret** that contains the certificate public key (**entity-operator.crt**) and private key (**entity-operator.key**) that is used for mTLS authentication against the Kafka cluster.
- ③ The name of the private key that is used for mTLS authentication against the Kafka cluster. The default is **entity-operator.key**.
- ④ The name of the public key that is used for mTLS authentication against the Kafka cluster. The default is **entity-operator.crt**.

### 3. Deploy the User Operator.

```
kubectl create -f install/user-operator
```

### 4. Check the status of the deployment:

```
kubectl get deployments
```

*Output shows the deployment name and readiness*

NAME	READY	UP-TO-DATE	AVAILABLE
strimzi-user-operator	1/1	1	1

**READY** shows the number of replicas that are ready/expected. The deployment is successful when the **AVAILABLE** output shows **1**.

# Chapter 8. Deploying Strimzi from OperatorHub.io

OperatorHub.io is a catalog of Kubernetes operators sourced from multiple providers. It offers you an alternative way to install a stable version of Strimzi.

The [Operator Lifecycle Manager](#) is used for the installation and management of all operators published on OperatorHub.io. [Operator Lifecycle Manager](#) is a prerequisite for installing the Strimzi Kafka operator

To install Strimzi, locate *Strimzi* from [OperatorHub.io](#), and follow the instructions provided to deploy the Cluster Operator. After you have deployed the Cluster Operator, you can deploy Strimzi components using custom resources. For example, you can deploy the [Kafka](#) custom resource, and the installed Cluster Operator will create a Kafka cluster.

**WARNING** While the Cluster Operator can be configured to watch multiple namespaces, each watched namespace should contain only one instance of a specific component type, such as one Kafka cluster, to avoid conflicts.

Upgrades between versions might include manual steps. Always read the release notes before upgrading.

For information on upgrades, see [Upgrading Strimzi](#).

**WARNING** Make sure you use the appropriate update channel. Installing Strimzi from the default *stable* channel is generally safe. However, we do not recommend enabling *automatic* OLM updates on the stable channel. An automatic upgrade will skip any necessary steps prior to upgrade. Use automatic upgrades only on version-specific channels.

# Chapter 9. Deploying Strimzi using Helm

Helm charts are used to package, configure, and deploy Kubernetes resources. Strimzi provides a Helm chart to deploy the Cluster Operator.

After you have deployed the Cluster Operator this way, you can deploy Strimzi components using custom resources. For example, you can deploy the Kafka custom resource, and the installed Cluster Operator will create a Kafka cluster.

**WARNING**

While the Cluster Operator can be configured to watch multiple namespaces, each watched namespace should contain only one instance of a specific component type, such as one Kafka cluster, to avoid conflicts.

For information on upgrades, see [Upgrading Strimzi](#).

*Prerequisites*

- The Helm client must be installed on a local machine.

*Procedure*

1. Install the Strimzi Cluster Operator using the Helm command line tool:

```
helm install strimzi-cluster-operator oci://quay.io/stimzi-helm/stimzi-kafka-operator
```

Alternatively, you can use parameter values to install a specific version of the Cluster Operator or specify any changes to the default configuration.

*Example configuration that installs a specific version of the Cluster Operator and changes the number of replicas*

```
helm install strimzi-cluster-operator --set replicas=2 --version 0.35.0  
oci://quay.io/stimzi-helm/stimzi-kafka-operator
```

2. Verify that the Cluster Operator has been deployed successfully using the Helm command line tool:

```
helm ls
```

3. [Deploy Kafka and other Kafka components](#) using custom resources.

# Chapter 10. Feature gates

Strimzi operators use feature gates to enable or disable specific features and functions. Enabling a feature gate alters the behavior of the associated operator, introducing the corresponding feature to your Strimzi deployment.

The purpose of feature gates is to facilitate the trial and testing of a feature before it is fully adopted. The state (enabled or disabled) of a feature gate may vary by default, depending on its maturity level.

As a feature gate graduates and reaches General Availability (GA), it transitions to an enabled state by default and becomes a permanent part of the Strimzi deployment. A feature gate at the GA stage cannot be disabled.

The supported feature gates are applicable to all Strimzi operators. While a particular feature gate might be used by one operator and ignored by the others, it can still be configured in all operators. When deploying the User Operator and Topic Operator within the context of the [Kafka](#) custom resource, the Cluster Operator automatically propagates the feature gates configuration to them. When the User Operator and Topic Operator are deployed standalone, without a Cluster Operator available to configure the feature gates, they must be directly configured within their deployments.

## 10.1. Feature gate releases

Feature gates have three stages of maturity:

### Alpha

Alpha stage features are disabled by default.

They might be experimental or unstable, subject to change, or not sufficiently tested for production use.

### Beta

Beta stage features are enabled by default, but they can be disabled if needed.

They are well tested and their functionality is not likely to change.

### General Availability (GA)

GA features are always enabled and cannot be disabled.

They are stable and should not change in the future. Feature gates might be removed when they reach GA, which means that the feature was incorporated into the Strimzi core features.

Alpha and beta stage features are removed if they do not prove to be useful. When a feature gate reaches GA, it is permanently enabled and can no longer be disabled. The following table shows the maturity of the feature gates introduced across Strimzi versions.

*Table 10. Feature gate maturity across Strimzi versions*

Feature gate	Alpha	Beta	GA
<a href="#">ControlPlaneListener</a>	0.23	0.27	0.32
<a href="#">ServiceAccountPatching</a>	0.24	0.27	0.30

Feature gate	Alpha	Beta	GA
UseStrimziPodSets	0.28	0.30	0.35
UseKRaft	0.29	0.40	0.42
StableConnectIdentities	0.34	0.37	0.39
KafkaNodePools	0.36	0.39	0.41
UnidirectionalTopicOperator	0.36	0.39	0.41
ContinueReconciliationOnManualRollingUpdateFailure	0.41	0.44	0.46
ServerSideApplyPhase1	0.48	0.51 (planned)	0.54 (planned)
UseConnectBuildWithBuildah	0.49	0.52 (planned)	0.55 (planned)

## 10.2. Graduated feature gates (GA)

For information on feature gates that have reached GA, see the documentation for the Strimzi version in which they were introduced.

## 10.3. Stable feature gates (Beta)

Stable feature gates have reached a beta level of maturity, and are generally enabled by default for all users. Stable feature gates are production-ready, but they can still be disabled. Currently, there are no beta level feature gates.

## 10.4. Early access feature gates (Alpha)

Early access feature gates have not yet reached the beta stage, and are disabled by default. An early access feature gate provides an opportunity for assessment before its functionality is permanently incorporated into Strimzi. Currently, there are no alpha level feature gates.

### 10.4.1. `ServerSideApplyPhase1` feature gate

The `ServerSideApplyPhase1` feature gate has a default state of *disabled*.

When enabled, the Cluster Operator uses Server-Side Apply (SSA) for creating and updating certain resources. SSA is a Kubernetes feature that provides better tracking of field ownership and more controlled resource updates. In Phase 1, Server-Side Apply is used for the following resources:

- `PersistentVolumeClaim`
- `ConfigMap`
- `Ingress`
- `ServiceAccount`

- **Service**

All other resources continue to be reconciled using the existing mechanism - client-side apply. If users change the fields of operator-managed resources, the operator reverts these changes.

With SSA, the Cluster Operator modifies only the fields it manages. User changes to the other fields are preserved as long as they don't conflict. On the first update attempt, the Cluster Operator sets `force = false` when applying changes, respecting user changes. If a conflict occurs in a managed field, the update is retried with `force = true`. The conflict and the switch to forced apply are logged by the Cluster Operator.

Ownership of individual fields is tracked through the `.metadata.managedFields` property of the resource being updated.

To enable the `ServerSideApplyPhase1` feature gate, specify `+ServerSideApplyPhase1` in the `STRIMZI_FEATURE_GATES` environment variable in the Cluster Operator configuration.

#### 10.4.2. **UseConnectBuildWithBuildah** feature gate

The `UseConnectBuildWithBuildah` feature gate has a default state of *disabled*.

When enabled, the Connect Build feature uses Buildah instead of Kaniko to build Kafka Connect container images. On Kubernetes, either Kaniko or Buildah is used depending on this feature gate. On OpenShift, the build mechanism remains unchanged. Builds continue to use the OpenShift Build API. Using Buildah does not change the overall build behavior. The difference is that you can specify additional options in the `build` section of the `KafkaConnect` custom resource. These options are grouped as follows:

- Build phase options (`additionalBuildOptions`)
- Push phase options (`additionalPushOptions`)

To enable the `UseConnectBuildWithBuildah` feature gate, specify `+UseConnectBuildWithBuildah` in the `STRIMZI_FEATURE_GATES` environment variable in the Cluster Operator configuration.

## 10.5. Enabling feature gates

To modify a feature gate's default state, use the `STRIMZI_FEATURE_GATES` environment variable in the operator's configuration. You can modify multiple feature gates using this single environment variable. Specify a comma-separated list of feature gate names and prefixes. A `+` prefix enables the feature gate and a `-` prefix disables it.

*Example feature gate configuration that enables FeatureGate1 and disables FeatureGate2*

```
env:  
  - name: STRIMZI_FEATURE_GATES  
    value: +FeatureGate1,-FeatureGate2
```

# Chapter 11. Configuring a deployment

Configure and manage a Strimzi deployment to your precise needs using Strimzi custom resources. Strimzi provides example custom resources with each release, allowing you to configure and create instances of supported Kafka components.

Use custom resources to configure and create instances of the following components:

- Kafka clusters
- Kafka Connect clusters
- Kafka MirrorMaker
- HTTP Bridge
- Cruise Control

You can use configuration to manage your instances or modify your deployment to introduce additional features. New features are sometimes introduced through feature gates, which are controlled through operator configuration.

The [Strimzi Custom Resource API Reference](#) describes the properties you can use in your configuration.

## *Centralizing configuration*

For key configuration areas, such as metrics, logging, and external Kafka Connect connector settings, you can centralize management as follows:

- [Using ConfigMap resources to incorporate configuration.](#)
- [Using configuration providers to load configuration from external sources.](#)

We recommend configuration providers for securely supplying Kafka Connect connector credentials.

## *TLS certificate management*

When deploying Kafka, the Cluster Operator automatically sets up and renews TLS certificates to enable encryption and authentication within your cluster. If required, you can manually renew the cluster and clients CA certificates before their renewal period starts. You can also replace the keys used by the cluster and clients CA certificates.

For more information, see [Renewing CA certificates manually](#) and [Replacing private keys](#).

**NOTE** Labels applied to a custom resource are also applied to the Kubernetes resources making up its cluster. This provides a convenient mechanism for resources to be labeled as required.

## 11.1. Using example configuration files

Further enhance your deployment by incorporating additional supported configuration. Example

configuration files are included in the Strimzi [deployment files](#). You can also access the example files directly from the [examples directory](#).

The example files include only the essential properties and values for custom resources by default. You can download and apply the examples using the `kubectl` command-line tool. The examples can serve as a starting point when building your own Kafka component configuration for deployment.

**NOTE**

If you installed Strimzi using the Operator, you can still download the example files and use them to upload configuration.

The release artifacts include an [examples](#) directory that contains the configuration examples.

*Example configuration files provided with Strimzi*

```
examples
├── user ①
├── topic ②
├── security ③
│   ├── tls-auth
│   ├── scram-sha-512-auth
│   └── keycloak-authorization
├── mirror-maker ④
├── metrics ⑤
├── kafka ⑥
├── cruise-control ⑦
├── connect ⑧
└── bridge ⑨
```

- ① [KafkaUser](#) custom resource configuration, which is managed by the User Operator.
- ② [KafkaTopic](#) custom resource configuration, which is managed by Topic Operator.
- ③ Authentication and authorization configuration for Kafka components. Includes example configuration for TLS and SCRAM-SHA-512 authentication. The Keycloak examples include a Keycloak realm specification and two [Kafka](#) custom resources with `type: custom` definitions for using OAuth 2.0 authentication and Keycloak authorization with or without metrics enabled.
- ④ [KafkaMirrorMaker2](#) custom resource configurations for a deployment of MirrorMaker 2. Includes example configuration for replication policy and synchronization frequency.
- ⑤ [Metrics configuration](#), including Prometheus installation and Grafana dashboard files.
- ⑥ [Kafka](#) and [KafkaNodePool](#) custom resource configurations for a deployment of Kafka clusters that use KRaft mode. Includes example configuration for an ephemeral or persistent single or multi-node deployment.
- ⑦ [Kafka](#) and [KafkaRebalance](#) configurations for deploying and using Cruise Control to manage clusters. [Kafka](#) configuration examples enable auto-rebalancing on scaling events and set default optimization goals. [KafkaRebalance](#) configuration examples set proposal-specific optimization goals and generate optimization proposals in various supported modes.
- ⑧ [KafkaConnect](#) and [KafkaConnector](#) custom resource configuration for a deployment of Kafka Connect. Includes example configurations for a single or multi-node deployment.

<sup>⑨</sup> KafkaBridge custom resource configuration for a deployment of HTTP Bridge.

## 11.2. Configuring Kafka

Configure your Kafka deployment by updating the `spec` properties of the `Kafka` custom resource.

Start with a minimal configuration that defines the core requirements for running a Kafka cluster. Extend it with optional settings to support the following:

- Securing client connections using TLS and authentication
- Configuring authorization
- Configuring broker behavior for durability and availability
- Capturing consumer lag metrics using Kafka Exporter
- Kafka cluster balancing and resource distribution using Cruise Control
- Controlling throughput and storage usage using quotas
- Recovering from broker storage issues
- Using node pools to define broker groups
- [Common configuration](#), including setting resource limits and requests (recommended), JVM tuning, and metrics
- [Logging configuration](#)

Certain settings, though optional, are recommended for production deployments. These include security, resource allocation, and broker replication settings.

For details of all available properties, see the [Strimzi Custom Resource API Reference](#).

**NOTE**

Kafka clusters use node pools. At least one `KafkaNodePool` resource is required. Node pools can define controller and broker roles separately or in combination. Using separate node pools is recommended for most production deployments. For details, see [Configuring node pools](#).

### 11.2.1. Minimal configuration for a Kafka cluster

A Kafka cluster requires a `Kafka` resource and at least one `KafkaNodePool` resource. The `Kafka` resource defines cluster-wide configuration such as listeners and broker settings.

*Minimal configuration for Kafka*

```
# Basic configuration (required)
apiVersion: kafka.strimzi.io/v1
kind: Kafka
metadata:
  name: my-cluster
# Deployment specifications
spec:
  kafka:
```

```

# Broker configuration for replication (recommended)
config:
  offsets.topic.replication.factor: 3
  transaction.state.log.replication.factor: 3
  transaction.state.log.min_isr: 2
  default.replication.factor: 3
  min.insync.replicas: 2
# Listener configuration (required)
listeners:
  - name: plain
    port: 9092
    type: internal
    tls: false
  - name: tls
    port: 9093
    type: internal
    tls: true
# Kafka version (recommended)
version: 4.1.1
# KRaft metadata version (recommended)
metadataVersion: 4.1
# Entity Operator (recommended)
entityOperator:
  topicOperator: {}
  userOperator: {}

```

- `spec.kafka.config` defines broker-level configuration. In this example, replication and ISR settings are configured to provide durable replication for internal topics.
- `spec.kafka.listeners` defines how clients connect to the Kafka cluster. At least one listener is required.
- `spec.kafka.version` specifies the Kafka version for the cluster. Change this value only by following the [Kafka upgrade procedure](#).
- `spec.kafka.metadataVersion` specifies the metadata version for KRaft-based clusters. The value must be compatible with the Kafka version.
- `spec.entityOperator` deploys the Topic Operator and User Operator using default settings. For more information, see [Configuring the Entity Operator](#).

**WARNING**

If you remove the `min.insync.replicas` property from `spec.kafka.config`, the Cluster Operator forces Kafka to fall back to the default value of `1`. To ensure durability, explicitly set `min.insync.replicas` to a value greater than `1`.

### 11.2.2. Securing Kafka client connections

Secure Kafka client connections by configuring Kafka listeners with TLS encryption and client authentication. Listeners control how clients connect to the Kafka cluster from inside or outside the Kubernetes cluster.

Internal listeners allow for in-cluster access. To allow access from outside the Kubernetes cluster, configure an external listener. Add TLS encryption and authentication as required by your access and security requirements.

Listener options include:

- **name**: Identifies the listener. The listener name must be unique within the Kafka cluster.
- **port**: Port used by Kafka brokers.
  - Must be **9092** or higher.
  - Ports **9404** and **9999** are reserved for Prometheus and JMX.
- **type**: How the listener is exposed.
  - Internal types include **internal** and **cluster-ip**.
  - External types include **route** (OpenShift only), **loadbalancer**, **nodeport**, and **ingress**.
- **tls**: Enables TLS encryption. TLS is required for **route** and **ingress** listener types.
- **authentication**: Client authentication mechanism.
  - Supported types include TLS (mTLS), SCRAM-SHA-512, and custom authentication.
- **configuration**: Listener-specific settings.
  - Examples include **useServiceDnsDomain** for **internal** and **cluster-ip** listeners, and **host** for **route** and **ingress** listeners.
- **configuration.brokerCertChainAndKey**: Uses a server certificate managed by an external certificate authority.

Listener configuration controls how clients connect to Kafka. Client credentials and authorization are configured separately.

**NOTE**

Depending on the listener type, the port number might not be the same as the port number that connects Kafka clients.

The following examples show listener configuration to enable TLS and authentication.

### Internal listener without TLS

```
spec:  
  kafka:  
    # Listener configuration (required)  
    listeners:  
      - name: plain  
        port: 9092  
        type: internal  
        tls: false
```

- **spec.kafka.listeners[].type: internal** exposes Kafka only inside the Kubernetes cluster.
- **spec.kafka.listeners[].tls: false** disables TLS encryption.

## Internal listener with TLS encryption and mTLS authentication

```
spec:  
  kafka:  
    # Listener configuration (required)  
    listeners:  
      - name: tls  
        port: 9093  
        type: internal  
        tls: true  
        authentication:  
          type: tls
```

- `spec.kafka.listeners[].tls: true` enables TLS encryption.
- `spec.kafka.listeners[].authentication.type: tls` enables mutual TLS (mTLS) authentication. Clients must present a trusted client certificate when connecting.

## External listener with TLS

The listener type determines how Kafka is exposed outside the Kubernetes cluster. For example, on OpenShift you can use the `route` listener type.

```
spec:  
  kafka:  
    # Listener configuration (required)  
    listeners:  
      - name: external  
        port: 9094  
        type: route  
        tls: true
```

- `spec.kafka.listeners[].type: route` exposes Kafka outside the cluster using OpenShift routes.
- `spec.kafka.listeners[].tls: true` is required for `route` listeners.

## Using externally managed listener certificates

By default, Strimzi manages listener certificates when TLS is enabled. If you use an external certificate authority (CA), you can configure Strimzi to use a listener certificate and private key stored in a `Secret`.

```
spec:  
  kafka:  
    # Listener configuration (required)  
    listeners:  
      - name: external  
        port: 9094  
        type: route
```

```
  tls: true
  configuration:
    brokerCertChainAndKey:
      secretName: my-secret
      certificate: my-certificate.crt
      key: my-key.key
```

- `spec.kafka.listeners[].configuration.brokerCertChainAndKey` references a `Secret` containing a server certificate and private key.
- Use this option when listener certificates are managed outside Strimzi.

For step-by-step procedures and examples for securing client access, see [Securing access to a Kafka cluster](#).

For details of listener configuration options, see the [Kafka schema reference](#) and [GenericKafkaListener schema reference](#).

### 11.2.3. Configuring authorization

Control what authenticated clients can do in a Kafka cluster by enabling authorization on Kafka brokers. Authorization is configured in the `Kafka` custom resource and enforced by the Kafka broker.

Authorization is separate from listener authentication. Authentication verifies a client identity. Authorization controls which operations the client is allowed to perform.

Kafka supports the following authorization types:

- `simple`
- `custom`

The `simple` authorization type uses the Kafka `StandardAuthorizer` plugin.

The following examples show how to enable authorization on Kafka brokers.

#### Enabling simple authorization

Simple authorization enables ACL-based access control using Kafka's built-in authorizer.

```
spec:
  kafka:
    # Authorization (optional)
    authorization:
      type: simple
```

- `spec.kafka.authorization.type: simple` enables ACL-based authorization using the Kafka `StandardAuthorizer`.
- Access to Kafka resources is controlled using ACL rules.

## Configuring super users

You can define super users that are allowed to perform all operations, regardless of ACL rules. Use super users for Kafka administrator identities and internal components that must always be able to operate.

```
spec:  
  kafka:  
    # Authorization (optional)  
    authorization:  
      type: simple  
      superUsers:  
        - CN=kafka-admin  
        - my-team-admin
```

- `spec.kafka.authorization.superUsers` defines super user identities.
- The identity format must match the principal used by your authentication mechanism.

The `CN=` prefix shown in the example applies to mTLS authentication, where the principal is derived from the certificate subject. Other authentication mechanisms, such as SASL-based authentication, use different principal formats.

## Enabling custom authorization

Custom authorization allows the use of a custom authorizer implementation. Use custom authorization when you need to integrate Kafka with an external authorization system.

```
spec:  
  kafka:  
    # Authorization (optional)  
    authorization:  
      type: custom  
      authorizerClass: com.example.CustomAuthorizer
```

- `spec.kafka.authorization.type: custom` enables a custom authorization implementation.
- `spec.kafka.authorization.authorizerClass` specifies the fully qualified class name of the custom authorizer.

Authorization configuration enables enforcement of access control on the Kafka broker. User identities, credentials, and ACL rules are configured separately.

For step-by-step procedures and examples for configuring authorized access, see [Securing access to a Kafka cluster](#).

For details of authorization configuration options, see the [KafkaAuthorizationSimple schema reference](#) and [KafkaAuthorizationCustom schema reference](#).

## 11.2.4. Configuring Kafka broker operation

Control how Kafka brokers operate by configuring broker properties in the [Kafka](#) custom resource. Broker configuration affects availability and performance of the cluster.

Broker configuration is applied using the `spec.kafka.config` property. Only Kafka configuration properties that are not managed directly by Strimzi can be set. Strimzi validates and applies these properties during reconciliation.

Use broker configuration to:

- Replicate topics for high availability
- Improve throughput or request handling
- Optimize performance for high-latency or high-bandwidth networks
- Control disk usage and log retention behavior
- Reduce unnecessary rebalances or leadership changes

Broker configuration options must be used in the correct combinations to meet your availability and fault-tolerance requirements. Kafka provides many additional broker properties for more granular configuration. However, certain properties are managed directly by Strimzi and cannot be set within this config property. For more information, see the [KafkaCluster Spec schema reference](#).

Broker tuning is workload-dependent. Test changes in a non-production environment before applying them to a production cluster. A practical approach is to adjust configuration incrementally and monitor the impact using broker and client metrics. For detailed guidance on selecting and tuning broker configuration properties, see [Kafka broker configuration tuning](#).

### Adding broker configuration

Kafka broker configuration is defined as key-value pairs under `spec.kafka.config`.

```
spec:  
  kafka:  
    # Kafka configuration (recommended)  
    config:  
      offsets.topic.replication.factor: 3  
      transaction.state.log.replication.factor: 3  
      transaction.state.log.min_isr: 2  
      default.replication.factor: 3  
      min.insync.replicas: 2
```

- `spec.kafka.config` defines Kafka broker properties as key-value pairs.
- `offsets.topic.replication.factor` sets the replication factor for the consumer offsets topic.
- `transaction.state.log.replication.factor` sets the replication factor for the transaction state log.
- `transaction.state.log.min_isr` sets the minimum number of in-sync replicas (ISR) required for

the transaction state log.

- `default.replication.factor` sets the default replication factor for topics that do not specify one.
- `min.insync.replicas` sets the minimum number of in-sync replicas (ISR) required to acknowledge writes.

**WARNING**

If you remove the `min.insync.replicas` property from `spec.kafka.config`, the Cluster Operator forces Kafka to fall back to the default value of `1`. To ensure fault tolerance, explicitly set `min.insync.replicas` to a value greater than `1`.

### 11.2.5. Kafka cluster balancing with Cruise Control

Automate Kafka cluster balancing by enabling Cruise Control. Cruise Control monitors broker workloads and helps redistribute partitions to keep the cluster balanced as usage changes.

Enable Cruise Control when you need to do the following:

- Rebalance partitions across brokers
- Redistribute replicas across JBOD disks
- Support broker scaling and maintenance operations

Enable Cruise Control by adding a `cruiseControl` section to the `Kafka` custom resource.

```
spec:  
  # Cruise Control (optional)  
  cruiseControl:  
    # Configuration for balancing goals and capacity limits  
    # ...
```

For more information on how to configure and use Cruise Control, see:

- [Using Cruise Control for cluster rebalancing](#)
- [Using Cruise Control to modify topic replication factor](#)
- [Using Cruise Control to reassign partitions on JBOD disks](#)

### 11.2.6. Capturing consumer lag metrics

Capture consumer group lag metrics to monitor whether consumers are keeping up with message production. Consumer lag metrics help you detect slow or stalled consumers and diagnose processing issues.

Kafka Exporter exposes consumer group lag metrics that are not available through broker JMX metrics. These metrics are typically scraped by Prometheus and visualized in monitoring dashboards.

Enable Kafka Exporter when you need to do the following:

- Monitor consumer group lag
- Track consumer progress over time
- Alert on slow consumers

Enable Kafka Exporter by adding a `kafkaExporter` section to the `Kafka` custom resource.

```
spec:
  # Kafka Exporter (optional)
  kafkaExporter:
    # Configuration for metrics selection and exposure
    # ...
```

For configuration details, metrics, and dashboards, see [Consumer lag monitoring](#).

### 11.2.7. Choosing a quota plugin

Control throughput and protect broker storage by applying quotas to Kafka brokers. Kafka supports the following quota plugin types:

- `strimzi`
- `kafka`

Only one quota plugin can be enabled at a time. The built-in `kafka` plugin is enabled by default. Enabling the `strimzi` plugin disables the built-in plugin.

#### When to use the `strimzi` quota plugin

Use the `strimzi` quota plugin to protect the Kafka cluster as a whole using aggregate limits.

The `strimzi` plugin is suitable when you need to:

- Prevent brokers from running out of disk space (limits apply to each disk volume individually)
- Apply total throughput limits across all clients
- Dynamically share throughput limits between active clients
- Exclude specific users from quota enforcement

The `strimzi` plugin distributes the total throughput limit dynamically. For example, if you set a 40 MBps producer limit, it is not statically divided. If one producer is using only 10 MBps, the remaining 30 MBps is automatically available to other producers.

**NOTE**

With the `strimzi` plugin, you see only aggregated quota metrics, not per-client metrics.

#### When to use the `kafka` quota plugin

Use the `kafka` quota plugin to control individual client behavior using static limits.

The [kafka](#) plugin is suitable when you need to:

- Apply per-broker and per-user limits (broker-level limits can be overridden by [user-specific quotas](#))
- Control CPU usage for client requests
- Limit the rate of mutations accepted for create topic, create partition, and delete topic requests

Default quotas apply to all users, including internal components.

### 11.2.8. Setting throughput and storage limits on brokers

Configure throughput and storage limits on Kafka brokers by enabling a quota plugin and defining limits in the [Kafka](#) custom resource.

#### Prerequisites

- The Cluster Operator that manages the Kafka cluster is running.

#### Procedure

1. Add quota configuration to the [quotas](#) section of the [Kafka](#) resource. Configure either the [strimzi](#) or [kafka](#) quota plugin.

Example [strimzi](#) quota plugin configuration:

```
apiVersion: kafka.strimzi.io/v1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    quotas:
      type: strimzi
      producerByteRate: 1000000
      consumerByteRate: 1000000
      minAvailableBytesPerVolume: 500000000000
      excludedPrincipals:
        - my-user
```

- [producerByteRate](#) and [consumerByteRate](#) set total throughput limits across all clients.
- [minAvailableBytesPerVolume](#) protects broker disk capacity by throttling producers when available storage drops below the configured byte threshold on any volume. You can also use [minAvailableRatioPerVolume](#) to set the threshold as a percentage of total volume size.
- [excludedPrincipals](#) excludes specified users from quota enforcement.

[minAvailableBytesPerVolume](#) and [minAvailableRatioPerVolume](#) are mutually exclusive. Configure only one of these properties.

Example `kafka` quota plugin configuration:

```
apiVersion: kafka.strimzi.io/v1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    quotas:
      type: kafka
      producerByteRate: 1000000
      consumerByteRate: 1000000
      requestPercentage: 55
      controllerMutationRate: 50
```

- `producerByteRate` and `consumerByteRate` apply throughput limits per user and per broker.
- `requestPercentage` limits CPU usage for client requests.
- `controllerMutationRate` limits the number of create topic, create partition, and delete topic requests per second.

**NOTE**

Default `kafka` quotas apply to internal users, including the Topic Operator and Cruise Control. To prevent the Topic Operator from being throttled, monitor controller metrics and explicitly set a `controllerMutationRate` that accommodates your topic operations. For Cruise Control and its metrics reporter, ensure sufficient throughput for rebalances. A minimum rate of at least 1 KB/s is recommended for producer and consumer quotas in small clusters (three brokers), which should be increased for larger or more active clusters.

2. Apply the changes to the `Kafka` configuration.

**NOTE**

Additional quota properties are available for both quota plugins. For a complete list of supported options, see the [plugin documentation](#).

*Additional resources*

- [QuotasPluginStrimzi schema reference](#)
- [QuotasPluginKafka schema reference](#)

### 11.2.9. Recovering from broker storage issues

Broker storage issues typically present as one or more of the following symptoms:

- Broker pods repeatedly restarting or failing to start
- Errors indicating full disks or insufficient storage
- Failed volume mounts or PVC-related errors

- Partitions remaining under-replicated due to unavailable brokers

## Recovery approach

To address these issues, consider the following actions to preserve broker data and minimize the risk of data loss:

- Free disk space or increase available storage capacity.
- Review topic retention and compaction settings to reduce disk usage.
- Review quota settings to limit producer throughput and prevent disks from filling too quickly.

## Recovering persistent volumes

If the issue involves persistent volumes or PVCs, such as failed mounts or unavailable volumes, see the information on [cluster recovery from persistent volumes](#).

## Replacing brokers

If a broker cannot be recovered and its storage must be reset, replace the broker by deleting its pod and associated PVC.

This is a destructive operation that can result in permanent data loss. Use this option only when non-destructive recovery actions are not possible.

For more information, see [Deleting Kafka nodes using annotations](#).

When recovery actions are complete, confirm that broker pods are stable and partitions are fully replicated, and continue to monitor logs and storage usage to ensure ongoing health.

### *Additional resources*

- [Storage considerations for brokers](#)
- [Choosing a quota plugin](#)
- [Setting throughput and storage limits on brokers](#)

### 11.2.10. Deleting Kafka nodes using annotations

This procedure describes how to delete an existing Kafka node by using a Kubernetes annotation. Deleting a Kafka node consists of deleting both the **Pod** on which the Kafka broker is running and the related **PersistentVolumeClaim** (if the cluster was deployed with persistent storage). After deletion, the **Pod** and its related **PersistentVolumeClaim** are recreated automatically.

Use this procedure only after attempting non-destructive recovery actions. For guidance on recovery options, see [Recovering from broker storage issues](#).

#### **WARNING**

Deleting a **PersistentVolumeClaim** can cause permanent data loss and the availability of your cluster cannot be guaranteed. The following procedure should only be performed if you have encountered storage issues.

### *Prerequisites*

- The Cluster Operator that manages the Kafka cluster is running.

### Procedure

1. Find the name of the **Pod** that you want to delete.

Kafka broker pods are named `<cluster_name>-kafka-<index_number>`, where `<index_number>` starts at zero and ends at the total number of replicas minus one. For example, `my-cluster-kafka-0`.

2. Use `kubectl annotate` to annotate the **Pod** resource in Kubernetes:

```
kubectl annotate pod <cluster_name>-kafka-<index_number> strimzi.io/delete-pod-and-pvc="true"
```

3. Wait for the next reconciliation. The annotated pod and its associated persistent volume claim are deleted and then recreated automatically.

## 11.3. Configuring node pools

Update the `spec` properties of the `KafkaNodePool` custom resource to configure a node pool deployment. A node pool is a distinct group of Kafka nodes within a Kafka cluster. The `strimzi.io/cluster` metadata label identifies the name of the Kafka custom resource the pool belongs to.

Node pool configuration must define:

- Node roles within the cluster
- Number of replica nodes
- Storage specifications

The `.spec.roles` property defines whether the nodes in the pool act as controllers, brokers, or both.

Other optional properties may also be set in node pools:

- `resources` to specify memory and cpu requests and limits
- `template` to specify custom configuration for pods and other Kubernetes resources
- `jvmOptions` to specify custom JVM configuration for heap size, runtime and other options

The relationship between `Kafka` and `KafkaNodePool` resources is as follows:

- `Kafka` resources represent the configuration for all nodes in a Kafka cluster.
- `KafkaNodePool` resources represent the configuration for nodes only in the node pool.

If a configuration property is not specified in `KafkaNodePool`, it is inherited from the `Kafka` resource. Configuration specified in the `KafkaNodePool` resource takes precedence if set in both resources. For example, if both the node pool and Kafka configuration includes `jvmOptions`, the values specified in the node pool configuration are used. When `-Xmx: 1024m` is set in `KafkaNodePool.spec.jvmOptions` and `-Xms: 512m` is set in `Kafka.spec.kafka.jvmOptions`, the node uses the value from its node pool configuration.

The same properties defined in the template sections in `Kafka` and `KafkaNodePool` custom resources are not combined. To clarify, if `KafkaNodePool.spec.template` includes only `podSet.metadata.labels`, and `Kafka.spec.kafka.template` includes `podSet.metadata.annotations` and `pod.metadata.labels`, the `pod.metadata.labels` template values from the Kafka configuration are used. But the `podSet.metadata.annotations` template values are ignored since there is a `podSet` template value already present in the node pool configuration.

For a deeper understanding of the node pool configuration options, refer to the [Strimzi Custom Resource API Reference](#).

*Example configuration for a node pool in a cluster using KRaft mode*

```
# Basic configuration (required)
apiVersion: kafka.strimzi.io/v1
kind: KafkaNodePool
metadata:
  name: kraft-dual-role ①
  labels:
    strimzi.io/cluster: my-cluster ②
# Node pool specifications
spec:
  # Replicas (required)
  replicas: 3 ③
  # Roles (required)
  roles: ④
    - controller
    - broker
  # Storage configuration (required)
  storage: ⑤
    type: jbod
    volumes:
      - id: 0
        type: persistent-claim
        size: 100Gi
        deleteClaim: false
  # Resources requests and limits (recommended)
  resources: ⑥
    requests:
      memory: 64Gi
      cpu: "8"
    limits:
      memory: 64Gi
      cpu: "12"
```

① Unique name for the node pool.

② The Kafka cluster the node pool belongs to. A node pool can only belong to a single cluster.

③ Number of replicas for the nodes.

④ Roles for the nodes in the node pool. In this example, the nodes have dual roles as controllers and brokers.

- ⑤ Storage specification for the nodes.
- ⑥ Requests for reservation of supported resources, currently `cpu` and `memory`, and limits to specify the maximum resources that can be consumed.

**NOTE**

The configuration for the `Kafka` resource must be suitable for KRaft mode. Currently, KRaft mode has [a number of limitations](#).

### 11.3.1. Assigning IDs to node pools for scaling operations

This procedure describes how to use annotations for advanced node ID handling by the Cluster Operator when performing scaling operations on node pools. You specify the node IDs to use, rather than the Cluster Operator using the next ID in sequence. Management of node IDs in this way gives greater control.

To add a range of IDs, you assign the following annotations to the `KafkaNodePool` resource:

- `strimzi.io/next-node-ids` to add a range of IDs that are used for new brokers
- `strimzi.io/remove-node-ids` to add a range of IDs for removing existing brokers

You can specify an array of individual node IDs, ID ranges, or a combination of both. For example, you can specify the following range of IDs: `[0, 1, 2, 10-20, 30]` for scaling up the Kafka node pool. This format allows you to specify a combination of individual node IDs (`0, 1, 2, 30`) as well as a range of IDs (`10-20`).

In a typical scenario, you might specify a range of IDs for scaling up and a single node ID to remove a specific node when scaling down.

In this procedure, we add the scaling annotations to node pools as follows:

- `pool-a` is assigned a range of IDs for scaling up
- `pool-b` is assigned a range of IDs for scaling down

During the scaling operation, IDs are used as follows:

- Scale up picks up the lowest available ID in the range for the new node.
- Scale down removes the node with the highest available ID in the range.

If there are gaps in the sequence of node IDs assigned in the node pool, the next node to be added is assigned an ID that fills the gap.

The annotations don't need to be updated after every scaling operation. Any unused IDs are still valid for the next scaling event.

The Cluster Operator allows you to specify a range of IDs in either ascending or descending order, so you can define them in the order the nodes are scaled. For example, when scaling up, you can specify a range such as `[1000-1999]`, and the new nodes are assigned the next lowest IDs: `1000, 1001, 1002, 1003`, and so on. Conversely, when scaling down, you can specify a range like `[1999-1000]`, ensuring that nodes with the next highest IDs are removed: `1003, 1002, 1001, 1000`, and so on.

If you don't specify an ID range using the annotations, the Cluster Operator follows its default behavior for handling IDs during scaling operations. Node IDs start at 0 (zero) and run sequentially across the Kafka cluster. The next lowest ID is assigned to a new node. Gaps to node IDs are filled across the cluster. This means that they might not run sequentially within a node pool. The default behavior for scaling up is to add the next lowest available node ID across the cluster; and for scaling down, it is to remove the node in the node pool with the highest available node ID. The default approach is also applied if the assigned range of IDs is misformatted, the scaling up range runs out of IDs, or the scaling down range does not apply to any in-use nodes.

#### Prerequisites

- [The Cluster Operator must be deployed.](#)
- (Optional) Use the `reserved.broker-max.id` configuration property to extend the allowable range for node IDs within your node pools.

By default, Apache Kafka restricts node IDs to numbers ranging from 0 to 999. To use node ID values greater than 999, add the `reserved.broker-max.id` configuration property to the [Kafka](#) custom resource and specify the required maximum node ID value.

In this example, the maximum node ID is set at 10000. Node IDs can then be assigned up to that value.

#### *Example configuration for the maximum node ID number*

```
apiVersion: kafka.strimzi.io/v1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    config:
      reserved.broker.max.id: 10000
    # ...
```

#### Procedure

1. Annotate the node pool with the IDs to use when scaling up or scaling down, as shown in the following examples.

IDs for scaling up are assigned to node pool `pool-a`:

#### *Assigning IDs for scaling up*

```
kubectl annotate kafkanodepool pool-a strimzi.io/next-node-ids="[0,1,2,10-20,30]"
```

The lowest available ID from this range is used when adding a node to `pool-a`.

IDs for scaling down are assigned to node pool `pool-b`:

## Assigning IDs for scaling down

```
kubectl annotate kafkanodepool pool-b strimzi.io/remove-node-ids="[60-50,9,8,7]"
```

The highest available ID from this range is removed when scaling down **pool-b**.

**NOTE** If you want to remove a specific node, you can assign a single node ID to the scaling down annotation: `kubectl annotate kafkanodepool pool-b strimzi.io/remove-node-ids="[3]"`.

2. You can now scale the node pool.

For more information, see the following:

- [Adding nodes to a node pool](#)
- [Removing nodes from a node pool](#)
- [Moving nodes between node pools](#)

On reconciliation, a warning is given if the annotations are misformatted.

3. After you have performed the scaling operation, you can remove the annotation if it's no longer needed.

## Removing the annotation for scaling up

```
kubectl annotate kafkanodepool pool-a strimzi.io/next-node-ids-
```

## Removing the annotation for scaling down

```
kubectl annotate kafkanodepool pool-b strimzi.io/remove-node-ids-
```

### 11.3.2. Impact on racks when moving nodes from node pools

If rack awareness is enabled on a Kafka cluster, replicas can be spread across different racks, data centers, or availability zones. When moving nodes from node pools, consider the implications on the cluster topology, particularly regarding rack awareness. Removing specific pods from node pools, especially out of order, may break the cluster topology or cause an imbalance in distribution across racks. An imbalance can impact both the distribution of nodes themselves and the partition replicas within the cluster. An uneven distribution of nodes and partitions across racks can affect the performance and resilience of the Kafka cluster.

Plan the removal of nodes strategically to maintain the required balance and resilience across racks. Use the `strimzi.io/remove-node-ids` annotation to move nodes with specific IDs with caution. Ensure that configuration to spread partition replicas across racks and for clients to consume from the closest replicas is not broken.

**TIP** Use Cruise Control and the `KafkaRebalance` resource with the `RackAwareGoal` to make

sure that replicas remain distributed across different racks.

### 11.3.3. Adding nodes to a node pool

This procedure describes how to scale up a node pool to add new nodes. Currently, scale up is only possible for broker-only node pools containing nodes that run as dedicated brokers.

In this procedure, we start with three nodes for node pool `pool-a`:

*Kafka nodes in the node pool*

NAME	READY	STATUS	RESTARTS
my-cluster-pool-a-0	1/1	Running	0
my-cluster-pool-a-1	1/1	Running	0
my-cluster-pool-a-2	1/1	Running	0

Node IDs are appended to the name of the node on creation. We add node `my-cluster-pool-a-3`, which has a node ID of `3`.

**NOTE**

During this process, the ID of the node that holds the partition replicas changes. Consider any dependencies that reference the node ID.

*Prerequisites*

- [The Cluster Operator must be deployed.](#)
- [Cruise Control is deployed with Kafka.](#)
- (Optional) [Auto-rebalancing is enabled.](#)

If auto-rebalancing is enabled, partition reassignment happens automatically during the node scaling process, so you don't need to manually initiate the reassignment through Cruise Control.

- (Optional) For scale up operations, [you can specify the node IDs to use in the operation.](#)

If you have assigned a range of node IDs for the operation, the ID of the node being added is determined by the sequence of nodes given. If you have assigned a single node ID, a node is added with the specified ID. Otherwise, the lowest available node ID across the cluster is used.

*Procedure*

1. Create a new node in the node pool.

For example, node pool `pool-a` has three replicas. We add a node by increasing the number of replicas:

```
kubectl scale kafkanodepool pool-a --replicas=4
```

2. Check the status of the deployment and wait for the pods in the node pool to be created and ready (`1/1`).

```
kubectl get pods -n <my_cluster_operator_namespace>
```

*Output shows four Kafka nodes in the node pool*

NAME	READY	STATUS	RESTARTS
my-cluster-pool-a-0	1/1	Running	0
my-cluster-pool-a-1	1/1	Running	0
my-cluster-pool-a-2	1/1	Running	0
my-cluster-pool-a-3	1/1	Running	0

### 3. Reassign the partitions after increasing the number of nodes in the node pool.

- If auto-rebalancing is enabled, partitions are reassigned to new nodes automatically, so you can skip this step.
- If auto-rebalancing is not enabled, use the Cruise Control `add-brokers` mode to move partition replicas from existing brokers to the newly added brokers.

*Using Cruise Control to reassign partition replicas*

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaRebalance
metadata:
  # ...
spec:
  mode: add-brokers
  brokers: [3]
```

We are reassigning partitions to node `my-cluster-pool-a-3`. The reassignment can take some time depending on the number of topics and partitions in the cluster.

#### 11.3.4. Removing nodes from a node pool

This procedure describes how to scale down a node pool to remove nodes. Currently, scale down is only possible for broker-only node pools containing nodes that run as dedicated brokers.

In this procedure, we start with four nodes for node pool `pool-a`:

*Kafka nodes in the node pool*

NAME	READY	STATUS	RESTARTS
my-cluster-pool-a-0	1/1	Running	0
my-cluster-pool-a-1	1/1	Running	0
my-cluster-pool-a-2	1/1	Running	0
my-cluster-pool-a-3	1/1	Running	0

Node IDs are appended to the name of the node on creation. We remove node `my-cluster-pool-a-3`, which has a node ID of `3`.

**NOTE**

During this process, the ID of the node that holds the partition replicas changes. Consider any dependencies that reference the node ID.

## Prerequisites

- The Cluster Operator must be deployed.
- Cruise Control is deployed with Kafka.
- (Optional) Auto-rebalancing is enabled.

If auto-rebalancing is enabled, partition reassignment happens automatically during the node scaling process, so you don't need to manually initiate the reassignment through Cruise Control.

- (Optional) For scale down operations, [you can specify the node IDs to use in the operation](#).

If you have assigned a range of node IDs for the operation, the ID of the node being removed is determined by the sequence of nodes given. If you have assigned a single node ID, the node with the specified ID is removed. Otherwise, the node with the highest available ID in the node pool is removed.

## Procedure

1. Reassign the partitions before decreasing the number of nodes in the node pool.
  - If auto-rebalancing is enabled, partitions are moved off brokers that are going to be removed automatically, so you can skip this step.
  - If auto-rebalancing is not enabled, use the Cruise Control `remove-brokers` mode to move partition replicas off the brokers that are going to be removed.

*Using Cruise Control to reassign partition replicas*

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaRebalance
metadata:
  # ...
spec:
  mode: remove-brokers
  brokers: [3]
```

We are reassigning partitions from node `my-cluster-pool-a-3`. The reassignment can take some time depending on the number of topics and partitions in the cluster.

2. After the reassignment process is complete, and the node being removed has no live partitions, reduce the number of Kafka nodes in the node pool.

For example, node pool `pool-a` has four replicas. We remove a node by decreasing the number of replicas:

```
kubectl scale kafkanodepool pool-a --replicas=3
```

*Output shows three Kafka nodes in the node pool*

NAME	READY	STATUS	RESTARTS
my-cluster-pool-b-kafka-0	1/1	Running	0
my-cluster-pool-b-kafka-1	1/1	Running	0

### 11.3.5. Moving nodes between node pools

This procedure describes how to move nodes between source and target Kafka node pools without downtime. You create a new node on the target node pool and reassign partitions to move data from the old node on the source node pool. When the replicas on the new node are in-sync, you can delete the old node.

In this procedure, we start with two node pools:

- **pool-a** with three replicas is the target node pool
- **pool-b** with four replicas is the source node pool

We scale up **pool-a**, and reassign partitions and scale down **pool-b**, which results in the following:

- **pool-a** with four replicas
- **pool-b** with three replicas

Currently, scaling is only possible for broker-only node pools containing nodes that run as dedicated brokers.

**NOTE**

During this process, the ID of the node that holds the partition replicas changes. Consider any dependencies that reference the node ID.

*Prerequisites*

- [The Cluster Operator must be deployed.](#)
- [Cruise Control is deployed with Kafka.](#)
- (Optional) [Auto-rebalancing is enabled.](#)

If auto-rebalancing is enabled, partition reassignment happens automatically during the node scaling process, so you don't need to manually initiate the reassignment through Cruise Control.

- (Optional) For scale up and scale down operations, [you can specify the range of node IDs to use](#). If you have assigned node IDs for the operation, the ID of the node being added or removed is determined by the sequence of nodes given. Otherwise, the lowest available node ID across the cluster is used when adding nodes; and the node with the highest available ID in the node pool is removed.

*Procedure*

1. Create a new node in the target node pool.

For example, node pool **pool-a** has three replicas. We add a node by increasing the number of replicas:

```
kubectl scale kafkanodepool pool-a --replicas=4
```

2. Check the status of the deployment and wait for the pods in the node pool to be created and

ready (1/1).

```
kubectl get pods -n <my_cluster_operator_namespace>
```

*Output shows four Kafka nodes in the source and target node pools*

NAME	READY	STATUS	RESTARTS
my-cluster-pool-a-0	1/1	Running	0
my-cluster-pool-a-1	1/1	Running	0
my-cluster-pool-a-4	1/1	Running	0
my-cluster-pool-a-7	1/1	Running	0
my-cluster-pool-b-2	1/1	Running	0
my-cluster-pool-b-3	1/1	Running	0
my-cluster-pool-b-5	1/1	Running	0
my-cluster-pool-b-6	1/1	Running	0

Node IDs are appended to the name of the node on creation. We add node **my-cluster-pool-a-7**, which has a node ID of **7**.

If auto-rebalancing is enabled, partitions are reassigned to new nodes and moved off brokers that are going to be removed automatically, so you can skip the next step.

3. If auto-rebalancing is not enabled, reassign partitions before decreasing the number of nodes in the source node pool.

Use the Cruise Control **remove-brokers** mode to move partition replicas off the brokers that are going to be removed.

*Using Cruise Control to reassign partition replicas*

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaRebalance
metadata:
  # ...
spec:
  mode: remove-brokers
  brokers: [6]
```

We are reassigning partitions from node **my-cluster-pool-b-6**. The reassignment can take some time depending on the number of topics and partitions in the cluster.

4. After the reassignment process is complete, reduce the number of Kafka nodes in the source node pool.

For example, node pool **pool-b** has four replicas. We remove a node by decreasing the number of replicas:

```
kubectl scale kafkanodepool pool-b --replicas=3
```

The node with the highest ID (6) within the pool is removed.

*Output shows three Kafka nodes in the source node pool*

NAME	READY	STATUS	RESTARTS
my-cluster-pool-b-kafka-2	1/1	Running	0
my-cluster-pool-b-kafka-3	1/1	Running	0
my-cluster-pool-b-kafka-5	1/1	Running	0

### 11.3.6. Changing node pool roles

Node pools are used with Kafka clusters that operate in KRaft mode (using Kafka Raft metadata). If you are using KRaft mode, you can specify roles for all nodes in the node pool to operate as brokers, controllers, or both.

In certain circumstances you might want to change the roles assigned to a node pool. For example, you may have a node pool that contains nodes that perform dual broker and controller roles, and then decide to split the roles between two node pools. In this case, you create a new node pool with nodes that act only as brokers, and then reassign partitions from the dual-role nodes to the new brokers. You can then switch the old node pool to a controller-only role.

You can also perform the reverse operation by moving from node pools with controller-only and broker-only roles to a node pool that contains nodes that perform dual broker and controller roles. In this case, you add the [broker](#) role to the existing controller-only node pool, reassign partitions from the broker-only nodes to the dual-role nodes, and then delete the broker-only node pool.

When removing [broker](#) roles in the node pool configuration, keep in mind that Kafka does not automatically reassign partitions. Before removing the broker role, ensure that nodes changing to controller-only roles do not have any assigned partitions. If partitions are assigned, the change is prevented. No replicas must be left on the node before removing the broker role. The best way to reassign partitions before changing roles is to apply a Cruise Control optimization proposal in [remove-brokers](#) mode. For more information, see [Generating optimization proposals](#).

**NOTE** Scaling controller nodes in node pools is currently not supported because the related Kafka feature is still under development. For more information, see [KAFKA-16538](#).

### 11.3.7. Transitioning to separate broker and controller roles

This procedure describes how to transition to using node pools with separate roles. If your Kafka cluster is using a node pool with combined controller and broker roles, you can transition to using two node pools with separate roles. To do this, rebalance the cluster to move partition replicas to a node pool with a broker-only role, and then switch the old node pool to a controller-only role.

In this procedure, we start with node pool [pool-a](#), which has [controller](#) and [broker](#) roles:

## *Dual-role node pool*

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaNodePool
metadata:
  name: pool-a
  labels:
    strimzi.io/cluster: my-cluster
spec:
  replicas: 3
  roles:
    - controller
    - broker
  storage:
    type: jbod
    volumes:
      - id: 0
        type: persistent-claim
        size: 20Gi
        deleteClaim: false
# ...
```

The node pool has three nodes:

### *Kafka nodes in the node pool*

NAME	READY	STATUS	RESTARTS
my-cluster-pool-a-0	1/1	Running	0
my-cluster-pool-a-1	1/1	Running	0
my-cluster-pool-a-2	1/1	Running	0

Each node performs a combined role of broker and controller. We create a second node pool called **pool-b**, with three nodes that act as brokers only.

**NOTE**

During this process, the ID of the node that holds the partition replicas changes.  
Consider any dependencies that reference the node ID.

### *Prerequisites*

- [The Cluster Operator must be deployed.](#)
- [Cruise Control is deployed with Kafka.](#)

### *Procedure*

1. Create a node pool with a **broker** role.

### *Example node pool configuration*

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaNodePool
metadata:
```

```

name: pool-b
labels:
  strimzi.io/cluster: my-cluster
spec:
  replicas: 3
  roles:
    - broker
  storage:
    type: jbod
  volumes:
    - id: 0
      type: persistent-claim
      size: 100Gi
      deleteClaim: false
# ...

```

The new node pool also has three nodes. If you already have a broker-only node pool, you can skip this step.

2. Apply the new **KafkaNodePool** resource to create the brokers.
3. Check the status of the deployment and wait for the pods in the node pool to be created and ready (1/1).

```
kubectl get pods -n <my_cluster_operator_namespace>
```

*Output shows pods running in two node pools*

NAME	READY	STATUS	RESTARTS
my-cluster-pool-a-0	1/1	Running	0
my-cluster-pool-a-1	1/1	Running	0
my-cluster-pool-a-2	1/1	Running	0
my-cluster-pool-b-3	1/1	Running	0
my-cluster-pool-b-4	1/1	Running	0
my-cluster-pool-b-5	1/1	Running	0

Node IDs are appended to the name of the node on creation.

4. Use the Cruise Control **remove-brokers** mode to reassign partition replicas from the dual-role nodes to the newly added brokers.

*Using Cruise Control to reassign partition replicas*

```

apiVersion: kafka.strimzi.io/v1
kind: KafkaRebalance
metadata:
  # ...
spec:
  mode: remove-brokers

```

```
brokers: [0, 1, 2]
```

The reassignment can take some time depending on the number of topics and partitions in the cluster.

**NOTE** If nodes changing to controller-only roles have any assigned partitions, the change is prevented. The `status.conditions` of the `Kafka` resource provide details of events preventing the change.

5. Remove the `broker` role from the node pool that originally had a combined role.

*Dual-role nodes switched to controllers*

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaNodePool
metadata:
  name: pool-a
  labels:
    strimzi.io/cluster: my-cluster
spec:
  replicas: 3
  roles:
    - controller
  storage:
    type: jbod
    volumes:
      - id: 0
        type: persistent-claim
        size: 20Gi
        deleteClaim: false
    # ...
```

6. Apply the configuration change so that the node pool switches to a controller-only role.

### 11.3.8. Transitioning to dual-role nodes

This procedure describes how to transition from separate node pools with broker-only and controller-only roles to using a dual-role node pool. If your Kafka cluster is using node pools with dedicated controller and broker nodes, you can transition to using a single node pool with both roles. To do this, add the `broker` role to the controller-only node pool, rebalance the cluster to move partition replicas to the dual-role node pool, and then delete the old broker-only node pool.

In this procedure, we start with two node pools `pool-a`, which has only the `controller` role and `pool-b` which has only the `broker` role:

*Single role node pools*

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaNodePool
```

```

metadata:
  name: pool-a
  labels:
    strimzi.io/cluster: my-cluster
spec:
  replicas: 3
  roles:
    - controller
  storage:
    type: jbod
    volumes:
      - id: 0
        type: persistent-claim
        size: 100Gi
        deleteClaim: false
# ...
---
apiVersion: kafka.strimzi.io/v1
kind: KafkaNodePool
metadata:
  name: pool-b
  labels:
    strimzi.io/cluster: my-cluster
spec:
  replicas: 3
  roles:
    - broker
  storage:
    type: jbod
    volumes:
      - id: 0
        type: persistent-claim
        size: 100Gi
        deleteClaim: false
# ...

```

The Kafka cluster has six nodes:

*Kafka nodes in the node pools*

NAME	READY	STATUS	RESTARTS
my-cluster-pool-a-0	1/1	Running	0
my-cluster-pool-a-1	1/1	Running	0
my-cluster-pool-a-2	1/1	Running	0
my-cluster-pool-b-3	1/1	Running	0
my-cluster-pool-b-4	1/1	Running	0
my-cluster-pool-b-5	1/1	Running	0

The **pool-a** nodes perform the role of controller. The **pool-b** nodes perform the role of broker.

**NOTE**

During this process, the ID of the node that holds the partition replicas changes.  
Consider any dependencies that reference the node ID.

*Prerequisites*

- [The Cluster Operator must be deployed.](#)
- [Cruise Control is deployed with Kafka.](#)

*Procedure*

1. Edit the node pool `pool-a` and add the `broker` role to it.

*Example node pool configuration*

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaNodePool
metadata:
  name: pool-a
  labels:
    strimzi.io/cluster: my-cluster
spec:
  replicas: 3
  roles:
    - controller
    - broker
  storage:
    type: jbod
    volumes:
      - id: 0
        type: persistent-claim
        size: 100Gi
        deleteClaim: false
    # ...
```

2. Check the status and wait for the pods in the node pool to be restarted and ready (1/1).

```
kubectl get pods -n <my_cluster_operator_namespace>
```

*Output shows pods running in two node pools*

NAME	READY	STATUS	RESTARTS
my-cluster-pool-a-0	1/1	Running	0
my-cluster-pool-a-1	1/1	Running	0
my-cluster-pool-a-2	1/1	Running	0
my-cluster-pool-b-3	1/1	Running	0
my-cluster-pool-b-4	1/1	Running	0
my-cluster-pool-b-5	1/1	Running	0

Node IDs are appended to the name of the node on creation.

3. Use the Cruise Control `remove-brokers` mode to reassign partition replicas from the broker-only nodes to the dual-role nodes.

*Using Cruise Control to reassign partition replicas*

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaRebalance
metadata:
  # ...
spec:
  mode: remove-brokers
  brokers: [3, 4, 5]
```

The reassignment can take some time depending on the number of topics and partitions in the cluster.

4. Remove the `pool-b` node pool that has the old broker-only nodes.

```
kubectl delete kafkanodepool pool-b -n <my_cluster_operator_namespace>
```

## 11.4. Configuring Kafka storage

Strimzi supports different Kafka storage options. You can choose between the following basic types:

### Ephemeral storage

Ephemeral storage is temporary and only persists while a pod is running. When a pod is deleted, the data is lost, though data can be recovered in a highly available environment. Due to its transient nature, ephemeral storage is only recommended for development and testing environments.

### Persistent storage

Persistent storage retains data across pod restarts and system disruptions, making it ideal for production environments.

JBOD (Just a Bunch of Disks) storage allows you to configure your Kafka cluster to use multiple disks or volumes as ephemeral or persistent storage.

#### *JBOD storage (multiple volumes)*

When specifying JBOD storage, you must still decide between using ephemeral or persistent volumes for each disk. Even if you start with only one volume, using JBOD allows for future scaling by adding more volumes as needed, and that is why it is always recommended.

#### NOTE

Persistent, ephemeral, and JBOD storage types cannot be changed after a Kafka cluster is deployed. However, you can add or remove volumes of different types from the JBOD storage. You can also create and migrate to node pools with new storage specifications.

### *Tiered storage (advanced)*

Tiered storage provides additional flexibility for managing Kafka data by combining different storage types with varying performance and cost characteristics. It allows Kafka to offload older data to cheaper, long-term storage (such as object storage) while keeping recent, frequently accessed data on faster, more expensive storage (such as block storage).

Tiered storage is an add-on capability. After configuring storage (ephemeral, persistent, or JBOD) for Kafka nodes, you can configure tiered storage at the cluster level and enable it for specific topics using the `remote.storage.enable` topic-level configuration.

#### **11.4.1. Storage considerations**

Efficient data storage is essential for Strimzi to operate effectively. Strimzi has been tested with block storage as the primary storage type for Kafka brokers, and block storage is strongly recommended. File system-based storage (such as NFS) is not guaranteed to work for primary broker storage and may cause stability or performance issues.

Common block storage types supported by Kubernetes include:

- Cloud-based block storage solutions:
  - Amazon EBS (for AWS)
  - Azure Disk Storage (for Microsoft Azure)
  - Persistent Disk (for Google Cloud)
- Persistent storage (for bare metal deployments) using [local persistent volumes](#)
- Storage Area Network (SAN) volumes accessed by protocols like Fibre Channel or iSCSI

**NOTE** Strimzi does not require Kubernetes raw block volumes.

### **File systems**

Kafka uses a file system for storing messages. Strimzi is compatible with the XFS and ext4 file systems, which are commonly used with Kafka. Consider the underlying architecture and requirements of your deployment when choosing and setting up your file system.

For more information, see the [Apache Kafka filesystem selection documentation](#).

### **Tiered storage**

Kafka's tiered storage feature is supported in Strimzi as an optional capability.

With tiered storage enabled:

- Primary broker storage, such as persistent volumes or JBOD, handles recent data
- Remote tiered storage, such as object storage, is used for historical data

Strimzi allows users to integrate their own tiered storage plugins to support specific remote storage backends. If using a custom plugin, ensure that it meets performance and compatibility

requirements before deploying to production.

## Disk usage

Solid-state drives (SSDs), though not essential, can improve the performance of Kafka in large clusters where data is sent to and received from multiple topics asynchronously.

**NOTE** Replicated storage is not required, as Kafka provides built-in data replication.

### 11.4.2. Configuring storage types

Use the `storage` properties of the `KafkaNodePool` custom resource to configure storage for a deployment of Kafka in KRaft mode.

#### Configuring ephemeral storage

To use ephemeral storage, specify `ephemeral` as the storage type.

*Example configuration for ephemeral storage*

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaNodePool
metadata:
  name: my-node-pool
  labels:
    strimzi.io/cluster: my-cluster
spec:
  replicas: 3
  roles:
    - broker
  storage:
    type: ephemeral
# ...
```

Ephemeral storage uses `emptyDir` volumes, which are created when a pod is assigned to a node. You can limit the size of the `emptyDir` volume with the `sizeLimit` property.

The ephemeral volume used by Kafka brokers for log directories is mounted at `/var/lib/kafka/data/kafka-log<pod_id>`.

**IMPORTANT**

Ephemeral storage is not suitable for Kafka topics with a replication factor of 1.

For more information on ephemeral storage configuration options, see the [EphemeralStorage schema reference](#).

#### Configuring persistent storage

To use persistent storage, specify one of the following as the storage type:

- **persistent-claim** for a single persistent volume
- **jbod** for multiple persistent volumes in a Kafka cluster (Recommended for Kafka in a production environment)

#### *Example configuration for persistent storage*

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaNodePool
metadata:
  name: my-node-pool
  labels:
    strimzi.io/cluster: my-cluster
spec:
  replicas: 3
  roles:
    - broker
  storage:
    type: persistent-claim
    size: 500Gi
    deleteClaim: true
# ...
```

Strimzi uses [Persistent Volume Claims](#) (PVCs) to request storage on persistent volumes (PVs). The PVC binds to a PV that meets the requested storage criteria, without needing to know the underlying storage infrastructure.

PVCs created for Kafka pods follow the naming convention `data-<kafka_cluster_name>-<pool_name>-<pod_id>`, and the persistent volumes for Kafka logs are mounted at `/var/lib/kafka/data/kafka-log<pod_id>`.

You can also specify custom storage classes ([StorageClass](#)) and volume selectors in the storage configuration.

#### *Example class and selector configuration*

```
# ...
storage:
  type: persistent-claim
  size: 500Gi
  class: my-storage-class
  selector:
    hdd-type: ssd
    deleteClaim: true
# ...
```

Storage classes define storage profiles and dynamically provision persistent volumes (PVs) based on those profiles. This is useful, for example, when storage classes are restricted to different availability zones or data centers. If a storage class is not specified, the default storage class in the Kubernetes cluster is used. Selectors specify persistent volumes that offer specific features, such as

solid-state drive (SSD) volumes.

For more information on persistent storage configuration options, see the [PersistentClaimStorage schema reference](#).

## Resizing persistent volumes

Persistent volumes can be resized by changing the `size` storage property without any risk of data loss, as long as the storage infrastructure supports it. Following a configuration update to change the size of the storage, Strimzi instructs the storage infrastructure to make the change.

Storage expansion is supported in Strimzi clusters that use persistent-claim volumes. Decreasing the size of persistent volumes is not supported in Kubernetes. For more information about resizing persistent volumes in Kubernetes, see [Resizing Persistent Volumes using Kubernetes](#).

After increasing the value of the `size` property, Kubernetes increases the capacity of the selected persistent volumes in response to a request from the Cluster Operator. When the resizing is complete, the Cluster Operator restarts all pods that use the resized persistent volumes. This happens automatically.

In this example, the volumes are increased to 2000Gi.

*Kafka configuration to increase volume size to 2000Gi*

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaNodePool
metadata:
  name: my-node-pool
  labels:
    strimzi.io/cluster: my-cluster
spec:
  replicas: 3
  roles:
    - broker
  storage:
    type: jbod
    volumes:
      - id: 0
        type: persistent-claim
        size: 2000Gi
        deleteClaim: false
      - id: 1
        type: persistent-claim
        size: 2000Gi
        deleteClaim: false
      - id: 2
        type: persistent-claim
        size: 2000Gi
        deleteClaim: false
  # ...
```

Returning information on the PVs verifies the changes:

```
kubectl get pv
```

*Storage capacity of PVs*

NAME	CAPACITY	CLAIM
pvc-0ca459ce-...	2000Gi	my-project/data-my-cluster-my-node-pool-2
pvc-6e1810be-...	2000Gi	my-project/data-my-cluster-my-node-pool-0
pvc-82dc78c9-...	2000Gi	my-project/data-my-cluster-my-node-pool-1

The output shows the names of each PVC associated with a broker pod.

**NOTE**

Storage *reduction* is only possible when using multiple disks per broker. You can remove a disk after moving all partitions on the disk to other volumes within the same broker (intra-broker) or to other brokers within the same cluster (intra-cluster).

## Configuring JBOD storage

To use JBOD storage, specify `jbod` as the storage type and add configuration for the JBOD volumes. JBOD volumes can be persistent or ephemeral, with the configuration options and constraints applicable to each type.

*Example configuration for JBOD storage*

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaNodePool
metadata:
  name: my-node-pool
  labels:
    strimzi.io/cluster: my-cluster
spec:
  replicas: 3
  roles:
    - broker
  storage:
    type: jbod
    volumes:
      - id: 0
        type: persistent-claim
        size: 100Gi
        deleteClaim: false
      - id: 1
        type: persistent-claim
        size: 100Gi
        deleteClaim: false
      - id: 2
        type: persistent-claim
```

```
    size: 100Gi
    deleteClaim: false
  # ...
```

PVCs are created for the JBOD volumes using the naming convention `data-<volume_id>-<kafka_cluster_name>-<pool_name>-<pod_id>`, and the JBOD volumes used for log directories are mounted at `/var/lib/kafka/data-<volume_id>/kafka-log<pod_id>`.

### Adding or removing volumes from JBOD storage

Volume IDs cannot be changed once JBOD volumes are created, though you can add or remove volumes. When adding a new volume to the `volumes` array under an `id` which was already used in the past and removed, make sure that the previously used `PersistentVolumeClaims` have been deleted.

Use Cruise Control to reassign partitions when adding or removing volumes. For information on intra-broker disk balancing, see [Tuning options for rebalances](#).

#### 11.4.3. Configuring KRaft metadata log storage

In KRaft mode, each node (including brokers and controllers) stores a copy of the Kafka cluster's metadata log on one of its data volumes. By default, the log is stored on the volume with the lowest ID, but you can specify a different volume using the `kraftMetadata` property.

For controller-only nodes, storage is exclusively for the metadata log. Since the log is always stored on a single volume, using JBOD storage with multiple volumes does not improve performance or increase available disk space.

In contrast, broker nodes or nodes that combine broker and controller roles can share the same volume for both the metadata log and partition replica data, optimizing disk utilization. They can also use JBOD storage, where one volume is shared for the metadata log and partition replica data, while additional volumes are used solely for partition replica data.

Changing the volume that stores the metadata log triggers a rolling update of the cluster nodes, involving the deletion of the old log and the creation of a new one in the specified location. If `kraftMetadata` isn't specified, adding a new volume with a lower ID also prompts an update and relocation of the metadata log.

*Example JBOD storage configuration using volume with ID 1 to store the KRaft metadata*

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaNodePool
metadata:
  name: pool-a
  # ...
spec:
  storage:
    type: jbod
    volumes:
      - id: 0
```

```
type: persistent-claim
size: 100Gi
deleteClaim: false
- id: 1
  type: persistent-claim
  size: 100Gi
  kraftMetadata: shared
  deleteClaim: false
# ...
```

#### 11.4.4. Managing storage using node pools

Storage management in Strimzi is usually straightforward, and requires little change when set up, but there might be situations where you need to modify your storage configurations. Node pools simplify this process, because you can set up separate node pools that specify your new storage requirements.

In this procedure we create and manage storage for a node pool called `pool-a` containing three nodes. The steps require a scaling operation to add a new node pool. Currently, scaling is only possible for broker-only node pools containing nodes that run as dedicated brokers.

We show how to change the storage class (`volumes.class`) that defines the type of persistent storage it uses. You can use the same steps to change the storage size (`volumes.size`). This approach is particularly useful if you want to reduce disk sizes. When increasing disk sizes, you have the option to [dynamically resize persistent volumes](#).

**NOTE**

We strongly recommend using block storage. Strimzi is only tested for use with block storage.

##### Prerequisites

- [The Cluster Operator must be deployed](#).
- [Cruise Control is deployed with Kafka](#).
- For storage that uses persistent volume claims for dynamic volume allocation, storage classes are defined and available in the Kubernetes cluster that correspond to the storage solutions you need.

##### Procedure

1. Create the node pool with its own storage settings.

For example, node pool `pool-a` uses JBOD storage with persistent volumes:

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaNodePool
metadata:
  name: pool-a
  labels:
    strimzi.io/cluster: my-cluster
```

```

spec:
  roles:
    - broker
  replicas: 3
  storage:
    type: jbod
    volumes:
      - id: 0
        type: persistent-claim
        size: 500Gi
        class: gp2-ebs
  # ...

```

Nodes in `pool-a` are configured to use Amazon EBS (Elastic Block Store) GP2 volumes.

2. Apply the node pool configuration for `pool-a`.
3. Check the status of the deployment and wait for the pods in `pool-a` to be created and ready (1/1).

```
kubectl get pods -n <my_cluster_operator_namespace>
```

*Output shows three Kafka nodes in the node pool*

NAME	READY	STATUS	RESTARTS
my-cluster-pool-a-0	1/1	Running	0
my-cluster-pool-a-1	1/1	Running	0
my-cluster-pool-a-2	1/1	Running	0

4. To migrate to a new storage class, create a new node pool with the required storage configuration:

```

apiVersion: kafka.strimzi.io/v1
kind: KafkaNodePool
metadata:
  name: pool-b
  labels:
    strimzi.io/cluster: my-cluster
spec:
  roles:
    - broker
  replicas: 3
  storage:
    type: jbod
    volumes:
      - id: 0
        type: persistent-claim
        size: 1Ti
        class: gp3-ebs

```

```
# ...
```

Nodes in `pool-b` are configured to use Amazon EBS (Elastic Block Store) GP3 volumes.

5. Apply the node pool configuration for `pool-b`.
6. Check the status of the deployment and wait for the pods in `pool-b` to be created and ready.
7. Reassign the partitions from `pool-a` to `pool-b`.

When migrating to a new storage configuration, use the Cruise Control `remove-brokers` mode to move partition replicas off the brokers that are going to be removed.

*Using Cruise Control to reassign partition replicas*

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaRebalance
metadata:
  # ...
spec:
  mode: remove-brokers
  brokers: [0, 1, 2]
```

We are reassigning partitions from `pool-a`. The reassignment can take some time depending on the number of topics and partitions in the cluster.

8. After the reassignment process is complete, delete the old node pool:

```
kubectl delete kafkanodepool pool-a
```

#### 11.4.5. Managing storage affinity using node pools

In situations where storage resources, such as local persistent volumes, are constrained to specific worker nodes, or availability zones, configuring storage affinity helps to schedule pods to use the right nodes.

Node pools allow you to configure affinity independently. In this procedure, we create and manage storage affinity for two availability zones: `zone-1` and `zone-2`.

You can configure node pools for separate availability zones, but use the same storage class. We define an `all-zones` persistent storage class representing the storage resources available in each zone.

We also use the `.spec.template.pod` properties to configure the node affinity and schedule Kafka pods on `zone-1` and `zone-2` worker nodes.

The storage class and affinity is specified in node pools representing the nodes in each availability zone:

- `pool-zone-1`

- pool-zone-2.

#### *Prerequisites*

- The Cluster Operator must be deployed.
- If you are not familiar with the concepts of affinity, see the [Kubernetes node and pod affinity documentation](#).

#### *Procedure*

1. Define the storage class for use with each availability zone:

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: all-zones
provisioner: kubernetes.io/my-storage
parameters:
  type: ssd
volumeBindingMode: WaitForFirstConsumer
```

2. Create node pools representing the two availability zones, specifying the **all-zones** storage class and the affinity for each zone:

#### *Node pool configuration for zone-1*

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaNodePool
metadata:
  name: pool-zone-1
  labels:
    strimzi.io/cluster: my-cluster
spec:
  replicas: 3
  storage:
    type: jbod
    volumes:
      - id: 0
        type: persistent-claim
        size: 500Gi
        class: all-zones
  template:
    pod:
      affinity:
        nodeAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
              - matchExpressions:
                  - key: topology.kubernetes.io/zone
                    operator: In
                    values:
```

```
- zone-1  
# ...
```

#### *Node pool configuration for zone-2*

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaNodePool
metadata:
  name: pool-zone-2
  labels:
    strimzi.io/cluster: my-cluster
spec:
  replicas: 4
  storage:
    type: jbod
    volumes:
      - id: 0
        type: persistent-claim
        size: 500Gi
        class: all-zones
  template:
    pod:
      affinity:
        nodeAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
              - matchExpressions:
                  - key: topology.kubernetes.io/zone
                    operator: In
                    values:
                      - zone-2
# ...
```

3. Apply the node pool configuration.
4. Check the status of the deployment and wait for the pods in the node pools to be created and ready (1/1).

```
kubectl get pods -n <my_cluster_operator_namespace>
```

*Output shows 3 Kafka nodes in pool-zone-1 and 4 Kafka nodes in pool-zone-2*

NAME	READY	STATUS	RESTARTS
my-cluster-pool-zone-1-kafka-0	1/1	Running	0
my-cluster-pool-zone-1-kafka-1	1/1	Running	0
my-cluster-pool-zone-1-kafka-2	1/1	Running	0
my-cluster-pool-zone-2-kafka-3	1/1	Running	0
my-cluster-pool-zone-2-kafka-4	1/1	Running	0
my-cluster-pool-zone-2-kafka-5	1/1	Running	0

### 11.4.6. Tiered storage

Tiered storage introduces a flexible approach to managing Kafka data whereby log segments are moved to a separate storage system. For example, you can combine the use of block storage on brokers for frequently accessed data and offload older or less frequently accessed data from the block storage to more cost-effective, scalable remote storage solutions, such as Amazon S3, without compromising data accessibility and durability.

**NOTE** Tiered storage is a production-ready feature in Kafka since version 3.9.0, and it is also supported in Strimzi. Before introducing tiered storage to your environment, review the [known limitations](#) of this feature.

Tiered storage requires an implementation of Kafka's `RemoteStorageManager` interface to handle communication between Kafka and the remote storage system, which is enabled through configuration of the `Kafka` resource. Strimzi uses Kafka's `TopicBasedRemoteLogMetadataManager` for Remote Log Metadata Management (RLMM) when custom tiered storage is enabled. The RLMM manages the metadata related to remote storage.

To use custom tiered storage, do the following:

- Include a tiered storage plugin for Kafka in the Strimzi image by building a custom container image. The plugin must provide the necessary functionality for a Kafka cluster managed by Strimzi to interact with the tiered storage solution.
- Configure Kafka for tiered storage using `tieredStorage` properties in the `Kafka` resource. Specify the class name and path for the custom `RemoteStorageManager` implementation, as well as any additional configuration.
- If required, specify RLMM-specific tiered storage configuration.

*Example custom tiered storage configuration for Kafka*

```
apiVersion: kafka.strimzi.io/v1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    tieredStorage:
      type: custom ①
      remoteStorageManager: ②
        className: com.example.kafka.tiered.storage.s3.S3RemoteStorageManager
        classPath: /opt/kafka/plugins/tiered-storage-s3/*
        config:
          storage.bucket.name: my-bucket ③
          # ...
      config:
        rlmm.config.remote.log.metadata.topic.replication.factor: 1 ④
```

```
# ...
```

- ① The `type` must be set to `custom`.
- ② The configuration for the custom `RemoteStorageManager` implementation, including class name and path.
- ③ Configuration to pass to the custom `RemoteStorageManager` implementation, which Strimzi automatically prefixes with `rsm.config..`
- ④ Tiered storage configuration to pass to the RLMM, which requires an `rlmm.config.` prefix. For more information on tiered storage configuration, see the [Apache Kafka documentation](#).

## 11.5. Configuring the Entity Operator

Use the `entityOperator` property in `Kafka.spec` to configure the Entity Operator. The Entity Operator is responsible for managing Kafka-related entities in a running Kafka cluster. It comprises the following operators:

- Topic Operator to manage Kafka topics
- User Operator to manage Kafka users

By configuring the `Kafka` resource, the Cluster Operator can deploy the Entity Operator, including one or both operators. Once deployed, the operators are automatically configured to handle the topics and users of the Kafka cluster.

Each operator can only monitor a single namespace. For more information, see [Operator-watched Kafka resources](#).

The `entityOperator` property supports several sub-properties:

- `topicOperator`
- `userOperator`
- `template`

The `template` property contains the configuration of the Entity Operator pod, such as labels, annotations, affinity, and tolerations. For more information on configuring templates, see [Customizing Kubernetes resources](#).

The `topicOperator` property contains the configuration of the Topic Operator. When this option is missing, the Entity Operator is deployed without the Topic Operator.

The `userOperator` property contains the configuration of the User Operator. When this option is missing, the Entity Operator is deployed without the User Operator.

For more information on the properties used to configure the Entity Operator, see the [EntityOperatorSpec schema reference](#).

*Example of basic configuration enabling both operators*

```
apiVersion: kafka.strimzi.io/v1
```

```
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
  entityOperator:
    topicOperator: {}
    userOperator: {}
```

If an empty object ({} ) is used for the `topicOperator` and `userOperator`, all properties use their default values.

When both `topicOperator` and `userOperator` properties are missing, the Entity Operator is not deployed.

### 11.5.1. Configuring the Topic Operator

Use `topicOperator` properties in `Kafka.spec.entityOperator` to configure the Topic Operator.

The following properties are supported:

#### `watchedNamespace`

The Kubernetes namespace in which the Topic Operator watches for `KafkaTopic` resources. Default is the namespace where the Kafka cluster is deployed.

#### `reconciliationIntervalMs`

The interval between periodic reconciliations in milliseconds. Default `120000`.

#### `image`

The `image` property can be used to configure the container image which is used. To learn more, refer to the information provided on [configuring the `image` property](#).

#### `resources`

The `resources` property configures the amount of resources allocated to the Topic Operator. You can specify requests and limits for `memory` and `cpu` resources. The requests should be enough to ensure a stable performance of the operator.

#### `logging`

The `logging` property configures the logging of the Topic Operator. To learn more, refer to the information provided on [Topic Operator logging](#).

*Example Topic Operator configuration*

```
apiVersion: kafka.strimzi.io/v1
kind: Kafka
metadata:
  name: my-cluster
spec:
```

```

kafka:
  # ...
entityOperator:
  # ...
topicOperator:
  watchedNamespace: my-topic-namespace
  reconciliationIntervalMs: 60000
  resources:
    requests:
      cpu: "1"
      memory: 500Mi
    limits:
      cpu: "1"
      memory: 500Mi
  # ...

```

## 11.5.2. Configuring the User Operator

Use `userOperator` properties in `Kafka.spec.entityOperator` to configure the User Operator. The following properties are supported:

### `watchedNamespace`

The Kubernetes namespace in which the User Operator watches for `KafkaUser` resources. Default is the namespace where the Kafka cluster is deployed.

### `reconciliationIntervalMs`

The interval between periodic reconciliations in milliseconds. Default `120000`.

### `image`

The `image` property can be used to configure the container image which will be used. To learn more, refer to the information provided on [configuring the `image` property](#).

### `resources`

The `resources` property configures the amount of resources allocated to the User Operator. You can specify requests and limits for `memory` and `cpu` resources. The requests should be enough to ensure a stable performance of the operator.

### `logging`

The `logging` property configures the logging of the User Operator. To learn more, refer to the information provided on [User Operator logging](#).

### `secretPrefix`

The `secretPrefix` property adds a prefix to the name of all Secrets created from the `KafkaUser` resource. For example, `secretPrefix: kafka-` would prefix all Secret names with `kafka-`. So a `KafkaUser` named `my-user` would create a Secret named `kafka-my-user`.

*Example User Operator configuration*

```
apiVersion: kafka.strimzi.io/v1
```

```

kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
  entityOperator:
    # ...
  userOperator:
    watchedNamespace: my-user-namespace
    reconciliationIntervalMs: 60000
    resources:
      requests:
        cpu: "1"
        memory: 500Mi
      limits:
        cpu: "1"
        memory: 500Mi
    # ...

```

## 11.6. Configuring the Cluster Operator

Use environment variables to configure the Cluster Operator. Specify the environment variables for the container image of the Cluster Operator in its [Deployment](#) configuration file. You can use the following environment variables to configure the Cluster Operator. If you are running Cluster Operator replicas in standby mode, there are additional [environment variables for enabling leader election](#).

Kafka, Kafka Connect, and Kafka MirrorMaker support multiple versions. Use their [STRIMZI\\_<COMPONENT\\_NAME>\\_IMAGES](#) environment variables to configure the default container images used for each version. The configuration provides a mapping between a version and an image. The required syntax is whitespace or comma-separated `<version> = <image>` pairs, which determine the image to use for a given version. For example, `4.1.1=quay.io/strimzi/kafka:0.50.0-kafka-4.1.1`. These default images are overridden if `image` property values are specified in the configuration of a component. For more information on `image` configuration of components, see the [Strimzi Custom Resource API Reference](#).

**NOTE**

The [Deployment](#) configuration file provided with the Strimzi release artifacts is `install/cluster-operator/060-Deployment-strimzi-cluster-operator.yaml`.

### STRIMZI\_NAMESPACE

A comma-separated list of namespaces that the operator operates in. When not set, set to empty string, or set to `*`, the Cluster Operator operates in all namespaces.

The Cluster Operator deployment might use the downward API to set this automatically to the namespace the Cluster Operator is deployed in.

## *Example configuration for Cluster Operator namespaces*

```
env:  
- name: STRIMZI_NAMESPACE  
  valueFrom:  
    fieldRef:  
      fieldPath: metadata.namespace
```

### **STRIMZI\_FULL\_RECONCILIATION\_INTERVAL\_MS**

Optional, default is 120000 ms. The interval between [periodic reconciliations](#), in milliseconds.

### **STRIMZI\_OPERATION\_TIMEOUT\_MS**

Optional, default 300000 ms. The timeout for internal operations, in milliseconds. Increase this value when using Strimzi on clusters where regular Kubernetes operations take longer than usual (due to factors such as prolonged download times for container images, for example).

### **STRIMZI\_OPERATIONS\_THREAD\_POOL\_SIZE**

Optional, default 10. The worker thread pool size, which is used for various asynchronous and blocking operations that are run by the Cluster Operator.

### **STRIMZI\_OPERATOR\_NAME**

Optional, defaults to the pod's hostname. The operator name identifies the Strimzi instance when [emitting Kubernetes events](#).

### **STRIMZI\_OPERATOR\_NAMESPACE**

The name of the namespace where the Cluster Operator is running. Do not configure this variable manually. Use the downward API.

```
env:  
- name: STRIMZI_OPERATOR_NAMESPACE  
  valueFrom:  
    fieldRef:  
      fieldPath: metadata.namespace
```

### **STRIMZI\_OPERATOR\_NAMESPACE\_LABELS**

Optional. The labels of the namespace where the Strimzi Cluster Operator is running. Use namespace labels to configure the namespace selector in [network policies](#). Network policies allow the Strimzi Cluster Operator access only to the operands from the namespace with these labels. When not set, the namespace selector in network policies is configured to allow access to the Cluster Operator from any namespace in the Kubernetes cluster.

```
env:  
- name: STRIMZI_OPERATOR_NAMESPACE_LABELS  
  value: label1=value1,label2=value2
```

## **STRIMZI\_POD\_DISRUPTION\_BUDGET\_GENERATION**

Optional. Default is `true`. Controls automatic creation of `PodDisruptionBudget` resources for Kafka, Kafka Connect, MirrorMaker2, and HTTP Bridge. Each budget applies across all pods deployed for the associated component. For Kafka clusters, this includes all node pool pods.

A pod disruption budget with the `maxUnavailable` value set to zero prevents Kubernetes from evicting pods automatically.

Set this environment variable to `false` to disable pod disruption budget generation. You might do this, for example, if you want to manage the pod disruption budgets yourself, or if you have a development environment where availability is not important.

## **STRIMZI\_LABELS\_EXCLUSION\_PATTERN**

Optional, default regex pattern is `(^app.kubernetes.io/(?!part-of).*)|^kustomize.toolkit.fluxcd.io.*`. The regex exclusion pattern used to filter labels propagation from the main custom resource to its subresources. The labels exclusion filter is not applied to labels in template sections such as `spec.kafka.template.pod.metadata.labels`.

```
env:  
  - name: STRIMZI_LABELS_EXCLUSION_PATTERN  
    value: "^key1.*"
```

## **STRIMZI\_CUSTOM\_<COMPONENT\_NAME>\_LABELS**

Optional. One or more custom labels to apply to all the pods created by the custom resource of the component. The Cluster Operator labels the pods when the custom resource is created or is next reconciled.

Labels can be applied to the following components:

- `KAFKA`
- `KAFKA_CONNECT`
- `KAFKA_CONNECT_BUILD`
- `ENTITY_OPERATOR`
- `KAFKA_MIRROR MAKER2`
- `KAFKA_MIRROR MAKER`
- `CRUISE_CONTROL`
- `KAFKA_BRIDGE`
- `KAFKA_EXPORTER`

## **STRIMZI\_CUSTOM\_RESOURCE\_SELECTOR**

Optional. The label selector to filter the custom resources handled by the Cluster Operator. The operator will operate only on those custom resources that have the specified labels set. Resources without these labels will not be seen by the operator. The label selector applies to `Kafka`, `KafkaConnect`, `KafkaBridge`, and `KafkaMirrorMaker2` resources. `KafkaRebalance` and `KafkaConnector` resources are operated only when their corresponding Kafka and Kafka Connect

clusters have the matching labels.

```
env:  
  - name: STRIMZI_CUSTOM_RESOURCE_SELECTOR  
    value: label1=value1,label2=value2
```

## STRIMZI\_KAFKA\_IMAGES

Required. The mapping from the Kafka version to the corresponding image containing a Kafka broker for that version. For example `4.0.0=quay.io/strimzi/kafka:0.50.0-kafka-4.0.0`, `4.1.1=quay.io/strimzi/kafka:0.50.0-kafka-4.1.1`.

## STRIMZI\_KAFKA\_CONNECT\_IMAGES

Required. The mapping from the Kafka version to the corresponding image of Kafka Connect for that version. For example `4.0.0=quay.io/strimzi/kafka:0.50.0-kafka-4.0.0`, `4.1.1=quay.io/strimzi/kafka:0.50.0-kafka-4.1.1`.

## STRIMZI\_KAFKA\_MIRROR MAKER2\_IMAGES

Required. The mapping from the Kafka version to the corresponding image of MirrorMaker 2 for that version. For example `4.0.0=quay.io/strimzi/kafka:0.50.0-kafka-4.0.0`, `4.1.1=quay.io/strimzi/kafka:0.50.0-kafka-4.1.1`.

## STRIMZI\_DEFAULT\_TOPIC\_OPERATOR\_IMAGE

Optional. The default is `quay.io/strimzi/operator:0.50.0`. The image name to use as the default when deploying the Topic Operator if no image is specified as the `Kafka.spec.entityOperator.topicOperator.image` in the `Kafka` resource.

## STRIMZI\_DEFAULT\_USER\_OPERATOR\_IMAGE

Optional. The default is `quay.io/strimzi/operator:0.50.0`. The image name to use as the default when deploying the User Operator if no image is specified as the `Kafka.spec.entityOperator.userOperator.image` in the `Kafka` resource.

## STRIMZI\_DEFAULT\_KAFKA\_EXPORTER\_IMAGE

Optional. The default is `quay.io/strimzi/kafka:0.50.0-kafka-4.1.1`. The image name to use as the default when deploying the Kafka Exporter if no image is specified as the `Kafka.spec.kafkaExporter.image` in the `Kafka` resource.

## STRIMZI\_DEFAULT\_CRUISE\_CONTROL\_IMAGE

Optional. The default is `quay.io/strimzi/kafka:0.50.0-kafka-4.1.1`. The image name to use as the default when deploying Cruise Control if no image is specified as the `Kafka.spec.cruiseControl.image` in the `Kafka` resource.

## STRIMZI\_DEFAULT\_KAFKA\_BRIDGE\_IMAGE

Optional. The default is `quay.io/strimzi/kafka-bridge:0.33.1`. The image name to use as the default when deploying the HTTP Bridge if no image is specified as the `Kafka.spec.kafkaBridge.image` in the `Kafka` resource.

## **STRIMZI\_DEFAULT\_KAFKA\_INIT\_IMAGE**

Optional. The default is `quay.io/strimzi/operator:0.50.0`. The image name to use as the default for the Kafka initializer container if no image is specified in the `brokerRackInitImage` of the `Kafka` resource or the `clientRackInitImage` of the Kafka Connect resource. The init container is started before the Kafka cluster for initial configuration work, such as rack support.

## **STRIMZI\_IMAGE\_PULL\_POLICY**

Optional. The `ImagePullPolicy` that is applied to containers in all pods managed by the Cluster Operator. The valid values are `Always`, `IfNotPresent`, and `Never`. If not specified, the Kubernetes defaults are used. Changing the policy will result in a rolling update of all your Kafka, Kafka Connect, and Kafka MirrorMaker clusters.

## **STRIMZI\_IMAGE\_PULL\_SECRETS**

Optional. A comma-separated list of `Secret` names. The secrets referenced here contain the credentials to the container registries where the container images are pulled from. The secrets are specified in the `imagePullSecrets` property for all pods created by the Cluster Operator. Changing this list results in a rolling update of all your Kafka, Kafka Connect, and Kafka MirrorMaker clusters.

## **STRIMZI\_KUBERNETES\_VERSION**

Optional. Overrides the Kubernetes version information detected from the API server.

*Example configuration for Kubernetes version override*

```
env:
  - name: STRIMZI_KUBERNETES_VERSION
    value: |
      major=1
      minor=16
      gitVersion=v1.16.2
      gitCommit=c97fe5036ef3df2967d086711e6c0c405941e14b
      gitTreeState=clean
      buildDate=2019-10-15T19:09:08Z
      goVersion=go1.12.10
      compiler=gc
      platform=linux/amd64
```

## **KUBERNETES\_SERVICE\_DNS\_DOMAIN**

Optional. Overrides the default Kubernetes DNS domain name suffix.

By default, services assigned in the Kubernetes cluster have a DNS domain name that uses the default suffix `cluster.local`.

For example, for broker `kafka-0`:

```
<cluster-name>-kafka-0.<cluster-name>-kafka-brokers.<namespace>.svc.cluster.local
```

The DNS domain name is added to the Kafka broker certificates used for hostname verification.

If you are using a different DNS domain name suffix in your cluster, change the `KUBERNETES_SERVICE_DNS_DOMAIN` environment variable from the default to the one you are using in order to establish a connection with the Kafka brokers.

#### **STRIMZI\_CONNECT\_BUILD\_TIMEOUT\_MS**

Optional, default 300000 ms. The timeout for building new Kafka Connect images with additional connectors, in milliseconds. Consider increasing this value when using Strimzi to build container images containing many connectors or using a slow container registry.

#### **STRIMZI\_NETWORK\_POLICY\_GENERATION**

Optional, default `true`. Network policy for resources. Network policies allow connections between Kafka components.

Set this environment variable to `false` to disable network policy generation. You might do this, for example, if you want to use custom network policies. Custom network policies allow more control over maintaining the connections between components.

#### **STRIMZI\_DNS\_CACHE\_TTL**

Optional, default `30`. Number of seconds to cache successful name lookups in local DNS resolver. Any negative value means cache forever. Zero means do not cache, which can be useful for avoiding connection errors due to long caching policies being applied.

#### **STRIMZI\_POD\_SET\_RECONCILIATION\_ONLY**

Optional, default `false`. When set to `true`, the Cluster Operator reconciles only the `StrimziPodSet` resources and any changes to the other custom resources (`Kafka`, `KafkaConnect`, and so on) are ignored. This mode is useful for ensuring that your pods are recreated if needed, but no other changes happen to the clusters.

#### **STRIMZI\_FEATURE\_GATES**

Optional. Enables or disables the features and functionality controlled by [feature gates](#).

#### **STRIMZI\_POD\_SECURITY\_PROVIDER\_CLASS**

Optional. Configuration for the pluggable `PodSecurityProvider` class, which can be used to provide the security context configuration for Pods and containers.

### **11.6.1. Restricting access to the Cluster Operator using network policy**

Use the `STRIMZI_OPERATOR_NAMESPACE_LABELS` environment variable to establish network policy for the Cluster Operator using namespace labels.

The Cluster Operator can run in the same namespace as the resources it manages, or in a separate namespace. By default, the `STRIMZI_OPERATOR_NAMESPACE` environment variable is configured to use the downward API to find the namespace the Cluster Operator is running in. If the Cluster Operator is running in the same namespace as the resources, only local access is required and allowed by Strimzi.

If the Cluster Operator is running in a separate namespace to the resources it manages, any namespace in the Kubernetes cluster is allowed access to the Cluster Operator unless network policy is configured. By adding namespace labels, access to the Cluster Operator is restricted to the

namespaces specified.

*Network policy configured for the Cluster Operator deployment*

```
#...
env:
# ...
- name: STRIMZI_OPERATOR_NAMESPACE_LABELS
  value: label1=value1,label2=value2
#...
```

## 11.6.2. Setting periodic reconciliation of custom resources

Use the `STRIMZI_FULL_RECONCILIATION_INTERVAL_MS` variable to set the time interval for periodic reconciliations by the Cluster Operator. Replace its value with the required interval in milliseconds.

*Reconciliation period configured for the Cluster Operator deployment*

```
#...
env:
# ...
- name: STRIMZI_FULL_RECONCILIATION_INTERVAL_MS
  value: "120000"
#...
```

The Cluster Operator reacts to all notifications about applicable cluster resources received from the Kubernetes cluster. If the operator is not running, or if a notification is not received for any reason, resources will get out of sync with the state of the running Kubernetes cluster. In order to handle failovers properly, a periodic reconciliation process is executed by the Cluster Operator so that it can compare the state of the resources with the current cluster deployments in order to have a consistent state across all of them.

*Additional resources*

- [Downward API](#)

## 11.6.3. Pausing reconciliation of custom resources using annotations

Sometimes it is useful to pause the reconciliation of custom resources managed by Strimzi operators, so that you can perform fixes or make updates. If reconciliations are paused, any changes made to custom resources are ignored by the operators until the pause ends.

If you want to pause reconciliation of a custom resource, set the `strimzi.io/pause-reconciliation` annotation to `true` in its configuration. This instructs the appropriate operator to pause reconciliation of the custom resource. For example, you can apply the annotation to the `KafkaConnect` resource so that reconciliation by the Cluster Operator is paused.

You can also create a custom resource with the pause annotation enabled. The custom resource is created, but it is ignored.

## Prerequisites

- The Strimzi Operator that manages the custom resource is running.

## Procedure

1. Annotate the custom resource in Kubernetes, setting `pause-reconciliation` to `true`:

```
kubectl annotate <kind_of_custom_resource> <name_of_custom_resource>
strimzi.io/pause-reconciliation="true"
```

For example, for the `KafkaConnect` custom resource:

```
kubectl annotate KafkaConnect my-connect strimzi.io/pause-reconciliation="true"
```

2. Check that the status conditions of the custom resource show a change to `ReconciliationPaused`:

```
kubectl describe <kind_of_custom_resource> <name_of_custom_resource>
```

The `type` condition changes to `ReconciliationPaused` at the `lastTransitionTime`.

*Example custom resource with a paused reconciliation condition type*

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaConnect
metadata:
  annotations:
    strimzi.io/pause-reconciliation: "true"
    strimzi.io/use-connector-resources: "true"
  creationTimestamp: 2021-03-12T10:47:11Z
  #...
spec:
  # ...
status:
  conditions:
  - lastTransitionTime: 2021-03-12T10:47:41.689249Z
    status: "True"
    type: ReconciliationPaused
```

## Resuming from pause

- To resume reconciliation, you can set the annotation to `false`, or remove the annotation.

## Additional resources

- [Finding the status of a custom resource](#)

### 11.6.4. Running multiple Cluster Operator replicas with leader election

The default Cluster Operator configuration enables leader election to run multiple parallel replicas

of the Cluster Operator. One replica is elected as the active leader and operates the deployed resources. The other replicas run in standby mode. When the leader stops or fails, one of the standby replicas is elected as the new leader and starts operating the deployed resources.

By default, Strimzi runs with a single Cluster Operator replica that is always the leader replica. When a single Cluster Operator replica stops or fails, Kubernetes starts a new replica.

Running the Cluster Operator with multiple replicas is not essential. But it's useful to have replicas on standby in case of large-scale disruptions caused by major failure. For example, suppose multiple worker nodes or an entire availability zone fails. This failure might cause the Cluster Operator pod and many Kafka pods to go down at the same time. If subsequent pod scheduling causes congestion through lack of resources, this can delay operations when running a single Cluster Operator.

### Enabling leader election for Cluster Operator replicas

Configure leader election environment variables when running additional Cluster Operator replicas. The following environment variables are supported:

#### **STRIMZI\_LEADER\_ELECTION\_ENABLED**

Optional, disabled (`false`) by default. Enables or disables leader election, which allows additional Cluster Operator replicas to run on standby.

**NOTE**

Leader election is disabled by default. It is only enabled when applying this environment variable on installation.

#### **STRIMZI\_LEADER\_ELECTIONLEASE\_NAME**

Required when leader election is enabled. The name of the Kubernetes `Lease` resource that is used for the leader election.

#### **STRIMZI\_LEADER\_ELECTIONLEASE\_NAMESPACE**

Required when leader election is enabled. The namespace where the Kubernetes `Lease` resource used for leader election is created. You can use the downward API to configure it to the namespace where the Cluster Operator is deployed.

```
env:  
  - name: STRIMZI_LEADER_ELECTIONLEASE_NAMESPACE  
    valueFrom:  
      fieldRef:  
        fieldPath: metadata.namespace
```

#### **STRIMZI\_LEADER\_ELECTION\_IDENTITY**

Required when leader election is enabled. Configures the identity of a given Cluster Operator instance used during the leader election. The identity must be unique for each operator instance. You can use the downward API to configure it to the name of the pod where the Cluster Operator is deployed.

```
env:  
  - name: STRIMZI_LEADER_ELECTION_IDENTITY  
    valueFrom:  
      fieldRef:  
        fieldPath: metadata.name
```

### **STRIMZI\_LEADER\_ELECTIONLEASE\_DURATION\_MS**

Optional, default 15000 ms. Specifies the duration the acquired lease is valid.

### **STRIMZI\_LEADER\_ELECTION\_RENEW\_DEADLINE\_MS**

Optional, default 10000 ms. Specifies the period the leader should try to maintain leadership.

### **STRIMZI\_LEADER\_ELECTION\_RETRY\_PERIOD\_MS**

Optional, default 2000 ms. Specifies the frequency of updates to the lease lock by the leader.

## **Configuring Cluster Operator replicas**

To run additional Cluster Operator replicas in standby mode, you will need to increase the number of replicas and enable leader election. To configure leader election, use the leader election environment variables.

To make the required changes, configure the following Cluster Operator installation files located in `install/cluster-operator/`:

- 060-Deployment-strimzi-cluster-operator.yaml
- 022-ClusterRole-strimzi-cluster-operator-role.yaml
- 022-RoleBinding-strimzi-cluster-operator.yaml

Leader election has its own `ClusterRole` and `RoleBinding` RBAC resources that target the namespace where the Cluster Operator is running, rather than the namespace it is watching.

The default deployment configuration creates a `Lease` resource called `strimzi-cluster-operator` in the same namespace as the Cluster Operator. The Cluster Operator uses leases to manage leader election. The RBAC resources provide the permissions to use the `Lease` resource. If you use a different `Lease` name or namespace, update the `ClusterRole` and `RoleBinding` files accordingly.

### *Prerequisites*

- You need an account with permission to create and manage `CustomResourceDefinition` and RBAC (`ClusterRole`, and `RoleBinding`) resources.

### *Procedure*

Edit the `Deployment` resource that is used to deploy the Cluster Operator, which is defined in the `060-Deployment-strimzi-cluster-operator.yaml` file.

1. Change the `replicas` property from the default (1) to a value that matches the required number of replicas.

## *Increasing the number of Cluster Operator replicas*

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: strimzi-cluster-operator
  labels:
    app: strimzi
spec:
  replicas: 3
```

2. Check that the leader election `env` properties are set.

If they are not set, configure them.

To enable leader election, `STRIMZI_LEADER_ELECTION_ENABLED` must be set to `true` (default).

In this example, the name of the lease is changed to `my-strimzi-cluster-operator`.

### *Configuring leader election environment variables for the Cluster Operator*

```
# ...
spec
containers:
- name: strimzi-cluster-operator
# ...
env:
- name: STRIMZI_LEADER_ELECTION_ENABLED
  value: "true"
- name: STRIMZI_LEADER_ELECTIONLEASE_NAME
  value: "my-strimzi-cluster-operator"
- name: STRIMZI_LEADER_ELECTIONLEASE_NAMESPACE
  valueFrom:
    fieldRef:
      fieldPath: metadata.namespace
- name: STRIMZI_LEADER_ELECTION_IDENTITY
  valueFrom:
    fieldRef:
      fieldPath: metadata.name
```

For a description of the available environment variables, see [Enabling leader election for Cluster Operator replicas](#).

If you specified a different name or namespace for the `Lease` resource used in leader election, update the RBAC resources.

3. (optional) Edit the `ClusterRole` resource in the `022-ClusterRole-strimzi-cluster-operator-role.yaml` file.

Update `resourceNames` with the name of the `Lease` resource.

*Updating the ClusterRole references to the lease*

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: strimzi-cluster-operator-leader-election
  labels:
    app: strimzi
rules:
- apiGroups:
  - coordination.k8s.io
resourceNames:
  - my-strimzi-cluster-operator
# ...
```

4. (optional) Edit the `RoleBinding` resource in the `022-RoleBinding-strimzi-cluster-operator.yaml` file.

Update `subjects.name` and `subjects.namespace` with the name of the `Lease` resource and the namespace where it was created.

*Updating the RoleBinding references to the lease*

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: strimzi-cluster-operator-leader-election
  labels:
    app: strimzi
subjects:
- kind: ServiceAccount
  name: my-strimzi-cluster-operator
  namespace: myproject
# ...
```

5. Deploy the Cluster Operator:

```
kubectl create -f install/cluster-operator -n myproject
```

6. Check the status of the deployment:

```
kubectl get deployments -n myproject
```

*Output shows the deployment name and readiness*

NAME	READY	UP-TO-DATE	AVAILABLE
strimzi-cluster-operator	3/3	3	3

**READY** shows the number of replicas that are ready/expected. The deployment is successful when the **AVAILABLE** output shows the correct number of replicas.

## 11.6.5. Configuring Cluster Operator HTTP proxy settings

If you are running a Kafka cluster behind a HTTP proxy, you can still pass data in and out of the cluster. For example, you can run Kafka Connect with connectors that push and pull data from outside the proxy. Or you can use a proxy to connect with an authorization server.

Configure the Cluster Operator deployment to specify the proxy environment variables. The Cluster Operator accepts standard proxy configuration (`HTTP_PROXY`, `HTTPS_PROXY` and `NO_PROXY`) as environment variables. The proxy settings are applied to all Strimzi containers.

The format for a proxy address is `http://<ip_address>:<port_number>`. To set up a proxy with a name and password, the format is `http://<username>:<password>@<ip-address>:<port_number>`.

### Prerequisites

- You need an account with permission to create and manage `CustomResourceDefinition` and RBAC (`ClusterRole`, and `RoleBinding`) resources.

### Procedure

1. To add proxy environment variables to the Cluster Operator, update its `Deployment` configuration ([install/cluster-operator/060-Deployment-stimzi-cluster-operator.yaml](#)).

#### Example proxy configuration for the Cluster Operator

```
apiVersion: apps/v1
kind: Deployment
spec:
  # ...
  template:
    spec:
      serviceAccountName: stimzi-cluster-operator
      containers:
        # ...
        env:
          # ...
          - name: "HTTP_PROXY"
            value: "http://proxy.com" ①
          - name: "HTTPS_PROXY"
            value: "https://proxy.com" ②
          - name: "NO_PROXY"
            value: "internal.com, other.domain.com" ③
        # ...
```

① Address of the proxy server.

② Secure address of the proxy server.

③ Addresses for servers that are accessed directly as exceptions to the proxy server. The URLs are comma-separated.

Alternatively, edit the [Deployment](#) directly:

```
kubectl edit deployment strimzi-cluster-operator
```

2. If you updated the YAML file instead of editing the [Deployment](#) directly, apply the changes:

```
kubectl create -f install/cluster-operator/060-Deployment-strimzi-cluster-operator.yaml
```

#### *Additional resources*

- [Host aliases](#)
- [Designating Strimzi administrators](#)

### 11.6.6. Disabling FIPS mode using Cluster Operator configuration

Strimzi automatically switches to FIPS mode when running on a FIPS-enabled Kubernetes cluster. Disable FIPS mode by setting the [FIPS\\_MODE](#) environment variable to [disabled](#) in the deployment configuration for the Cluster Operator. With FIPS mode disabled, Strimzi automatically disables FIPS in the OpenJDK for all components. With FIPS mode disabled, Strimzi is not FIPS compliant. The Strimzi operators, as well as all operands, run in the same way as if they were running on an Kubernetes cluster without FIPS enabled.

#### *Procedure*

1. To disable the FIPS mode in the Cluster Operator, update its [Deployment](#) configuration ([install/cluster-operator/060-Deployment-strimzi-cluster-operator.yaml](#)) and add the [FIPS\\_MODE](#) environment variable.

#### *Example FIPS configuration for the Cluster Operator*

```
apiVersion: apps/v1
kind: Deployment
spec:
  # ...
  template:
    spec:
      serviceAccountName: strimzi-cluster-operator
      containers:
        # ...
        env:
          # ...
          - name: "FIPS_MODE"
            value: "disabled" ①
        # ...
```

① Disables the FIPS mode.

Alternatively, edit the [Deployment](#) directly:

```
kubectl edit deployment strimzi-cluster-operator
```

2. If you updated the YAML file instead of editing the [Deployment](#) directly, apply the changes:

```
kubectl apply -f install/cluster-operator/060-Deployment-strimzi-cluster-operator.yaml
```

## 11.7. Configuring Kafka Connect

Update the `spec` properties of the [KafkaConnect](#) custom resource to configure your Kafka Connect deployment.

Use Kafka Connect to set up external data connections to your Kafka cluster. Use the properties of the [KafkaConnect](#) resource to configure your Kafka Connect deployment.

You can also use the [KafkaConnect](#) resource to specify the following:

- Connector plugin configuration to build a container image that includes the plugins to make connections
- Configuration for the Kafka Connect worker pods that run connectors
- An annotation to enable use of the [KafkaConnector](#) resource to manage connectors

The Cluster Operator manages Kafka Connect clusters deployed using the [KafkaConnect](#) resource and connectors created using the [KafkaConnector](#) resource.

For a deeper understanding of the Kafka Connect cluster configuration options, refer to the [Strimzi Custom Resource API Reference](#).

*Handling high volumes of messages*

You can tune the configuration to handle high volumes of messages. For more information, see [Handling high volumes of messages](#).

*Example KafkaConnect custom resource configuration*

```
# Basic configuration (required)
apiVersion: kafka.strimzi.io/v1
kind: KafkaConnect ①
metadata:
  name: my-connect-cluster
  annotations:
    strimzi.io/use-connector-resources: "true" ②
# Deployment specifications
spec:
  # Replicas (required)
  replicas: 3 ③
  # Bootstrap servers (required)
  bootstrapServers: my-cluster-kafka-bootstrap:9092 ④
```

```

# Connect cluster group ID (required in v1 CRD API)
groupId: my-connect-cluster ⑤
# Connect cluster storage topics (required in v1 CRD API)
configStorageTopic: my-connect-cluster-configs ⑥
offsetStorageTopic: my-connect-cluster-offsets ⑦
statusStorageTopic: my-connect-cluster-status ⑧
# Kafka Connect configuration (recommended)
config: ⑨
  key.converter: org.apache.kafka.connect.json.JsonConverter
  value.converter: org.apache.kafka.connect.json.JsonConverter
  key.converter.schemas.enable: true
  value.converter.schemas.enable: true
  config.storage.replication.factor: 3
  offset.storage.replication.factor: 3
  status.storage.replication.factor: 3
# Resources requests and limits (recommended)
resources: ⑩
  requests:
    cpu: "1"
    memory: 2Gi
  limits:
    cpu: "2"
    memory: 2Gi
# Authentication (optional)
authentication: ⑪
  type: tls
  certificateAndKey:
    certificate: source.crt
    key: source.key
    secretName: my-user-source
# TLS configuration (optional)
tls: ⑫
  trustedCertificates:
    - secretName: my-cluster-cluster-cert
      pattern: "*.crt"
    - secretName: my-cluster-cluster-cert
      pattern: "*.crt"
# Build configuration (optional)
build: ⑬
  output: ⑭
    type: docker
    image: my-registry.io/my-org/my-connect-cluster:latest
    pushSecret: my-registry-credentials
  plugins: ⑮
    - name: connector-1
      artifacts:
        - type: tgz
          url: <url_to_download_connector_1_artifact>
          sha512sum: <SHA-512_checksum_of_connector_1_artifact>
    - name: connector-2
      artifacts:

```

```

    - type: jar
      url: <url_to_download_connector_2_artifact>
      sha512sum: <SHA-512_checksum_of_connector_2_artifact>
# Logging configuration (optional)
logging: ⑯
  type: inline
  loggers:
    # Kafka 4.0+ uses Log4j2
    rootLogger.level: INFO
# Readiness probe (optional)
readinessProbe: ⑰
  initialDelaySeconds: 15
  timeoutSeconds: 5
# Liveness probe (optional)
livenessProbe:
  initialDelaySeconds: 15
  timeoutSeconds: 5
# Metrics configuration (optional)
metricsConfig: ⑱
  type: jmxPrometheusExporter
  valueFrom:
    configMapKeyRef:
      name: my-config-map
      key: my-key
# JVM options (optional)
jvmOptions: ⑲
  "-Xmx": "1g"
  "-Xms": "1g"
# Custom image (optional)
image: my-org/my-image:latest ⑳
# Rack awareness (optional)
rack:
  topologyKey: topology.kubernetes.io/zone
# Pod and container template (optional)
template:
  pod:
    affinity:
      podAntiAffinity:
        requiredDuringSchedulingIgnoredDuringExecution:
          - labelSelector:
              matchExpressions:
                - key: application
                  operator: In
                  values:
                    - postgresql
                    - mongodb
            topologyKey: "kubernetes.io/hostname"
connectContainer:
  env:
    - name: OTEL_SERVICE_NAME
      value: my-otel-service

```

```

- name: OTEL_EXPORTER_OTLP_ENDPOINT
  value: "http://otlp-host:4317"
- name: AWS_ACCESS_KEY_ID
  valueFrom:
    secretKeyRef:
      name: aws-creds
      key: awsAccessKey
- name: AWS_SECRET_ACCESS_KEY
  valueFrom:
    secretKeyRef:
      name: aws-creds
      key: awsSecretAccessKey
# Tracing configuration (optional)
tracing:
  type: opentelemetry

```

- ① Use [KafkaConnect](#).
- ② Enables the use of [KafkaConnector](#) resources to start, stop, and manage connector instances.
- ③ The number of replica nodes for the workers that run tasks.
- ④ Bootstrap address for connection to the Kafka cluster. The address takes the format `<cluster_name>-kafka-bootstrap:<port_number>`. The Kafka cluster doesn't need to be managed by Strimzi or deployed to a Kubernetes cluster.
- ⑤ A unique ID that identifies the Connect cluster group. Required in [v1](#) CRD API.
- ⑥ Name of the Kafka topic where connector configurations are stored. Required in [v1](#) CRD API.
- ⑦ Name of the Kafka topic where source connector offsets are stored. Required in [v1](#) CRD API.
- ⑧ Name of the Kafka topic where connector and task statuses are stored. Required in [v1](#) CRD API.
- ⑨ Kafka Connect configuration of workers (not connectors) that run connectors and their tasks. Standard Apache Kafka configuration may be provided, restricted to those properties not managed directly by Strimzi. In this example, JSON converters are specified. A replication factor of 3 is set for the internal topics used by Kafka Connect (minimum requirement for production environment). Changing the replication factor after the topics have been created has no effect.
- ⑩ Requests for reservation of supported resources, currently `cpu` and `memory`, and limits to specify the maximum resources that can be consumed.
- ⑪ Authentication for the Kafka Connect cluster, specified as `tls`, `scram-sha-256`, `scram-sha-512`, `plain`, or `custom`. By default, Kafka Connect connects to Kafka brokers using a plaintext connection. For details on configuring authentication, see the [KafkaConnectSpec schema properties](#).
- ⑫ TLS configuration for encrypted connections to the Kafka cluster, with trusted certificates stored in X.509 format within the specified secrets.
- ⑬ Build configuration properties for building a container image with connector plugins automatically.
- ⑭ (Required) Configuration of the container registry where new images are pushed.
- ⑮ (Required) List of connector plugins and their artifacts to add to the new container image. Each

plugin must be configured with at least one [artifact](#).

- ⑯ Kafka Connect loggers and log levels added directly ([inline](#)) or indirectly ([external](#)) through a [ConfigMap](#). Custom Log4j configuration must be placed under the [log4j2.properties](#) key in the [ConfigMap](#). You can set log levels to [INFO](#), [ERROR](#), [WARN](#), [TRACE](#), [DEBUG](#), [FATAL](#) or [OFF](#).
- ⑰ Healthchecks to know when to restart a container (liveness) and when a container can accept traffic (readiness).
- ⑱ Prometheus metrics, which are enabled by referencing a ConfigMap containing configuration for the Prometheus JMX exporter in this example. You can enable metrics without further configuration using a reference to a ConfigMap containing an empty file under [metricsConfig.valueFrom.configMapKeyRef.key](#).
- ⑲ JVM configuration options to optimize performance for the Virtual Machine (VM) running Kafka Connect.
- ⑳ ADVANCED OPTION: Container image configuration, which is recommended only in special situations.

SPECIALIZED OPTION: Rack awareness configuration for the deployment. This is a specialized option intended for a deployment within the same location, not across regions. Use this option if you want connectors to consume from the closest replica rather than the leader replica. In certain cases, consuming from the closest replica can improve network utilization or reduce costs . The [topologyKey](#) must match a node label containing the rack ID. The example used in this configuration specifies a zone using the standard [topology.kubernetes.io/zone](#) label. To consume from the closest replica, enable the [RackAwareReplicaSelector](#) in the Kafka broker configuration.

Template customization. Here a pod is scheduled with anti-affinity, so the pod is not scheduled on nodes with the same hostname.

Environment variables are set for distributed tracing and to pass credentials to connectors.

Distributed tracing is enabled by using OpenTelemetry.

### 11.7.1. Configuring Kafka Connect for multiple instances

When running multiple instances of Kafka Connect, you must make sure that these settings are different for each instance:

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaConnect
metadata:
  name: my-connect
spec:
  groupId: my-connect-cluster ①
  configStorageTopic: my-connect-cluster-configs ②
  offsetStorageTopic: my-connect-cluster-offsets ③
  statusStorageTopic: my-connect-cluster-status ④
  # ...
```

① The Kafka Connect cluster group ID within Kafka.

② Kafka topic that stores connector and task configurations.

- ③ Kafka topic that stores connector offsets.
- ④ Kafka topic that stores connector and task status updates.

This requirement applies across all Kafka Connect-based deployments, including Kafka Connect and MirrorMaker 2 instances that target the same Kafka cluster.

Unless you use different settings for each instance, all instances form a cluster and use the same internal topics. Multiple instances attempting to use the same internal topics will cause unexpected errors, so you must change the values of these properties for each instance.

### 11.7.2. Configuring Kafka Connect user authorization

When using authorization in Kafka, a Kafka Connect user requires read/write access to the cluster group and internal topics of Kafka Connect. This procedure outlines how access is granted using [simple](#) authorization and ACLs.

Properties for the Kafka Connect cluster group ID and internal topics are configured by Strimzi by default. Alternatively, you can define them explicitly in the [spec](#) of the [KafkaConnect](#) resource. This is useful when [configuring Kafka Connect for multiple instances](#), as the values for the group ID and topics must differ when running multiple Kafka Connect instances.

Simple authorization uses ACL rules managed by the Kafka [StandardAuthorizer](#) plugin to ensure appropriate access levels. For more information on configuring a [KafkaUser](#) resource to use simple authorization, see the [AclRule schema reference](#).

#### Prerequisites

- A Kubernetes cluster
- A running Cluster Operator

#### Procedure

1. Edit the [authorization](#) property in the [KafkaUser](#) resource to provide access rights to the user.

Access rights are configured for the Kafka Connect topics and cluster group using [literal](#) name values. The following table shows the default names configured for the topics and cluster group ID.

*Table 11. Names for the access rights configuration*

Property	Name
<code>offset.storage.topic</code>	<code>connect-cluster-offsets</code>
<code>status.storage.topic</code>	<code>connect-cluster-status</code>
<code>config.storage.topic</code>	<code>connect-cluster-configs</code>
<code>group</code>	<code>connect-cluster</code>

In this example configuration, the default names are used to specify access rights. If you are using different names for a Kafka Connect instance, use those names in the ACLs configuration.

*Example configuration for simple authorization*

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaUser
metadata:
  name: my-user
  labels:
    strimzi.io/cluster: my-cluster
spec:
  # ...
  authorization:
    type: simple
    acls:
      # access to offset.storage.topic
      - resource:
          type: topic
          name: connect-cluster-offsets
          patternType: literal
        operations:
          - Create
          - Describe
          - Read
          - Write
        host: "*"
      # access to status.storage.topic
      - resource:
          type: topic
          name: connect-cluster-status
          patternType: literal
        operations:
          - Create
          - Describe
          - Read
          - Write
        host: "*"
      # access to config.storage.topic
      - resource:
          type: topic
          name: connect-cluster-configs
          patternType: literal
        operations:
          - Create
          - Describe
          - Read
          - Write
        host: "*"
      # cluster group
      - resource:
          type: group
          name: connect-cluster
          patternType: literal
```

```
operations:
```

```
  - Read
```

```
host: "*"
```

2. Create or update the resource.

```
kubectl apply -f KAFKA-USER-CONFIG-FILE
```

## 11.8. Configuring Kafka Connect connectors

The `KafkaConnector` resource provides a Kubernetes-native approach to management of connectors by the Cluster Operator. To create, delete, or reconfigure connectors with `KafkaConnector` resources, you must set the `use-connector-resources` annotation to `true` in your `KafkaConnect` custom resource.

*Annotation to enable KafkaConnectors*

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaConnect
metadata:
  name: my-connect-cluster
  annotations:
    strimzi.io/use-connector-resources: "true"
# ...
```

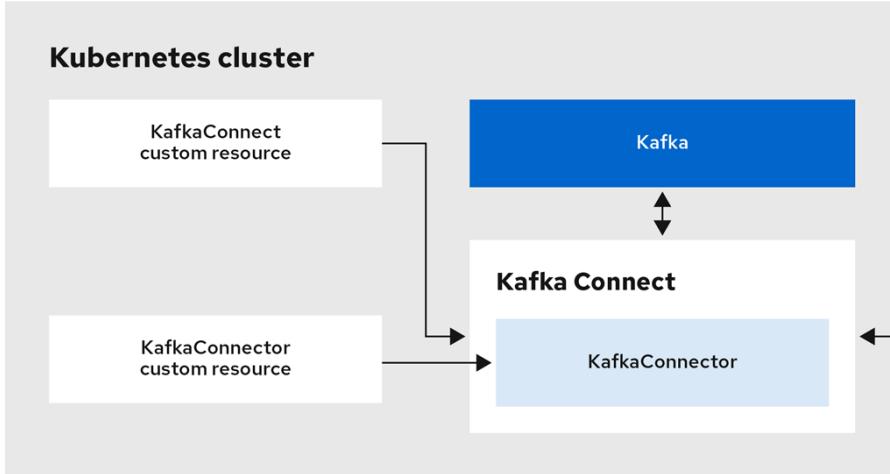
When the `use-connector-resources` annotation is enabled in your `KafkaConnect` configuration, you must define and manage connectors using `KafkaConnector` resources.

**NOTE**

Alternatively, you can manage connectors using the Kafka Connect REST API instead of `KafkaConnector` resources. To use the API, you must remove the `strimzi.io/use-connector-resources` annotation to use `KafkaConnector` resources in the `KafkaConnect` the resource.

`KafkaConnector` resources provide the configuration needed to create connectors within a Kafka Connect cluster, which interacts with a Kafka cluster as specified in the `KafkaConnect` configuration. The Kafka cluster does not need to be managed by Strimzi or deployed to a Kubernetes cluster.

*Kafka components contained in the same Kubernetes cluster*



195\_1121

The configuration also specifies how the connector instances interact with external data systems, including any required authentication methods. Additionally, you must define the data to watch. For example, in a source connector that reads data from a database, the configuration might include the database name. You can also define where this data should be placed in Kafka by specifying the target topic name.

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

#### *Example source connector configuration*

```

apiVersion: kafka.strimzi.io/v1
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
  autoRestart: ⑤
    enabled: true
  config: ⑥
    file: "/opt/kafka/LICENSE" ⑦
    topic: my-topic ⑧
  # ...

```

- 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 Enables automatic restarts of failed connectors and tasks. By default, the number of restarts is indefinite, but you can set a maximum on the number of automatic restarts using the `maxRestarts` property.
- ⑥ [Connector configuration](#) as key-value pairs.
- ⑦ Location of the external data file. In this example, we're configuring the `FileStreamSourceConnector` to read from the `/opt/kafka/LICENSE` file.
- ⑧ Kafka topic to publish the source data to.

To include external connector configurations, such as user access credentials stored in a secret, use the `template` property of the `KafkaConnect` resource. You can also load values using [configuration providers](#).

### 11.8.1. Manually stopping or pausing Kafka Connect connectors

If you are using `KafkaConnector` resources to configure connectors, use the `state` configuration to either stop or pause a connector. In contrast to the paused state, where the connector and tasks remain instantiated, stopping a connector retains only the configuration, with no active processes. Stopping a connector from running may be more suitable for longer durations than just pausing. While a paused connector is quicker to resume, a stopped connector has the advantages of freeing up memory and resources.

**NOTE**

The `state` configuration replaces the (deprecated) `pause` configuration in the `KafkaConnectorSpec` schema, which allows pauses on connectors. If you were previously using the `pause` configuration to pause connectors, we encourage you to transition to using the `state` configuration only to avoid conflicts.

*Prerequisites*

- The Cluster Operator is running.

*Procedure*

1. Find the name of the `KafkaConnector` custom resource that controls the connector you want to pause or stop:

```
kubectl get KafkaConnector
```

2. Edit the `KafkaConnector` resource to stop or pause the connector.

*Example configuration for stopping a Kafka Connect connector*

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaConnector
metadata:
  name: my-source-connector
  labels:
    strimzi.io/cluster: my-connect-cluster
spec:
  class: org.apache.kafka.connect.file.FileStreamSourceConnector
```

```
tasksMax: 2
config:
  file: "/opt/kafka/LICENSE"
  topic: my-topic
state: stopped
# ...
```

Change the `state` configuration to `stopped` or `paused`. The default state for the connector when this property is not set is `running`.

3. Apply the changes to the `KafkaConnector` configuration.

You can resume the connector by changing `state` to `running` or removing the configuration.

**NOTE**

Alternatively, you can [expose the Kafka Connect API](#) and use the `stop` and `pause` endpoints to stop a connector from running. For example, `PUT /connectors/<connector_name>/stop`. You can then use the `resume` endpoint to restart it.

### 11.8.2. Manually restarting Kafka Connect connectors

If you are using `KafkaConnector` resources to manage connectors, use the `strimzi.io/restart` annotation to manually trigger a restart of a connector.

#### *Prerequisites*

- The Cluster Operator is running.

#### *Procedure*

1. Find the name of the `KafkaConnector` custom resource that controls the Kafka connector you want to restart:

```
kubectl get KafkaConnector
```

2. Restart the connector by annotating the `KafkaConnector` resource in Kubernetes:

```
kubectl annotate KafkaConnector <kafka_connector_name> strimzi.io/restart="true"
```

The `restart` annotation is set to `true`.

You can also refine the restart behavior with the `includeTasks` and `onlyFailed` parameters, which both default to `false`.

- `includeTasks` restarts both the connector instance and its task instances.
- `onlyFailed` restarts only instances with a `FAILED` status when set to `true`.

For example:

```
strimzi.io/restart="includeTasks,onlyFailed"
```

3. Wait for the next reconciliation to occur (every two minutes by default).

The Kafka connector is restarted, as long as the annotation was detected by the reconciliation process. When Kafka Connect accepts the restart request, the annotation is removed from the [KafkaConnector](#) custom resource.

### 11.8.3. Manually restarting Kafka Connect connector tasks

If you are using [KafkaConnector](#) resources to manage connectors, use the [strimzi.io/restart-task](#) annotation to manually trigger a restart of a connector task.

#### Prerequisites

- The Cluster Operator is running.

#### Procedure

1. Find the name of the [KafkaConnector](#) custom resource that controls the Kafka connector task you want to restart:

```
kubectl get KafkaConnector
```

2. Find the ID of the task to be restarted from the [KafkaConnector](#) custom resource:

```
kubectl describe KafkaConnector <kafka_connector_name>
```

Task IDs are non-negative integers, starting from 0.

3. Use the ID to restart the connector task by annotating the [KafkaConnector](#) resource in Kubernetes:

```
kubectl annotate KafkaConnector <kafka_connector_name> strimzi.io/restart-task="0"
```

In this example, task **0** is restarted.

4. Wait for the next reconciliation to occur (every two minutes by default).

The Kafka connector task is restarted, as long as the annotation was detected by the reconciliation process. When Kafka Connect accepts the restart request, the annotation is removed from the [KafkaConnector](#) custom resource.

### 11.8.4. Listing connector offsets

To track connector offsets using [KafkaConnector](#) resources, add the [listOffsets](#) configuration. The offsets, which keep track of the flow of data, are written to a config map specified in the

configuration. If the config map does not exist, Strimzi creates it.

After the configuration is in place, annotate the `KafkaConnector` resource to write the list to the config map.

Sink connectors use Kafka's standard consumer offset mechanism, while source connectors store offsets in a custom format within a Kafka topic.

- For sink connectors, the list shows Kafka topic partitions and the last committed offset for each partition.
- For source connectors, the list shows the source system's partition and the last offset processed.

#### Prerequisites

- The Cluster Operator is running.

#### Procedure

1. Edit the `KafkaConnector` resource for the connector to include the `listOffsets` configuration.

##### *Example configuration to list offsets*

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaConnector
metadata:
  name: my-source-connector
  labels:
    strimzi.io/cluster: my-connect-cluster
spec:
  listOffsets:
    toConfigMap: ①
      name: my-connector-offsets ②
    # ...
```

① The reference to the config map where the list of offsets will be written to.

② The name of the config map, which is named `my-connector-offsets` in this example.

2. Run the command to write the list to the config map by annotating the `KafkaConnector` resource:

```
kubectl annotate kafkaconnector my-source-connector strimzi.io/connector-
offsets=list -n <namespace>
```

The annotation remains until either the list operation succeeds or it is manually removed from the resource.

3. After the `KafkaConnector` resource is updated, use the following command to check if the config map with the offsets was created:

```
kubectl get configmap my-connector-offsets -n <namespace>
```

4. Inspect the contents of the config map to verify the offsets are being listed:

```
kubectl describe configmap my-connector-offsets -n <namespace>
```

Strimzi puts the offset information into the `offsets.json` property. This does not overwrite any other properties when updating an existing config map.

*Example source connector offset list*

```
apiVersion: v1
kind: ConfigMap
metadata:
  # ...
  ownerReferences: ①
  - apiVersion: kafka.strimzi.io/v1
    blockOwnerDeletion: true
    controller: false
    kind: KafkaConnector
    name: my-source-connector
    uid: 637e3be7-bd96-43ab-abde-c55b4c4550e0
  resourceVersion: "66951"
  uid: 641d60a9-36eb-4f29-9895-8f2c1eb9638e
data:
  offsets.json: |-  
  {  
    "offsets" : [ {  
      "partition" : {  
        "filename" : "/data/myfile.txt" ②  
      },  
      "offset" : {  
        "position" : 15295 ③  
      }  
    } ]  
  }
```

① The owner reference pointing to the `KafkaConnector` resource for the source connector. To provide a custom owner reference, create the config map in advance and set the owner reference.

② The source partition, represented by the filename `/data/myfile.txt` in this example for a file-based connector.

③ The last processed offset position in the source partition.

*Example sink connector offset list*

```
apiVersion: v1
kind: ConfigMap
metadata:
  # ...
  ownerReferences: ①
```

```

- apiVersion: kafka.strimzi.io/v1
  blockOwnerDeletion: true
  controller: false
  kind: KafkaConnector
  name: my-sink-connector
  uid: 84a29d7f-77e6-43ac-bfbb-719f9b9a4b3b
  resourceVersion: "79241"
  uid: 721e30bc-23df-41a2-9b48-fb2b7d9b042c
  data:
    offsets.json: |-
      {
        "offsets": [
          {
            "partition": {
              "kafka_topic": "my-topic", ②
              "kafka_partition": 2 ③
            },
            "offset": {
              "kafka_offset": 4 ④
            }
          }
        ]
      }

```

① The owner reference pointing to the `KafkaConnector` resource for the sink connector.

② The Kafka topic that the sink connector is consuming from.

③ The partition of the Kafka topic.

④ The last committed Kafka offset for this topic and partition.

### 11.8.5. Altering connector offsets

To alter connector offsets using `KafkaConnector` resources, configure the resource to stop the connector and add `alterOffsets` configuration to specify the offset changes in a config map. You can reuse the same config map used to [list offsets](#).

After the connector is stopped and the configuration is in place, annotate the `KafkaConnector` resource to apply the offset alteration, then restart the connector.

Altering connector offsets can be useful, for example, to skip a *poison* record or replay a record.

In this procedure, we alter the offset position for a source connector named `my-source-connector`.

#### Prerequisites

- The Cluster Operator is running.

#### Procedure

1. Edit the `KafkaConnector` resource to stop the connector and include the `alterOffsets` configuration.

*Example configuration to stop a connector and alter offsets*

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaConnector
metadata:
  name: my-source-connector
  labels:
    strimzi.io/cluster: my-connect-cluster
spec:
  state: stopped ①
  alterOffsets:
    fromConfigMap: ②
    name: my-connector-offsets ③
  # ...
```

① Changes the state of the connector to `stopped`. The default state for the connector when this property is not set is `running`.

② The reference to the config map that provides the update.

③ The name of the config map, which is named `my-connector-offsets` in this example.

2. Edit the config map to make the alteration.

In this example, we're resetting the offset position for a source connector to 15000.

*Example source connector offset list configuration*

```
apiVersion: v1
kind: ConfigMap
metadata:
  # ...
data:
  offsets.json: |- ①
  {
    "offsets" : [ {
      "partition" : {
        "filename" : "/data/myfile.txt"
      },
      "offset" : {
        "position" : 15000 ②
      }
    } ]
  }
```

① Edits must be made within the `offsets.json` property.

② The updated offset position in the source partition.

3. Run the command to update the offset position by annotating the `KafkaConnector` resource:

```
kubectl annotate kafkaconnector my-source-connector strimzi.io/connector-
```

```
offsets=alter -n <namespace>
```

The annotation remains until either the update operation succeeds or it is manually removed from the resource.

4. Check the changes by using the procedure to [list connector offsets](#).
5. Restart the connector by changing the state to `running`.

*Example configuration to start a connector*

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaConnector
metadata:
  name: my-source-connector
  labels:
    strimzi.io/cluster: my-connect-cluster
spec:
  state: running
  # ...
```

### 11.8.6. Resetting connector offsets

To reset connector offsets using `KafkaConnector` resources, configure the resource to stop the connector.

After the connector is stopped, annotate the `KafkaConnector` resource to clear the offsets, then restart the connector.

In this procedure, we reset the offset position for a source connector named `my-source-connector`.

*Prerequisites*

- The Cluster Operator is running.

*Procedure*

1. Edit the `KafkaConnector` resource to stop the connector.

*Example configuration to stop a connector*

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaConnector
metadata:
  name: my-source-connector
  labels:
    strimzi.io/cluster: my-connect-cluster
spec:
  # ...
  state: stopped ①
  # ...
```

- ① Changes the state of the connector to `stopped`. The default state for the connector when this property is not set is `running`.
- Run the command to reset the offset position by annotating the `KafkaConnector` resource:

```
kubectl annotate kafkaconnector my-source-connector strimzi.io/connector-offsets=reset -n <namespace>
```

The annotation remains until either the reset operation succeeds or it is manually removed from the resource.

- Check the changes by using the procedure to [list connector offsets](#).

After resetting, the `offsets.json` property is empty.

*Example source connector offset list*

```
apiVersion: v1
kind: ConfigMap
metadata:
  # ...
data:
  offsets.json: |- 
    {
      "offsets" : []
    }
```

- Restart the connector by changing the state to `running`.

*Example configuration to start a connector*

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaConnector
metadata:
  name: my-source-connector
  labels:
    strimzi.io/cluster: my-connect-cluster
spec:
  state: running
  # ...
```

## 11.9. Configuring MirrorMaker 2

Configure your MirrorMaker 2 deployment by updating the `spec` properties of the `KafkaMirrorMaker2` custom resource.

Start with a minimal configuration that defines the core requirements. Extend it with optional settings to support the following:

- Securing connections (TLS and authentication)
- Managing of replicated topic names
- Synchronizing consumer group offsets
- Synchronizing ACL rules
- Tuning Kafka Connect worker settings
- [Common configuration](#), including setting resource limits and requests (recommended), JVM tuning, and metrics
- [Logging configuration](#)

Certain settings, though optional, are recommended for production deployments, such as security and resource allocation.

For details of all configuration options, see the [Strimzi Custom Resource API Reference](#).

### 11.9.1. Minimal configuration for MirrorMaker 2

A minimal [KafkaMirrorMaker2](#) resource requires the following in its spec:

- [.spec.target](#): The connection details for the target Kafka cluster. Requires a unique alias. MirrorMaker 2 runs on the Kafka Connect framework, and its configuration and status are stored on this cluster.
- [.spec.mirrors](#): The replication flow from a source cluster to a target cluster.
- [.spec.mirrors\[ \].source](#): The connection details for the source Kafka cluster. Requires a unique alias.
- [.spec.replicas](#): The number of Kafka Connect worker pods to deploy.

By default, MirrorMaker 2 replicates all topics and consumer groups. To reduce load on the cluster and avoid replicating unnecessary data, specify which topics and groups to include using filters:

- [topicsPattern](#): A regex to select topics for replication.
- [groupsPattern](#): A regex to select consumer groups for offset synchronization.

*Minimal configuration for KafkaMirrorMaker2*

```
# Basic configuration (required)
apiVersion: kafka.strimzi.io/v1
kind: KafkaMirrorMaker2
metadata:
  name: my-mirror-maker2
spec:
  # Kafka version (recommended)
  version: 4.1.1
  # Replicas (required)
  replicas: 3 ①
  # Target / Connect cluster configuration (required)
  target: ②
```

```

alias: "my-cluster-target"
bootstrapServers: my-cluster-target-kafka-bootstrap:9092
groupId: my-mirror-maker2-group
configStorageTopic: my-mirror-maker2-config
offsetStorageTopic: my-mirror-maker2-offset
statusStorageTopic: my-mirror-maker2-status
# Mirroring configurations (required)
mirrors: ③
- source:
    alias: "my-cluster-source" ④
    bootstrapServers: my-cluster-source-kafka-bootstrap:9092
# Source connector configuration (required) ⑤
sourceConnector: {}
# Topic and group patterns (required)
topicsPattern: "topic1.|topic2." ⑥
groupsPattern: "group1.*|group2-[0-9]{2}" ⑦

```

- ① Number of Kafka Connect worker replicas to run.
- ② Configuration of the Kafka cluster used as the target cluster. The cluster is also used by Kafka Connect to store internal topics.
- ③ Defines replication between the source and target clusters.
- ④ Source cluster configuration must define a unique alias and a bootstrap address. The bootstrap address takes the format <cluster\_name>-kafka-bootstrap:<port>.
- ⑤ Configuration for the source connector that replicates topics. At a minimum, you can add an empty `sourceConnector` property to use the default configuration.
- ⑥ Replicates only topics starting with `topic1` or `topic2`.
- ⑦ Replicates consumer groups starting with `group1` or `group2` followed by two digits.

**NOTE** The Kafka clusters do not need to be managed by Strimzi or run on Kubernetes.

### 11.9.2. Securing MirrorMaker 2 connections

This procedure describes in outline the configuration required to secure MirrorMaker 2 connections.

You configure these settings independently for the source Kafka cluster and the target Kafka cluster. You also need separate user configuration to provide the credentials required for MirrorMaker to connect to the source and target Kafka clusters.

For the Kafka clusters, you specify internal listeners for secure connections within a Kubernetes cluster and external listeners for connections outside the Kubernetes cluster.

You can configure authentication and authorization mechanisms. The security options implemented for the source and target Kafka clusters must be compatible with the security options implemented for MirrorMaker 2.

After you have created the cluster and user authentication credentials, you specify them in your

## MirrorMaker configuration for secure connections.

### NOTE

In this procedure, the certificates generated by the Cluster Operator are used, but you can replace them by [installing your own certificates](#). You can also configure your listener to [use a Kafka listener certificate managed by an external CA \(certificate authority\)](#).

### *Before you start*

Before starting this procedure, take a look at the [example configuration files](#) provided by Strimzi. They include examples for securing a deployment of MirrorMaker 2 using mTLS or SCRAM-SHA-512 authentication. The examples specify internal listeners for connecting within a Kubernetes cluster.

The examples also provide the configuration for full authorization, including the ACLs that allow user operations on the source and target Kafka clusters.

When configuring user access to source and target Kafka clusters, ACLs must grant access rights to internal MirrorMaker 2 connectors and read/write access to the cluster group and internal topics used by the underlying Kafka Connect framework in the target cluster. If you've renamed the cluster group or internal topics, such as when [configuring MirrorMaker 2 for multiple instances](#), use those names in the ACLs configuration.

Simple authorization uses ACL rules managed by the Kafka `StandardAuthorizer` plugin to ensure appropriate access levels. For more information on configuring a `KafkaUser` resource to use simple authorization, see the [AclRule schema reference](#).

### *Prerequisites*

- Strimzi is running
- Separate namespaces for source and target clusters

The procedure assumes that the source and target Kafka clusters are installed to separate namespaces. If you want to use the Topic Operator, you'll need to do this. The Topic Operator only watches a single cluster in a specified namespace.

By separating the clusters into namespaces, you will need to copy the cluster secrets so they can be accessed outside the namespace. You need to reference the secrets in the MirrorMaker configuration.

### *Procedure*

1. Configure two `Kafka` resources, one to secure the source Kafka cluster and one to secure the target Kafka cluster.

You can add listener configuration for authentication and enable authorization.

In this example, an internal listener is configured for a Kafka cluster with TLS encryption and mTLS authentication. Kafka `simple` authorization is enabled.

#### *Example source Kafka cluster configuration with TLS encryption and mTLS authentication*

```
apiVersion: kafka.strimzi.io/v1
```

```

kind: Kafka
metadata:
  name: my-source-cluster
spec:
  kafka:
    version: 4.1.1
    listeners:
      - name: tls
        port: 9093
        type: internal
        tls: true
        authentication:
          type: tls
    authorization:
      type: simple
  config:
    offsets.topic.replication.factor: 1
    transaction.state.log.replication.factor: 1
    transaction.state.log.min_isr: 1
    default.replication.factor: 1
    min.insync.replicas: 1
  entityOperator:
    topicOperator: {}
    userOperator: {}

```

*Example target Kafka cluster configuration with TLS encryption and mTLS authentication*

```

apiVersion: kafka.strimzi.io/v1
kind: Kafka
metadata:
  name: my-target-cluster
spec:
  kafka:
    version: 4.1.1
    listeners:
      - name: tls
        port: 9093
        type: internal
        tls: true
        authentication:
          type: tls
    authorization:
      type: simple
  config:
    offsets.topic.replication.factor: 1
    transaction.state.log.replication.factor: 1
    transaction.state.log.min_isr: 1
    default.replication.factor: 1
    min.insync.replicas: 1
  entityOperator:

```

```
topicOperator: {}
userOperator: {}
```

2. Create or update the **Kafka** resources in separate namespaces.

```
kubectl apply -f <kafka_configuration_file> -n <namespace>
```

The Cluster Operator creates the listeners and sets up the cluster and client certificate authority (CA) certificates to enable authentication within the Kafka cluster.

The certificates are created in the secret `<cluster_name>-cluster-ca-cert`.

3. Configure two **KafkaUser** resources, one for a user of the source Kafka cluster and one for a user of the target Kafka cluster.

- Configure the same authentication and authorization types as the corresponding source and target Kafka cluster. For example, if you used `tls` authentication and the `simple` authorization type in the **Kafka** configuration for the source Kafka cluster, use the same in the **KafkaUser** configuration.
- Configure the ACLs needed by MirrorMaker 2 to allow operations on the source and target Kafka clusters.

*Example source user configuration for mTLS authentication*

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaUser
metadata:
  name: my-source-user
  labels:
    strimzi.io/cluster: my-source-cluster
spec:
  authentication:
    type: tls
  authorization:
    type: simple
  acls:
    # MirrorSourceConnector
    - resource: # Not needed if offset-syncs.topic.location=target
      type: topic
      name: mm2-offset-syncs.my-target-cluster.internal
    operations:
      - Create
      - DescribeConfigs
      - Read
      - Write
    - resource: # Needed for every topic which is mirrored
      type: topic
      name: "*"
    operations:
```

```

    - DescribeConfigs
    - Read
# MirrorCheckpointConnector
- resource:
    type: cluster
operations:
- Describe
- resource: # Needed for every group for which offsets are synced
    type: group
    name: "*"
operations:
- Describe
- resource: # Not needed if offset-syncs.topic.location=target
    type: topic
    name: mm2-offset-syncs.my-target-cluster.internal
operations:
- Read

```

*Example target user configuration for mTLS authentication*

```

apiVersion: kafka.strimzi.io/v1
kind: KafkaUser
metadata:
  name: my-target-user
  labels:
    strimzi.io/cluster: my-target-cluster
spec:
  authentication:
    type: tls
  authorization:
    type: simple
  acls:
    # cluster group
    - resource:
        type: group
        name: mirrormaker2-cluster
    operations:
      - Read
  # access to config.storage.topic
  - resource:
      type: topic
      name: mirrormaker2-cluster-configs
    operations:
      - Create
      - Describe
      - DescribeConfigs
      - Read
      - Write
  # access to status.storage.topic
  - resource:

```

```

type: topic
name: mirrormaker2-cluster-status
operations:
- Create
- Describe
- DescribeConfigs
- Read
- Write
# access to offset.storage.topic
- resource:
  type: topic
  name: mirrormaker2-cluster-offsets
operations:
- Create
- Describe
- DescribeConfigs
- Read
- Write
# MirrorSourceConnector
- resource: # Needed for every topic which is mirrored
  type: topic
  name: "*"
operations:
- Create
- Alter
- AlterConfigs
- Write
# MirrorCheckpointConnector
- resource:
  type: cluster
operations:
- Describe
- resource:
  type: topic
  name: my-source-cluster.checkpoints.internal
operations:
- Create
- Describe
- Read
- Write
- resource: # Needed for every group for which the offset is synced
  type: group
  name: "*"
operations:
- Read
- Describe

```

**NOTE**

You can use a certificate issued outside the User Operator by setting `type` to `tls-external`. For more information, see the [KafkaUserSpec schema reference](#).

4. Create or update a `KafkaUser` resource in each of the namespaces you created for the source and target Kafka clusters.

```
kubectl apply -f <kafka_user_configuration_file> -n <namespace>
```

The User Operator creates the users representing the client (MirrorMaker), and the security credentials used for client authentication, based on the chosen authentication type.

The User Operator creates a new secret with the same name as the `KafkaUser` resource. The secret contains a private and public key for mTLS authentication. The public key is contained in a user certificate, which is signed by the clients CA.

5. Configure a `KafkaMirrorMaker2` resource with the authentication details to connect to the source and target Kafka clusters.

*Example MirrorMaker 2 configuration with TLS encryption and mTLS authentication*

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaMirrorMaker2
metadata:
  name: my-mirror-maker-2
spec:
  version: 4.1.1
  replicas: 1
  target:
    alias: "my-cluster-target"
    bootstrapServers: my-cluster-target-kafka-bootstrap:9092
    groupId: my-mirror-maker2-group
    configStorageTopic: my-mirror-maker2-config
    offsetStorageTopic: my-mirror-maker2-offset
    statusStorageTopic: my-mirror-maker2-status
    tls: ①
      trustedCertificates:
        - secretName: my-target-cluster-cluster-ca-cert
          pattern: "*.crt"
    authentication: ②
      type: tls
      certificateAndKey:
        secretName: my-target-user
        certificate: user.crt
        key: user.key
    config:
      # -1 means it will use the default replication factor configured in the
      broker
        config.storage.replication.factor: -1
        offset.storage.replication.factor: -1
        status.storage.replication.factor: -1
    mirrors:
      - source:
          alias: "my-source-cluster"
```

```

bootstrapServers: my-source-cluster-kafka-bootstrap:9093
tls: ③
  trustedCertificates:
    - secretName: my-source-cluster-cluster-ca-cert
      pattern: "*.crt"
authentication: ④
  type: tls
  certificateAndKey:
    secretName: my-source-user
    certificate: user.crt
    key: user.key
sourceConnector:
  config:
    replication.factor: 1
    offset-syncs.topic.replication.factor: -1
    sync.topic.acls.enabled: "false"
checkpointConnector:
  config:
    checkpoints.topic.replication.factor: 1
    sync.group.offsets.enabled: "true"
topicsPattern: "topic1|topic2|topic3"
groupsPattern: "group1|group2|group3"

```

① The TLS certificates for the target Kafka cluster.

② The user authentication for accessing the target Kafka cluster.

③ The TLS certificates for the source Kafka cluster. If they are in a separate namespace, copy the cluster secrets from the namespace of the Kafka cluster.

④ The user authentication for accessing the source Kafka cluster using the TLS mechanism. Supported authentication methods include `tls`, `scram-sha-256`, `scram-sha-512`, `plain`, and `custom`.

6. Apply the changes to the `KafkaMirrorMaker2` resource to the same namespace as the target Kafka cluster.

### 11.9.3. Configuring replicated topic naming

By default, MirrorMaker 2 renames replicated topics by prepending the source cluster's alias. For example, a topic named `topic1` from a cluster called `my-cluster-source` is replicated as `my-cluster-source.topic1`. This allows MirrorMaker 2 to detect mirroring cycles and is especially useful when deploying complex topologies or performing bidirectional replication.

You can change this behavior using the `replication.policy.class` property in the connector configuration. There are two built-in policies available:

- `org.apache.kafka.connect.mirror.IdentityReplicationPolicy` keeps original topic names. This approach is suitable for unidirectional replication, migration, or failover scenarios.
- `org.apache.kafka.connect.mirror.DefaultReplicationPolicy` (default) prefixes topic names. This is recommended for bidirectional replication. Use `replication.policy.separator` to specify

the character that separates the cluster name from the topic name in the replicated topic.

*Example configuration to keep topic names (IdentityReplicationPolicy)*

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaMirrorMaker2
metadata:
  name: my-mirror-maker2
spec:
  # ...
  mirrors:
    - source:
        # ...
        sourceConnector:
          config:
            replication.policy.class:
              "org.apache.kafka.connect.mirror.IdentityReplicationPolicy" ①
          checkpointConnector:
            config:
              replication.policy.class:
                "org.apache.kafka.connect.mirror.IdentityReplicationPolicy"
```

① Replicates topics without prefixing the source cluster name.

**IMPORTANT** Values for `replication.policy.class` and `replication.policy.separator` must be the same across all MirrorMaker 2 connectors (`sourceConnector` and `checkpointConnector`).

#### 11.9.4. Synchronizing consumer group offsets

Configure MirrorMaker 2 to synchronize consumer group offsets from the source cluster to the target cluster.

The `MirrorSourceConnector` and `MirrorCheckpointConnector` work together using internal topics to coordinate offset tracking between clusters.

##### `offset-syncs` topic

Format: `mm2-offset-syncs<separator><cluster_alias><separator>internal`

Populated by the `MirrorSourceConnector`, this topic stores offset mappings between source and target clusters. By default, it's created in the source cluster.

##### `checkpoints` topic

Format: `<cluster_alias><separator>checkpoints<separator>internal`

Populated by the `MirrorCheckpointConnector` in the target cluster, this topic captures the last committed offsets for each consumer group.

Configuring the `MirrorCheckpointConnector` to emit periodic offset checkpoints enables:

- Active/passive offset synchronization

- Failover recovery as consumers switch to the target cluster at the correct position

Offset synchronization occurs at regular intervals. The `MirrorCheckpointConnector` emits checkpoints for all consumer groups enabled by the group configuration. However, applying those checkpoints in the target cluster requires the target group to be inactive. If consumers switch to the target cluster between checkpoints, they might reprocess some messages because offset synchronization is asynchronous and the last synchronized offset lags behind the source. This duplication is expected.

**NOTE**

The connector continues to emit checkpoints to the checkpoints topic even if the target consumer group is active. However, the offsets cannot be committed while the consumer group has active members. In this case, the connector logs a warning and the offsets are not synchronized.

Each connector is configured separately.

*Example configuration to enable offset synchronization*

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaMirrorMaker2
metadata:
  name: my-mirror-maker2
  # ...
mirrors:
  - source:
      alias: "my-cluster-source"
      bootstrapServers: my-cluster-source-kafka-bootstrap:9092
      sourceConnector:
        config:
          offset-syncs.topic.replication.factor: -1 ①
          refresh.topics.interval.seconds: 60 ②
    checkpointConnector:
      config:
        sync.group.offsets.enabled: true ③
        sync.group.offsets.interval.seconds: 60 ④
        emit.checkpoints.interval.seconds: 60 ⑤
        refresh.groups.interval.seconds: 600 ⑥
        checkpoints.topic.replication.factor: -1 ⑦
```

① Replication factor for the `offset-syncs` internal topic that maps the offsets of the source and target clusters. A value of -1 uses the broker's default replication factor, which is typically set to provide resilience (for example, 3).

② Optional setting to change the frequency of checks for new topics.

③ Enables consumer group offset synchronization.

④ The frequency of the synchronization.

⑤ Adjusts the frequency of checks for offset tracking. If you change the frequency of offset synchronization, you might also need to adjust the frequency of these checks.

- ⑥ The frequency of checks for new consumer groups.
- ⑦ Replication factor for the internal checkpoints topic that stores the last committed offsets for each consumer group. A value of -1 uses the broker's default replication factor, which is typically set to provide resilience (for example, 3).

**TIP** Java applications can use the `RemoteClusterUtils` utility to fetch remote offsets for a consumer group from the checkpoints topic.

## Changing the location of the offset synchronization topic

By default, the `offset-syncs` topic used by the `MirrorSourceConnector` is created in the source cluster. You can change this behavior by setting the `offset-syncs.topic.location` connector configuration to `target`.

Placing the `offset-syncs` topic on the target cluster allows the `MirrorSourceConnector` to run when it has read-only access to the source cluster. The connector must still have read and write access to the `offset-syncs` topic on the cluster where the topic is located.

## Listing the offsets of connectors

To list the offset positions of the internal MirrorMaker 2 connectors, use the same configuration that's used to manage Kafka Connect connectors. For more information on setting up the configuration and listing offsets, see [Listing connector offsets](#).

In this example, the `sourceConnector` configuration is updated to return the connector offset position. The offset information is written to a specified `ConfigMap`.

*Example configuration for MirrorMaker 2 connector*

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaMirrorMaker2
metadata:
  name: my-mirror-maker2
spec:
  version: 4.1.1
  # ...
  target:
    alias: "my-cluster-target"
    bootstrapServers: my-cluster-target-kafka-bootstrap:9092
    groupId: my-mirror-maker2-group
    configStorageTopic: my-mirror-maker2-config
    offsetStorageTopic: my-mirror-maker2-offset
    statusStorageTopic: my-mirror-maker2-status
  mirrors:
  - source:
      alias: "my-cluster-source"
      bootstrapServers: my-cluster-source-kafka-bootstrap:9092
      sourceConnector:
        listOffsets:
          toConfigMap:
```

```
name: my-connector-offsets  
# ...
```

You must apply the following annotations to the `KafkaMirrorMaker2` resource be able to manage connector offsets:

- `strimzi.io/connector-offsets`
- `strimzi.io/mirrormaker-connector`

The `strimzi.io/mirrormaker-connector` annotation must be set to the name of the connector. These annotations remain until the operation succeeds or they are manually removed from the resource.

MirrorMaker 2 connectors are named using the aliases of the source and target clusters, followed by the connector type: `<source_alias>-><target_alias>. <connector_type>`.

In the following example, the annotations are applied for a connector named `my-cluster-source->my-cluster-target.MirrorSourceConnector`.

#### *Example application of annotations for connector*

```
kubectl annotate kafkamirrormaker2 my-mirror-maker-2 strimzi.io/connector-offsets=list  
strimzi.io/mirrormaker-connector="my-cluster-source->my-cluster-  
target.MirrorSourceConnector" -n kafka
```

The offsets are listed in the specified `ConfigMap`. Strimzi puts the offset information into a `.json` property named after the connector. This does not overwrite any other properties when updating an existing `ConfigMap`.

#### *Example source connector offset list*

```
apiVersion: v1  
kind: ConfigMap  
metadata:  
# ...  
ownerReferences: ①  
- apiVersion: kafka.strimzi.io/v1  
blockOwnerDeletion: false  
controller: false  
kind: KafkaMirrorMaker2  
name: my-mirror-maker2  
uid: 637e3be7-bd96-43ab-abde-c55b4c4550e0  
data:  
my-cluster-source--my-cluster-target.MirrorSourceConnector.json: |- ②  
{  
"offsets": [  
{  
"partition": {  
"cluster": "east-kafka",  
"partition": 0,  
"topic": "mirrormaker2-cluster-configs"
```

```

        },
        "offset": {
            "offset": 0
        }
    }
]
}

```

- ① The owner reference pointing to the [KafkaMirrorMaker2](#) resource. To provide a custom owner reference, create the [ConfigMap](#) in advance and set the owner reference.
- ② The `.json` property uses the connector name. Since `->` characters are not allowed in [ConfigMap](#) keys, `->` is changed to `--` in the connector name.

**NOTE** It is possible to use configuration to [alter](#) or [reset](#) connector offsets. This should be done with caution.

### 11.9.5. Synchronizing ACL rules

MirrorMaker 2 can synchronize topic ACL rules from source to target clusters. Enable this feature by configuring the [MirrorSourceConnector](#).

When using simple ACL-based authorization:

- Topic ACL rules can be synchronized between source and target topics
- MirrorMaker 2 copies topic-related ACL rules from the source cluster to the target cluster
- The `ALLOW ALL` topic ACL is not copied

Copying ACL rules does not guarantee equivalent access for all operations on the target cluster. Additional ACL rules may still be required to allow producers or consumers to operate as expected on the target cluster, for example, ACLs for other resource types such as consumer groups or transactional IDs.

**IMPORTANT** This feature is not compatible with the User Operator. If you use the User Operator, set `sync.topic.acls.enabled` to `false`. When using OAuth 2.0 authorization, the setting has no effect.

*Example configuration to enable ACL synchronization*

```

apiVersion: kafka.strimzi.io/v1
kind: KafkaMirrorMaker2
metadata:
  name: my-mirror-maker2
  # ...
mirrors:
  - source:
      # ...
    sourceConnector:
      config:

```

```
sync.topic.acls.enabled: "true" ①
```

- ① Copies topic-related ACL rules from the source cluster to the target cluster.

### 11.9.6. Tuning Kafka Connect worker settings

Adjust Kafka Connect worker runtime behavior for MirrorMaker 2 by configuring worker properties on the Kafka Connect cluster that runs the MirrorMaker 2 connectors.

Kafka Connect workers running MirrorMaker 2 connectors are configured to store their internal state in a single target Kafka cluster. The worker configuration specifies Kafka Connect runtime settings, such as task rebalance behavior, shutdown handling, and how frequently offsets are written to internal topics. To preserve records without modification, configure byte array converters for record keys and values.

*Example Kafka Connect configuration*

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaMirrorMaker2
metadata:
  name: my-mirror-maker2
spec:
  # ...
  # Target cluster configurations (required)
  target: ①
    alias: "my-cluster-target"
    bootstrapServers: my-cluster-target-kafka-bootstrap:9092
    groupId: my-mirror-maker2-group
    configStorageTopic: my-mirror-maker2-config
    offsetStorageTopic: my-mirror-maker2-offset
    statusStorageTopic: my-mirror-maker2-status
    config: ②
      # Key and value converters
      key.converter: org.apache.kafka.connect.converters.ByteArrayConverter
      value.converter: org.apache.kafka.connect.converters.ByteArrayConverter
      # Worker shutdown and rebalance behavior
      task.shutdown.graceful.timeout.ms: 30000
      scheduled.rebalance.max.delay.ms: 300000
      # Offset flush interval
      offset.flush.interval.ms: 10000
      # Kafka Connect internal topic replication
      config.storage.replication.factor: -1
      offset.storage.replication.factor: -1
      status.storage.replication.factor: -1
```

- ① The target cluster configuration. The target cluster is always used to store Kafka Connect's internal topics.
- ② These properties configure the Kafka Connect workers created by this [KafkaMirrorMaker2](#) resource. The settings apply to all MirrorMaker 2 connectors managed by the resource.

## 11.9.7. Configuring MirrorMaker 2 using a shared target Kafka cluster

MirrorMaker 2 runs on the Kafka Connect framework and uses internal Kafka topics to store configuration, offsets, and status information.

You can run multiple MirrorMaker 2 connectors in a single Kafka Connect cluster, for example to replicate data from multiple source Kafka clusters to the same target Kafka cluster.

When running multiple Kafka Connect clusters that replicate data to the same target Kafka cluster, you must configure unique Kafka Connect worker settings so that each cluster operates independently.

This requirement applies across all Kafka Connect-based deployments, including Kafka Connect and MirrorMaker 2 instances that target the same Kafka cluster.

Kafka Connect identifies a worker cluster by its group ID and by the names of the internal topics it uses. If multiple Kafka Connect clusters use the same values, they form a single Kafka Connect cluster and share internal state.

When deploying multiple Kafka Connect clusters against the same target Kafka cluster, configure unique values for the following Kafka Connect worker properties in each deployment:

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

These settings are configured in the `.spec.target` section.

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaMirrorMaker2
metadata:
  name: my-mirror-maker2
spec:
  # Target / Connect cluster configuration (required)
  target:
    alias: "my-cluster-target"
    bootstrapServers: my-cluster-target-kafka-bootstrap:9092
    groupId: my-mirror-maker2-group ①
    configStorageTopic: my-mirror-maker2-config ②
    offsetStorageTopic: my-mirror-maker2-offset ③
    statusStorageTopic: my-mirror-maker2-status ④
  # ...
```

① The Kafka Connect cluster group ID within Kafka.

② Kafka topic that stores connector offsets.

③ Kafka topic that stores connector and task status configurations.

- ④ Kafka topic that stores connector and task status updates.

If multiple Kafka Connect clusters are deployed with the same worker group ID and internal topic names, they form a single Kafka Connect cluster and share internal state. This can result in unexpected behavior and replication errors.

You do not need to change these settings when:

- Running multiple MirrorMaker 2 connectors in a single Kafka Connect cluster
- Replicating data from the same source Kafka cluster to different target Kafka clusters by using separate Kafka Connect clusters
- Deploying active-active replication, where each Kafka Connect cluster targets a different Kafka cluster

### 11.9.8. Disaster recovery in an active/passive configuration

MirrorMaker 2 can be configured for active/passive disaster recovery. To support this, the Kafka cluster should also be monitored for health and performance to detect issues that require failover promptly.

If failover occurs, which can be automated, operations switch from the active cluster to the passive cluster when the active cluster becomes unavailable. The original active cluster is typically considered permanently lost. The passive cluster is promoted to active status, taking over as the source and target for all application traffic. In this state, MirrorMaker 2 no longer replicates data from the original active cluster while it remains unavailable.

Fallback, or restoring operations to the original active cluster, requires careful planning.

It is technically possible to reverse roles in MirrorMaker 2 by swapping the source and target clusters and deploying this configuration as a new instance. However, this approach risks data duplication, as records mirrored to the passive cluster may be mirrored back to the original active cluster. Avoiding duplicates requires resetting consumer offsets, which adds complexity. For a simpler and more reliable fallback process, rebuild the original active cluster in a clean state and mirror data from the disaster recovery cluster.

Follow these best practices for disaster recovery in the event of failure of the active cluster in an active/passive configuration:

1. Promote the passive recovery cluster to an active role. Designate the passive cluster as the active cluster for all client connections. This minimizes downtime and ensures operations can continue.
2. Redirect applications to the new active recovery cluster. MirrorMaker 2 synchronizes committed offsets to passive clusters, allowing consumer applications to resume from the last transferred offset when switching to the recovery cluster. However, because of the time lag in offset synchronization, switching consumers may result in some message duplication. To minimize duplication, switch all members of a consumer group together as soon as possible. Keeping the group intact minimizes the chance of a consumer processing duplicate messages.
3. Remove the MirrorMaker 2 configuration for replication from the original active cluster to the

passive cluster. After failover, the original configuration is no longer needed and should be removed to avoid conflicts.

4. Re-create the failed cluster in a clean state, adhering to the original configuration.
5. Deploy a new MirrorMaker 2 instance to replicate data from the active recovery cluster to the rebuilt cluster. Treat the rebuilt cluster as the passive cluster during this replication process. To prevent automatic renaming of topics, configure MirrorMaker 2 to use the [IdentityReplicationPolicy](#) by setting the `replication.policy.class` property in the connector configuration. With this configuration applied, topics retain their original names in the target cluster.
6. Ensure the rebuilt cluster mirrors all data from the now-active recovery cluster.
7. (Optional) Promote the rebuilt cluster back to active status by redirecting applications to the rebuilt cluster, after ensuring it is fully synchronized with the active cluster.

**NOTE**

Before implementing any failover or fallback processes, test your recovery approach in a controlled environment to minimize downtime and maintain data integrity.

## 11.10. Configuring MirrorMaker 2 connectors

MirrorMaker 2 comprises a set of connectors to synchronize data between Kafka clusters. You configure these connectors in the [KafkaMirrorMaker2](#) custom resource.

MirrorMaker 2 consists of the following connectors:

### [MirrorSourceConnector](#)

(Required) Replicates topics (and, if configured, ACLs) from the source cluster to the target cluster.

### [MirrorCheckpointConnector](#)

(Optional) Synchronizes consumer group offsets for failover support. Use this connector only when offset tracking is required.

### [MirrorHeartbeatConnector](#)

(Optional) Monitors connectivity between clusters. This connector is useful for monitoring and alerting but is not required for replication.

[MirrorSourceConnector](#) and [MirrorCheckpointConnector](#) are configured in the `spec.mirrors` section of the [KafkaMirrorMaker2](#) custom resource using the `sourceConnector` and `checkpointConnector` properties.

These connectors run in the Kafka Connect cluster associated with the target Kafka cluster.

[MirrorHeartbeatConnector](#) is not configurable through the [KafkaMirrorMaker2](#) custom resource. To run a [MirrorHeartbeatConnector](#), use a separate [KafkaConnect](#) and [KafkaConnector](#) custom resource. Because it produces heartbeat records to the source Kafka cluster, the [MirrorHeartbeatConnector](#) typically runs in a Kafka Connect cluster that stores its state in the source cluster.

For a full list of supported MirrorMaker 2 connector properties, including common configuration and connector-specific options, see the [Apache Kafka documentation](#).

**WARNING**

Configure the following properties with the same values in both the `MirrorSourceConnector` and `MirrorCheckpointConnector`: `replication.policy.class`, `replication.policy.separator`, `offset-syncs.topic.location`, and `topic.filter.class`. If these properties do not match, replication or offset synchronization can fail.

### 11.10.1. Setting a maximum number of data replication tasks

Use `tasksMax` to control how many tasks are assigned to connectors. Increasing the number of tasks helps improve performance when replicating many partitions or synchronizing the offsets of a large number of consumer groups.

MirrorMaker 2 connectors create distributed tasks that move data between clusters. These tasks follow a set of operational behaviors and allocation rules.

Task behavior:

- Each task runs on a single worker pod.
- Tasks run in parallel.
- A worker pod can run multiple tasks, but each task runs in isolation.
- You don't need more pods than tasks. If there are fewer pods, tasks are distributed across available workers.

Task allocation rules:

- By default, connectors run with 1 task, unless `tasksMax` is set.
- For the `MirrorSourceConnector`, the maximum possible tasks = number of partitions.
- For the `MirrorCheckpointConnector`, the maximum possible tasks = number of consumer groups.
- Actual tasks started is the lower value of `tasksMax` and the maximum possible tasks.

If the infrastructure can support it, increasing the number of tasks improves throughput and reduces latency during high-volume replication.

*Increasing the number of tasks for the source connector*

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaMirrorMaker2
metadata:
  name: my-mirror-maker2
spec:
  # ...
  mirrors:
  - source:
      # ...
    sourceConnector:
```

```
tasksMax: 10
autoRestart:
  enabled: true
# ...
```

Enable automatic restarts of failed connectors and tasks using `autoRestart`. By default, the number of restarts is indefinite, but you can set a maximum on the number of automatic restarts using the `maxRestarts` property.

Increasing the number of tasks for the checkpoint connector is useful when you have a large number of consumer groups.

*Increasing the number of tasks for the checkpoint connector*

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaMirrorMaker2
metadata:
  name: my-mirror-maker2
spec:
  # ...
  mirrors:
  - source:
    # ...
    checkpointConnector:
      tasksMax: 10
      autoRestart:
        enabled: true
  # ...
```

By default, MirrorMaker 2 checks for new consumer groups every 10 minutes. You can adjust the `refresh.groups.interval.seconds` configuration to change the frequency. Take care when adjusting lower. More frequent checks can have a negative impact on performance.

### Checking connector task operations

If you are using Prometheus and Grafana to monitor your deployment, you can check MirrorMaker 2 performance. The example MirrorMaker 2 Grafana dashboard provided with Strimzi shows the following metrics related to tasks and latency.

- The number of tasks
- Replication latency
- Offset synchronization latency

### 11.10.2. Manually stopping or pausing MirrorMaker 2 connectors

If you are using `KafkaMirrorMaker2` resources to configure internal MirrorMaker connectors, use the `state` configuration to either stop or pause a connector. In contrast to the paused state, where the connector and tasks remain instantiated, stopping a connector retains only the configuration, with

no active processes. Stopping a connector from running may be more suitable for longer durations than just pausing. While a paused connector is quicker to resume, a stopped connector has the advantages of freeing up memory and resources.

**NOTE**

The `state` configuration replaces the (deprecated) `pause` configuration in the `KafkaMirrorMaker2ConnectorSpec` schema, which allows pauses on connectors. If you were previously using the `pause` configuration to pause connectors, we encourage you to transition to using the `state` configuration only to avoid conflicts.

*Prerequisites*

- The Cluster Operator is running.

*Procedure*

1. Find the name of the `KafkaMirrorMaker2` custom resource that controls the MirrorMaker 2 connector you want to pause or stop:

```
kubectl get KafkaMirrorMaker2
```

2. Edit the `KafkaMirrorMaker2` resource to stop or pause the connector.

*Example configuration for stopping a MirrorMaker 2 connector*

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaMirrorMaker2
metadata:
  name: my-mirror-maker2
spec:
  version: 4.1.1
  replicas: 3
  target:
    # ...
  mirrors:
    - source:
        # ...
      sourceConnector:
        tasksMax: 10
        autoRestart:
          enabled: true
          state: stopped
    # ...
```

Change the `state` configuration to `stopped` or `paused`. The default state for the connector when this property is not set is `running`.

3. Apply the changes to the `KafkaMirrorMaker2` configuration.

You can resume the connector by changing `state` to `running` or removing the configuration.

**NOTE**

Alternatively, you can [expose the Kafka Connect API](#) and use the `stop` and `pause` endpoints to stop a connector from running. For example, `PUT /connectors/<connector_name>/stop`. You can then use the `resume` endpoint to restart it.

### 11.10.3. Manually restarting MirrorMaker 2 connectors

Use the `strimzi.io/restart-connector` annotation to manually trigger a restart of a MirrorMaker 2 connector.

#### Prerequisites

- The Cluster Operator is running.

#### Procedure

1. Find the name of the `KafkaMirrorMaker2` custom resource that controls the Kafka MirrorMaker 2 connector you want to restart:

```
kubectl get KafkaMirrorMaker2
```

2. Find the name of the Kafka MirrorMaker 2 connector to be restarted from the `KafkaMirrorMaker2` custom resource:

```
kubectl describe KafkaMirrorMaker2 <mirrormaker_cluster_name>
```

3. Use the name of the connector to restart the connector by annotating the `KafkaMirrorMaker2` resource in Kubernetes:

```
kubectl annotate KafkaMirrorMaker2 <mirrormaker_cluster_name> "strimzi.io/restart-connector=<mirrormaker_connector_name>"
```

In this example, connector `source-cluster->target-cluster.MirrorCheckpointConnector` in the `my-mirror-maker-2` cluster is restarted:

```
kubectl annotate KafkaMirrorMaker2 my-mirror-maker-2 "strimzi.io/restart-connector=source-cluster->target-cluster.MirrorCheckpointConnector"
```

You can also refine the restart behavior with the `includeTasks` and `onlyFailed` parameters, which both default to `false`.

- `includeTasks` restarts both the connector instance and its task instances.
- `onlyFailed` restarts only instances with a `FAILED` status when set to `true`.

Separate the connector name and the parameters with a colon (`:`). For example:

```
kubectl annotate KafkaMirrorMaker2 my-mirror-maker-2 "strimzi.io/restart-connector=source-cluster->target-cluster.MirrorCheckpointConnector:includeTasks,onlyFailed"
```

4. Wait for the next reconciliation to occur (every two minutes by default).

The MirrorMaker 2 connector is restarted, as long as the annotation was detected by the reconciliation process. When MirrorMaker 2 accepts the request, the annotation is removed from the `KafkaMirrorMaker2` custom resource.

#### 11.10.4. Manually restarting MirrorMaker 2 connector tasks

Use the `strimzi.io/restart-connector-task` annotation to manually trigger a restart of a MirrorMaker 2 connector.

##### Prerequisites

- The Cluster Operator is running.

##### Procedure

1. Find the name of the `KafkaMirrorMaker2` custom resource that controls the MirrorMaker 2 connector task you want to restart:

```
kubectl get KafkaMirrorMaker2
```

2. Find the name of the connector and the ID of the task to be restarted from the `KafkaMirrorMaker2` custom resource:

```
kubectl describe KafkaMirrorMaker2 <mirrormaker_cluster_name>
```

Task IDs are non-negative integers, starting from 0.

3. Use the name and ID to restart the connector task by annotating the `KafkaMirrorMaker2` resource in Kubernetes:

```
kubectl annotate KafkaMirrorMaker2 <mirrormaker_cluster_name> "strimzi.io/restart-connector-task=<mirrormaker_connector_name>:<task_id>"
```

In this example, task `0` for connector `source-cluster->target-cluster.MirrorSourceConnector` in the `my-mirror-maker-2` cluster is restarted:

```
kubectl annotate KafkaMirrorMaker2 my-mirror-maker-2 "strimzi.io/restart-connector-task=source-cluster->target-cluster.MirrorSourceConnector:2"
```

4. Wait for the next reconciliation to occur (every two minutes by default).

The MirrorMaker 2 connector task is restarted, as long as the annotation was detected by the reconciliation process. When MirrorMaker 2 accepts the request, the annotation is removed from the [KafkaMirrorMaker2](#) custom resource.

### 11.10.5. Configuring MirrorMaker 2 connector producers and consumers

MirrorMaker 2 connectors use internal producers and consumers. If required, you can configure these producers and consumers to override the default settings.

For example, you can increase the `batch.size` for the source producer that sends topics to the target Kafka cluster to better accommodate large volumes of messages.

**IMPORTANT**

Producer and consumer configuration options depend on the MirrorMaker 2 implementation, and may be subject to change.

The following tables describe the producers and consumers for each of the connectors and where you can add configuration.

Configuration can be applied as follows:

- Overriding the configuration used by the internal producer or consumer by using the `producer.override.` or `consumer.override.` properties.
- Configuring the internal producer or consumer directly in the connector by using the `producer.` or `consumer.` properties.

*Table 12. Source connector producers and consumers*

Type	Description	Configuration
Producer	Sends replicated topic messages to the target Kafka cluster. Tune this producer when handling large volumes of data.	<code>producer.override.*</code>
Producer	Writes to the <code>offset-syncs</code> topic, which maps source and target offsets.	<code>producer.*</code>
Consumer	Consumes topic messages from the source Kafka cluster.	<code>consumer.*</code>

*Table 13. Checkpoint connector producers and consumers*

Type	Description	Configuration
Producer	Emits consumer group offset checkpoints.	<code>producer.override.*</code>
Consumer	Consumes data from the <code>offset-syncs</code> topic.	<code>consumer.*</code>

**NOTE**

You can set `offset-syncs.topic.location` to `target` to use the target Kafka cluster as the location of the `offset-syncs` topic.

The following example shows how you configure the producers and consumers.

*Example configuration for connector producers and consumers*

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaMirrorMaker2
metadata:
  name: my-mirror-maker2
spec:
  version: 4.1.1
  # ...
  mirrors:
  - source:
      # ...
    sourceConnector:
      tasksMax: 5
      config:
        producer.override.batch.size: 327680
        producer.override.linger.ms: 100
        producer.request.timeout.ms: 30000
        consumer.fetch.max.bytes: 52428800
        # ...
    checkpointConnector:
      config:
        producer.override.request.timeout.ms: 30000
        consumer.max.poll.interval.ms: 300000
        # ...
```

## 11.11. Configuring the HTTP Bridge

Update the `spec` properties of the `KafkaBridge` custom resource to configure your HTTP Bridge deployment.

In order to prevent issues arising when client consumer requests are processed by different HTTP Bridge instances, address-based routing must be employed to ensure that requests are routed to the right HTTP Bridge instance. Additionally, each independent HTTP Bridge instance must have a replica. A HTTP Bridge instance has its own state which is not shared with another instances.

For a deeper understanding of the HTTP Bridge and its cluster configuration options, refer to the [Using the HTTP Bridge](#) guide and the [Strimzi Custom Resource API Reference](#).

*Example KafkaBridge custom resource configuration*

```
# Basic configuration (required)
apiVersion: kafka.strimzi.io/v1
kind: KafkaBridge
metadata:
  name: my-bridge
spec:
  # Replicas (required)
```

```

replicas: 3 ①
# Kafka bootstrap servers (required)
bootstrapServers: <cluster_name>-cluster-kafka-bootstrap:9092 ②
# HTTP configuration (required)
http: ③
  port: 8080
  # CORS configuration (optional)
  cors: ④
    allowedOrigins: "https://strimzi.io"
    allowedMethods: "GET,POST,PUT,DELETE,OPTIONS,PATCH"
# Resources requests and limits (recommended)
resources: ⑤
  requests:
    cpu: "1"
    memory: 2Gi
  limits:
    cpu: "2"
    memory: 2Gi
# TLS configuration (optional)
tls: ⑥
  trustedCertificates:
    - secretName: my-cluster-cluster-cert
      pattern: "*.crt"
    - secretName: my-cluster-cluster-cert
      certificate: ca2.crt
# Authentication (optional)
authentication: ⑦
  type: tls
  certificateAndKey:
    secretName: my-secret
    certificate: public.crt
    key: private.key
# Consumer configuration (optional)
consumer: ⑧
  config:
    auto.offset.reset: earliest
# Producer configuration (optional)
producer: ⑨
  config:
    delivery.timeout.ms: 300000
# Logging configuration (optional)
logging: ⑩
  type: inline
  loggers:
    rootLogger.level: INFO
    # Enabling DEBUG just for send operation
    logger.send.name: http.openapi.operation.send
    logger.send.level: DEBUG
# JVM options (optional)
jvmOptions: ⑪
  "-Xmx": "1g"

```

```

"-Xms": "1g"
# Readiness probe (optional)
readinessProbe: ⑫
  initialDelaySeconds: 15
  timeoutSeconds: 5
# Liveness probe (optional)
livenessProbe:
  initialDelaySeconds: 15
  timeoutSeconds: 5
# Custom image (optional)
image: my-org/my-image:latest ⑬
# Pod template (optional)
template: ⑭
  pod:
    affinity:
      podAntiAffinity:
        requiredDuringSchedulingIgnoredDuringExecution:
          - labelSelector:
              matchExpressions:
                - key: application
                  operator: In
                  values:
                    - postgresql
                    - mongodb
            topologyKey: "kubernetes.io/hostname"
bridgeContainer: ⑮
  env:
    - name: OTEL_SERVICE_NAME
      value: my-otel-service
    - name: OTEL_EXPORTER_OTLP_ENDPOINT
      value: "http://otlp-host:4317"
# Tracing configuration (optional)
tracing:
  type: opentelemetry ⑯
# Metrics configuration (optional)
metricsConfig:
  type: jmxPrometheusExporter ⑰
  valueFrom:
    configMapKeyRef:
      name: kafka-metrics
      key: kafka-metrics-config.yml

```

① The number of replica nodes.

② Bootstrap address for connection to the target Kafka cluster. The address takes the format <cluster\_name>-kafka-bootstrap:<port\_number>. The Kafka cluster doesn't need to be managed by Strimzi or deployed to a Kubernetes cluster.

③ HTTP access to Kafka brokers.

④ CORS access specifying selected resources and access methods. Additional HTTP headers in requests describe the origins that are permitted access to the Kafka cluster.

- ⑤ Requests for reservation of supported resources, currently `cpu` and `memory`, and limits to specify the maximum resources that can be consumed.
- ⑥ TLS configuration for encrypted connections to the Kafka cluster, with trusted certificates stored in X.509 format within the specified secrets.
- ⑦ Authentication for the HTTP Bridge cluster, specified as `tls`, `scram-sha-256`, `scram-sha-512`, `plain`, or `custom`. By default, the HTTP Bridge connects to Kafka brokers without authentication. For details on configuring authentication, see the [KafkaBridgeSpec schema properties](#)
- ⑧ Consumer configuration options.
- ⑨ Producer configuration options.
- ⑩ HTTP Bridge loggers and log levels added directly (`inline`) or indirectly (`external`) through a `ConfigMap`. Custom Log4j configuration must be placed under the `log4j2.properties` key in the `ConfigMap`. You can set log levels to `INFO`, `ERROR`, `WARN`, `TRACE`, `DEBUG`, `FATAL` or `OFF`.
- ⑪ JVM configuration options to optimize performance for the Virtual Machine (VM) running the HTTP Bridge.
- ⑫ Healthchecks to know when to restart a container (liveness) and when a container can accept traffic (readiness).
- ⑬ Optional: Container image configuration, which is recommended only in special situations.
- ⑭ Template customization. Here a pod is scheduled with anti-affinity, so the pod is not scheduled on nodes with the same hostname.
- ⑮ Environment variables are set for distributed tracing.
- ⑯ Distributed tracing is enabled by using OpenTelemetry.
- ⑰ Prometheus metrics enabled. In this example, metrics are configured for the Prometheus JMX Exporter.

## 11.12. Applying optional common configuration

You can further configure Strimzi components by applying any of the following optional common configuration settings. Common configuration is configured independently for each component, such as the following:

- Resource limits and requests (Recommended)
- Metrics configuration
- Liveness and readiness probes
- JVM options for maximum and minimum memory allocation
- Adding additional volumes and volume mounts
- Template configuration for pods and containers
- Logging frequency

Advanced or specialized options include:

- Custom container images

- Rack awareness
- Distributed tracing

Configure common options for Strimzi custom resources in the `.spec` section of the custom resource. For more information on these configuration options, refer to [Common configuration properties](#).

### 11.12.1. Resource limits and requests (recommended)

To ensure stability and optimal performance for your Kafka clusters, we recommend defining CPU and memory resource limits and requests for all Strimzi containers. By default, the Strimzi Cluster Operator does not set these values, but fine-tuning them based on your workload requirements helps performance and improves reliability.

*Example resource configuration*

```
# ...
spec:
  resources:
    requests:
      cpu: "1"
      memory: 2Gi
    limits:
      cpu: "2"
      memory: 2Gi
# ...
```

### 11.12.2. Metrics configuration

Enable metrics collection for monitoring.

*Example metrics configuration*

```
# ...
spec:
  metricsConfig:
    type: jmxPrometheusExporter
    valueFrom:
      configMapKeyRef:
        name: my-metrics-config
        key: kafka-metrics-config.yml
# ...
```

Configuration varies depending on the component and exporter used: Prometheus JMX Exporter or Strimzi Metrics Reporter. For more information, see [Introducing metrics](#).

### 11.12.3. Liveness and readiness probes

Configure health checks for the container.

*Example liveness and readiness probes*

```
# ...
spec:
  livenessProbe:
    initialDelaySeconds: 15
    timeoutSeconds: 5
  readinessProbe:
    initialDelaySeconds: 10
    timeoutSeconds: 5
# ...
```

### 11.12.4. JVM options

Configure the Java Virtual Machine (JVM) for the component. To enable garbage collector (GC) logging, set `gcLoggingEnabled` to `true`.

*Example JVM options*

```
# ...
spec:
  jvmOptions:
    -Xms: "512m"
    -Xmx: "1g"
    gcLoggingEnabled: true
# ...
```

### 11.12.5. Additional volumes and mounts

Add extra volumes to the container and mount them in specific locations.

*Example additional volumes*

```
# ...
spec:
  kafka:
    template:
      pod:
        volumes:
          - name: example-secret
            secret:
              secretName: secret-name
          - name: example-configmap
            configMap:
              name: config-map-name
  kafkaContainer:
```

```
volumeMounts:  
  - name: example-secret  
    mountPath: /mnt/secret-volume  
  - name: example-configmap  
    mountPath: /mnt/cm-volume  
# ...
```

**NOTE**

You can use [template](#) configuration to add other customizations to pods and containers, such as affinity and security context. For more information, see [Configuring pod scheduling](#) and [Applying security context to Strimzi pods and containers](#).

### 11.12.6. Custom container image

Override the default container image. **Use only in special situations.**

*Example custom image*

```
# ...  
spec:  
  image: my-org/custom-kafka-image:latest  
# ...
```

### 11.12.7. Rack awareness

Enable rack-aware broker assignment to improve fault tolerance. **This is a specialized option intended for a deployment within the same location, not across regions.**

*Example rack awareness configuration*

```
# ...  
rack:  
  topologyKey: topology.kubernetes.io/zone  
# ...
```

**IMPORTANT**

When rack awareness is enabled, Strimzi sets the rack (zone) information in the [Kafka](#) configuration using the [topologyKey](#) label. It also adds a node affinity rule to schedule Pods only on nodes that include this label. Strimzi does not add anti-affinity rules or topology spread constraints to distribute Pods across racks or zones. To configure distribution, define your own rules in the [Kafka](#) or [KafkaNodePool](#) custom resources. For more information see [Distributing brokers across availability zones](#)

### 11.12.8. Distributed tracing configuration

Enable distributed tracing using OpenTelemetry to monitor Kafka component operations.

## Example tracing configuration

```
# ...
spec:
  tracing:
    type: opentelemetry
# ...
```

For more information see [Introducing distributed tracing](#).

### 11.12.9. Configuring pod scheduling

To optimize the resilience and performance of your Kafka cluster, you can control how its pods are scheduled across Kubernetes nodes. Pod scheduling strategies can help you to achieve the following:

- Increase fault tolerance by spreading Kafka pods across different nodes.
- Avoid resource contention by separating pods from critical workloads.
- Maintain resource availability by assigning Kafka pods to nodes with sufficient capacity.

#### Scheduling strategies

Kafka components can be scheduled onto Kubernetes nodes using affinity rules, tolerations, and topology constraints. These strategies help isolate workloads, optimize resource usage, and improve overall cluster performance.

The following scheduling techniques support different deployment goals:

##### Use pod anti-affinity to avoid critical applications sharing nodes

Use pod anti-affinity to prevent critical applications from being scheduled on the same disk. In Kafka deployments, configure pod anti-affinity to ensure that Kafka brokers do not share nodes with other workloads, such as databases.

##### Use node affinity to schedule workloads onto specific nodes

Kubernetes clusters often include nodes optimized for different workloads, such as CPU, memory, storage, or network. Node affinity enables scheduling Kafka components onto nodes that match specific labels, such as `node.kubernetes.io/instance-type` or custom labels, to optimize performance and cost.

##### Use node affinity, taints, and tolerations for dedicated nodes

To reserve nodes for Kafka, you can taint them to exclude them from general workloads. Kafka pods can still be scheduled onto these nodes by configuring:

- Tolerations, which allow the pods to bypass the taint.
- Node affinity, for the pods to run on those specific nodes.

These settings direct Kafka pods to dedicated nodes while preventing other workloads from being scheduled there. This approach isolates Kafka from other workloads, reducing resource

contention and improving stability.

## Use topology spread constraints to balance pods across zones or nodes

Topology spread constraints help distribute pods evenly across specified topology domains, such as zones, regions, or nodes. For Kafka, this strategy reduces the risk of scheduling too many brokers on the same zone or node, improving availability and resilience.

You can apply these scheduling strategies in the `template.pod` property within the `spec` of a Strimzi custom resource.

For Kafka brokers specifically, Strimzi provides a two-level configuration:

- `Kafka.spec.kafka.template.pod`

Use this to set a cluster-wide default scheduling policy for all Kafka broker and controller pods.

- `KafkaNodePool.spec.template.pod`

For more granular control, if needed, you can override the default policy for specific node pools using the `KafkaNodePool` resource.

**IMPORTANT**

If a property is defined in both `KafkaNodePool.spec.template` and `Kafka.spec.kafka.template`, the property defined in the node pool is used. For example, node pools that use `KafkaNodePool.spec.template.pod` settings do not inherit pod settings from the cluster-wide template.

Other Strimzi components have their own template properties for scheduling:

- `Kafka.spec.kafka.template.pod`
- `KafkaNodePool.spec.template.pod`
- `Kafka.spec.entityOperator.template.pod`
- `Kafka.spec.cruiseControl.template.pod`
- `KafkaConnect.spec.template.pod`
- `KafkaBridge.spec.template.pod`
- `KafkaMirrorMaker2.spec.template.pod`

The following procedures provide scheduling configuration examples for the `Kafka` and `KafkaNodePool` resources. The same configurations shown in these examples can be applied to the `template.pod` property of any other component.

Scheduling properties follow the Kubernetes specification.

### *Additional resources*

- [Kubernetes node and pod affinity documentation](#)
- [Kubernetes taints and tolerations](#)
- [Kubernetes Topology Spread Constraints](#)

## Configuring pod anti-affinity for Kafka nodes

To improve fault tolerance, you can prevent multiple Kafka broker pods from running on the same worker node. Configure `podAntiAffinity` so that the pods are scheduled on different worker nodes.

This procedure provides configuration examples for the cluster-wide `Kafka` resource and the pool-specific `KafkaNodePool` resource.

**NOTE** If you configure a property in `KafkaNodePool.spec.template`, the configuration overrides the same property defined in `Kafka.spec.kafka.template` for that node pool. For more information, see [Scheduling strategies](#).

### Prerequisites

- The Cluster Operator must be deployed.

### Procedure

1. Configure `podAntiAffinity` in either the `Kafka` or `KafkaNodePool` resource.

- To set a cluster-wide rule, edit the `affinity` property in `spec.kafka.template.pod` of your `Kafka` resource. Use the `strimzi.io/name` label to select all broker pods.
- To set a pool-specific rule, edit the `affinity` property in `spec.template.pod` of your `KafkaNodePool` resource. Use the `strimzi.io/pool-name` label to select only the pods in that pool.

In both cases, set the `topologyKey` to `"kubernetes.io/hostname"` to prevent pods from being placed on the same host.

This example applies a rule to a `Kafka` resource named `my-cluster` that prevents any of its broker pods from running on the same node.

### Example cluster-wide anti-affinity configuration

```
apiVersion: kafka.strimzi.io/v1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    template:
      pod:
        affinity:
          podAntiAffinity:
            requiredDuringSchedulingIgnoredDuringExecution:
              - labelSelector:
                  matchExpressions:
                    - key: strimzi.io/name
                      operator: In
                      values:
                        - my-cluster-kafka
```

```
        topologyKey: "kubernetes.io/hostname"
# ...
```

This example applies a rule to a [KafkaNodePool](#) resource named `broker` that prevents pods from that specific pool from running on the same node.

*Example node pool-specific anti-affinity configuration*

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaNodePool
metadata:
  name: broker
  labels:
    strimzi.io/cluster: my-cluster
spec:
  replicas: 3
  roles:
    - broker
  template:
    pod:
      affinity:
        podAntiAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            - labelSelector:
                matchExpressions:
                  - key: strimzi.io/pool-name
                    operator: In
                    values:
                      - broker
            topologyKey: "kubernetes.io/hostname"
# ...
```

2. Apply the changes to your custom resource configuration.

## Distributing brokers across availability zones

To improve fault tolerance, distribute Kafka nodes across multiple different availability zones (racks). This is typically combined with the [Rack awareness](#) feature.

This procedure provides configuration examples for the [Kafka](#) resource with rack awareness enabled, and the pool-specific [KafkaNodePool](#) resources with the topology spread constraints configuration to balance pods across zones.

### Prerequisites

- The Cluster Operator must be deployed.

### Procedure

1. Configure `rack` in the [Kafka](#) resource and `topologySpreadConstraints` in the [KafkaNodePool](#) resources.

Make sure the `strimzi.io/cluster` label in the `topologySpreadConstraint[].labelSelector` fields is set to the name of your Kafka cluster.

The following configuration shows a Kafka cluster with rack awareness enabled and topology spread constraints defined in the `KafkaNodePool` resources:

```
apiVersion: kafka.strimzi.io/v1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    rack:
      topologyKey: topology.kubernetes.io/zone
    # ...
# ...
---
apiVersion: kafka.strimzi.io/v1
kind: KafkaNodePool
metadata:
  name: controllers
  labels:
    strimzi.io/cluster: my-cluster
spec:
  roles:
    - controller
  template:
    pod:
      topologySpreadConstraints:
        - maxSkew: 1
          topologyKey: topology.kubernetes.io/zone
          whenUnsatisfiable: DoNotSchedule
          labelSelector:
            matchLabels:
              strimzi.io/cluster: my-cluster
              strimzi.io/controller-role: "true"
        # ...
# ...
apiVersion: kafka.strimzi.io/v1
kind: KafkaNodePool
metadata:
  name: brokers
  labels:
    strimzi.io/cluster: my-cluster
spec:
  roles:
    - broker
  template:
    pod:
```

```

topologySpreadConstraints:
  - maxSkew: 1
    topologyKey: topology.kubernetes.io/zone
    whenUnsatisfiable: DoNotSchedule
    labelSelector:
      matchLabels:
        strimzi.io/cluster: my-cluster
        strimzi.io/broker-role: "true"
    # ...

```

2. Apply the changes to your custom resource configuration.

## Configuring pod anti-affinity against other workloads

To improve stability and performance, you can prevent Kafka pods from running on the same worker nodes as other resource-intensive applications, such as databases. Configure `podAntiAffinity` so that these workloads are scheduled on separate nodes.

This procedure provides configuration examples for the cluster-wide `Kafka` resource and the pool-specific `KafkaNodePool` resource.

**NOTE** If a property is defined in both `KafkaNodePool.spec.template` and `Kafka.spec.kafka.template`, the property defined in the node pool is used. Properties are not merged. For more information, see [Scheduling strategies](#).

### Prerequisites

- [The Cluster Operator must be deployed](#).
- The other workloads in your cluster use consistent labels.

### Procedure

1. Configure `podAntiAffinity` in either the `Kafka` or `KafkaNodePool` resource.

- To set a cluster-wide rule, edit the `affinity` property in `spec.kafka.template.pod` of your `Kafka` resource.
- To set a pool-specific rule, edit the `affinity` property in `spec.template.pod` of your `KafkaNodePool` resource.

In both cases, use a `labelSelector` to identify the application pods you want to keep separate from your Kafka pods and set the `topologyKey` to `"kubernetes.io/hostname"` to prevent pods from being placed on the same host.

This example applies a rule to a `Kafka` resource named `my-cluster` that prevents any of its broker pods from running on the same node as pods labeled `postgresql` and `mongodb`.

### Example cluster-wide anti-affinity configuration

```

apiVersion: kafka.strimzi.io/v1
kind: Kafka
metadata:

```

```

name: my-cluster
spec:
  kafka:
    # ...
    template:
      pod:
        affinity:
          podAntiAffinity:
            requiredDuringSchedulingIgnoredDuringExecution:
              - labelSelector:
                  matchExpressions:
                    - key: application
                      operator: In
                      values:
                        - postgresql
                        - mongodb
            topologyKey: "kubernetes.io/hostname"
# ...

```

This example applies a rule to a `KafkaNodePool` resource named `broker` that prevents pods from that pool from running on the same node as pods labeled `postgresql` and `mongodb`.

#### *Example node pool-specific anti-affinity configuration*

```

apiVersion: kafka.strimzi.io/v1
kind: KafkaNodePool
metadata:
  name: broker
  labels:
    strimzi.io/cluster: my-cluster
spec:
  replicas: 3
  roles:
    - broker
  template:
    pod:
      affinity:
        podAntiAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            - labelSelector:
                matchExpressions:
                  - key: application
                    operator: In
                    values:
                      - postgresql
                      - mongodb
            topologyKey: "kubernetes.io/hostname"
# ...

```

2. Apply the changes to your custom resource configuration.

## Configuring pod scheduling for specific nodes

Your Kubernetes cluster might have different types of worker nodes, some with specialized hardware (fast SSDs, powerful CPUs) or a particular location (a specific availability zone). To optimize performance and resource usage, configure `nodeAffinity` so that Kafka pods are scheduled only on the nodes that match your requirements.

This procedure provides configuration examples for the cluster-wide `Kafka` resource and the pool-specific `KafkaNodePool` resource.

**NOTE** If a property is defined in both `KafkaNodePool.spec.template` and `Kafka.spec.kafka.template`, the property defined in the node pool is used. Properties are not merged. For more information, see [Scheduling strategies](#).

### Prerequisites

- [The Cluster Operator must be deployed](#).
- Worker nodes are labeled to identify their specific characteristics (such as `disk:sdd`).

### Procedure

1. Label the worker nodes to identify them when scheduling the kafka pods.

```
kubectl label node <name_of_node> disk=sdd
```

2. Configure `nodeAffinity` in either the `Kafka` or `KafkaNodePool` resource to match the label.

- To set a cluster-wide rule, edit the `affinity` property in `spec.kafka.template.pod` of your `Kafka` resource.
- To set a pool-specific rule, edit the `affinity` property in `spec.template.pod` of your `KafkaNodePool` resource.

In both cases, use `nodeSelectorTerms` with `matchExpressions` to specify the key-value label of the nodes you want to schedule pods on.

This example applies a rule to a `Kafka` resource that assigns all its broker pods to run only on nodes with the label `disk: ssd`.

### Example cluster-wide affinity configuration for specific nodes

```
apiVersion: kafka.strimzi.io/v1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    template:
      pod:
        affinity:
```

```

nodeAffinity:
  requiredDuringSchedulingIgnoredDuringExecution:
    nodeSelectorTerms:
      - matchExpressions:
          - key: disk
            operator: In
            values:
              - sdd
# ...

```

#### *Example availability zone scheduling for node pools*

For node pools, a common production scenario is to configure node pools so that pods are scheduled only on nodes within a specified availability zone. For more information, see [Managing storage affinity using node pools](#)

3. Apply the changes to your custom resource configuration.

### Configuring pod scheduling for dedicated nodes

You can dedicate a set of worker nodes exclusively to your Kafka brokers so that no other applications can compete with Kafka for resources on those nodes.

To configure dedicated worker nodes for Kafka pods in a specific pool, combine the following:

#### Taints

Apply taints to worker nodes to prevent other pods from being scheduled on them.

#### Tolerations

Apply tolerations to Kafka pods to allow them to be scheduled on tainted nodes.

#### Affinity

Apply affinity to Kafka pods to schedule them on specifically labeled nodes.

This procedure provides configuration examples for the cluster-wide `Kafka` resource and the pool-specific `KafkaNodePool` resource.

**NOTE** If a property is defined in both `KafkaNodePool.spec.template` and `Kafka.spec.kafka.template`, the property defined in the node pool is used. Properties are not merged. For more information, see [Scheduling strategies](#).

#### Prerequisites

- [The Cluster Operator must be deployed.](#)
- Dedicated worker nodes without scheduled workloads

#### Procedure

1. Taint and label the dedicated worker nodes to prevent other pods from being scheduled on them and to identify them when scheduling the Kafka pods:

```
kubectl taint node <name_of_node> dedicated=kafka:NoSchedule  
kubectl label node <name_of_node> dedicated=kafka
```

2. Configure `tolerations` and `nodeAffinity` in either your `Kafka` or `KafkaNodePool` custom resource to match the taint and label.

- To set a cluster-wide rule, edit the `affinity` property in `spec.kafka.template.pod` of your `Kafka` resource.
- To set a pool-specific rule, edit the `affinity` property in `spec.template.pod` of your `KafkaNodePool` resource.

In both cases, use `nodeSelectorTerms` with `matchExpressions` to specify the key-value label of the nodes you want to schedule pods on.

This example applies a rule to a `Kafka` resource that assigns all its broker pods to run only on nodes that have been tainted and labeled with `dedicated=kafka`.

*Example cluster-wide affinity configuration for dedicated nodes*

```
apiVersion: kafka.strimzi.io/v1  
kind: Kafka  
metadata:  
  name: my-cluster  
spec:  
  kafka:  
    # ...  
    template:  
      pod:  
        tolerations:  
          - key: "dedicated"  
            operator: "Equal"  
            value: "kafka"  
            effect: "NoSchedule"  
        affinity:  
          nodeAffinity:  
            requiredDuringSchedulingIgnoredDuringExecution:  
              nodeSelectorTerms:  
                - matchExpressions:  
                  - key: dedicated  
                    operator: In  
                    values:  
                      - kafka  
    # ...
```

This example applies a rule to a `KafkaNodePool` resource named `broker` that assigns its pods to run on dedicated nodes marked with `dedicated=broker-kafka`.

*Example node pool-specific affinity configuration for dedicated nodes*

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaNodePool
metadata:
  name: broker
  labels:
    strimzi.io/cluster: my-cluster
spec:
  replicas: 3
  roles:
    - broker
  template:
    pod:
      tolerations:
        - key: "dedicated"
          operator: "Equal"
          value: "kafka"
          effect: "NoSchedule"
      affinity:
        nodeAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
              - matchExpressions:
                  - key: dedicated
                    operator: In
                    values:
                      - broker-kafka
# ...
```

3. Apply the changes to your custom resource configuration.

### 11.12.10. Disabling PodDisruptionBudget generation

Strimzi automatically generates a `PodDisruptionBudget` resource for each of the following components:

- Kafka cluster
- Kafka Connect
- MirrorMaker 2
- HTTP Bridge

Each budget applies across all pods deployed for that component.

The Kafka cluster's `PodDisruptionBudget` covers all associated node pool pods.

To disable automatic `PodDisruptionBudget` generation, set the `STRIMZI_POD_DISRUPTION_BUDGET_GENERATION` environment variable to `false` in the Cluster Operator configuration. You can then define custom `PodDisruptionBudget` resources if needed. For more

information, see [Configuring the Cluster Operator](#).

### 11.12.11. Using ConfigMap resources to add configuration

Add specific configuration to your Strimzi deployment using [ConfigMap](#) resources. Config maps use key-value pairs to store non-confidential data. Configuration data added to config maps is maintained in one place and can be reused amongst components.

Config maps can only store the following types of configuration data:

- Logging configuration
- Metrics configuration
- External configuration for Kafka Connect connectors

You can't use config maps for other areas of configuration.

When you configure a component, you can add a reference to a [ConfigMap](#) using the `configMapKeyRef` property.

For example, you can use `configMapKeyRef` to reference a [ConfigMap](#) that provides configuration for logging. You define logging levels using the `log4j2.properties` key in the [ConfigMap](#) and then reference it in the `logging` configuration of the resource.

*Example reference to a ConfigMap*

```
# ...
logging:
  type: external
  valueFrom:
    configMapKeyRef:
      name: my-config-map
      key: log4j2.properties
# ...
```

To use a [ConfigMap](#) for metrics configuration, you add a reference to the `metricsConfig` configuration of the component in the same way.

`template` properties allow data from a [ConfigMap](#) or [Secret](#) to be mounted in a pod as environment variables or volumes. You can use external configuration data for the connectors used by Kafka Connect. The data might be related to an external data source, providing the values needed for the connector to communicate with that data source.

For example, you can use the `configMapKeyRef` property to pass configuration data from a [ConfigMap](#) as an environment variable.

*Example ConfigMap providing environment variable values*

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaConnect
metadata:
```

```

name: my-connect
spec:
  # ...
  template:
    connectContainer:
      env:
        - name: MY_ENVIRONMENT_VARIABLE
          valueFrom:
            configMapKeyRef:
              name: my-config-map
              key: my-key

```

If you are using config maps that are managed externally, use configuration providers to load the data in the config maps.

### Naming custom config maps

Strimzi [creates its own config maps and other resources](#) when it is deployed to Kubernetes. The config maps contain data necessary for running components. The config maps created by Strimzi must not be edited.

Make sure that any custom config maps you create do not have the same name as these default config maps. If they have the same name, they are overwritten. For example, if the custom [ConfigMap](#) has the same name as the [ConfigMap](#) for the Kafka cluster, it is overwritten when there is an update to the Kafka cluster.

#### *Additional resources*

- [List of Kafka cluster resources](#) (including config maps)
- [Logging configuration](#)
- [metricsConfig](#)
- [ContainerTemplate schema reference](#)
- [Loading configuration values from external sources](#)

## 11.12.12. Loading configuration values from external sources

Use configuration providers to load configuration data from external sources. The providers operate independently of Strimzi. You can use them to load configuration data for all Kafka components, including producers and consumers. You reference the external source in the configuration of the component and provide access rights. The provider loads data without needing to restart the Kafka component or extracting files, even when referencing a new external source. For example, use providers to supply the credentials for the Kafka Connect connector configuration. The configuration must include any access rights to the external source.

### Enabling configuration providers

You can enable one or more configuration providers using the [config.providers](#) properties in the [spec](#) configuration of a component.

*Example configuration to enable a configuration provider*

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaConnect
metadata:
  name: my-connect
  annotations:
    strimzi.io/use-connector-resources: "true"
spec:
  # ...
  config:
    # ...
    config.providers: env
    config.providers.env.class:
      org.apache.kafka.common.config.provider.EnvVarConfigProvider
      # ...
```

## KubernetesSecretConfigProvider

Loads configuration data from Kubernetes secrets. You specify the name of the secret and the key within the secret where the configuration data is stored. This provider is useful for storing sensitive configuration data like passwords or other user credentials.

## KubernetesConfigMapConfigProvider

Loads configuration data from Kubernetes config maps. You specify the name of the config map and the key within the config map where the configuration data is stored. This provider is useful for storing non-sensitive configuration data.

## EnvVarConfigProvider

Loads configuration data from environment variables. You specify the name of the environment variable where the configuration data is stored. This provider is useful for configuring applications running in containers, for example, to load certificates or JAAS configuration from environment variables mapped from secrets.

## FileConfigProvider

Loads configuration data from a file. You specify the path to the file where the configuration data is stored. This provider is useful for loading configuration data from files that are mounted into containers.

## DirectoryConfigProvider

Loads configuration data from files within a directory. You specify the path to the directory where the configuration files are stored. This provider is useful for loading multiple configuration files and for organizing configuration data into separate files.

To use [KubernetesSecretConfigProvider](#) and [KubernetesConfigMapConfigProvider](#), which are part of the Kubernetes Configuration Provider plugin, you must set up access rights to the namespace that contains the configuration file.

You can use the other providers without setting up access rights. You can supply connector configuration for Kafka Connect or MirrorMaker 2 in this way by doing the following:

- Mount config maps or secrets into the Kafka Connect pod as environment variables or volumes
- Enable `EnvVarConfigProvider`, `FileConfigProvider`, or `DirectoryConfigProvider` in the Kafka Connect or MirrorMaker 2 configuration
- Pass connector configuration using the `template` property in the `spec` of the `KafkaConnect` or `KafkaMirrorMaker2` resource

Using providers help prevent the passing of restricted information through the Kafka Connect REST interface. You can use this approach in the following scenarios:

- Mounting environment variables with the values a connector uses to connect and communicate with a data source
- Mounting a properties file with values that are used to configure Kafka Connect connectors
- Mounting files in a directory that contains values for the TLS truststore and keystore used by a connector

**NOTE**

A restart is required when using a new `Secret` or `ConfigMap` for a connector, which can disrupt other connectors.

*Additional resources*

[KafkaConnectTemplate schema reference](#)

## Loading configuration values from secrets or config maps

Use the `KubernetesSecretConfigProvider` to provide configuration properties from a secret or the `KubernetesConfigMapConfigProvider` to provide configuration properties from a config map.

In this procedure, a config map provides configuration properties for a connector. The properties are specified as key values of the config map. The config map is mounted into the Kafka Connect pod as a volume.

*Prerequisites*

- A Kafka cluster is running.
- The Cluster Operator is running.
- You have a config map containing the connector configuration.

*Example config map with connector properties*

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: my-connector-configuration
data:
  option1: value1
  option2: value2
```

*Procedure*

## 1. Configure the `KafkaConnect` resource.

- Enable the `KubernetesConfigMapConfigProvider`

The specification shown here can support loading values from config maps and secrets.

*Example Kafka Connect configuration to use config maps and secrets*

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaConnect
metadata:
  name: my-connect
  annotations:
    strimzi.io/use-connector-resources: "true"
spec:
  # ...
  config:
    # ...
    config.providers: secrets,configmaps ①
    config.providers.configmaps.class:
      io.strimzi.kafka.KubernetesConfigMapConfigProvider ②
    config.providers.secrets.class: io.strimzi.kafka.KubernetesSecretConfigProvider
    ③
    # ...
```

① The alias for the configuration provider is used to define other configuration parameters. The provider parameters use the alias from `config.providers`, taking the form `config.providers.${alias}.class`.

② `KubernetesConfigMapConfigProvider` provides values from config maps.

③ `KubernetesSecretConfigProvider` provides values from secrets.

## 2. Create or update the resource to enable the provider.

```
kubectl apply -f <kafka_connect_configuration_file>
```

## 3. Create a role that permits access to the values in the external config map.

*Example role to access values from a config map*

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: connector-configuration-role
rules:
  - apiGroups: []
    resources: ["configmaps"]
    resourceNames: ["my-connector-configuration"]
    verbs: ["get"]
```

```
# ...
```

The rule gives the role permission to access the `my-connector-configuration` config map.

4. Create a role binding to permit access to the namespace that contains the config map.

*Example role binding to access the namespace that contains the config map*

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: connector-configuration-role-binding
subjects:
- kind: ServiceAccount
  name: my-connect-connect
  namespace: my-project
roleRef:
  kind: Role
  name: connector-configuration-role
  apiGroup: rbac.authorization.k8s.io
# ...
```

The role binding gives the role permission to access the `my-project` namespace.

The service account must be the same one used by the Kafka Connect deployment. The service account name format is `<cluster_name>-connect`, where `<cluster_name>` is the name of the `KafkaConnect` custom resource.

5. Reference the config map in the connector configuration.

*Example connector configuration referencing the config map*

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaConnector
metadata:
  name: my-connector
  labels:
    strimzi.io/cluster: my-connect
spec:
# ...
config:
  option: ${configmaps:my-project/my-connector-configuration:option1}
  # ...
# ...
```

The placeholder structure is `configmaps:<path_and_file_name>:<property>`. `KubernetesConfigMapConfigProvider` reads and extracts the `option1` property value from the external config map.

## Loading configuration values from environment variables

Use the [EnvVarConfigProvider](#) to provide configuration properties as environment variables. Environment variables can contain values from config maps or secrets.

In this procedure, environment variables provide configuration properties for a connector to communicate with Amazon AWS. The connector must be able to read the [AWS\\_ACCESS\\_KEY\\_ID](#) and [AWS\\_SECRET\\_ACCESS\\_KEY](#). The values of the environment variables are derived from a secret mounted into the Kafka Connect pod.

**NOTE**

The names of user-defined environment variables cannot start with [KAFKA\\_](#) or [STRIMZI\\_](#).

### *Prerequisites*

- A Kafka cluster is running.
- The Cluster Operator is running.
- You have a secret containing the connector configuration.

### *Example secret with values for environment variables*

```
apiVersion: v1
kind: Secret
metadata:
  name: aws-creds
type: Opaque
data:
  awsAccessKey: QUtJQVhYWFnYWFnYWFnYWFn=
  awsSecretAccessKey: Ylhsd1lYTnpkMj15WkE=
```

### *Procedure*

1. Configure the [KafkaConnect](#) resource.
  - Enable the [EnvVarConfigProvider](#)
  - Specify the environment variables using the [template](#) property.

### *Example Kafka Connect configuration to use external environment variables*

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaConnect
metadata:
  name: my-connect
  annotations:
    strimzi.io/use-connector-resources: "true"
spec:
  # ...
  config:
    # ...
  config.providers: env ①
```

```

    config.providers.env.class:
org.apache.kafka.common.config.provider.EnvVarConfigProvider ②
# ...
template:
connectContainer:
env:
- name: AWS_ACCESS_KEY_ID ③
valueFrom:
secretKeyRef:
name: aws-creds ④
key: awsAccessKey ⑤
- name: AWS_SECRET_ACCESS_KEY
valueFrom:
secretKeyRef:
name: aws-creds
key: awsSecretAccessKey
# ...

```

① The alias for the configuration provider is used to define other configuration parameters. The provider parameters use the alias from `config.providers`, taking the form `config.providers.${alias}.class`.

② `EnvVarConfigProvider` provides values from environment variables.

③ The environment variable takes a value from the secret.

④ The name of the secret containing the environment variable.

⑤ The name of the key stored in the secret.

**NOTE** The `secretKeyRef` property references keys in a secret. If you are using a config map instead of a secret, use the `configMapKeyRef` property.

2. Create or update the resource to enable the provider.

```
kubectl apply -f <kafka_connect_configuration_file>
```

3. Reference the environment variable in the connector configuration.

*Example connector configuration referencing the environment variable*

```

apiVersion: kafka.strimzi.io/v1
kind: KafkaConnector
metadata:
  name: my-connector
  labels:
    strimzi.io/cluster: my-connect
spec:
# ...
config:
  option: ${env:AWS_ACCESS_KEY_ID}

```

```
option: ${env:AWS_SECRET_ACCESS_KEY}
# ...
# ...
```

The placeholder structure is `env:<environment_variable_name>`. `EnvVarConfigProvider` reads and extracts the environment variable values from the mounted secret.

## Loading configuration values from a file within a directory

Use the `FileConfigProvider` to provide configuration properties from a file within a directory. Files can be stored in config maps or secrets.

In this procedure, a file provides configuration properties for a connector. A database name and password are specified as properties of a secret. The secret is mounted to the Kafka Connect pod as a volume. Volumes are mounted on the path `/mnt/<volume-name>`.

### Prerequisites

- A Kafka cluster is running.
- The Cluster Operator is running.
- You have a secret containing the connector configuration.

### Example secret with database properties

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
stringData:
  connector.properties: |- ①
    dbUsername: my-username ②
    dbPassword: my-password
```

① The connector configuration in properties file format.

② Database username and password properties used in the configuration.

### Procedure

1. Configure the `KafkaConnect` resource.
  - Enable the `FileConfigProvider`
  - Specify the additional volume using the `template` property.

### Example Kafka Connect configuration to use an external property file

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaConnect
metadata:
  name: my-connect
```

```

spec:
  # ...
  config:
    config.providers: file ①
    config.providers.file.class:
      org.apache.kafka.common.config.provider.FileConfigProvider ②
      #...
    template:
      pod:
        volumes:
          - name: connector-config-volume ③
            secret:
              secretName: mysecret ④
    connectContainer:
      volumeMounts:
        - name: connector-config-volume ⑤
          mountPath: /mnt/mysecret ⑥

```

① The alias for the configuration provider is used to define other configuration parameters.

② `FileConfigProvider` provides values from properties files. The parameter uses the alias from `config.providers`, taking the form `config.providers.${alias}.class`.

③ The name of the volume containing the secret.

④ The name of the secret.

⑤ The name of the mounted volume, which must match the volume name in the `volumes` list.

⑥ The path where the secret is mounted, which must start with `/mnt/`.

2. Create or update the resource to enable the provider.

```
kubectl apply -f <kafka_connect_configuration_file>
```

3. Reference the file properties in the connector configuration as placeholders.

*Example connector configuration referencing the file*

```

apiVersion: kafka.strimzi.io/v1
kind: KafkaConnector
metadata:
  name: my-source-connector
  labels:
    strimzi.io/cluster: my-connect-cluster
spec:
  class: io.debezium.connector.mysql.MySqlConnector
  tasksMax: 2
  config:
    database.hostname: 192.168.99.1
    database.port: "3306"
    database.user: "${file:/mnt/mysecret/connector.properties:dbUsername}"

```

```
database.password: "${file:/mnt/mysecret/connector.properties:dbPassword}"  
database.server.id: "184054"  
#...
```

The placeholder structure is `file:<path_and_file_name>:<property>`. `FileConfigProvider` reads and extracts the database username and password property values from the mounted secret.

## Loading configuration values from multiple files within a directory

Use the `DirectoryConfigProvider` to provide configuration properties from multiple files within a directory. Files can be config maps or secrets.

In this procedure, a secret provides the TLS keystore and truststore user credentials for a connector. The credentials are in separate files. The secrets are mounted into the Kafka Connect pod as volumes. Volumes are mounted on the path `/mnt/<volume-name>`.

### *Prerequisites*

- A Kafka cluster is running.
- The Cluster Operator is running.
- You have a secret containing the user credentials.

### *Example secret with user credentials*

```
apiVersion: v1  
kind: Secret  
metadata:  
  name: my-user  
  labels:  
    strimzi.io/kind: KafkaUser  
    strimzi.io/cluster: my-cluster  
type: Opaque  
data:  
  ca.crt: <public_key> # Public key of the clients CA used to sign this user  
  certificate  
  user.crt: <user_certificate> # Public key of the user  
  user.key: <user_private_key> # Private key of the user  
  user.p12: <store> # PKCS #12 store for user certificates and keys  
  user.password: <password_for_store> # Protects the PKCS #12 store
```

The `my-user` secret provides the keystore credentials (`user.crt` and `user.key`) for the connector.

The `<cluster_name>-cluster-ca-cert` secret generated when deploying the Kafka cluster provides the cluster CA certificate as truststore credentials (`ca.crt`).

### *Procedure*

1. Configure the `KafkaConnect` resource.
  - Enable the `DirectoryConfigProvider`

- Specify the additional volume using the `template` property.

*Example Kafka Connect configuration to use external property files*

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaConnect
metadata:
  name: my-connect
spec:
  # ...
  config:
    config.providers: directory ①
    config.providers.directory.class:
      org.apache.kafka.common.config.provider.DirectoryConfigProvider ②
      #...
    template:
      pod:
        volumes:
          - name: my-user-volume ③
            secret:
              secretName: my-user ④
          - name: cluster-ca-volume
            secret:
              secretName: my-cluster-cluster-ca-cert
    connectContainer:
      volumeMounts:
        - name: my-user-volume ⑤
          mountPath: /mnt/my-user ⑥
        - name: cluster-ca-volume
          mountPath: /mnt/cluster-ca
```

① The alias for the configuration provider is used to define other configuration parameters.

② `DirectoryConfigProvider` provides values from files in a directory. The parameter uses the alias from `config.providers`, taking the form `config.providers.${alias}.class`.

③ The name of the volume containing the secret.

④ The name of the secret.

⑤ The name of the mounted volume, which must match the volume name in the `volumes` list.

⑥ The path where the secret is mounted, which must start with `/mnt/`.

- Create or update the resource to enable the provider.

```
kubectl apply -f <kafka_connect_configuration_file>
```

- Reference the file properties in the connector configuration as placeholders.

*Example connector configuration referencing the files*

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaConnector
metadata:
  name: my-source-connector
  labels:
    strimzi.io/cluster: my-connect-cluster
spec:
  class: io.debezium.connector.mysql.MySqlConnector
  tasksMax: 2
  config:
    # ...
    database.history.producer.security.protocol: SSL
    database.history.producer.ssl.truststore.type: PEM
    database.history.producer.ssl.truststore.certificates:
"${{directory:/mnt/cluster-ca:ca.crt}}"
    database.history.producer.ssl.keystore.type: PEM
    database.history.producer.ssl.keystore.certificate.chain: "${{directory:/mnt/my-
user:user.crt}}"
    database.history.producer.ssl.keystore.key: "${{directory:/mnt/my-
user:user.key}}"
    #...
```

The placeholder structure is `directory:<path>:<file_name>`. `DirectoryConfigProvider` reads and extracts the credentials from the mounted secrets.

### 11.12.13. Customizing Kubernetes resources

A Strimzi deployment creates Kubernetes resources, such as `Deployment`, `Pod`, and `Service` resources. These resources are managed by Strimzi operators. Only the operator that is responsible for managing a particular Kubernetes resource can change that resource. If you try to manually change an operator-managed Kubernetes resource, the operator will revert your changes back.

Changing an operator-managed Kubernetes resource can be useful if you want to perform certain tasks, such as the following:

- Adding custom labels or annotations that control how `Pods` are treated by Istio or other services
- Managing how `Loadbalancer`-type Services are created by the cluster

To make the changes to a Kubernetes resource, you can use the `template` property within the `spec` section of various Strimzi custom resources.

Here is a list of the custom resources where you can apply the changes:

- `Kafka.spec.kafka`
- `Kafka.spec.entityOperator`
- `Kafka.spec.kafkaExporter`

- `Kafka.spec.cruiseControl`
- `KafkaNodePool.spec`
- `KafkaConnect.spec`
- `KafkaMirrorMaker2.spec`
- `KafkaBridge.spec`
- `KafkaUser.spec`

For more information about these properties, see the [Strimzi Custom Resource API Reference](#).

The Strimzi Custom Resource API Reference provides more details about the customizable fields.

In the following example, the `template` property is used to modify the labels in a Kafka broker's pod.

#### *Example template customization*

```
apiVersion: kafka.strimzi.io/v1
kind: Kafka
metadata:
  name: my-cluster
  labels:
    app: my-cluster
spec:
  kafka:
    # ...
    template:
      pod:
        metadata:
          labels:
            mylabel: myvalue
# ...
```

## Customizing the image pull policy

Strimzi allows you to customize the image pull policy for containers in all pods deployed by the Cluster Operator. The image pull policy is configured using the environment variable `STRIMZI_IMAGE_PULL_POLICY` in the Cluster Operator deployment. The `STRIMZI_IMAGE_PULL_POLICY` environment variable can be set to three different values:

### **Always**

Container images are pulled from the registry every time the pod is started or restarted.

### **IfNotPresent**

Container images are pulled from the registry only when they were not pulled before.

### **Never**

Container images are never pulled from the registry.

Currently, the image pull policy can only be customized for all Kafka, Kafka Connect, and Kafka

MirrorMaker clusters at once. Changing the policy will result in a rolling update of all your Kafka, Kafka Connect, and Kafka MirrorMaker clusters.

#### Additional resources

- [Updating images](#).

### Applying a termination grace period

Apply a termination grace period to give a Kafka cluster enough time to shut down cleanly.

Specify the time using the `terminationGracePeriodSeconds` property. Add the property to the `template.pod` configuration of the `Kafka` custom resource.

The time you add will depend on the size of your Kafka cluster. The Kubernetes default for the termination grace period is 30 seconds. If you observe that your clusters are not shutting down cleanly, you can increase the termination grace period.

A termination grace period is applied every time a pod is restarted. The period begins when Kubernetes sends a *term* (termination) signal to the processes running in the pod. The period should reflect the amount of time required to transfer the processes of the terminating pod to another pod before they are stopped. After the period ends, a *kill* signal stops any processes still running in the pod.

The following example adds a termination grace period of 120 seconds to the `Kafka` custom resource. You can also specify the configuration in the custom resources of other Kafka components.

#### Example termination grace period configuration

```
apiVersion: kafka.strimzi.io/v1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    template:
      pod:
        terminationGracePeriodSeconds: 120
        # ...
    # ...
```

## 11.13. Configuring logging

Strimzi operators and Kafka components use log4j2 for logging.

You can set log levels to `INFO`, `ERROR`, `WARN`, `TRACE`, `DEBUG`, `FATAL` or `OFF`.

By default, logs are produced at predefined levels. Loggers that are not explicitly configured inherit their configuration from parent loggers.

**WARNING**

Setting log levels to `DEBUG` or `TRACE` can produce large volumes of output and impact performance.

### 11.13.1. Choosing a logging method

You configure logging in custom resources by setting the `spec.logging` property.

Two methods are available:

#### Inline logging

- Define loggers and log levels directly in the resource.
- Suitable when you only need to adjust log levels.
- Quick to set up, but limited in flexibility.

#### External logging

- Reference a `ConfigMap` that contains a complete `log4j2.properties` file.
- Suitable when you need advanced options, such as the following:
  - Centralized, reusable configuration across resources
  - Logging filters to target logs for a specific Kafka cluster, topic, or operator
  - Custom appenders and layouts

Which method should you use?

- Use inline logging for simple changes to log levels.
- Use external logging for complex, reusable, or filtered configurations.

### 11.13.2. Configuring inline logging

Use inline logging when you only need to adjust log levels for Strimzi components.

Inline logging is defined directly in the `spec.logging` property of the custom resource. Use the `loggers` property to set the root logger level and levels for specific classes or loggers.

*Example inline logging configuration*

```
# ...
logging:
  type: inline
  loggers:
    rootLogger.level: INFO
# ...
```

When a resource managed by the Cluster Operator is updated, changes to inline logging are applied dynamically without requiring a pod restart.

### 11.13.3. Configuring external logging

Use external logging when you need a reusable or advanced configuration.

External logging requires two steps:

1. Create a [ConfigMap](#) that contains a complete `log4j2.properties` file.
2. Reference the [ConfigMap](#) in the `spec.logging` property of the custom resource.

*Example external logging configuration*

```
# ...
logging:
  type: external
  valueFrom:
    configMapKeyRef:
      # name and key are mandatory
      name: my-config-map
      key: log4j2.properties
# ...
```

The Cluster Operator detects changes to the referenced [ConfigMap](#) and applies them dynamically without requiring a pod restart.

### 11.13.4. Creating a [ConfigMap](#) for logging

To configure external logging, define logging properties in a [ConfigMap](#) and reference it in the `spec.logging` property of the resource. Place the configuration under the key `log4j2.properties`.

*Procedure*

1. Create the [ConfigMap](#) as a YAML file or from a properties file.

For example, this YAML file defines a root logger for a [Kafka](#) resource:

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: logging-configmap
data:
  log4j2.properties: |
    rootLogger.level = "INFO"
    appender.console.type = Console
    appender.console.name = STDOUT
    appender.console.layout.type = PatternLayout
    appender.console.layout.pattern = %d{yyyy-MM-dd HH:mm:ss} %-5p [%t] %c{1}:%L -
%m%n
    rootLogger.level = INFO
    rootLogger.appenderRefs = console
    rootLogger.appenderRef.console.ref = STDOUT
```

```
rootLogger.additivity = false
```

If you are using a properties file, define the logging configuration and specify the file when creating a [ConfigMap](#).

For example, in a properties file:

```
# Define the logger
rootLogger.level = "INFO"
# ...
```

Then create the [ConfigMap](#):

```
kubectl create configmap logging-configmap --from-file=log4j2.properties
```

2. Add `external` logging to the `spec` of the `Kafka` resource, specifying the `name` and `key` of the [ConfigMap](#):

```
# ...
logging:
  type: external
  valueFrom:
    configMapKeyRef:
      name: logging-configmap
      key: log4j2.properties
# ...
```

3. Apply the changes to the `Kafka` configuration.

### 11.13.5. Configuring Cluster Operator logging

Cluster Operator logging is controlled through a [ConfigMap](#) named `strimzi-cluster-operator`.

This [ConfigMap](#) is created with default values during installation and is described in the file: `install/cluster-operator/050-ConfigMap-strimzi-cluster-operator.yaml`.

To update the logging configuration, you can do one of the following:

- Edit the YAML file and reapply it:

```
kubectl create -f install/cluster-operator/050-ConfigMap-strimzi-cluster-
operator.yaml
```

- Edit the [ConfigMap](#) directly:

```
kubectl edit configmap strimzi-cluster-operator
```

Using this [ConfigMap](#), you can control:

- Root logger level
- Log output format
- Log levels for different components
- Kafka [AdminClient](#) logging levels
- Netty logging Levels
- How often logging configuration is loaded

The `monitorInterval` setting determines how often, in seconds, the logging configuration is dynamically reloaded. The default is 30 seconds.

If the [ConfigMap](#) is missing when the Cluster Operator is deployed, the default logging configuration is used.

If the [ConfigMap](#) is deleted after deployment, the most recently loaded logging configuration remains in effect. Create a new [ConfigMap](#) to load a new logging configuration.

**WARNING** Do not remove the `monitorInterval` option from the [ConfigMap](#).

### 11.13.6. Adding logging filters to Strimzi operators

You can add logging filters to Strimzi operators by providing a complete `log4j2.properties` file in a [ConfigMap](#).

Filters are useful when logging output is too verbose. For example, if `rootLogger.level="DEBUG"`, you can use filters to reduce noise and focus on a specific resource.

Filters use **markers** to include or exclude logs. A marker identifies a resource by kind, namespace, and name. For example, a marker might isolate logs for a failing Kafka cluster: `Kafka(my-namespace/my-kafka-cluster)`.

The following example defines a single logging filter (`filter1`):

```
appender.console.filter.filter1.type = MarkerFilter ①
appender.console.filter.filter1.onMatch = ACCEPT ②
appender.console.filter.filter1.onMismatch = DENY ③
appender.console.filter.filter1.marker = Kafka(my-namespace/my-kafka-cluster) ④
```

① The `MarkerFilter` compares a specified marker.

② `onMatch` accepts logs if the marker matches.

③ `onMismatch` rejects logs if the marker does not match.

④ Marker format: `kind(namespace/resource_name)`.

To define multiple filters, configure each one separately with a unique filter name (for example, `filter1`, `filter2`).

#### *Additional resources*

- [Apache Filters](#)
- [Configuring Kafka](#)
- [Cluster Operator logging](#)
- [Topic Operator logging](#)
- [User Operator logging](#)

### 11.13.7. Configuring logging filters for the Cluster Operator

To add logging filters to the Cluster Operator, update its logging [ConfigMap](#).

#### *Procedure*

1. Edit the [ConfigMap](#) used by the Cluster Operator.
  - Update the file `install/cluster-operator/050-ConfigMap-strimzi-cluster-operator.yaml`.
  - Or edit the [ConfigMap](#) directly in the cluster and save the changes: `kubectl edit configmap strimzi-cluster-operator`.

For example, adding a filter to the ConfigMap:

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: strimzi-cluster-operator
data:
  log4j2.properties: |
    rootLogger.level = "INFO"
    rootLogger.appenderRefs = console
    rootLogger.appenderRef.console.ref = STDOUT

    appender.console.type = Console
    appender.console.name = STDOUT
    appender.console.layout.type = PatternLayout
    appender.console.layout.pattern = %d{yyyy-MM-dd HH:mm:ss} %-5p [%t] %c{1}:%L
    - %m%n

    appender.console.filter.filter1.type=MarkerFilter
    appender.console.filter.filter1.onMatch=ACCEPT
    appender.console.filter.filter1.onMismatch=DENY
    appender.console.filter.filter1.marker=Kafka(my-namespace/my-kafka-cluster)
```

2. If you edited the YAML file, apply the changes to the [ConfigMap](#) configuration.

## 11.13.8. Configuring logging filters for the Topic or User Operator

To add logging filters to the Topic or User Operator, create or edit a logging [ConfigMap](#) and reference it in the [Kafka](#) resource.

### Procedure

1. Create or edit a logging [ConfigMap](#).

For example, adding filter properties for `my-topic` topic:

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: logging-configmap
data:
  log4j2.properties: |
    rootLogger.level = "INFO"
    rootLogger.appenderRefs = console
    rootLogger.appenderRef.console.ref = STDOUT

    appender.console.type = Console
    appender.console.name = STDOUT
    appender.console.layout.type = PatternLayout
    appender.console.layout.pattern = %d{yyyy-MM-dd HH:mm:ss} %-5p [%t] %c{1}:%L -
%m%n

    appender.console.filter.filter1.type = MarkerFilter
    appender.console.filter.filter1.onMatch = ACCEPT
    appender.console.filter.filter1.onMismatch = DENY
    appender.console.filter.filter1.marker = KafkaTopic(my-namespace/my-topic)
```

Or define the configuration in a properties file and specify the file when creating the [ConfigMap](#):

For example, in a properties file:

```
# Define the logger
rootLogger.level = "INFO"
rootLogger.appenderRefs = console
rootLogger.appenderRef.console.ref = STDOUT
# Define the appenders
appender.console.type = Console
appender.console.name = STDOUT
appender.console.layout.type = PatternLayout
appender.console.layout.pattern = %d{yyyy-MM-dd HH:mm:ss} %-5p [%t] %c{1}:%L - %m%n
# Set the filters
appender.console.filter.filter1.type = MarkerFilter
appender.console.filter.filter1.onMatch = ACCEPT
appender.console.filter.filter1.onMismatch = DENY
appender.console.filter.filter1.marker = KafkaTopic(my-namespace/my-topic)
```

```
# ...
```

Then create the [ConfigMap](#):

```
kubectl create configmap logging-configmap --from-file=log4j2.properties
```

2. Add `external` logging to the `spec` of the [Kafka](#) resource, specifying the `name` and `key` of the [ConfigMap](#) under the `topicOperator` or `userOperator` configuration:

```
spec:  
  # ...  
  entityOperator:  
    topicOperator:  
      logging:  
        type: external  
        valueFrom:  
          configMapKeyRef:  
            name: logging-configmap  
            key: log4j2.properties
```

3. Apply the changes to the [Kafka](#) configuration.

### 11.13.9. Configurable loggers

You can configure specific loggers for Kafka components and Strimzi operators.

#### Kafka components

- [Kafka loggers](#)
- [Kafka Connect and MirrorMaker 2 loggers](#)
- [HTTP Bridge loggers](#)
- [Cruise Control loggers](#)

#### Strimzi operators

- [Cluster Operator loggers](#)
- [Topic Operator loggers](#)
- [User Operator loggers](#)

For information about log levels, see [Apache logging services](#).

### 11.13.10. Accessing and using logs

You can view logs for Strimzi operators and Kafka components without any configuration. If you configure logging, you can control the volume and type of information shown. Logging filters further narrow output to specific resources.

Logs can help you:

- Confirm successful component startup
- Monitor cluster activity
- Diagnose issues such as connection failures, authorization errors, or replication problems

View logs with `kubectl` commands. For example:

- View logs from a pod:

```
kubectl logs my-cluster-kafka-0
```

- Follow logs in real time:

```
kubectl logs -f my-cluster-kafka-0
```

- View logs from a specific container in a pod:

```
kubectl logs my-cluster-entity-operator-0 -c topic-operator
```

**NOTE**

Use the `kubectl logs` command for direct, real-time debugging of recent pod logs. For production environments, implement a persistent, Kubernetes-supported cluster-level logging architecture to capture logs from all components and support monitoring and troubleshooting.

## 11.14. Restrictions on Kubernetes labels

Kubernetes labels make it easier to organize, manage, and discover Kubernetes resources within your applications. The Cluster Operator is responsible for applying the following Kubernetes labels to the operands it deploys. These labels cannot be overridden through template configuration of Strimzi resources:

- `app.kubernetes.io/name`: Identifies the component type within Strimzi, such as `kafka` or `cruise-control`.
- `app.kubernetes.io/instance`: Represents the name of the custom resource to which the operand belongs to. For instance, if a Kafka custom resource is named `my-cluster`, this label will bear that name on the associated pods.
- `app.kubernetes.io/part-of`: Similar to `app.kubernetes.io/instance`, but prefixed with `strimzi-`.
- `app.kubernetes.io/managed-by`: Defines the application responsible for managing the operand, such as `strimzi-cluster-operator` or `strimzi-user-operator`.

Example Kubernetes labels on a Kafka pod when deploying a Kafka custom resource named `my-cluster`

```
apiVersion: kafka.strimzi.io/v1
```

```
kind: Pod
metadata:
  name: my-cluster-kafka-0
  labels:
    app.kubernetes.io/instance: my-cluster
    app.kubernetes.io/managed-by: strimzi-cluster-operator
    app.kubernetes.io/name: kafka
    app.kubernetes.io/part-of: strimzi-my-cluster
spec:
  # ...
```

# Chapter 12. Using the Topic Operator to manage Kafka topics

The [KafkaTopic](#) resource configures topics, including partition and replication factor settings. When you create, modify, or delete a topic using [KafkaTopic](#), the Topic Operator ensures that these changes are reflected in the Kafka cluster.

For more information on the [KafkaTopic](#) resource, see the [KafkaTopic schema reference](#).

## 12.1. Topic management

The [KafkaTopic](#) resource is responsible for managing a single topic within a Kafka cluster.

The Topic Operator operates as follows:

- When a [KafkaTopic](#) is created, deleted, or changed, the Topic Operator performs the corresponding operation on the Kafka topic.

If a topic is created, deleted, or modified directly within the Kafka cluster, without the presence of a corresponding [KafkaTopic](#) resource, the Topic Operator does not manage that topic. The Topic Operator will only manage Kafka topics associated with [KafkaTopic](#) resources and does not interfere with topics managed independently within the Kafka cluster. If a [KafkaTopic](#) does exist for a Kafka topic, any configuration changes made outside the resource are reverted.

The Topic Operator can detect cases where multiple [KafkaTopic](#) resources are attempting to manage a Kafka topic using the same `.spec.topicName`. Only the oldest resource is reconciled, while the other resources fail with a resource conflict error.

## 12.2. Topic naming conventions

A [KafkaTopic](#) resource includes a name for the topic and a label that identifies the name of the Kafka cluster it belongs to.

*Label identifying a Kafka cluster for topic handling*

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaTopic
metadata:
  name: topic-name-1
  labels:
    strimzi.io/cluster: my-cluster
spec:
  topicName: topic-name-1
```

The label provides the cluster name of the [Kafka](#) resource. The Topic Operator uses the label as a mechanism for determining which [KafkaTopic](#) resources to manage. If the label does not match the Kafka cluster, the Topic Operator cannot see the [KafkaTopic](#), and the topic is not created.

Kafka and Kubernetes have their own naming validation rules, and a Kafka topic name might not be a valid resource name in Kubernetes. If possible, try and stick to a naming convention that works for both.

Consider the following guidelines:

- Use topic names that reflect the nature of the topic
- Be concise and keep the name under 63 characters
- Use all lower case and hyphens
- Avoid special characters, spaces or symbols

The `KafkaTopic` resource allows you to specify the Kafka topic name using the `metadata.name` field. However, if the desired Kafka topic name is not a valid Kubernetes resource name, you can use the `spec.topicName` property to specify the actual name. The `spec.topicName` field is optional, and when it's absent, the Kafka topic name defaults to the `metadata.name` of the topic. When a topic is created, the topic name cannot be changed later.

*Example of supplying a valid Kafka topic name*

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaTopic
metadata:
  name: my-topic-1 ①
spec:
  topicName: My.Topic.1 ②
  # ...
```

① A valid topic name that works in Kubernetes.

② A Kafka topic name that uses upper case and periods, which are invalid in Kubernetes.

If more than one `KafkaTopic` resource refers to the same Kafka topic, the resource that was created first is considered to be the one managing the topic. The status of the newer resources is updated to indicate a conflict, and their `Ready` status is changed to `False`.

A Kafka client application, such as Kafka Streams, can automatically create topics with invalid Kubernetes resource names. If you want to manage these topics, you must create `KafkaTopic` resources with a different `.metadata.name`, as shown in the previous example.

**NOTE**

For more information on the requirements for identifiers and names in a cluster, refer to the Kubernetes documentation [Object Names and IDs](#).

## 12.3. Handling changes to topics

Configuration changes only go in one direction: from the `KafkaTopic` resource to the Kafka topic. Any changes to a Kafka topic managed outside the `KafkaTopic` resource are reverted.

### 12.3.1. Downgrading to a Strimzi version that uses internal topics to store topic metadata

If you are reverting back to a version of Strimzi earlier than 0.41, which uses internal topics for the storage of topic metadata, you still downgrade your Cluster Operator to the previous version, then downgrade Kafka brokers and client applications to the previous Kafka version as standard.

### 12.3.2. Automatic creation of topics

Applications can trigger the automatic creation of topics in the Kafka cluster. By default, the Kafka broker configuration `auto.create.topics.enable` is set to `true`, allowing the broker to create topics automatically when an application attempts to produce or consume from a non-existing topic. Applications might also use the Kafka `AdminClient` to automatically create topics. When an application is deployed along with its `KafkaTopic` resources, it is possible that automatic topic creation in the cluster happens before the Topic Operator can react to the `KafkaTopic`.

The topics created for an application deployment are initially created with default topic configuration. If the Topic Operator attempts to reconfigure the topics based on `KafkaTopic` resource specifications included with the application deployment, the operation might fail because the required change to the configuration is not allowed. For example, if the change means lowering the number of topic partitions. For this reason, it is recommended to disable `auto.create.topics.enable` in the Kafka cluster configuration.

## 12.4. Configuring Kafka topics

Use the properties of the `KafkaTopic` resource to configure Kafka topics. Changes made to topic configuration in the `KafkaTopic` are propagated to Kafka.

You can use `kubectl apply` to create or modify topics, and `kubectl delete` to delete existing topics.

For example:

- `kubectl apply -f <topic_config_file>`
- `kubectl delete KafkaTopic <topic_name>`

To be able to delete topics, `delete.topic.enable` must be set to `true` (default) in the `spec.kafka.config` of the Kafka resource.

This procedure shows how to create a topic with 10 partitions and 2 replicas.

*Before you begin*

The KafkaTopic resource does not allow the following changes:

- Renaming the topic defined in `spec.topicName`. A mismatch between `spec.topicName` and `status.topicName` will be detected.
- Decreasing the number of partitions using `spec.partitions` (not supported by Kafka).
- Modifying the number of replicas specified in `spec.replicas`.

**WARNING**

Increasing `spec.partitions` for topics with keys will alter the partitioning of records, which can cause issues, especially when the topic uses semantic partitioning.

*Prerequisites*

- A running Kafka cluster configured with a Kafka broker listener using mTLS authentication and TLS encryption.
- A running Topic Operator (typically deployed with the Entity Operator).
- For deleting a topic, `delete.topic.enable=true` (default) in the `spec.kafka.config` of the `Kafka` resource.

*Procedure*

1. Configure the `KafkaTopic` resource.

*Example Kafka topic configuration*

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaTopic
metadata:
  name: my-topic-1
  labels:
    strimzi.io/cluster: my-cluster
spec:
  partitions: 10
  replicas: 2
```

**TIP**

When modifying a topic, you can get the current version of the resource using `kubectl get kafkatopic my-topic-1 -o yaml`.

2. Create the `KafkaTopic` resource in Kubernetes.

```
kubectl apply -f <topic_config_file>
```

3. Wait for the ready status of the topic to change to `True`:

```
kubectl get kafkatopics -o wide -w -n <namespace>
```

*Kafka topic status*

NAME	CLUSTER	PARTITIONS	REPLICATION FACTOR	READY
my-topic-1	my-cluster	10	3	True
my-topic-2	my-cluster	10	3	
my-topic-3	my-cluster	10	3	True

Topic creation is successful when the `READY` output shows `True`.

4. If the **READY** column stays blank, get more details on the status from the resource YAML or from the Topic Operator logs.

Status messages provide details on the reason for the current status.

```
oc get kafkatopics my-topic-2 -o yaml
```

*Details on a topic with a **NotReady** status*

```
# ...
status:
  conditions:
    - lastTransitionTime: "2022-06-13T10:14:43.351550Z"
      message: Number of partitions cannot be decreased
      reason: PartitionDecreaseException
      status: "True"
      type: NotReady
```

In this example, the reason the topic is not ready is because the original number of partitions was reduced in the **KafkaTopic** configuration. Kafka does not support this.

After resetting the topic configuration, the status shows the topic is ready.

```
kubectl get kafkatopics my-topic-2 -o wide -w -n <namespace>
```

*Status update of the topic*

NAME	CLUSTER	PARTITIONS	REPLICATION FACTOR	READY
my-topic-2	my-cluster	10	3	True

Fetching the details shows no messages

```
kubectl get kafkatopics my-topic-2 -o yaml
```

*Details on a topic with a **READY** status*

```
# ...
status:
  conditions:
    - lastTransitionTime: '2022-06-13T10:15:03.761084Z'
      status: 'True'
      type: Ready
```

## 12.5. Configuring topics for replication and number of partitions

The recommended configuration for topics managed by the Topic Operator is a topic replication factor of 3, and a minimum of 2 in-sync replicas.

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaTopic
metadata:
  name: my-topic
  labels:
    strimzi.io/cluster: my-cluster
spec:
  partitions: 10 ①
  replicas: 3 ②
  config:
    min.insync.replicas: 2 ③
  #...
```

① The number of partitions for the topic.

② The number of replica topic partitions. Changing the number of replicas in the topic configuration requires a deployment of Cruise Control. For more information, see [Using Cruise Control to modify topic replication factor](#).

③ The minimum number of replica partitions that a message must be successfully written to, or an exception is raised.

**NOTE**

In-sync replicas are used in conjunction with the `acks` configuration for producer applications. The `acks` configuration determines the number of follower partitions a message must be replicated to before the message is acknowledged as successfully received. Replicas need to be reassigned when adding or removing brokers (see [Scaling clusters by adding or removing brokers](#)).

### Additional resources

- [Downgrading Strimzi](#)
- [Partition reassignment tool overview](#)
- [Using Cruise Control for cluster rebalancing](#)

## 12.6. Managing KafkaTopic resources without impacting Kafka topics

This procedure describes how to convert Kafka topics that are currently managed through the `KafkaTopic` resource into unmanaged topics. This capability can be useful in various scenarios. For instance, you might want to update the `metadata.name` of a `KafkaTopic` resource. You can only do that by deleting the original `KafkaTopic` resource and recreating a new one.

By annotating a `KafkaTopic` resource with `strimzi.io/managed=false`, you indicate that the Topic Operator should no longer manage that particular topic. This allows you to retain the Kafka topic while making changes to the resource's configuration or other administrative tasks.

#### Prerequisites

- The Cluster Operator must be deployed.

#### Procedure

1. Annotate the `KafkaTopic` resource in Kubernetes, setting `strimzi.io/managed` to `false`:

```
kubectl annotate kafkatopic my-topic-1 strimzi.io/managed="false" --overwrite
```

Specify the `metadata.name` of the topic in your `KafkaTopic` resource, which is `my-topic-1` in this example.

2. Check the status of the `KafkaTopic` resource to make sure the request was successful:

```
kubectl get kafkatopic my-topic-1 -o yaml
```

*Example topic with an `Unmanaged` status*

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaTopic
metadata:
  generation: 124
  name: my-topic-1
  finalizer:
  - strimzi.io/topic-operator
  labels:
    strimzi.io/cluster: my-cluster
  annotations:
    strimzi.io/managed: "false"
spec:
  partitions: 10
  replicas: 2
  config:
    retention.ms: 432000000
status:
  observedGeneration: 124 ①
  conditions:
  - lastTransitionTime: "2024-08-22T06:07:57.671085635Z"
    status: "True"
    type: Unmanaged ②
```

① The value of `metadata.generation` must match `status.observedGeneration`.

② The `Unmanaged` condition means that the `KafkaTopic` is no longer reconciled.

3. You can now make changes to the `KafkaTopic` resource without it affecting the Kafka topic it was

managing.

For example, to change the `metadata.name`, do as follows:

- a. Delete the original `KafkaTopic` resource:

```
kubectl delete kafkatopic <kafka_topic_name>
```

- b. Recreate the `KafkaTopic` resource with a different `metadata.name`, but use `spec.topicName` to refer to the same topic that was managed by the original
4. If you haven't deleted the original `KafkaTopic` resource, and you wish to resume management of the Kafka topic again, set the `strimzi.io/managed` annotation to `true` or remove the annotation.

## 12.7. Enabling topic management for existing Kafka topics

This procedure describes how to enable topic management for topics that are not currently managed through the `KafkaTopic` resource. You do this by creating a matching `KafkaTopic` resource.

### Prerequisites

- The Cluster Operator must be deployed.

### Procedure

1. Create a `KafkaTopic` resource with a `metadata.name` that is the same as the Kafka topic.

Or use `spec.topicName` if the name of the topic in Kafka would not be a legal Kubernetes resource name.

### Example Kafka topic configuration

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaTopic
metadata:
  name: my-topic-1
  labels:
    strimzi.io/cluster: my-cluster
spec:
  partitions: 10
  replicas: 2
```

In this example, the Kafka topic is named `my-topic-1`.

The Topic Operator checks whether the topic is managed by another `KafkaTopic` resource. If it is, the older resource takes precedence and a resource conflict error is returned in the status of the new resource.

2. Apply the `KafkaTopic` resource:

```
kubectl apply -f <topic_configuration_file>
```

3. Wait for the operator to update the topic in Kafka.

The operator updates the Kafka topic with the `spec` of the `KafkaTopic` that has the same name.

4. Check the status of the `KafkaTopic` resource to make sure the request was successful:

```
oc get kafkatopics my-topic-1 -o yaml
```

*Example topic with a Ready status*

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaTopic
metadata:
  generation: 1
  name: my-topic-1
  labels:
    strimzi.io/cluster: my-cluster
spec:
  partitions: 10
  replicas: 2
# ...
status:
  observedGeneration: 1 ①
  topicName: my-topic-1
  conditions:
  - type: Ready
    status: True
  lastTransitionTime: 20230301T103000Z
```

① Successful reconciliation of the resource means the topic is now managed.

The value of `metadata.generation` (the current version of the deployment) must `match` `status.observedGeneration` (the latest reconciliation of the resource).

## 12.8. Deleting managed topics

The Topic Operator supports the deletion of topics managed through the `KafkaTopic` resource with or without Kubernetes finalizers. This is determined by the `STRIMZI_USE_FINALIZERS` Topic Operator environment variable. By default, this is set to `true`, though it can be set to `false` in the Topic Operator `env` configuration if you do not want the Topic Operator to add finalizers.

Finalizers ensure orderly and controlled deletion of `KafkaTopic` resources. A finalizer for the Topic Operator is added to the metadata of the `KafkaTopic` resource:

*Finalizer to control topic deletion*

```
apiVersion: kafka.strimzi.io/v1
```

```
kind: KafkaTopic
metadata:
  generation: 1
  name: my-topic-1
  finalizers:
    - strimzi.io/topic-operator
  labels:
    strimzi.io/cluster: my-cluster
```

In this example, the finalizer is added for topic `my-topic-1`. The finalizer prevents the topic from being fully deleted until the finalization process is complete. If you then delete the topic using `kubectl delete kafkatopic my-topic-1`, a timestamp is added to the metadata:

*Finalizer timestamp on deletion*

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaTopic
metadata:
  generation: 1
  name: my-topic-1
  finalizers:
    - strimzi.io/topic-operator
  labels:
    strimzi.io/cluster: my-cluster
  deletionTimestamp: 20230301T000000.000
```

The resource is still present. If the deletion fails, it is shown in the status of the resource.

When the finalization tasks are successfully executed, the finalizer is removed from the metadata, and the resource is fully deleted.

Finalizers also serve to prevent related resources from being deleted. If the Topic Operator is not running, it won't be able to remove its finalizer from the `metadata.finalizers`. And any attempt to directly delete the `KafkaTopic` resources or the namespace will fail or timeout, leaving the namespace in a stuck terminating state. If this happens, you can bypass the finalization process by [removing the finalizers on topics](#).

## 12.9. Removing finalizers on topics

If the Topic Operator is not running, and you want to bypass the finalization process when deleting managed topics, you must remove the finalizers. You can do this manually by editing the resources directly or by using a command.

To remove finalizers on all topics, use the following command:

*Removing finalizers on topics*

```
kubectl get kt -o=json | jq '.items[].metadata.finalizers = null' | kubectl apply -f -
```

The command uses the [jq command line JSON parser tool](#) to modify the [KafkaTopic \(kt\)](#) resources by setting the finalizers to `null`. You can also use the command for a specific topic:

*Removing a finalizer on a specific topic*

```
kubectl get kt <topic_name> -o=json | jq '.metadata.finalizers = null' | kubectl apply -f -
```

After running the command, you can go ahead and delete the topics. Alternatively, if the topics were already being deleted but were blocked due to outstanding finalizers then their deletion should complete.

**WARNING**

Be careful when removing finalizers, as any cleanup operations associated with the finalization process are not performed if the Topic Operator is not running. For example, if you remove the finalizer from a [KafkaTopic](#) resource and subsequently delete the resource, the related Kafka topic won't be deleted.

## 12.10. Considerations when disabling topic deletion

When the `delete.topic.enable` configuration in Kafka is set to `false`, topics cannot be deleted. This might be required in certain scenarios, but it introduces a consideration when using the Topic Operator.

As topics cannot be deleted, finalizers added to the metadata of a [KafkaTopic](#) resource to control topic deletion are never removed by the Topic Operator (though they can be [removed manually](#)). Similarly, any Custom Resource Definitions (CRDs) or namespaces associated with topics cannot be deleted.

Before configuring `delete.topic.enable=false`, assess these implications to ensure it aligns with your specific requirements.

**NOTE**

To avoid using finalizers, you can set the `STRIMZI_USE_FINALIZERS` Topic Operator environment variable to `false`.

## 12.11. Tuning request batches for topic operations

The Topic Operator uses the request batching capabilities of the Kafka Admin API for operations on topic resources. You can fine-tune the batching mechanism using the following operator configuration properties:

- `STRIMZI_MAX_QUEUE_SIZE` to set the maximum size of the topic event queue. The default value is 1024.
- `STRIMZI_MAX_BATCH_SIZE` to set the maximum number of topic events allowed in a single batch. The default value is 100.
- `MAX_BATCH_LINGER_MS` to specify the maximum time to wait for a batch to accumulate items before processing. The default is 100 milliseconds.

If the maximum size of the request batching queue is exceeded, the Topic Operator shuts down and is restarted. To prevent frequent restarts, consider adjusting the `STRIMZI_MAX_QUEUE_SIZE` property to accommodate the typical load.

# Chapter 13. Using the User Operator to manage Kafka users

When you create, modify or delete a user using the `KafkaUser` resource, the User Operator ensures that these changes are reflected in the Kafka cluster.

For more information on the `KafkaUser` resource, see the [KafkaUser schema reference](#).

## 13.1. Configuring Kafka users

Use the properties of the `KafkaUser` resource to configure Kafka users.

You can use `kubectl apply` to create or modify users, and `kubectl delete` to delete existing users.

For example:

- `kubectl apply -f <user_config_file>`
- `kubectl delete KafkaUser <user_name>`

Users represent Kafka clients. When you configure Kafka users, you enable the user authentication and authorization mechanisms required by clients to access Kafka. The mechanism used must match the equivalent `Kafka` configuration. For more information on using `Kafka` and `KafkaUser` resources to secure access to Kafka brokers, see [Securing access to a Kafka cluster](#).

### Prerequisites

- A running Kafka cluster configured with a Kafka broker listener using mTLS authentication and TLS encryption.
- A running User Operator (typically deployed with the Entity Operator).

### Procedure

1. Configure the `KafkaUser` resource.

This example specifies mTLS authentication and simple authorization using ACLs.

#### Example Kafka user configuration

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaUser
metadata:
  name: my-user-1
  labels:
    strimzi.io/cluster: my-cluster
spec:
  authentication:
    type: tls
  authorization:
    type: simple
  acls:
```

```

# Example consumer Acls for topic my-topic using consumer group my-group
- resource:
    type: topic
    name: my-topic
    patternType: literal
  operations:
    - Describe
    - Read
  host: "*"
- resource:
    type: group
    name: my-group
    patternType: literal
  operations:
    - Read
  host: "*"
# Example Producer Acls for topic my-topic
- resource:
    type: topic
    name: my-topic
    patternType: literal
  operations:
    - Create
    - Describe
    - Write
  host: "*"

```

2. Create the **KafkaUser** resource in Kubernetes.

```
kubectl apply -f <user_config_file>
```

3. Wait for the ready status of the user to change to **True**:

```
kubectl get kafkausers -o wide -w -n <namespace>
```

#### *Kafka user status*

NAME	CLUSTER	AUTHENTICATION	AUTHORIZATION	READY
my-user-1	my-cluster	tls	simple	True
my-user-2	my-cluster	tls	simple	
my-user-3	my-cluster	tls	simple	True

User creation is successful when the **READY** output shows **True**.

4. If the **READY** column stays blank, get more details on the status from the resource YAML or User Operator logs.

Messages provide details on the reason for the current status.

```
kubectl get kafkausers my-user-2 -o yaml
```

*Details on a user with a NotReady status*

```
# ...
status:
  conditions:
  - lastTransitionTime: "2022-06-10T10:07:37.238065Z"
    message: Simple authorization ACL rules are configured but not supported in the
              Kafka cluster configuration.
    reason: InvalidResourceException
    status: "True"
    type: NotReady
```

In this example, the reason the user is not ready is because simple authorization is not enabled in the [Kafka](#) configuration.

*Kafka configuration for simple authorization*

```
apiVersion: kafka.strimzi.io/v1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    authorization:
      type: simple
```

After updating the Kafka configuration, the status shows the user is ready.

```
kubectl get kafkausers my-user-2 -o wide -w -n <namespace>
```

*Status update of the user*

NAME	CLUSTER	AUTHENTICATION	AUTHORIZATION	READY
my-user-2	my-cluster	tls	simple	True

Fetching the details shows no messages.

```
kubectl get kafkausers my-user-2 -o yaml
```

*Details on a user with a READY status*

```
# ...
status:
```

```
conditions:
- lastTransitionTime: "2022-06-10T10:33:40.166846Z"
  status: "True"
  type: Ready
```

## 13.2. Ignoring specific users

In some scenarios, you might want to exclude certain users from being managed by the User Operator. This is useful for the following types of users:

- Special built-in users (for example, `ANONYMOUS` or `*`)
- Internal users managed by other mechanisms

When you exclude users from being managed, the User Operator ignores their:

- ACLs
- Quotas
- SCRAM-SHA credentials

To define which users should be ignored, set the `STRIMZI_IGNORED_USERS_PATTERN` environment variable with a regular expression. For example, to ignore `ANONYMOUS` or `*` users, set it to `^\*|ANONYMOUS$`. You can set the environment variable in your `Kafka` custom resource (`.spec.template.entityOperator.userOperatorContainer.env`) or in the `Deployment` resource of your standalone User Operator.

*Example Kafka configuration excluding users*

```
apiVersion: kafka.strimzi.io/v1
kind: Kafka
metadata:
  # ...
spec:
  #...
  entityOperator:
    template:
      userOperatorContainer:
        env:
          - name: STRIMZI_IGNORED_USERS_PATTERN
            value: "\\\*|ANONYMOUS$" # Double \ needed for escaping
#...
```

*Example standalone Deployment configuration excluding users*

```
apiVersion: apps/v1
kind: Deployment
metadata:
  # ...
spec:
```

```
# ...
template:
  spec:
    containers:
      - name: strimzi-user-operator
        # ...
      env:
        # ...
        - name: STRIMZI_IGNORED_USERS_PATTERN
          value: "^\\*|ANONYMOUS$" # Double \ needed for escaping
```

# Chapter 14. Using the Access Operator to manage client connections

The Strimzi Access Operator is an optional feature that simplifies the sharing of Kafka connection information and credentials between namespaces. The Access Operator is deployed separately and manages the distribution of connection details, which are stored as a `Secret` resource. The secret contains cluster connection details and optional user credentials, and can be referenced by applications across different namespaces.

The `KafkaAccess` resource configures Kafka cluster connection information and credentials. When you create, modify, or delete connection information using a `KafkaAccess` resource, the Access Operator updates a service binding secret. This secret stores the details in multiple formats, allowing applications across different namespaces to connect to the clusters.

## 14.1. Deploying the Access Operator

This procedure shows how to deploy the Access Operator for access management of a Kafka cluster. You cannot use the Access Operator with a Kafka cluster that is not managed by the Cluster Operator.

### Prerequisites

- The Access Operator installation files must be downloaded and extracted from the latest release archive (`strimzi-0.50.0.*`) from the [GitHub releases page](#).  
The installation files are contained in the `./install` directory.
- [The Cluster Operator must be deployed](#).

By default, the Access Operator is deployed in the `strimzi-access-operator` namespace. To deploy the operator in a different namespace, update the `namespace` property in the installation files.

### Procedure

1. Deploy the Access Operator using the installation files from the `install` directory:

```
kubectl create -f ./install/access-operator
```

2. Check the status of the deployment:

```
kubectl get deployments
```

*Output shows the deployment name and readiness*

NAME	READY	UP-TO-DATE	AVAILABLE
strimzi-access-operator	1/1	1	1

`READY` shows the number of replicas that are ready/expected. The deployment is successful when

the **AVAILABLE** output shows 1.

## 14.2. Using the Access Operator

This procedure shows how to use the Access Operator to manage access to a Kafka cluster. The Access Operator simplifies access management by creating a binding **Secret** containing the connection details for a Kafka cluster and optional user credentials.

Kafka instances and Kafka users are specified in a **KafkaAccess** resource. Optionally, a listener to connect to the Kafka cluster can also be specified. The operator will look up the Kafka instance and users specified in the **KafkaAccess** resource to create a secret with the details required for connection. Any authentication credentials required by a user to connect to Kafka are also added to the secret. If no listener is specified it will choose according to these rules:

- If only one listener is configured in the **Kafka** resource, that listener is selected.
- If multiple listeners are configured, the operator selects one with the appropriate security based on **tls** and **authentication** properties in the **Kafka** and **KafkaUser** resources.
- If multiple listeners with appropriate security are available, the operator selects an internal listener, if present.
- If multiple internal listeners with appropriate security are available, the operator selects the first listener alphabetically by name.

### Prerequisites

- A Kafka cluster deployed using a **Kafka** resource and managed by the Cluster Operator.
- **KafkaUser** resources must be created for any users needing authentication to the Kafka cluster.

Example **KafkaAccess** resource specifications are contained in the `./examples` directory of the latest release archive (`strimzi-0.50.0.*`) from the [GitHub releases page](#).

### Procedure

1. Configure a **KafkaAccess** custom resource to bind your applications to the Kafka cluster.
  - Specify the name of the **Kafka** resource to connect to the Kafka cluster.
  - (Optional) Specify the name of the listener used to connect to the Kafka cluster.
  - (Optional) Specify the **KafkaUser** resources representing users requiring access to the Kafka cluster.

Make sure the name, namespace and listener in the **KafkaAccess** custom resource match those of your Kafka instance.

### Example access configuration

```
apiVersion: access.strimzi.io/v1alpha1
kind: KafkaAccess
metadata:
  name: my-kafka-access
spec:
```

```
kafka:  
  name: my-cluster  
  namespace: kafka  
  listener: tls  
  user:  
    kind: KafkaUser  
    apiGroup: kafka.strimzi.io  
    name: my-user  
    namespace: kafka
```

2. Create the [KafkaAccess](#) resource in Kubernetes:

```
kubectl apply -f <kafka_access_config_file> -n strimzi-access-operator
```

3. Check the status of the [KafkaAccess](#) custom resource:

```
kubectl get kafkaaccess my-kafka-access -o yaml
```

The status of the resource is updated once the operator creates the binding [Secret](#).

*Status shows binding secret created*

```
# ...  
status:  
  binding:  
    name:  
      kafka-binding
```

4. Inspect the created binding secret:

```
kubectl get secret kafka-binding -o yaml
```

The secret contains connection details and user credentials in various formats to accommodate the requirements of different development environments and applications. The user credentials included depend on the authentication method specified for the [KafkaUser](#) resource. For example, SCRAM-SHA-512 credentials are provided if that type of authentication is used.

*Example binding secret*

```
apiVersion: v1  
kind: Secret  
metadata:  
  name: kafka-binding  
  type: servicebinding.io/kafka  
data:  
  type: kafka
```

```

provider: strimzi
# Kafka cluster connection properties ①
bootstrap.servers: my-cluster-kafka-bootstrap:9092
bootstrap-servers: my-cluster-kafka-bootstrap:9092
bootstrapServers: my-cluster-kafka-bootstrap:9092
# Security protocol option ②
security.protocol: SSL
securityProtocol: SSL
# TLS enabled
ssl.truststore.crt: my-cluster-cluster-ca-cert ③
# SCRAM-SHA-512 authentication properties ④
username: admin
password: password
sasl.jaas.config: com.example.ScramLoginModule required;
sasl.mechanism: SCRAM-SHA-512
saslMechanism: SCRAM-SHA-512
# mTLS authentication properties ⑤
ssl.keystore.crt: user.crt
ssl.keystore.key: user.key

```

① Comma-separated host:port connection details for Kafka clusters.

② The security protocol used for communication with Kafka, which can be `PLAINTEXT`, `SSL`, `SASL_PLAINTEXT` or `SASL_SSL`.

③ The Strimzi cluster CA certificate if TLS is enabled.

④ Credentials for SCRAM-SHA-512 client authentication.

⑤ Credentials for mTLS client authentication.

## 5. Make the secret available to your application.

Applications can reference the secret directly, or you can inject it into an application's environment using an operator to implement the [Service Binding specification](#).

Optionally, you can specify the name of the secret by adding the `secretName` field to the `KafkaAccess` resource specification:

```

apiVersion: access.strimzi.io/v1alpha1
kind: KafkaAccess
metadata:
  name: my-kafka-access
spec:
  kafka:
    name: my-cluster
    namespace: kafka
    listener: tls
    secretName: my-custom-secret

```

If no `secretName` is provided, the secret name defaults to the name of the `KafkaAccess` resource.

# Chapter 15. Setting up client access to a Kafka cluster

After you have [deployed Strimzi](#), you can set up client access to your Kafka cluster. To verify the deployment, you can deploy example producer and consumer clients. Otherwise, create listeners that provide client access within or outside the Kubernetes cluster.

## 15.1. Deploying example clients

Send and receive messages from a Kafka cluster installed on Kubernetes.

This procedure describes how to deploy Kafka clients to the Kubernetes cluster, then produce and consume messages to test your installation. The clients are deployed using the Kafka container image.

### *Prerequisites*

- The Kafka cluster is available for the clients.

### *Procedure*

1. Deploy a Kafka producer.

This example deploys a Kafka producer that connects to the Kafka cluster `my-cluster`.

A topic named `my-topic` is created.

#### *Deploying a Kafka producer to Kubernetes*

```
kubectl run kafka-producer -ti --image=quay.io/stimzi/kafka:0.50.0-kafka-4.1.1  
--rm=true --restart=Never -- bin/kafka-console-producer.sh --bootstrap-server my-  
cluster-kafka-bootstrap:9092 --topic my-topic
```

2. Type a message into the console where the producer is running.
3. Press **Enter** to send the message.
4. Deploy a Kafka consumer.

The consumer should consume messages produced to `my-topic` in the Kafka cluster `my-cluster`.

#### *Deploying a Kafka consumer to Kubernetes*

```
kubectl run kafka-consumer -ti --image=quay.io/stimzi/kafka:0.50.0-kafka-4.1.1  
--rm=true --restart=Never -- bin/kafka-console-consumer.sh --bootstrap-server my-  
cluster-kafka-bootstrap:9092 --topic my-topic --from-beginning
```

5. Confirm that you see the incoming messages in the consumer console.

## 15.2. Configuring listeners to connect to Kafka

Use listeners to enable client connections to Kafka. Strimzi provides a generic [GenericKafkaListener](#) schema with properties to configure listeners through the [Kafka](#) resource.

When configuring a Kafka cluster, you specify a listener [type](#) based on your requirements, environment, and infrastructure. Services, routes, load balancers, and ingresses for clients to connect to a cluster are created according to the listener type.

Internal and external listener types are supported.

### Internal listeners

Use internal listener types to connect clients within a Kubernetes cluster.

- [internal](#) to connect within the same Kubernetes cluster
- [cluster-ip](#) to expose Kafka using per-broker [ClusterIP](#) services

Internal listeners use a headless service and the DNS names assigned to the broker pods. By default, they do not use the Kubernetes service DNS domain (typically [.cluster.local](#)). However, you can customize this configuration using the [useServiceDnsDomain](#) property. Consider using a [cluster-ip](#) type listener if routing through the headless service isn't feasible or if you require a custom access mechanism, such as when integrating with specific Ingress controllers or the Kubernetes Gateway API.

### External listeners

Use external listener types to connect clients outside a Kubernetes cluster.

- [nodeport](#) to use ports on Kubernetes nodes
- [loadbalancer](#) to use loadbalancer services
- [ingress](#) to use Kubernetes [Ingress](#) and the [Ingress NGINX Controller for Kubernetes](#) (Kubernetes only)
- [route](#) to use OpenShift [Route](#) and the default HAProxy router (OpenShift only)

External listeners handle access to a Kafka cluster from networks that require different authentication mechanisms. For example, loadbalancers might not be suitable for certain infrastructure, such as bare metal, where node ports provide a better option.

**IMPORTANT**

Do not use the built-in [ingress](#) controller on OpenShift, use the [route](#) type instead. The Ingress NGINX Controller is only intended for use on Kubernetes. The [route](#) type is only supported on OpenShift.

Each listener is defined as an array in the [Kafka](#) resource.

#### *Example listener configuration*

```
apiVersion: kafka.strimzi.io/v1
kind: Kafka
metadata:
```

```

name: my-cluster
spec:
  kafka:
    # ...
    listeners:
      - name: plain
        port: 9092
        type: internal
        tls: false
        configuration:
          useServiceDnsDomain: true
      - name: tls
        port: 9093
        type: internal
        tls: true
        authentication:
          type: tls
      - name: external1
        port: 9094
        type: route
        tls: true
        configuration:
          brokerCertChainAndKey:
            secretName: my-secret
            certificate: my-certificate.crt
            key: my-key.key
    # ...

```

You can configure as many listeners as required, as long as their names and ports are unique. You can also configure listeners for secure connection using authentication.

**NOTE**

If you scale your Kafka cluster while using external listeners, it might trigger a rolling update of all Kafka brokers. This depends on the configuration.

*Additional resources*

- [GenericKafkaListener schema reference](#)

## 15.3. Listener naming conventions

From the listener configuration, the resulting listener bootstrap and per-broker service names are structured according to the following naming conventions:

*Table 14. Listener naming conventions*

Listener type	Bootstrap service name	Per-Broker service name
internal	<cluster_name>-kafka-bootstrap	<i>Not applicable</i>

Listener type	Bootstrap service name	Per-Broker service name
<code>loadbalancer</code> <code>nodeport</code> <code>ingress</code> <code>route</code> <code>cluster-ip</code>	<cluster_name>-kafka-<listener-name>-bootstrap	<cluster_name>-kafka-<listener-name>-<idx>

For example, `my-cluster-kafka-bootstrap`, `my-cluster-kafka-external1-bootstrap`, and `my-cluster-kafka-external1-0`. The names are assigned to the services, routes, load balancers, and ingresses created through the listener configuration.

You can use certain backwards compatible names and port numbers to transition listeners initially configured under the retired `KafkaListeners` schema. The resulting external listener naming convention varies slightly. The specific combinations of listener name and port configuration values in the following table are backwards compatible.

*Table 15. Backwards compatible listener name and port combinations*

Listener name	Port	Bootstrap service name	Per-Broker service name
<code>plain</code>	<code>9092</code>	<cluster_name>-kafka-bootstrap	<i>Not applicable</i>
<code>tls</code>	<code>9093</code>	<cluster-name>-kafka-bootstrap	<i>Not applicable</i>
<code>external</code>	<code>9094</code>	<cluster_name>-kafka-bootstrap	<cluster_name>-kafka-bootstrap-<idx>

## 15.4. Accessing Kafka using node ports

Use node ports to access a Kafka cluster from an external client outside the Kubernetes cluster.

To connect to a broker, you specify a hostname and port number for the Kafka bootstrap address, as well as the certificate used for TLS encryption.

The procedure shows basic `nodeport` listener configuration. You can use listener properties to enable TLS encryption (`tls`) and specify a client authentication mechanism (`authentication`). Add additional configuration using `configuration` properties. For example, you can use the following configuration properties with `nodeport` listeners:

### `preferredNodePortAddressType`

Specifies the first address type that's checked as the node address.

### `externalTrafficPolicy`

Specifies whether the service routes external traffic to node-local or cluster-wide endpoints.

### `nodePort`

Overrides the assigned node port numbers for the bootstrap and broker services.

For more information on listener configuration, see the [GenericKafkaListener schema reference](#).

### *Prerequisites*

- A running Cluster Operator

In this procedure, the Kafka cluster name is `my-cluster`. The name of the listener is `external4`.

#### *Procedure*

1. Configure a `Kafka` resource with an external listener set to the `nodeport` type.

For example:

```
apiVersion: kafka.strimzi.io/v1
kind: Kafka
metadata:
  labels:
    app: my-cluster
    name: my-cluster
    namespace: myproject
spec:
  kafka:
    # ...
    listeners:
      - name: external4
        port: 9094
        type: nodeport
        tls: true
        authentication:
          type: tls
        # ...
    # ...
```

2. Create or update the resource.

```
kubectl apply -f <kafka_configuration_file>
```

A cluster CA certificate to verify the identity of the kafka brokers is created in the secret `my-cluster-cluster-ca-cert`.

`NodePort` type services are created for each Kafka broker, as well as an external bootstrap service.

*Node port services created for the bootstrap and brokers*

NAME	TYPE	CLUSTER-IP	PORT(S)
my-cluster-kafka-external4-0	NodePort	172.30.55.13	9094:31789/TCP
my-cluster-kafka-external4-1	NodePort	172.30.250.248	9094:30028/TCP
my-cluster-kafka-external4-2	NodePort	172.30.115.81	9094:32650/TCP
my-cluster-kafka-external4-bootstrap	NodePort	172.30.30.23	9094:32650/TCP

The bootstrap address used for client connection is propagated to the `status` of the `Kafka`

resource.

*Example status for the bootstrap address*

```
status:
  clusterId: Y_RJQDGKRXmNF7fEcWldJQ
  conditions:
    - lastTransitionTime: '2023-01-31T14:59:37.113630Z'
      status: 'True'
      type: Ready
  kafkaVersion: 4.1.1
  listeners:
    # ...
    - addresses:
        - host: ip-10-0-224-199.us-west-2.compute.internal
          port: 32650
  bootstrapServers: 'ip-10-0-224-199.us-west-2.compute.internal:32650'
  certificates:
    - |
      -----BEGIN CERTIFICATE-----
      -----END CERTIFICATE-----
    name: external4
  observedGeneration: 2
  operatorLastSuccessfulVersion: 0.50.0
# ...
```

3. Retrieve the bootstrap address you can use to access the Kafka cluster from the status of the **Kafka** resource.

```
kubectl get kafka my-cluster
-o=jsonpath='{.status.listeners[?(@.name=="external4")].bootstrapServers}{"\n"}'
ip-10-0-224-199.us-west-2.compute.internal:32650
```

4. Extract the cluster CA certificate.

```
kubectl get secret my-cluster-cluster-ca-cert -o jsonpath='{.data.ca\.crt}' |
base64 -d > ca.crt
```

5. Configure your client to connect to the brokers.

- Specify the bootstrap host and port in your Kafka client as the bootstrap address to connect to the Kafka cluster. For example, **ip-10-0-224-199.us-west-2.compute.internal:32650**.
- Add the extracted certificate to the truststore of your Kafka client to configure a TLS connection.

If you enabled a client authentication mechanism, you will also need to configure it in your

client.

**NOTE** If you are using your own listener certificates, check whether you need to add the CA certificate to the client's truststore configuration. If it is a public (external) CA, you usually won't need to add it.

## 15.5. Accessing Kafka using loadbalancers

Use loadbalancers to access a Kafka cluster from an external client outside the Kubernetes cluster.

To connect to a broker, you specify a hostname and port number for the Kafka bootstrap address, as well as the certificate used for TLS encryption.

The procedure shows basic `loadbalancer` listener configuration. You can use listener properties to enable TLS encryption (`tls`) and specify a client authentication mechanism (`authentication`). Add additional configuration using `configuration` properties. For example, you can use the following configuration properties with `loadbalancer` listeners:

### `loadBalancerSourceRanges`

Restricts traffic to a specified list of CIDR (Classless Inter-Domain Routing) ranges.

### `externalTrafficPolicy`

Specifies whether the service routes external traffic to node-local or cluster-wide endpoints.

### `loadBalancerIP`

Requests a specific IP address when creating a loadbalancer.

For more information on listener configuration, see the [GenericKafkaListener schema reference](#).

#### *Prerequisites*

- A running Cluster Operator

In this procedure, the Kafka cluster name is `my-cluster`. The name of the listener is `external3`.

#### *Procedure*

1. Configure a `Kafka` resource with an external listener set to the `loadbalancer` type.

For example:

```
apiVersion: kafka.strimzi.io/v1
kind: Kafka
metadata:
  labels:
    app: my-cluster
    name: my-cluster
    namespace: myproject
spec:
  kafka:
    # ...
```

```

listeners:
  - name: external3
    port: 9094
    type: loadbalancer
    tls: true
    authentication:
      type: tls
    # ...
# ...

```

2. Create or update the resource.

```
kubectl apply -f <kafka_configuration_file>
```

A cluster CA certificate to verify the identity of the kafka brokers is also created in the secret **my-cluster-cluster-ca-cert**.

**loadbalancer** type services and loadbalancers are created for each Kafka broker, as well as an external bootstrap service.

*Loadbalancer services and loadbalancers created for the bootstraps and brokers*

NAME	TYPE	CLUSTER-IP	PORT(S)
my-cluster-kafka-external3-0 9094:30011/TCP	LoadBalancer	172.30.204.234	
my-cluster-kafka-external3-1 9094:32544/TCP	LoadBalancer	172.30.164.89	
my-cluster-kafka-external3-2 9094:32504/TCP	LoadBalancer	172.30.73.151	
my-cluster-kafka-external3-bootstrap 9094:30371/TCP	LoadBalancer	172.30.30.228	

NAME	EXTERNAL-IP (loadbalancer)
my-cluster-kafka-external3-0 1132975133.us-west-2.elb.amazonaws.com	a8a519e464b924000b6c0f0a05e19f0d-
my-cluster-kafka-external3-1 611832211.us-west-2.elb.amazonaws.com	ab6adc22b556343afb0db5ea05d07347-
my-cluster-kafka-external3-2 777597560.us-west-2.elb.amazonaws.com	a9173e8ccb1914778aeb17eca98713c0-
my-cluster-kafka-external3-bootstrap west-2.elb.amazonaws.com	a8d4a6fb363bf447fb6e475fc3040176-36312313.us-

The bootstrap address used for client connection is propagated to the **status** of the **Kafka** resource.

*Example status for the bootstrap address*

```

status:
  clusterId: Y_RJQDGKRXmNF7fEcWldJQ

```

```

conditions:
  - lastTransitionTime: '2023-01-31T14:59:37.113630Z'
    status: 'True'
    type: Ready
kafkaVersion: 4.1.1
listeners:
# ...
  - addresses:
    - host: >-
      a8d4a6fb363bf447fb6e475fc3040176-36312313.us-west-2.elb.amazonaws.com
      port: 9094
bootstrapServers: >-
  a8d4a6fb363bf447fb6e475fc3040176-36312313.us-west-2.elb.amazonaws.com:9094
certificates:
  - |
-----BEGIN CERTIFICATE-----

-----END CERTIFICATE-----
  name: external3
  observedGeneration: 2
  operatorLastSuccessfulVersion: 0.50.0
# ...

```

The DNS addresses used for client connection are propagated to the **status** of each loadbalancer service.

*Example status for the bootstrap loadbalancer*

```

status:
  loadBalancer:
    ingress:
      - hostname: >-
        a8d4a6fb363bf447fb6e475fc3040176-36312313.us-west-2.elb.amazonaws.com
# ...

```

3. Retrieve the bootstrap address you can use to access the Kafka cluster from the status of the **Kafka** resource.

```

kubectl get kafka my-cluster
-o=jsonpath='{.status.listeners[?(@.name=="external3")].bootstrapServers}{"\n"}'
a8d4a6fb363bf447fb6e475fc3040176-36312313.us-west-2.elb.amazonaws.com:9094

```

4. Extract the cluster CA certificate.

```

kubectl get secret my-cluster-cluster-ca-cert -o jsonpath='{.data.ca\.crt}' |
base64 -d > ca.crt

```

5. Configure your client to connect to the brokers.
  - a. Specify the bootstrap host and port in your Kafka client as the bootstrap address to connect to the Kafka cluster. For example, `a8d4a6fb363bf447fb6e475fc3040176-36312313.us-west-2.elb.amazonaws.com:9094`.
  - b. Add the extracted certificate to the truststore of your Kafka client to configure a TLS connection.

If you enabled a client authentication mechanism, you will also need to configure it in your client.

**NOTE** If you are using your own listener certificates, check whether you need to add the CA certificate to the client's truststore configuration. If it is a public (external) CA, you usually won't need to add it.

## 15.6. Accessing Kafka using an Ingress NGINX Controller for Kubernetes

Use an [Ingress NGINX Controller for Kubernetes](#) to access a Kafka cluster from clients outside the Kubernetes cluster.

To be able to use an Ingress NGINX Controller for Kubernetes, add configuration for an `ingress` type listener in the `Kafka` custom resource. When applied, the configuration creates a dedicated ingress and service for an external bootstrap and each broker in the cluster. Clients connect to the bootstrap ingress, which routes them through the bootstrap service to connect to a broker. Per-broker connections are then established using DNS names, which route traffic from the client to the broker through the broker-specific ingresses and services.

To connect to a broker, you specify a hostname for the ingress bootstrap address, as well as the certificate used for TLS encryption. For access using an ingress, the port used in the Kafka client is typically 443.

The procedure shows basic `ingress` listener configuration. TLS encryption (`tls`) must be enabled. You can also specify a client authentication mechanism (`authentication`). Add additional configuration using `configuration` properties. For example, you can use the `class` configuration property with `ingress` listeners to specify the ingress controller used.

For more information on listener configuration, see the [GenericKafkaListener schema reference](#).

### *TLS passthrough*

Make sure that you enable TLS passthrough in your Ingress NGINX Controller for Kubernetes deployment. Kafka uses a binary protocol over TCP, but the Ingress NGINX Controller for Kubernetes is designed to work with a HTTP protocol. To be able to route TCP traffic through ingresses, Strimzi uses TLS passthrough with Server Name Indication (SNI).

SNI helps with identifying and passing connection to Kafka brokers. In passthrough mode, TLS encryption is always used. Because the connection passes to the brokers, the listeners use the TLS certificates signed by the internal cluster CA and not the ingress certificates. To configure listeners

to use your own listener certificates, [use the `brokerCertChainAndKey` property](#).

For more information about enabling TLS passthrough, see the [TLS passthrough documentation](#).

#### Prerequisites

- An Ingress NGINX Controller for Kubernetes is running with TLS passthrough enabled
- A running Cluster Operator

In this procedure, the Kafka cluster name is `my-cluster`. The name of the listener is `external2`.

#### Procedure

1. Configure a `Kafka` resource with an external listener set to the `ingress` type.

Specify an ingress hostname for the bootstrap service and for the Kafka brokers in the Kafka cluster.

For example:

```
apiVersion: kafka.strimzi.io/v1
kind: Kafka
metadata:
  labels:
    app: my-cluster
    name: my-cluster
    namespace: myproject
spec:
  kafka:
    # ...
    listeners:
      - name: external2
        port: 9094
        type: ingress
        tls: true ①
        authentication:
          type: tls
        configuration:
          class: nginx ②
          hostTemplate: broker-{nodeId}.myingress.com ③
        bootstrap:
          host: bootstrap.myingress.com ④
    # ...
```

① For `ingress` type listeners, TLS encryption must be enabled (`true`).

② (Optional) Class that specifies the ingress controller to use. You might need to add a class if you have not set up a default and a class name is missing in the ingresses created.

③ The host template used to generate the hostnames for the per-broker Ingress resources.

④ The host used as the hostnames for the bootstrap Ingress resource.

2. Create or update the resource.

```
kubectl apply -f <kafka_configuration_file>
```

A cluster CA certificate to verify the identity of the kafka brokers is created in the secret **my-cluster-cluster-ca-cert**.

**ClusterIP** type services are created for each Kafka broker, as well as an external bootstrap service.

An **ingress** is also created for each service, with a DNS address to expose them using the Ingress NGINX Controller for Kubernetes.

*Ingresses created for the bootstrap and brokers*

NAME	CLASS	HOSTS	ADDRESS
PORTS			
my-cluster-kafka-external2-0 80,443	nginx	broker-0.myingress.com	192.168.49.2
my-cluster-kafka-external2-1 80,443	nginx	broker-1.myingress.com	192.168.49.2
my-cluster-kafka-external2-2 80,443	nginx	broker-2.myingress.com	192.168.49.2
my-cluster-kafka-external2-bootstrap 80,443	nginx	bootstrap.myingress.com	192.168.49.2

The DNS addresses used for client connection are propagated to the **status** of each ingress.

*Status for the bootstrap ingress*

```
status:  
  loadBalancer:  
    ingress:  
      - ip: 192.168.49.2  
    # ...
```

3. Use a target broker to check the client-server TLS connection on port 443 using the OpenSSL **s\_client**.

```
openssl s_client -connect broker-0.myingress.com:443 -servername broker-0.myingress.com -showcerts
```

The server name is the SNI for passing the connection to the broker.

If the connection is successful, the certificates for the broker are returned.

**Certificate chain**

```
0 s:0 = io.strimzi, CN = my-cluster-kafka  
i:0 = io.strimzi, CN = cluster-ca v0
```

4. Extract the cluster CA certificate.

```
kubectl get secret my-cluster-cluster-ca-cert -o jsonpath='{.data.ca\.crt}' |  
base64 -d > ca.crt
```

5. Configure your client to connect to the brokers.

- a. Specify the bootstrap host (from the listener [configuration](#)) and port 443 in your Kafka client as the bootstrap address to connect to the Kafka cluster. For example, [bootstrap.myingress.com:443](#).
- b. Add the extracted certificate to the truststore of your Kafka client to configure a TLS connection.

If you enabled a client authentication mechanism, you will also need to configure it in your client.

**NOTE** If you are using your own listener certificates, check whether you need to add the CA certificate to the client's truststore configuration. If it is a public (external) CA, you usually won't need to add it.

## 15.7. Accessing Kafka using OpenShift routes

Use OpenShift routes to access a Kafka cluster from clients outside the OpenShift cluster.

To be able to use routes, add configuration for a [route](#) type listener in the [Kafka](#) custom resource. When applied, the configuration creates a dedicated route and service for an external bootstrap and each broker in the cluster. Clients connect to the bootstrap route, which routes them through the bootstrap service to connect to a broker. Per-broker connections are then established using DNS names, which route traffic from the client to the broker through the broker-specific routes and services.

To connect to a broker, you specify a hostname for the route bootstrap address, as well as the certificate used for TLS encryption. For access using routes, the port is always 443.

**WARNING**

An OpenShift route address comprises the Kafka cluster name, the listener name, the project name, and the domain of the router. For example, [my-cluster-kafka-external1-bootstrap-my-project.domain.com](#) (<cluster\_name>-kafka-<listener\_name>-bootstrap-<namespace>.domain). Each DNS label (between periods ".") must not exceed 63 characters, and the total length of the address must not exceed 255 characters.

The procedure shows basic listener configuration. TLS encryption (`tls`) must be enabled. You can also specify a client authentication mechanism (`authentication`). Add additional configuration using `configuration` properties. For example, you can use the `host` configuration property with `route` listeners to specify the hostnames used by the bootstrap and per-broker services.

For more information on listener configuration, see the [GenericKafkaListener schema reference](#).

#### *TLS passthrough*

TLS passthrough is enabled for routes created by Strimzi. Kafka uses a binary protocol over TCP, but routes are designed to work with a HTTP protocol. To be able to route TCP traffic through routes, Strimzi uses TLS passthrough with Server Name Indication (SNI).

SNI helps with identifying and passing connection to Kafka brokers. In passthrough mode, TLS encryption is always used. Because the connection passes to the brokers, the listeners use TLS certificates signed by the internal cluster CA and not the ingress certificates. To configure listeners to use your own listener certificates, [use the `brokerCertChainAndKey` property](#).

#### *Prerequisites*

- A running Cluster Operator

In this procedure, the Kafka cluster name is `my-cluster`. The name of the listener is `external1`.

#### *Procedure*

1. Configure a `Kafka` resource with an external listener set to the `route` type.

For example:

```
apiVersion: kafka.strimzi.io/v1
kind: Kafka
metadata:
  labels:
    app: my-cluster
    name: my-cluster
    namespace: myproject
spec:
  kafka:
    # ...
    listeners:
      - name: external1
        port: 9094
        type: route
        tls: true ①
        authentication:
          type: tls
        # ...
    # ...
```

① For `route` type listeners, TLS encryption must be enabled (`true`).

2. Create or update the resource.

```
kubectl apply -f <kafka_configuration_file>
```

A cluster CA certificate to verify the identity of the kafka brokers is created in the secret **my-cluster-cluster-ca-cert**.

**ClusterIP** type services are created for each Kafka broker, as well as an external bootstrap service.

A **route** is also created for each service, with a DNS address (host/port) to expose them using the default OpenShift HAProxy router.

The routes are preconfigured with TLS passthrough.

#### *Routes created for the bootstraps and brokers*

NAME	HOST/PORT
SERVICES	PORT TERMINATION
my-cluster-kafka-external1-0	my-cluster-kafka-external1-0-my-project.router.com 9094 passthrough
my-cluster-kafka-external1-1	my-cluster-kafka-external1-1-my-project.router.com 9094 passthrough
my-cluster-kafka-external1-2	my-cluster-kafka-external1-2-my-project.router.com 9094 passthrough
my-cluster-kafka-external1-bootstrap	my-cluster-kafka-external1-bootstrap-my-project.router.com 9094 passthrough

The DNS addresses used for client connection are propagated to the **status** of each route.

#### *Example status for the bootstrap route*

```
status:  
  ingress:  
    - host: >-  
      my-cluster-kafka-external1-bootstrap-my-project.router.com  
    # ...
```

3. Use a target broker to check the client-server TLS connection on port 443 using the OpenSSL **s\_client**.

```
openssl s_client -connect my-cluster-kafka-external1-0-my-project.router.com:443  
-servername my-cluster-kafka-external1-0-my-project.router.com -showcerts
```

The server name is the Server Name Indication (SNI) for passing the connection to the broker.

If the connection is successful, the certificates for the broker are returned.

### Certificates for the broker

```
Certificate chain
0 s:0 = io.strimzi, CN = my-cluster-kafka
i:0 = io.strimzi, CN = cluster-ca v0
```

4. Retrieve the address of the bootstrap service from the status of the **Kafka** resource.

```
kubectl get kafka my-cluster
-o=jsonpath='{.status.listeners[?(@.name=="external1")].bootstrapServers}{"\n"}'
my-cluster-kafka-external1-bootstrap-my-project.router.com:443
```

The address comprises the Kafka cluster name, the listener name, the project name and the domain of the router (**router.com** in this example).

5. Extract the cluster CA certificate.

```
kubectl get secret my-cluster-cluster-ca-cert -o jsonpath='{.data.ca\.crt}' |
base64 -d > ca.crt
```

6. Configure your client to connect to the brokers.

- Specify the address for the bootstrap service and port 443 in your Kafka client as the bootstrap address to connect to the Kafka cluster.
- Add the extracted certificate to the truststore of your Kafka client to configure a TLS connection.

If you enabled a client authentication mechanism, you will also need to configure it in your client.

**NOTE** If you are using your own listener certificates, check whether you need to add the CA certificate to the client's truststore configuration. If it is a public (external) CA, you usually won't need to add it.

## 15.8. Discovering connection details for clients

Service discovery makes it easier for client applications running in the same Kubernetes cluster as Strimzi to interact with a Kafka cluster.

A service discovery label and annotation are created for the following services:

- Internal Kafka bootstrap service
- HTTP Bridge service

## Service discovery label

The service discovery label, `strimzi.io/discovery`, is set to `true` for `Service` resources to make them discoverable for client connections.

## Service discovery annotation

The service discovery annotation provides connection details in JSON format for each service for client applications to use to establish connections.

*Example internal Kafka bootstrap service*

```
apiVersion: v1
kind: Service
metadata:
  annotations:
    strimzi.io/discovery: |-  
      [ {  
        "port" : 9092,  
        "tls" : false,  
        "protocol" : "kafka",  
        "auth" : "scram-sha-512"  
      }, {  
        "port" : 9093,  
        "tls" : true,  
        "protocol" : "kafka",  
        "auth" : "tls"  
      } ]  
  labels:  
    strimzi.io/cluster: my-cluster  
    strimzi.io/discovery: "true"  
    strimzi.io/kind: Kafka  
    strimzi.io/name: my-cluster-kafka-bootstrap  
  name: my-cluster-kafka-bootstrap  
spec:  
  #...
```

*Example HTTP Bridge service*

```
apiVersion: v1
kind: Service
metadata:
  annotations:
    strimzi.io/discovery: |-  
      [ {  
        "port" : 8080,  
        "tls" : false,  
        "auth" : "none",  
        "protocol" : "http"  
      } ]  
  labels:  
    strimzi.io/cluster: my-bridge
```

```
strimzi.io/discovery: "true"
strimzi.io/kind: KafkaBridge
strimzi.io/name: my-bridge-bridge-service
```

Find services by specifying the discovery label when fetching services from the command line or a corresponding API call.

*Returning services using the discovery label*

```
kubectl get service -l strimzi.io/discovery=true
```

Connection details are returned when retrieving the service discovery label.

# Chapter 16. Securing access to a Kafka cluster

Secure connections by configuring Kafka and Kafka users. Through configuration, you can implement encryption, authentication, and authorization mechanisms.

## *Kafka configuration*

To establish secure access to Kafka, configure the [Kafka](#) resource to set up the following configurations based on your specific requirements:

- Listeners with specified authentication types to define how clients authenticate
  - TLS encryption for communication between Kafka and clients
  - Supported TLS versions and cipher suites for additional security
- Authorization for the entire Kafka cluster
- Network policies for restricting access
- Super users for unconstrained access to brokers

Authentication is configured independently for each listener, while authorization is set up for the whole Kafka cluster.

For more information on access configuration for Kafka, see the [Kafka schema reference](#) and [GenericKafkaListener schema reference](#).

## *User (client-side) configuration*

To enable secure client access to Kafka, configure [KafkaUser](#) resources. These resources represent clients and determine how they authenticate and authorize with the Kafka cluster.

Configure the [KafkaUser](#) resource to set up the following configurations based on your specific requirements:

- Authentication that must match the enabled listener authentication
  - Supported TLS versions and cipher suites that must match the Kafka configuration
- Simple authorization to apply Access Control List (ACL) rules
  - ACLs for fine-grained control over user access to topics and actions
- Quotas to limit client access based on byte rates or CPU utilization

The User Operator creates the user representing the client and the security credentials used for client authentication, based on the chosen authentication type.

For more information on access configuration for users, see the [KafkaUser schema reference](#).

## 16.1. Configuring client authentication on listeners

Configure client authentication for Kafka brokers when creating listeners. Specify the listener

authentication type using the `Kafka.spec.kafka.listeners.authentication` property in the `Kafka` resource.

For clients inside the Kubernetes cluster, you can create `plain` (without encryption) or `tls internal` listeners. The `internal` listener type use a headless service and the DNS names given to the broker pods. As an alternative to the headless service, you can also create a `cluster-ip` type of internal listener to expose Kafka using per-broker `ClusterIP` services. For clients outside the Kubernetes cluster, you create `external` listeners and specify a connection mechanism, which can be `nodeport`, `loadbalancer`, `ingress` (Kubernetes only), or `route` (OpenShift only).

For more information on the configuration options for connecting an external client, see [Setting up client access to a Kafka cluster](#).

Supported authentication options:

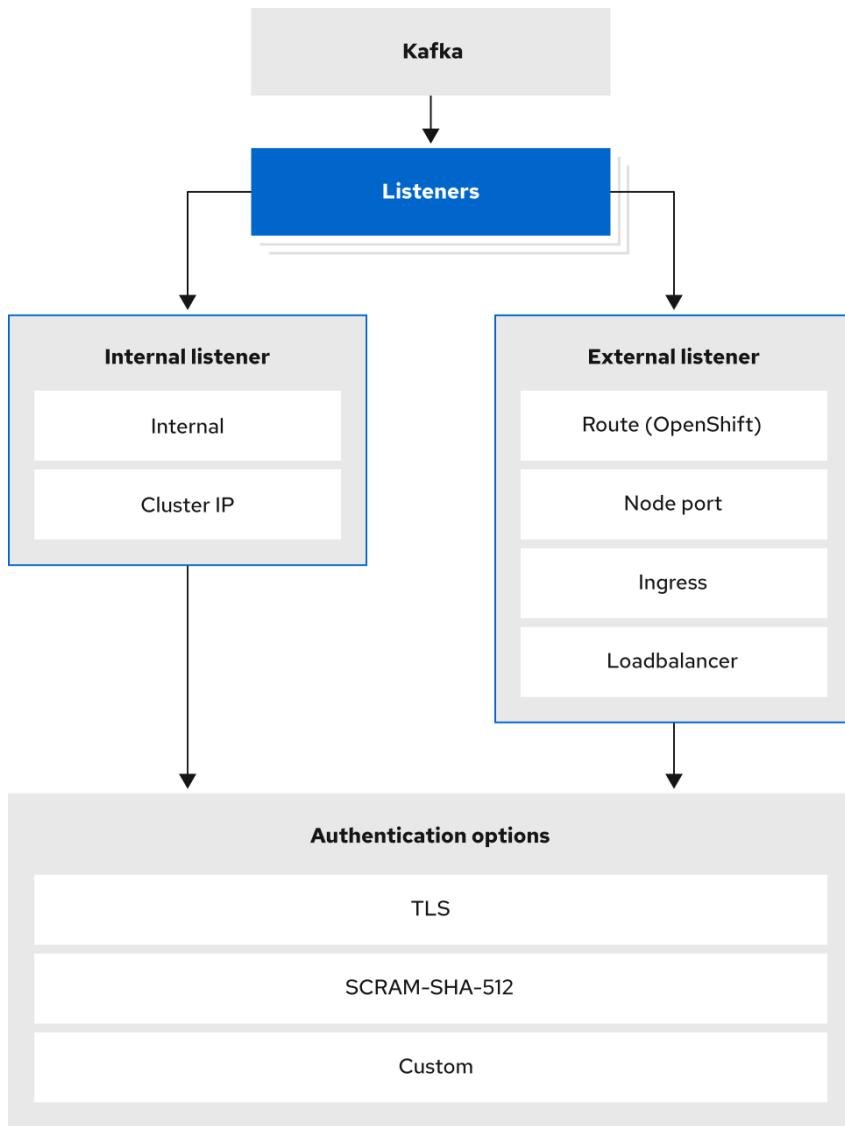
1. mTLS authentication (only on the listeners with TLS enabled encryption)
2. SCRAM-SHA-512 authentication
3. [OAuth 2.0 token-based authentication](#)
4. [Custom authentication](#)
5. [TLS versions and cipher suites](#)

Use custom authentication to configure token-based client access management, so that user authentication and authorization credentials are handled through an authorization server.

The authentication option you choose depends on how you wish to authenticate client access to Kafka brokers.

**NOTE**

Try exploring the standard authentication options before using custom authentication. Custom authentication allows for any type of Kafka-supported authentication. It can provide more flexibility, but also adds complexity.



117\_AMQ\_0920

*Figure 3. Kafka listener authentication options*

The listener `authentication` property is used to specify an authentication mechanism specific to that listener.

If no `authentication` property is specified then the listener does not authenticate clients which connect through that listener. The listener will accept all connections without authentication.

Authentication must be configured when using the User Operator to manage `KafkaUsers`.

The following example shows:

- A `plain` listener configured for SCRAM-SHA-512 authentication
- A `tls` listener with mTLS authentication
- An `external` listener with mTLS authentication

Each listener is configured with a unique name and port within a Kafka cluster.

#### IMPORTANT

When configuring listeners for client access to brokers, you can use port 9092 or higher (9093, 9094, and so on), but with a few exceptions. The

listeners cannot be configured to use the ports reserved for interbroker communication (9090 and 9091), Prometheus metrics (9404), and JMX (Java Management Extensions) monitoring (9999).

#### *Example listener authentication configuration*

```
apiVersion: kafka.strimzi.io/v1
kind: Kafka
metadata:
  name: my-cluster
  namespace: myproject
spec:
  kafka:
    # ...
    listeners:
      - name: plain
        port: 9092
        type: internal
        tls: true
        authentication:
          type: scram-sha-512
      - name: tls
        port: 9093
        type: internal
        tls: true
        authentication:
          type: tls
      - name: external3
        port: 9094
        type: loadbalancer
        tls: true
        authentication:
          type: tls
    # ...
```

### 16.1.1. mTLS authentication

mTLS authentication is always used for the communication between Kafka nodes.

Strimzi can configure Kafka to use TLS (Transport Layer Security) to provide encrypted communication between Kafka brokers and clients either with or without mutual authentication. For mutual, or two-way, authentication, both the server and the client present certificates. When you configure mTLS authentication, the broker authenticates the client (client authentication) and the client authenticates the broker (server authentication).

mTLS listener configuration in the [Kafka](#) resource requires the following:

- **tls: true** to specify TLS encryption and server authentication
- **authentication.type: tls** to specify the client authentication

When a Kafka cluster is created by the Cluster Operator, it creates a new secret with the name `<cluster_name>-cluster-ca-cert`. The secret contains a CA certificate. The CA certificate is in [PEM](#) and [PKCS #12 format](#). To verify a Kafka cluster, add the CA certificate to the truststore in your client configuration. To verify a client, add a user certificate and key to the keystore in your client configuration. For more information on configuring a client for mTLS, see [Configuring user authentication](#).

**NOTE** TLS authentication is more commonly one-way, with one party authenticating the identity of another. For example, when HTTPS is used between a web browser and a web server, the browser obtains proof of the identity of the web server.

### 16.1.2. SCRAM-SHA-512 authentication

SCRAM (Salted Challenge Response Authentication Mechanism) is an authentication protocol that can establish mutual authentication using passwords. Strimzi can configure Kafka to use SASL (Simple Authentication and Security Layer) SCRAM-SHA-512 to provide authentication on both unencrypted and encrypted client connections.

When SCRAM-SHA-512 authentication is used with a TLS connection, the TLS protocol provides the encryption, but is not used for authentication.

The following properties of SCRAM make it safe to use SCRAM-SHA-512 even on unencrypted connections:

- The passwords are not sent in the clear over the communication channel. Instead the client and the server are each challenged by the other to offer proof that they know the password of the authenticating user.
- The server and client each generate a new challenge for each authentication exchange. This means that the exchange is resilient against replay attacks.

When `KafkaUser.spec.authentication.type` is configured with `scram-sha-512` the User Operator will generate a random 32-character password consisting of upper and lowercase ASCII letters and numbers.

### 16.1.3. Restricting access to listeners with network policies

Control listener access by configuring the `networkPolicyPeers` property in the `Kafka` resource.

By default, Strimzi automatically creates a `NetworkPolicy` resource for every enabled Kafka listener, allowing connections from all namespaces.

To restrict listener access to specific applications or namespaces at the network level, configure the `networkPolicyPeers` property. Each listener can have its own `networkPolicyPeers` configuration. For more information on network policy peers, refer to the [NetworkPolicyPeer API reference](#).

If you want to use custom network policies, you can set the `STRIMZI_NETWORK_POLICY_GENERATION` environment variable to `false` in the Cluster Operator configuration. For more information, see [Configuring the Cluster Operator](#).

**NOTE**

Your configuration of Kubernetes must support ingress [NetworkPolicies](#) in order to use network policies.

*Prerequisites*

- A Kubernetes cluster with support for Ingress NetworkPolicies.
- The Cluster Operator is running.

*Procedure*

1. Configure the `networkPolicyPeers` property to define the application pods or namespaces allowed to access the Kafka cluster.

This example shows configuration for a `tls` listener to allow connections only from application pods with the label `app` set to `kafka-client`:

```
apiVersion: kafka.strimzi.io/v1
kind: Kafka
spec:
  kafka:
    # ...
    listeners:
      - name: tls
        port: 9093
        type: internal
        tls: true
        authentication:
          type: tls
      networkPolicyPeers:
        - podSelector:
            matchLabels:
              app: kafka-client
    # ...
```

2. Apply the changes to the [Kafka](#) resource configuration.

*Additional resources*

- [networkPolicyPeers configuration](#)
- [NetworkPolicyPeer API reference](#)

### 16.1.4. Using custom listener certificates for TLS encryption

This procedure shows how to configure custom server certificates for TLS listeners or external listeners which have TLS encryption enabled.

By default, Kafka listeners use certificates signed by Strimzi's internal CA (certificate authority). The Cluster Operator automatically generates a CA certificate when creating a Kafka cluster. To configure a client for TLS, the CA certificate is included in its truststore configuration to authenticate the Kafka cluster. Alternatively, you have the option to [install and use your own CA](#)

[certificates](#).

However, if you prefer more granular control by using your own custom certificates at the listener-level, you can configure listeners using `brokerCertChainAndKey` properties. You create a secret with your own private key and server certificate, then specify them in the `brokerCertChainAndKey` configuration.

User-provided certificates allow you to leverage existing security infrastructure. You can use a certificate signed by a public (external) CA or a private CA. Kafka clients need to trust the CA which was used to sign the listener certificate. If signed by a public CA, you usually won't need to add it to a client's truststore configuration.

Custom certificates are not managed by Strimzi, so you need to renew them manually.

**NOTE**

Listener certificates are used for TLS encryption and server authentication only. They are not used for TLS client authentication. If you want to use your own certificate for TLS client authentication as well, you must [install and use your own clients CA](#).

*Prerequisites*

- The Cluster Operator is running.
- Each listener requires the following:
  - A compatible server certificate signed by an external CA. (Provide an X.509 certificate in PEM format.)

You can use one listener certificate for multiple listeners.

- Subject Alternative Names (SANs) are specified in the certificate for each listener. For more information, see [Specifying SANs for custom listener certificates](#).

If you are not using a self-signed certificate, you can provide a certificate that includes the whole CA chain in the certificate.

You can only use the `brokerCertChainAndKey` properties if TLS encryption (`tls: true`) is configured for the listener.

**NOTE**

Strimzi does not support the use of encrypted private keys for TLS. The private key stored in the secret must be unencrypted for this to work.

*Procedure*

1. Create a `Secret` containing your private key and server certificate:

```
kubectl create secret generic <my_secret> --from-file=<my_listener_key.key> --from-file=<my_listener_certificate.crt>
```

2. Edit the `Kafka` resource for your cluster.

Configure the listener to use your [Secret](#), certificate file, and private key file in the `configuration.brokerCertChainAndKey` property.

*Example configuration for a `loadbalancer` external listener with TLS encryption enabled*

```
# ...
listeners:
  - name: plain
    port: 9092
    type: internal
    tls: false
  - name: external3
    port: 9094
    type: loadbalancer
    tls: true
  configuration:
    brokerCertChainAndKey:
      secretName: my-secret
      certificate: my-listener-certificate.crt
      key: my-listener-key.key
# ...
```

*Example configuration for a TLS listener*

```
# ...
listeners:
  - name: plain
    port: 9092
    type: internal
    tls: false
  - name: tls
    port: 9093
    type: internal
    tls: true
  configuration:
    brokerCertChainAndKey:
      secretName: my-secret
      certificate: my-listener-certificate.crt
      key: my-listener-key.key
# ...
```

### 3. Apply the changes to the [Kafka](#) resource configuration.

The Cluster Operator starts a rolling update of the Kafka cluster, which updates the configuration of the listeners.

**NOTE**

A rolling update is also started if you update a Kafka listener certificate in a [Secret](#) that is already used by a listener.

## 16.1.5. Specifying SANs for custom listener certificates

In order to use TLS hostname verification with custom [Kafka listener certificates](#), you must specify the correct Subject Alternative Names (SANs) for each listener.

The certificate SANs must specify hostnames for the following:

- All of the Kafka brokers in your cluster
- The Kafka cluster bootstrap service

You can use wildcard certificates if they are supported by your CA.

### Examples of SANs for internal listeners

Use the following examples to help you specify hostnames of the SANs in your certificates for your internal listeners.

Replace `<cluster-name>` with the name of the Kafka cluster and `<namespace>` with the Kubernetes namespace where the cluster is running.

*Wildcards example for a type: internal listener*

```
//Kafka brokers
*.<cluster_name>-kafka-brokers
*.<cluster_name>-kafka-brokers.<namespace>.svc

// Bootstrap service
<cluster_name>-kafka-bootstrap
<cluster_name>-kafka-bootstrap.<namespace>.svc
```

*Non-wildcards example for a type: internal listener*

```
// Kafka brokers
<cluster_name>-kafka-0.<cluster_name>-kafka-brokers
<cluster_name>-kafka-0.<cluster_name>-kafka-brokers.<namespace>.svc
<cluster_name>-kafka-1.<cluster_name>-kafka-brokers
<cluster_name>-kafka-1.<cluster_name>-kafka-brokers.<namespace>.svc
# ...

// Bootstrap service
<cluster_name>-kafka-bootstrap
<cluster_name>-kafka-bootstrap.<namespace>.svc
```

*Non-wildcards example for a type: cluster-ip listener*

```
// Kafka brokers
<cluster_name>-kafka-<listener-name>-0
<cluster_name>-kafka-<listener-name>-0.<namespace>.svc
<cluster_name>-kafka_-listener-name>-1
<cluster_name>-kafka-<listener-name>-1.<namespace>.svc
```

```
# ...

// Bootstrap service
<cluster_name>-kafka-<listener-name>-bootstrap
<cluster_name>-kafka-<listener-name>-bootstrap.<namespace>.svc
```

## Examples of SANs for external listeners

For external listeners which have TLS encryption enabled, the hostnames you need to specify in certificates depends on the external listener [type](#).

*Table 16. SANs for each type of external listener*

External listener type	In the SANs, specify...
<code>ingress</code>	Addresses of all Kafka broker <a href="#">Ingress</a> resources and the address of the bootstrap <a href="#">Ingress</a> .  You can use a matching wildcard name.
<code>route</code>	Addresses of all Kafka broker <a href="#">Routes</a> and the address of the bootstrap <a href="#">Route</a> .  You can use a matching wildcard name.
<code>loadbalancer</code>	Addresses of all Kafka broker <a href="#">loadbalancers</a> and the bootstrap <a href="#">loadbalancer</a> address.  You can use a matching wildcard name.
<code>nodeport</code>	Addresses of all Kubernetes worker nodes that the Kafka broker pods might be scheduled to.  You can use a matching wildcard name.

## 16.2. Configuring authorized access to Kafka

Configure authorized access to a Kafka cluster using the [Kafka.spec.kafka.authorization](#) property in the [Kafka](#) resource. If the [authorization](#) property is missing, no authorization is enabled and clients have no restrictions. When enabled, authorization is applied to all enabled listeners. The authorization method is defined in the [type](#) field.

Supported authorization options:

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

Use custom authorization to configure token-based authorization.

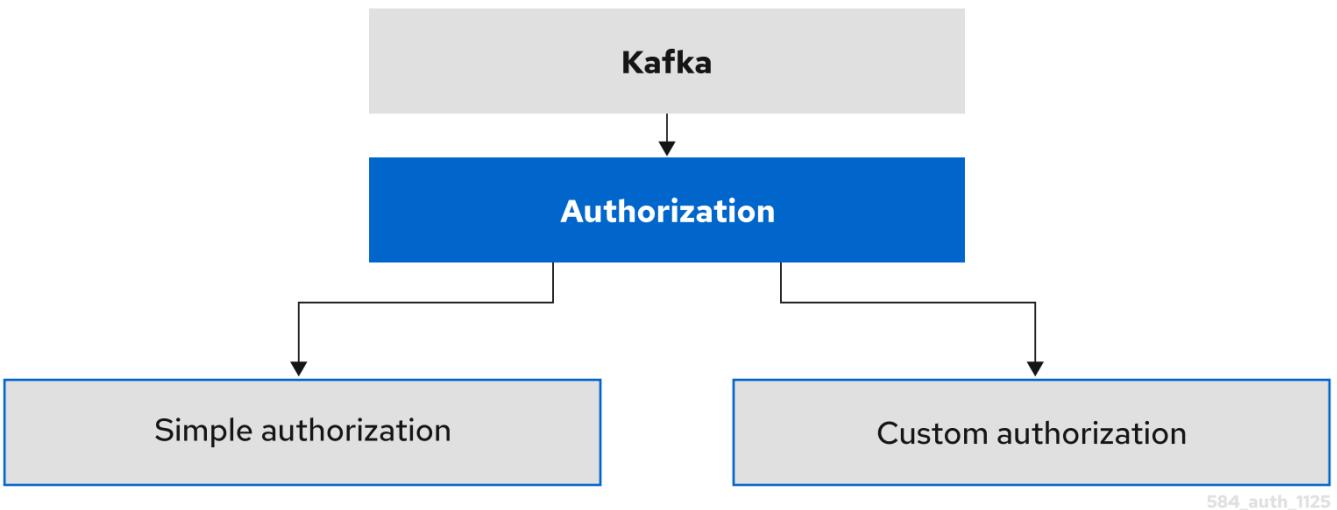


Figure 4. Kafka cluster authorization options

### 16.2.1. Designating super users

Super users can access all resources in your Kafka cluster regardless of any access restrictions, and are supported by all authorization mechanisms.

To designate super users for a Kafka cluster, add a list of user principals to the `superUsers` property of the `Kafka` resource. If a user uses mTLS authentication, the username is the common name from the TLS certificate subject prefixed with `CN=`. If you are not using the User Operator and using your own certificates for mTLS, the username is the full certificate subject.

A full certificate subject can include the following fields:

- `CN=<common_name>`
- `OU=<organizational_unit>`
- `O=<organization>`
- `L=<locality>`
- `ST=<state>`
- `C=<country_code>`

Omit any fields that are not applicable.

*An example configuration with super users*

```

apiVersion: kafka.strimzi.io/v1
kind: Kafka
metadata:
  name: my-cluster
  namespace: myproject
spec:
  kafka:
    # ...
    authorization:
      type: simple

```

```
superUsers:  
  - CN=user-1  
  - user-2  
  - CN=user-3  
  - CN=user-4,OU=my-ou,O=my-org,L=my-location,ST=my-state,C=US  
  - CN=user-5,OU=my-ou,O=my-org,C=GB  
  - CN=user-6,O=my-org  
# ...
```

## 16.3. Configuring user (client-side) security mechanisms

When configuring security mechanisms in clients, the clients are represented as users. Use the [KafkaUser](#) resource to configure the authentication, authorization, and access rights for Kafka clients.

Authentication permits user access, and authorization constrains user access to permissible actions. You can also create *super users* that have unconstrained access to Kafka brokers.

The authentication and authorization mechanisms must match the [specification for the listener used to access the Kafka brokers](#).

For more information on configuring a [KafkaUser](#) resource to access Kafka brokers securely, see [Example: Setting up secure client access](#).

### 16.3.1. Associating users with Kafka clusters

A [KafkaUser](#) resource includes a label that defines the appropriate name of the Kafka cluster (derived from the name of the [Kafka](#) resource) to which it belongs.

```
apiVersion: kafka.strimzi.io/v1  
kind: KafkaUser  
metadata:  
  name: my-user  
  labels:  
    strimzi.io/cluster: my-cluster
```

The label enables the User Operator to identify the [KafkaUser](#) resource and create and manage the user.

If the label does not match the Kafka cluster, the User Operator cannot identify the [KafkaUser](#), and the user is not created.

If the status of the [KafkaUser](#) resource remains empty, check your label configuration.

## 16.3.2. Configuring user authentication

Use the `KafkaUser` custom resource to configure authentication credentials for users (clients) that require access to a Kafka cluster. Configure the credentials using the `authentication` property in `KafkaUser.spec`. By specifying a `type`, you control what credentials are generated.

Supported authentication types:

- `tls` for mTLS authentication
- `tls-external` for mTLS authentication using external certificates
- `scram-sha-512` for SCRAM-SHA-512 authentication

If `tls` or `scram-sha-512` is specified, the User Operator creates authentication credentials when it creates the user. If `tls-external` is specified, the user still uses mTLS, but no authentication credentials are created. Use this option when you're providing your own certificates. When no authentication type is specified, the User Operator does not create the user or its credentials.

You can use `tls-external` to authenticate with mTLS using a certificate issued outside the User Operator. The User Operator does not generate a TLS certificate or a secret. You can still manage ACL rules and quotas through the User Operator in the same way as when you're using the `tls` mechanism. This means that you use the `CN=USER-NAME` format when specifying ACL rules and quotas. `USER-NAME` is the common name given in a TLS certificate.

### mTLS authentication

To use mTLS authentication, you set the `type` field in the `KafkaUser` resource to `tls`.

*Example user with mTLS authentication enabled*

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaUser
metadata:
  name: my-user
  labels:
    strimzi.io/cluster: my-cluster
spec:
  authentication:
    type: tls
  # ...
```

The authentication type must match the equivalent configuration for the `Kafka` listener used to access the Kafka cluster.

When the user is created by the User Operator, it creates a new secret with the same name as the `KafkaUser` resource. The secret contains a private and public key for mTLS. The public key is contained in a user certificate, which is signed by a clients CA (certificate authority) when it is created. All keys are in X.509 format.

**NOTE** If you are using the clients CA generated by the Cluster Operator, the user

certificates generated by the User Operator are also renewed when the clients CA is renewed by the Cluster Operator.

The user secret provides keys and certificates in PEM and PKCS #12 formats.

*Example secret with user credentials*

```
apiVersion: v1
kind: Secret
metadata:
  name: my-user
  labels:
    strimzi.io/kind: KafkaUser
    strimzi.io/cluster: my-cluster
type: Opaque
data:
  ca.crt: <public_key> # Public key of the clients CA used to sign this user certificate
  user.crt: <user_certificate> # Public key of the user
  user.key: <user_private_key> # Private key of the user
  user.p12: <store> # PKCS #12 store for user certificates and keys
  user.password: <password_for_store> # Protects the PKCS #12 store
```

When you configure a client, you specify the following:

- **Truststore** properties for the public cluster CA certificate to verify the identity of the Kafka cluster
- **Keystore** properties for the user authentication credentials to verify the client

The configuration depends on the file format (PEM or PKCS #12). This example uses PKCS #12 stores, and the passwords required to access the credentials in the stores.

*Example client configuration using mTLS in PKCS #12 format*

```
bootstrap.servers=<kafka_cluster_name>-kafka-bootstrap:9093 ①
security.protocol=SSL ②
ssl.truststore.location=/tmp/ca.p12 ③
ssl.truststore.password=<truststore_password> ④
ssl.keystore.location=/tmp/user.p12 ⑤
ssl.keystore.password=<keystore_password> ⑥
```

① The bootstrap server address to connect to the Kafka cluster.

② The security protocol option when using TLS for encryption.

③ The truststore location contains the public key certificate (**ca.p12**) for the Kafka cluster. A cluster CA certificate and password is generated by the Cluster Operator in the **<cluster\_name>-cluster-ca-cert** secret when the Kafka cluster is created.

④ The password (**ca.password**) for accessing the truststore.

⑤ The keystore location contains the public key certificate (`user.p12`) for the Kafka user.

⑥ The password (`user.password`) for accessing the keystore.

## mTLS authentication using a certificate issued outside the User Operator

To use mTLS authentication using a certificate issued outside the User Operator, you set the `type` field in the `KafkaUser` resource to `tls-external`. A secret and credentials are not created for the user.

*Example user with mTLS authentication that uses a certificate issued outside the User Operator*

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaUser
metadata:
  name: my-user
  labels:
    strimzi.io/cluster: my-cluster
spec:
  authentication:
    type: tls-external
  # ...
```

## SCRAM-SHA-512 authentication

To use the SCRAM-SHA-512 authentication mechanism, you set the `type` field in the `KafkaUser` resource to `scram-sha-512`.

*Example user with SCRAM-SHA-512 authentication enabled*

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaUser
metadata:
  name: my-user
  labels:
    strimzi.io/cluster: my-cluster
spec:
  authentication:
    type: scram-sha-512
  # ...
```

When the user is created by the User Operator, it creates a new secret with the same name as the `KafkaUser` resource. The secret contains the generated password in the `password` key, which is encoded with base64. In order to use the password, it must be decoded.

*Example secret with user credentials*

```
apiVersion: v1
kind: Secret
metadata:
  name: my-user
```

```

labels:
  strimzi.io/kind: KafkaUser
  strimzi.io/cluster: my-cluster
type: Opaque
data:
  password: Z2VuZXJhdGVkcGFzc3dvcmQ= ①
  sasl.jaas.config:
b3JnLmFwYWNoZS5rYWZrYS5jb21tb24uc2VjdXJpdHkuc2NyYW0uU2NyYW1Mb2dpbk1vZHVsZSBYXF1aXJlZC
B1c2VybmFtZT0ibXktDXNlcIlgcGFzc3dvcmQ9ImdlbmVyYXR1ZHBhc3N3b3JkIjsK ②

```

① The generated password, base64 encoded.

② The JAAS configuration string for SASL SCRAM-SHA-512 authentication, base64 encoded.

Decoding the generated password:

```
echo "Z2VuZXJhdGVkcGFzc3dvcmQ=" | base64 --decode
```

### Custom password configuration

When a user is created, Strimzi generates a random password. You can use your own password instead of the one generated by Strimzi. To do so, create a secret with the password and reference it in the `KafkaUser` resource.

*Example user with a password set for SCRAM-SHA-512 authentication*

```

apiVersion: kafka.strimzi.io/v1
kind: KafkaUser
metadata:
  name: my-user
  labels:
    strimzi.io/cluster: my-cluster
spec:
  authentication:
    type: scram-sha-512
    password:
      valueFrom:
        secretKeyRef:
          name: my-secret ①
          key: my-password ②
# ...

```

① The name of the secret containing the predefined password.

② The key for the password stored inside the secret.

### 16.3.3. Configuring user authorization

Use the `KafkaUser` custom resource to configure authorization rules for users (clients) that require access to a Kafka cluster. Configure the rules using the `authorization` property in `KafkaUser.spec`. By

specifying a `type`, you control what rules are used.

To use simple authorization, you set the `type` property to `simple` in `KafkaUser.spec.authorization`. The simple authorization uses the Kafka Admin API to manage the ACL rules inside your Kafka cluster. Whether ACL management in the User Operator is enabled or not depends on your authorization configuration in the Kafka cluster.

- For simple authorization, ACL management is always enabled.
- For OPA authorization (deprecated), ACL management is always disabled.
- For Keycloak authorization, you can manage the ACL rules directly in Keycloak. You can also delegate authorization to the simple authorizer as a fallback option in the configuration. When delegation to the simple authorizer is enabled, the User Operator will enable management of ACL rules as well.
- For custom authorization using a custom authorization plugin, use the `supportsAdminApi` property in the `.spec.kafka.authorization` configuration of the `Kafka` custom resource to enable or disable the support.

Authorization is cluster-wide. The authorization type must match the equivalent configuration in the `Kafka` custom resource.

If ACL management is not enabled, Strimzi rejects a resource if it contains any ACL rules.

If you're using a standalone deployment of the User Operator, ACL management is enabled by default. You can disable it using the `STRIMZI_ACLS_ADMIN_API_SUPPORTED` environment variable.

If no authorization is specified, the User Operator does not provision any access rights for the user. Whether such a `KafkaUser` can still access resources depends on the authorizer being used. For example, for `simple` authorization, this is determined by the `allow.everyone.if.no.acl.found` configuration in the Kafka cluster.

#### 16.3.4. Defining ACL rules

The `simple` authorization mechanism uses ACL rules to manage access to Kafka brokers. Enabling `simple` authorization for a Kafka cluster means access is denied unless explicit ACL rules are configured in the `KafkaUser` resource. ACL rules define access rights for users, which you specify using the `acls` property.

*Example user with ACL rules configuration*

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaUser
metadata:
  name: my-user
  labels:
    strimzi.io/cluster: my-cluster
spec:
  authentication:
    type: tls
  authorization:
```

```

type: simple
acls:
  - resource: ①
    type: topic
    name: my-topic
    patternType: literal
    operations: ②
      - Describe
      - Read
  - resource:
    type: group
    name: my-group
    patternType: literal
    operations:
      - Read

```

① Defines the Kafka resource type to which the ACL rules apply, such as `topic` or `group`, the name of the resource, and how the name is matched by `patternType`.

② Specifies the operations, such as `Describe`, `Read`, that the user is allowed to perform on the resource.

For more information about the `AclRule` object and the properties used to configure ACLs, see the [AclRule schema reference](#).

### 16.3.5. Configuring user quotas

Configure the `spec` for the `KafkaUser` resource to enforce quotas so that a user does not overload Kafka brokers. Set size-based network usage and time-based CPU utilization thresholds.

Partition mutations occur in response to the following types of user requests:

- Creating partitions for a new topic
- Adding partitions to an existing topic
- Deleting partitions from a topic

You can also add a partition mutation quota to control the rate at which requests to change partitions are accepted.

*Example KafkaUser with user quotas*

```

apiVersion: kafka.strimzi.io/v1
kind: KafkaUser
metadata:
  name: my-user
  labels:
    strimzi.io/cluster: my-cluster
spec:
  # ...
  quotas:

```

```
producerByteRate: 1048576 ①
consumerByteRate: 2097152 ②
requestPercentage: 55 ③
controllerMutationRate: 10 ④
```

- ① Byte-per-second quota on the amount of data the user can push to a Kafka broker.
- ② Byte-per-second quota on the amount of data the user can fetch from a Kafka broker.
- ③ CPU utilization limit as a percentage of time for a client group.
- ④ Number of concurrent partition creation and deletion operations (mutations) allowed per second.

Using quotas for Kafka clients might be useful in a number of situations. Consider a wrongly configured Kafka producer which is sending requests at too high a rate. Such misconfiguration can cause a denial of service to other clients, so the problematic client ought to be blocked. By using a network limiting quota, it is possible to prevent this situation from significantly impacting other clients.

**NOTE** Strimzi supports user-level quotas, but not client-level quotas.

## 16.4. Example: Setting up secure client access

This procedure shows how to configure client access to a Kafka cluster from outside Kubernetes or from another Kubernetes cluster. It's split into two parts:

- Securing Kafka brokers
- Securing user access to Kafka

### *Resource configuration*

Client access to the Kafka cluster is secured with the following configuration:

1. An external listener is configured with TLS encryption and mutual TLS (mTLS) authentication in the [Kafka](#) resource, as well as [simple](#) authorization.
2. A [KafkaUser](#) is created for the client, utilizing mTLS authentication, and Access Control Lists (ACLs) are defined for [simple](#) authorization.

At least one listener supporting the desired authentication must be configured for the [KafkaUser](#).

Listeners can be configured for mutual [TLS](#), [SCRAM-SHA-512](#), or [custom](#) authentication. While mTLS always uses encryption, it's also recommended when using SCRAM-SHA-512 or a custom configuration for OAuth 2.0 authentication.

Authorization options for Kafka include [simple](#) or [custom](#). When enabled, authorization is applied to all enabled listeners.

To ensure compatibility between Kafka and clients, configuration of the following authentication and authorization mechanisms must align:

- For `type: tls` and `type: scram-sha-512` authentication types, `Kafka.spec.kafka.listeners[*].authentication` must match `KafkaUser.spec.authentication`
- For `type: simple` authorization, `Kafka.spec.kafka.authorization` must match `KafkaUser.spec.authorization`

For example, mTLS authentication for a user is only possible if it's also enabled in the Kafka configuration.

#### *Automation and certificate management*

Strimzi operators automate the configuration process and create the certificates required for authentication:

- The Cluster Operator creates the listeners and sets up the cluster and client certificate authority (CA) certificates to enable authentication within the Kafka cluster.
- The User Operator creates the user representing the client and the security credentials used for client authentication, based on the chosen authentication type.

You add the certificates to your client configuration.

In this procedure, the CA certificates generated by the Cluster Operator are used. Alternatively, you can replace them by [installing your own custom CA certificates](#). You can also configure listeners to use [Kafka listener certificates managed by an external CA](#).

Certificates are available in PEM (.crt) and PKCS #12 (.p12) formats. This procedure uses PEM certificates. Use PEM certificates with clients that support the X.509 certificate format.

**NOTE** For internal clients in the same Kubernetes cluster and namespace, you can mount the cluster CA certificate in the pod specification. For more information, see [Configuring internal clients to trust the cluster CA](#).

#### *Prerequisites*

- The Kafka cluster is available for connection by a client running outside the Kubernetes cluster
- The Cluster Operator and User Operator are running in the cluster

### 16.4.1. Securing Kafka brokers

#### 1. Configure the Kafka cluster with a Kafka listener.

- Define the authentication required to access the Kafka broker through the listener.
- Enable authorization on the Kafka broker.

#### *Example listener configuration*

```
apiVersion: kafka.strimzi.io/v1
kind: Kafka
metadata:
  name: my-cluster
  namespace: myproject
spec:
```

```

kafka:
  # ...
  listeners: ①
    - name: external1 ②
      port: 9094 ③
      type: <listener_type> ④
      tls: true ⑤
      authentication:
        type: tls ⑥
      configuration: ⑦
        #...
    authorization: ⑧
      type: simple
    superUsers:
      - super-user-name ⑨
  # ...

```

- ① Configuration options for enabling external listeners are described in the [Generic Kafka listener schema reference](#).
- ② Name to identify the listener. Must be unique within the Kafka cluster.
- ③ Port number used by the listener inside Kafka. The port number has to be unique within a given Kafka cluster. Allowed port numbers are 9092 and higher with the exception of ports 9404 and 9999, which are already used for Prometheus and JMX. Depending on the listener type, the port number might not be the same as the port number that connects Kafka clients.
- ④ External listener type specified as `route` (OpenShift only), `loadbalancer`, `nodeport` or `ingress` (Kubernetes only). An internal listener is specified as `internal` or `cluster-ip`.
- ⑤ Required. TLS encryption on the listener. For `route` and `ingress` type listeners it must be set to `true`. For mTLS authentication, also use the `authentication` property.
- ⑥ Client authentication mechanism on the listener. For server and client authentication using mTLS, you specify `tls: true` and `authentication.type: tls`.
- ⑦ (Optional) Depending on the requirements of the listener type, you can specify additional [listener configuration](#).
- ⑧ Authorization specified as `simple`, which uses the `StandardAuthorizer` Kafka plugin.
- ⑨ (Optional) Super users can access all brokers regardless of any access restrictions defined in ACLs.

**WARNING**

An OpenShift route address comprises the Kafka cluster name, the listener name, the project name, and the domain of the router. For example, `my-cluster-kafka-external1-bootstrap-my-project.domain.com` (`<cluster_name>-kafka-<listener_name>-bootstrap-<namespace>.domain.com`). Each DNS label (between periods ".") must not exceed 63 characters, and the total length of the address must not exceed 255 characters.

## 2. Apply the changes to the **Kafka** resource configuration.

The Kafka cluster is configured with a Kafka broker listener using mTLS authentication.

A service is created for each Kafka broker pod.

A service is created to serve as the *bootstrap address* for connection to the Kafka cluster.

A service is also created as the *external bootstrap address* for external connection to the Kafka cluster using **nodeport** listeners.

The cluster CA certificate to verify the identity of the kafka brokers is also created in the secret **<cluster\_name>-cluster-ca-cert**.

**NOTE**

If you scale your Kafka cluster while using external listeners, it might trigger a rolling update of all Kafka brokers. This depends on the configuration.

## 3. Retrieve the bootstrap address you can use to access the Kafka cluster from the status of the **Kafka** resource.

```
kubectl get kafka <kafka_cluster_name>
-o=jsonpath='{.status.listeners[?(@.name=="<listener_name>")].bootstrapServers}{"\n"}'
```

For example:

```
kubectl get kafka my-cluster
-o=jsonpath='{.status.listeners[?(@.name=="external")].bootstrapServers}{"\n"}'
```

Use the bootstrap address in your Kafka client to connect to the Kafka cluster.

### 16.4.2. Securing user access to Kafka

#### 1. Create or modify a user representing the client that requires access to the Kafka cluster.

- Specify the same authentication type as the **Kafka** listener.
- Specify the authorization ACLs for **simple** authorization.

*Example user configuration*

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaUser
metadata:
  name: my-user
  labels:
    strimzi.io/cluster: my-cluster ①
spec:
  authentication:
    type: tls ②
```

```

authorization:
  type: simple
  acls: ③
    - resource:
        type: topic
        name: my-topic
        patternType: literal
    operations:
      - Describe
      - Read
    - resource:
        type: group
        name: my-group
        patternType: literal
    operations:
      - Read

```

- ① The label must match the label of the Kafka cluster.
- ② Authentication specified as mutual `tls`.
- ③ Simple authorization requires an accompanying list of ACL rules to apply to the user. The rules define the operations allowed on Kafka resources based on the `username` (`my-user`).

## 2. Apply the changes to the `KafkaUser` resource configuration.

The user is created, as well as a secret with the same name as the `KafkaUser` resource. The secret contains a public and private key for mTLS authentication.

*Example secret with user credentials*

```

apiVersion: v1
kind: Secret
metadata:
  name: my-user
  labels:
    strimzi.io/kind: KafkaUser
    strimzi.io/cluster: my-cluster
type: Opaque
data:
  ca.crt: <public_key> # Public key of the clients CA used to sign this user
  certificate
  user.crt: <user_certificate> # Public key of the user
  user.key: <user_private_key> # Private key of the user
  user.p12: <store> # PKCS #12 store for user certificates and keys
  user.password: <password_for_store> # Protects the PKCS #12 store

```

## 3. Extract the cluster CA certificate from the `<cluster_name>-cluster-ca-cert` secret of the Kafka cluster.

```
kubectl get secret <cluster_name>-cluster-ca-cert -o jsonpath='{.data.ca\.crt}' |
```

```
base64 -d > ca.crt
```

4. Extract the user CA certificate from the `<user_name>` secret.

```
kubectl get secret <user_name> -o jsonpath='{.data.user\.crt}' | base64 -d > user.crt
```

5. Extract the private key of the user from the `<user_name>` secret.

```
kubectl get secret <user_name> -o jsonpath='{.data.user\.key}' | base64 -d > user.key
```

6. Configure your client with the bootstrap address hostname and port for connecting to the Kafka cluster:

```
props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "<hostname>:<port>");
```

7. Configure your client with the truststore credentials to verify the identity of the Kafka cluster.

Specify the public cluster CA certificate.

*Example truststore configuration*

```
props.put(CommonClientConfigs.SECURITY_PROTOCOL_CONFIG, "SSL");
props.put(SslConfigs.SSL_TRUSTSTORE_TYPE_CONFIG, "PEM");
props.put(SslConfigs.SSL_TRUSTSTORE_CERTIFICATES_CONFIG, "<ca.crt_file_content>");
```

SSL is the specified security protocol for mTLS authentication. Specify `SASL_SSL` for SCRAM-SHA-512 authentication over TLS. PEM is the file format of the truststore.

8. Configure your client with the keystore credentials to verify the user when connecting to the Kafka cluster.

Specify the public certificate and private key.

*Example keystore configuration*

```
props.put(CommonClientConfigs.SECURITY_PROTOCOL_CONFIG, "SSL");
props.put(SslConfigs.SSL_KEYSTORE_TYPE_CONFIG, "PEM");
props.put(SslConfigs.SSL_KEYSTORE_CERTIFICATE_CHAIN_CONFIG,
"<user.crt_file_content>");
props.put(SslConfigs.SSL_KEYSTORE_KEY_CONFIG, "<user.key_file_content>");
```

Add the keystore certificate and the private key directly to the configuration. Add as a single-line format. Between the `BEGIN CERTIFICATE` and `END CERTIFICATE` delimiters, start with a newline character (`\n`). End each line from the original certificate with `\n` too.

#### *Example keystore configuration*

```
props.put(SslConfigs.SSL_KEYSTORE_CERTIFICATE_CHAIN_CONFIG, "-----BEGIN  
CERTIFICATE-----  
\n<user_certificate_content_line_1>\n<user_certificate_content_line_n>\n-----END  
CERTIFICATE---");  
props.put(SslConfigs.SSL_KEYSTORE_KEY_CONFIG, "-----BEGIN PRIVATE KEY-----  
\n<user_key_content_line_1>\n<user_key_content_line_n>\n-----END PRIVATE KEY-----  
");
```

## 16.5. Troubleshooting TLS hostname verification with node ports

Off-cluster access using node ports with TLS encryption enabled does not support TLS hostname verification. This is because Strimzi does not know the address of the node where the broker pod is scheduled and cannot include it in the broker certificate. Consequently, clients that perform hostname verification will fail to connect.

For example, a Java client will fail with the following exception:

#### *Exception for TLS hostname verification*

```
Caused by: java.security.cert.CertificateException: No subject alternative names  
matching IP address 168.72.15.231 found  
...
```

To connect, you must disable hostname verification. In the Java client, set the `ssl.endpoint.identification.algorithm` configuration option to an empty string.

When configuring the client using a properties file, you can do it this way:

```
ssl.endpoint.identification.algorithm=
```

When configuring the client directly in Java, set the configuration option to an empty string:

```
props.put("ssl.endpoint.identification.algorithm", "");
```

Alternatively, if you know the addresses of the worker nodes where the brokers are scheduled, you can add them as additional SANs (Subject Alternative Names) to the broker certificates manually. For example, this might apply if your cluster is running on a bare metal deployment with a limited number of available worker nodes. Use the `alternativeNames` property to specify additional SANS.

# Chapter 17. Configuring OAuth 2.0 token-based security

Strimzi supports OAuth 2.0 token-based authentication and authorization for securing Kafka clusters and Kafka-related components.

With OAuth 2.0 integration you can:

- Enable token-based authentication on Kafka brokers
- Configure Kafka components, such as Kafka Connect, MirrorMaker 2, and the HTTP Bridge, to authenticate using access tokens
- Use token claims or Keycloak Authorization Services to perform fine-grained authorization

OAuth 2.0 provides centralized identity and access control using access tokens issued by an authorization server. Kafka brokers validate these tokens when clients connect, and authorization providers can use token claims to make access decisions.

This section describes the minimal configuration required to deploy a Kafka cluster and clients with token-based authentication and authorization.

## 17.1. Migrating to the `custom` authentication type

The `oauth` and `keycloak` authentication types are deprecated and will be removed in a future release. Kafka listeners and Kafka clients must now configure OAuth 2.0 using the `custom` authentication type. OAuth 2.0 support remains the same; only the configuration model changes.

This change affects **both server-side listener configuration** and **client-side authentication** for Strimzi-managed components.

For more information on the configuration options, see the following API schema references:

- [KafkaListenerAuthenticationCustom schema reference](#)
- [KafkaClientAuthenticationCustom schema reference](#)

## 17.2. Before you begin configuring OAuth 2.0

Strimzi enables OAuth 2.0 for Kafka using the Strimzi Kafka OAuth libraries to handle authentication and authorization. This section describes only the configuration that is required when deploying OAuth with Strimzi.

For reference information, advanced configuration options, and platform-specific examples, see the [Strimzi Kafka OAuth project documentation](#).

Strimzi can integrate with any OAuth 2.0 or OpenID Connect compliant authorization server. If you are using a specific provider, refer to its documentation for information on how the following must be configured:

- Client registration
- Token and JWKS endpoints
- Authorization features (such as scopes, roles, or policies)
- Realm, tenant, or application definitions

Strimzi also provides **Keycloak-specific examples**, including realm files and Kafka cluster configurations that demonstrate how to use Keycloak Authorization Services with Kafka:

- [Example configuration files](#)

If you use Keycloak, you may also find the Keycloak project documentation helpful:

- [Keycloak documentation](#)

Review the documentation for your OAuth provider to understand how tokens are issued and validated, and how authorization rules are managed, before configuring OAuth on Kafka listeners or Kafka clients in Strimzi.

## 17.3. Server-side configuration

Server-side configuration enables OAuth 2.0 token-based authentication on Kafka listeners.

To configure OAuth 2.0 on the server side, define the **custom** authentication type on the listeners used by clients. The configuration specifies how Kafka brokers validate tokens.

Strimzi supports OAuth 2.0 for both server-side and client-side authentication. Kafka brokers validate tokens using the OAuth configuration defined on the listener, and Kafka components, such as Kafka Connect, MirrorMaker 2, and the HTTP Bridge authenticate as OAuth clients when connecting to Kafka.

You can also enable authorization based on token claims when using an OAuth-compatible authorization service, such as Keycloak Authorization Services. For more information, see [Enabling authorization on Kafka brokers](#).

The following sections describe the minimal configuration required for:

- Preparing an OAuth 2.0 authorization server
- Configuring token-based authentication on Kafka listeners

**NOTE**

Advanced OAuth settings such as timeouts, JWKS refresh, introspection, and TLS options are described in the [Strimzi Kafka OAuth project documentation](#). Keycloak-specific examples for use with Strimzi are provided in the [example configuration files](#).

### 17.3.1. Requirements for the authorization server

To use OAuth 2.0 token-based security with Strimzi, you need an OAuth 2.0-compliant authorization server that issues access tokens for Kafka clients.

The authorization server must provide the following capabilities:

- An endpoint for issuing access tokens.
- A JWKS endpoint for publishing signing keys, or an introspection endpoint for validating opaque tokens.
- TLS support for secure communication with Kafka brokers.
- Client registrations when the broker or Kafka clients must authenticate to protected endpoints.  
For example, a client ID and secret are required when using a protected introspection endpoint.

Only one validation method is required:

- **Local: JWT validation** using the JWKS endpoint
- **Remote: Token introspection** using the introspection endpoint

The authorization server must expose the selected endpoint externally so that Kafka brokers can validate tokens at runtime.

### 17.3.2. Configuring authentication on Kafka listeners

To enable OAuth 2.0 token-based authentication on a Kafka listener, configure the `custom` authentication type with the `OAUTHBEARER` SASL mechanism. The listener configuration specifies the callback handler and JAAS module used to validate tokens.

All OAuth 2.0 validation settings (such as JWKS or token introspection) are provided through the JAAS configuration string inside the listener configuration.

#### JWT validation example

The following example shows a minimal listener configuration that validates JSON Web Tokens (JWTs) using a JWKS endpoint.

```
# ...
listeners:
- name: tls
  port: 9093
  type: internal
  tls: true
  authentication:
    type: custom
    sasl: true
    listenerConfig:
      sasl.enabled.mechanisms: OAUTHBEARER
      oauthbearer.sasl.server.callback.handler.class:
        io.strimzi.kafka.oauth.server.JaasServerOAuthValidatorCallbackHandler
        oauthbearer.sasl.jaas.config: |
          org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule required
          oauth.valid.issuer.uri="https://${SSO_HOST}/realms/kafka-authz"
          oauth.jwks.endpoint.uri="https://${SSO_HOST}/realms/kafka-
```

```

authz/protocol/openid-connect/certs"
    oauth.username.claim="preferred_username"
    oauth.ssl.truststore.location="/mnt/oauth-certs/tls.crt"
    oauth.ssl.truststore.type="PEM";
connections.max.reauth.ms: 3600

```

In this configuration:

- The listener uses the **OAUTHBEARER** SASL mechanism.
- Tokens are validated locally using the JWKS endpoint published by the authorization server.
- The JAAS module and callback handler perform token validation and map a specified token claim to the Kafka principal.
- The truststore is mounted into the Kafka pod and used for TLS connections to the JWKS endpoint.
- The listener requires periodic re-authentication. The **connections.max.reauth.ms** setting ensures that the broker prompts clients to re-authenticate at regular intervals, allowing expiring OAuth access tokens to be refreshed without closing the connection.

## Token introspection example

The following example shows a listener configuration that validates opaque access tokens using an OAuth 2.0 introspection endpoint. The Kafka broker authenticates to the authorization server using a client ID and a client secret configured in the JAAS module.

```

# ...
spec:
  kafka:
    listeners:
      - name: tls
        port: 9093
        type: internal
        tls: true
        authentication:
          type: custom
          sasl: true
          listenerConfig:
            sasl.enabled.mechanisms: OAUTHBEARER
            oauthbearer.sasl.server.callback.handler.class:
              io.strimzi.kafka.oauth.server.JaasServerOAuthValidatorCallbackHandler
              oauthbearer.sasl.jaas.config: |
                org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule
required
          oauth.access.token.is.jwt="false"

  oauth.introspection.endpoint.uri="https://${SSO_HOST}/realms/myrealm/protocol/openid-
connect/token/introspect"
          oauth.client.id="kafka-broker"
          oauth.client.secret=<client-secret-value>

```

```
oauth.valid.issuer.uri="https://${SSO_HOST}/realms/myrealm"  
oauth.username.claim="preferred_username"  
oauth.ssl.truststore.location="/opt/oauth-certs/tls.crt"  
oauth.ssl.truststore.type="PEM";  
# ...
```

In this configuration:

- The listener uses the `OAUTHBEARER` SASL mechanism.
- Tokens are validated by calling the OAuth 2.0 introspection endpoint for each new client connection.
- The broker authenticates to the protected introspection endpoint using a client ID and client secret configured in the JAAS module.
- The JAAS login module and callback handler perform token validation and map the `preferred_username` value from the introspection endpoint response to a Kafka principal.
- The truststore used for TLS connections to the authorization server is specified by `oauth.ssl.truststore.location` and `oauth.ssl.truststore.type`.
- The `oauth.access.token.is.jwt="false"` setting ensures that the broker does not attempt to interpret access tokens as JWTs (for example, if debug logging is enabled).

**NOTE** In production deployments, store the client secret securely, such as in a `Secret`, and inject it through an environment variable or other secure mechanism. For more information, see [Loading configuration values from external sources](#).

## Authenticating brokers to protected authorization-server endpoints

Some authorization servers expose the JWKS endpoint publicly but require authentication when accessing protected endpoints, such as the introspection endpoint.

When the broker accesses a protected endpoint, it must authenticate to the authorization server.

To configure HTTP Basic authentication, set the following properties:

- `oauth.client.id`
- `oauth.client.secret`

To configure HTTP Bearer authentication, set one of the following properties:

- `oauth.server.bearer.token.location` to specify the file path that contains the bearer token.
- `oauth.server.bearer.token` to specify the bearer token in clear text.

**NOTE** In production environments, avoid storing credentials or bearer tokens in clear text. Use a secure mechanism to provide these values to the broker.

## 17.4. Client-side configuration

Kafka clients, including Strimzi-managed components and external applications, can authenticate to Kafka using OAuth 2.0 access tokens. Clients obtain a token from an authorization server and present it to Kafka using the **OAUTHBEARER** SASL mechanism. Kafka brokers validate the token using the OAuth configuration defined on their listeners.

Client applications must use OAuth settings that match the broker configuration, including token type, issuer, and TLS trust configuration.

This section describes:

- OAuth requirements common to all Kafka clients
- How Strimzi components authenticate as OAuth clients
- How external Kafka applications configure **OAUTHBEARER** authentication
- How clients obtain and refresh tokens

**NOTE**

Advanced OAuth client settings are described in the [Strimzi Kafka OAuth project documentation](#).

### 17.4.1. Common requirements for OAuth 2.0 clients

Kafka clients that use OAuth-based authentication typically require:

- The **OAUTHBEARER** SASL mechanism
- A token endpoint to obtain OAuth 2.0 access tokens
- A client ID and credentials registered with the authorization server
- TLS trust configuration for HTTPS calls to the authorization server
- A way to refresh tokens before they expire
- Truststore configuration that matches the Kafka broker listener

Strimzi does not decide which OAuth 2.0 flow is used. The chosen flow depends on the authorization server and the client application.

### 17.4.2. Configuring Strimzi-managed components

Strimzi-managed components such as Kafka Connect, MirrorMaker 2, and HTTP Bridge authenticate as OAuth 2.0 clients when connecting to Kafka brokers. These components use the **KafkaClientAuthenticationCustom** schema and configure OAuth by supplying SASL and JAAS settings.

Each component requires:

- A token endpoint URL
- A client ID and credentials stored in a Kubernetes secret
- TLS trust configuration for connecting to the authorization server

- **OAUTHBEARER** SASL settings to authenticate to Kafka brokers

The following example shows a simplified OAuth configuration for the HTTP Bridge. The same configuration pattern applies to Kafka Connect and MirrorMaker 2.

```

apiVersion: kafka.strimzi.io/v1
kind: KafkaBridge
metadata:
  name: my-bridge
spec:
  # ...
  authentication:
    type: custom
    sasl: true
    config:
      sasl.mechanism: OAUTHBEARER
      sasl.login.callback.handler.class:
        io.strimzi.kafka.oauth.client.JaasClientOAuthLoginCallbackHandler
      sasl.jaas.config: |
        org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule required
        oauth.token.endpoint.uri="https://<auth_server_address>/<path_to_token_endpoint>"
        oauth.client.id="kafka-bridge"
        oauth.client.secret="${strimzienv:OAUTH_CLIENT_SECRET}";
  template:
    bridgeContainer:
      env:
        - name: OAUTH_CLIENT_SECRET
          valueFrom:
            secretKeyRef:
              name: oauth-secret
              key: oauth-client-secret
      # ...

```

In this configuration:

- The HTTP Bridge uses the **custom** authentication type.
- SASL is enabled and configured with the **OAUTHBEARER** mechanism.
- The Strimzi OAuth client callback handler retrieves and refreshes access tokens.
- The JAAS login module defines how the HTTP Bridge authenticates to the authorization server.
- The **oauth.token.endpoint.uri** property specifies where tokens are obtained.
- The OAuth client ID identifies the HTTP Bridge as a registered OAuth client.
- The client secret is injected into the HTTP Bridge container as an environment variable from a Kubernetes secret and consumed through the **`\${strimzienv:OAUTH\_CLIENT\_SECRET}`** reference in the JAAS configuration.

### 17.4.3. Configuring external Kafka client applications

External Kafka clients authenticate using the **OAUTHBEARER** SASL mechanism. Clients obtain an access token from the authorization server and present the token when connecting to Kafka.

Client configuration requires:

- The **OAUTHBEARER** SASL mechanism
- TLS trust configuration for Kafka broker communication
- OAuth settings that match the broker's validation method (JWKS or introspection)
- A mechanism to retrieve, refresh, or supply access tokens at runtime

The following example shows a minimal Java client configuration:

```
security.protocol=SASL_SSL
sasl.mechanism=OAUTHBEARER

# Truststore for connecting to Kafka brokers
ssl.truststore.location=/tmp/kafka-truststore.p12
ssl.truststore.password=$KAFKA_STOREPASS
ssl.truststore.type=PKCS12

# Strimzi OAuth login callback handler
sasl.login.callback.handler.class=io.strimzi.kafka.oauth.client.JaasClientOAuthLoginCallbackHandler

# JAAS configuration for obtaining access tokens
sasl.jaas.config=org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule
required \
    oauth.token.endpoint.uri="https://auth.url.com/oauth/token" \
    oauth.client.id="kafka-client" \
    oauth.client.secret="client-secret" \
    oauth.ssl.truststore.location="/tmp/oauth-truststore.p12" \
    oauth.ssl.truststore.password="$OAUTH_STOREPASS" \
    oauth.ssl.truststore.type="PKCS12" \
    oauth.scope="example-scope" \
    oauth.audience="example-audience" ;
```

In this configuration:

- **SASL\_SSL** enables encrypted communication with Kafka brokers.
- The SASL mechanism is set to **OAUTHBEARER** for token-based authentication.
- A truststore validates the Kafka broker certificate.
- The OAuth login module handles token acquisition.
- The client uses the client-credentials flow to obtain tokens.
- A separate truststore is used for communication with the authorization server.

- Optional OAuth parameters such as `scope` and `audience` can be added when required.
- The Strimzi OAuth callback handler refreshes tokens automatically.

#### 17.4.4. Obtaining tokens

Kafka clients must obtain an OAuth 2.0 access token before connecting to a Kafka listener that is configured for token-based authentication. The method for obtaining a token depends on the client application and the authorization server.

In practice, Kafka clients can use a range of token acquisition patterns. Common examples include the following:

##### Client credentials with remote validation

The client requests a short-lived access token using a client ID and credentials. The broker validates the token by calling the authorization server's introspection endpoint.

##### Client credentials with local validation

The client requests a short-lived JWT access token using a client ID and credentials. The broker validates the token locally using a JWKS endpoint.

##### Long-lived access token with remote validation

The client is issued a long-lived opaque token outside the application (for example, by a CI/CD system). The broker validates the token through the introspection endpoint. Token refresh is the responsibility of the application.

##### Long-lived JWT token with local validation

The client is issued a long-lived JWT outside the application. The broker validates the token locally against the JWKS endpoint. The token cannot be refreshed, so the connection fails after expiry.

Strimzi does not manage token acquisition for external applications. Applications must request, store, and refresh tokens according to their security requirements.

#### 17.4.5. Token expiry and re-authentication

Client sessions may expire when a token expires or when the broker requires re-authentication.

If the broker is configured with `connections.max.reauth.ms`, clients must refresh the access token before re-authentication occurs. When using the client-credentials flow, the Strimzi OAuth client library refreshes tokens automatically. If an application supplies a long-lived, manually generated token, the client cannot refresh it and authentication fails when the token expires.

### 17.5. Enabling authorization on Kafka brokers

You can enable authorization so that Kafka brokers use information in access tokens to determine whether clients are permitted to perform specific operations.

Authorization builds on OAuth 2.0 authentication. Clients must connect through a Kafka listener

that is configured for OAuth 2.0 using the `custom` authentication type. Only tokens that are successfully validated by the listener are evaluated by the authorizer.

To enable authorization, set the `authorization` field on the Kafka cluster resource and configure the `custom` authorization type. The broker evaluates authorization decisions using token claims provided by the OAuth 2.0 authorization server.

**NOTE** Advanced authorization settings are described in the [Strimzi Kafka OAuth project documentation](#). Keycloak-specific examples for use with Strimzi are provided in the [example configuration files](#).

The following example shows a Kafka cluster that uses OAuth 2.0 authentication and Keycloak Authorization Services for authorization:

```
spec:
  kafka:
    # ...
    listeners:
      - name: tls
        port: 9093
        type: internal
        tls: true
        authentication:
          type: custom
          sasl: true
          listenerConfig:
            sasl.enabled.mechanisms: OAUTHBEARER
            oauthbearer.sasl.server.callback.handler.class:
              io.strimzi.kafka.oauth.server.JaasServerOAuthValidatorCallbackHandler
              oauthbearer.sasl.jaas.config: |
                org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule
required
  oauth.valid.issuer.uri="https://${SSO_HOST}/realms/kafka-authz"
  oauth.jwks.endpoint.uri="https://${SSO_HOST}/realms/kafka-
authz/protocol/openid-connect/certs"
  oauth.username.claim="preferred_username"
  oauth.ssl.truststore.location="/mnt/oauth-certs/tls.crt"
  oauth.ssl.truststore.type="PEM";
  connections.max.reauth.ms: 3600
  authorization:
    type: custom
    authorizerClass: io.strimzi.kafka.oauth.server.authorizer.KeycloakAuthorizer
    superUsers:
      - service-account-kafka
  config:
    # Needed for OAuth authentication and Keycloak authorization
    principal.builder.class:
      io.strimzi.kafka.oauth.server.OAuthKafkaPrincipalBuilder
      # Needed for Keycloak authorization
      strimzi.authorization.client.id: kafka
```

```

strimzi.authorization.token.endpoint.uri: https://${SSO_HOST}/realms/kafka-
authz/protocol/openid-connect/token
    strimzi.authorization.delegate.to.kafka.acl: "false"
    strimzi.authorization.ssl.truststore.location: /mnt/oauth-certs/tls.crt
    strimzi.authorization.ssl.truststore.type: PEM
# ...

```

In this configuration:

- The listener is configured with the **OAUTHBEARER** SASL mechanism and validates access tokens using the JWKS endpoint published by Keycloak.
- The JAAS module and callback handler (**JaasServerOAuthValidatorCallbackHandler**) validate the token and extract the authenticated principal.
- **connections.max.reauth.ms** enforces periodic re-authentication so that expiring access tokens can be refreshed without disconnecting the client.
- The authorization block enables the **custom** authorizer and loads the **KeycloakAuthorizer** class, which evaluates permissions derived from Keycloak Authorization Services.
- The **superUsers** list defines identities that bypass authorization checks.
- The **principal.builder.class** maps validated OAuth tokens to Kafka principals, which the authorizer uses when evaluating access decisions.
- **strimzi.authorization.client.id** identifies the broker to Keycloak Authorization Services.
- **strimzi.authorization.token.endpoint.uri** specifies the token endpoint used by the authorizer to obtain Keycloak permission grants.
- TLS configuration for the authorizer (**strimzi.authorization.ssl.\***) ensures secure HTTPS communication with the authorization server.
- **strimzi.authorization.delegate.to.kafka.acl** controls whether failed authorization checks are passed to Kafka's built-in ACL authorizer (disabled in this example).

### 17.5.1. How Kafka permissions map to Keycloak

Kafka uses an authorization model based on **resource types** and **operations**. For example, a user might have **Describe** and **Write** permissions on a **Topic** resource.

Keycloak Authorization Services use **resources**, **scopes**, **policies**, and **permissions**. When you use **KeycloakAuthorizer**, Kafka resources are mapped to Keycloak resources, and Kafka operations are mapped to Keycloak authorization scopes. The authorizer evaluates Keycloak permissions for the authenticated user or service account to decide whether an operation is allowed.

Resources in Keycloak follow a naming pattern that matches Kafka resources. The general format is **resourceType:pattern**.

Examples:

- **Topic:my-topic**
- **Topic:topic-\***

- `Group:group-*`
- `Cluster:*`

A resource pattern can also be prefixed with the Kafka cluster name:

Examples:

- `kafka-cluster:my-cluster,Topic:*`
- `kafka-cluster:*,Group:b_*`

If the `kafka-cluster:` prefix is omitted, it is treated as `kafka-cluster:*`.

Authorization scopes normally mirror Kafka operations, such as the following:

- `Create, Write, Read, Delete, Describe, Alter, DescribeConfigs, AlterConfigs, ClusterAction, IdempotentWrite`

The `All` permission is not supported. Define explicit scopes for each permitted operation.

Permissions in Keycloak combine resources, scopes, and policies that target users, groups, or service accounts.

Common patterns include:

- **Role policies** for service accounts (for example, microservices using client credentials)
- **Group policies** for human users (for example, CLI users who need controlled access)

## 17.6. Using Keycloak as an OAuth 2.0 provider

Keycloak is a commonly used OAuth 2.0 authorization server for securing Kafka clusters. Strimzi supports using Keycloak both for token-based authentication and, optionally, for fine-grained authorization.

A Keycloak authorizer (`KeycloakAuthorizer`) is provided with Strimzi. The authorizer retrieves permission grants from Keycloak Authorization Services and enforces authorization locally on each Kafka broker.

When using Keycloak with Strimzi:

- Configure OAuth 2.0 **authentication** on Kafka listeners using the `custom` authentication type.
- Enable **authorization** using the `KeycloakAuthorizer`.
- Model resources, scopes, policies, and permissions in Keycloak according to your Kafka access-control requirements.

### NOTE

Keycloak examples for Strimzi, including a preconfigured realm and `Kafka` custom resource examples, are provided in the [example configuration files](#). For information on configuring realms, clients, and authorization services in Keycloak, see the [Keycloak documentation](#).

# Chapter 18. Managing TLS certificates

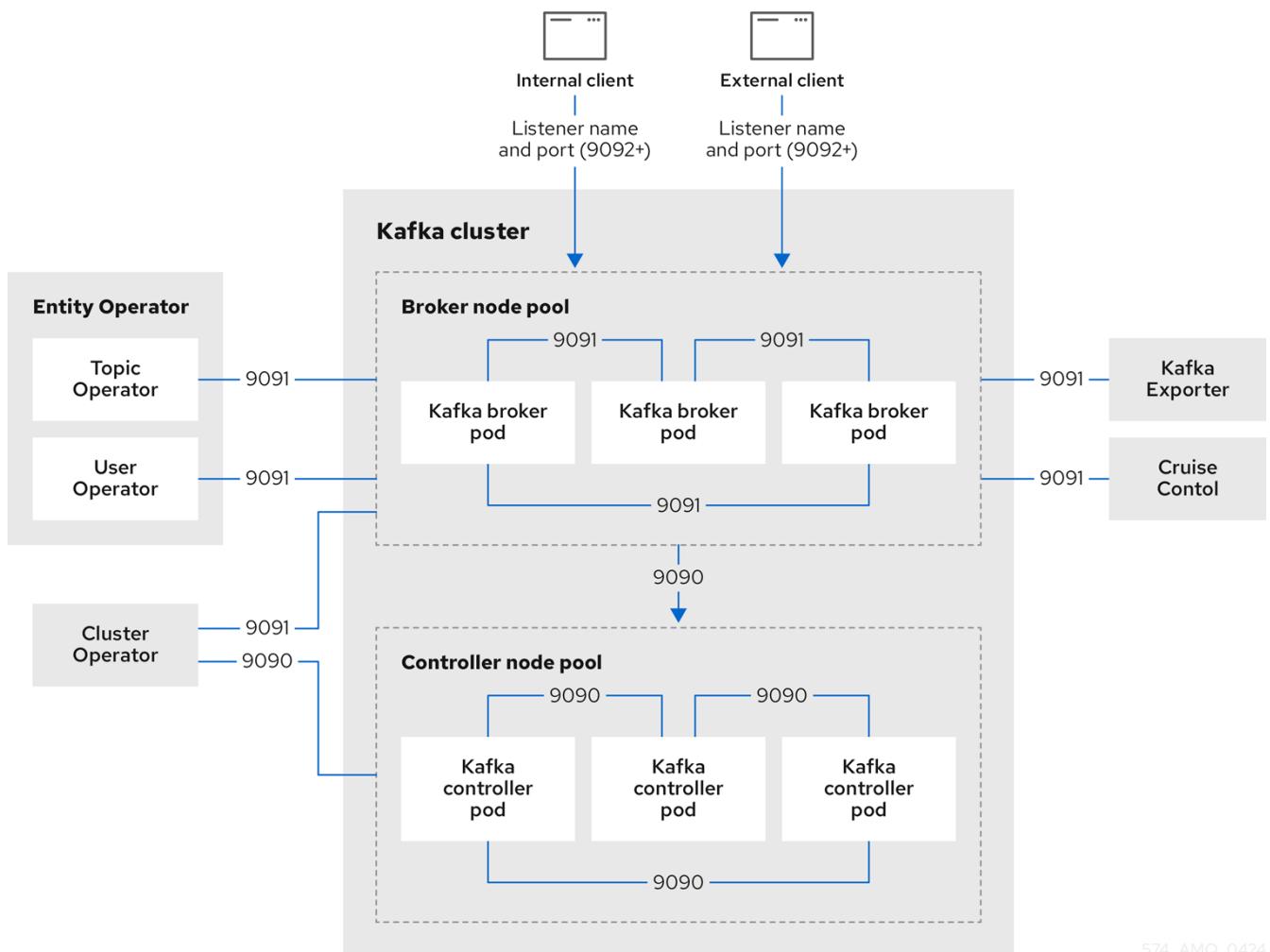
Strimzi supports TLS for encrypted communication between Kafka and Strimzi components.

Strimzi establishes encrypted TLS connections for communication between the following components when using Kafka in KRaft mode:

- Kafka brokers
- Kafka controllers
- Kafka brokers and controllers
- Strimzi operators and Kafka
- Cruise Control and Kafka brokers
- Kafka Exporter and Kafka brokers

Connections between clients and Kafka brokers use listeners that you must configure to use TLS-encrypted communication. You configure these listeners in the [Kafka](#) custom resource and each listener name and port number must be unique within the cluster. Communication between Kafka brokers and Kafka clients is encrypted according to how the `tls` property is configured for the listener. For more information, see [Setting up client access to a Kafka cluster](#).

The following diagram shows the connections for secure communication.



574\_AMQ\_0424

Figure 5. Kafka communication secured by TLS encryption

The ports shown in the diagram are used as follows:

#### Control plane listener (9090)

The internal control plane listener on port 9090 facilitates interbroker communication between Kafka controllers and broker-to-controller communication. Additionally, the Cluster Operator communicates with the controllers through the listener. This listener is not accessible to Kafka clients.

#### Replication listener (9091)

Data replication between brokers, as well as internal connections to the brokers from Strimzi operators, Cruise Control, and the Kafka Exporter, use the replication listener on port 9091. This listener is not accessible to Kafka clients.

#### Listeners for client connections (9092 or higher)

For TLS-encrypted communication (through configuration of the listener), internal and external clients connect to Kafka brokers. External clients (producers and consumers) connect to the Kafka brokers through the advertised listener port.

#### IMPORTANT

When configuring listeners for client access to brokers, you can use port 9092 or higher (9093, 9094, and so on), but with a few exceptions. The

listeners cannot be configured to use the ports reserved for interbroker communication (9090 and 9091), Prometheus metrics (9404), and JMX (Java Management Extensions) monitoring (9999).

#### *Node status monitoring using the [KafkaAgent](#) (8443)*

Strimzi includes a component called [KafkaAgent](#) that runs inside each Kafka node. The agent is responsible for collecting and providing node-specific information, such as current state and readiness, to the Cluster Operator. It listens on port 8443 for secure HTTPS connections and exposes this information through a REST API, which the Cluster Operator uses to retrieve data from the nodes.

## 18.1. Internal cluster CA and clients CA

To support encryption, each Strimzi component needs its own private keys and public key certificates. All component certificates are signed by an internal CA (certificate authority) called the *cluster CA*.

CA (Certificate Authority) certificates are generated by the Cluster Operator to verify the identities of components and clients.

Similarly, each Kafka client application connecting to Strimzi using mTLS needs to use private keys and certificates. A second internal CA, named the *clients CA*, is used to sign certificates for the Kafka clients.

Both the cluster CA and clients CA have a self-signed public key certificate.

Kafka brokers are configured to trust certificates signed by either the cluster CA or clients CA. Components that clients do not need to connect to only trust certificates signed by the cluster CA. Unless TLS encryption for external listeners is disabled, client applications must trust certificates signed by the cluster CA. This is also true for client applications that perform mTLS authentication.

By default, Strimzi automatically generates and renews CA certificates issued by the cluster CA or clients CA. You can configure the management of these CA certificates using [Kafka.spec.clusterCa](#) and [Kafka.spec.clientsCa](#) properties.

**NOTE**

If you don't want to use the CAs generated by the Cluster Operator, you can [install your own cluster and clients CA certificates](#). Any certificates you provide are not renewed by the Cluster Operator.

## 18.2. Secrets generated by the operators

The Cluster Operator automatically sets up and renews TLS certificates to enable encryption and authentication within a cluster. It also sets up other TLS certificates if you want to enable encryption or mTLS authentication between Kafka brokers and clients.

Secrets are created when custom resources are deployed, such as [Kafka](#) and [KafkaUser](#). Strimzi uses these secrets to store private and public key certificates for Kafka clusters, clients, and users. The secrets are used for establishing TLS encrypted connections between Kafka brokers, and between

brokers and clients. They are also used for mTLS authentication.

Cluster and clients secrets are always pairs: one contains the public key and one contains the private key.

### Cluster secret

A cluster secret contains the *cluster CA* to sign Kafka broker certificates. Connecting clients use the certificate to establish a TLS encrypted connection with a Kafka cluster. The certificate verifies broker identity.

### Client secret

A client secret contains the *clients CA* for a user to sign its own client certificate. This allows mutual authentication against the Kafka cluster. The broker validates a client's identity through the certificate.

### User secret

A user secret contains a private key and certificate. The secret is created and signed by the clients CA when a new user is created. The key and certificate are used to authenticate and authorize the user when accessing the cluster.

**NOTE** You can provide [Kafka listener certificates](#) for TLS listeners or external listeners that have TLS encryption enabled. Use Kafka listener certificates to incorporate the security infrastructure you already have in place.

## 18.2.1. TLS authentication using keys and certificates in PEM or PKCS #12 format

The secrets created by Strimzi provide private keys and certificates in PEM (Privacy Enhanced Mail) and PKCS #12 (Public-Key Cryptography Standards) formats. PEM and PKCS #12 are OpenSSL-generated key formats for TLS communications using the SSL protocol.

You can configure mutual TLS (mTLS) authentication that uses the credentials contained in the secrets generated for a Kafka cluster and user.

To set up mTLS, you must first do the following:

- [Configure your Kafka cluster with a listener that uses mTLS](#)
- [Create a KafkaUser that provides client credentials for mTLS](#)

When you deploy a Kafka cluster, a `<cluster_name>-cluster-ca-cert` secret is created with public key to verify the cluster. You use the public key to configure a truststore for the client.

When you create a `KafkaUser`, a `<kafka_user_name>` secret is created with the keys and certificates to verify the user (client). Use these credentials to configure a keystore for the client.

With the Kafka cluster and client set up to use mTLS, you extract credentials from the secrets and add them to your client configuration.

## PEM keys and certificates

For PEM, you add the following to your client configuration:

### Truststore

- `ca.crt` from the `<cluster_name>-cluster-ca-cert` secret, which is the CA certificate for the cluster.

### Keystore

- `user.crt` from the `<kafka_user_name>` secret, which is the public certificate of the user.
- `user.key` from the `<kafka_user_name>` secret, which is the private key of the user.

## PKCS #12 keys and certificates

For PKCS #12, you add the following to your client configuration:

### Truststore

- `ca.p12` from the `<cluster_name>-cluster-ca-cert` secret, which is the CA certificate for the cluster.
- `ca.password` from the `<cluster_name>-cluster-ca-cert` secret, which is the password to access the public cluster CA certificate.

### Keystore

- `user.p12` from the `<kafka_user_name>` secret, which is the public key certificate of the user.
- `user.password` from the `<kafka_user_name>` secret, which is the password to access the public key certificate of the Kafka user.

PKCS #12 is supported by Java, so you can add the values of the certificates directly to your Java client configuration. You can also reference the certificates from a secure storage location. With PEM files, you must add the certificates directly to the client configuration in single-line format. Choose a format that's suitable for establishing TLS connections between your Kafka cluster and client. Use PKCS #12 if you are unfamiliar with PEM.

**NOTE**

All keys are 2048 bits in size and, by default, are valid for 365 days from the initial generation. You can [change the validity period](#).

### 18.2.2. Secrets generated by the Cluster Operator

The Cluster Operator generates the following certificates, which are saved as secrets in the Kubernetes cluster. Strimzi uses these secrets by default.

The cluster CA and clients CA have separate secrets for the private key and public key.

#### `<cluster_name>-cluster-ca`

Contains the private key of the cluster CA. Strimzi and Kafka components use the private key to sign server certificates.

#### `<cluster_name>-cluster-ca-cert`

Contains the public key of the cluster CA. Kafka clients use the public key to verify the identity of the Kafka brokers they are connecting to with TLS server authentication.

#### `<cluster_name>-clients-ca`

Contains the private key of the clients CA. Kafka clients use the private key to sign new user certificates for mTLS authentication when connecting to Kafka brokers.

#### `<cluster_name>-clients-ca-cert`

Contains the public key of the clients CA. Kafka brokers use the public key to verify the identity of clients accessing the Kafka brokers when mTLS authentication is used.

Secrets for communication between Strimzi components contain a private key and a public key certificate signed by the cluster CA.

#### `<cluster_name>-<pool_name>-<pod_id>`

Contains the private and public keys for a Kafka node. Each pod has its own secret.

#### `<cluster_name>-cluster-operator-certs`

Contains the private and public keys for encrypting communication between the Cluster Operator and Kafka.

#### `<cluster_name>-entity-topic-operator-certs`

Contains the private and public keys for encrypting communication between the Topic Operator and Kafka.

#### `<cluster_name>-entity-user-operator-certs`

Contains the private and public keys for encrypting communication between the User Operator and Kafka.

#### `<cluster_name>-cruise-control-certs`

Contains the private and public keys for encrypting communication between Cruise Control and Kafka.

#### `<cluster_name>-kafka-exporter-certs`

Contains the private and public keys for encrypting communication between Kafka Exporter and Kafka.

**NOTE** You can [provide your own server certificates and private keys](#) to connect to Kafka brokers using *Kafka listener certificates* rather than certificates signed by the cluster CA.

### 18.2.3. Cluster CA secrets

Cluster CA secrets are managed by the Cluster Operator in a Kafka cluster.

Only the `<cluster_name>-cluster-ca-cert` secret is required by clients. All other cluster secrets are accessed by Strimzi components. You can enforce this using Kubernetes role-based access controls, if necessary.

**NOTE** The CA certificates in `<cluster_name>-cluster-ca-cert` must be trusted by Kafka client applications so that they validate the Kafka broker certificates when

connecting to Kafka brokers over TLS.

*Table 17. Fields in the <cluster\_name>-cluster-ca secret*

Field	Description
ca.key	The current private key for the cluster CA.

*Table 18. Fields in the <cluster\_name>-cluster-ca-cert secret*

Field	Description
ca.p12	PKCS #12 store for storing certificates and keys.
ca.password	Password for protecting the PKCS #12 store.
ca.crt	The current certificate for the cluster CA.

*Table 19. Fields in the <cluster\_name>-<pool\_name>-<node\_id> secret*

Field	Description
<cluster_name>-kafka-<pod_id>.crt	Certificate for a Kafka pod <num>. Signed by a current or former cluster CA private key in <cluster_name>-cluster-ca.
<cluster_name>-kafka-<pod_id>.key	Private key for a Kafka pod <num>.

*Table 20. Fields in the <cluster\_name>-cluster-operator-certs secret*

Field	Description
cluster-operator.p12	PKCS #12 store for storing certificates and keys.
cluster-operator.password	Password for protecting the PKCS #12 store.
cluster-operator.crt	Certificate for mTLS communication between the Cluster Operator and Kafka. Signed by a current or former cluster CA private key in <cluster_name>-cluster-ca.
cluster-operator.key	Private key for mTLS communication between the Cluster Operator and Kafka.

*Table 21. Fields in the <cluster\_name>-entity-topic-operator-certs secret*

Field	Description
entity-operator.p12	PKCS #12 store for storing certificates and keys.
entity-operator.password	Password for protecting the PKCS #12 store.
entity-operator.crt	Certificate for mTLS communication between the Topic Operator and Kafka. Signed by a current or former cluster CA private key in <cluster_name>-cluster-ca.
entity-operator.key	Private key for mTLS communication between the Topic Operator and Kafka.

*Table 22. Fields in the <cluster\_name>-entity-user-operator-certs secret*

Field	Description
entity-operator.p12	PKCS #12 store for storing certificates and keys.
entity-operator.password	Password for protecting the PKCS #12 store.
entity-operator.crt	Certificate for mTLS communication between the User Operator and Kafka. Signed by a current or former cluster CA private key in <cluster_name>-cluster-ca.
entity-operator.key	Private key for mTLS communication between the User Operator and Kafka.

Table 23. Fields in the <cluster\_name>-cruise-control-certs secret

Field	Description
cruise-control.p12	PKCS #12 store for storing certificates and keys.
cruise-control.password	Password for protecting the PKCS #12 store.
cruise-control.crt	Certificate for mTLS communication between Cruise Control and Kafka. Signed by a current or former cluster CA private key in <cluster_name>-cluster-ca.
cruise-control.key	Private key for mTLS communication between the Cruise Control and Kafka.

Table 24. Fields in the <cluster\_name>-kafka-exporter-certs secret

Field	Description
kafka-exporter.p12	PKCS #12 store for storing certificates and keys.
kafka-exporter.password	Password for protecting the PKCS #12 store.
kafka-exporter.crt	Certificate for mTLS communication between Kafka Exporter and Kafka. Signed by a current or former cluster CA private key in <cluster_name>-cluster-ca.
kafka-exporter.key	Private key for mTLS communication between the Kafka Exporter and Kafka.

## 18.2.4. Clients CA secrets

Clients CA secrets are managed by the Cluster Operator in a Kafka cluster.

The certificates in <cluster\_name>-clients-ca-cert are those which the Kafka brokers trust.

The <cluster\_name>-clients-ca secret is used to sign the certificates of client applications. This secret must be accessible to the Strimzi components and for administrative access if you are intending to issue application certificates without using the User Operator. You can enforce this using Kubernetes role-based access controls, if necessary.

Table 25. Fields in the <cluster\_name>-clients-ca secret

Field	Description
<code>ca.key</code>	The current private key for the clients CA.

Table 26. Fields in the `<cluster_name>-clients-ca-cert` secret

Field	Description
<code>ca.p12</code>	PKCS #12 store for storing certificates and keys.
<code>ca.password</code>	Password for protecting the PKCS #12 store.
<code>ca.crt</code>	The current certificate for the clients CA.

## 18.2.5. User secrets generated by the User Operator

User secrets are managed by the User Operator.

When a user is created using the User Operator, a secret is generated using the name of the user.

Table 27. Fields in the `user_name` secret

Secret name	Field within secret	Description
<code>&lt;user_name&gt;</code>	<code>user.p12</code>	PKCS #12 store for storing certificates and keys.
	<code>user.password</code>	Password for protecting the PKCS #12 store.
	<code>user.crt</code>	Certificate for the user, signed by the clients CA
	<code>user.key</code>	Private key for the user

## 18.2.6. Adding labels and annotations to cluster CA secrets

By configuring the `clusterCaCert` template property in the `Kafka` custom resource, you can add custom labels and annotations to the Cluster CA secrets created by the Cluster Operator. Labels and annotations are useful for identifying objects and adding contextual information. You configure template properties in Strimzi custom resources.

*Example template customization to add labels and annotations to secrets*

```
apiVersion: kafka.strimzi.io/v1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    template:
      clusterCaCert:
        metadata:
          labels:
```

```
label1: value1
label2: value2
annotations:
  annotation1: value1
  annotation2: value2
# ...
```

### 18.2.7. Disabling `ownerReference` in the CA secrets

By default, the cluster and clients CA secrets are created with an `ownerReference` property that is set to the `Kafka` custom resource. This means that, when the `Kafka` custom resource is deleted, the CA secrets are also deleted (garbage collected) by Kubernetes.

If you want to reuse the CA for a new cluster, you can disable the `ownerReference` by setting the `generateSecretOwnerReference` property for the cluster and clients CA secrets to `false` in the `Kafka` configuration. When the `ownerReference` is disabled, CA secrets are not deleted by Kubernetes when the corresponding `Kafka` custom resource is deleted.

*Example Kafka configuration with disabled `ownerReference` for cluster and clients CAs*

```
apiVersion: kafka.strimzi.io/v1
kind: Kafka
# ...
spec:
# ...
  clusterCa:
    generateSecretOwnerReference: false
  clientsCa:
    generateSecretOwnerReference: false
# ...
```

*Additional resources*

- [CertificateAuthority schema reference](#)

## 18.3. Certificate renewal and validity periods

Cluster CA and clients CA certificates are valid for a limited time, known as the validity period. This is defined as the number of days from the date the certificate was generated.

For CA certificates automatically created by the Cluster Operator, configure the validity period for certificates in the `kafka` resource:

- `Kafka.spec.clusterCa.validityDays` for Cluster CA certificates
- `Kafka.spec.clientsCa.validityDays` for Clients CA certificates

The default validity period for both certificates is 365 days. For manually-installed custom CA certificates, set validity through your certificate management system.

When a CA certificate expires, components and clients that still trust the old certificate do not accept connections from peers whose certificates were signed by the CA private key. The components and clients must trust the *new* CA certificate instead.

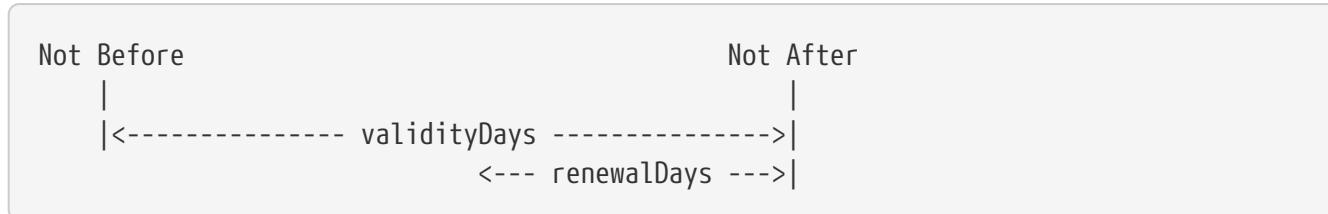
To prevent loss of service, the Cluster Operator initiates certificate renewal before the old CA certificates expire.

Configure the renewal period of the certificates created by the Cluster Operator in the `kafka` resource using the following properties:

- `Kafka.spec.clusterCa.renewalDays` for Cluster CA certificates
- `Kafka.spec.clientsCa.renewalDays` for Clients CA certificates

The default renewal period for both certificates is 30 days from the expiry date of the current certificate. Changing the `validityDays` does not trigger immediate certificate renewal. The updated value is applied the next time the certificate is renewed, either automatically or through manual renewal.

*Validity period against renewal period*



Changes to `renewalDays` may trigger renewal earlier if the new value places the certificate within the renewal window. To schedule the renewal period at a convenient time, use [maintenance time windows](#).

**IMPORTANT**

`maintenanceTimeWindows` apply **only** to certificates generated automatically by the Cluster Operator. They do **not** apply to custom or externally managed certificates, so restarts triggered by updates to those certificates may occur outside the defined windows. With a custom Certificate Authority (CA), the Cluster Operator still manages the validity and renewal of the server certificates it generates. In this case, `validityDays` and `renewalDays` apply to those server certificates, not to the CA itself.

To change validity and renewal periods after creating the Kafka cluster:

1. Modify the `Kafka` custom resource.
2. [Manually renew the CA certificates](#).

If you do not manually renew the certificates, the new settings take effect the next time the certificate is renewed automatically.

*Example Kafka configuration for certificate validity and renewal periods*

```
apiVersion: kafka.strimzi.io/v1
kind: Kafka
```

```

# ...
spec:
# ...
  clusterCa:
    renewalDays: 30
    validityDays: 365
    generateCertificateAuthority: true
  clientsCa:
    renewalDays: 30
    validityDays: 365
    generateCertificateAuthority: true
# ...

```

Automatic certificate renewal depends on the `generateCertificateAuthority` setting:

- If `true`, the Cluster Operator handles renewal.
- If `false`, certificates must be managed externally.  
Use this option if you are [installing your own certificates](#).

### 18.3.1. Cluster CA renewal

To renew the Cluster CA certificate, the Cluster Operator does the following:

1. Generates a new Cluster CA certificate, retaining the existing private key.

The renewed certificate replaces `ca.crt` in the Cluster CA secret.

2. Regenerates internal client certificates for the following components:

- Kafka nodes
- Entity Operator (Topic Operator and User Operator)
- Kafka Exporter
- Cruise Control

These new certificates are not strictly required, because the signing key hasn't changed. However, the Cluster Operator regenerates them to align their validity period with the new CA certificate.

3. Restarts the components to trust the new Cluster CA certificate and use the renewed internal certificates.

### 18.3.2. Clients CA renewal

To renew the Clients CA certificate, the Cluster Operator and User Operator each perform part of the process:

1. The Cluster Operator generates a new Clients CA certificate, retaining the existing private key.

The renewed certificate replaces `ca.crt` in the Clients CA secret.

2. The User Operator detects the updated Clients CA certificate and regenerates the user certificates that are signed by it.

**IMPORTANT**

After renewal, you must ensure client applications update their truststores and keystores with the renewed user certificates before the old ones expire to avoid connection failures.

### 18.3.3. Managing certificate renewal for client applications

The Strimzi operators do not manage external client applications. You are responsible for ensuring that clients continue to connect successfully after certificate renewal.

When the Clients CA is renewed, the User Operator automatically regenerates user certificates. Client applications must be updated to use these renewed credentials before the old certificates expire.

Client applications must be configured with the following:

- A truststore that includes credentials from the `<cluster_name>-cluster-ca-cert` secret, which is created by the Cluster Operator and contains the public key to verify the Kafka cluster.
- A keystore built from the `<kafka_user_name>` secret, which is created by the User Operator and contains the user's certificate and key.

User secrets provide credentials in PEM and PKCS #12 formats, or a password if using SCRAM-SHA authentication. The User Operator creates these secrets when a user is created. For an example of configuring secure clients, see [Securing user access to Kafka](#).

If you provision client certificates manually, generate and distribute new certificates before the current ones expire. Failure to do so can result in clients being unable to connect to the Kafka cluster.

**NOTE**

For workloads in the same Kubernetes cluster and namespace, you can mount secrets as volumes. This allows client pods to construct keystores and truststores dynamically from the current state of the secrets. For details, see [Configuring internal clients to trust the cluster CA](#).

### 18.3.4. Scheduling maintenance time windows

Use maintenance time windows to control when the Cluster Operator performs certificate renewals and the related rolling restarts. This helps minimize disruption to Kafka clients by scheduling updates at convenient times.

Maintenance time windows apply only to automatic certificate renewals and rolling restarts managed by the Cluster Operator.

They **apply** in the following scenarios:

- Automatic time-based renewal of Strimzi-managed internal certificates, such as those used by Kafka nodes.

- Automatic time-based renewal of server certificates signed by a custom Certificate Authority (CA), if Strimzi manages the renewal process based on certificate expiry.

They **do not apply** in the following scenarios:

- User-driven changes to the [Kafka](#) custom resource, including configuration updates.
- Environment-driven changes to certificate fields, such as the Common Name (CN) or Subject Alternative Names (SANs). For example, when a new load balancer address is introduced.
- Manual updates to Cluster CA or Clients CA certificates, even if they are Strimzi-managed, since these updates are treated as user-initiated actions.
- Externally managed certificates, such as:
  - Custom listener certificates provided directly by the user.
  - Certificates issued by an external certificate manager.

In these cases, restarts may occur immediately and outside any configured maintenance time window.

Use time windows in conjunction with the [renewal periods for CA certificates](#) set in the [Kafka](#) resource ([Kafka.spec.clusterCa.renewalDays](#) and [Kafka.spec.clientsCa.renewalDays](#)).

Rolling restarts can be triggered in two ways:

- By user-driven changes to the [Kafka](#) custom resource
- Automatically, when Cluster Operator-managed certificates near expiration

User-driven restarts are not restricted by maintenance time windows. However, automatically triggered restarts for expiring certificates are subject to the configured maintenance windows.

Although restarts typically do not affect service availability, they can temporarily impact client performance. Use maintenance time windows to schedule automatic restarts occur during periods of low client activity.

Configure maintenance time windows as follows:

- Configure an array of strings using the [Kafka.spec.maintenanceTimeWindows](#) property of the [Kafka](#) resource.
- Each string is a [cron expression](#) interpreted as being in UTC (Coordinated Universal Time)

The following example configures a single maintenance time window that starts at midnight and ends at 01:59am (UTC), on Sundays, Mondays, Tuesdays, Wednesdays, and Thursdays.

*Example maintenance time window configuration*

```
apiVersion: kafka.strimzi.io/v1
kind: Kafka
metadata:
  name: my-cluster
spec:
```

```
kafka:  
  #...  
  maintenanceTimeWindows:  
    - "* * 0-1 ? * SUN,MON,TUE,WED,THU *"  
  #...
```

**NOTE**

The Cluster Operator doesn't adhere strictly to the time windows. Maintenance operations are triggered by the first reconciliation that occurs within the specified time window. If the time window is shorter than the interval between reconciliations, there's a risk that the reconciliation may happen outside of the time window. Therefore, maintenance time windows must be at least as long as the interval between reconciliations.

### 18.3.5. Manually renewing Cluster Operator-managed CA certificates

Cluster and clients CA certificates generated by the Cluster Operator auto-renew at the start of their respective certificate renewal periods. However, you can use the [strimzi.io/force-renew](#) annotation to manually renew one or both of these certificates before the certificate renewal period starts. You might do this for security reasons, or if you have [changed the renewal or validity periods for the certificates](#).

A renewed certificate uses the same private key as the old certificate.

**NOTE**

If you are using your own CA certificates, the [force-renew](#) annotation cannot be used. Instead, follow the procedure for [renewing your own CA certificates](#).

#### Prerequisites

- [The Cluster Operator must be deployed](#).
- A Kafka cluster in which CA certificates and private keys are installed.
- The OpenSSL TLS management tool to check the period of validity for CA certificates.

In this procedure, we use a Kafka cluster named [my-cluster](#) within the [my-project](#) namespace.

#### Procedure

1. Apply the [strimzi.io/force-renew](#) annotation to the secret that contains the CA certificate that you want to renew.

##### *Renewing the Cluster CA secret*

```
kubectl annotate secret my-cluster-cluster-ca-cert -n my-project strimzi.io/force-renew="true"
```

##### *Renewing the Clients CA secret*

```
kubectl annotate secret my-cluster-clients-ca-cert -n my-project strimzi.io/force-renew="true"
```

- At the next reconciliation, the Cluster Operator generates new certificates.

If maintenance time windows are configured, the Cluster Operator generates the new CA certificate at the first reconciliation within the next maintenance time window.

- Check the period of validity for the new CA certificates.

*Checking the period of validity for the new cluster CA certificate*

```
kubectl get secret my-cluster-cluster-ca-cert -n my-project  
-o=jsonpath='{.data.ca\.crt}' | base64 -d | openssl x509 -noout -dates
```

*Checking the period of validity for the new clients CA certificate*

```
kubectl get secret my-cluster-clients-ca-cert -n my-project  
-o=jsonpath='{.data.ca\.crt}' | base64 -d | openssl x509 -noout -dates
```

The command returns a `notBefore` and `notAfter` date, which is the valid start and end date for the CA certificate.

- Update client configurations to trust the new cluster CA certificate.

See:

- Configuring internal clients to trust the cluster CA
- Configuring external clients to trust the cluster CA

### 18.3.6. Manually recovering from expired Cluster Operator-managed CA certificates

The Cluster Operator automatically renews the cluster and clients CA certificates when their renewal periods begin. Nevertheless, unexpected operational problems or disruptions may prevent the renewal process, such as prolonged downtime of the Cluster Operator or unavailability of the Kafka cluster. If CA certificates expire, Kafka cluster components cannot communicate with each other and the Cluster Operator cannot renew the CA certificates without manual intervention.

To promptly perform a recovery, follow the steps outlined in this procedure in the order given. You can recover from expired cluster and clients CA certificates. The process involves deleting the secrets containing the expired certificates so that new ones are generated by the Cluster Operator. For more information on the secrets managed in Strimzi, see [Secrets generated by the Cluster Operator](#).

**NOTE** If you are using your own CA certificates and they expire, the process is similar, but you need to [renew the CA certificates](#) rather than use certificates generated by the Cluster Operator.

#### Prerequisites

- The Cluster Operator must be deployed.

- A Kafka cluster in which CA certificates and private keys are installed.
- The OpenSSL TLS management tool to check the period of validity for CA certificates.

In this procedure, we use a Kafka cluster named `my-cluster` within the `my-project` namespace.

#### Procedure

1. Delete the secret containing the expired CA certificate.

##### *Deleting the Cluster CA secret*

```
kubectl delete secret my-cluster-cluster-ca-cert -n my-project
```

##### *Deleting the Clients CA secret*

```
kubectl delete secret my-cluster-clients-ca-cert -n my-project
```

2. Wait for the Cluster Operator to generate new certificates.

- A new CA cluster certificate to verify the identity of the Kafka brokers is created in a secret of the same name (`my-cluster-cluster-ca-cert`).
- A new CA clients certificate to verify the identity of Kafka users is created in a secret of the same name (`my-cluster-clients-ca-cert`).

3. Check the period of validity for the new CA certificates.

##### *Checking the period of validity for the new cluster CA certificate*

```
kubectl get secret my-cluster-cluster-ca-cert -n my-project  
-o=jsonpath='{.data.ca\.crt}' | base64 -d | openssl x509 -noout -dates
```

##### *Checking the period of validity for the new clients CA certificate*

```
kubectl get secret my-cluster-clients-ca-cert -n my-project  
-o=jsonpath='{.data.ca\.crt}' | base64 -d | openssl x509 -noout -dates
```

The command returns a `notBefore` and `notAfter` date, which is the valid start and end date for the CA certificate.

4. Delete the component pods and secrets that use the CA certificates.

- a. Delete the Kafka secret.
- b. Wait for the Cluster Operator to detect the missing Kafka secret and recreate it.
- c. Delete all Kafka pods.

If you are only recovering the clients CA certificate, you only need to delete the Kafka secret and pods.

You can use the following `kubectl` command to find resources and also verify that they have

been removed.

```
kubectl get <resource_type> --all-namespaces | grep <kafka_cluster_name>
```

Replace `<resource_type>` with the type of the resource, such as `Pod` or `Secret`.

5. Wait for the Cluster Operator to detect the missing Kafka pods and recreate them with the updated CA certificates.

On reconciliation, the Cluster Operator automatically updates other components to trust the new CA certificates.

6. Verify that there are no issues related to certificate validation in the Cluster Operator log.
7. Update client configurations to trust the new cluster CA certificate.

See:

- [Configuring internal clients to trust the cluster CA](#)
- [Configuring external clients to trust the cluster CA](#)

### 18.3.7. Replacing private keys used by Cluster Operator-managed CA certificates

You can replace the private keys used by the cluster CA and clients CA certificates generated by the Cluster Operator. When a private key is replaced, the Cluster Operator generates a new CA certificate for the new private key.

**NOTE**

If you are using your own CA certificates, the `force-replace` annotation cannot be used. Instead, follow the procedure for [renewing your own CA certificates](#).

#### Prerequisites

- The Cluster Operator is running.
- A Kafka cluster in which CA certificates and private keys are installed.

#### Procedure

- Apply the `strimzi.io/force-replace` annotation to the `Secret` that contains the private key that you want to renew.

*Table 28. Commands for replacing private keys*

Private key for	Secret	Annotate command
Cluster CA	<code>&lt;cluster_name&gt;-cluster-ca</code>	<code>kubectl annotate secret &lt;cluster_name&gt;-cluster-ca strimzi.io/force-replace="true"</code>

Private key for	Secret	Annotate command
Clients CA	<cluster_name>-clients-ca	<code>kubectl annotate secret &lt;cluster_name&gt;-clients-ca strimzi.io/force-replace="true"</code>

At the next reconciliation the Cluster Operator will:

- Generate a new private key for the **Secret** that you annotated
- Generate a new CA certificate

If maintenance time windows are configured, the Cluster Operator will generate the new private key and CA certificate at the first reconciliation within the next maintenance time window.

Client applications must reload the cluster and clients CA certificates that were renewed by the Cluster Operator.

#### *Additional resources*

- [Secrets generated by the operators](#)
- [Scheduling maintenance time windows](#)

## 18.4. Configuring internal clients to trust the cluster CA

This procedure describes how to configure a Kafka client that resides inside the Kubernetes cluster — connecting to a TLS listener — to trust the cluster CA certificate.

The easiest way to achieve this for an internal client is to use a volume mount to access the **Secrets** containing the necessary certificates and keys.

Follow the steps to configure trust certificates that are signed by the cluster CA for Java-based Kafka Producer, Consumer, and Streams APIs.

Choose the steps to follow according to the certificate format of the cluster CA: PKCS #12 ([.p12](#)) or PEM ([.crt](#)).

The steps describe how to mount the Cluster Secret that verifies the identity of the Kafka cluster to the client pod.

#### *Prerequisites*

- The Cluster Operator must be running.
- There needs to be a **Kafka** resource within the Kubernetes cluster.
- You need a Kafka client application inside the Kubernetes cluster that will connect using TLS, and needs to trust the cluster CA certificate.
- The client application must be running in the same namespace as the **Kafka** resource.

#### *Using PKCS #12 format (.p12)*

## 1. Mount the cluster Secret as a volume when defining the client pod.

For example:

```
kind: Pod
apiVersion: v1
metadata:
  name: client-pod
spec:
  containers:
    - name: client-name
      image: client-name
      volumeMounts:
        - name: secret-volume
          mountPath: /data/p12
      env:
        - name: SECRET_PASSWORD
          valueFrom:
            secretKeyRef:
              name: my-secret
              key: my-password
      volumes:
        - name: secret-volume
          secret:
            secretName: my-cluster-cluster-ca-cert
```

Here we're mounting the following:

- The PKCS #12 file into an exact path, which can be configured
- The password into an environment variable, where it can be used for Java configuration

## 2. Configure the Kafka client with the following properties:

- A security protocol option:

- **security.protocol: SSL** when using TLS for encryption (with or without mTLS authentication).
- **security.protocol: SASL\_SSL** when using SCRAM-SHA authentication over TLS.
- **ssl.truststore.location** with the truststore location where the certificates were imported.
- **ssl.truststore.password** with the password for accessing the truststore.
- **ssl.truststore.type=PKCS12** to identify the truststore type.

*Using PEM format (.crt)*

## 1. Mount the cluster Secret as a volume when defining the client pod.

For example:

```
kind: Pod
```

```

apiVersion: v1
metadata:
  name: client-pod
spec:
  containers:
    - name: client-name
      image: client-name
      volumeMounts:
        - name: secret-volume
          mountPath: /data/crt
  volumes:
    - name: secret-volume
      secret:
        secretName: my-cluster-cluster-ca-cert

```

2. Use the extracted certificate to configure a TLS connection in clients that use certificates in X.509 format.

## 18.5. Configuring external clients to trust the cluster CA

This procedure describes how to configure a Kafka client that resides outside the Kubernetes cluster – connecting to an [external](#) listener – to trust the cluster CA certificate. Follow this procedure when setting up the client and during the renewal period, when the old clients CA certificate is replaced.

Follow the steps to configure trust certificates that are signed by the cluster CA for Java-based Kafka Producer, Consumer, and Streams APIs.

Choose the steps to follow according to the certificate format of the cluster CA: PKCS #12 ([.p12](#)) or PEM ([.crt](#)).

The steps describe how to obtain the certificate from the Cluster Secret that verifies the identity of the Kafka cluster.

### IMPORTANT

The `<cluster_name>-cluster-ca-cert` secret contains more than one CA certificate during the CA certificate renewal period. Clients must add *all* of them to their truststores.

#### *Prerequisites*

- The Cluster Operator must be running.
- There needs to be a [Kafka](#) resource within the Kubernetes cluster.
- You need a Kafka client application outside the Kubernetes cluster that will connect using TLS, and needs to trust the cluster CA certificate.

#### *Using PKCS #12 format (.p12)*

1. Extract the cluster CA certificate and password from the `<cluster_name>-cluster-ca-cert` Secret

of the Kafka cluster.

```
kubectl get secret <cluster_name>-cluster-ca-cert -o jsonpath='{.data.ca\.p12}' |  
base64 -d > ca.p12
```

```
kubectl get secret <cluster_name>-cluster-ca-cert -o  
jsonpath='{.data.ca\.password}' | base64 -d > ca.password
```

Replace *<cluster\_name>* with the name of the Kafka cluster.

## 2. Configure the Kafka client with the following properties:

- A security protocol option:
  - **security.protocol: SSL** when using TLS.
  - **security.protocol: SASL\_SSL** when using SCRAM-SHA authentication over TLS.
- **ssl.truststore.location** with the truststore location where the certificates were imported.
- **ssl.truststore.password** with the password for accessing the truststore. This property can be omitted if it is not needed by the truststore.
- **ssl.truststore.type=PKCS12** to identify the truststore type.

*Using PEM format (.crt)*

## 1. Extract the cluster CA certificate from the **<cluster\_name>-cluster-ca-cert** secret of the Kafka cluster.

```
kubectl get secret <cluster_name>-cluster-ca-cert -o jsonpath='{.data.ca\.crt}' |  
base64 -d > ca.crt
```

## 2. Use the extracted certificate to configure a TLS connection in clients that use certificates in X.509 format.

## 18.6. Using your own CA certificates and private keys

Install and use your own CA certificates and private keys instead of using the defaults generated by the Cluster Operator. You can replace the cluster and clients CA certificates and private keys.

You can switch to using your own CA certificates and private keys in the following ways:

- Install your own CA certificates and private keys before deploying your Kafka cluster
- Replace the default CA certificates and private keys with your own after deploying a Kafka cluster

The steps to replace the default CA certificates and private keys after deploying a Kafka cluster are the same as those used to renew your own CA certificates and private keys.

If you use your own certificates, they won't be renewed automatically. You need to renew the CA certificates and private keys before they expire.

Renewal options:

- Renew the CA certificates only
- Renew CA certificates and private keys (or replace the defaults)

### 18.6.1. Installing your own CA certificates and private keys

Install your own CA certificates and private keys instead of using the cluster and clients CA certificates and private keys generated by the Cluster Operator.

By default, Strimzi uses the following [cluster CA and clients CA secrets](#), which are renewed automatically.

- Cluster CA secrets
  - `<cluster_name>-cluster-ca`
  - `<cluster_name>-cluster-ca-cert`
- Clients CA secrets
  - `<cluster_name>-clients-ca`
  - `<cluster_name>-clients-ca-cert`

To install your own certificates, use the same names.

#### *Prerequisites*

- The Cluster Operator is running.
- A Kafka cluster is not yet deployed.

If you have already deployed a Kafka cluster, you can [replace the default CA certificates with your own](#).

- Your own X.509 certificates and keys in PEM format for the cluster CA or clients CA.
  - If you want to use a cluster or clients CA which is not a Root CA, you have to include the whole chain in the certificate file. The chain should be in the following order:
    1. The cluster or clients CA
    2. One or more intermediate CAs
    3. The root CA
    - All CAs in the chain should be configured using the X509v3 Basic Constraints extension. Basic Constraints limit the path length of a certificate chain.
- The OpenSSL TLS management tool for converting certificates.

#### *Before you begin*

The Cluster Operator generates keys and certificates in PEM (Privacy Enhanced Mail) and PKCS #12

(Public-Key Cryptography Standards) formats. Only the keys and certificates in the PEM format are used internally by Strimzi. The PKCS #12 store is there only for user applications that do not support using the PEM format directly. When using custom CA certificates, adding the PKCS #12 store and its password to the secret is optional only.

#### Procedure

1. Create a new secret that contains the CA certificate.

*Client secret creation with a certificate in PEM format only*

```
kubectl create secret generic <cluster_name>-clients-ca-cert --from-file=ca.crt=ca.crt
```

*Cluster secret creation with certificates in PEM and PKCS #12 format*

```
kubectl create secret generic <cluster_name>-cluster-ca-cert \
--from-file=ca.crt=ca.crt \
--from-file=ca.p12=ca.p12 \
--from-literal=ca.password=P12-PASSWORD
```

Replace <cluster\_name> with the name of your Kafka cluster.

2. Create a new secret that contains the private key.

```
kubectl create secret generic <ca_key_secret> --from-file=ca.key=ca.key
```

3. Label the secrets.

```
kubectl label secret <ca_certificate_secret> strimzi.io/kind=Kafka
strimzi.io/cluster="<cluster_name>"
```

```
kubectl label secret <ca_key_secret> strimzi.io/kind=Kafka
strimzi.io/cluster="<cluster_name>"
```

- Label `strimzi.io/kind=Kafka` identifies the Kafka custom resource.
- Label `strimzi.io/cluster="<cluster_name>"` identifies the Kafka cluster.

4. Annotate the secrets

```
kubectl annotate secret <ca_certificate_secret> strimzi.io/ca-cert-generation="<ca_certificate_generation>"
```

```
kubectl annotate secret <ca_key_secret> strimzi.io/ca-key-
```

```
generation="<ca_key_generation>"
```

- Annotation `strimzi.io/ca-cert-generation="<ca_certificate_generation>"` defines the generation of a new CA certificate.
- Annotation `strimzi.io/ca-key-generation="<ca_key_generation>"` defines the generation of a new CA key.

Start from 0 (zero) as the incremental value (`strimzi.io/ca-cert-generation=0`) for your own CA certificate. Set a higher incremental value when you renew the certificates.

5. Create the `Kafka` resource for your cluster, configuring either the `Kafka.spec.clusterCa` or the `Kafka.spec.clientsCa` object to *not* use generated CAs.

*Example fragment Kafka resource configuring the cluster CA to use certificates you supply for yourself*

```
kind: Kafka
version: kafka.strimzi.io/v1
spec:
  # ...
  clusterCa:
    generateCertificateAuthority: false
```

#### *Additional resources*

- [Renewing your own CA certificates](#)
- [Renewing or replacing CA certificates and private keys with your own](#)
- [Using custom listener certificates for TLS encryption](#)

### **18.6.2. Renewing your own CA certificates**

If you are using your own CA certificates, you need to renew them manually. The Cluster Operator will not renew them automatically. Renew the CA certificates in the renewal period before they expire.

Perform the steps in this procedure when you are renewing CA certificates and continuing with the same private key. If you are renewing your own CA certificates *and* private keys, see [Renewing or replacing CA certificates and private keys with your own](#).

The procedure describes the renewal of CA certificates in PEM format.

#### *Prerequisites*

- The Cluster Operator is running.
- You have new cluster or clients X.509 certificates in PEM format.

#### *Procedure*

1. Update the `Secret` for the CA certificate.

Edit the existing secret to add the new CA certificate and update the certificate generation

annotation value.

```
kubectl edit secret <ca_certificate_secret_name>
```

`<ca_certificate_secret_name>` is the name of the `Secret`, which is `<kafka_cluster_name>-cluster-ca-cert` for the cluster CA certificate and `<kafka_cluster_name>-clients-ca-cert` for the clients CA certificate.

The following example shows a secret for a cluster CA certificate that's associated with a Kafka cluster named `my-cluster`.

*Example secret configuration for a cluster CA certificate*

```
apiVersion: v1
kind: Secret
data:
  ca.crt: LS0tLS1CRUdJTiBDRVJUSUZJQ0F... ①
metadata:
  annotations:
    strimzi.io/ca-cert-generation: "0" ②
  labels:
    strimzi.io/cluster: my-cluster
    strimzi.io/kind: Kafka
  name: my-cluster-cluster-ca-cert
  #...
type: Opaque
```

① Current base64-encoded CA certificate

② Current CA certificate generation annotation value

2. Encode your new CA certificate into base64.

```
cat <path_to_new_certificate> | base64
```

3. Update the CA certificate.

Copy the base64-encoded CA certificate from the previous step as the value for the `ca.crt` property under `data`.

4. Increase the value of the CA certificate generation annotation.

Update the `strimzi.io/ca-cert-generation` annotation with a higher incremental value. For example, change `strimzi.io/ca-cert-generation=0` to `strimzi.io/ca-cert-generation=1`. If the `Secret` is missing the annotation, the value is treated as `0`, so add the annotation with a value of `1`.

When Strimzi generates certificates, the certificate generation annotation is automatically incremented by the Cluster Operator. For your own CA certificates, set the annotations with a

higher incremental value. The annotation needs a higher value than the one from the current secret so that the Cluster Operator can roll the pods and update the certificates. The `strimzi.io/ca-cert-generation` has to be incremented on each CA certificate renewal.

5. Save the secret with the new CA certificate and certificate generation annotation value.

*Example secret configuration updated with a new CA certificate*

```
apiVersion: v1
kind: Secret
data:
  ca.crt: GCa6LS3RTHeKFiFDGBOUDYFAZ0F... ①
metadata:
  annotations:
    strimzi.io/ca-cert-generation: "1" ②
  labels:
    strimzi.io/cluster: my-cluster
    strimzi.io/kind: Kafka
    name: my-cluster-cluster-ca-cert
    #...
  type: Opaque
```

① New base64-encoded CA certificate

② New CA certificate generation annotation value

On the next reconciliation, the Cluster Operator performs a rolling update of Kafka and other components to trust the new CA certificate.

If maintenance time windows are configured, the Cluster Operator will roll the pods at the first reconciliation within the next maintenance time window.

### 18.6.3. Renewing or replacing CA certificates and private keys with your own

If you are using your own CA certificates and private keys, you need to renew them manually. The Cluster Operator will not renew them automatically. Renew the CA certificates in the renewal period before they expire. You can also use the same procedure to replace the CA certificates and private keys generated by the Strimzi operators with your own.

Perform the steps in this procedure when you are renewing or replacing CA certificates and private keys. If you are only renewing your own CA certificates, see [Renewing your own CA certificates](#).

The procedure describes the renewal of CA certificates and private keys in PEM format.

Before going through the following steps, make sure that the CN (Common Name) of the new CA certificate is different from the current one. For example, when the Cluster Operator renews certificates automatically it adds a `v<version_number>` suffix to identify a version. Do the same with your own CA certificate by adding a different suffix on each renewal. By using a different key to generate a new CA certificate, you retain the current CA certificate stored in the `Secret`.

## Prerequisites

- The Cluster Operator is running.
- You have new cluster or clients X.509 certificates and keys in PEM format.

## Procedure

1. Pause the reconciliation of the **Kafka** custom resource.
  - a. Annotate the custom resource in Kubernetes, setting the **pause-reconciliation** annotation to **true**:

```
kubectl annotate Kafka <name_of_custom_resource> strimzi.io/pause-reconciliation="true"
```

For example, for a **Kafka** custom resource named **my-cluster**:

```
kubectl annotate Kafka my-cluster strimzi.io/pause-reconciliation="true"
```

- b. Check that the status conditions of the custom resource show a change to **ReconciliationPaused**:

```
kubectl describe Kafka <name_of_custom_resource>
```

The **type** condition changes to **ReconciliationPaused** at the **lastTransitionTime**.

2. Check the settings for the **generateCertificateAuthority** properties in your **Kafka** custom resource.

If a property is set to **false**, a CA certificate is not generated by the Cluster Operator. You require this setting if you are using your own certificates.

3. If needed, edit the existing **Kafka** custom resource and set the **generateCertificateAuthority** properties to **false**.

```
kubectl edit Kafka <name_of_custom_resource>
```

The following example shows a **Kafka** custom resource with both cluster and clients CA certificates generation delegated to the user.

*Example Kafka configuration using your own CA certificates*

```
apiVersion: kafka.strimzi.io/v1
kind: Kafka
# ...
spec:
# ...
clusterCa:
```

```
    generateCertificateAuthority: false ①
  clientsCa:
    generateCertificateAuthority: false ②
# ...
```

① Use your own cluster CA

② Use your own clients CA

#### 4. Update the `Secret` for the CA certificate.

- Edit the existing secret to add the new CA certificate and update the certificate generation annotation value.

```
kubectl edit secret <ca_certificate_secret_name>
```

`<ca_certificate_secret_name>` is the name of the `Secret`, which is `<kafka_cluster_name>-cluster-ca-cert` for the cluster CA certificate and `<kafka_cluster_name>-clients-ca-cert` for the clients CA certificate.

The following example shows a secret for a cluster CA certificate that's associated with a Kafka cluster named `my-cluster`.

*Example secret configuration for a cluster CA certificate*

```
apiVersion: v1
kind: Secret
data:
  ca.crt: LS0tLS1CRUdJTiBDRVJUSUZJQ0F... ①
metadata:
  annotations:
    strimzi.io/ca-cert-generation: "0" ②
  labels:
    strimzi.io/cluster: my-cluster
    strimzi.io/kind: Kafka
  name: my-cluster-cluster-ca-cert
  #...
type: Opaque
```

① Current base64-encoded CA certificate

② Current CA certificate generation annotation value

#### b. Rename the current CA certificate to retain it.

Rename the current `ca.crt` property under `data` as `ca-<date>.crt`, where `<date>` is the certificate expiry date in the format `YEAR-MONTH-DAYTHOUR-MINUTE-SECONDZ`. For example `ca-2023-01-26T17-32-00Z.crt`. Leave the value for the property as it is to retain the current CA certificate.

#### c. Encode your new CA certificate into base64.

```
cat <path_to_new_certificate> | base64
```

d. Update the CA certificate.

Create a new `ca.crt` property under `data` and copy the base64-encoded CA certificate from the previous step as the value for `ca.crt` property.

e. Increase the value of the CA certificate generation annotation.

Update the `strimzi.io/ca-cert-generation` annotation with a higher incremental value. For example, change `strimzi.io/ca-cert-generation=0` to `strimzi.io/ca-cert-generation=1`. If the `Secret` is missing the annotation, the value is treated as `0`, so add the annotation with a value of `1`.

When Strimzi generates certificates, the certificate generation annotation is automatically incremented by the Cluster Operator. For your own CA certificates, set the annotations with a higher incremental value. The annotation needs a higher value than the one from the current secret so that the Cluster Operator can roll the pods and update the certificates. The `strimzi.io/ca-cert-generation` has to be incremented on each CA certificate renewal.

f. Save the secret with the new CA certificate and certificate generation annotation value.

*Example secret configuration updated with a new CA certificate*

```
apiVersion: v1
kind: Secret
data:
  ca.crt: GCa6LS3RTHeKFiFDGBOUDYFAZ0F... ①
  ca-2023-01-26T17-32-00Z.crt: LS0tLS1CRUdJTiBDRVJUSUZJQ0F... ②
metadata:
  annotations:
    strimzi.io/ca-cert-generation: "1" ③
  labels:
    strimzi.io/cluster: my-cluster
    strimzi.io/kind: Kafka
  name: my-cluster-cluster-ca-cert
  #...
type: Opaque
```

① New base64-encoded CA certificate

② Old base64-encoded CA certificate

③ New CA certificate generation annotation value

5. Update the `Secret` for the CA key used to sign your new CA certificate.

a. Edit the existing secret to add the new CA key and update the key generation annotation value.

```
kubectl edit secret <ca_key_name>
```

<ca\_key\_name> is the name of CA key, which is `<kafka_cluster_name>-cluster-ca` for the cluster CA key and `<kafka_cluster_name>-clients-ca` for the clients CA key.

The following example shows a secret for a cluster CA key that's associated with a Kafka cluster named `my-cluster`.

*Example secret configuration for a cluster CA key*

```
apiVersion: v1
kind: Secret
data:
  ca.key: SA1cKF1GFDz0IiPOIUQBHDNFGDFS... ①
metadata:
  annotations:
    strimzi.io/ca-key-generation: "0" ②
  labels:
    strimzi.io/cluster: my-cluster
    strimzi.io/kind: Kafka
    name: my-cluster-cluster-ca
  #...
type: Opaque
```

① Current base64-encoded CA key

② Current CA key generation annotation value

b. Encode the CA key into base64.

```
cat <path_to_new_key> | base64
```

c. Update the CA key.

Copy the base64-encoded CA key from the previous step as the value for the `ca.key` property under `data`.

d. Increase the value of the CA key generation annotation.

Update the `strimzi.io/ca-key-generation` annotation with a higher incremental value. For example, change `strimzi.io/ca-key-generation=0` to `strimzi.io/ca-key-generation=1`. If the `Secret` is missing the annotation, it is treated as `0`, so add the annotation with a value of `1`.

When Strimzi generates certificates, the key generation annotation is automatically incremented by the Cluster Operator. For your own CA certificates together with a new CA key, set the annotation with a higher incremental value. The annotation needs a higher value than the one from the current secret so that the Cluster Operator can roll the pods and update the certificates and keys. The `strimzi.io/ca-key-generation` has to be incremented on each CA certificate renewal.

- e. Save the secret with the new CA key and key generation annotation value.

*Example secret configuration updated with a new CA key*

```
apiVersion: v1
kind: Secret
data:
  ca.key: AB0cKF1GFDzOIIPOIUQWERZJQ0F... ①
metadata:
  annotations:
    strimzi.io/ca-key-generation: "1" ②
  labels:
    strimzi.io/cluster: my-cluster
    strimzi.io/kind: Kafka
  name: my-cluster-cluster-ca
  #...
type: Opaque
```

① New base64-encoded CA key

② New CA key generation annotation value

6. Resume from the pause.

To resume the **Kafka** custom resource reconciliation, set the **pause-reconciliation** annotation to **false**.

```
kubectl annotate --overwrite Kafka <name_of_custom_resource> strimzi.io/pause-reconciliation="false"
```

You can also do the same by removing the **pause-reconciliation** annotation.

```
kubectl annotate Kafka <name_of_custom_resource> strimzi.io/pause-reconciliation-
```

On the next reconciliation, the Cluster Operator performs a rolling update of Kafka and other components to trust the new CA certificate. When the rolling update is complete, the Cluster Operator will start a new one to generate new server certificates signed by the new CA key.

If maintenance time windows are configured, the Cluster Operator will roll the pods at the first reconciliation within the next maintenance time window.

7. Wait until the rolling updates to move to the new CA certificate are complete.
8. Remove any outdated certificates from the secret configuration to ensure that the cluster no longer trusts them.

```
kubectl edit secret <ca_certificate_secret_name>
```

*Example secret configuration with the old certificate removed*

```
apiVersion: v1
kind: Secret
data:
  ca.crt: GCa6LS3RTHeKFiFDGBOUDYFAZ0F...
metadata:
  annotations:
    strimzi.io/ca-cert-generation: "1"
  labels:
    strimzi.io/cluster: my-cluster
    strimzi.io/kind: Kafka
  name: my-cluster-cluster-ca-cert
  #...
type: Opaque
```

9. Start a manual rolling update of your cluster to pick up the changes made to the secret configuration.

See [Managing rolling updates](#).

# Chapter 19. Applying security context to Strimzi pods and containers

Security context defines constraints on pods and containers. By specifying a security context, pods and containers only have the permissions they need. For example, permissions can control runtime operations or access to resources.

## 19.1. How to configure security context

Use security provider plugins or template configuration to apply security context to Strimzi pods and containers.

Apply security context at the pod or container level:

### Pod-level security context

Pod-level security context is applied to all containers in a specific pod.

### Container-level security context

Container-level security context is applied to a specific container.

With Strimzi, security context is applied through one or both of the following methods:

#### Template configuration

Use `template` configuration of Strimzi custom resources to specify security context at the pod or container level.

#### Pod security provider plugins

Use pod security provider plugins to automatically set security context across all pods and containers using preconfigured settings.

Pod security providers offer a simpler alternative to specifying security context through `template` configuration. You can use both approaches. The `template` approach has a higher priority. Security context configured through `template` properties overrides the configuration set by pod security providers. So you might use pod security providers to automatically configure the security context for most containers. And also use `template` configuration to set container-specific security context where needed.

The `template` approach provides flexibility, but it also means you have to configure security context in numerous places to capture the security you want for all pods and containers. For example, you'll need to apply the configuration to each pod in a Kafka cluster, as well as the pods for deployments of other Kafka components.

To avoid repeating the same configuration, you can use the following pod security provider plugins so that the security configuration is in one place.

#### Baseline Provider

The Baseline Provider is based on the Kubernetes *baseline* security profile. The baseline profile

prevents privilege escalations and defines other standard access controls and limitations.

## Restricted Provider

The Restricted Provider is based on the Kubernetes *restricted* security profile. The restricted profile is more restrictive than the baseline profile, and is used where security needs to be tighter.

For more information on the Kubernetes security profiles, see [Pod security standards](#).

### 19.1.1. Template configuration for security context

In the following example, security context is configured for Kafka brokers in the `template` configuration of the `Kafka` resource. Security context is specified at the pod and container level.

*Example template configuration for security context*

```
apiVersion: kafka.strimzi.io/v1
kind: Kafka
metadata:
  name: my-cluster
spec:
  # ...
  kafka:
    template:
      pod: ①
        securityContext:
          runAsUser: 1000001
          fsGroup: 0
      kafkaContainer: ②
        securityContext:
          runAsUser: 2000
    # ...
```

① Pod security context

② Container security context of the Kafka broker container

### 19.1.2. Baseline Provider for pod security

The Baseline Provider is the default pod security provider. It configures the pods managed by Strimzi with a baseline security profile. The baseline profile is compatible with previous versions of Strimzi.

The Baseline Provider is enabled by default if you don't specify a provider. Though you can enable it explicitly by setting the `STRIMZI_POD_SECURITY_PROVIDER_CLASS` environment variable to `baseline` when configuring the Cluster Operator.

*Configuration for the Baseline Provider*

```
# ...
env:
```

```
# ...
- name: STRIMZI_POD_SECURITY_PROVIDER_CLASS
  value: baseline
# ...
```

Instead of specifying `baseline` as the value, you can specify the `io.strimzi.plugin.securityprofiles.impl.BaselinePodSecurityProvider` fully-qualified domain name.

### 19.1.3. Restricted Provider for pod security

The Restricted Provider provides a higher level of security than the Baseline Provider. It configures the pods managed by Strimzi with a restricted security profile.

You enable the Restricted Provider by setting the `STRIMZI_POD_SECURITY_PROVIDER_CLASS` environment variable to `restricted` when configuring the Cluster Operator.

*Configuration for the Restricted Provider*

```
# ...
env:
# ...
- name: STRIMZI_POD_SECURITY_PROVIDER_CLASS
  value: restricted
# ...
```

Instead of specifying `restricted` as the value, you can specify the `io.strimzi.plugin.securityprofiles.impl.RestrictedPodSecurityProvider` fully-qualified domain name.

If you change to the Restricted Provider from the default Baseline Provider, the following restrictions are implemented in addition to the constraints defined in the baseline security profile:

- Limits allowed volume types
- Disallows privilege escalation
- Requires applications to run under a non-root user
- Requires `seccomp` (secure computing mode) profiles to be set as `RuntimeDefault` or `Localhost`
- Limits container capabilities to use only the `NET_BIND_SERVICE` capability

With the Restricted Provider enabled, containers created by the Cluster Operator are set with the following security context.

*Cluster Operator with restricted security context configuration*

```
# ...
securityContext:
  allowPrivilegeEscalation: false
  capabilities:
```

```
drop:  
  - ALL  
runAsNonRoot: true  
seccompProfile:  
  type: RuntimeDefault  
# ...
```

Container capabilities and `seccomp` are Linux kernel features that support container security.

- Capabilities add fine-grained privileges for processes running on a container. The `NET_BIND_SERVICE` capability allows non-root user applications to bind to ports below 1024.
- `seccomp` profiles limit the processes running in a container to only a subset of system calls. The `RuntimeDefault` profile provides a default set of system calls. A `LocalHost` profile uses a profile defined in a file on the node.

#### *Additional resources*

- [Security context](#) on Kubernetes
- [Pod security standards](#) on Kubernetes (including profile descriptions)

## 19.2. Enabling the Restricted Provider for the Cluster Operator

Security pod providers configure the security context constraints of the pods and containers created by the Cluster Operator. The Baseline Provider is the default pod security provider used by Strimzi. You can switch to the Restricted Provider by changing the `STRIMZI_POD_SECURITY_PROVIDER_CLASS` environment variable in the Cluster Operator configuration.

To make the required changes, configure the `060-Deployment-strimzi-cluster-operator.yaml` Cluster Operator installation file located in `install/cluster-operator/`.

By enabling a new pod security provider, any pods or containers created by the Cluster Operator are subject to the limitations it imposes. Pods and containers that are already running are restarted for the changes to take affect.

#### *Prerequisites*

- You need an account with permission to create and manage `CustomResourceDefinition` and RBAC (`ClusterRole`, and `RoleBinding`) resources.

#### *Procedure*

Edit the `Deployment` resource that is used to deploy the Cluster Operator, which is defined in the `060-Deployment-strimzi-cluster-operator.yaml` file.

1. Add or amend the `STRIMZI_POD_SECURITY_PROVIDER_CLASS` environment variable with a value of `restricted`.

```
# ...
env:
# ...
- name: STRIMZI_POD_SECURITY_PROVIDER_CLASS
  value: restricted
# ...
```

Or you can specify the fully-qualified domain name.

## 2. Deploy the Cluster Operator:

```
kubectl create -f install/cluster-operator -n myproject
```

## 3. (Optional) Use `template` configuration to set security context for specific components at the pod or container level.

*Adding security context through `template` configuration*

```
template:
pod:
  securityContext:
    runAsUser: 1000001
    fsGroup: 0
kafkaContainer:
  securityContext:
    runAsUser: 2000
# ...
```

If you apply specific security context for a component using `template` configuration, it takes priority over the general configuration provided by the pod security provider.

## 19.3. Implementing a custom pod security provider

If Strimzi's Baseline Provider and Restricted Provider don't quite match your needs, you can develop a custom pod security provider to deliver all-encompassing pod and container security context constraints.

Implement a custom pod security provider to apply your own security context profile. You can decide what applications and privileges to include in the profile.

Your custom pod security provider can implement the `PodSecurityProvider.java` interface that gets the security context for pods and containers; or it can extend the Baseline Provider or Restricted Provider classes.

The pod security provider plugins use the Java Service Provider Interface, so your custom pod security provider also requires a provider configuration file for service discovery.

To implement your own provider, the general steps include the following:

1. Build the JAR file for the provider.
2. Add the JAR file to the Cluster Operator image.
3. Specify the custom pod security provider when setting the Cluster Operator environment variable `STRIMZI_POD_SECURITY_PROVIDER_CLASS`.

*Additional resources*

- [Pod security provider interface](#)
- [Baseline Provider and Restricted Provider classes](#)
- [Provider configuration file](#)
- [Java Service Provider Interface](#)

## 19.4. Handling of security context by Kubernetes platform

Handling of security context depends on the tooling of the Kubernetes platform you are using.

For example, OpenShift uses built-in security context constraints (SCCs) to control permissions. SCCs are the settings and strategies that control the security features a pod has access to.

By default, OpenShift injects security context configuration automatically. In most cases, this means you don't need to configure security context for the pods and containers created by the Cluster Operator. Although you can still create and manage your own SCCs.

For more information, see the [OpenShift documentation](#).

# Chapter 20. Scaling clusters by adding or removing brokers

Scaling Kafka clusters by adding brokers can improve performance and reliability. Increasing the number of brokers provides more resources, enabling the cluster to handle larger workloads and process more messages. It also enhances fault tolerance by providing additional replicas. Conversely, removing underutilized brokers can reduce resource consumption and increase efficiency. Scaling must be done carefully to avoid disruption or data loss. Redistributing partitions across brokers reduces the load on individual brokers, increasing the overall throughput of the cluster.

Adjusting the `replicas` configuration changes the number of brokers in a cluster. A replication factor of 3 means each partition is replicated across three brokers, ensuring fault tolerance in case of broker failure:

*Example node pool configuration for the number of replicas*

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaNodePool
metadata:
  name: my-node-pool
  labels:
    strimzi.io/cluster: my-cluster
spec:
  replicas: 3
  # ...
```

The actual replication factor for topics depends on the number of available brokers and how many brokers store replicas of each topic partition (configured by `default.replication.factor`). The minimum number of replicas that must acknowledge a write for it to be considered successful is defined by `min.insync.replicas`:

*Example configuration for topic replication*

```
apiVersion: kafka.strimzi.io/v1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    config:
      default.replication.factor: 3
      min.insync.replicas: 2
    # ...
```

When adding brokers by changing the number of `replicas`, node IDs start at 0, and the Cluster Operator assigns the next lowest available ID to new brokers. Removing brokers starts with the pod

that has the highest node ID. Additionally, when scaling clusters with node pools, you can [assign node IDs for scaling operations](#).

Strimzi can automatically reassign partitions when brokers are added or removed if Cruise Control is deployed and auto-rebalancing is enabled in the Kafka resource. If auto-rebalancing is disabled, you can use Cruise Control to generate optimization proposals before manually rebalancing the cluster.

Cruise Control provides `add-brokers` and `remove-brokers` modes for scaling:

- Use the `add-brokers` mode after scaling up to move partition replicas to the new brokers.
- Use the `remove-brokers` mode before scaling down to move partition replicas off the brokers being removed.

With auto-rebalancing, these modes run automatically using the default Cruise Control configuration or custom settings from a rebalancing template.

**NOTE**

To increase the throughput of a Kafka topic, you can increase the number of partitions for that topic, distributing the load across multiple brokers. However, if all brokers are constrained by a resource (such as I/O), adding more partitions won't improve throughput, and adding more brokers is necessary.

## 20.1. Triggering auto-rebalances when scaling clusters

Set up auto-rebalancing to automatically redistribute topic partitions when scaling a cluster. You can scale a Kafka cluster by adjusting the number of brokers using the `spec.replicas` property in the `Kafka` or `KafkaNodePool` custom resource used in deployment. When auto-rebalancing is enabled, the cluster is rebalanced without further intervention.

- After adding brokers, topic partitions are redistributed across the new brokers.
- Before removing brokers, partitions are moved off the brokers being removed.

Auto-rebalancing helps maintain balanced load distribution across Kafka brokers during scaling operations, depending on how the rebalancing configuration is set up.

Scaling operates in two modes: `add-brokers` and `remove-brokers`. Each mode can have its own auto-rebalancing configuration specified in the `Kafka` resource under `spec.cruiseControl.autoRebalance` properties. Use the `template` property to specify a predefined `KafkaRebalance` resource, which serves as a rebalance configuration template. If a template is not specified in the `autorebalance` configuration, the default Cruise Control rebalancing configuration is used. You can apply the same template configuration for both scaling modes, use different configurations for each, or enable auto-rebalancing for only one mode. If `autorebalance` configuration is not set for a mode, auto-rebalancing will not occur for that mode.

The template `KafkaRebalance` resource must include the `strimzi.io/rebalance-template: "true"` annotation. The template does not represent an actual rebalance request but holds the rebalancing configuration. During scaling, the Cluster Operator creates a `KafkaRebalance` resource based on this template, named `<cluster_name>-auto-rebalancing-<mode>`, where `<mode>` is either `add-brokers` or

`remove-brokers`. The Cluster Operator applies a finalizer (`strimzi.io/auto-rebalancing`) to prevent the resource's deletion during the rebalancing process.

Progress is reflected in the status of the `Kafka` resource. The `status.autoRebalance` property indicates the state of the rebalance. A `modes` property lists the brokers being added or removed during the operation to help track progress across reconciliations.

#### Prerequisites

- The Cluster Operator must be deployed.
- Cruise Control is deployed with Kafka.
- You have configured optimization goals and, optionally, capacity limits on broker resources.

#### Procedure

1. Create a rebalancing template for the auto-rebalancing operation (if required).

Configure a `KafkaRebalance` resource with the `strimzi.io/rebalance-template: "true"` annotation. The rebalance configuration template does not require `mode` and `brokers` properties unlike when [when generating an optimization proposal for rebalancing](#).

#### Example rebalancing template configuration

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaRebalance
metadata:
  name: my-add-remove-brokers-rebalancing-template
  annotations:
    strimzi.io/rebalance-template: "true" ①
spec:
  goals:
    - CpuCapacityGoal
    - NetworkInboundCapacityGoal
    - DiskCapacityGoal
    - RackAwareGoal
    - MinTopicLeadersPerBrokerGoal
    - NetworkOutboundCapacityGoal
    - ReplicaCapacityGoal
  skipHardGoalCheck: true
  # ... other rebalancing configuration
```

① The annotation designates the resource as a rebalance configuration template.

2. Apply the configuration to create the template.
3. Add auto-rebalancing configuration to the `Kafka` resource.

In this example, the same template is used for adding and removing brokers.

#### Example using template specifications for auto-rebalancing

```
apiVersion: kafka.strimzi.io/v1
```

```

kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
  cruiseControl:
    autoRebalance:
      - mode: add-brokers
        template:
          name: my-add-remove-brokers-rebalancing-template
      - mode: remove-brokers
        template:
          name: my-add-remove-brokers-rebalancing-template

```

To use default Cruise Control configuration for rebalancing, omit the template configuration. In this example, the default configuration is used when adding brokers.

*Example using default specifications for auto-rebalancing*

```

apiVersion: kafka.strimzi.io/v1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
  cruiseControl:
    autoRebalance:
      - mode: add-brokers
      - mode: remove-brokers
        template:
          name: my-add-remove-brokers-rebalancing-template

```

4. Apply the changes to the **Kafka** configuration.

Wait for the Cluster Operator to update the cluster.

5. Scale the cluster by adjusting the **spec.replicas** property representing the number of brokers in the cluster.

The following example shows a node pool configuration for a cluster using three brokers (**replicas: 3**).

*Example node pool configuration*

```

apiVersion: kafka.strimzi.io/v1
kind: KafkaNodePool
metadata:
  name: pool-b
  labels:

```

```

        strimzi.io/cluster: my-cluster
spec:
  replicas: 3
  roles:
    - broker
  storage:
    type: jbod
    volumes:
      - id: 0
        type: persistent-claim
        size: 100Gi
        deleteClaim: false
    # ...

```

For more information on scaling through node pools, see the following:

- [Adding nodes to a node pool.](#)
  - [Removing nodes from a node pool.](#)
6. Check the rebalance status.  
The status is visible in the **Kafka** resource.

*Example status for auto-rebalancing*

```

apiVersion: kafka.strimzi.io/v1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
  cruiseControl:
    autoRebalance:
      - mode: add-brokers
        template:
          name: my-add-remove-brokers-rebalancing-template
      - mode: remove-brokers
        template:
          name: my-add-remove-brokers-rebalancing-template
status:
  autoRebalance:
    lastTransitionTime: <timestamp_for_last_rebalance_state>
    state: RebalanceOnScaleDown ①
    modes: ②
      - mode: add-brokers
        brokers: <broker_ids>
      - mode: remove-brokers
        brokers: <broker_ids>

```

① The state of the rebalance, which shows **RebalanceOnScaleUp** when adding brokers, and

`RebalanceOnScaleDown` when removing brokers. Scale-down operations take precedence. Initial and final state (failed or successful) shows as `Idle`.

- ② Rebalance operations grouped by mode, with a list of nodes to be added or removed.

**NOTE** During a rebalance, the status of the `KafkaRebalance` resource used for the rebalance is checked, and the auto-rebalance state is adjusted accordingly.

## 20.2. Skipping checks on scale-down operations

By default, Strimzi performs a check to ensure that there are no partition replicas on brokers before initiating a scale-down operation on a Kafka cluster. The check applies to nodes in node pools that perform the role of broker only or a dual role of broker and controller.

If replicas are found, the scale-down is not done in order to prevent potential data loss. To scale-down the cluster, no replicas must be left on the broker before trying to scale it down again.

However, there may be scenarios where you want to bypass this mechanism. Disabling the check might be necessary on busy clusters, for example, because new topics keep generating replicas for the broker. This situation can indefinitely block the scale-down, even when brokers are nearly empty. Overriding the blocking mechanism in this way has an impact: the presence of topics on the broker being scaled down will likely cause a reconciliation failure for the Kafka cluster.

You can bypass the blocking mechanism by annotating the `Kafka` resource for the Kafka cluster. Annotate the resource by setting the `strimzi.io/skip-broker-scaledown-check` annotation to `true`:

*Adding the annotation to skip checks on scale-down operations*

```
kubectl annotate Kafka my-kafka-cluster strimzi.io/skip-broker-scaledown-check="true"
```

This annotation instructs Strimzi to skip the scale-down check. Replace `my-kafka-cluster` with the name of your specific `Kafka` resource.

To restore the check for scale-down operations, remove the annotation:

*Removing the annotation to skip checks on scale-down operations*

```
kubectl annotate Kafka my-kafka-cluster strimzi.io/skip-broker-scaledown-check-
```

# Chapter 21. Using Cruise Control for cluster rebalancing

Cruise Control is an open-source application designed to run alongside Kafka to help optimize use of cluster resources by doing the following:

- Monitoring cluster workload
- Rebalancing partitions based on predefined constraints

Cruise Control operations help with running a more balanced Kafka cluster that uses brokers more efficiently.

As Kafka clusters evolve, some brokers may become overloaded while others remain underutilized. Cruise Control addresses this imbalance by modeling resource utilization at the replica level—including, CPU, disk, network load—and generating optimization proposals (which you can approve or reject) for balanced partition assignments based on configurable optimization goals.

Optimization proposals are configured and generated using a [KafkaRebalance](#) resource. You can configure the resource using an annotation so that optimization proposals are approved automatically or manually.

**NOTE** Strimzi provides [example configuration files for Cruise Control](#).

## 21.1. Cruise Control components and features

Cruise Control comprises four main components:

### Load Monitor

Load Monitor collects the metrics and analyzes cluster workload data.

### Analyzer

Analyzer generates optimization proposals based on collected data and configured goals.

### Anomaly Detector

Anomaly Detector identifies and reports irregularities in cluster behavior.

### Executor

Executor applies approved optimization proposals to the cluster.

### REST API

Cruise Control provides a REST API for client interactions, which Strimzi uses to support these features:

- Generating optimization proposals from optimization goals
- Rebalancing a Kafka cluster based on an optimization proposal
- Changing topic replication factor

- Reassigning partitions between JBOD disks

**NOTE**

Cruise Control self-healing is not supported. Notifications and custom goals may be introduced through customization. For example, an anomaly notifier or custom goals can be added by including the appropriate JAR files in a custom image based on the Strimzi Kafka image.

### 21.1.1. Optimization goals

Optimization goals define objectives for rebalancing, such as distributing topic replicas evenly across brokers.

They are categorized as follows:

- **Supported goals** are a list of goals supported by the Cruise Control instance that can be used in its operations. By default, this list includes all goals included with Cruise Control. For a goal to be used in other categories, such as default or hard goals, it must first be listed in supported goals. To prevent a goal's usage, remove it from this list.
- **Hard goals** are preset and must be satisfied for a proposal to succeed.
- **Soft goals** are preset goals with objectives that are prioritized during optimization as much as possible, without preventing a proposal from being created if all hard goals are satisfied.
- **Default goals** refer to the goals used by default when generating proposals. They match the supported goals unless specifically set by the user.
- **Proposal-specific goals** are a subset of supported goals configured for specific proposals.

Configure optimization goals in the [Kafka](#) and [KafkaRebalance](#) custom resources.

- [Kafka](#) resource for supported, hard, and default goals.
  - Supported goals: `Kafka.spec.cruiseControl.config.goals`
  - Hard goals: `Kafka.spec.cruiseControl.config.hard.goals`
  - Default goals: `Kafka.spec.cruiseControl.config.default.goals`
- [KafkaRebalance](#) resource for proposal-specific goals.
  - Proposal-specific goals: `KafkaRebalance.spec.goals`

#### Supported goals

Supported goals are predefined and available to use for generating Cruise Control optimization proposals. Goals not listed as supported goals cannot be used in Cruise Control operations. Some supported goals are preset as hard goals.

Configure supported goals in `Kafka.spec.cruiseControl.config.goals`:

- To accept inherited supported goals, omit the `goals` property.
- To modify supported goals, specify the goals in descending priority order in the `goals` property.

## Hard and soft goals

Hard goals must be satisfied for optimization proposals to be generated. Soft goals are best-effort objectives that Cruise Control tries to meet after all hard goals are satisfied. The classification of hard and soft goals is fixed in Cruise Control code and cannot be changed.

Cruise Control first prioritizes satisfying hard goals, and then addresses soft goals in the order they are listed. A proposal meeting all hard goals is valid, even if it violates some soft goals.

For example, a soft goal might be to evenly distribute a topic's replicas. Cruise Control continues to generate an optimization proposal even if the soft goal isn't completely satisfied.

Configure hard goals in your Cruise Control deployment using `Kafka.spec.cruiseControl.config.hard.goals`:

- To enforce all hard goals, omit the `hard.goals` property.
- To specify hard goals, list them in `hard.goals`.
- To exclude a hard goal, ensure it's not in either `default.goals` or `hard.goals`.

Increasing the number of configured hard goals will reduce the likelihood of Cruise Control generating optimization proposals.

## Default goals

Cruise Control uses default goals to generate an optimization proposal.

If `default.goals` is not specified in the Cruise Control deployment configuration, Strimzi configures `default.goals` to the list of supported goals specified in `goals`.

The optimization proposal based on this supported goals list is then generated and cached.

Configure default goals in `Kafka.spec.cruiseControl.config.default.goals`:

- To use supported goals as default, omit the `default.goals` property.
- To modify default goals, specify a subset of supported goals in the `default.goals` property.  
You can adjust the priority order in the default goals configuration.

## Proposal-specific goals

Proposal-specific optimization goals support the creation of optimization proposals based on a specific list of goals. If proposal-specific goals are not set in the `KafkaRebalance` resource, then default goals are used

Configure proposal-specific goals in `KafkaRebalance.spec.goals`, specifying a subset of supported optimization goals for customization.

For example, you can optimize topic leader replica distribution across the Kafka cluster without considering disk capacity or utilization by defining a single proposal-specific goal.

## Goals order of priority

Unless you change the Cruise Control [deployment configuration](#), Strimzi inherits goals from Cruise Control, in descending priority order.

The following list shows supported goals inherited by Strimzi from Cruise Control in descending priority order. Goals labeled as hard are mandatory constraints that must be satisfied for optimization proposals.

- `RackAwareGoal` (hard)
- `MinTopicLeadersPerBrokerGoal` (hard)
- `ReplicaCapacityGoal` (hard)
- `DiskCapacityGoal` (hard)
- `NetworkInboundCapacityGoal` (hard)
- `NetworkOutboundCapacityGoal` (hard)
- `CpuCapacityGoal` (hard)
- `ReplicaDistributionGoal`
- `PotentialNwOutGoal`
- `DiskUsageDistributionGoal`
- `NetworkInboundUsageDistributionGoal`
- `NetworkOutboundUsageDistributionGoal`
- `CpuUsageDistributionGoal`
- `TopicReplicaDistributionGoal`
- `LeaderReplicaDistributionGoal`
- `LeaderBytesInDistributionGoal`
- `PreferredLeaderElectionGoal`
- `IntraBrokerDiskCapacityGoal` (hard)
- `IntraBrokerDiskUsageDistributionGoal`

Resource distribution goals are subject to [capacity limits](#) on broker resources.

For more information on each optimization goal, see [Goals](#) in the Cruise Control Wiki.

*Example Kafka configuration for default and hard goals*

```
apiVersion: kafka.strimzi.io/v1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
  entityOperator:
```

```

topicOperator: {}
userOperator: {}
cruiseControl:
  brokerCapacity:
    inboundNetwork: 10000KB/s
    outboundNetwork: 10000KB/s
  config:
    #'default.goals' (superset) must also include all 'hard.goals' (subset)
    default.goals: >
      com.linkedin.kafka.cruisecontrol.analyzer.goals.RackAwareGoal,
      com.linkedin.kafka.cruisecontrol.analyzer.goals.ReplicaCapacityGoal,
      com.linkedin.kafka.cruisecontrol.analyzer.goals.DiskCapacityGoal
      com.linkedin.kafka.cruisecontrol.analyzer.goals.NetworkInboundCapacityGoal,
      com.linkedin.kafka.cruisecontrol.analyzer.goals.NetworkOutboundCapacityGoal
    hard.goals: >
      com.linkedin.kafka.cruisecontrol.analyzer.goals.RackAwareGoal
      com.linkedin.kafka.cruisecontrol.analyzer.goals.NetworkInboundCapacityGoal,
      com.linkedin.kafka.cruisecontrol.analyzer.goals.NetworkOutboundCapacityGoal
    # ...

```

## IMPORTANT

Ensure that the supported `goals`, `default.goals`, and (unless `skipHardGoalCheck` is set to `true`) proposal-specific `spec.goals` include all hard goals specified in `hard.goals` to avoid errors when generating optimization proposals. Hard goals must be included as a subset in the supported, default, and proposal-specific goals.

*Example KafkaRebalance configuration for proposal-specific goals*

```

apiVersion: kafka.strimzi.io/v1
kind: KafkaRebalance
metadata:
  name: my-rebalance
  labels:
    strimzi.io/cluster: my-cluster
spec:
  goals:
    - RackAwareGoal
    - TopicReplicaDistributionGoal
  skipHardGoalCheck: true

```

## Skipping hard goal checks

If `skipHardGoalCheck: true` is specified in the `KafkaRebalance` custom resource, Cruise Control does not verify that the proposal-specific goals include all the configured hard goals. This allows for more flexibility in generating optimization proposals, but may lead to proposals that do not satisfy all hard goals.

However, any hard goals included in the proposal-specific goals will still be treated as hard goals by Cruise Control, even with `skipHardGoalCheck: true`.

## 21.1.2. Optimization proposals

Optimization proposals are summaries of proposed changes based on the defined optimization goals, assessed in a specific order of priority. You can approve or reject proposals and rerun them with adjusted goals if needed.

With Cruise Control deployed for use in Strimzi, the process to generate and approve an optimization proposal is as follows:

1. Create a [KafkaRebalance](#) resource specifying optimization goals and any specific configurations. This resource triggers Cruise Control to initiate the optimization proposal generation process.
2. A Cruise Control Metrics Reporter runs in every Kafka broker, collecting raw metrics and publishing them to a dedicated Kafka topic ([strimzi.cruisecontrol.metrics](#)). Metrics for brokers, topics, and partitions are aggregated, sampled, and stored in other [topics automatically created when Cruise Control is deployed](#).
3. Load Monitor collects, processes, and stores the metrics as a *workload model*--including CPU, disk, and network utilization data—which is used by the Analyzer and Anomaly Detector.
4. Anomaly Detector continuously monitors the health and performance of the Kafka cluster, checking for things like broker failures or disk capacity issues, that could impact cluster stability.
5. Analyzer creates optimization proposals based on the workload model from the Load Monitor. Based on configured goals and capacities, it generates an optimization proposal for balancing partitions across brokers. Through the REST API, a summary of the proposal is reflected in the status of the [KafkaRebalance](#) resource.
6. The optimization proposal is approved or rejected (manually or automatically) based on its alignment with cluster management goals.
7. If approved, the Executor applies the optimization proposal to rebalance the Kafka cluster. This involves reassigning partitions and redistributing workload across brokers according to the approved proposal.

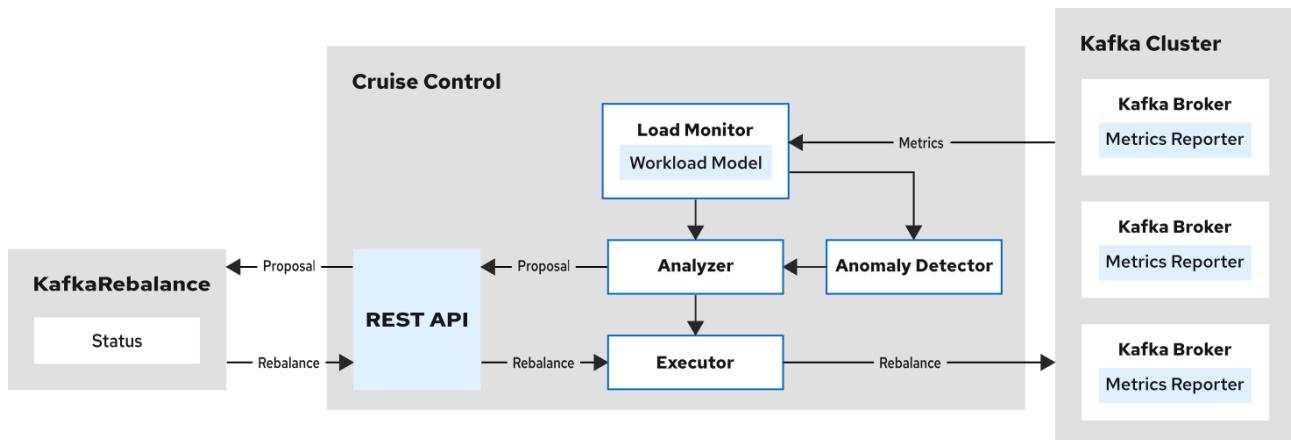


Figure 6. Cruise Control optimization process

Optimization proposals comprise a list of partition reassignment mappings. When you approve a proposal, the Cruise Control server applies these partition reassignments to the Kafka cluster.

A partition reassignment consists of either of the following types of operations:

- Partition movement: Involves transferring the partition replica and its data to a new location. Partition movements can take one of two forms:
  - Inter-broker movement: The partition replica is moved to a log directory on a different broker.
  - Intra-broker movement: The partition replica is moved to a different log directory on the same broker.
- Leadership movement: Involves switching the leader of the partition's replicas.

Cruise Control issues partition reassessments to the Kafka cluster in batches. The performance of the cluster during the rebalance is affected by the number and magnitude of each type of movement contained in each batch.

## Rebalancing modes

Proposals for rebalances can be generated in four modes, which are specified using the `spec.mode` property of the `KafkaRebalance` custom resource.

### full mode

The `full` mode runs a full rebalance by moving replicas across all the brokers in the cluster. This is the default mode if the `spec.mode` property is not defined in the `KafkaRebalance` custom resource.

### add-brokers mode

The `add-brokers` mode is used after scaling up a Kafka cluster by adding one or more brokers. Normally, after scaling up a Kafka cluster, new brokers are used to host only the partitions of newly created topics. If no new topics are created, the newly added brokers are not used and the existing brokers remain under the same load. By using the `add-brokers` mode immediately after adding brokers to the cluster, the rebalancing operation moves replicas from existing brokers to the newly added brokers. You specify the new brokers as a list using the `spec.brokers` property of the `KafkaRebalance` custom resource.

### remove-brokers mode

The `remove-brokers` mode is used before scaling down a Kafka cluster by removing one or more brokers. The `remove-brokers` mode moves replicas off the brokers that are going to be removed. When these brokers are not hosting replicas anymore, you can safely run the scaling down operation. You specify the brokers you're removing as a list in the `spec.brokers` property in the `KafkaRebalance` custom resource.

### remove-disks mode

The `remove-disks` mode is used specifically to reassign partitions between JBOD disks used for storage on the same broker. You specify a list of broker IDs with corresponding volume IDs for partition reassignment.

#### NOTE

Brokers are shut down even if they host replicas when [checks are skipped on scale-down operations](#).

In general, use the `full` rebalance mode to rebalance a Kafka cluster by spreading the load across brokers. Use the `add-brokers` and `remove-brokers` modes only if you want to scale your cluster up or down and rebalance the replicas accordingly.

The procedure to run a rebalance is actually the same across the three different modes. The only difference is with specifying a mode through the `spec.mode` property and, if needed, listing brokers that have been added or will be removed through the `spec.brokers` property.

## The results of an optimization proposal

When an optimization proposal is generated, a summary and broker load is returned.

### Summary

The summary is contained in the `KafkaRebalance` resource. The summary provides an overview of the proposed cluster rebalance and indicates the scale of the changes involved. A summary of a successfully generated optimization proposal is contained in the `Status.optimizationResult` property of the `KafkaRebalance` resource. The information provided is a summary of the full optimization proposal.

### Broker load

The broker load is stored in a ConfigMap that contains data as a JSON string. The broker load shows before and after values for the proposed rebalance, so you can see the impact on each of the brokers in the cluster.

## Manually approving or rejecting an optimization proposal

An optimization proposal summary shows the proposed scope of changes.

You can use the name of the `KafkaRebalance` resource to return a summary from the command line.

### *Returning an optimization proposal summary*

```
kubectl describe kafka-rebalance <kafka_rebalance_resource_name> -n <namespace>
```

You can also use the [jq command line JSON parser tool](#).

### *Returning an optimization proposal result using jq*

```
kubectl get kafka-rebalance <kafka_rebalance_resource_name> -n <namespace> -o json | jq '.status.optimizationResult'
```

Use the summary to decide whether to approve or reject an optimization proposal.

## Approving an optimization proposal

You [approve the optimization](#) proposal by setting the `strimzi.io/rebalance` annotation of the `KafkaRebalance` resource to `approve`. Cruise Control applies the proposal to the Kafka cluster and starts a cluster rebalance operation.

## Rejecting an optimization proposal

If you choose not to approve an optimization proposal, you can [change the optimization goals](#) or [update any of the rebalance performance tuning options](#), and then generate another proposal. You can generate a new optimization proposal for a `KafkaRebalance` resource by setting the `strimzi.io/rebalance` annotation to `refresh`.

Use optimization proposals to assess the movements required for a rebalance. For example, a summary describes inter-broker and intra-broker movements. Inter-broker rebalancing moves data between separate brokers. Intra-broker rebalancing moves data between disks on the same broker when you are using a JBOD storage configuration. Such information can be useful even if you don't go ahead and approve the proposal.

You might reject an optimization proposal, or delay its approval, because of the additional load on a Kafka cluster when rebalancing. If the proposal is delayed for too long, the cluster load may change significantly, so it may be better to request a new proposal.

In the following example, the proposal suggests the rebalancing of data between separate brokers. The rebalance involves the movement of 55 partition replicas, totaling 12MB of data, across the brokers. The proposal will also move 24 partition leaders to different brokers. This requires a change to the cluster metadata, which has a low impact on performance.

The balancedness scores are measurements of the overall balance of the Kafka cluster before and after the optimization proposal is approved. A balancedness score is based on optimization goals. If all goals are satisfied, the score is 100. The score is reduced for each goal that will not be met. Compare the balancedness scores to see whether the Kafka cluster is less balanced than it could be following a rebalance.

### *Example optimization proposal summary*

```
Name:      my-rebalance
Namespace: myproject
Labels:    strimzi.io/cluster=my-cluster
Annotations: API Version: kafka.strimzi.io/v1alpha1
Kind:      KafkaRebalance
Metadata:
# ...
Status:
  Conditions:
    Last Transition Time: 2022-04-05T14:36:11.900Z
    Status:          ProposalReady
    Type:           State
  Observed Generation: 1
Optimization Result:
  Data To Move MB: 0
  Excluded Brokers For Leadership:
  Excluded Brokers For Replica Move:
  Excluded Topics:
    Intra Broker Data To Move MB: 12
    Monitored Partitions Percentage: 100
    Num Intra Broker Replica Movements: 0
```

Num Leader Movements:	24
Num Replica Movements:	55
On Demand Balancedness Score After:	82.91290759174306
On Demand Balancedness Score Before:	78.01176356230222
Recent Windows:	5
Session Id:	a4f833bd-2055-4213-bfdd-ad21f95bf184

Though the inter-broker movement of partition replicas has a high impact on performance, the total amount of data is not large. If the total data was much larger, you could reject the proposal, or time when to approve the rebalance to limit the impact on the performance of the Kafka cluster.

[Rebalance performance tuning options](#) can help reduce the impact of data movement. If you can extend the rebalance period, you can divide the rebalance into smaller batches. Fewer data movements at a single time reduces the load on the cluster.

## Optimization proposal summary properties

The following table explains the properties contained in the optimization proposal's summary.

*Table 29. Properties contained in an optimization proposal summary*

JSON property	Description
<code>numIntraBrokerReplicaMovements</code>	The total number of partition replicas that will be transferred between the disks of the cluster's brokers.  <b>Performance impact during rebalance operation:</b> Relatively high, but lower than <code>numReplicaMovements</code> .
<code>excludedBrokersForLeadership</code>	Not yet supported. An empty list is returned.
<code>numReplicaMovements</code>	The number of partition replicas that will be moved between separate brokers.  <b>Performance impact during rebalance operation:</b> Relatively high.

JSON property	Description
<a href="#">onDemandBalancednessScoreBefore</a> <a href="#">onDemandBalancednessScoreAfter</a>	<p>A measurement of the overall <i>balancedness</i> of a Kafka Cluster, before and after the optimization proposal was generated.</p> <p>The score is calculated by subtracting the sum of the <a href="#">BalancednessScore</a> of each violated soft goal from 100. Cruise Control assigns a <a href="#">BalancednessScore</a> to every optimization goal based on several factors, including priority—the goal's position in the list of <a href="#">default.goals</a> or proposal-specific goals.</p> <p>The <a href="#">Before</a> score is based on the workload model of the Kafka cluster. The <a href="#">After</a> score is based on the predicted workload model after applying the generated optimization proposal.</p>
<a href="#">intraBrokerDataToMoveMB</a>	<p>The sum of the size of each partition replica that will be moved between disks on the same broker (see also <a href="#">numIntraBrokerReplicaMovements</a>).</p> <p><b>Performance impact during rebalance operation:</b> Variable. The larger the number, the longer the cluster rebalance will take to complete. Moving a large amount of data between disks on the same broker has less impact than between separate brokers (see <a href="#">dataToMoveMB</a>).</p>
<a href="#">recentWindows</a>	<p>The number of metrics windows upon which the optimization proposal is based.</p>
<a href="#">dataToMoveMB</a>	<p>The sum of the size of each partition replica that will be moved to a separate broker (see also <a href="#">numReplicaMovements</a>).</p> <p><b>Performance impact during rebalance operation:</b> Variable. The larger the number, the longer the cluster rebalance will take to complete.</p>
<a href="#">monitoredPartitionsPercentage</a>	<p>The percentage of partitions in the Kafka cluster covered by the optimization proposal. Affected by the number of <a href="#">excludedTopics</a>.</p>

JSON property	Description
<code>excludedTopics</code>	If you specified a regular expression in the <code>spec.excludedTopicsRegex</code> property in the <a href="#">KafkaRebalance</a> resource, all topic names matching that expression are listed here. These topics are excluded from the calculation of partition replica/leader movements in the optimization proposal.
<code>numLeaderMovements</code>	The number of partitions whose leaders will be switched to different replicas.  <b>Performance impact during rebalance operation:</b> Relatively low.
<code>excludedBrokersForReplicaMove</code>	Not yet supported. An empty list is returned.

## Automatically approving an optimization proposal

To save time, you can automate the process of approving optimization proposals. With automation, when you generate an optimization proposal it goes straight into a cluster rebalance.

To enable the optimization proposal auto-approval mechanism, create the [KafkaRebalance](#) resource with the `strimzi.io/rebalance-auto-approval` annotation set to `true`. If the annotation is not set or set to `false`, the optimization proposal requires manual approval.

*Example rebalance request with auto-approval mechanism enabled*

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaRebalance
metadata:
  name: my-rebalance
  labels:
    strimzi.io/cluster: my-cluster
  annotations:
    strimzi.io/rebalance-auto-approval: "true"
spec:
  mode: # any mode
  # ...
```

You can still check the status when automatically approving an optimization proposal. The status of the [KafkaRebalance](#) resource moves to [Ready](#) when the rebalance is complete.

## Comparing broker load data

Broker load data provides insights into current and anticipated usage of resources following a rebalance. The data is stored in a [ConfigMap](#) (with the same name as the [KafkaRebalance](#) resource) as a JSON formatted string

When a Kafka rebalance proposal reaches the [ProposalReady](#) state, Strimzi creates a [ConfigMap](#)

(named after the `KafkaRebalance` custom resource) containing a JSON string of broker metrics generated from Cruise Control. Each broker has a set of key metrics represented by three values:

- The current metric value before the optimization proposal is applied
- The expected metric value after applying the proposal
- The difference between the two values (after minus before)

This `ConfigMap` remains accessible even after the rebalance completes.

To view this data from the command line, use the `ConfigMap` name.

#### *Returning ConfigMap data*

```
kubectl describe configmaps <my_rebalance_configmap_name> -n <namespace>
```

You can also use the [jq command line JSON parser tool](#) to extract the JSON string.

#### *Extracting the JSON string from the ConfigMap using jq*

```
kubectl get configmaps <my_rebalance_configmap_name> -o json | jq  
'.[>"data"]["brokerLoad.json"]|fromjson|.'
```

*Table 30. Properties captured in the config map*

JSON property	Description
<code>leaders</code>	The number of replicas on this broker that are partition leaders.
<code>replicas</code>	The number of replicas on this broker.
<code>cpuPercentage</code>	The CPU utilization as a percentage of the defined capacity.
<code>diskUsedPercentage</code>	The disk utilization as a percentage of the defined capacity.
<code>diskUsedMB</code>	The absolute disk usage in MB.
<code>networkOutRate</code>	The total network output rate for the broker.
<code>leaderNetworkInRate</code>	The network input rate for all partition leader replicas on this broker.
<code>followerNetworkInRate</code>	The network input rate for all follower replicas on this broker.
<code>potentialMaxNetworkOutRate</code>	The hypothetical maximum network output rate that would be realized if this broker became the leader of all the replicas it currently hosts.

#### **Adjusting the cached proposal refresh rate**

Cruise Control maintains a *cached optimization proposal* based on the configured default optimization goals. This proposal is generated from the workload model and updated every 15 minutes to reflect the current state of the Kafka cluster. When you generate an optimization proposal using the default goals, Cruise Control returns the latest cached version.

For clusters with rapidly changing workloads, you may want to shorten the refresh interval to ensure the optimization proposal reflects the most recent state. However, reducing the interval increases the load on the Cruise Control server. To adjust the refresh rate, modify the `proposal.expiration.ms` setting in the Cruise Control deployment configuration.

#### Additional resources

- [Cruise Control documentation](#)

### 21.1.3. Tuning options for rebalances

Configuration options allow you to fine-tune cluster rebalance performance. These settings control the movement of partition replicas and leadership, as well as the bandwidth allocated for rebalances.

#### Selecting replica movement strategies

Cluster rebalance performance is also influenced by the *replica movement strategy* that is applied to the batches of partition reassignment commands. By default, Cruise Control uses the `BaseReplicaMovementStrategy`, which applies the reassessments in the order they were generated. However, this strategy could lead to the delay of other partition reassessments if large partition reassessments are generated then ordered first.

Cruise Control provides four alternative replica movement strategies that can be applied to optimization proposals:

- `PrioritizeSmallReplicaMovementStrategy`: Reassign smaller partitions first.
  - `PrioritizeLargeReplicaMovementStrategy`: Reassign larger partitions first.
  - `PostponeUrpReplicaMovementStrategy`: Prioritize partitions without out-of-sync replicas.
  - `PrioritizeMinIsrWithOfflineReplicasStrategy`: Prioritize reassessments for partitions at or below their minimum in-sync replicas (MinISR) with offline replicas.
- Set `cruiseControl.config.concurrency.adjuster.min.isr.check.enabled` to `true` in the `Kafka` resource to enable this strategy.

These strategies can be configured as a sequence. The first strategy attempts to compare two partition reassessments using its internal logic. If the reassessments are equivalent, then it passes them to the next strategy in the sequence to decide the order, and so on.

#### Intra-broker disk balancing

Intra-broker balancing shifts data between disks on the same broker, useful for deployments with JBOD storage and multiple disks. This type of balancing incurs less network overhead than inter-broker balancing.

**NOTE**

If you are using JBOD storage with a single disk, intra-broker disk balancing will result in a proposal with 0 partition movements since there are no disks to balance.

To enable intra-broker balancing, set `rebalanceDisk` to `true` in `KafkaRebalance.spec`. When this is enabled, do not specify a `goals` field, as Cruise Control will automatically configure intra-broker

goals and disregard inter-broker goals. Cruise Control does not perform inter-broker and intra-broker balancing at the same time.

## Rebalance tuning

You can set the following rebalance tuning options when configuring Cruise Control or individual rebalances:

- Set Cruise Control server configurations in `Kafka.spec.cruiseControl.config` in the `Kafka` resource.
- Set proposal-specific configurations in `KafkaRebalance.spec` in the `KafkaRebalance` resource.

*Table 31. Rebalance configuration tuning properties*

Cruise Control properties	KafkaRebalance properties	Default	Description
<code>num.concurrent.partition.movements.per.broker</code>	<code>concurrentPartitionMovementsPerBroker</code>	5	The maximum number of inter-broker partition movements in each partition reassignment batch
<code>num.concurrent.intra.broker.partition.movements</code>	<code>concurrentIntraBrokerPartitionMovements</code>	2	The maximum number of intra-broker partition movements in each partition reassignment batch
<code>num.concurrent.leader.movements</code>	<code>concurrentLeaderMovements</code>	1000	The maximum number of partition leadership changes in each partition reassignment batch
<code>default.replication.throttle</code>	<code>replicationThrottle</code>	Null (no limit)	The bandwidth (in bytes per second) to assign to partition reassignment

Cruise Control properties	KafkaRebalance properties	Default	Description
<code>default.replica.movement.strategies</code>	<code>replicaMovementStrategies</code>	<code>BaseReplicaMovementStrategy</code>	The list of strategies (in priority order) used to determine the order in which partition reassignment commands are executed for generated proposals. For the server setting, use a comma separated string with the fully qualified names of the strategy class (add <code>com.linkedin.kafka.cruisecontrol.executor.strategy</code> . to the start of each class name). For the KafkaRebalance resource setting use a YAML array of strategy class names.
-	<code>rebalanceDisk</code>	false	Enables intra-broker disk balancing, which balances disk space utilization between disks on the same broker. Only applies to Kafka deployments that use JBOD storage with multiple disks.

Changing the default settings affects the length of time that the rebalance takes to complete, as well

as the load placed on the Kafka cluster during the rebalance. Using lower values reduces the load but increases the amount of time taken, and vice versa.

#### *Additional resources*

- [CruiseControlSpec schema reference](#)
- [KafkaRebalanceSpec schema reference](#)

## 21.2. Deploying Cruise Control with Kafka

Configure a [Kafka](#) resource to deploy Cruise Control alongside a Kafka cluster. You can use the `cruiseControl` properties of the [Kafka](#) resource to configure the deployment. Deploy one instance of Cruise Control per Kafka cluster.

Use `goals` configuration in the Cruise Control `config` to specify optimization goals for generating optimization proposals. You can use `brokerCapacity` to change the default capacity limits for goals related to resource distribution. If brokers are running on nodes with heterogeneous network resources, you can use `overrides` to set network capacity limits for each broker.

If an empty object `({})` is used for the `cruiseControl` configuration, all properties use their default values.

Strimzi provides [example configuration files](#), which include [Kafka](#) custom resources with Cruise Control configuration. For more information on the configuration options for Cruise Control, see the [Strimzi Custom Resource API Reference](#).

#### *Prerequisites*

- [The Cluster Operator must be deployed.](#)

#### *Procedure*

1. Edit the `cruiseControl` property for the [Kafka](#) resource.

The properties you can configure are shown in this example configuration:

```
apiVersion: kafka.strimzi.io/v1
kind: Kafka
metadata:
  name: my-cluster
spec:
  # ...
  cruiseControl:
    brokerCapacity: ①
      inboundNetwork: 10000KB/s
      outboundNetwork: 10000KB/s
    overrides: ②
      - brokers: [0]
        inboundNetwork: 20000KiB/s
        outboundNetwork: 20000KiB/s
      - brokers: [1, 2]
        inboundNetwork: 30000KiB/s
```

```

    outboundNetwork: 30000KiB/s
    # ...
config: ③
    # Note that 'default.goals' (superset) must also include all 'hard.goals'
(subset)
    default.goals: > ④
        com.linkedin.kafka.cruisecontrol.analyzer.goals.RackAwareGoal,
        com.linkedin.kafka.cruisecontrol.analyzer.goals.ReplicaCapacityGoal,
        com.linkedin.kafka.cruisecontrol.analyzer.goals.DiskCapacityGoal
        # ...
    hard.goals: >
        com.linkedin.kafka.cruisecontrol.analyzer.goals.RackAwareGoal
        # ...
    cpu.balance.threshold: 1.1
    metadata.max.age.ms: 300000
    send.buffer.bytes: 131072
    webserver.http.cors.enabled: true ⑤
    webserver.http.cors.origin: "*"
    webserver.http.cors.exposeheaders: "User-Task-ID,Content-Type"
    # ...
resources: ⑥
    requests:
        cpu: 1
        memory: 512Mi
    limits:
        cpu: 2
        memory: 2Gi
logging: ⑦
    type: inline
    loggers:
        rootLogger.level: INFO
template: ⑧
    pod:
        metadata:
            labels:
                label1: value1
        securityContext:
            runAsUser: 1000001
            fsGroup: 0
            terminationGracePeriodSeconds: 120
readinessProbe: ⑨
    initialDelaySeconds: 15
    timeoutSeconds: 5
livenessProbe:
    initialDelaySeconds: 15
    timeoutSeconds: 5
metricsConfig: ⑩
    type: jmxPrometheusExporter
    valueFrom:
        configMapKeyRef:
            name: cruise-control-metrics

```

```
key: metrics-config.yml
```

```
# ...
```

- ① Capacity limits for broker resources.
- ② Overrides set network capacity limits for specific brokers when running on nodes with heterogeneous network resources.
- ③ Cruise Control configuration. Standard Cruise Control configuration may be provided, restricted to those properties not managed directly by Strimzi.
- ④ Optimization goals configuration, which can include configuration for default optimization goals (`default.goals`), supported optimization goals (`goals`), and hard goals (`hard.goals`).
- ⑤ CORS enabled and configured for read-only access to the Cruise Control API.
- ⑥ Requests for reservation of supported resources, currently `cpu` and `memory`, and limits to specify the maximum resources that can be consumed.
- ⑦ Cruise Control loggers and log levels added directly (`inline`) or indirectly (`external`) through a `ConfigMap`. Custom Log4j configuration must be placed under the `log4j2.properties` key in the `ConfigMap`. You can set log levels to `INFO`, `ERROR`, `WARN`, `TRACE`, `DEBUG`, `FATAL` or `OFF`.
- ⑧ Template customization. Here a pod is scheduled with additional security attributes.
- ⑨ Healthchecks to know when to restart a container (liveness) and when a container can accept traffic (readiness).
- ⑩ Prometheus metrics enabled. In this example, metrics are configured for the Prometheus JMX Exporter (the default metrics exporter).

## 2. Create or update the resource:

```
kubectl apply -f <kafka_configuration_file>
```

## 3. Check the status of the deployment:

```
kubectl get deployments -n <my_cluster_operator_namespace>
```

*Output shows the deployment name and readiness*

NAME	READY	UP-TO-DATE	AVAILABLE
my-cluster-cruise-control	1/1	1	1

`my-cluster` is the name of the Kafka cluster.

`READY` shows the number of replicas that are ready/expected. The deployment is successful when the `AVAILABLE` output shows `1`.

### 21.2.1. Auto-created Cruise Control topics

The following table shows the three topics that are automatically created when Cruise Control is

deployed. These topics are required for Cruise Control to work properly and must not be deleted or changed. You can change the name of the topic using the specified configuration option.

Table 32. Topics created when Cruise Control is deployed

Auto-created topic configuration	Default topic name	Created by	Function
<code>metric.reporter.topic</code>	<code>strimzi.cruisecontrol.metrics</code>	Strimzi Metrics Reporter	Stores the raw metrics from the Metrics Reporter in each Kafka broker.
<code>partition.metric.sample.store.topic</code>	<code>strimzi.cruisecontrol.partitionmetricsamples</code>	Cruise Control	Stores the derived metrics for each partition. These are created by the <a href="#">Metric Sample Aggregator</a> .
<code>broker.metric.sample.store.topic</code>	<code>strimzi.cruisecontrol.modeltrainingsamples</code>	Cruise Control	Stores the metrics samples used to create the <a href="#">Cluster Workload Model</a> .

To prevent the removal of records that are needed by Cruise Control, log compaction is disabled in the auto-created topics.

**NOTE** If the names of the auto-created topics are changed in a Kafka cluster that already has Cruise Control enabled, the old topics will not be deleted and should be manually removed.

#### What to do next

After configuring and deploying Cruise Control, you can [generate optimization proposals](#).

## 21.3. Generating optimization proposals

When you create or update a [KafkaRebalance](#) resource, Cruise Control generates an optimization proposal for the Kafka cluster based on a set of optimization goals. Analyze the information in the optimization proposal and decide whether to approve it. You can use the results of the optimization proposal to rebalance your Kafka cluster.

This procedure covers using the following modes for generating optimization proposals related to rebalances:

- `full` (default)
- `add-brokers`
- `remove-brokers`

The mode you use depends on whether you are rebalancing across all the brokers already running in the Kafka cluster; or you want to rebalance after scaling up or before scaling down your Kafka cluster. For more information, see [Rebalancing modes with broker scaling](#).

#### Prerequisites

- You have [deployed Cruise Control](#) to your Strimzi cluster.
- You have configured optimization goals and, optionally, capacity limits on broker resources.

For more information on configuring Cruise Control, see [Deploying Cruise Control with Kafka](#).

#### Procedure

1. Create a [KafkaRebalance](#) resource and specify the appropriate mode.

#### **full mode (default)**

To use the *default optimization goals* defined in the [Kafka](#) resource, leave the `spec` property empty. Cruise Control rebalances a Kafka cluster in **full** mode by default.

*Example configuration with full rebalancing by default*

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaRebalance
metadata:
  name: my-rebalance
  labels:
    strimzi.io/cluster: my-cluster
spec: {}
```

You can also run a full rebalance by specifying the **full** mode through the `spec.mode` property.

*Example configuration specifying full mode*

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaRebalance
metadata:
  name: my-rebalance
  labels:
    strimzi.io/cluster: my-cluster
spec:
  mode: full
```

#### **add-brokers mode**

If you want to rebalance a Kafka cluster after scaling up, specify the **add-brokers** mode.

In this mode, existing replicas are moved to the newly added brokers. You need to specify the brokers as a list.

*Example configuration specifying add-brokers mode*

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaRebalance
metadata:
  name: my-rebalance
  labels:
    strimzi.io/cluster: my-cluster
```

```
spec:  
  mode: add-brokers  
  brokers: [3, 4] ①
```

① List of newly added brokers added by the scale up operation. This property is mandatory.

### remove-brokers mode

If you want to rebalance a Kafka cluster before scaling down, specify the `remove-brokers` mode.

In this mode, replicas are moved off the brokers that are going to be removed. You need to specify the brokers that are being removed as a list.

*Example configuration specifying `remove-brokers` mode*

```
apiVersion: kafka.strimzi.io/v1  
kind: KafkaRebalance  
metadata:  
  name: my-rebalance  
  labels:  
    strimzi.io/cluster: my-cluster  
spec:  
  mode: remove-brokers  
  brokers: [3, 4] ①
```

① List of brokers to be removed by the scale down operation. This property is mandatory.

**NOTE**

The following steps and the steps to approve or stop a rebalance are the same regardless of the rebalance mode you are using.

2. To configure proposal-specific optimization goals instead of using the default goals, add the `goals` property and enter one or more goals.

In the following example, rack awareness and replica capacity are configured as proposal-specific optimization goals:

```
apiVersion: kafka.strimzi.io/v1  
kind: KafkaRebalance  
metadata:  
  name: my-rebalance  
  labels:  
    strimzi.io/cluster: my-cluster  
spec:  
  goals:  
    - RackAwareGoal  
    - ReplicaCapacityGoal
```

3. To ignore the configured hard goals, add the `skipHardGoalCheck: true` property:

```

apiVersion: kafka.strimzi.io/v1
kind: KafkaRebalance
metadata:
  name: my-rebalance
  labels:
    strimzi.io/cluster: my-cluster
spec:
  goals:
    - RackAwareGoal
    - ReplicaCapacityGoal
  skipHardGoalCheck: true

```

4. (Optional) To approve the optimization proposal automatically, set the `strimzi.io/rebalance-auto-approval` annotation to `true`:

```

apiVersion: kafka.strimzi.io/v1
kind: KafkaRebalance
metadata:
  name: my-rebalance
  labels:
    strimzi.io/cluster: my-cluster
  annotations:
    strimzi.io/rebalance-auto-approval: "true"
spec:
  goals:
    - RackAwareGoal
    - ReplicaCapacityGoal
  skipHardGoalCheck: true

```

5. Create or update the resource:

```
kubectl apply -f <kafka_rebalance_configuration_file>
```

The Cluster Operator requests the optimization proposal from Cruise Control. This might take a few minutes depending on the size of the Kafka cluster.

6. If you used the automatic approval mechanism, wait for the status of the optimization proposal to change to `Ready`. If you haven't enabled the automatic approval mechanism, wait for the status of the optimization proposal to change to `ProposalReady`:

```
kubectl get kafkarebalance -o wide -w -n <namespace>
```

### PendingProposal

A `PendingProposal` status means the rebalance operator is polling the Cruise Control API to check if the optimization proposal is ready.

## ProposalReady

A `ProposalReady` status means the optimization proposal is ready for review and approval.

When the status changes to `ProposalReady`, the optimization proposal is ready to approve.

### 7. Review the optimization proposal.

The optimization proposal is contained in the `Status.Optimization Result` property of the `KafkaRebalance` resource.

```
kubectl describe kafka_rebalance <kafka_rebalance_resource_name>
```

*Example optimization proposal*

```
Status:  
Conditions:  
  Last Transition Time: 2020-05-19T13:50:12.533Z  
  Status: ProposalReady  
  Type: State  
  Observed Generation: 1  
Optimization Result:  
  Data To Move MB: 0  
  Excluded Brokers For Leadership:  
  Excluded Brokers For Replica Move:  
  Excluded Topics:  
    Intra Broker Data To Move MB: 0  
    Monitored Partitions Percentage: 100  
    Num Intra Broker Replica Movements: 0  
    Num Leader Movements: 0  
    Num Replica Movements: 26  
    On Demand Balancedness Score After: 81.8666802863978  
    On Demand Balancedness Score Before: 78.01176356230222  
    Recent Windows: 1  
Session Id: 05539377-ca7b-45ef-b359-e13564f1458c
```

The properties in the `Optimization Result` section describe the pending cluster rebalance operation. For descriptions of each property, see [Contents of optimization proposals](#).

## Insufficient CPU capacity

If a Kafka cluster is overloaded in terms of CPU utilization, you might see an insufficient CPU capacity error in the `KafkaRebalance` status. It's worth noting that this utilization value is unaffected by the `excludedTopics` configuration. Although optimization proposals will not reassign replicas of excluded topics, their load is still considered in the utilization calculation.

*Example CPU utilization error*

```
com.linkedin.kafka.cruisecontrol.exception.OptimizationFailureException:  
[CpuCapacityGoal] Insufficient capacity for cpu (Utilization 615.21, Allowed Capacity  
420.00, Threshold: 0.70). Add at least 3 brokers with the same cpu capacity (100.00)
```

as broker-0. Add at least 3 brokers with the same cpu capacity (100.00) as broker-0.

**NOTE**

The error shows CPU capacity as a percentage rather than the number of CPU cores. For this reason, it does not directly map to the number of CPUs configured in the Kafka custom resource. It is like having a single *virtual* CPU per broker, which has the cycles of the CPUs configured in `Kafka.spec.kafka.resources.limits.cpu`. This has no effect on the rebalance behavior, since the ratio between CPU utilization and capacity remains the same.

*What to do next*

[Approving optimization proposals](#)

*Additional resources*

- [Optimization proposals](#)

## 21.4. Approving optimization proposals

You can approve an [optimization proposal](#) generated by Cruise Control, if its status is `ProposalReady`. Cruise Control will then apply the optimization proposal to the Kafka cluster, reassigning partitions to brokers and changing partition leadership.

**This is not a dry run.** Before you approve an optimization proposal, you must:

**CAUTION**

- Refresh the proposal in case it has become out of date.
- Carefully review the [contents of the proposal](#).

*Prerequisites*

- You have [generated an optimization proposal](#) from Cruise Control.
- The `KafkaRebalance` custom resource status is `ProposalReady`.

*Procedure*

Perform these steps for the optimization proposal that you want to approve.

1. Unless the optimization proposal is newly generated, check that it is based on current information about the state of the Kafka cluster. To do so, annotate the `KafkaRebalance` resource to refresh the optimization proposal and make sure it uses the latest cluster metrics:

```
kubectl annotate kafka-rebalance <kafka_rebalance_resource_name>
strimzi.io/rebalance="refresh"
```

2. Wait for the status of the optimization proposal to change to `ProposalReady`:

```
kubectl get kafka-rebalance -o wide -w -n <namespace>
```

### PendingProposal

A `PendingProposal` status means the rebalance operator is polling the Cruise Control API to check if the optimization proposal is ready.

### ProposalReady

A `ProposalReady` status means the optimization proposal is ready for review and approval.

When the status changes to `ProposalReady`, the optimization proposal is ready to approve.

3. Annotate the `KafkaRebalance` resource to approve the optimization proposal:

```
kubectl annotate kafka-rebalance <kafka_rebalance_resource_name>
strimzi.io/rebalance="approve"
```

4. The Cluster Operator detects the annotated resource and instructs Cruise Control to rebalance the Kafka cluster.
5. Wait for the status of the optimization proposal to change to `Ready`:

```
kubectl get kafka-rebalance -o wide -w -n <namespace>
```

### Rebalancing

A `Rebalancing` status means the rebalancing is in progress.

### Ready

A `Ready` status means the rebalance is complete.

### NotReady

A `NotReady` status means an error occurred—see [Fixing problems with a KafkaRebalance resource](#).

When the status changes to `Ready`, the rebalance is complete.

To use the same `KafkaRebalance` custom resource to generate another optimization proposal, apply the `refresh` annotation to the custom resource. This moves the custom resource to the `PendingProposal` or `ProposalReady` state. You can then review the optimization proposal and approve it, if desired.

#### *Additional resources*

- [Optimization proposals](#)
- [Stopping rebalances](#)

## 21.5. Tracking rebalances

You can track the progress of a partition rebalance using status information in the `KafkaRebalance` resource. Tracking an active partition rebalance supports planning of cluster operations, including worker node maintenance, scaling, and broker upgrades. Understanding key details, such as

duration and the remaining data to transfer, helps with scheduling maintenance windows and assessing the impact of proceeding or canceling the rebalance.

When a rebalance is in progress, Strimzi stores progress information in a [ConfigMap](#). The [ConfigMap](#) is referenced by the `status.progress.rebalanceProgressConfigMap` property of the [KafkaRebalance](#) resource and has the same name as the [KafkaRebalance](#) resource. The [ConfigMap](#) includes the following fields:

- `estimatedTimeToCompletionInMinutes`: The estimated time it will take in minutes until the partition rebalance is complete.
- `completedByteMovementPercentage`: The percentage of data movement completed (in bytes), as a rounded down integer from 0-100.
- `executorState.json`: The “non-verbose” JSON payload from the `/kafkacruisecontrol/state?substates=executor` endpoint, providing details about the executor’s current status, including partition movement progress, concurrency limits, and total data to move.

The information is updated on each reconciliation of the [KafkaRebalance](#) resource.

#### *Prerequisites*

- The [KafkaRebalance](#) custom resource is in a [Rebalancing](#) state.

#### *Procedure*

- Accessing `estimatedTimeToCompletionInMinutes` field.:  

```
kubectl get configmaps <my_rebalance> -o
jsonpath="[['data']]['estimatedTimeToCompletionInMinutes']]"
```

- Accessing `completedByteMovementPercentage` field.:  

```
kubectl get configmaps <my_rebalance> -o
jsonpath="[['data']]['completedByteMovementPercentage']]"
```

- Accessing `executorState.json` field.  

```
kubectl get configmaps <my_rebalance> -o
jsonpath="[['data']]['executorState.json']]"
```

## 21.6. Stopping rebalances

Once started, a cluster rebalance operation might take some time to complete and affect the overall performance of the Kafka cluster.

If you want to stop a cluster rebalance operation that is in progress, apply the `stop` annotation to the [KafkaRebalance](#) custom resource. This instructs Cruise Control to finish the current batch of

partition reassessments and then stop the rebalance. When the rebalance has stopped, completed partition reassessments have already been applied; therefore, the state of the Kafka cluster is different when compared to prior to the start of the rebalance operation. If further rebalancing is required, you should generate a new optimization proposal.

**NOTE**

The performance of the Kafka cluster in the intermediate (stopped) state might be worse than in the initial state.

*Prerequisites*

- The status of the [KafkaRebalance](#) custom resource is [Rebalancing](#).

*Procedure*

1. Annotate the [KafkaRebalance](#) resource to stop the rebalance:

```
kubectl annotate kafka-rebalance <kafka_rebalance_resource_name>  
strimzi.io/rebalance="stop"
```

2. Check the status of the [KafkaRebalance](#) resource:

```
kubectl describe kafka-rebalance <kafka_rebalance_resource_name>
```

3. Wait until the status changes to [Stopped](#).

*Additional resources*

- [Optimization proposals](#)

## 21.7. Troubleshooting and refreshing rebalances

When creating a [KafkaRebalance](#) resource or interacting with Cruise Control, errors are reported in the resource status, along with guidance on how to fix them. In such cases, the resource transitions to the [NotReady](#) state.

To continue with a cluster rebalance operation, you must rectify any configuration issues in the [KafkaRebalance](#) resource or address any problems with the Cruise Control deployment.

Common issues include the following:

- Misconfigured parameters in the [KafkaRebalance](#) resource.
- The [strimzi.io/cluster](#) label for specifying the Kafka cluster in the [KafkaRebalance](#) resource is missing.
- The Cruise Control server is not deployed as the [cruiseControl](#) property in the [Kafka](#) resource is missing.
- The Cruise Control server is not reachable.

After fixing any issues, you need to add the [refresh](#) annotation to the [KafkaRebalance](#) resource.

During a “refresh”, a new optimization proposal is requested from the Cruise Control server.

#### *Prerequisites*

- You have [approved an optimization proposal](#).
- The status of the [KafkaRebalance](#) custom resource for the rebalance operation is [NotReady](#).

#### *Procedure*

1. Get information about the error from the [KafkaRebalance](#) status:

```
kubectl describe kafka-rebalance <kafka_rebalance_resource_name>
```

2. Attempt to resolve the issue by annotating the [KafkaRebalance](#) resource to refresh the proposal:

```
kubectl annotate kafka-rebalance <kafka_rebalance_resource_name>
strimzi.io/rebalance="refresh"
```

3. Check the status of the [KafkaRebalance](#) resource:

```
kubectl describe kafka-rebalance <kafka_rebalance_resource_name>
```

4. Wait until the status changes to [PendingProposal](#), or directly to [ProposalReady](#).

# Chapter 22. Using Cruise Control to modify topic replication factor

Change the replication factor of topics by updating the [KafkaTopic](#) resource managed by the Topic Operator. You can adjust the replication factor for specific purposes, such as:

- Setting a lower replication factor for non-critical topics or because of resource shortages
- Setting a higher replication factor to improve data durability and fault tolerance

The Topic Operator uses Cruise Control to make the necessary changes, so Cruise Control must be deployed with Strimzi.

The Topic Operator watches and periodically reconciles all managed and unpause Kafka resources to detect changes to `.spec.replicas` configuration by comparing the replication factor of the topic in Kafka. One or more replication factor updates are then sent to Cruise Control for processing in a single request.

Progress is reflected in the status of the [KafkaTopic](#) resource.

## Prerequisites

- [The Cluster Operator must be deployed](#).
- [The Topic Operator must be deployed](#) to manage topics through the [KafkaTopic](#) custom resource.
- [Cruise Control is deployed with Kafka](#).

## Procedure

1. Edit the [KafkaTopic](#) resource to change the `replicas` value.

In this procedure, we change the `replicas` value for `my-topic` from 1 to 3.

### Kafka topic replication factor configuration

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaTopic
metadata:
  name: my-topic
  labels:
    strimzi.io/cluster: my-cluster
spec:
  partitions: 1
  replicas: 3
  # ...
```

2. Apply the change to the [KafkaTopic](#) configuration and wait for the Topic Operator to update the topic.
3. Check the status of the [KafkaTopic](#) resource to make sure the request was successful:

```
oc get kafkatopics my-topic -o yaml
```

*Status for the replication factor change*

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaTopic
metadata:
  name: my-topic
  labels:
    strimzi.io/cluster: my-cluster
spec:
  partitions: 1
  replicas: 3
  # ...
# ...
status:
  conditions:
  - lastTransitionTime: "2024-01-18T16:13:50.490918232Z"
    status: "True"
    type: Ready
  observedGeneration: 2
  replicasChange:
    sessionId: 1aa418ca-53ed-4b93-b0a4-58413c4fc0cb ①
    state: ongoing ②
    targetReplicas: 3 ③
  topicName: my-topic
```

- ① The session ID for the Cruise Control operation, which is shown when process moves out of a pending state.
- ② The state of the update. Moves from `pending` to `ongoing`, and then the entire `replicasChange` status is removed when the change is complete.
- ③ The requested change to the number of replicas.

An error message is shown in the status if the request fails before completion. The request is periodically retried if it enters a failed state.

#### *Changing topic replication factor using the standalone Topic Operator*

If you are using the standalone Topic Operator and aim to change the topic replication factor through configuration, you still need to use the Topic Operator in unidirectional mode alongside a Cruise Control deployment. You also need to include the following environment variables in the standalone Topic Operator deployment so that it can integrate with Cruise Control.

#### *Example standalone Topic Operator deployment configuration*

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: strimzi-topic-operator
```

```

labels:
  app: strimzi
spec:
# ...
template:
# ...
spec:
# ...
containers:
- name: strimzi-topic-operator
# ...
env:
# ...
- name: STRIMZI_CRUISE_CONTROL_ENABLED ①
  value: true
- name: STRIMZI_CRUISE_CONTROL_RACK_ENABLED ②
  value: false
- name: STRIMZI_CRUISE_CONTROL_HOSTNAME ③
  value: cruise-control-api.namespace.svc
- name: STRIMZI_CRUISE_CONTROL_PORT ④
  value: 9090
- name: STRIMZI_CRUISE_CONTROL_SSL_ENABLED ⑤
  value: true
- name: STRIMZI_CRUISE_CONTROL_AUTH_ENABLED ⑥
  value: true

```

① Integrates Cruise Control with the Topic Operator.

② Flag to indicate whether rack awareness is enabled on the Kafka cluster. If so, replicas can be spread across different racks, data centers, or availability zones.

③ Cruise Control hostname.

④ Cruise control port.

⑤ Enables TLS authentication and encryption for accessing the Kafka cluster.

⑥ Enables basic authorization for accessing the Cruise Control API.

If you enable TLS authentication and authorization, mount the required certificates as follows:

- Public certificates of the Cluster CA (certificate authority) in `/etc/cluster-ca-certs/ca.crt`
- Basic authorization credentials (user name and password) in `/etc/eto-cc-api/topic-operator.apiAdminName` and `/etc/eto-cc-api/topic-operator.apiAdminPassword`

# Chapter 23. Using Cruise Control to reassign partitions on JBOD disks

If you are using JBOD storage and have Cruise Control installed with Strimzi, you can reassign partitions between the JBOD disks used for storage on the same broker. This capability also allows you to remove JBOD disks without data loss.

To reassign partitions, configure a [KafkaRebalance](#) resource in `remove-disks` mode and specify a list of broker IDs with corresponding volume IDs for partition reassignment. Cruise Control generates an optimization proposal based on the configuration and reassigns the partitions when approved manually or automatically.

Use the Kafka [kafka-log-dirs.sh](#) tool to check information about Kafka topic partitions and their location on brokers before and after moving them. Use an interactive pod to avoid running the tool within the broker container and causing any disruptions.

## Prerequisites

- [The Cluster Operator must be deployed.](#)
- [Cruise Control is deployed with Kafka.](#)
- Kafka operates in KRaft mode and brokers use JBOD storage.
- More than one JBOD disk must be configured on the broker. For more information on configuring Kafka storage, see [Configuring Kafka storage](#).

In this procedure, we use a Kafka cluster named `my-cluster`, which is deployed to the `my-project` namespace with node pools and Cruise Control enabled.

## Example Kafka cluster configuration

```
apiVersion: kafka.strimzi.io/v1
kind: Kafka
metadata:
  name: my-cluster
  namespace: my-project
spec:
  kafka:
    # ...
  cruiseControl: {}
    # ...
```

A node pool named `pool-a` is configured with three broker replicas that use three JBOD storage volumes. In the procedure, we show how partitions are reassigned from volume 1 and 2 to volume 0.

## Example node pool configuration with JBOD storage

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaNodePool
```

```

metadata:
  name: pool-a
  labels:
    strimzi.io/cluster: my-cluster
spec:
  replicas: 3
  roles:
    - broker
  storage:
    type: jbod
    volumes:
      - id: 0
        type: persistent-claim
        size: 2000Gi
        deleteClaim: false
      - id: 1
        type: persistent-claim
        size: 2000Gi
        deleteClaim: false
      - id: 2
        type: persistent-claim
        size: 2000Gi
        deleteClaim: false
# ...

```

### *Procedure*

1. Run a new interactive pod container using the Kafka image to connect to a running Kafka broker.

```
kubectl run --restart=Never --image=quay.io/strimzi/kafka:0.50.0-kafka-4.1.1
helper-pod -- /bin/sh -c "sleep 3600"
```

In this procedure, we use a pod named **helper-pod**.

2. (Optional) Check the partition replica data on broker 0 by opening a terminal inside the interactive pod and running the Kafka **kafka-log-dirs.sh** tool:

```
kubectl exec -n myproject -ti my-cluster-pool-a-0 bin/kafka-log-dirs.sh --describe
--bootstrap-server my-cluster-kafka-bootstrap:9092 --broker-list 0,1,2 --topic-list
my-topic
```

**my-cluster-pool-a-0** is the pod name for broker 0. The tool returns topic information for each log directory. In this example, we are restricting the information to **my-topic** to show the steps against a single topic. The JBOD volumes used for log directories are mounted at **/var/lib/kafka/data-<volume\_id>/kafka-log<pod\_id>**.

*Example output data for each log directory*

```
{  
  "brokers": [  
    {  
      "broker": 0, ①  
      "logDirs": [  
        {  
          "partitions": [ ②  
            {  
              "partition": "my-topic-5",  
              "size": 0,  
              "offsetLag": 0,  
              "isFuture": false  
            },  
            {  
              "partition": "my-topic-2",  
              "size": 0,  
              "offsetLag": 0,  
              "isFuture": false  
            }  
          ],  
          "error": null, ③  
          "logDir": "/var/lib/kafka/data-2/kafka-log0" ④  
        },  
        {  
          "partitions": [  
            {  
              "partition": "my-topic-0",  
              "size": 0,  
              "offsetLag": 0,  
              "isFuture": false  
            },  
            {  
              "partition": "my-topic-3",  
              "size": 0,  
              "offsetLag": 0,  
              "isFuture": false  
            }  
          ],  
          "error": null,  
          "logDir": "/var/lib/kafka/data-0/kafka-log0"  
        },  
        {  
          "partitions": [  
            {  
              "partition": "my-topic-4",  
              "size": 0,  
              "offsetLag": 0,  
              "isFuture": false  
            },  
            {  
              "partition": "my-topic-1",  
              "size": 0,  
              "offsetLag": 0,  
              "isFuture": false  
            }  
          ]  
        }  
      ]  
    }  
  ]  
}
```

```
{
    "partition": "my-topic-1",
    "size": 0,
    "offsetLag": 0,
    "isFuture": false
}
],
"error": null,
"logDir": "/var/lib/kafka/data-1/kafka-log0"
}
]
}
```

- ① The broker ID.
  - ② Partition details: name, size, offset lag. The (`isFuture`) property indicates that the partition is moving between log directories when showing as `true`.
  - ③ If `error` is not `null`, there is an issue with the disk hosting the log directory.
  - ④ The path and name of the log directory.
3. Create a `KafkaRebalance` resource in `remove-disks` mode, listing the brokers and volume IDs to reassign partitions from. Without specific configuration, the default rebalance goals are used.

*Example Cruise Control configuration*

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaRebalance
metadata:
  name: my-rebalance
  labels:
    strimzi.io/cluster: my-cluster
spec:
  mode: remove-disks
  moveReplicasOffVolumes:
    - brokerId: 0 ①
      volumeIds: [1, 2] ②
```

- ① The broker from which to reassign partitions.
  - ② The volume IDs to reassign partitions from.
- In this example, `my-rebalance` reassigns partitions from volumes with IDs 1 and 2 on broker 0.
4. (Optional) To approve the optimization proposal automatically, set the `strimzi.io/rebalance-auto-approval` annotation to `true`:

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaRebalance
metadata:
```

```

name: my-rebalance
labels:
  strimzi.io/cluster: my-cluster
annotations:
  strimzi.io/rebalance-auto-approval: "true"
spec:
  mode: remove-disks
  moveReplicasOffVolumes:
    - brokerId: 0
      volumeIds: [1, 2]

```

5. Apply the **KafkaRebalance** configuration.
6. If manually approving, wait for the status of the proposal to move to **ProposalReady** before approving the changes.
  - a. Check the summary of the changes in the **KafkaRebalance** status:

```
kubectl get kafka-rebalance my-rebalance -n my-project -o yaml
```

*Example summary of changes*

```

apiVersion: kafka.strimzi.io/v1
kind: KafkaRebalance
metadata:
  name: my-rebalance
  labels:
    strimzi.io/cluster: my-cluster
spec:
  mode: remove-disks
  moveReplicasOffVolumes:
    - brokerId: 0
      volumeIds: [1, 2]
status:
  - lastTransitionTime: "2024-11-13T06:55:42.217794891Z"
    status: "True"
    type: ProposalReady
  observedGeneration: 1
  optimizationResult:
    afterBeforeLoadConfigMap: my-rebalance
    dataToMoveMB: 0
    excludedBrokersForLeadership: []
    excludedBrokersForReplicaMove: []
    excludedTopics: []
    intraBrokerDataToMoveMB: 0
    monitoredPartitionsPercentage: 100
    numIntraBrokerReplicaMovements: 26
    numLeaderMovements: 0
    numReplicaMovements: 0
    onDemandBalancednessScoreAfter: 100

```

```
onDemandBalancednessScoreBefore: 0
provisionRecommendation: ""
provisionStatus: UNDECIDED
recentWindows: 1
sessionId: 24537b9c-a315-4715-8e86-01481e914771
```

**NOTE**

The summary only shows the changes after optimization, not the load before optimization.

- b. Annotate the `KafkaRebalance` resource to approve the changes:

```
kubectl annotate kafka my-rebalance strimzi.io/rebalance="approve"
```

7. Wait for the status of the proposal to change to `Ready`.
8. Use the Kafka `kafka-log-dirs.sh` tool again to verify data movement.

In this example, the log directories for volumes 1 and 2 no longer have partitions assigned to them and volume 0 holds 6 partitions for `my-topic`, indicating that the partitions have been successfully reassigned.

*Example output data following reassignment of partitions*

```
{
  "brokers": [
    {
      "broker": 0,
      "logDirs": [
        {
          "partitions": [],
          "error": null,
          "logDir": "/var/lib/kafka/data-2/kafka-log0"
        },
        {
          "partitions": [
            {
              "partition": "my-topic-4",
              "size": 0,
              "offsetLag": 0,
              "isFuture": false
            },
            {
              "partition": "my-topic-5",
              "size": 0,
              "offsetLag": 0,
              "isFuture": false
            },
            {
              "partition": "my-topic-0",
              "size": 0,
              "offsetLag": 0,
              "isFuture": false
            }
          ]
        }
      ]
    }
  ]
}
```

```

        "size": 0,
        "offsetLag": 0,
        "isFuture": false
    },
    {
        "partition": "my-topic-1",
        "size": 0,
        "offsetLag": 0,
        "isFuture": false
    },
    {
        "partition": "my-topic-2",
        "size": 0,
        "offsetLag": 0,
        "isFuture": false
    },
    {
        "partition": "my-topic-3",
        "size": 0,
        "offsetLag": 0,
        "isFuture": false
    }
],
"error": null,
"logDir": "/var/lib/kafka/data-0/kafka-log0"
},
{
    "partitions": [],
    "error": null,
    "logDir": "/var/lib/kafka/data-1/kafka-log0"
}
]
}

```

9. To prevent empty volumes from being used in future rebalances or topic allocations, update the configuration and remove the associated persistent volume claims (PVCs).
  - a. Update the node pool configuration to exclude the volumes.

**WARNING**

Before making changes, verify that all partitions have been successfully moved using `kafka-log-dirs.sh`. Removing volumes prematurely can cause data loss.

In this example, volumes 1 and 2 are removed, and only volume 0 is retained:

*Updated node pool configuration with single volume JBOD storage*

```

apiVersion: kafka.strimzi.io/v1
kind: KafkaNodePool
metadata:
  name: pool-a

```

```
labels:
  strimzi.io/cluster: my-cluster
spec:
  replicas: 3
  roles:
    - broker
  storage:
    type: jbod
    volumes:
      - id: 0
        type: persistent-claim
        size: 2000Gi
        deleteClaim: false
# ...
```

b. Delete the unused PVCs.

Unless `deleteClaim: true` is set in the volume configuration, the PVCs are not deleted automatically when the node pool configuration is updated, and you have to delete them manually.

PVCs are named using the format `data-<id>-<kafka_cluster_name>-kafka-<pod_id>`. You can list them using:

```
kubectl get pvc -n my-project
```

Then delete the unused PVCs:

```
kubectl delete pvc data-<id>-<kafka_cluster_name>-kafka-<pod_id> -n my-project
```

**NOTE** Deleting a PVC removes the underlying storage.

c. (Optional) Delete the helper pod used earlier:

```
kubectl delete pod helper-pod -n my-project
```

# Chapter 24. Using the partition reassignment tool

You can use the `kafka-reassign-partitions.sh` tool for the following:

- Adding or removing brokers
- Reassigning partitions across brokers
- Changing the replication factor of topics

However, while `kafka-reassign-partitions.sh` supports these operations, it is generally easier with Cruise Control. Cruise Control can move topics from one broker to another without any downtime, and it is the most efficient way to reassign partitions.

To use the `kafka-reassign-partitions.sh` tool, run it as a separate interactive pod rather than within the broker container. Running the Kafka `bin/` scripts within the broker container may cause a JVM to start with the same settings as the Kafka broker, which can potentially cause disruptions. By running the `kafka-reassign-partitions.sh` tool in a separate pod, you can avoid this issue. Running a pod with the `-ti` option creates an interactive pod with a terminal for running shell commands inside the pod.

*Running an interactive pod with a terminal*

```
kubectl run helper-pod -ti --image=quay.io/stimzi/kafka:0.50.0-kafka-4.1.1 --rm=true  
--restart=Never -- bash
```

## 24.1. Partition reassignment tool overview

The partition reassignment tool provides the following capabilities for managing Kafka partitions and brokers:

### Redistributing partition replicas

Scale your cluster up and down by adding or removing brokers, and move Kafka partitions from heavily loaded brokers to under-utilized brokers. To do this, you must create a partition reassignment plan that identifies which topics and partitions to move and where to move them. Cruise Control is recommended for this type of operation as it [automates the cluster rebalancing process](#).

### Scaling topic replication factor up and down

Increase or decrease the replication factor of your Kafka topics. To do this, you must create a partition reassignment plan that identifies the existing replication assignment across partitions and an updated assignment with the replication factor changes.

### Changing the preferred leader

Change the preferred leader of a Kafka partition. In Kafka, the partition leader is the only partition that accepts writes from message producers, and therefore has the most complete log across all replicas.

Changing the preferred leader can be useful if you want to redistribute load across the brokers in the cluster. If the preferred leader is unavailable, another in-sync replica is automatically elected as leader, or the partition goes offline if there are no in-sync replicas. A background thread moves the leader role to the preferred replica when it is in sync. Therefore, changing the preferred replicas only makes sense in the context of a cluster rebalancing.

To do this, you must create a partition reassignment plan that specifies the new preferred leader for each partition by changing the order of replicas. In Kafka's leader election process, the preferred leader is prioritized by the order of replicas. The first broker in the order of replicas is designated as the preferred leader. This designation is important for load balancing by distributing partition leaders across the Kafka cluster. However, this alone might not be sufficient for optimal load balancing, as some partitions may have higher usage than others. Cruise Control can help address this by providing more comprehensive cluster rebalancing.

### Changing the log directories to use a specific JBOD volume

Change the log directories of your Kafka brokers to use a specific JBOD volume. This can be useful if you want to move your Kafka data to a different disk or storage device. To do this, you must create a partition reassignment plan that specifies the new log directory for each topic.

#### 24.1.1. Generating a partition reassignment plan

The partition reassignment tool ([kafka-reassign-partitions.sh](#)) works by generating a partition assignment plan that specifies which partitions should be moved from their current broker to a new broker.

If you are satisfied with the plan, you can execute it. The tool then does the following:

- Migrates the partition data to the new broker
- Updates the metadata on the Kafka brokers to reflect the new partition assignments
- Triggers a rolling restart of the Kafka brokers to ensure that the new assignments take effect

The partition reassignment tool has three different modes:

##### --generate

Takes a set of topics and brokers and generates a *reassignment JSON file* which will result in the partitions of those topics being assigned to those brokers. Because this operates on whole topics, it cannot be used when you only want to reassign some partitions of some topics.

##### --execute

Takes a *reassignment JSON file* and applies it to the partitions and brokers in the cluster. Brokers that gain partitions as a result become followers of the partition leader. For a given partition, once the new broker has caught up and joined the ISR (in-sync replicas) the old broker will stop being a follower and will delete its replica.

##### --verify

Using the same *reassignment JSON file* as the [--execute](#) step, [--verify](#) checks whether all the partitions in the file have been moved to their intended brokers. If the reassignment is complete, [--verify](#) also removes any traffic throttles ([--throttle](#)) that are in effect. Unless removed,

throttles will continue to affect the cluster even after the reassignment has finished.

It is only possible to have one reassignment running in a cluster at any given time, and it is not possible to cancel a running reassignment. If you must cancel a reassignment, wait for it to complete and then perform another reassignment to revert the effects of the first reassignment. The `kafka-reassign-partitions.sh` will print the reassignment JSON for this reversion as part of its output. Very large reassessments should be broken down into a number of smaller reassessments in case there is a need to stop in-progress reassignment.

#### 24.1.2. Specifying topics in a partition reassignment JSON file

The `kafka-reassign-partitions.sh` tool uses a reassignment JSON file that specifies the topics to reassign. You can generate a reassignment JSON file or create a file manually if you want to move specific partitions.

A basic reassignment JSON file has the structure presented in the following example, which describes three partitions belonging to two Kafka topics. Each partition is reassigned to a new set of replicas, which are identified by their broker IDs. The `version`, `topic`, `partition`, and `replicas` properties are all required.

*Example partition reassignment JSON file structure*

```
{
  "version": 1, ①
  "partitions": [ ②
    {
      "topic": "example-topic-1", ③
      "partition": 0, ④
      "replicas": [1, 2, 3] ⑤
    },
    {
      "topic": "example-topic-1",
      "partition": 1,
      "replicas": [2, 3, 4]
    },
    {
      "topic": "example-topic-2",
      "partition": 0,
      "replicas": [3, 4, 5]
    }
  ]
}
```

① The version of the reassignment JSON file format. Currently, only version 1 is supported, so this should always be 1.

② An array that specifies the partitions to be reassigned.

③ The name of the Kafka topic that the partition belongs to.

④ The ID of the partition being reassigned.

- ⑤ An ordered array of the IDs of the brokers that should be assigned as replicas for this partition. The first broker in the list is the leader replica.

**NOTE** Partitions not included in the JSON are not changed.

If you specify only topics using a `topics` array, the partition reassignment tool reassigns all the partitions belonging to the specified topics.

*Example reassignment JSON file structure for reassigning all partitions for a topic*

```
{  
  "version": 1,  
  "topics": [  
    { "topic": "my-topic"}  
  ]  
}
```

### 24.1.3. Reassigning partitions between JBOD volumes

When using JBOD storage in your Kafka cluster, you can reassign the partitions between specific volumes and their log directories (each volume has a single log directory).

To reassign a partition to a specific volume, add `log_dirs` values for each partition in the reassignment JSON file. Each `log_dirs` array contains the same number of entries as the `replicas` array, since each replica should be assigned to a specific log directory. The `log_dirs` array contains either an absolute path to a log directory or the special value `any`. The `any` value indicates that Kafka can choose any available log directory for that replica, which can be useful when reassigning partitions between JBOD volumes.

*Example reassignment JSON file structure with log directories*

```
{  
  "version": 1,  
  "partitions": [  
    {  
      "topic": "example-topic-1",  
      "partition": 0,  
      "replicas": [1, 2, 3]  
      "log_dirs": ["/var/lib/kafka/data-0/kafka-log1", "any", "/var/lib/kafka/data-  
1/kafka-log2"]  
    },  
    {  
      "topic": "example-topic-1",  
      "partition": 1,  
      "replicas": [2, 3, 4]  
      "log_dirs": ["any", "/var/lib/kafka/data-2/kafka-log3", "/var/lib/kafka/data-  
3/kafka-log4"]  
    },  
    {  
      "topic": "example-topic-2",  
      "partition": 2,  
      "replicas": [3, 4, 5]  
      "log_dirs": ["/var/lib/kafka/data-4/kafka-log5", "any", "/var/lib/kafka/data-  
5/kafka-log6"]  
    }  
  ]  
}
```

```
        "partition": 0,
        "replicas": [3, 4, 5]
        "log_dirs": ["/var/lib/kafka/data-4/kafka-log5", "any", "/var/lib/kafka/data-5/kafka-log6"]
    }
]
}
```

#### 24.1.4. Throttling partition reassignment

Partition reassignment can be a slow process because it involves transferring large amounts of data between brokers. To avoid a detrimental impact on clients, you can throttle the reassignment process. Use the `--throttle` parameter with the `kafka-reassign-partitions.sh` tool to throttle a reassignment. You specify a maximum threshold in bytes per second for the movement of partitions between brokers. For example, `--throttle 5000000` sets a maximum threshold for moving partitions of 50 MBps.

Throttling might cause the reassignment to take longer to complete.

- If the throttle is too low, the newly assigned brokers will not be able to keep up with records being published and the reassignment will never complete.
- If the throttle is too high, clients will be impacted.

For example, for producers, this could manifest as higher than normal latency waiting for acknowledgment. For consumers, this could manifest as a drop in throughput caused by higher latency between polls.

## 24.2. Generating a reassignment JSON file to reassign partitions

Generate a reassignment JSON file with the `kafka-reassign-partitions.sh` tool to reassign partitions after scaling a Kafka cluster. Adding or removing brokers does not automatically redistribute the existing partitions. To balance the partition distribution and take full advantage of the new brokers, you can reassign the partitions using the `kafka-reassign-partitions.sh` tool.

You run the tool from an interactive pod container connected to the Kafka cluster.

The following procedure describes a secure reassignment process that uses mTLS. You'll need a Kafka cluster that uses TLS encryption and mTLS authentication.

You'll need the following to establish a connection:

- The cluster CA certificate and password generated by the Cluster Operator when the Kafka cluster is created
- The user CA certificate and password generated by the User Operator when a user is created for client access to the Kafka cluster

In this procedure, the CA certificates and corresponding passwords are extracted from the cluster

and user secrets that contain them in PKCS #12 (.p12 and .password) format. The passwords allow access to the .p12 stores that contain the certificates. You use the .p12 stores to specify a truststore and keystore to authenticate connection to the Kafka cluster.

#### Prerequisites

- You have a running Cluster Operator.
- You have a running Kafka cluster based on a Kafka resource configured with internal TLS encryption and mTLS authentication.

#### Kafka configuration with TLS encryption and mTLS authentication

```
apiVersion: kafka.strimzi.io/v1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    listeners:
      # ...
      - name: tls
        port: 9093
        type: internal
        tls: true ①
        authentication:
          type: tls ②
    # ...
```

① Enables TLS encryption for the internal listener.

② Listener authentication mechanism specified as mutual `tls`.

- The running Kafka cluster contains a set of topics and partitions to reassign.

#### Example topic configuration for `my-topic`

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaTopic
metadata:
  name: my-topic
  labels:
    strimzi.io/cluster: my-cluster
spec:
  partitions: 10
  replicas: 3
  config:
    retention.ms: 7200000
    segment.bytes: 1073741824
  # ...
```

- You have a `KafkaUser` configured with ACL rules that specify permission to produce and consume topics from the Kafka brokers.

*Example Kafka user configuration with ACL rules to allow operations on `my-topic` and `my-cluster`*

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaUser
metadata:
  name: my-user
  labels:
    strimzi.io/cluster: my-cluster
spec:
  authentication: ①
    type: tls
  authorization:
    type: simple ②
    acls:
      # access to the topic
      - resource:
          type: topic
          name: my-topic
        operations:
          - Create
          - Describe
          - Read
          - AlterConfigs
        host: "*"
      # access to the cluster
      - resource:
          type: cluster
        operations:
          - Alter
          - AlterConfigs
        host: "*"
      # ...
    # ...
```

① User authentication mechanism defined as mutual `tls`.

② Simple authorization and accompanying list of ACL rules.

#### *Procedure*

1. Extract the cluster CA certificate and password from the `<cluster_name>-cluster-ca-cert` secret of the Kafka cluster.

```
kubectl get secret <cluster_name>-cluster-ca-cert -o jsonpath='{.data.ca\.p12}' |
base64 -d > ca.p12
```

```
kubectl get secret <cluster_name>-cluster-ca-cert -o
```

```
jsonpath='{.data.ca\\.password}' | base64 -d > ca.password
```

Replace `<cluster_name>` with the name of the Kafka cluster. When you deploy Kafka using the **Kafka** resource, a secret with the cluster CA certificate is created with the Kafka cluster name (`<cluster_name>-cluster-ca-cert`). For example, `my-cluster-cluster-ca-cert`.

2. Run a new interactive pod container using the Kafka image to connect to a running Kafka broker.

```
kubectl run --restart=Never --image=quay.io/strimzi/kafka:0.50.0-kafka-4.1.1  
<interactive_pod_name> -- /bin/sh -c "sleep 3600"
```

Replace `<interactive_pod_name>` with the name of the pod.

3. Copy the cluster CA certificate to the interactive pod container.

```
kubectl cp ca.p12 <interactive_pod_name>:/tmp
```

4. Extract the user CA certificate and password from the secret of the Kafka user that has permission to access the Kafka brokers.

```
kubectl get secret <kafka_user> -o jsonpath='{.data.user\\.p12}' | base64 -d >  
user.p12
```

```
kubectl get secret <kafka_user> -o jsonpath='{.data.user\\.password}' | base64 -d >  
user.password
```

Replace `<kafka_user>` with the name of the Kafka user. When you create a Kafka user using the **KafkaUser** resource, a secret with the user CA certificate is created with the Kafka user name. For example, `my-user`.

5. Copy the user CA certificate to the interactive pod container.

```
kubectl cp user.p12 <interactive_pod_name>:/tmp
```

The CA certificates allow the interactive pod container to connect to the Kafka broker using TLS.

6. Create a `config.properties` file to specify the truststore and keystore used to authenticate connection to the Kafka cluster.

Use the certificates and passwords you extracted in the previous steps.

```
bootstrap.servers=<kafka_cluster_name>-kafka-bootstrap:9093 ①  
security.protocol=SSL ②
```

```
ssl.truststore.location=/tmp/ca.p12 ③  
ssl.truststore.password=<truststore_password> ④  
ssl.keystore.location=/tmp/user.p12 ⑤  
ssl.keystore.password=<keystore_password> ⑥
```

① The bootstrap server address to connect to the Kafka cluster. Use your own Kafka cluster name to replace <kafka\_cluster\_name>.

② The security protocol option when using TLS for encryption.

③ The truststore location contains the public key certificate (**ca.p12**) for the Kafka cluster.

④ The password (**ca.password**) for accessing the truststore.

⑤ The keystore location contains the public key certificate (**user.p12**) for the Kafka user.

⑥ The password (**user.password**) for accessing the keystore.

7. Copy the **config.properties** file to the interactive pod container.

```
kubectl cp config.properties <interactive_pod_name>:/tmp/config.properties
```

8. Prepare a JSON file named **topics.json** that specifies the topics to move.

Specify topic names as a comma-separated list.

*Example JSON file to reassign all the partitions of my-topic*

```
{  
  "version": 1,  
  "topics": [  
    { "topic": "my-topic"}  
  ]  
}
```

You can also use this file to [change the replication factor of a topic](#).

9. Copy the **topics.json** file to the interactive pod container.

```
kubectl cp topics.json <interactive_pod_name>:/tmp/topics.json
```

10. Start a shell process in the interactive pod container.

```
kubectl exec -n <namespace> -ti <interactive_pod_name> /bin/bash
```

Replace <namespace> with the Kubernetes namespace where the pod is running.

11. Use the **kafka-reassign-partitions.sh** command to generate the reassignment JSON.

*Example command to move the partitions of my-topic to specified brokers*

```
bin/kafka-reassign-partitions.sh --bootstrap-server my-cluster-kafka-bootstrap:9093
 \
--command-config /tmp/config.properties \
--topics-to-move-json-file /tmp/topics.json \
--broker-list 0,1,2,3,4 \
--generate
```

#### *Additional resources*

- [Configuring Kafka](#)
- [Configuring Kafka topics](#)
- [Configuring Kafka users](#)

## 24.3. Using the partition reassignment tool to reassign partitions after adding brokers

Use a reassignment file generated by the [kafka-reassign-partitions.sh](#) tool to reassign partitions after increasing the number of brokers in a Kafka cluster. The reassignment file should describe how partitions are reassigned to brokers in the enlarged Kafka cluster. You apply the reassignment specified in the file to the brokers and then verify the new partition assignments.

This procedure describes a secure scaling process that uses TLS. You'll need a Kafka cluster that uses TLS encryption and mTLS authentication.

The [kafka-reassign-partitions.sh](#) tool can be used to reassign partitions within a Kafka cluster, regardless of whether you are managing all nodes through the cluster or using the node pools to manage groups of nodes within the cluster.

**NOTE**

Though you can use the [kafka-reassign-partitions.sh](#) tool for this operation, Cruise Control is recommended for automated partition reassessments and cluster rebalancing. Cruise Control can move topics from one broker to another without any downtime, and it is the most efficient way to reassign partitions.

#### *Prerequisites*

- You have a running Kafka cluster based on a [Kafka](#) resource configured with internal TLS encryption and mTLS authentication.
- You have [generated a reassignment JSON file named reassignment.json](#).
- You are running an interactive pod container that is connected to the running Kafka broker.
- You are connected as a [KafkaUser](#) configured with ACL rules that specify permission to manage the Kafka cluster and its topics.

#### *Procedure*

1. Add as many new brokers as you need by increasing the [Kafka.spec.kafka.replicas](#) configuration option.

- Verify that the new broker pods have started.
- If you haven't done so, run an interactive pod container to generate a reassignment JSON file named `reassignment.json`.
- Copy the `reassignment.json` file to the interactive pod container.

```
kubectl cp reassignment.json <interactive_pod_name>:/tmp/reassignment.json
```

Replace `<interactive_pod_name>` with the name of the pod.

- Start a shell process in the interactive pod container.

```
kubectl exec -n <namespace> -ti <interactive_pod_name> /bin/bash
```

Replace `<namespace>` with the Kubernetes namespace where the pod is running.

- Run the partition reassignment using the `kafka-reassign-partitions.sh` script from the interactive pod container.

```
bin/kafka-reassign-partitions.sh --bootstrap-server  
<cluster_name>-kafka-bootstrap:9093 \  
--command-config /tmp/config.properties \  
--reassignment-json-file /tmp/reassignment.json \  
--execute
```

Replace `<cluster_name>` with the name of your Kafka cluster. For example, `my-cluster-kafka-bootstrap:9093`

If you are going to throttle replication, you can also pass the `--throttle` option with an inter-broker throttled rate in bytes per second. For example:

```
bin/kafka-reassign-partitions.sh --bootstrap-server  
<cluster_name>-kafka-bootstrap:9093 \  
--command-config /tmp/config.properties \  
--reassignment-json-file /tmp/reassignment.json \  
--throttle 5000000 \  
--execute
```

This command will print out two reassignment JSON objects. The first records the current assignment for the partitions being moved. You should save this to a local file (not a file in the pod) in case you need to revert the reassignment later on. The second JSON object is the target reassignment you have passed in your reassignment JSON file.

If you need to change the throttle during reassignment, you can use the same command with a different throttled rate. For example:

```
bin/kafka-reassign-partitions.sh --bootstrap-server  
<cluster_name>-kafka-bootstrap:9093 \  
--command-config /tmp/config.properties \  
--reassignment-json-file /tmp/reassignment.json \  
--throttle 10000000 \  
--execute
```

5. Verify that the reassignment has completed using the `kafka-reassign-partitions.sh` command line tool from any of the broker pods. This is the same command as the previous step, but with the `--verify` option instead of the `--execute` option.

```
bin/kafka-reassign-partitions.sh --bootstrap-server  
<cluster_name>-kafka-bootstrap:9093 \  
--command-config /tmp/config.properties \  
--reassignment-json-file /tmp/reassignment.json \  
--verify
```

The reassignment has finished when the `--verify` command reports that each of the partitions being moved has completed successfully. This final `--verify` will also have the effect of removing any reassignment throttles.

6. You can now delete the revert file if you saved the JSON for reverting the assignment to their original brokers.

## 24.4. Using the partition reassignment tool to reassign partitions before removing brokers

Use a reassignment file generated by the `kafka-reassign-partitions.sh` tool to reassign partitions before decreasing the number of brokers in a Kafka cluster. The reassignment file must describe how partitions are reassigned to the remaining brokers in the Kafka cluster. You apply the reassignment specified in the file to the brokers and then verify the new partition assignments. Brokers in the highest numbered pods are removed first.

This procedure describes a secure scaling process that uses TLS. You'll need a Kafka cluster that uses TLS encryption and mTLS authentication.

The `kafka-reassign-partitions.sh` tool can be used to reassign partitions within a Kafka cluster, regardless of whether you are managing all nodes through the cluster or using the node pools to manage groups of nodes within the cluster.

**NOTE**

Though you can use the `kafka-reassign-partitions.sh` tool for this operation, Cruise Control is recommended for automated partition reassessments and cluster rebalancing. Cruise Control can move topics from one broker to another without any downtime, and it is the most efficient way to reassign partitions.

### Prerequisites

- You have a running Kafka cluster based on a `Kafka` resource configured with internal TLS encryption and mTLS authentication.
- You have generated a reassignment JSON file named `reassignment.json`.
- You are running an interactive pod container that is connected to the running Kafka broker.
- You are connected as a `KafkaUser` configured with ACL rules that specify permission to manage the Kafka cluster and its topics.

*Procedure*

1. If you haven't done so, run an interactive pod container to generate a reassignment JSON file named `reassignment.json`.
2. Copy the `reassignment.json` file to the interactive pod container.

```
kubectl cp reassignment.json <interactive_pod_name>:/tmp/reassignment.json
```

Replace `<interactive_pod_name>` with the name of the pod.

3. Start a shell process in the interactive pod container.

```
kubectl exec -n <namespace> -ti <interactive_pod_name> /bin/bash
```

Replace `<namespace>` with the Kubernetes namespace where the pod is running.

4. Run the partition reassignment using the `kafka-reassign-partitions.sh` script from the interactive pod container.

```
bin/kafka-reassign-partitions.sh --bootstrap-server
<cluster_name>-kafka-bootstrap:9093 \
--command-config /tmp/config.properties \
--reassignment-json-file /tmp/reassignment.json \
--execute
```

Replace `<cluster_name>` with the name of your Kafka cluster. For example, `my-cluster-kafka-bootstrap:9093`

If you are going to throttle replication, you can also pass the `--throttle` option with an inter-broker throttled rate in bytes per second. For example:

```
bin/kafka-reassign-partitions.sh --bootstrap-server
<cluster_name>-kafka-bootstrap:9093 \
--command-config /tmp/config.properties \
--reassignment-json-file /tmp/reassignment.json \
--throttle 5000000 \
--execute
```

This command will print out two reassignment JSON objects. The first records the current assignment for the partitions being moved. You should save this to a local file (not a file in the pod) in case you need to revert the reassignment later on. The second JSON object is the target reassignment you have passed in your reassignment JSON file.

If you need to change the throttle during reassignment, you can use the same command with a different throttled rate. For example:

```
bin/kafka-reassign-partitions.sh --bootstrap-server  
<cluster_name>-kafka-bootstrap:9093 \  
--command-config /tmp/config.properties \  
--reassignment-json-file /tmp/reassignment.json \  
--throttle 10000000 \  
--execute
```

5. Verify that the reassignment has completed using the `kafka-reassign-partitions.sh` command line tool from any of the broker pods. This is the same command as the previous step, but with the `--verify` option instead of the `--execute` option.

```
bin/kafka-reassign-partitions.sh --bootstrap-server  
<cluster_name>-kafka-bootstrap:9093 \  
--command-config /tmp/config.properties \  
--reassignment-json-file /tmp/reassignment.json \  
--verify
```

The reassignment has finished when the `--verify` command reports that each of the partitions being moved has completed successfully. This final `--verify` will also have the effect of removing any reassignment throttles.

6. You can now delete the revert file if you saved the JSON for reverting the assignment to their original brokers.
7. When all the partition reassessments have finished, the brokers being removed should not have responsibility for any of the partitions in the cluster. You can verify this by checking that the broker's data log directory does not contain any live partition logs. If the log directory on the broker contains a directory that does not match the extended regular expression `\.[a-zA-Z0-9]-delete$`, the broker still has live partitions and should not be stopped.

You can check this by executing the command:

```
kubectl exec my-cluster-kafka-0 -c kafka -it -- \  
/bin/bash -c \  
"ls -l /var/lib/kafka/kafka-log_<n>_ | grep -E '^d' | grep -vE '[a-zA-Z0-9.-]'  
]+\. [a-zA-Z0-9]+-delete$'"
```

where  $n$  is the number of the pods being deleted.

If the above command prints any output then the broker still has live partitions. In this case, either the reassignment has not finished or the reassignment JSON file was incorrect.

- When you have confirmed that the broker has no live partitions, you can edit the `Kafka.spec.kafka.replicas` property of your `Kafka` resource to reduce the number of brokers.

## 24.5. Using the partition reassignment tool to modify topic replication factor

Use a reassignment file generated by the `kafka-reassign-partitions.sh` tool to change the replication factor of topics.

This procedure describes a secure process that uses TLS. You'll need a Kafka cluster that uses TLS encryption and mTLS authentication.

**NOTE**

Though you can use the `kafka-reassign-partitions.sh` tool for this operation, the simplest approach is to enable Cruise Control and make the change through the `KafkaTopic` configuration. For more information, see [Using Cruise Control to modify topic replication factor](#).

*Prerequisites*

- You have a running Kafka cluster based on a `Kafka` resource configured with internal TLS encryption and mTLS authentication.
- You are running an interactive pod container that is connected to the running Kafka broker.
- You have generated a reassignment JSON file named `reassignment.json`.
- You are connected as a `KafkaUser` configured with ACL rules that specify permission to manage the Kafka cluster and its topics.

See [Generating reassignment JSON files](#).

In this procedure, a topic called `my-topic` has 4 replicas and we want to reduce it to 3. A JSON file named `topics.json` specifies the topic, and was used to generate the `reassignment.json` file.

*Example JSON file specifies my-topic*

```
{  
  "version": 1,  
  "topics": [  
    { "topic": "my-topic"}  
  ]  
}
```

*Procedure*

- If you haven't done so, [run an interactive pod container to generate a reassignment JSON file named `reassignment.json`](#).

*Example reassignment JSON file showing the current and proposed replica assignment*

```
Current partition replica assignment
{"version":1,"partitions":[{"topic":"my-topic","partition":0,"replicas":[3,4,2,0],"log_dirs":["any","any","any","any"]}, {"topic":"my-topic","partition":1,"replicas":[0,2,3,1],"log_dirs":["any","any","any","any"]}, {"topic":"my-topic","partition":2,"replicas":[1,3,0,4],"log_dirs":["any","any","any","any"]}]}

Proposed partition reassignment configuration
{"version":1,"partitions":[{"topic":"my-topic","partition":0,"replicas":[0,1,2,3],"log_dirs":["any","any","any","any"]}, {"topic":"my-topic","partition":1,"replicas":[1,2,3,4],"log_dirs":["any","any","any","any"]}, {"topic":"my-topic","partition":2,"replicas":[2,3,4,0],"log_dirs":["any","any","any","any"]}]}
```

Save a copy of this file locally in case you need to revert the changes later on.

2. Edit the `reassignment.json` to remove a replica from each partition.

For example use the [jq command line JSON parser tool](#) to remove the last replica in the list for each partition of the topic:

*Removing the last topic replica for each partition*

```
jq '.partitions[].replicas |= del(.-[-1])' reassignment.json > reassignment.json
```

*Example reassignment file showing the updated replicas*

```
{"version":1,"partitions":[{"topic":"my-topic","partition":0,"replicas":[0,1,2],"log_dirs":["any","any","any","any"]}, {"topic":"my-topic","partition":1,"replicas":[1,2,3],"log_dirs":["any","any","any","any"]}, {"topic":"my-topic","partition":2,"replicas":[2,3,4],"log_dirs":["any","any","any","any"]}]}
```

3. Copy the `reassignment.json` file to the interactive pod container.

```
kubectl cp reassignment.json <interactive_pod_name>:/tmp/reassignment.json
```

Replace `<interactive_pod_name>` with the name of the pod.

4. Start a shell process in the interactive pod container.

```
kubectl exec -n <namespace> -ti <interactive_pod_name> /bin/bash
```

Replace <namespace> with the Kubernetes namespace where the pod is running.

5. Make the topic replica change using the `kafka-reassign-partitions.sh` script from the interactive pod container.

```
bin/kafka-reassign-partitions.sh --bootstrap-server
<cluster_name>-kafka-bootstrap:9093 \
--command-config /tmp/config.properties \
--reassignment-json-file /tmp/reassignment.json \
--execute
```

**NOTE** Removing replicas from a broker does not require any inter-broker data movement, so there is no need to throttle replication. If you are adding replicas, then you may want to change the throttle rate.

6. Verify that the change to the topic replicas has completed using the `kafka-reassign-partitions.sh` command line tool from any of the broker pods. This is the same command as the previous step, but with the `--verify` option instead of the `--execute` option.

```
bin/kafka-reassign-partitions.sh --bootstrap-server
<cluster_name>-kafka-bootstrap:9093 \
--command-config /tmp/config.properties \
--reassignment-json-file /tmp/reassignment.json \
--verify
```

The reassignment has finished when the `--verify` command reports that each of the partitions being moved has completed successfully. This final `--verify` will also have the effect of removing any reassignment throttles.

7. Run the `bin/kafka-topics.sh` command with the `--describe` option to see the results of the change to the topics.

```
bin/kafka-topics.sh --bootstrap-server
<cluster_name>-kafka-bootstrap:9093 \
--command-config /tmp/config.properties \
--describe
```

*Results of reducing the number of replicas for a topic*

```
my-topic Partition: 0 Leader: 0 Replicas: 0,1,2 Isr: 0,1,2
my-topic Partition: 1 Leader: 2 Replicas: 1,2,3 Isr: 1,2,3
my-topic Partition: 2 Leader: 3 Replicas: 2,3,4 Isr: 2,3,4
```

8. Finally, edit the `KafkaTopic` custom resource to change `.spec.replicas` to 3, and then wait the reconciliation.

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaTopic
metadata:
  name: my-topic
  labels:
    strimzi.io/cluster: my-cluster
spec:
  partitions: 3
  replicas: 3
```

# Chapter 25. Introducing metrics

Metrics provide insight into the health, performance, and capacity of your Kafka deployment. By observing these metrics, you can detect potential issues early, optimize resource usage, and improve reliability.

Metrics from Strimzi components can be collected using Prometheus and visualized in Grafana. Prometheus collects metrics from running pods in your cluster, while Grafana provides an intuitive interface for exploring and visualizing these metrics.

The Strimzi operators expose Prometheus metrics by default, including reconciliation counts, durations, JVM data, and resource processing statistics. Other components and features require configuration to expose their metrics.

In addition, you can use kube-state-metrics (KSM) to monitor the state and readiness of Strimzi custom resources, and to raise alerts for warning conditions.

Strimzi provides example Prometheus rules, Grafana dashboards, and kube-state-metrics configuration files that you can adapt to your environment. You can also track message flow across components by [setting up distributed tracing](#).

**NOTE** Strimzi provides example installation files for Prometheus, Grafana, and kube-state-metrics to help get you started. For further information, refer to the resources of those projects.

## *Additional resources*

- [Prometheus](#)
- [Prometheus configuration](#)
- [Kafka Exporter](#)
- [Grafana Labs](#)
- [Apache Kafka Monitoring](#)
- [kube-state-metrics](#)

## 25.1. Example metrics files

You can find example Grafana dashboards and other metrics configuration files in the [example configuration files](#) provided by Strimzi.

### *Example metrics files provided with Strimzi*

```
metrics
└── grafana-dashboards ①
    ├── strimzi-cruise-control.json
    ├── strimzi-kafka-bridge.json
    ├── strimzi-kafka-connect.json
    └── strimzi-kafka-exporter.json
```

```
|   ├── strimzi-kafka-mirror-maker-2.json
|   ├── strimzi-kafka-oauth.json
|   ├── strimzi-kafka.json
|   ├── strimzi-kraft.json
|   ├── strimzi-operators.json
|   └── grafana-install
|       └── grafana.yaml ②
|   └── kube-state-metrics ③
|       ├── configmap.yaml
|       ├── ksm.yaml
|       └── prometheus-rules.yaml
|   └── prometheus-additional-properties
|       └── prometheus-additional.yaml ④
|   └── prometheus-alertmanager-config
|       └── alert-manager-config.yaml ⑤
|   └── prometheus-install
|       ├── alert-manager.yaml ⑥
|       ├── pod-monitors ⑦
|       |   ├── bridge-metrics.yaml
|       |   ├── cluster-operator-metrics.yaml
|       |   ├── entity-operator-metrics.yaml
|       |   └── kafka-resources-metrics.yaml
|       ├── prometheus-rules ⑧
|       |   ├── prometheus-kafka-bridge-rules.yaml
|       |   ├── prometheus-kafka-certificate-rules.yaml
|       |   ├── prometheus-kafka-connect-rules.yaml
|       |   ├── prometheus-kafka-rules.yaml
|       |   ├── prometheus-kafka-exporter-topic-rules.yaml
|       |   ├── prometheus-mirrormaker2-rules.yaml
|       |   ├── prometheus-strimzi-cluster-operator-rules.yaml
|       |   └── prometheus-strimzi-entity-operator-rules.yaml
|       └── prometheus.yaml ⑨
|   └── strimzi-metrics-reporter
|       ├── grafana-dashboards ⑩
|       |   ├── strimzi-kafka.json
|       |   ├── strimzi-kafka-bridge.json
|       |   ├── strimzi-kafka-connect.json
|       |   ├── strimzi-kafka-mirror-maker-2.json
|       |   └── strimzi-kraft.json
|       ├── kafka-bridge-metrics.yaml ⑪
|       ├── kafka-connect-metrics.yaml ⑫
|       ├── kafka-metrics.yaml ⑬
|       └── kafka-mirror-maker-2-metrics.yaml ⑭
|   ├── kafka-bridge-metrics.yaml ⑮
|   ├── kafka-connect-metrics.yaml ⑯
|   ├── kafka-cruise-control-metrics.yaml ⑰
|   ├── kafka-metrics.yaml ⑱
|   ├── kafka-mirror-maker-2-metrics.yaml ⑲
|   └── oauth-metrics.yaml ⑳
```

- ① Grafana dashboards for components using the JMX Exporter.
- ② Installation file for the Grafana image.
- ③ Kube-state-metrics deployment and configuration files for custom resource monitoring.
- ④ Additional configuration to scrape metrics for CPU, memory and disk volume usage, which comes directly from the Kubernetes cAdvisor agent and kubelet on the nodes.
- ⑤ Hook definitions for sending notifications through Alertmanager.
- ⑥ Resources for deploying and configuring Alertmanager.
- ⑦ PodMonitor definitions translated by the Prometheus Operator into jobs for the Prometheus server to be able to scrape metrics data directly from pods.
- ⑧ Alerting rules examples for use with Prometheus Alertmanager (deployed with Prometheus).
- ⑨ Installation resource file for the Prometheus image.
- ⑩ Grafana dashboards for components using the Strimzi Metrics Reporter.
- ⑪ **KafkaBridge** resource for deploying HTTP Bridge with Strimzi Metrics Reporter.
- ⑫ **KafkaConnect** resource for deploying Kafka Connect with Strimzi Metrics Reporter.
- ⑬ **Kafka** resource for deploying Kafka with Strimzi Metrics Reporter.
- ⑭ **KafkaMirrorMaker2** resource for deploying MirrorMaker 2 with Strimzi Metrics Reporter.
- ⑮ Metrics configuration that defines Prometheus JMX Exporter relabeling rules for HTTP Bridge.
- ⑯ Metrics configuration that defines Prometheus JMX Exporter relabeling rules for Kafka Connect.
- ⑰ Metrics configuration that defines Prometheus JMX Exporter relabeling rules for Cruise Control.
- ⑱ Metrics configuration that defines Prometheus JMX Exporter relabeling rules for Kafka.
- ⑲ Metrics configuration that defines Prometheus JMX Exporter relabeling rules for MirrorMaker 2.
- ⑳ Metrics configuration that defines Prometheus JMX Exporter relabeling rules for OAuth 2.0.

**NOTE**

The Prometheus JMX Exporter collects Java Management Extensions (JMX) from Kafka components and converts them into Prometheus metrics. You do not require **jmxOptions** configuration in the custom resource of the component for the Prometheus JMX Exporter to function. **jmxOptions** is only required if you need direct access to JMX from Kafka components.

Strimzi uses the [Strimzi Metrics Reporter](#) or the [Prometheus JMX Exporter](#) to expose metrics through an HTTP endpoint, which can be scraped by the Prometheus server.

Strimzi provides example custom resource configuration YAML files with relabeling rules. When deploying Prometheus metrics configuration, you can deploy the example custom resource or copy the metrics configuration to your own custom resource definition.

### 25.1.1. Example Prometheus JMX Exporter metrics configuration

Grafana dashboards with Prometheus JMX Exporter are dependent on relabeling rules, which are defined for Strimzi components in the custom resource configuration.

A label is a name-value pair. Relabeling is the process of writing a label dynamically. For example, the value of a label may be derived from the name of a Kafka server and client ID.

*Table 33. Example custom resources with metrics configuration*

Component	Custom resource	Example YAML file
Kafka nodes	Kafka	kafka-metrics.yaml
Kafka Connect	KafkaConnect	kafka-connect-metrics.yaml
Kafka MirrorMaker 2	KafkaMirrorMaker2	kafka-mirror-maker-2-metrics.yaml
HTTP Bridge	KafkaBridge	kafka-bridge-metrics.yaml
Cruise Control	Kafka	kafka-cruise-control-metrics.yaml

### 25.1.2. Example Metrics Reporter metrics configuration

*Table 34. Example custom resources with metrics configuration*

Component	Custom resource	Example YAML file
Kafka nodes	Kafka	kafka-metrics.yaml
Kafka Connect	KafkaConnect	kafka-connect-metrics.yaml
Kafka MirrorMaker2	KafkaMirrorMaker2	kafka-mirror-maker-2metrics.yaml
HTTP Bridge	KafkaBridge	kafka-bridge-metrics.yaml

### 25.1.3. Example Prometheus rules for alert notifications

Example Prometheus rules for alert notifications are provided with the [example metrics configuration files](#) provided by Strimzi. The rules are specified in the [example prometheus-rules](#) directory for use in a [Prometheus deployment](#).

The `prometheus-rules` directory contains example rules for the following components:

- Kafka
- Entity Operator
- Kafka Connect
- HTTP Bridge
- MirrorMaker2
- Kafka Exporter

A description of each of the example rules is provided in the file.

Alerting rules provide notifications about specific conditions observed in metrics. Rules are declared on the Prometheus server, but Prometheus Alertmanager is responsible for alert notifications.

Prometheus alerting rules describe conditions using [PromQL](#) expressions that are continuously evaluated.

When an alert expression becomes true, the condition is met and the Prometheus server sends alert data to the Alertmanager. Alertmanager then sends out a notification using the communication method configured for its deployment.

General points about the alerting rule definitions:

- A `for` property is used with the rules to determine the period of time a condition must persist before an alert is triggered.
- The availability of the `KafkaRunningOutOfSpace` metric and alert is dependent on the Kubernetes configuration and storage implementation used. Storage implementations for certain platforms may not be able to supply the information on available space required for the metric to provide an alert.

Alertmanager can be configured to use email, chat messages or other notification methods. Adapt the default configuration of the example rules according to your specific needs.

#### 25.1.4. Example Grafana dashboards

If you deploy Prometheus to provide metrics, you can use the example Grafana dashboards provided with Strimzi to monitor Strimzi components.

Example dashboards are provided in the `examples/metrics/grafana-dashboards` directory as JSON files.

All dashboards provide JVM metrics, as well as metrics specific to the component. For example, the Grafana dashboard for Strimzi operators provides information on the number of reconciliations or custom resources they are processing.

The example dashboards don't show all the metrics supported by Kafka. The dashboards are populated with a representative set of metrics for monitoring.

*Table 35. Example Grafana dashboard files*

Component	Example JSON file
Strimzi operators	<code>strimzi-operators.json</code>
Kafka	<code>strimzi-kafka.json</code>
Kafka KRaft controllers	<code>strimzi-kraft.json</code>
Kafka Connect	<code>strimzi-kafka-connect.json</code>
Kafka MirrorMaker 2	<code>strimzi-kafka-mirror-maker-2.json</code>
HTTP Bridge	<code>strimzi-kafka-bridge.json</code>
Cruise Control	<code>strimzi-cruise-control.json</code>
Kafka Exporter	<code>strimzi-kafka-exporter.json</code>

**NOTE** When metrics are not available to the Kafka Exporter, because there is no traffic in

the cluster yet, the Kafka Exporter Grafana dashboard will show **N/A** for numeric fields and **No data to show** for graphs.

## 25.2. Using Prometheus with Strimzi

Use Prometheus to provide monitoring data for the example Grafana dashboards provided with Strimzi.

To expose metrics in Prometheus format, you add configuration to a custom resource. You must also make sure that the metrics are scraped by your monitoring stack. Prometheus and Prometheus Alertmanager are used in the examples provided by Strimzi, but you can use also other compatible tools.

You can expose metrics using one of the following approaches:

- [Enabling Prometheus JMX Exporter](#)
- [Enabling Strimzi Metrics Reporter](#)

After enabling metrics, you can integrate with Prometheus:

1. [Setting up Prometheus](#)
2. [Deploying Prometheus Alertmanager](#)

Strimzi provides [example Grafana dashboards](#) to display visualizations of metrics. The exposed metrics provide the monitoring data when you [enable the Grafana dashboard](#).

### 25.2.1. Enabling Prometheus JMX Exporter

To enable and expose metrics in Strimzi for Prometheus, configure the appropriate properties in the custom resources for the components you want to monitor.

Use `metricsConfig` to expose JMX-based metrics for these components:

- Kafka
- Kafka Connect
- Kafka MirrorMaker 2
- HTTP Bridge
- Cruise Control

The [Prometheus JMX Exporter](#) exposes metrics on port 9404 through an HTTP endpoint. Prometheus scrapes this endpoint to collect Kafka metrics.

You can create your own Prometheus configuration or start with the [example custom resource files](#) provided with Strimzi:

- `kafka-metrics.yaml`
- `kafka-connect-metrics.yaml`

- `kafka-mirror-maker-2-metrics.yaml`
- `kafka-bridge-metrics.yaml`
- `kafka-cruise-control-metrics.yaml`
- `oauth-metrics.yaml`

These files include relabeling rules and example metrics configuration, and are a good starting point for trying Prometheus with Strimzi.

**NOTE**

OAuth 2.0-related metrics are also available when OAuth 2.0 authentication or authorization is configured by using `type: custom`. For a working example that enables OAuth 2.0 metrics for Kafka, see the `kafka-ephemeral-oauth-single-keycloak-authz-metrics.yaml` example file.

This procedure shows how to deploy the example Prometheus metrics configuration to the `Kafka` resource. The same steps apply when deploying the example files for other resources.

*Procedure*

1. Deploy the example custom resource with the Prometheus configuration.

For example, for each `Kafka` resource you can apply the `kafka-metrics.yaml` file:

```
kubectl apply -f kafka-metrics.yaml
```

Alternatively, copy the example configuration in `kafka-metrics.yaml` to your own `Kafka` resource:

```
kubectl edit kafka <kafka_configuration_file>
```

Add the `metricsConfig` property and corresponding `ConfigMap` as shown in the following example:

```
apiVersion: kafka.strimzi.io/v1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    metricsConfig: ①
      type: jmxPrometheusExporter
      valueFrom:
        configMapKeyRef:
          name: kafka-metrics
          key: kafka-metrics-config.yml
    ---
    kind: ConfigMap ②
    apiVersion: v1
```

```
metadata:  
  name: kafka-metrics  
  labels:  
    app: strimzi  
data:  
  kafka-metrics-config.yml: |  
    # metrics configuration...
```

- ① Add the `metricsConfig` property that references the metrics `ConfigMap`.
- ② Add the complete `ConfigMap` definition with the JMX Exporter configuration.

## 25.2.2. Enabling Strimzi Metrics Reporter

To enable and expose metrics in Strimzi for Prometheus, configure the appropriate properties in the custom resources for the components you want to monitor.

Use `metricsConfig` configuration to expose metrics for these components:

- Kafka
- HTTP Bridge
- Kafka Connect
- Kafka MirrorMaker 2

This enables the [Strimzi Metrics Reporter](#), which exposes metrics on port 9404 through an HTTP endpoint. Prometheus scrapes this endpoint to collect Kafka metrics.

You can create your own Prometheus configuration or use the [example custom resource file](#) provided with Strimzi:

- `kafka-metrics.yaml`
- `kafka-connect-metrics.yaml`
- `kafka-mirror-maker-2-metrics.yaml`
- `kafka-bridge-metrics.yaml`

These files contain the necessary configuration, and are a good starting point for trying Prometheus with Strimzi.

This procedure shows how to deploy example Prometheus metrics configuration to the [Kafka](#) resource.

### Procedure

1. Deploy the example custom resource with the Prometheus configuration.

For example, for each [Kafka](#) resource you can apply the `kafka-metrics.yaml` file.

*Deploying the example configuration*

```
kubectl apply -f kafka-metrics.yaml
```

Alternatively, copy the example configuration in `kafka-metrics.yaml` to your own `Kafka` resource.

*Copying the example configuration*

```
kubectl edit kafka <kafka_configuration_file>
```

Copy the `metricsConfig` property to your `Kafka` resource.

*Example metrics configuration for Kafka*

```
apiVersion: kafka.strimzi.io/v1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    metricsConfig:
      type: strimziMetricsReporter
      values:
        allowList:
        - "kafka_log."
        - "kafka_network."
```

### 25.2.3. Setting up Prometheus

Prometheus provides an open source set of components for systems monitoring and alert notification.

We describe here how you can use the [CoreOS Prometheus Operator](#) to run and manage a Prometheus server that is suitable for use in production environments, but with the correct configuration you can run any Prometheus server.

**NOTE** The Prometheus server configuration uses service discovery to discover the pods in the cluster from which it gets metrics. For this feature to work correctly, the service account used for running the Prometheus service pod must have access to the API server so it can retrieve the pod list. For more information, see link:[Discovering services](#).

#### Prometheus configuration

Strimzi provides [example configuration files for the Prometheus server](#).

A Prometheus YAML file is provided for deployment:

- `prometheus.yaml`

Additional Prometheus-related configuration is also provided in the following files and directories:

- `pod-monitors/*.yaml`
- `prometheus-additional.yaml`
- `prometheus-rules/*.yaml`

For Prometheus to obtain monitoring data:

- [Deploy the Prometheus Operator](#)

Then use the configuration files to:

- [Deploy Prometheus](#)

#### *Alerting rules*

The `prometheus-rules` directory provides [example alerting rules examples for use with Alertmanager](#).

## Prometheus resources

When you apply the Prometheus configuration, the following resources are created in your Kubernetes cluster and managed by the Prometheus Operator:

- A `ClusterRole` granting Prometheus permissions to read Kafka pod health endpoints and container metrics from cAdvisor and the kubelet.
- A `ServiceAccount` for the Prometheus pods to run under.
- A `ClusterRoleBinding` which binds the `ClusterRole` to the `ServiceAccount`.
- A `Deployment` to manage the Prometheus Operator pod.
- A `PodMonitor` to manage the configuration of the Prometheus pod.
- A `Prometheus` to manage the configuration of the Prometheus pod.
- A `PrometheusRule` to manage alerting rules for the Prometheus pod.
- A `Secret` to manage additional Prometheus settings.
- A `Service` to allow applications running in the cluster to connect to Prometheus (for example, Grafana using Prometheus as datasource).

## Deploying the CoreOS Prometheus Operator

To deploy the Prometheus Operator to your Kafka cluster, apply the YAML bundle resources file from the [Prometheus CoreOS repository](#).

#### *Procedure*

1. Download the `bundle.yaml` resources file from the repository:

```
curl -s https://raw.githubusercontent.com/coreos/prometheus-
```

```
operator/master/bundle.yaml > prometheus-operator-deployment.yaml
```

- Use the latest `master` release as shown, or choose a release that is compatible with your version of Kubernetes (see the [Kubernetes compatibility matrix](#)). The `master` release of the Prometheus Operator works with Kubernetes 1.18+.
- Update the namespace by replacing the example `my-namespace` with your own.

On Linux, use:

```
sed -E -i '/[[:space:]]*namespace: [a-zA-Z0-9-]*$/s(namespace:[[:space:]]*[a-zA-Z0-9-]*$/namespace: my-namespace/' prometheus-operator-deployment.yaml
```

On MacOS, use:

```
sed -i '' -e '/[[:space:]]*namespace: [a-zA-Z0-9-]*$/s(namespace:[[:space:]]*[a-zA-Z0-9-]*$/namespace: my-namespace/' prometheus-operator-deployment.yaml
```

**NOTE**

If using OpenShift, specify a release of the [OpenShift fork](#) of the Prometheus Operator repository.

2. (Optional) If it is not required, you can manually remove the `spec.template.spec.securityContext` property from the `prometheus-operator-deployment.yaml` file.
3. Deploy the Prometheus Operator:

```
kubectl create -f prometheus-operator-deployment.yaml
```

## Deploying Prometheus

Use Prometheus to obtain monitoring data in your Kafka cluster.

You can use your own Prometheus deployment or deploy Prometheus using the [example metrics configuration files](#) provided by Strimzi. The example files include a configuration file for a Prometheus deployment and files for Prometheus-related resources:

- `examples/metrics/prometheus-install/prometheus.yaml`
- `examples/metrics/prometheus-install/prometheus-rules/*.yaml`
- `examples/metrics/prometheus-install/pod-monitors/*.yaml`
- `examples/metrics/prometheus-additional-properties/prometheus-additional.yaml`

The deployment process creates a `ClusterRoleBinding` and discovers an Alertmanager instance in the namespace specified for the deployment.

**NOTE**

By default, the Prometheus Operator only supports jobs that include an `endpoints` role for service discovery. Targets are discovered and scraped for each endpoint

port address. For endpoint discovery, the port address may be derived from service (`role: service`) or pod (`role: pod`) discovery.

#### Prerequisites

- Check the [example alerting rules provided](#)

#### Procedure

1. Modify the Prometheus installation file (`prometheus.yaml`) according to the namespace Prometheus is going to be installed into:

On Linux, use:

```
sed -i 's/namespace: .*/namespace: my-namespace/' prometheus.yaml
```

On MacOS, use:

```
sed -i '' 's/namespace: .*/namespace: my-namespace/' prometheus.yaml
```

2. Edit the `PodMonitor` resource in the `pod-monitors` directory to define Prometheus jobs that will scrape the metrics data from pods.

Update the `namespaceSelector.matchNames` property with the namespace where the pods to scrape the metrics from are running.

`PodMonitor` is used to scrape data directly from pods for Apache Kafka, Operators, the HTTP Bridge and Cruise Control.

3. Edit the `prometheus.yaml` installation file to include additional configuration for scraping metrics directly from nodes.

The Grafana dashboards provided show metrics for CPU, memory and disk volume usage, which come directly from the Kubernetes cAdvisor agent and kubelet on the nodes.

The Prometheus Operator does not have a monitoring resource like `PodMonitor` for scraping the nodes, so the `prometheus-additional.yaml` file contains the additional configuration needed.

- a. Create a `Secret` resource from the configuration file (`prometheus-additional.yaml` in the `examples/metrics/prometheus-additional-properties` directory):

```
kubectl apply -f prometheus-additional.yaml
```

- b. Edit the `additionalScrapeConfigs` property in the `prometheus.yaml` file to include the name of the `Secret` in the `prometheus-additional.yaml` file.

4. Deploy the Prometheus resources:

```
kubectl apply -f pod-monitors/.yaml
```

```
kubectl apply -f prometheus-rules/.yaml  
kubectl apply -f prometheus.yaml
```

## 25.2.4. Deploying Alertmanager

Use Alertmanager to route alerts to a notification service. [Prometheus Alertmanager](#) is a component for handling alerts and routing them to a notification service. Alertmanager supports an essential aspect of monitoring, which is to be notified of conditions that indicate potential issues based on alerting rules.

You can use the [example metrics configuration files](#) provided by Strimzi to deploy Alertmanager to send notifications to a Slack channel. A configuration file defines the resources for deploying Alertmanager:

- [examples/metrics/prometheus-install/alert-manager.yaml](#)

An additional configuration file provides the hook definitions for sending notifications from your Kafka cluster:

- [examples/metrics/prometheus-alertmanager-config/alert-manager-config.yaml](#)

The following resources are defined on deployment:

- An [Alertmanager](#) to manage the Alertmanager pod.
- A [Secret](#) to manage the configuration of the Alertmanager.
- A [Service](#) to provide an easy to reference hostname for other services to connect to Alertmanager (such as Prometheus).

### Prerequisites

- [Metrics are enabled and exposed](#)
- [Prometheus is deployed](#)

### Procedure

1. Update the [alert-manager-config.yaml](#) file in the [examples/metrics/prometheus-alertmanager-config](#) directory to replace the following:
  - [slack\\_api\\_url](#) property with the actual value of the Slack API URL related to the application for the Slack workspace
  - [channel](#) property with the actual Slack channel on which to send notifications
2. Create a [Secret](#) resource from the Alertmanager configuration file:

```
kubectl apply -f alert-manager-config.yaml
```

3. Deploy Alertmanager:

```
kubectl apply -f alert-manager.yaml
```

## 25.3. Enabling the example Grafana dashboards

Use Grafana to provide visualizations of Prometheus metrics on customizable dashboards.

You can use your own Grafana deployment or deploy Grafana using the [example metrics configuration files](#) provided by Strimzi. The example files include a configuration file for a Grafana deployment

- [examples/metrics/grafana-install/grafana.yaml](#)

Strimzi also provides [example dashboard configuration files for Grafana](#) in JSON format.

- [examples/metrics/grafana-dashboards](#)

This procedure uses the example Grafana configuration file and example dashboards.

The example dashboards are a good starting point for monitoring key metrics, but they don't show all the metrics supported by Kafka. You can modify the example dashboards or add other metrics, depending on your infrastructure.

**NOTE** No alert notification rules are defined.

When accessing a dashboard, you can use the [port-forward](#) command to forward traffic from the Grafana pod to the host. The name of the Grafana pod is different for each user.

### Prerequisites

- Metrics are enabled and exposed:
  - [Enabling Prometheus JMX Exporter](#)
  - [Enabling Strimzi Metrics Reporter](#)
- [Prometheus and Prometheus Alertmanager are deployed](#)

### Procedure

1. Deploy Grafana.

```
kubectl apply -f grafana.yaml
```

2. Get the details of the Grafana service.

```
kubectl get service grafana
```

For example:

NAME	TYPE	CLUSTER-IP	PORT(S)
grafana	ClusterIP	172.30.123.40	3000/TCP

Note the port number for port forwarding.

3. Use **port-forward** to redirect the Grafana user interface to **localhost:3000**:

```
kubectl port-forward svc/grafana 3000:3000
```

4. In a web browser, access the Grafana login screen using the URL <http://localhost:3000>.

The Grafana Log In page appears.

5. Enter your user name and password, and then click **Log In**.

The default Grafana user name and password are both **admin**. After logging in for the first time, you can change the password.

6. In **Configuration > Data Sources**, add Prometheus as a *data source*.

- Specify a name
- Add *Prometheus* as the type
- Specify a Prometheus server URL

The Prometheus operator service (**prometheus-operated**) is accessible internally within the Kubernetes cluster on port 9090: <http://prometheus-operated:9090>.

Save and test the connection when you have added the details.

7. Click the + icon and then click **Import**.

8. In **examples/metrics/grafana-dashboards**, copy the JSON of the dashboard to import.

9. Paste the JSON into the text box, and then click **Load**.

10. Repeat steps 7-9 for the other example Grafana dashboards.

The imported Grafana dashboards are available to view from the **Dashboards** home page.

## 25.4. Monitoring custom resources with kube-state-metrics

Use kube-state-metrics (KSM) to monitor the state and readiness of custom resources managed by Strimzi. **KSM** is a scalable Kubernetes native service that listens to the Kubernetes API server and exports metrics about the state of the objects. Monitoring through KSM allows you to track the health of Strimzi components, identify resources that are not ready, and configure alerts for warning conditions.

You can use your own KSM deployment or deploy KSM using the [example metrics configuration](#)

files provided by Strimzi. The example files include a configuration file for a KSM deployment:

- [examples/metrics/kube-state-metrics/ksm.yaml](#)

If you are using the example deployment file, you can update the namespace by replacing the example `myproject` with your own:

On Linux, use:

```
sed -E -i '/[[:space:]]*namespace: [a-zA-Z0-9-]*$/s/namespace:[[:space:]]*[a-zA-Z0-9-]*$/namespace: myproject/' examples/metrics/kube-state-metrics/ksm.yaml
```

On MacOS, use:

```
sed -i '' -e '/[[:space:]]*namespace: [a-zA-Z0-9-]*$/s/namespace:[[:space:]]*[a-zA-Z0-9-]*$/namespace: myproject/' examples/metrics/kube-state-metrics/ksm.yaml
```

Strimzi also provides an [example configuration ConfigMap for KSM](#):

- [examples/metrics/kube-state-metrics/configmap.yaml](#)

This procedure uses the example KSM deployment and configuration file.

#### Prerequisites

- [Prometheus and Prometheus Alertmanager are deployed](#)

#### Procedure

##### 1. Deploy KSM:

```
kubectl apply -f configmap.yaml  
kubectl apply -f ksm.yaml
```

##### 2. Verify that Prometheus is scraping KSM metrics.

Metrics are scraped from the `strimzi-kube-state-metrics` service using the `ServiceMonitor` configured for the KSM deployment.

For alerting on these metrics, check the provided `PrometheusRule` resource:

- [examples/metrics/kube-state-metrics/prometheus-rules.yaml](#)

## 25.5. Consumer lag monitoring

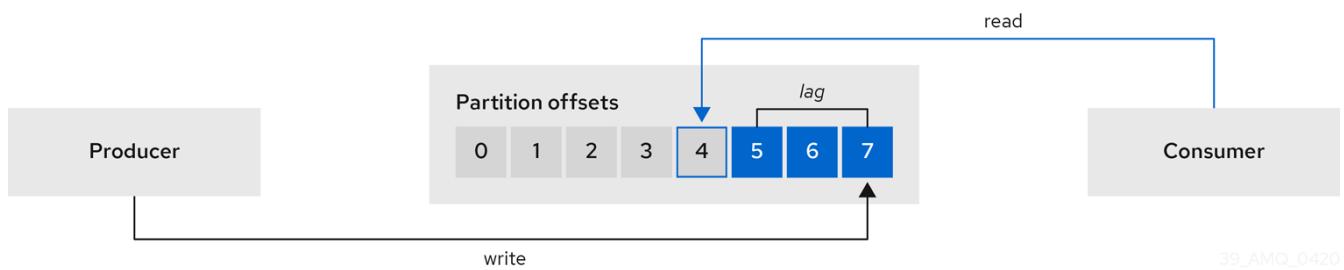
[Kafka Exporter](#) is an open source project that enhances the monitoring of Apache Kafka brokers and clients. Kafka Exporter extracts additional metrics data from Kafka brokers related to consumer groups, consumer lag, topic offsets, and partitions. The metrics are exposed in Prometheus format and can be collected by Prometheus, then visualized in Grafana.

Kafka Exporter relies on data from the `_consumer_offsets` topic to report consumer lag metrics. This topic only contains information if consumer groups are actively committing offsets. Consumer groups must therefore be in use for Kafka Exporter to function correctly.

Consumer lag indicates the difference in the rate of production and consumption of messages. Specifically, consumer lag for a given consumer group indicates the delay between the last message in the partition and the message being currently picked up by that consumer.

The lag reflects the position of the consumer offset in relation to the end of the partition log.

*Consumer lag between the producer and consumer offset*



39\_AMQ\_0420

This difference is sometimes referred to as the *delta* between the producer offset and consumer offset: the read and write positions in the Kafka broker topic partitions.

Suppose a topic streams 100 messages a second. A lag of 1000 messages between the producer offset (the topic partition head) and the last offset the consumer has read means a 10-second delay.

*Why monitor consumer lag?*

For applications that rely on the processing of (near) real-time data, it is critical to monitor consumer lag to check that it does not become too big. The greater the lag becomes, the further the process moves from the real-time processing objective.

Consumer lag, for example, might be a result of consuming too much old data that has not been purged, or through unplanned shutdowns.

*Reducing consumer lag*

Use the Grafana charts to analyze lag and to check if actions to reduce lag are having an impact on an affected consumer group. If, for example, Kafka brokers are adjusted to reduce lag, the dashboard will show the *Lag by consumer group* chart going down and the *Messages consumed per minute* chart going up.

Typical actions to reduce lag include:

- Scaling-up consumer groups by adding new consumers
- Increasing the retention time for a message to remain in a topic
- Adding more disk capacity to increase the message buffer

Actions to reduce consumer lag depend on the underlying infrastructure and the use cases Strimzi is supporting. For instance, a lagging consumer is less likely to benefit from the broker being able to service a fetch request from its disk cache. And in certain cases, it might be acceptable to

automatically drop messages until a consumer has caught up.

### 25.5.1. Deploying Kafka Exporter

To monitor consumer lag in your Kafka cluster, configure Kafka Exporter in the [Kafka](#) custom resource. Kafka Exporter exposes lag data as Prometheus metrics, which can be visualized in Grafana. A Grafana dashboard for Kafka Exporter is included in the [example Grafana dashboards](#) provided by Strimzi.

**IMPORTANT**

Kafka Exporter provides only metrics related to consumer groups and lag. To collect general Kafka metrics, configure metrics on the Kafka brokers. For more information, see [Using Prometheus with Strimzi](#).

#### Prerequisites

- Consumer groups must be in use.

Kafka Exporter relies on data from the `__consumer_offsets` topic to report consumer lag metrics. This topic only contains information if consumer groups are actively committing offsets.

#### Procedure

1. Add `kafkaExporter` configuration to the `spec` section of the `Kafka` resource.

#### Example configuration for deploying Kafka Exporter

```
apiVersion: kafka.strimzi.io/v1
kind: Kafka
metadata:
  name: my-cluster
spec:
  # ...
  kafkaExporter:
    # Collection filters (recommended)
    groupRegex: ".*" ①
    topicRegex: ".*" ②
    groupExcludeRegex: "^excluded-.*" ③
    topicExcludeRegex: "^excluded-.*" ④
    # Resources requests and limits (recommended)
    resources: ⑥
      requests:
        cpu: 200m
        memory: 64Mi
      limits:
        cpu: 500m
        memory: 128Mi
    # Metrics for all consumers (optional)
    showAllOffsets: false ⑤
    # Logging configuration (optional)
    logging: debug ⑦
    # Sarama logging (optional)
    enableSaramaLogging: true ⑧
```

```

# Readiness probe (optional)
readinessProbe: ⑨
  initialDelaySeconds: 15
  timeoutSeconds: 5
# Liveness probe (optional)
livenessProbe: ⑩
  initialDelaySeconds: 15
  timeoutSeconds: 5
# Pod template (optional)
template: ⑪
  pod:
    metadata:
      labels:
        label1: value1
    imagePullSecrets:
      - name: my-docker-credentials
    securityContext:
      runAsUser: 1000001
      fsGroup: 0
    terminationGracePeriodSeconds: 120
# Custom image (optional)
image: my-registry.io/my-org/my-exporter-cluster:latest ⑫
# ...

```

- ① Regular expression to specify consumer groups to include in metrics.
- ② Regular expression to specify topics to include in metrics.
- ③ Regular expression to specify consumer groups to exclude from metrics.
- ④ Regular expression to specify topics to exclude from metrics.
- ⑤ By default, metrics are collected for all consumers regardless of connection status. Setting `showAllOffsets` to `false` stops collecting metrics for disconnected consumers.
- ⑥ CPU and memory resources to reserve.
- ⑦ Logging configuration, to log messages with a given severity (debug, info, warn, error, fatal) or above.
- ⑧ Boolean to enable Sarama logging, which provides detailed logs from the Go client library used by Kafka Exporter. Useful for debugging Kafka client interactions.
- ⑨ Readiness probe to check when Kafka Exporter is ready to serve metrics.
- ⑩ Liveness probe to detect and restart Kafka Exporter if it becomes unresponsive.
- ⑪ Template customization. Here a pod is scheduled with additional security attributes.
- ⑫ **ADVANCED OPTION:** Container image configuration, which is recommended only in special situations.

## 2. Apply the changes to the **Kafka** configuration.

Resources, including a **Service** and **Pod**, are created for the Kafka Exporter with the naming convention `<kafka_cluster_name>-kafka-exporter`.

### 3. Configure Prometheus to scrape metrics from the Kafka Exporter endpoint.

If you are using the example Prometheus deployment, it is already set up to discover and scrape Kafka Exporter metrics. The `PodMonitor` resource named `kafka-resources-metrics` matches the `strimzi.io/kind: Kafka` label used to identify the Kafka Exporter. For more information, see [Deploying Prometheus](#).

### 4. Import the Kafka Exporter dashboard into Grafana to visualize consumer lag.

For more information, see [Enabling the example Grafana dashboards](#).

**TIP**

Use the *Lag by consumer group* and *Messages consumed per second* panels to evaluate lag and the impact of tuning actions.

## 25.6. Cruise Control operations monitoring

Cruise Control monitors Kafka brokers in order to track the utilization of brokers, topics, and partitions. Cruise Control also provides a set of metrics for monitoring its own performance.

The Cruise Control metrics reporter collects raw metrics data from Kafka brokers. The data is produced to topics that are automatically created by Cruise Control. The metrics are used to [generate optimization proposals for Kafka clusters](#).

Cruise Control metrics are available for real-time monitoring of Cruise Control operations. For example, you can use Cruise Control metrics to monitor the status of rebalancing operations that are running or provide alerts on any anomalies that are detected in an operation's performance.

For more information on exposing Cruise Control metrics, see [Using Prometheus with Strimzi](#).

**NOTE**

For a full list of available Cruise Control metrics, which are known as *sensors*, see the [Cruise Control documentation](#)

### 25.6.1. Monitoring balancedness scores

Cruise Control metrics include a balancedness score. Balancedness is the measure of how evenly a workload is distributed in a Kafka cluster.

The Cruise Control metric for balancedness score (`balancedness-score`) might differ from the balancedness score in the `KafkaRebalance` resource. Cruise Control calculates each score using `anomaly.detection.goals` which might not be the same as the `default.goals` used in the `KafkaRebalance` resource. The `anomaly.detection.goals` are specified in the `spec.cruiseControl.config` of the `Kafka` custom resource.

Refreshing the `KafkaRebalance` resource fetches an optimization proposal. The latest cached optimization proposal is fetched if one of the following conditions applies:

- KafkaRebalance `goals` match the goals configured in the `default.goals` section of the `Kafka` resource
- KafkaRebalance `goals` are not specified

Otherwise, Cruise Control generates a new optimization proposal based on KafkaRebalance [goals](#). If new proposals are generated with each refresh, this can impact performance monitoring.

## 25.6.2. Setting up alerts for anomaly detection

Cruise control's *anomaly detector* provides metrics data for conditions that block the generation of optimization goals, such as broker failures. If you want more visibility, you can use the metrics provided by the anomaly detector to set up alerts and send out notifications. You can set up Cruise Control's *anomaly notifier* to route alerts based on these metrics through a specified notification channel. Alternatively, you can set up Prometheus to scrape the metrics data provided by the anomaly detector and generate alerts. Prometheus Alertmanager can then route the alerts generated by Prometheus.

The [Cruise Control documentation](#) provides information on `AnomalyDetector` metrics and the anomaly notifier.

# Chapter 26. Introducing distributed tracing

Distributed tracing allows you to track 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, distributed tracing facilitates end-to-end tracking of messages: from source systems to Kafka, and then from Kafka to target systems and applications. This complements the monitoring of metrics in Grafana dashboards and component loggers.

Strimzi provides built-in support for tracing for 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
- HTTP Bridge to trace messages between Kafka and HTTP client applications

Tracing is not supported for Kafka brokers.

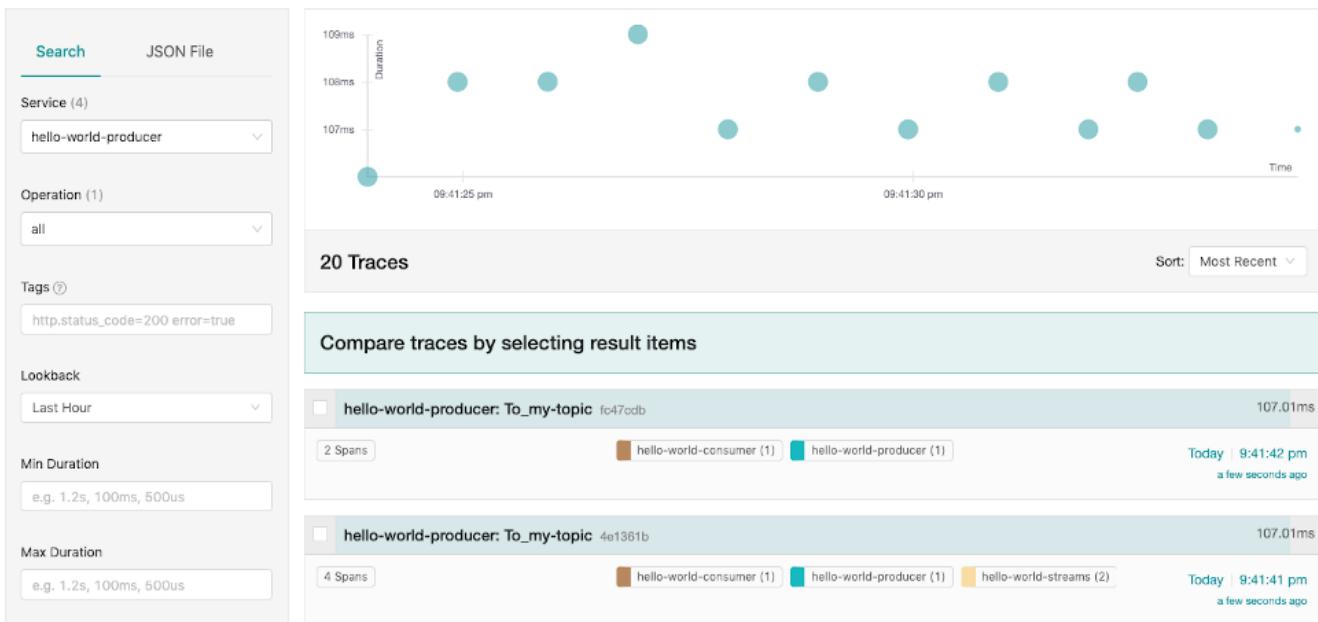
## 26.1. Tracing options

Distributed traces consist of spans, which represent individual units of work performed over a specific time period. When instrumented with tracers, applications generate traces that follow requests as they move through the system, making it easier to identify delays or issues.

OpenTelemetry, a telemetry framework, provides APIs for tracing that are independent of any specific backend tracing system. In Strimzi, the default protocol for transmitting traces between Kafka components and tracing systems is OpenTelemetry's OTLP (OpenTelemetry Protocol), a vendor-neutral protocol.

While OTLP is the default, Strimzi also supports other tracing systems, such as Jaeger. Jaeger is a distributed tracing system designed for monitoring microservices, and its user interface allows you to query, filter, and analyze trace data in detail.

*The Jaeger user interface showing a simple query*



#### Additional resources

- [Jaeger documentation](#)
- [OpenTelemetry documentation](#)

## 26.2. Environment variables for tracing

Use environment variables to enable tracing for Kafka components or to initialize a tracer for Kafka clients.

Tracing environment variables are subject to change. For the latest information, see the [OpenTelemetry documentation](#).

The following table describes the key environment variables for setting up tracing with OpenTelemetry.

*Table 36. OpenTelemetry environment variables*

Property	Required	Description
OTEL_SERVICE_NAME	Yes	The name of the tracing service for OpenTelemetry, such as OTLP or Jaeger.
OTEL_EXPORTER_OTLP_ENDPOINT	Yes (if using OTLP exporter)	The OTLP endpoint for exporting trace data to the tracing system. For Jaeger tracing, specify the <code>OTEL_EXPORTER_JAEGER_ENDPOINT</code> . For other tracing systems, <a href="#">specify the appropriate endpoint</a> .

Property	Required	Description
OTEL_TRACES_EXPORTER	No (unless using a non-OTLP exporter)	The exporter used for tracing. The default is <a href="#">otlp</a> , which does not need to be specified. For Jaeger tracing, set this variable to <a href="#">jaeger</a> . For other tracing systems, <a href="#">specify the appropriate exporter</a> .
OTEL_EXPORTER_OTLP_CERTIFICATE	No (required if using TLS with OTLP)	The path to the file containing trusted certificates for TLS authentication. Required to secure communication between Kafka components and the OpenTelemetry endpoint when using TLS with the <a href="#">otlp</a> exporter.

## 26.3. Setting up distributed tracing

Enable distributed tracing in Kafka components by specifying a tracing type in the custom resource. Instrument tracers in Kafka clients for end-to-end tracking of messages.

To set up distributed tracing, follow these procedures in order:

- [Enable tracing for supported Kafka components](#)
- [Initialize a tracer for Kafka clients](#)
- Instrument clients with tracers, embedding telemetry-gathering functionality into the code:
  - [Instrument producers and consumers for tracing](#)
  - [Instrument Kafka Streams applications for tracing](#)

### 26.3.1. Prerequisites

Before setting up distributed tracing, make sure backend components are deployed to your Kubernetes cluster.

We recommend using the Jaeger operator for deploying Jaeger on your Kubernetes cluster. For deployment instructions, see the [Jaeger documentation](#).

**NOTE** Setting up tracing systems is outside the scope of this content.

### 26.3.2. Enabling tracing in supported Kafka components

Distributed tracing is supported for MirrorMaker, MirrorMaker 2, Kafka Connect, and the HTTP Bridge. Enable tracing using OpenTelemetry by setting the `spec.tracing.type` property to `opentelemetry`. Configure the custom resource of the component to specify and enable a tracing system using `spec.template` properties.

By default, OpenTelemetry uses the OTLP (OpenTelemetry Protocol) exporter and endpoint to gather trace data. This procedure shows the configuration to use OTLP as the tracing system.

If you prefer to use a different tracing system supported by OpenTelemetry, such as Jaeger, you can modify the exporter and endpoint settings in the tracing configuration.

**CAUTION**

Strimzi no longer supports OpenTracing. If you were previously using OpenTracing with the `type: jaeger` option, we encourage you to transition to using OpenTelemetry instead.

Enabling tracing in a resource triggers the following events:

- Interceptor classes are updated in the integrated consumers and producers of the component.
- For MirrorMaker, MirrorMaker 2, and Kafka Connect, the tracing agent initializes a tracer based on the tracing configuration defined in the resource.
- For the HTTP Bridge, a tracer based on the tracing configuration defined in the resource is initialized by the HTTP Bridge itself.

#### *Tracing in MirrorMaker 2*

For MirrorMaker 2, messages are traced from the source cluster to the target cluster. The trace data records messages entering and leaving the MirrorMaker 2 component.

#### *Tracing in Kafka Connect*

For Kafka Connect, only messages produced and consumed by Kafka Connect are traced. To trace messages sent between Kafka Connect and external systems, you must configure tracing in the connectors for those systems.

#### *Tracing in the HTTP Bridge*

For the HTTP Bridge, messages produced and consumed by the HTTP Bridge are traced. Incoming HTTP requests from client applications to send and receive messages through the HTTP Bridge are also traced. To have end-to-end tracing, you must configure tracing in your HTTP clients.

#### *Procedure*

Perform these steps for each `KafkaMirrorMaker2`, `KafkaConnect`, and `KafkaBridge` resource.

1. In the `spec.template` property, configure the tracer service.

- Use the `tracing environment variables` as template configuration properties.
- For OpenTelemetry, set the `spec.tracing.type` property to `opentelemetry`.

#### *Example tracing configuration for Kafka Connect using OpenTelemetry*

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaConnect
metadata:
  name: my-connect-cluster
spec:
  #...
  template:
    connectContainer:
      env:
```

```

    - name: OTEL_SERVICE_NAME
      value: my-otel-service
    - name: OTEL_EXPORTER_OTLP_ENDPOINT
      value: "http://otlp-host:4317"
  tracing:
    type: opentelemetry
  #...

```

*Example tracing configuration for MirrorMaker 2 using OpenTelemetry*

```

apiVersion: kafka.strimzi.io/v1
kind: KafkaMirrorMaker2
metadata:
  name: my-mm2-cluster
spec:
  #...
  template:
    connectContainer:
      env:
        - name: OTEL_SERVICE_NAME
          value: my-otel-service
        - name: OTEL_EXPORTER_OTLP_ENDPOINT
          value: "http://otlp-host:4317"
    tracing:
      type: opentelemetry
  #...

```

*Example tracing configuration for the HTTP Bridge using OpenTelemetry*

```

apiVersion: kafka.strimzi.io/v1
kind: KafkaBridge
metadata:
  name: my-bridge
spec:
  #...
  template:
    bridgeContainer:
      env:
        - name: OTEL_SERVICE_NAME
          value: my-otel-service
        - name: OTEL_EXPORTER_OTLP_ENDPOINT
          value: "http://otlp-host:4317"
    tracing:
      type: opentelemetry
  #...

```

2. (Optional) If TLS authentication is configured on the OTLP endpoint, use the **OTEL\_EXPORTER\_OTLP\_CERTIFICATE** environment variable to specify the path to a trusted certificate. This secures communication between Kafka components and the OpenTelemetry endpoint.

To provide the certificate, mount a volume containing the secret that holds the trusted certificate. Unless the endpoint address is redirected from [http](#), use [https](#).

*Example configuration for TLS*

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaConnect
metadata:
  name: my-connect-cluster
spec:
  #...
  template:
    connectContainer:
      env:
        - name: OTEL_SERVICE_NAME
          value: my-otel-service
        - name: OTEL_EXPORTER_OTLP_ENDPOINT
          value: "https://otlp-host:4317"
        - name: OTEL_EXPORTER_OTLP_CERTIFICATE
          value: "/mnt/mysecret/my-certificate.crt"
      volumeMounts:
        - name: tracing-secret-volume
          mountPath: /mnt/mysecret
    pod:
      volumes:
        - name: tracing-secret-volume
          secret:
            secretName: mysecret
    tracing:
      type: opentelemetry
    #...
```

3. Apply the changes to the custom resource configuration.

### 26.3.3. Initializing tracing for Kafka clients

Initialize a tracer for OpenTelemetry, then instrument your client applications for distributed tracing. You can instrument Kafka producer and consumer clients, and Kafka Streams API applications.

Configure and initialize a tracer using a set of [tracing environment variables](#).

#### Procedure

In each client application add the dependencies for the tracer:

1. Add the Maven dependencies to the [pom.xml](#) file for the client application:

*Dependencies for OpenTelemetry*

```
<dependency>
```

```

<groupId>io.opentelemetry.semconv</groupId>
<artifactId>opentelemetry-semconv</artifactId>
<version>1.21.0-alpha</version>
</dependency>
<dependency>
  <groupId>io.opentelemetry</groupId>
  <artifactId>opentelemetry-exporter-otlp</artifactId>
  <version>1.34.1</version>
  <exclusions>
    <exclusion>
      <groupId>io.opentelemetry</groupId>
      <artifactId>opentelemetry-exporter-sender-okhttp</artifactId>
    </exclusion>
  </exclusions>
</dependency>
<dependency>
  <groupId>io.opentelemetry</groupId>
  <artifactId>opentelemetry-exporter-sender-grpc-managed-channel</artifactId>
  <version>1.34.1</version>
  <scope>runtime</scope>
</dependency>
<dependency>
  <groupId>io.opentelemetry</groupId>
  <artifactId>opentelemetry-sdk-extension-autoconfigure</artifactId>
  <version>1.34.1</version>
</dependency>
<dependency>
  <groupId>io.opentelemetry.instrumentation</groupId>
  <artifactId>opentelemetry-kafka-clients-2.6</artifactId>
  <version>1.32.0-alpha</version>
</dependency>
<dependency>
  <groupId>io.opentelemetry</groupId>
  <artifactId>opentelemetry-sdk</artifactId>
  <version>1.34.1</version>
</dependency>
<dependency>
  <groupId>io.opentelemetry</groupId>
  <artifactId>opentelemetry-exporter-sender-jdk</artifactId>
  <version>1.34.1-alpha</version>
  <scope>runtime</scope>
</dependency>
<dependency>
  <groupId>io.grpc</groupId>
  <artifactId>grpc-netty-shaded</artifactId>
  <version>1.75.0</version>
</dependency>

```

2. Define the configuration of the tracer using the [tracing environment variables](#).
3. Create a tracer, which is initialized with the environment variables:

## *Creating a tracer for OpenTelemetry*

```
OpenTelemetry ot = GlobalOpenTelemetry.get();
```

4. Register the tracer as a global tracer:

```
GlobalTracer.register(tracer);
```

5. Instrument your client:

- [Instrumenting producers and consumers for tracing](#)
- [Instrumenting Kafka Streams applications for tracing](#)

### **26.3.4. Instrumenting producers and consumers for tracing**

Instrument application code to enable tracing in Kafka producers and consumers. Use a decorator pattern or interceptors to instrument your Java producer and consumer application code for tracing. You can then record traces when messages are produced or retrieved from a topic.

OpenTelemetry instrumentation project provides classes that support instrumentation of producers and consumers.

#### **Decorator instrumentation**

For decorator instrumentation, create a modified producer or consumer instance for tracing.

#### **Interceptor instrumentation**

For interceptor instrumentation, add the tracing capability to the consumer or producer configuration.

#### *Prerequisites*

- You have [initialized tracing for the client](#).

You enable instrumentation in producer and consumer applications by adding the tracing JARs as dependencies to your project.

#### *Procedure*

Perform these steps in the application code of each producer and consumer application. Instrument your client application code using either a decorator pattern or interceptors.

- To use a decorator pattern, create a modified producer or consumer instance to send or receive messages.

You pass the original [KafkaProducer](#) or [KafkaConsumer](#) class.

#### *Example decorator instrumentation for OpenTelemetry*

```
// Producer instance  
Producer<String, String> op = new KafkaProducer<>(
```

```

    configs,
    new StringSerializer(),
    new StringSerializer()
);
Producer < String, String > producer = tracing.wrap(op);
KafkaTracing tracing = KafkaTracing.create(GlobalOpenTelemetry.get());
producer.send(...);

//consumer instance
Consumer<String, String> oc = new KafkaConsumer<>(
    configs,
    new StringDeserializer(),
    new StringDeserializer()
);
Consumer<String, String> consumer = tracing.wrap(oc);
consumer.subscribe(Collections.singleton("mytopic"));
ConsumerRecords<Integer, String> records = consumer.poll(1000);
ConsumerRecord<Integer, String> record = ...
SpanContext spanContext = TracingKafkaUtils.extractSpanContext(record.headers(),
tracer);

```

- To use interceptors, set the interceptor class in the producer or consumer configuration.

You use the `KafkaProducer` and `KafkaConsumer` classes in the usual way. The `TracingProducerInterceptor` and `TracingConsumerInterceptor` interceptor classes take care of the tracing capability.

#### *Example producer configuration using interceptors*

```

senderProps.put(ProducerConfig.INTERCEPTOR_CLASSES_CONFIG,
    TracingProducerInterceptor.class.getName());

KafkaProducer<Integer, String> producer = new KafkaProducer<>(senderProps);
producer.send(...);

```

#### *Example consumer configuration using interceptors*

```

consumerProps.put(ConsumerConfig.INTERCEPTOR_CLASSES_CONFIG,
    TracingConsumerInterceptor.class.getName());

KafkaConsumer<Integer, String> consumer = new KafkaConsumer<>(consumerProps);
consumer.subscribe(Collections.singletonList("messages"));
ConsumerRecords<Integer, String> records = consumer.poll(1000);
ConsumerRecord<Integer, String> record = ...
SpanContext spanContext = TracingKafkaUtils.extractSpanContext(record.headers(),
tracer);

```

## 26.3.5. Instrumenting Kafka Streams applications for tracing

Instrument application code to enable tracing in Kafka Streams API applications. Use a decorator pattern or interceptors to instrument your Kafka Streams API applications for tracing. You can then record traces when messages are produced or retrieved from a topic.

### Decorator instrumentation

For decorator instrumentation, create a modified Kafka Streams instance for tracing. For OpenTelemetry, you need to create a custom `TracingKafkaClientSupplier` class to provide tracing instrumentation for Kafka Streams.

### Interceptor instrumentation

For interceptor instrumentation, add the tracing capability to the Kafka Streams producer and consumer configuration.

#### Prerequisites

- You have [initialized tracing for the client](#).

You enable instrumentation in Kafka Streams applications by adding the tracing JARs as dependencies to your project.

- To instrument Kafka Streams with OpenTelemetry, you'll need to write a custom `TracingKafkaClientSupplier`.
- The custom `TracingKafkaClientSupplier` can extend Kafka's `DefaultKafkaClientSupplier`, overriding the producer and consumer creation methods to wrap the instances with the telemetry-related code.

#### Example custom `TracingKafkaClientSupplier`

```
private class TracingKafkaClientSupplier extends DefaultKafkaClientSupplier {  
    @Override  
    public Producer<byte[], byte[]> getProducer(Map<String, Object> config) {  
        KafkaTelemetry telemetry =  
            KafkaTelemetry.create(GlobalOpenTelemetry.get());  
        return telemetry.wrap(super.getProducer(config));  
    }  
  
    @Override  
    public Consumer<byte[], byte[]> getConsumer(Map<String, Object> config) {  
        KafkaTelemetry telemetry =  
            KafkaTelemetry.create(GlobalOpenTelemetry.get());  
        return telemetry.wrap(super.getConsumer(config));  
    }  
  
    @Override  
    public Consumer<byte[], byte[]> getRestoreConsumer(Map<String, Object> config)  
    {  
        return this.getConsumer(config);  
    }  
}
```

```
@Override  
public Consumer<byte[], byte[]> getGlobalConsumer(Map<String, Object> config) {  
    return this.getConsumer(config);  
}  
}
```

#### *Procedure*

Perform these steps for each Kafka Streams API application.

- To use a decorator pattern, create an instance of the [TracingKafkaClientSupplier](#) supplier interface, then provide the supplier interface to [KafkaStreams](#).

#### *Example decorator instrumentation*

```
KafkaClientSupplier supplier = new TracingKafkaClientSupplier(tracer);  
KafkaStreams streams = new KafkaStreams(builder.build(), new StreamsConfig(config),  
supplier);  
streams.start();
```

- To use interceptors, set the interceptor class in the Kafka Streams producer and consumer configuration.

The [TracingProducerInterceptor](#) and [TracingConsumerInterceptor](#) interceptor classes take care of the tracing capability.

#### *Example producer and consumer configuration using interceptors*

```
props.put(StreamsConfig.PRODUCER_PREFIX +  
ProducerConfig.INTERCEPTOR_CLASSES_CONFIG,  
TracingProducerInterceptor.class.getName());  
props.put(StreamsConfig.CONSUMER_PREFIX +  
ConsumerConfig.INTERCEPTOR_CLASSES_CONFIG,  
TracingConsumerInterceptor.class.getName());
```

### 26.3.6. Introducing a different OpenTelemetry tracing system

Instead of the default OTLP system, you can specify other tracing systems that are supported by OpenTelemetry. You do this by adding the required artifacts to the Kafka image provided with Strimzi. Any required implementation specific environment variables must also be set. You then enable the new tracing implementation using the [OTEL\\_TRACES\\_EXPORTER](#) environment variable.

This procedure shows how to implement Zipkin tracing.

#### *Procedure*

1. Add the tracing artifacts to the [/opt/kafka/libs/](#) directory of the Kafka image.

You can use the Kafka container image on the [Container Registry](#) as a base image for creating a new custom image.

```
io.opentelemetry:opentelemetry-exporter-zipkin
```

- Set the tracing exporter and endpoint for the new tracing implementation.

*Example Zikpin tracer configuration*

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaMirrorMaker2
metadata:
  name: my-mm2-cluster
spec:
  #...
  template:
    connectContainer:
      env:
        - name: OTEL_SERVICE_NAME
          value: my-zipkin-service
        - name: OTEL_EXPORTER_ZIPKIN_ENDPOINT
          value: http://zipkin-exporter-host-name:9411/api/v2/spans ①
        - name: OTEL_TRACES_EXPORTER
          value: zipkin ②
      tracing:
        type: opentelemetry
    #...
```

① Specifies the Zipkin endpoint to connect to.

② The Zipkin exporter.

### 26.3.7. Specifying custom span names for OpenTelemetry

A tracing *span* is a logical unit of work in Jaeger, with an operation name, start time, and duration. Spans have built-in names, but you can specify custom span names in your Kafka client instrumentation where used.

Specifying custom span names is optional and only applies when using a decorator pattern [in producer and consumer client instrumentation](#) or [Kafka Streams instrumentation](#).

Custom span names cannot be specified directly with OpenTelemetry. Instead, you retrieve span names by adding code to your client application to extract additional tags and attributes.

*Example code to extract attributes*

```
//Defines attribute extraction for a producer
private static class ProducerAttribExtractor implements AttributesExtractor <
ProducerRecord < ?, ? > , Void > {
  @Override
  public void onStart(AttributesBuilder attributes, ProducerRecord < ?, ? >
producerRecord) {
```

```

        set(attributes, AttributeKey.stringKey("prod_start"), "prod1");
    }
    @Override
    public void onEnd(AttributesBuilder attributes, ProducerRecord < ? , ? >
producerRecord, @Nullable Void unused, @Nullable Throwable error) {
        set(attributes, AttributeKey.stringKey("prod_end"), "prod2");
    }
}
//Defines attribute extraction for a consumer
private static class ConsumerAttribExtractor implements AttributesExtractor <
ConsumerRecord < ? , ? > , Void > {
    @Override
    public void onStart(AttributesBuilder attributes, ConsumerRecord < ? , ? >
producerRecord) {
        set(attributes, AttributeKey.stringKey("con_start"), "con1");
    }
    @Override
    public void onEnd(AttributesBuilder attributes, ConsumerRecord < ? , ? >
producerRecord, @Nullable Void unused, @Nullable Throwable error) {
        set(attributes, AttributeKey.stringKey("con_end"), "con2");
    }
}
//Extracts the attributes
public static void main(String[] args) throws Exception {
    Map < String, Object > configs = new HashMap < >
(Collections.singletonMap(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092"));
    System.setProperty("otel.traces.exporter", "jaeger");
    System.setProperty("otel.service.name", "myapp1");
    KafkaTracing tracing = KafkaTracing.newBuilder(GlobalOpenTelemetry.get())
        .addProducerAttributesExtractors(new ProducerAttribExtractor())
        .addConsumerAttributesExtractors(new ConsumerAttribExtractor())
        .build();
}

```

# Chapter 27. Evicting pods with the Strimzi Drain Cleaner

Kafka pods might be evicted during Kubernetes upgrades, maintenance, or pod rescheduling. If your Kafka pods were deployed by Strimzi, you can use the Strimzi Drain Cleaner tool to handle the pod evictions. The Strimzi Drain Cleaner handles the eviction instead of Kubernetes.

By deploying the Strimzi Drain Cleaner, you can use the Cluster Operator to move Kafka pods instead of Kubernetes. The Cluster Operator ensures that the number of in sync replicas for topics are at or above the configured `min.insync.replicas` and Kafka can remain operational during the eviction process. The Cluster Operator waits for topics to synchronize, as the Kubernetes worker nodes drain consecutively.

An admission webhook notifies the Strimzi Drain Cleaner of pod eviction requests to the Kubernetes API. The Strimzi Drain Cleaner then adds a rolling update annotation to the pods to be drained. This informs the Cluster Operator to perform a rolling update of an evicted pod.

**NOTE**

If you are not using the Strimzi Drain Cleaner, you can [add pod annotations to perform rolling updates manually](#).

## 27.1. Default webhook configuration

The Strimzi Drain Cleaner deployment includes a `ValidatingWebhookConfiguration` resource that registers the webhook with the Kubernetes API:

```
apiVersion: admissionregistration.k8s.io/v1
kind: ValidatingWebhookConfiguration
# ...
webhooks:
- name: strimzi-drain-cleaner.strimzi.io
  rules:
    - apiGroups: []
      apiVersions: ["v1"]
      operations: ["CREATE"]
      resources: ["pods/eviction"]
      scope: "Namespaced"
  clientConfig:
    service:
      namespace: "strimzi-drain-cleaner"
      name: "strimzi-drain-cleaner"
      path: /drainer
      port: 443
      caBundle: Cg==
# ...
```

Unless you are using your own TLS certificates, no manual configuration is required.

The webhook intercepts pod eviction requests based on the `rules` defined in the configuration. Only `CREATE` operations targeting the `pods/eviction` sub-resource are evaluated. When these conditions are met, the API forwards the request to the webhook.

The `clientConfig` section specifies the target service and endpoint for the webhook. The webhook listens on the `/drainer` path and requires a secure TLS connection.

The `caBundle` property provides the Base64-encoded certificate chain used to validate HTTPS communication. By default, the TLS certificates are generated and injected into the configuration automatically. If you supply your own TLS certificates, you must manually update the `caBundle` value.

## 27.2. Deploying the Strimzi Drain Cleaner using installation files

Deploy the Strimzi Drain Cleaner to the Kubernetes cluster where the Cluster Operator and Kafka cluster are running.

Strimzi Drain Cleaner can run in two different modes. By default, the Drain Cleaner denies (blocks) the Kubernetes eviction request to prevent Kubernetes from evicting the pods and instead uses the Cluster Operator to move the pod. This mode has better compatibility with various cluster autoscaling tools and does not require any specific `PodDisruptionBudget` configuration. Alternatively, you can enable the legacy mode where it allows the eviction request while also instructing the Cluster Operator to move the pod. For the legacy mode to work, you have to configure the `PodDisruptionBudget` to not allow any pod evictions by setting the `maxUnavailable` option to `0`.

### Prerequisites

- The Drain Cleaner deployment files, which are included in the Strimzi [deployment files](#).
- You have a highly available Kafka cluster deployment running with Kubernetes worker nodes that you would like to update.
- Topics are replicated for high availability.

Topic configuration specifies a replication factor of at least 3 and a minimum number of in-sync replicas to 1 less than the replication factor.

### *Kafka topic replicated for high availability*

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaTopic
metadata:
  name: my-topic
  labels:
    strimzi.io/cluster: my-cluster
spec:
  partitions: 1
  replicas: 3
  config:
    # ...
```

```
min.insync.replicas: 2  
# ...
```

### Using Drain Cleaner in legacy mode

To use the Drain Cleaner in legacy mode, change the default environment variables in the Drain Cleaner [Deployment](#) configuration file:

- Set `STRIMZI_DENY_EVICTION` to `false` to use the legacy mode relying on the [PodDisruptionBudget](#) configuration.

### Example configuration to use legacy mode

```
apiVersion: apps/v1  
kind: Deployment  
spec:  
  # ...  
  template:  
    spec:  
      serviceAccountName: strimzi-drain-cleaner  
      containers:  
        - name: strimzi-drain-cleaner  
          # ...  
          env:  
            - name: STRIMZI_DENY_EVICTION  
              value: "false"  
            - name: STRIMZI_DRAIN_KAFKA  
              value: "true"  
          # ...
```

### Procedure

1. If you are using the legacy mode activated by setting the `STRIMZI_DENY_EVICTION` environment variable to `false`, you must also configure the [PodDisruptionBudget](#) resource. Set `maxUnavailable` to `0` (zero) in the Kafka section of the [Kafka](#) resource using `template` settings.

### Specifying a pod disruption budget

```
apiVersion: kafka.strimzi.io/v1  
kind: Kafka  
metadata:  
  name: my-cluster  
  namespace: myproject  
spec:  
  kafka:  
    template:  
      podDisruptionBudget:  
        maxUnavailable: 0  
    # ...
```

This setting prevents the automatic eviction of pods in case of planned disruptions, leaving the

Strimzi Drain Cleaner and Cluster Operator to roll the pods on different worker nodes.

## 2. Update the `Kafka` resource:

```
kubectl apply -f <kafka_configuration_file>
```

## 3. Deploy the Strimzi Drain Cleaner.

- If you are using `cert-manager` with Kubernetes, apply the resources in the `/install/drain-cleaner/certmanager` directory.

```
kubectl apply -f ./install/drain-cleaner/certmanager
```

The TLS certificates for the webhook are generated automatically and injected into the webhook configuration.

- If you are not using `cert-manager` with Kubernetes, do the following:

- a. [Add TLS certificates to use in the deployment.](#)

Any certificates you add must be renewed before they expire.

- b. Apply the resources in the `/install/drain-cleaner/kubernetes` directory.

```
kubectl apply -f ./install/drain-cleaner/kubernetes
```

- To run the Drain Cleaner on OpenShift, apply the resources in the `/install/drain-cleaner/openshift` directory.

```
kubectl apply -f ./install/drain-cleaner/openshift
```

## 27.3. Deploying the Strimzi Drain Cleaner using Helm

[Helm](#) charts are used to package, configure, and deploy Kubernetes resources. Strimzi provides a Helm chart to deploy the Strimzi Drain Cleaner.

The Drain Cleaner is deployed on the Kubernetes cluster with the default chart configuration, which assumes that `cert-manager` issues the TLS certificates required by the Drain Cleaner.

You can install the Drain Cleaner with `cert-manager` support or provide your own TLS certificates.

### *Prerequisites*

- The Helm client must be installed on a local machine.

### *Default configuration values*

Default configuration values are passed into the chart using parameters defined in a `values.yaml`

file. If you don't want to use the default configuration, you can override the defaults when you install the chart using the `--set` argument. You specify values in the format `--set key=value[,key=value]`. The `values.yaml` file supplied with the Helm deployment files describes the available configuration parameters, including those shown in the following table.

You can override the default image settings. You can also set `secret.create` as `true` and add your own TLS certificates instead of using `cert-manager` to generate the certificates. For information on using OpenSSL to generate certificates, see [Adding or renewing the TLS certificates used by the Strimzi Drain Cleaner](#).

Any certificates you add must be renewed before they expire. You can use the configuration to control how certificates are watched for updates using environment variables. For more information on how the environment variables work, see [Watching the TLS certificates used by the Strimzi Drain Cleaner](#).

*Table 37. Chart configuration options*

Parameter	Description	Default
<code>replicaCount</code>	Number of replicas of the Drain Cleaner webhook	<code>1</code>
<code>image.registry</code>	Drain Cleaner image registry	<code>quay.io</code>
<code>image.repository</code>	Drain Cleaner image repository	<code>strimzi</code>
<code>image.name</code>	Drain Cleaner image name	<code>drain-cleaner</code>
<code>image.tag</code>	Drain Cleaner image tag	<code>latest</code>
<code>image.imagePullPolicy</code>	Image pull policy for all pods deployed by the Drain Cleaner	<code>nil</code>
<code>secret.create</code>	Set to <code>true</code> and add certificates when not using <code>cert-manager</code>	<code>false</code>
<code>namespace.name</code>	Default namespace for the Drain Cleaner deployment.	<code>strimzi-drain-cleaner</code>
<code>resources</code>	Configures resources for the Drain Cleaner pod	<code>[]</code>
<code>nodeSelector</code>	Add a node selector to the Drain Cleaner pod	<code>{}</code>
<code>tolerations</code>	Add tolerations to the Drain Cleaner pod	<code>[]</code>
<code>topologySpreadConstraints</code>	Add topology spread constraints to the Drain Cleaner pod	<code>{}</code>
<code>affinity</code>	Add affinities to the Drain Cleaner pod	<code>{}</code>

#### *Procedure*

1. Deploy the Drain Cleaner:

```
helm install strimzi-drain-cleaner oci://quay.io/strimzi-helm/strimzi-drain-cleaner
```

Alternatively, you can use parameter values to install a specific version of the Drain Cleaner or specify any changes to the default configuration.

*Example configuration that installs a specific version of the Drain Cleaner and changes the number of replicas*

```
helm install strimzi-drain-cleaner --set replicaCount=2 --version 1.5.0  
oci://quay.io/strimzi-helm/strimzi-drain-cleaner
```

2. Verify that the Drain Cleaner has been deployed successfully:

```
helm ls
```

## 27.4. Using the Strimzi Drain Cleaner

Use the Strimzi Drain Cleaner in combination with the Cluster Operator to move Kafka broker pods from nodes that are being drained. When you run the Strimzi Drain Cleaner, it annotates pods with a rolling update pod annotation. The Cluster Operator performs rolling updates based on the annotation.

### Prerequisites

- You have [deployed the Strimzi Drain Cleaner](#).

### Considerations when using anti-affinity configuration

When using [anti-affinity](#) with your Kafka pods, consider adding a spare worker node to your cluster. Including spare nodes ensures that your cluster has the capacity to reschedule pods during node draining or temporary unavailability of other nodes. When a worker node is drained, and anti-affinity rules restrict pod rescheduling on alternative nodes, spare nodes help prevent restarted pods from becoming unschedulable. This mitigates the risk of the draining operation failing.

### Procedure

1. Drain a specified Kubernetes node hosting the Kafka broker.

```
kubectl get nodes  
kubectl drain <name_of_node> --delete-emptydir-data --ignore-daemonsets  
--timeout=600s --force
```

2. Check the eviction events in the Strimzi Drain Cleaner log to verify that the pods have been annotated for restart.

#### *Strimzi Drain Cleaner log show annotations of pods*

```
INFO ... Received eviction webhook for Pod my-cluster-kafka-0 in namespace my-project
INFO ... Pod my-cluster-kafka-0 in namespace my-project will be annotated for restart
INFO ... Pod my-cluster-kafka-0 in namespace my-project found and annotated for restart
```

3. Check the reconciliation events in the Cluster Operator log to verify the rolling updates.

#### *Cluster Operator log shows rolling updates*

```
INFO PodOperator:68 - Reconciliation #13(timer) Kafka(my-project/my-cluster):
Rolling Pod my-cluster-kafka-0
INFO AbstractOperator:500 - Reconciliation #13(timer) Kafka(my-project/my-cluster): reconciled
```

## 27.5. Adding or renewing the TLS certificates used by the Strimzi Drain Cleaner

The Drain Cleaner uses a webhook to receive eviction notifications from the Kubernetes API. The webhook uses a secure TLS connection and authenticates using TLS certificates. If you are not deploying the Drain Cleaner using the [cert-manager](#) or on Openshift, you must create and renew the TLS certificates. You must then add them to the files used to deploy the Drain Cleaner. The certificates must also be renewed before they expire. To renew the certificates, you repeat the steps used to generate and add the certificates to the initial deployment of the Drain Cleaner.

Generate and add certificates to the standard installation files or your Helm configuration when deploying the Drain Cleaner on Kubernetes without [cert-manager](#).

**NOTE**

If you are using [cert-manager](#) to deploy the Drain Cleaner, you don't need to add or renew TLS certificates. The same applies when deploying the Drain Cleaner on OpenShift, as OpenShift injects the certificates. In both cases, TLS certificates are added and renewed automatically.

#### *Prerequisites*

- The [OpenSSL](#) TLS management tool for generating certificates.

Use [openssl help](#) for command-line descriptions of the options used.

#### *Generating and renewing TLS certificates*

1. From the command line, create a directory called [tls-certificate](#):

```
mkdir tls-certificate
cd tls-certificate
```

Now use OpenSSL to create the certificates in the `tls-certificate` directory.

2. Generate a CA (Certificate Authority) public certificate and private key:

```
openssl req -nodes -new -x509 -keyout ca.key -out ca.crt -subj "/CN=Strimzi Drain  
Cleaner CA"
```

A `ca.crt` and `ca.key` file are created.

3. Generate a private key for the Drain Cleaner:

```
openssl genrsa -out tls.key 2048
```

A `tls.key` file is created.

4. Generate a CSR (Certificate Signing Request) and sign it by adding the CA public certificate (`ca.crt`) you generated:

```
openssl req -new -key tls.key -subj "/CN=strimzi-drain-cleaner.strimzi-drain-  
cleaner.svc" \  
| openssl x509 -req -CA ca.crt -CAkey ca.key -CAcreateserial -extfile <(printf  
"subjectAltName=DNS:strimzi-drain-cleaner.strimzi-drain-cleaner.svc") -out tls.crt
```

A `tls.crt` file is created.

**NOTE** If you change the name of the Strimzi Drain Cleaner service or install it into a different namespace, you must change the SAN (Subject Alternative Name) of the certificate, following the format `<service_name>. <namespace>. svc`.

5. Encode the CA public certificate into base64.

```
base64 tls-certificate/ca.crt
```

With the certificates generated, add them to the installation files or to your Helm configuration depending on your deployment method.

#### *Adding the TLS certificates to the Drain Cleaner installation files*

1. Copy the base64-encoded CA public certificate as the value for the `caBundle` property of the `070-ValidatingWebhookConfiguration.yaml` installation file:

```
# ...  
clientConfig:  
  service:  
    namespace: "strimzi-drain-cleaner"  
    name: "strimzi-drain-cleaner"
```

```
path: /drainer
port: 443
caBundle: LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0...
# ...
```

2. Create a namespace called `strimzi-drain-cleaner` in your Kubernetes cluster:

```
kubectl create ns strimzi-drain-cleaner
```

3. Create a secret named `strimzi-drain-cleaner` with the `tls.crt` and `tls.key` files you generated:

```
kubectl create secret tls strimzi-drain-cleaner \
-n strimzi-drain-cleaner \
--cert=tls-certificate/tls.crt \
--key=tls-certificate/tls.key
```

The secret is used in the Drain Cleaner deployment.

*Example secret for the Drain Cleaner deployment*

```
apiVersion: v1
kind: Secret
metadata:
# ...
name: strimzi-drain-cleaner
namespace: strimzi-drain-cleaner
# ...
data:
tls.crt: LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS...
tls.key: LS0tLS1CRUdJTiBSU0EgUFJJVkFURSBLR...
```

You can now use the certificates and updated installation files to deploy the Drain Cleaner using [installation files](#).

*Adding the TLS certificates to a Helm deployment*

1. Edit the `values.yaml` configuration file used in the Helm deployment.
2. Set the `certManager.create` parameter to `false`.
3. Set the `secret.create` parameter to `true`.
4. Copy the certificates as `secret` parameters.

*Example secret configuration for the Drain Cleaner deployment*

```
# ...
certManager:
create: false
```

```

secret:
  create: true
  tls_crt: "Cg==" ①
  tls_key: "Cg==" ②
  ca_bundle: "Cg==" ③

```

① The public key (`tls.crt`) signed by the CA public certificate.

② The private key (`tls.key`).

③ The base-64 encoded CA public certificate (`ca.crt`).

You can now use the certificates and updated configuration file to deploy the Drain Cleaner using [Helm](#).

## 27.6. Watching the TLS certificates used by the Strimzi Drain Cleaner

By default, the Drain Cleaner deployment watches the secret containing the TLS certificates it uses for authentication. The Drain Cleaner watches for changes, such as certificate renewals. If it detects a change, it restarts to reload the TLS certificates. The Drain Cleaner installation files enable this behavior by default. But you can disable the watching of certificates by setting the `STRIMZI_CERTIFICATE_WATCH_ENABLED` environment variable to `false` in the [Deployment](#) configuration (`060-Deployment.yaml`) of the Drain Cleaner installation files.

With `STRIMZI_CERTIFICATE_WATCH_ENABLED` enabled, you can also use the following environment variables for watching TLS certificates.

*Table 38. Drain Cleaner environment variables for watching TLS certificates*

Environment Variable	Description	Default
<code>STRIMZI_CERTIFICATE_WATCH_ENABLED</code>	Enables or disables the certificate watch	<code>false</code>
<code>STRIMZI_CERTIFICATE_WATCH_NAMESPACE</code>	The namespace where the Drain Cleaner is deployed and where the certificate secret exists	<code>strimzi-drain-cleaner</code>
<code>STRIMZI_CERTIFICATE_WATCH_POD_NAME</code>	The Drain Cleaner pod name	-
<code>STRIMZI_CERTIFICATE_WATCH_SECRET_NAME</code>	The name of the secret containing TLS certificates	<code>strimzi-drain-cleaner</code>
<code>STRIMZI_CERTIFICATE_WATCH_SECRET_KEYS</code>	The list of fields inside the secret that contain the TLS certificates	<code>tls.crt, tls.key</code>

*Example environment variable configuration to control watch operations*

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: strimzi-drain-cleaner

```

```
labels:
  app: strimzi-drain-cleaner
  namespace: strimzi-drain-cleaner
spec:
  # ...
  spec:
    serviceAccountName: strimzi-drain-cleaner
    containers:
      - name: strimzi-drain-cleaner
        # ...
        env:
          - name: STRIMZI_DRAIN_KAFKA
            value: "true"
          - name: STRIMZI_CERTIFICATE_WATCH_ENABLED
            value: "true"
          - name: STRIMZI_CERTIFICATE_WATCH_NAMESPACE
            valueFrom:
              fieldRef:
                fieldPath: metadata.namespace
          - name: STRIMZI_CERTIFICATE_WATCH_POD_NAME
            valueFrom:
              fieldRef:
                fieldPath: metadata.name
        # ...
```

TIP

Use the [Downward API](#) mechanism to configure `STRIMZI_CERTIFICATE_WATCH_NAMESPACE` and `STRIMZI_CERTIFICATE_WATCH_POD_NAME`.

# Chapter 28. Managing rolling updates

Use annotations to manually trigger a rolling update of Kafka and other operands through the Cluster Operator. Initiate rolling updates of Kafka, Kafka Connect, and MirrorMaker 2 clusters.

Manually performing a rolling update on a specific pod or set of pods is usually only required in exceptional circumstances. However, rather than deleting the pods directly, if you perform the rolling update through the Cluster Operator you ensure the following:

- The manual deletion of the pod does not conflict with simultaneous Cluster Operator operations, such as deleting other pods in parallel.
- The Cluster Operator logic handles the Kafka configuration specifications, such as the number of in-sync replicas.

## 28.1. Performing a rolling update using a pod management annotation

This procedure describes how to trigger a rolling update of Kafka, Kafka Connect, or MirrorMaker 2 clusters. To trigger the update, you add an annotation to the **StrimziPodSet** that manages the pods running on the cluster.

### Prerequisites

To perform a manual rolling update, you need a running Cluster Operator. The cluster for the component you are updating, whether it's Kafka, Kafka Connect, or MirrorMaker 2 must also be running.

### Procedure

1. Find the name of the resource that controls the pods you want to manually update.

For example, if your Kafka cluster is named *my-cluster*, the corresponding name is *my-cluster-kafka*. For a Kafka Connect cluster named *my-connect-cluster*, the corresponding name is *my-connect-cluster-connect*. And for a MirrorMaker 2 cluster named *my-mm2-cluster*, the corresponding name is *my-mm2-cluster-mirrormaker2*.

2. Use **kubectl annotate** to annotate the appropriate resource in Kubernetes.

### Annotating a StrimziPodSet

```
kubectl annotate strimzipodset <cluster_name>-kafka strimzi.io/manual-rolling-update="true"
```

```
kubectl annotate strimzipodset <cluster_name>-connect strimzi.io/manual-rolling-update="true"
```

```
kubectl annotate strimzipodset <cluster_name>-mirrormaker2 strimzi.io/manual-rolling-update="true"
```

3. Wait for the next reconciliation to occur (every two minutes by default). A rolling update of all pods within the annotated resource is triggered, as long as the annotation was detected by the reconciliation process. When the rolling update of all the pods is complete, the annotation is automatically removed from the resource.

## 28.2. Performing a rolling update using a pod annotation

This procedure describes how to manually trigger a rolling update of existing Kafka, Kafka Connect, or MirrorMaker 2 clusters using a Kubernetes [Pod](#) annotation. When multiple pods are annotated, consecutive rolling updates are performed within the same reconciliation run.

### *Prerequisites*

To perform a manual rolling update, you need a running Cluster Operator. The cluster for the component you are updating, whether it's Kafka, Kafka Connect, or MirrorMaker 2, must also be running.

You can perform a rolling update on a Kafka cluster regardless of the topic replication factor used. But for Kafka to stay operational during the update, you'll need the following:

- A highly available Kafka cluster deployment running with nodes that you wish to update.
- Topics replicated for high availability.

Topic configuration specifies a replication factor of at least 3 and a minimum number of in-sync replicas to 1 less than the replication factor.

### *Kafka topic replicated for high availability*

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaTopic
metadata:
  name: my-topic
  labels:
    strimzi.io/cluster: my-cluster
spec:
  partitions: 1
  replicas: 3
  config:
    # ...
    min.insync.replicas: 2
    # ...
```

### *Procedure*

1. Find the name of the [Pod](#) you want to manually update.

Pod naming conventions are as follows:

- <cluster\_name>-kafka-<index\_number> for a Kafka cluster

- <cluster\_name>-connect-<index\_number> for a Kafka Connect cluster
- <cluster\_name>-mirrormaker2-<index\_number> for a MirrorMaker 2 cluster

The <index\_number> assigned to a pod starts at zero and ends at the total number of replicas minus one.

2. Use `kubectl annotate` to annotate the **Pod** resource in Kubernetes:

```
kubectl annotate pod <cluster_name>-kafka-<index_number> strimzi.io/manual-rolling-
update="true"

kubectl annotate pod <cluster_name>-connect-<index_number> strimzi.io/manual-
rolling-update="true"

kubectl annotate pod <cluster_name>-mirrormaker2-<index_number> strimzi.io/manual-
rolling-update="true"
```

3. Wait for the next reconciliation to occur (every two minutes by default). A rolling update of the annotated **Pod** is triggered, as long as the annotation was detected by the reconciliation process. When the rolling update of a pod is complete, the annotation is automatically removed from the **Pod**.

**NOTE**

The reconciliation continues even if the manual rolling update of the cluster fails. This allows the Cluster Operator to recover from certain rectifiable situations that can be addressed later in the reconciliation. For example, it can recreate a missing Persistent Volume Claim (PVC) or Persistent Volume (PV) that caused the update to fail.

# Chapter 29. Finding information on Kafka restarts

After the Cluster Operator restarts a Kafka pod in a Kubernetes cluster, it emits a Kubernetes event into the pod's namespace explaining why the pod restarted. For help in understanding cluster behavior, you can check restart events from the command line.

**TIP** You can export and monitor restart events using metrics collection tools like Prometheus. Use the metrics tool with an *event exporter* that can export the output in a suitable format.

## 29.1. Reasons for a restart event

The Cluster Operator initiates a restart event for a specific reason. You can check the reason by fetching information on the restart event.

Table 39. Restart reasons

Event	Description
CaCertHasOldGeneration	The pod is still using a server certificate signed with an old CA, so needs to be restarted as part of the certificate update.
CaCertRemoved	Expired CA certificates have been removed, and the pod is restarted to run with the current certificates.
CaCertRenewed	CA certificates have been renewed, and the pod is restarted to run with the updated certificates.
ClusterCaCertKeyReplaced	The key used to sign the cluster's CA certificates has been replaced, and the pod is being restarted as part of the CA renewal process.
ConfigChangeRequiresRestart	Some Kafka configuration properties are changed dynamically, but others require that the broker be restarted.
FileSystemResizeNeeded	The file system size has been increased, and a restart is needed to apply it.
KafkaCertificatesChanged	One or more TLS certificates used by the Kafka broker have been updated, and a restart is needed to use them.
ManualRollingUpdate	A user annotated the pod, or the <a href="#">StrimziPodSet</a> set it belongs to, to trigger a restart.
PodForceRestartOnError	An error occurred that requires a pod restart to rectify.
PodHasOldRevision	A disk was added or removed from the Kafka volumes, and a restart is needed to apply the change. When using <a href="#">StrimziPodSet</a> resources, the same reason is given if the pod needs to be recreated.

Event	Description
PodHasOldRevision	The <a href="#">StrimziPodSet</a> that the pod is a member of has been updated, so the pod needs to be recreated. When using <a href="#">StrimziPodSet</a> resources, the same reason is given if a disk was added or removed from the Kafka volumes.
PodStuck	The pod is still pending, and is not scheduled or cannot be scheduled, so the operator has restarted the pod in a final attempt to get it running.
PodUnresponsive	Strimzi was unable to connect to the pod, which can indicate a broker not starting correctly, so the operator restarted it in an attempt to resolve the issue.

## 29.2. Restart event filters

When checking restart events from the command line, you can specify a [field-selector](#) to filter on Kubernetes event fields.

The following fields are available when filtering events with [field-selector](#).

### `regardingObject.kind`

The resource that owns the Pod being restarted, and for restart events, the kind is always [Kafka](#).

### `regarding.namespace`

The namespace that the resource belongs to.

### `regardingObject.name`

The resource's name, for example, [strimzi-cluster](#).

### `regardingObject.uid`

The unique ID of the resource.

### `reason`

The reason the Pod was restarted, for example, [JbodVolumesChanged](#).

### `reportingController`

The reporting component is always [strimzi.io/cluster-operator](#) for Strimzi restart events.

### `source`

`source` is an older version of `reportingController`. The reporting component is always [strimzi.io/cluster-operator](#) for Strimzi restart events.

### `type`

The event type, which is either [Warning](#) or [Normal](#). For Strimzi restart events, the type is [Normal](#).

**NOTE**

In older versions of Kubernetes, the fields using the `regarding` prefix might use an `involvedObject` prefix instead. `reportingController` was previously called

`reportingComponent`.

## 29.3. Checking Kafka restarts

Use a `kubectl` command to list restart events initiated by the Cluster Operator. Filter restart events emitted by the Cluster Operator by setting the Cluster Operator as the reporting component using the `reportingController` or `source` event fields.

### Prerequisites

- The Cluster Operator is running in the Kubernetes cluster.

### Procedure

1. Get all restart events emitted by the Cluster Operator:

```
kubectl -n kafka get events --field-selector  
reportingController=strimzi.io/cluster-operator
```

#### Example showing events returned

LAST SEEN	TYPE	REASON	OBJECT	MESSAGE
2m	Normal	CaCertRenewed	kafka/strimzi-cluster	Rolling Pod
strimzi-cluster-kafka-0		due to CA certificate renewed		
58m	Normal	PodForceRestartOnError	kafka/strimzi-cluster	Rolling Pod
strimzi-cluster-kafka-1		due to Pod needs to be forcibly restarted due to an error		
5m47s	Normal	ManualRollingUpdate	kafka/strimzi-cluster	Rolling Pod
strimzi-cluster-kafka-2		due to Pod was manually annotated to be rolled		

You can also specify a `reason` or other `field-selector` options to constrain the events returned.

Here, a specific reason is added:

```
kubectl -n kafka get events --field-selector  
reportingController=strimzi.io/cluster-operator,reason=PodForceRestartOnError
```

2. Use an output format, such as YAML, to return more detailed information about one or more events.

```
kubectl -n kafka get events --field-selector  
reportingController=strimzi.io/cluster-operator,reason=PodForceRestartOnError -o  
yaml
```

#### Example showing detailed events output

```
apiVersion: v1  
items:
```

```
- action: StrimziInitiatedPodRestart
apiVersion: v1
eventTime: "2022-05-13T00:22:34.168086Z"
firstTimestamp: null
involvedObject:
  kind: Kafka
  name: strimzi-cluster
  namespace: kafka
kind: Event
lastTimestamp: null
message: Rolling Pod strimzi-cluster-kafka-1 due to Pod needs to be forcibly
restarted due to an error
metadata:
  creationTimestamp: "2022-05-13T00:22:34Z"
  generateName: strimzi-event
  name: strimzi-eventwppk6
  namespace: kafka
  resourceVersion: "432961"
  uid: 29fcdb9e-f2cf-4c95-a165-a5efcd48edfc
reason: PodForceRestartOnError
related:
  kind: Pod
  name: strimzi-cluster-kafka-1
  namespace: kafka
reportingController: strimzi.io/cluster-operator
reportingInstance: strimzi-cluster-operator-6458cfb4c6-6bpdp
source: {}
type: Normal
kind: List
metadata:
  resourceVersion: ""
  selfLink: ""
```

The following fields are deprecated, so they are not populated for these events:

- `firstTimestamp`
- `lastTimestamp`
- `source`

# Chapter 30. Converting Strimzi custom resources to the v1 API

This section describes how to prepare Strimzi custom resources and CRDs for the `v1` API before performing an upgrade.

Prepare Strimzi custom resources and CRDs for the `v1` API by using the Strimzi API conversion tool or by applying the required changes manually. The tool is provided with the release artifacts. To use the tool, extract it and run the conversion script from the `bin` directory.

The migration to the `v1` API has two stages.

- Version 0.49 introduces the `v1` API and deprecates the older `v1alpha1`, `v1beta1`, and `v1beta2` versions. (`v1alpha1` and `v1beta1` versions are used only for `KafkaUser` and `KafkaTopic` resources).
- A later release (planned for 0.52.0 or 1.0.0) removes the `v1alpha1`, `v1beta1`, and `v1beta2` APIs and updates all components to use `v1`.

Complete the conversion of custom resources and CRDs after upgrading to version 0.49 or later, and before upgrading to the later release that removes the `v1beta2` API (Strimzi 0.52.0 / 1.0.0). Resources converted to `v1` are compatible with Strimzi versions 0.49 and later.

## 30.1. Conversion tool commands

The Strimzi API conversion tool provides commands to convert Strimzi custom resources and upgrade Custom Resource Definitions (CRDs) to the `v1` API.

Each command runs through a shell script located in the `bin` directory.

Run the tool using the following command format:

```
bin/v1-api-conversion.sh <command>
```

If you are using Windows, run `bin/v1-api-conversion.cmd` instead.

The following commands are available for converting custom resources and upgrading CRDs:

### `convert-file`

Converts Strimzi custom resources from local YAML files. Use this command when applying configuration through GitOps or when direct cluster access is restricted.

### `convert-resource`

Converts existing Strimzi custom resources directly in Kubernetes. Use this command when updating live environments managed by the Cluster Operator.

### `crd-upgrade`

Upgrades Strimzi CRDs to use the `v1` API as the storage version. Run this command only after all

custom resources have been converted.

## help

Use the `help` command to list available commands and options or to display detailed help for a specific command.

```
bin/v1-api-conversion.sh help  
bin/v1-api-conversion.sh help <command>
```

For example, run `bin/v1-api-conversion.sh help convert-file` to see the supported options for converting YAML files.

**NOTE** As an alternative to running the tool locally from your computer, you can also run it as a Kubernetes [Job](#). For more information, see the [README.md](#) file in the conversion tool.

## 30.2. Preparing for conversion

Before running the Strimzi API conversion tool, confirm that your environment meets the following prerequisites.

### 30.2.1. General prerequisites

Applies to all conversion commands (`convert-file`, `convert-resource`, `crd-upgrade`):

- All Strimzi custom resources are backed up.
- You have reviewed any manual changes required for deprecated or unsupported `v1` API fields.

### 30.2.2. Cluster conversion prerequisites

Applies only when running the tool against a Kubernetes cluster (`convert-resource`, `crd-upgrade`):

- The deployed Strimzi Cluster Operator version is 0.49.0 or later.
- You have identified the namespaces containing Strimzi resources to convert.
- The user or service account running the tool has the required RBAC permissions:
  - For `convert-resource`: `list`, `get`, and `replace` permissions for all Strimzi custom resources.
  - For `crd-upgrade`: `patch` permissions for Custom Resource Definitions.

## 30.3. Converting custom resource YAML files

Use the Strimzi API conversion tool with the `convert-file` command to convert Strimzi custom resources from YAML files to the `v1` API.

Each input file can contain multiple resource definitions. The conversion operation converts all Strimzi custom resources while leaving other Kubernetes resources unchanged.

#### *Procedure*

1. Run `convert-file` with the `--file` option to specify the input file.

Choose one of the following output options:

- Use `--in-place` to overwrite the source file.
- Use `--output <file>` to create a new file.
- Omit both options to print the converted resource to standard output.

To report on the operation, add a `--log-level` flag.

Example commands:

```
# Convert the input file and print the output with INFO logging:  
bin/v1-api-conversion.sh convert-file --file <input.yaml> --log-level INFO  
  
# Convert and overwrite the input file:  
bin/v1-api-conversion.sh convert-file --file <input.yaml> --in-place  
  
# Convert and write to a new file:  
bin/v1-api-conversion.sh convert-file --file <input.yaml> --output <output.yaml>  
  
# Convert multiple input files and print the output:  
bin/v1-api-conversion.sh convert-file --file <input1.yaml> --file <input2.yaml>
```

2. Review the log output to ensure that all custom resources converted successfully and address any errors. Inspect the output YAML and confirm that the `apiVersion` for Strimzi resources is set to `kafka.strimzi.io/v1`.
3. Apply the converted files to the cluster using `kubectl apply -f` or `kubectl replace -f`.
4. Ensure that all conversions complete successfully before proceeding to upgrade the CRDs.

## 30.4. Converting custom resources in a Kubernetes cluster

Use the Strimzi API conversion tool with the `convert-resource` command to convert existing Strimzi custom resources directly in a Kubernetes cluster to the `v1` API.

#### *Procedure*

1. Run `convert-resource` using one of the following options:

- `--namespace <namespace>` to convert resources in a specific namespace
- `--all-namespaces` to convert resources in all namespaces
- Omit both options to convert resources in the current namespace
- `--kind <kind>` to convert only resources of a specific kind (for example, `Kafka`, `KafkaConnect`, or `KafkaBridge`)

- `--name <name>` with `--kind` to convert a single resource by name

To report on the operation, add a `--log-level` flag.

Example commands:

```
# Convert all Strimzi resources in the current namespace with INFO logging:  
bin/v1-api-conversion.sh convert-resource --log-level INFO  
  
# Convert all Strimzi resources in all namespaces:  
bin/v1-api-conversion.sh convert-resource --all-namespaces  
  
# Convert all Strimzi resources in the 'my-kafka' namespace:  
bin/v1-api-conversion.sh convert-resource --namespace my-kafka  
  
# Convert only Kafka resources in all namespaces:  
bin/v1-api-conversion.sh convert-resource --all-namespaces --kind Kafka  
  
# Convert 'Kafka' and 'KafkaConnect' resources in all namespaces:  
bin/v1-api-conversion.sh convert-resource --all-namespaces --kind Kafka --kind KafkaConnect  
  
# Convert a Kafka cluster named 'my-cluster' in namespace 'my-kafka':  
bin/v1-api-conversion.sh convert-resource --namespace my-kafka --kind Kafka  
--name my-cluster
```

2. Review the log output to ensure that all custom resources converted successfully and address any errors.
3. Ensure that all conversions complete successfully before proceeding to upgrade the CRDs.

## 30.5. Running the conversion tool as a Kubernetes job

Use the Strimzi API conversion tool with the `convert-resource` command to convert custom resources by running the tool as a Kubernetes `Job` when `kubectl` you cannot access the cluster locally or do not have a direct connection.

### *Procedure*

1. Create a manifest that defines the resources required to run the conversion tool in the cluster. Include the following:
  - A `ServiceAccount` that runs the job
  - RBAC permissions (`ClusterRole` and `ClusterRoleBinding`) that allow access to Strimzi custom resources and CRDs
  - A `Job` resource that runs the conversion tool using those permissions

If you are running the job in a specific namespace, ensure that the namespace name is the same in the `ServiceAccount`, `ClusterRoleBinding`, and `Job` definitions. The following example combines all required resources into one YAML file:

```

apiVersion: v1
kind: ServiceAccount
metadata:
  name: strimzi-v1-api-conversion
  namespace: <namespace>
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: strimzi-v1-api-conversion
rules:
  - apiGroups:
      - kafka.strimzi.io
    resources:
      - kafkas
      - kafkaconnects
      - kafkaconnectors
      - kafkabridges
      - kafkamirrormaker2s
      - kafkarebalances
      - kafkatopics
      - kafkausers
    verbs:
      - get
      - list
      - patch
      - update
  - apiGroups:
      - core.strimzi.io
    resources:
      - strimzipodsets
    verbs:
      - get
      - list
      - patch
      - update
  - apiGroups:
      - apiextensions.k8s.io
    resources:
      - customresourcedefinitions
      - customresourcedefinitions/status
  resourceNames:
    - kafkabridges.kafka.strimzi.io
    - kafkaconnectors.kafka.strimzi.io
    - kafkaconnects.kafka.strimzi.io
    - kafkamirrormaker2s.kafka.strimzi.io
    - kafkaconnectors.kafka.strimzi.io
    - kafkarebalances.kafka.strimzi.io
    - kafkas.kafka.strimzi.io

```

```

    - kafkatopics.kafka.strimzi.io
    - kafkausers.kafka.strimzi.io
    - strimzipodsets.core.strimzi.io
  verbs:
    - get
    - list
    - patch
    - update
  - apiGroups:
    - ""
resources:
  # Required when converting KafkaBridge metrics configuration
  - configmaps
  verbs:
    - get
    - list
    - create
    - patch
    - update
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: strimzi-v1-api-conversion
subjects:
  - kind: ServiceAccount
    name: strimzi-v1-api-conversion
    namespace: <namespace>
roleRef:
  kind: ClusterRole
  name: strimzi-v1-api-conversion
  apiGroup: rbac.authorization.k8s.io
---
apiVersion: batch/v1
kind: Job
metadata:
  name: strimzi-v1-api-conversion
  # Namespace where the job runs
  namespace: <namespace>
spec:
  template:
    spec:
      serviceAccountName: strimzi-v1-api-conversion
      containers:
        - name: strimzi-v1-api-conversion
          # Use an image that includes the conversion tool
          image: quay.io/strimzi/operator:0.50.0
          command:
            - /opt/v1-api-conversion/bin/v1-api-conversion.sh
            - convert-resource

```

```
restartPolicy: OnFailure
```

2. Apply the manifest to the cluster:

```
kubectl apply -f strimzi-v1-api-conversion-job.yaml
```

3. Check the job is complete:

```
kubectl get jobs -n <namespace>
kubectl logs job/strimzi-v1-api-conversion -n <namespace>
```

4. Verify that the converted resources now use the [kafka.strimzi.io/v1](#) API version.

If direct CLI access is unavailable, check the logs or monitoring dashboard to confirm that the job completed successfully.

5. Ensure that all conversions complete successfully before proceeding to upgrade the CRDs.
6. (Optional) Switch the job to upgrade CRDs after all conversions complete. Edit the container command to run `crd-upgrade` instead of `convert-resource`:

```
command:
- /opt/v1-api-conversion/bin/v1-api-conversion.sh
- crd-upgrade
```

## 30.6. Upgrading CRD storage version to v1

Use the Strimzi API conversion tool with the `crd-upgrade` command to upgrade the Strimzi Custom Resource Definitions (CRDs) to use the `v1` API as the storage version.

If you run this operation from a Kubernetes `Job`, replace `convert-resource` with `crd-upgrade` in the container command.

### IMPORTANT

This command finalizes the API conversion by removing the `v1beta2` version from the CRDs. You can run this command any time after converting all your custom resources to `v1`, but it **must** be completed as a **final step just before** upgrading to the release that completely removes the `v1beta2` API (planned for 0.52.0 or 1.0.0).

#### Prerequisites

- All Strimzi custom resources have been converted to `v1`.
- Permissions to patch CRD resources and replace their status.

#### Procedure

1. Run `crd-upgrade`:

```
bin/v1-api-conversion.sh crd-upgrade
```

- Verify that all Strimzi CRDs use the v1 API as the storage version:

```
kubectl get crd -o name | grep kafka.strimzi.io | while read crd; do  
  echo -n "$crd "  
  kubectl get "$crd" -o jsonpath='{.status.storedVersions}{"\n"}'  
done
```

Confirm that all CRDs return [v1] as the stored version.

- After upgrading the CRDs, use only the properties supported in the v1 API in all custom resources.

## 30.7. Updating custom resources for the v1 API

Use this section to review the configuration changes introduced in the Strimzi v1 API and determine which updates are required for your deployment.

The reference information describes updates for each Strimzi custom resource type. Some updates are applied automatically by the API conversion tool, while others require manual editing of custom resources.

The following sections describe the manual and automatic updates required for each Strimzi custom resource type.

### 30.7.1. Manual and automatic v1 API changes

The Strimzi v1 API introduces structural and configuration changes to custom resources. Some properties and sections from earlier API versions have been removed or replaced.

Custom resources can be updated for the v1 API in two ways:

- By using the Strimzi API conversion tool to apply supported automatic changes.
- By editing and reapplying resources manually without using the tool.

Even when the conversion tool is used, some changes must still be applied manually because the tool cannot modify or remove the unsupported configuration. The reference information in this section lists which changes the tool can perform automatically and which require manual updates.

#### *Manual changes*

Manual changes are updates that the conversion tool does not apply automatically.

To apply a manual change:

- Edit the affected custom resource.
- Update or remove the required properties.

3. Reapply the resource using `kubectl apply -f` or `kubectl replace -f`.

#### *Automatic changes*

Automatic changes are updates that the conversion tool applies during conversion. Even when the conversion tool is used, some manual changes are still required for the custom resources.

### 30.7.2. Kafka

Changes required for the `Kafka` custom resource in the `v1` API.

#### **Manual changes required**

The conversion tool does not apply the following updates. You must make these changes manually to ensure compatibility:

- Ensure the `.spec` section is present (required).
- Replace unsupported authentication `type: oauth` with `type: custom`.
- In `type: custom` authentication, remove the `secrets` property. Mount secrets using the `template` section instead.
- Replace unsupported authorization types `keycloak` and `opa` with `type: custom`.
- Remove the `.spec.kafka.resources` property. Define resource requests and limits in `KafkaNodePool` resources.

#### **Automatic changes supported**

The following updates are applied automatically by the conversion tool during conversion. If the conversion tool is not used, the same changes can be made manually.

- Removes the `.spec.zookeeper` section. ZooKeeper-based clusters are not supported.
- Removes the `.spec.jmxTrans` section. JMXTrans is not supported.
- Removes the following Kafka properties:
  - `.spec.kafka.replicas` (replicas are configured through `KafkaNodePool` resources)
  - `.spec.kafka.storage` (storage is configured through `KafkaNodePool` resources)
  - `.spec.kafka.template.statefulset`
- Removes the following Cruise Control properties:
  - `.spec.cruiseControl.tlsSidecar`
  - `.spec.cruiseControl.template.tlsSidecarContainer`
  - `.spec.cruiseControl.brokerCapacity.disk`
  - `.spec.cruiseControl.brokerCapacity.cpuUtilization`
- Removes the `.spec.kafkaExporter.template.service` property.
- Removes the following Entity Operator properties:
  - `.spec.entityOperator.tlsSidecar`

- .spec.entityOperator.template.tlsSidecarContainer
- .spec.entityOperator.topicOperator.topicMetadataMaxAttempts
- .spec.entityOperator.topicOperator.zookeeperSessionTimeoutSeconds
- .spec.entityOperator.userOperator.zookeeperSessionTimeoutSeconds
- Replaces .spec.entityOperator.topicOperator.reconciliationIntervalSeconds with .spec.entityOperator.topicOperator.reconciliationIntervalMs (value multiplied by 1000).
- Replaces .spec.entityOperator.userOperator.reconciliationIntervalSeconds with .spec.entityOperator.userOperator.reconciliationIntervalMs (value multiplied by 1000).
- Removes the following properties from YAML definitions that include the status section:
  - .status.registeredNodeIds
  - .status.kafkaMetadataState
  - .status.listeners[].type

Status removal applies only to local YAML files (using the `convert-file` command).

**NOTE** The conversion tool does not modify the status of live resources within a Kubernetes cluster, as their status is actively managed by the Cluster Operator.

### 30.7.3. KafkaConnect

Changes required for the `KafkaConnect` custom resource in the `v1` API.

#### Manual changes required

The conversion tool does not apply the following updates. You must make these changes manually to ensure compatibility:

- Ensure the `.spec` section is present (required).
- Replace unsupported authentication `type: oauth` with `type: custom`.
- Remove the `.spec.externalConfiguration` property. Use the `.spec.template` section to add custom volumes or environment variables.

#### Automatic changes supported

The following updates are applied automatically by the conversion tool during conversion. If the conversion tool is not used, you can apply the same changes manually:

- Adds the `.spec.replicas` property if missing, defaulting to 3.
- Renames .spec.build.output.additionalKanikoOptions to .spec.build.output.additionalBuildOptions.
- Removes the `.spec.template.deployment` property.
- Removes unsupported tracing `type: jaeger`.
- Adds the following required properties and sets them from existing configuration or default values:

- `.spec.groupId`: set from `.spec.config.group.id`, or default to `connect-cluster`.
- `.spec.configStorageTopic`: set from `.spec.config.config.storage.topic`, or default to `connect-cluster-configs`.
- `.spec.offsetStorageTopic`: set from `.spec.config.offset.storage.topic`, or default to `connect-cluster-offsets`.
- `.spec.statusStorageTopic`: set from `.spec.config.status.storage.topic`, or default to `connect-cluster-status`.

**NOTE**

After setting the group ID and internal topic properties, remove the corresponding values from `.spec.config`.

### 30.7.4. KafkaMirrorMaker2

Changes required for the `KafkaMirrorMaker2` custom resource in the `v1` API.

#### Manual changes required

The conversion tool does not apply the following updates. You must make these changes manually to ensure compatibility:

- Ensure the `.spec` section is present (required).
- Replace unsupported authentication `type: oauth` for source or target Kafka clusters with `type: custom`.
- Remove the `.spec.externalConfiguration` property. Use the `.spec.template` section to add custom volumes or environment variables.

#### Automatic changes supported

The following updates are applied automatically by the conversion tool during conversion. If the conversion tool is not used, you can apply the same changes manually:

- Adds the `.spec.replicas` property if missing.
- Removes the `.spec.template.deployment` property.
- Removes unsupported tracing `type: jaeger`.
- Removes the heartbeat connector property: `.spec.mirrors[].heartbeatConnector`.
  - If you want to configure the `MirrorHeartbeatConnector`, use the `KafkaConnect` and `KafkaConnector` custom resources directly.
- Updates connector pause handling: removes `.spec.mirrors[].connectors[].pause`. If `pause: true` was set, it is converted to `state: paused`.
- Renames exclude pattern properties:
  - `.spec.mirrors[].topicBlacklistPattern` to `.spec.mirrors[].topicsExcludePattern`
  - `.spec.mirrors[].groupBlacklistPattern` to `.spec.mirrors[].groupsExcludePattern`
- Replaces cluster properties:

- Removes `.spec.connectCluster`, `.spec.clusters`, `.spec.mirrors[].sourceCluster`, and `.spec.mirrors[].targetCluster`.
- Configures the target and sources using `.spec.target` and `.spec.mirrors[].source`.
- Adds the following required properties and sets them from existing configuration or default values:
  - `.spec.groupId`: set from `.spec.config.group.id`, or default to `connect-cluster`.
  - `.spec.configStorageTopic`: set from `.spec.config.config.storage.topic`, or default to `connect-cluster-configs`.
  - `.spec.offsetStorageTopic`: set from `.spec.config.offset.storage.topic`, or default to `connect-cluster-offsets`.
  - `.spec.statusStorageTopic`: set from `.spec.config.status.storage.topic`, or default to `connect-cluster-status`.

**NOTE**

After setting the target-side properties, remove the corresponding entries from the target `config`.

### 30.7.5. KafkaBridge

Changes required for the `KafkaBridge` custom resource in the `v1` API.

#### Manual changes required

The conversion tool does not apply the following updates. You must make these changes manually to ensure compatibility:

- Ensure the `.spec` section is present (required).
- Replace unsupported authentication `type: oauth` with `type: custom`.

#### Automatic changes supported

The following updates are applied automatically by the conversion tool during conversion. If the conversion tool is not used, you can apply the same changes manually:

- Adds the `.spec.replicas` property if missing, defaulting to `1`.
- Removes unsupported tracing `type: jaeger`.
- Removes the `.spec.enableMetrics` property. If `.spec.enableMetrics` was set to `true`, a new `ConfigMap` is created automatically with the metrics configuration. Adds the `.spec.metricsConfig` property to reference this `ConfigMap`.

### 30.7.6. KafkaConnector

Changes required for the `KafkaConnector` custom resource in the `v1` API.

## **Manual changes required**

The conversion tool does not apply the following updates. You must make these changes manually to ensure compatibility:

- Ensure the `.spec` section is present (required).

## **Automatic changes supported**

The following updates are applied automatically by the conversion tool during conversion. If the conversion tool is not used, you can apply the same changes manually:

- Removes the `.spec.pause` property. If `pause: true` was set, it is converted to `state: paused`.

### **30.7.7. KafkaRebalance**

Changes required for the `KafkaRebalance` custom resource in the `v1` API.

## **Manual changes required**

The conversion tool does not apply the following updates. You must make these changes manually to ensure compatibility:

- Ensure the `.spec` section is present (required).

### **30.7.8. KafkaNodePool**

Changes required for the `KafkaNodePool` custom resource in the `v1` API.

## **Manual changes required**

The conversion tool does not apply the following updates. You must make these changes manually to ensure compatibility:

- Ensure the `.spec` section is present (required).

## **Automatic changes supported**

The following updates are applied automatically by the conversion tool during conversion. If the conversion tool is not used, you can apply the same changes manually:

- Removes the `overrides` property from `.spec.storage`.

To configure different storage settings for specific nodes, define separate `KafkaNodePool` resources.

### **30.7.9. KafkaTopic**

Changes required for the `KafkaTopic` custom resource in the `v1` API.

## Manual changes required

The conversion tool does not apply the following updates. You must make these changes manually to ensure compatibility:

- Ensure the `.spec` section is present (required).

### 30.7.10. KafkaUser

Changes required for the `KafkaUser` custom resource in the `v1` API.

#### Manual changes required

The conversion tool does not apply the following updates. You must make these changes manually to ensure compatibility:

- Ensure the `.spec` section is present (required).

#### Automatic changes supported

The following updates are applied automatically by the conversion tool during conversion. If the conversion tool is not used, you can apply the same changes manually:

- Replaces the `operation` property in `.spec.authorization.acls[]` with the `operations` array. **This change must be applied before upgrading to Strimzi version 0.49.0 or later.**

### 30.7.11. StrimziPodSet

The `StrimziPodSet` is an internal resource managed automatically by the Cluster Operator. Conversion is handled by Strimzi.

Changes required for the `StrimziPodSet` custom resource in the `v1` API (for reference):

- The `.spec` section is required.

# Chapter 31. Upgrading Strimzi

Download the latest Strimzi [deployment files](#) and upgrade your Strimzi installation to version 0.50.0 to benefit from new features, performance improvements, and enhanced security options. During the upgrade, Kafka is also updated to the latest supported version, introducing additional features and bug fixes to your Strimzi deployment.

Use the same method to upgrade the Cluster Operator as the initial method of deployment. For example, if you used the Strimzi installation files, modify those files to perform the upgrade. After you have upgraded your Cluster Operator to 0.50.0, the next step is to upgrade all Kafka nodes to the latest supported version of Kafka. Kafka upgrades are performed by the Cluster Operator through rolling updates of the Kafka nodes.

If you encounter any issues with the new version, Strimzi can be [downgraded](#) to the previous version.

For topics configured with high availability (replication factor of at least 3 and evenly distributed partitions), the upgrade process should not cause any downtime for consumers and producers.

The upgrade triggers rolling updates, where brokers are restarted one by one at different stages of the process. During this time, overall cluster availability is temporarily reduced, which may increase the risk of message loss in the event of a broker failure.

## 31.1. Required upgrade sequence

To upgrade brokers and clients without downtime, you *must* complete the Strimzi upgrade procedures in the following order:

1. Make sure your Kubernetes cluster version is supported.

Strimzi 0.50.0 requires Kubernetes 1.27 and later.

You can [upgrade Kubernetes with minimal downtime](#).

2. [Ensure Kafka clusters are KRaft-based](#).

If upgrading from a version earlier than 0.39, you **must** migrate from ZooKeeper to KRaft **before** upgrading the Cluster Operator. Upgrades from ZooKeeper-based clusters are not supported.

3. [Prepare for v1 API conversion](#).

If upgrading from a version prior to 0.49, this requirement includes a [KafkaUser](#) update that **must** be done **before** upgrading the Cluster Operator.

4. [Upgrade the Cluster Operator](#).

5. Update the [Kafka version](#) and [metadataVersion](#).

**NOTE**

From Strimzi 0.39, upgrades and downgrades between KRaft-based clusters are

supported.

## 31.2. Upgrading Kubernetes with minimal downtime

If you are upgrading Kubernetes, refer to the Kubernetes upgrade documentation to check the upgrade path and the steps to upgrade your nodes correctly. Before upgrading Kubernetes, [check the supported versions for your version of Strimzi](#).

When performing your upgrade, ensure the availability of your Kafka clusters by following these steps:

1. Configure pod disruption budgets
2. Roll pods using one of these methods:
  - a. Use the Strimzi Drain Cleaner (recommended)
  - b. Apply an annotation to your pods to roll them manually

For Kafka to stay operational, topics must also be replicated for high availability. This requires topic configuration that specifies a replication factor of at least 3 and a minimum number of in-sync replicas to 1 less than the replication factor.

*Kafka topic replicated for high availability*

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaTopic
metadata:
  name: my-topic
  labels:
    strimzi.io/cluster: my-cluster
spec:
  partitions: 1
  replicas: 3
  config:
    # ...
    min.insync.replicas: 2
    # ...
```

In a highly available environment, the Cluster Operator maintains a minimum number of in-sync replicas for topics during the upgrade process so that there is no downtime.

### 31.2.1. Rolling pods using Drain Cleaner

When using the Strimzi Drain Cleaner to evict nodes during Kubernetes upgrade, it annotates pods with a manual rolling update annotation to inform the Cluster Operator to perform a rolling update of the pod that should be evicted and have it moved away from the Kubernetes node that is being upgraded.

For more information, see [Evicting pods with the Strimzi Drain Cleaner](#).

### 31.2.2. Rolling pods manually (alternative to Drain Cleaner)

As an alternative to using the Drain Cleaner to roll pods, you can trigger a manual rolling update of pods through the Cluster Operator. Using [Pod](#) resources, rolling updates restart the pods of resources with new pods. To replicate the operation of the Drain Cleaner by keeping topics available, you must also set the `maxUnavailable` value to zero for the pod disruption budget. Reducing the pod disruption budget to zero prevents Kubernetes from evicting pods automatically.

*Specifying a pod disruption budget*

```
apiVersion: kafka.strimzi.io/v1
kind: Kafka
metadata:
  name: my-cluster
  namespace: myproject
spec:
  kafka:
    # ...
    template:
      podDisruptionBudget:
        maxUnavailable: 0
# ...
```

The [PodDisruptionBudget](#) resource created for Kafka clusters covers all associated node pool pods.

Watch the node pool pods that need to be drained and add a pod annotation to make the update. Here, the annotation updates a Kafka pod named [my-cluster-pool-a-1](#).

*Performing a manual rolling update on a Kafka pod*

```
kubectl annotate pod my-cluster-pool-a-1 strimzi.io/manual-rolling-update="true"
```

*Additional resources*

- [Draining pods using the Strimzi Drain Cleaner](#)
- [Performing a rolling update using a pod annotation](#)
- [PodDisruptionBudgetTemplate schema reference](#)
- [Kubernetes documentation](#)[Kubernetes documentation<sup>^</sup>]

## 31.3. Migrating to KRaft from versions earlier than 0.39

Direct upgrades to Strimzi 0.50.0 from versions earlier than 0.39 are **not supported**.

Before you can upgrade to 0.50.0, you **must** follow this required path:

1. Upgrade your cluster to a Strimzi version between 0.39 and 0.45.
2. Migrate your cluster from ZooKeeper to KRaft.

For more information, see [Using Kafka in KRaft mode](#).

After your cluster is running on KRaft, you can proceed with the upgrade to Strimzi 0.50.0.

## 31.4. Converting to v1 API from versions earlier than 0.49

When upgrading from a version prior to Strimzi 0.49.0, you **must** convert your custom resources and CRDs to the `v1` API. The `v1beta2` API is deprecated and will be removed in a future release (planned for 0.52.0 or 1.0.0).

**IMPORTANT**

This conversion applies to KRaft-based clusters. If you are upgrading from a version earlier than 0.39, you must first [migrate to KRaft](#) before proceeding with this conversion and the upgrade.

Follow this specific sequence for the `v1` API conversion:

**1. Before upgrading the Cluster Operator to 0.49.0 or later:**

- If you have `KafkaUser` resources using the `.spec.authorization.acls[].operation` field (singular), you **must** update them to use `.spec.authorization.acls[].operations` (plural).

**2. After upgrading the Cluster Operator to 0.49.0 or later:**

- You can now convert all other custom resources and CRDs to the `v1` API.
- This step **must** be done **after** the Cluster Operator is upgraded, as the new operator version is required to interpret the `v1` format.
- This conversion can be done any time after the 0.49.0 upgrade but **must** be completed before the future release that removes the `v1beta2` API.

For complete conversion instructions, see [Converting Strimzi custom resources to the `v1` API](#).

## 31.5. Upgrading the Cluster Operator

Use the same method to upgrade the Cluster Operator as the initial method of deployment.

### 31.5.1. Upgrading the Cluster Operator using installation files

This procedure describes how to upgrade a Cluster Operator deployment to use Strimzi 0.50.0.

Follow this procedure if you deployed the Cluster Operator using the installation YAML files in the `install/cluster-operator/` directory. The steps include the necessary configuration changes when the Cluster Operator watches multiple or all namespaces.

The availability of Kafka clusters managed by the Cluster Operator is not affected by the upgrade operation.

**NOTE**

Refer to the documentation supporting a specific version of Strimzi for information on how to upgrade to that version.

## Prerequisites

- An existing Cluster Operator deployment is available.
- You have [downloaded the release artifacts for Strimzi 0.50.0](#).
- You need an account with permission to create and manage [CustomResourceDefinition](#) and RBAC ([ClusterRole](#), and [RoleBinding](#)) resources.

Before upgrading, review the feature gates available in this release. Some feature gates provide functionality that has not yet graduated to general availability and can be enabled or disabled through the `STRIMZI_FEATURE_GATES` environment variable in the Cluster Operator configuration. If you want to change the state of a feature gate, configure it before you apply the upgrade.

For information on the available feature gates, their default states, and how to configure them, see [Feature gates](#).

## Procedure

1. Take note of any configuration changes made during the previous Cluster Operator installation.

Any changes will be **overwritten** by the new version of the Cluster Operator.

2. Update your custom resources to reflect the supported configuration options available for Strimzi version 0.50.0.
3. Modify the installation files for the new Cluster Operator version to reflect the namespace in which the Cluster Operator is running.

On Linux, use:

```
sed -i 's/namespace: .*/namespace: my-cluster-operator-namespace/' install/cluster-operator/*RoleBinding*.yaml
```

On MacOS, use:

```
sed -i '' 's/namespace: .*/namespace: my-cluster-operator-namespace/' install/cluster-operator/*RoleBinding*.yaml
```

4. If you modified environment variables in the [Deployment](#) configuration, edit the `060-Deployment-strimzi-cluster-operator.yaml` file to use those environment variables.
  - If the Cluster Operator is watching multiple namespaces, add the list of namespaces to the `STRIMZI_NAMESPACE` environment variable.
  - If the Cluster Operator is watching all namespaces, specify `value: "*"` for the `STRIMZI_NAMESPACE` environment variable.
5. If the Cluster Operator is watching more than one namespace, update the role bindings.
  - If watching multiple namespaces, replace the `namespace` in the [RoleBinding](#) installation files with the actual namespace name and create the role bindings for **each** namespace:

### *Creating role bindings for a namespace*

```
kubectl create -f install/cluster-operator/020-RoleBinding-strimzi-cluster-operator.yaml -n <watched_namespace>
kubectl create -f install/cluster-operator/023-RoleBinding-strimzi-cluster-operator.yaml -n <watched_namespace>
kubectl create -f install/cluster-operator/031-RoleBinding-strimzi-cluster-operator-entity-operator-delegation.yaml -n <watched_namespace>
```

For example, if the Cluster Operator is watching three namespaces, create three sets of role bindings by substituting `<watched_namespace>` with the name of each namespace.

- If watching all namespaces, recreate the cluster role bindings that grant cluster-wide access (if needed):

### *Granting cluster-wide access using role bindings*

```
kubectl create clusterrolebinding strimzi-cluster-operator-namespaced
--clusterrole=strimzi-cluster-operator-namespaced --serviceaccount my-cluster-
operator-namespace:strimzi-cluster-operator
kubectl create clusterrolebinding strimzi-cluster-operator-watched
--clusterrole=strimzi-cluster-operator-watched --serviceaccount my-cluster-
operator-namespace:strimzi-cluster-operator
kubectl create clusterrolebinding strimzi-cluster-operator-entity-operator-
delegation --clusterrole=strimzi-entity-operator --serviceaccount my-cluster-
operator-namespace:strimzi-cluster-operator
```

6. When you have an updated configuration, deploy it along with the rest of the installation resources:

```
kubectl replace -f install/cluster-operator
```

Wait for the rolling updates to complete.

7. If the new operator version no longer supports the Kafka version you are upgrading from, an error message is returned.

To resolve this, upgrade to a supported Kafka version:

- Edit the `Kafka` custom resource.
- Change the `spec.kafka.version` property to a supported Kafka version.

If no error message is returned, you can proceed to the next step and upgrade the Kafka version later.

8. Get the image for the Kafka pod to ensure the upgrade was successful:

```
kubectl get pods my-cluster-kafka-0 -o jsonpath='{.spec.containers[0].image}'
```

The image tag shows the new Strimzi version followed by the Kafka version:

```
quay.io/stimzi/kafka:0.50.0-kafka-4.1.1
```

You can also [check the upgrade has completed successfully from the status of the Kafka resource](#).

The Cluster Operator is upgraded to version 0.50.0, but the version of Kafka running in the cluster it manages is unchanged.

### 31.5.2. Upgrading the Cluster Operator using the OperatorHub

If you deployed Strimzi from [OperatorHub.io](#), use the Operator Lifecycle Manager (OLM) to change the update channel for the Strimzi operators to a new Strimzi version.

Updating the channel starts one of the following types of upgrade, depending on your chosen upgrade strategy:

- An automatic upgrade is initiated
- A manual upgrade that requires approval before installation begins

**NOTE**

If you subscribe to the *stable* channel, you can get automatic updates without changing channels. However, enabling automatic updates is not recommended because of the potential for missing any pre-installation upgrade steps. Use automatic upgrades only on version-specific channels.

For more information on using OperatorHub to upgrade Operators, see the [Operator Lifecycle Manager documentation](#).

### 31.5.3. Upgrading the Cluster Operator using a Helm chart

If you deployed the Cluster Operator using a Helm chart, use `helm upgrade`.

The `helm upgrade` command does not upgrade the [Custom Resource Definitions for Helm](#). Install the new CRDs manually after upgrading the Cluster Operator. You can download the CRDs from the [GitHub releases page](#) or find them in the `crd` subdirectory inside the Helm Chart.

### 31.5.4. Upgrading the Cluster Operator returns Kafka version error

If you upgrade the Cluster Operator to a version that does not support the current version of Kafka you are using, you get an *unsupported Kafka version* error. This error applies to all installation methods and means that you must upgrade Kafka to a supported Kafka version. Change the `spec.kafka.version` in the [Kafka](#) resource to the supported version.

You can use `kubectl` to check for error messages like this in the `status` of the [Kafka](#) resource.

## Checking the Kafka status for errors

```
kubectl get kafka <kafka_cluster_name> -n <namespace> -o  
jsonpath='{.status.conditions}'
```

Replace `<kafka_cluster_name>` with the name of your Kafka cluster and `<namespace>` with the Kubernetes namespace where the pod is running.

## 31.6. Upgrading Kafka clusters

Upgrade a Kafka cluster to a newer supported Kafka version and KRaft metadata version.

**NOTE** Refer to the Apache Kafka documentation for the latest on support for Kafka upgrades.

### Prerequisites

- The Cluster Operator is up and running.
- Before you upgrade the Kafka cluster, check that the properties of the `Kafka` resource do *not* contain configuration options that are not supported in the new Kafka version.

### Procedure

1. Update the Kafka cluster configuration:

```
kubectl edit kafka <kafka_configuration_file>
```

2. If configured, check that the current `spec.kafka.metadataVersion` is set to a version supported by the version of Kafka you are upgrading to.

For example, the current version is `4.0-IV0` if upgrading from Kafka version `4.0.0` to `4.1.1`:

```
apiVersion: kafka.strimzi.io/v1  
kind: Kafka  
metadata:  
  name: my-cluster  
spec:  
  kafka:  
    replicas: 3  
    metadataVersion: 4.0-IV0  
    version: 4.0.0  
    # ...
```

If `metadataVersion` is not configured, Strimzi automatically updates it to the current default after the update to the Kafka version in the next step.

**NOTE** The value of `metadataVersion` must be a string to prevent it from being

interpreted as a floating point number.

3. Change the `Kafka.spec.kafka.version` to specify the new Kafka version; leave the `metadataVersion` at the default for the *current* Kafka version.

**NOTE**

Changing the `kafka.version` ensures that all brokers in the cluster are upgraded to start using the new broker binaries. During this process, some brokers are using the old binaries while others have already upgraded to the new ones. Leaving the `metadataVersion` unchanged at the current setting ensures that the Kafka brokers and controllers can continue to communicate with each other throughout the upgrade.

For example, if upgrading from Kafka 4.0.0 to 4.1.1:

```
apiVersion: kafka.strimzi.io/v1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    replicas: 3
    metadataVersion: 4.0-IV0 ①
    version: 4.1.1 ②
    # ...
```

① Metadata version is unchanged

② Kafka version is changed to the new version.

4. If the image for the Kafka cluster is defined in `Kafka.spec.kafka.image` of the `Kafka` custom resource, update the `image` to point to a container image with the new Kafka version.

See [Kafka version and image mappings](#)

5. Save and exit the editor, then wait for the rolling updates to upgrade the Kafka nodes to complete.

Check the progress of the rolling updates by watching the pod state transitions:

```
kubectl get pods my-cluster-kafka-0 -o jsonpath='{.spec.containers[0].image}'
```

The rolling updates ensure that each pod is using the broker binaries for the new version of Kafka.

6. If required, set the `version` property for Kafka Connect and MirrorMaker as the new version of Kafka:
  - a. For Kafka Connect, update `KafkaConnect.spec.version`.
  - b. For MirrorMaker 2, update `KafkaMirrorMaker2.spec.version`.

**NOTE**

If you are using custom images that are built manually, you must rebuild those images to ensure that they are up-to-date with the latest Strimzi base image. For example, if you [created a container image from the base Kafka Connect image](#), update the Dockerfile to point to the latest base image and build configuration.

7. If configured, update the Kafka resource to use the new `metadataVersion` version. Otherwise, go to step 9.

For example, if upgrading to Kafka 4.1.1:

```
apiVersion: kafka.strimzi.io/v1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    replicas: 3
    metadataVersion: 4.1-IV1
    version: 4.1.1
    # ...
```

**WARNING**

Exercise caution when changing the `metadataVersion`, as downgrading may not be possible. You cannot downgrade Kafka if the `metadataVersion` for the new Kafka version is higher than the Kafka version you wish to downgrade to. However, understand the potential implications on support and compatibility when maintaining an older version.

8. Wait for the Cluster Operator to update the cluster.

Check the upgrade has completed successfully from the [status of the Kafka resource](#).

*Upgrading client applications*

Ensure all Kafka client applications are updated to use the new version of the client binaries as part of the upgrade process and verify their compatibility with the Kafka upgrade. If needed, coordinate with the team responsible for managing the client applications.

**TIP**

To check that a client is using the latest message format, use the `kafka.server:type=BrokerTopicMetrics,name={Produce|Fetch}MessageConversionsPerSec` metric. The metric shows `0` if the latest message format is being used.

## 31.7. Checking the status of an upgrade

When performing an upgrade (or downgrade), you can check it completed successfully in the status of the `Kafka` custom resource. The status provides information on the Strimzi and Kafka versions being used.

To ensure that you have the correct versions after completing an upgrade, verify the `kafkaVersion` and `operatorLastSuccessfulVersion` values in the Kafka status.

- `operatorLastSuccessfulVersion` is the version of the Strimzi operator that last performed a successful reconciliation.
- `kafkaVersion` is the version of Kafka being used by the Kafka cluster.
- `kafkaMetadataVersion` is the metadata version used by Kafka clusters.

You can use these values to check an upgrade of Strimzi or Kafka has completed.

*Checking an upgrade from the Kafka status*

```
apiVersion: kafka.strimzi.io/v1
kind: Kafka
metadata:
spec:
  # ...
status:
  # ...
  kafkaVersion: 4.1.1
  operatorLastSuccessfulVersion: 0.50.0
  kafkaMetadataVersion: 4.1
```

## 31.8. Strimzi upgrade paths

Two upgrade paths are available for Strimzi.

### Incremental upgrade

An incremental upgrade moves between consecutive minor versions (such as 0.49.1 to 0.50.0), following a supported upgrade path.

### Multi-version upgrade

A multi-version upgrade skips one or more minor versions. While temporary errors due to Kafka version changes may occur, they can typically be resolved during the upgrade.

Before upgrading in production, test your specific scenario in a controlled environment to identify potential issues.

#### 31.8.1. Support for Kafka versions when upgrading

When upgrading Strimzi, it is important to ensure compatibility with the Kafka version being used.

Multi-version upgrades are possible even if the supported Kafka versions differ between the old and new versions. However, if you attempt to upgrade to a new Strimzi version that does not support the current Kafka version, [an error indicating that the Kafka version is not supported is generated](#). In this case, you must upgrade the Kafka version as part of the Strimzi upgrade by changing the `spec.kafka.version` in the `Kafka` custom resource to the supported version for the new Strimzi version.

Review supported Kafka versions in the [Supported versions](#) table on the Strimzi website.

- The **Operators** column lists all released Strimzi versions (the Strimzi version is often called the "Operator version").
- The **Kafka versions** column lists the supported Kafka versions for each Strimzi version.

### 31.8.2. Kafka version and image mappings

When upgrading Kafka, consider your settings for the `STRIMZI_KAFKA_IMAGES` environment variable and the `Kafka.spec.kafka.version` property.

- Each **Kafka** resource can be configured with a `Kafka.spec.kafka.version`, which defaults to the latest supported Kafka version (4.1.1) if not specified.
- The Cluster Operator's `STRIMZI_KAFKA_IMAGES` environment variable provides a mapping (`<kafka_version>=<image>`) between a Kafka version and the image to be used when a specific Kafka version is requested in a given **Kafka** resource. For example, `4.1.1=quay.io/strimzi/kafka:0.50.0-kafka-4.1.1`.
  - If `Kafka.spec.kafka.image` is not configured, the default image for the given version is used.
  - If `Kafka.spec.kafka.image` is configured, the default image is overridden.

**WARNING**

The Cluster Operator cannot validate that an image actually contains a Kafka broker of the expected version. Take care to ensure that the given image corresponds to the given Kafka version.

## 31.9. Strategies for upgrading clients

Upgrading Kafka clients ensures that they benefit from the features, fixes, and improvements that are introduced in new versions of Kafka. Upgraded clients maintain compatibility with other upgraded Kafka components. The performance and stability of the clients might also be improved.

Consider the best approach for upgrading Kafka clients and brokers to ensure a smooth transition. The chosen upgrade strategy depends on whether you are upgrading brokers or clients first. Since Kafka 3.0, you can upgrade brokers and client independently and in any order. The decision to upgrade clients or brokers first depends on several factors, such as the number of applications that need to be upgraded and how much downtime is tolerable.

If you upgrade clients before brokers, some new features may not work as they are not yet supported by brokers. However, brokers can handle producers and consumers running with different versions and supporting different log message versions.

# Chapter 32. Downgrading Strimzi

If you are encountering issues with the version of Strimzi you upgraded to, you can revert to a previous version.

If you installed Strimzi using the YAML installation files, you can use the files from the previous release to perform the downgrade.

Downgrading Strimzi involves two key steps:

1. Downgrading the Kafka cluster to a version supported by the target Cluster Operator (if required).
2. Downgrading the Cluster Operator to the previous release.

If downgrading through multiple versions, you may need to repeat these steps.

Always downgrade Kafka before the Cluster Operator to maintain compatibility. Ensure that the Kafka version you downgrade to is supported by the target Cluster Operator. The metadata version of the Kafka cluster must not be higher than the maximum version supported by the downgraded Kafka version.

**WARNING**

The following downgrade instructions are only suitable if you installed Strimzi using the installation files. If you installed Strimzi using another method, such as [OperatorHub.io](#), downgrade may not be supported by that method unless specified in its documentation.

## 32.1. Downgrading Kafka clusters and client applications

You can downgrade a Kafka cluster to an earlier version only when the metadata format is compatible. Kafka permits downgrades only if there are no metadata changes between your current Kafka version and the target version. In practice, this means the cluster's metadata version must be set to a value that the target Kafka version supports, and there must be no intervening metadata changes between the two versions.

**IMPORTANT**

If the cluster is already using the latest metadata version for the current release, you cannot downgrade. The downgrade is only possible if there is a metadata format that is valid in both the current and target Kafka versions. Consult the Apache Kafka documentation for information on supported metadata version values and downgrade limitations.

### *Prerequisites*

- The Cluster Operator is up and running.
- Before you downgrade the Kafka cluster, check the following for the [Kafka](#) resource:
  - The [Kafka](#) custom resource does not include options that are unsupported by the target Kafka version.

- The target Kafka version supports the metadata version (`spec.kafka.metadataVersion`) you plan to use.

You can verify the current Kafka and metadata version by checking the status of the `Kafka` custom resource.

#### Procedure

1. Update the Kafka cluster configuration:

```
kubectl edit kafka <kafka_configuration_file>
```

2. Change the `metadataVersion` version to a value that is supported by both the current Kafka version and the target version. The Kafka version must be able to operate with this metadata format.

Leave the `Kafka.spec.kafka.version` unchanged at the *current* Kafka version.

For example, if downgrading from Kafka 4.1.1 to 4.0.0:

```
apiVersion: kafka.strimzi.io/v1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    replicas: 3
    metadataVersion: 4.0-IV0 ①
    version: 4.1.1 ②
    # ...
```

① Metadata version is changed to a version supported by the earlier Kafka version.

② Kafka version is unchanged.

**NOTE**

The value of `metadataVersion` must be a string to prevent it from being interpreted as a floating point number.

3. Save the change, and wait for Cluster Operator to update `.status.kafkaMetadataVersion` for the `Kafka` resource.
4. Change the `Kafka.spec.kafka.version` to the previous version.

For example, if downgrading from Kafka 4.1.1 to 4.0.0:

```
apiVersion: kafka.strimzi.io/v1
kind: Kafka
metadata:
  name: my-cluster
spec:
```

```
kafka:  
  replicas: 3  
  metadataVersion: 4.0-IV0 ①  
  version: 4.0.0 ②  
  # ...
```

① Metadata version is supported by the Kafka version.

② Kafka version is changed to the new version.

5. If the image for the Kafka version is different from the image defined in `STRIMZI_KAFKA_IMAGES` for the Cluster Operator, update `Kafka.spec.kafka.image`.

See [Kafka version and image mappings](#).

6. Wait for the Cluster Operator to update the cluster.

You can [check the downgrade has completed successfully](#) from the status of the `Kafka` resource.

7. Downgrade all client applications (consumers) to use the previous version of the client binaries.

The Kafka cluster and clients are now using the previous Kafka version.

## 32.2. Strimzi downgrade paths

Two downgrade paths are available for Strimzi.

### Incremental downgrade

An incremental downgrade moves between consecutive minor versions (such as 0.50.0 to 0.49.1), following a supported downgrade path.

### Multi-version downgrade

A multi-version downgrade skips one or more minor versions. It may result in errors due to configuration changes or incompatible Kafka versions. See [Known limitations in downgrade paths](#) for details on specific paths.

Before downgrading in production, test your specific scenario in a controlled environment to identify potential issues.

### 32.2.1. Support for Kafka versions when downgrading

When downgrading Strimzi, it is important to ensure compatibility with the Kafka version being used.

Multi-version downgrades are possible even if the supported Kafka versions differ between the old and new versions. However, if you attempt to downgrade to an older Strimzi version that does not support the current Kafka version, [an error indicating that the Kafka version is not supported is generated](#). In this case, you must, if possible, downgrade the Kafka version as part of the Strimzi downgrade by changing the `spec.kafka.version` in the `Kafka` custom resource to the supported version for the target Strimzi version. If it is not possible to downgrade the Kafka version (for

example in Strimzi version 0.45.0 or earlier), then the Strimzi downgrade path you are attempting is not supported.

The following table details known limitations for specific downgrade paths.

*Table 40. Known limitations in downgrade paths*

Target Strimzi Version	Limitation
<= 0.45.0	<p>The current Kafka version must be supported by the target Strimzi version.</p> <p>It is only possible to downgrade from unsupported Kafka versions in Strimzi version 0.46.0 and higher.</p>

## 32.3. Downgrading the Cluster Operator

Downgrading the Cluster Operator involves reverting the Strimzi Cluster Operator to a previous version. You may also need to downgrade to a supported Kafka version.

### 32.3.1. Downgrading the Cluster Operator to a previous version

This procedure describes how to downgrade a Cluster Operator deployment to a previous version.

**NOTE**

Kafka clusters managed by Strimzi always use node pools. Downgrading to a cluster without node pools is not possible.

*Prerequisites*

- An existing Cluster Operator deployment is available.
- You have [downloaded the installation files for the previous version](#).

Currently, no feature gates require enabling or disabling before downgrading. If a feature gate introduces such a requirement, the details will be provided here.

*Procedure*

1. Take note of any configuration changes made during the previous Cluster Operator installation.  
Any changes will be **overwritten** by the previous version of the Cluster Operator.
2. Revert your custom resources to reflect the supported configuration options available for the version of Strimzi you are downgrading to.
3. Update the Cluster Operator.
  - a. Modify the installation files for the previous version according to the namespace the Cluster Operator is running in.

On Linux, use:

```
sed -i 's/namespace: .*/namespace: my-cluster-operator-namespace/'
```

```
install/cluster-operator/*RoleBinding*.yaml
```

On MacOS, use:

```
sed -i '' 's/namespace: .*namespace: my-cluster-operator-namespace/'  
install/cluster-operator/*RoleBinding*.yaml
```

- b. If you modified one or more environment variables in your existing Cluster Operator [Deployment](#), edit the [install/cluster-operator/060-Deployment-stimzi-cluster-operator.yaml](#) file to use those environment variables.
4. When you have an updated configuration, deploy it along with the rest of the installation resources:

```
kubectl replace -f install/cluster-operator
```

Wait for the rolling updates to complete.

5. If the target operator version does not support the current Kafka version, an error message is returned.

To resolve this, downgrade to a supported Kafka version if possible:

- a. Edit the [Kafka](#) custom resource.
- b. Change the [spec.kafka.version](#) property to a supported Kafka version.

If no error message is returned, you can proceed to the next step and upgrade the Kafka version later.

6. Get the image for the Kafka pod to ensure the downgrade was successful:

```
kubectl get pod my-cluster-kafka-0 -o jsonpath='{.spec.containers[0].image}'
```

The image tag shows the new Strimzi version followed by the Kafka version. For example, [`<stimzi\_version>-kafka-<kafka\_version>`](#).

You can also [check the downgrade has completed successfully from the status of the Kafka resource](#).

### 32.3.2. Downgrading the Cluster Operator returns Kafka version error

If you downgrade the Cluster Operator to a version that does not support the current version of Kafka you are using, you get an *unsupported Kafka version* error. This error applies to all installation methods and means that you must downgrade Kafka to a supported Kafka version. Change the [spec.kafka.version](#) in the [Kafka](#) resource to the supported version.

You can use [kubectl](#) to check for error messages like this in the [status](#) of the [Kafka](#) resource.

### *Checking the Kafka status for errors*

```
kubectl get kafka <kafka_cluster_name> -n <namespace> -o  
jsonpath='{.status.conditions}'
```

Replace <kafka\_cluster\_name> with the name of your Kafka cluster and <namespace> with the Kubernetes namespace where the pod is running.

# Chapter 33. Uninstalling Strimzi

You can uninstall Strimzi using the CLI or by unsubscribing from OperatorHub.io.

Use the same approach you used to install Strimzi.

When you uninstall Strimzi, you will need to identify resources created specifically for a deployment and referenced from the Strimzi resource.

Such resources include:

- Secrets (Custom CAs and certificates, Kafka Connect secrets, and other Kafka secrets)
- Logging [ConfigMaps](#) (of type `external`)

These are resources referenced by [Kafka](#), [KafkaConnect](#), [KafkaMirrorMaker2](#), or [KafkaBridge](#) configuration.

**WARNING**

When a [CustomResourceDefinition](#) is deleted, custom resources of that type are also deleted. This includes the [Kafka](#), [KafkaConnect](#), [KafkaMirrorMaker2](#), and [KafkaBridge](#) resources managed by Strimzi, as well as the [StrimziPodSet](#) resource Strimzi uses to manage the pods of the Kafka components. In addition, any Kubernetes resources created by these custom resources, such as [Deployment](#), [Pod](#), [Service](#), and [ConfigMap](#) resources, are also removed. Always exercise caution when deleting these resources to avoid unintended data loss.

## 33.1. Uninstalling Strimzi using the CLI

This procedure describes how to use the `kubectl` command-line tool to uninstall Strimzi and remove resources related to the deployment.

*Prerequisites*

- Access to a Kubernetes cluster using an account with `cluster-admin` or `strimzi-admin` permissions.
- You have identified the resources to be deleted.

You can use the following `kubectl` CLI command to find resources and also verify that they have been removed when you have uninstalled Strimzi.

*Command to find resources related to a Strimzi deployment*

```
kubectl get <resource_type> --all-namespaces | grep <kafka_cluster_name>
```

Replace `<resource_type>` with the type of the resource you are checking, such as `secret` or `configmap`.

*Procedure*

1. Delete the Cluster Operator [Deployment](#), related [CustomResourceDefinitions](#), and [RBAC](#) resources.

Specify the installation files used to deploy the Cluster Operator.

```
kubectl delete -f install/cluster-operator
```

2. Delete the resources you identified in the prerequisites.

```
kubectl delete <resource_type> <resource_name> -n <namespace>
```

Replace *<resource\_type>* with the type of resource you are deleting and *<resource\_name>* with the name of the resource.

*Example to delete a secret*

```
kubectl delete secret my-cluster-clients-ca-cert -n my-project
```

## 33.2. Uninstalling Strimzi from OperatorHub.io

This procedure describes how to uninstall Strimzi from OperatorHub.io and remove resources related to the deployment.

You perform the steps using the `kubectl` command-line tool.

### Prerequisites

- Access to a Kubernetes cluster using an account with `cluster-admin` or `strimzi-admin` permissions.
- You have identified the resources to be deleted.

You can use the following `kubectl` CLI command to find resources and also verify that they have been removed when you have uninstalled Strimzi.

*Command to find resources related to a Strimzi deployment*

```
kubectl get <resource_type> --all-namespaces | grep <kafka_cluster_name>
```

Replace *<resource\_type>* with the type of the resource you are checking, such as `secret` or `configmap`.

### Procedure

1. Delete the Strimzi subscription.

```
kubectl delete subscription strimzi-cluster-operator -n <namespace>
```

2. Delete the cluster service version (CSV).

```
kubectl delete csv strimzi-cluster-operator.<version> -n <namespace>
```

3. Remove related CRDs.

```
kubectl get crd -l app=strimzi -o name | xargs kubectl delete
```

# Chapter 34. Cluster recovery from persistent volumes

You can recover a Kafka cluster from persistent volumes (PVs) if they are still present.

## 34.1. Cluster recovery scenarios

Recovering from PVs is possible in the following scenarios:

- Unintentional deletion of a namespace
- Loss of an entire Kubernetes cluster while PVs remain in the infrastructure

The recovery procedure for both scenarios is to recreate the original `PersistentVolumeClaim` (PVC) resources.

### 34.1.1. Recovering from namespace deletion

When you delete a namespace, all resources within that namespace—including PVCs, pods, and services—are deleted. If the `reclaimPolicy` for the PV resource specification is set to `Retain`, the PV retains its data and is not deleted. This configuration allows you to recover from namespace deletion.

*PV configuration to retain data*

```
apiVersion: v1
kind: PersistentVolume
# ...
spec:
  # ...
  persistentVolumeReclaimPolicy: Retain
```

Alternatively, PVs can inherit the reclaim policy from an associated storage class. Storage classes are used for dynamic volume allocation.

By configuring the `reclaimPolicy` property for the storage class, PVs created with this class use the specified reclaim policy. The storage class is assigned to the PV using the `storageClassName` property.

*Storage class configuration to retain data*

```
apiVersion: v1
kind: StorageClass
metadata:
  name: gp2-retain
parameters:
  # ...
# ...
reclaimPolicy: Retain
```

*Storage class specified for PV*

```
apiVersion: v1
kind: PersistentVolume
# ...
spec:
# ...
storageClassName: gp2-retain
```

**NOTE**

When using **Retain** as the reclaim policy, you must manually delete PVs if you intend to delete the entire cluster.

### 34.1.2. Recovering from cluster loss

If you lose the entire Kubernetes cluster, all resources—including PVs, PVCs, and namespaces—are lost. However, it's possible to recover if the physical storage backing the PVs remains intact.

To recover, you need to set up a new Kubernetes cluster and manually reconfigure the PVs to use the existing storage.

## 34.2. Recovering a deleted Kafka cluster

This procedure describes how to recover a deleted Kafka cluster from persistent volumes (PVs) by recreating the original **PersistentVolumeClaim** (PVC) resources.

If the Topic Operator and User Operator are deployed, you can recover **KafkaTopic** and **KafkaUser** resources by recreating them. It is important that you recreate the **KafkaTopic** resources with the same configurations, or the Topic Operator will try to update them in Kafka. This procedure shows how to recreate both resources.

**WARNING**

If the User Operator is enabled and Kafka users are not recreated, users are deleted from the Kafka cluster immediately after recovery.

*Before you begin*

In this procedure, it is essential that PVs are mounted into the correct PVC to avoid data corruption. A **volumeName** is specified for the PVC and this must match the name of the PV.

For more information, see [Configuring Kafka storage](#).

*Procedure*

1. Check information on the PVs in the cluster:

```
kubectl get pv
```

Information is presented for PVs with data.

### *Example PV output*

NAME	RECLAIMPOLICY	CLAIM
pvc-5e9c5c7f-3317-11ea-a650-06e1eadd9a4c ...	Retain ...	myproject/data-0-my-cluster-broker-0
pvc-5e9cc72d-3317-11ea-97b0-0aef8816c7ea ...	Retain ...	myproject/data-0-my-cluster-broker-1
pvc-5ead43d1-3317-11ea-97b0-0aef8816c7ea ...	Retain ...	myproject/data-0-my-cluster-broker-2
pvc-7e1f67f9-3317-11ea-a650-06e1eadd9a4c ...	Retain ...	myproject/data-0-my-cluster-controller-3
pvc-7e21042e-3317-11ea-9786-02deaf9aa87e ...	Retain ...	myproject/data-0-my-cluster-controller-4
pvc-7e226978-3317-11ea-97b0-0aef8816c7ea ...	Retain ...	myproject/data-0-my-cluster-controller-5

- **NAME** is the name of each PV.
- **RECLAIMPOLICY** shows that PVs are retained, meaning that the PV is not automatically deleted when the PVC is deleted.
- **CLAIM** shows the link to the original PVCs.

### 2. Recreate the original namespace:

```
kubectl create namespace myproject
```

Here, we recreate the **myproject** namespace.

### 3. Recreate the original PVC resource specifications, linking the PVCs to the appropriate PV:

#### *Example PVC resource specification*

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: data-0-my-cluster-broker-0
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 100Gi
  storageClassName: gp2-retain
  volumeMode: Filesystem
  volumeName: pvc-7e1f67f9-3317-11ea-a650-06e1eadd9a4c
```

### 4. Edit the PV specifications to delete the **claimRef** properties that bound the original PVC.

### Example PV specification

```
apiVersion: v1
kind: PersistentVolume
metadata:
  annotations:
    kubernetes.io/createdby: aws-ebs-dynamic-provisioner
    pv.kubernetes.io/bound-by-controller: "yes"
    pv.kubernetes.io/provisioned-by: kubernetes.io/aws-ebs
  creationTimestamp: "<date>"
  finalizers:
  - kubernetes.io/pv-protection
  labels:
    failure-domain.beta.kubernetes.io/region: eu-west-1
    failure-domain.beta.kubernetes.io/zone: eu-west-1c
  name: pvc-5ead43d1-3317-11ea-97b0-0aef8816c7ea
  resourceVersion: "39431"
  selfLink: /api/v1/persistentvolumes/pvc-7e226978-3317-11ea-97b0-0aef8816c7ea
  uid: 7efe6b0d-3317-11ea-a650-06e1eadd9a4c
spec:
  accessModes:
  - ReadWriteOnce
  awsElasticBlockStore:
    fsType: xfs
    volumeID: aws://eu-west-1c/vol-09db3141656d1c258
  capacity:
    storage: 100Gi
  claimRef:
    apiVersion: v1
    kind: PersistentVolumeClaim
    name: data-0-my-cluster-kafka-2
    namespace: myproject
    resourceVersion: "39113"
    uid: 54be1c60-3319-11ea-97b0-0aef8816c7ea
  nodeAffinity:
    required:
      nodeSelectorTerms:
      - matchExpressions:
        - key: failure-domain.beta.kubernetes.io/zone
          operator: In
          values:
          - eu-west-1c
        - key: failure-domain.beta.kubernetes.io/region
          operator: In
          values:
          - eu-west-1
  persistentVolumeReclaimPolicy: Retain
  storageClassName: gp2-retain
  volumeMode: Filesystem
```

In the example, the following properties are deleted:

```
claimRef:  
  apiVersion: v1  
  kind: PersistentVolumeClaim  
  name: data-0-my-cluster-broker-2  
  namespace: myproject  
  resourceVersion: "39113"  
  uid: 54be1c60-3319-11ea-97b0-0aef8816c7ea
```

5. Deploy the Cluster Operator:

```
kubectl create -f install/cluster-operator -n myproject
```

6. Recreate all **KafkaTopic** resources by applying the **KafkaTopic** resource configuration:

```
kubectl apply -f <topic_configuration_file> -n myproject
```

7. Recreate all **KafkaUser** resources:

- If user passwords and certificates need to be retained, recreate the user secrets before recreating the **KafkaUser** resources.

If the secrets are not recreated, the User Operator will generate new credentials automatically. Ensure that the recreated secrets have exactly the same name, labels, and fields as the original secrets.

- Apply the **KafkaUser** resource configuration:

```
kubectl apply -f <user_configuration_file> -n myproject
```

8. Deploy the Kafka cluster using the original configuration for the **Kafka** resource. Add the annotation `strimzi.io/pause-reconciliation="true"` to the original configuration for the **Kafka** resource, and then deploy the Kafka cluster using the updated configuration.

```
kubectl apply -f <kafka_resource_configuration>.yaml -n myproject
```

9. Recover the original **clusterId** from logs or copies of the **Kafka** custom resource. Otherwise, you can retrieve it from one of the volumes by spinning up a temporary pod.

```
PVC_NAME="data-0-my-cluster-kafka-0"  
COMMAND="grep cluster.id /disk/kafka-log*/meta.properties | awk -F=' '{print  
\$2}'"  
kubectl run tmp -itq --rm --restart "Never" --image "foo" --overrides "{\"spec\":":
```

```
{"\"containers\":[{\\"name\":\\\"busybox\\\",\\\"image\":\\\"busybox\\\",\\\"command\\\":[\\\"/bin/sh\\\",\\\"-c\\\",\\\"$COMMAND\\\"],\\\"volumeMounts\\\": [{\\\"name\\\":\\\"disk\\\",\\\"mountPath\\\":\\\"/disk\\\"}]}],\\\"volumes\\\": [{\\\"name\\\":\\\"disk\\\",\\\"persistentVolumeClaim\\\":{\\\"claimName\\\":\\\"$PVC_NAME\\\"}}}]}" -n myproject
```

10. Edit the **Kafka** resource to set the `.status.clusterId` with the recovered value:

```
kubectl edit kafka <cluster-name> --subresource status -n myproject
```

11. Unpause the **Kafka** resource reconciliation:

```
kubectl annotate kafka my-cluster strimzi.io/pause-reconciliation=false \
--overwrite -n myproject
```

12. Verify the recovery of the **KafkaTopic** resources:

```
kubectl get kafkatopics -o wide -w -n myproject
```

*Kafka topic status*

NAME	CLUSTER	PARTITIONS	REPLICATION FACTOR	READY
my-topic-1	my-cluster	10	3	True
my-topic-2	my-cluster	10	3	True
my-topic-3	my-cluster	10	3	True

**KafkaTopic** custom resource creation is successful when the `READY` output shows `True`.

13. Verify the recovery of the **KafkaUser** resources:

```
kubectl get kafkausers -o wide -w -n myproject
```

*Kafka user status*

NAME	CLUSTER	AUTHENTICATION	AUTHORIZATION	READY
my-user-1	my-cluster	tls	simple	True
my-user-2	my-cluster	tls	simple	True
my-user-3	my-cluster	tls	simple	True

**KafkaUser** custom resource creation is successful when the `READY` output shows `True`.

# Chapter 35. Tuning Kafka configuration

Fine-tuning the performance of your Kafka deployment involves optimizing various configuration properties according to your specific requirements. This section provides an introduction to common configuration options available for Kafka brokers, producers, and consumers.

While a minimum set of configurations is necessary for Kafka to function, Kafka properties allow for extensive adjustments. Through configuration properties, you can enhance latency, throughput, and overall efficiency, ensuring that your Kafka deployment meets the demands of your applications.

For effective tuning, take a methodical approach. Begin by analyzing relevant metrics to identify potential bottlenecks or areas for improvement. Adjust configuration parameters iteratively, monitoring the impact on performance metrics, and then refine your settings accordingly.

For more information about Apache Kafka configuration properties, see the [Apache Kafka documentation](#).

**NOTE** The guidance provided here offers a starting point for tuning your Kafka deployment. Finding the optimal configuration depends on factors such as workload, infrastructure, and performance objectives.

## 35.1. Tools that help with tuning

The following tools help with Kafka tuning:

- Cruise Control generates optimization proposals that you can use to assess and implement a cluster rebalance
- Strimzi Quotas plugin sets limits on brokers
- Rack configuration spreads broker partitions across racks and allows consumers to fetch data from the nearest replica

## 35.2. Managed broker configurations

When you deploy Strimzi on Kubernetes, you can specify broker configuration through the `config` property of the `Kafka` custom resource. However, certain broker configuration options are managed directly by Strimzi and cannot be set within this `config` property.

As such, you cannot configure the following options through the `config` property:

- `node.id` to specify the ID of the Kafka node
- `log.dirs` directories for log data
- `listeners` to expose the Kafka cluster to clients
- `authorization` mechanisms to allow or decline actions executed by users
- `authentication` mechanisms to prove the identity of users requiring access to Kafka

Node IDs start from 0 (zero) and run sequentially across the Kafka cluster. Log directories are mounted to `/var/lib/kafka/data/kafka-log<pod_id>` based on the `spec.storage` configuration specified in the `KafkaNodePool` custom resource. For JBOD storage, they are mounted at `/var/lib/kafka/data-<volume_id>/kafka-log<pod_id>`.

For a list of exclusions, see the [KafkaClusterSpec schema reference](#).

## 35.3. Kafka broker configuration tuning

Use configuration properties to optimize the performance of Kafka brokers. You can use standard Kafka broker configuration options, except for properties managed directly by Strimzi.

### 35.3.1. Basic broker configuration

A typical broker configuration will include settings for properties related to topics, threads and logs.

*Basic broker configuration properties*

```
# ...
num.partitions=1
default.replication.factor=3
offsets.topic.replication.factor=3
transaction.state.log.replication.factor=3
transaction.state.log.min_isr=2
log.retention.hours=168
log.segment.bytes=1073741824
log.retention.check.interval.ms=300000
num.network.threads=3
num.io.threads=8
num.recovery.threads.per.data.dir=1
socket.send.buffer.bytes=102400
socket.receive.buffer.bytes=102400
socket.request.max.bytes=104857600
group.initial.rebalance.delay.ms=0
# ...
```

### 35.3.2. Replicating topics for high availability

Basic topic properties set the default number of partitions and replication factor for topics, which will apply to topics that are created without these properties being explicitly set, including when topics are created automatically.

```
# ...
num.partitions=1
auto.create.topics.enable=false
default.replication.factor=3
min.insync.replicas=2
replica.fetch.max.bytes=1048576
```

```
# ...
```

For high availability environments, it is advisable to increase the replication factor to at least 3 for topics and set the minimum number of in-sync replicas required to 1 less than the replication factor.

The `auto.create.topics.enable` property is enabled by default so that topics that do not already exist are created automatically when needed by producers and consumers. If you are using automatic topic creation, you can set the default number of partitions for topics using `num.partitions`. Generally, however, this property is disabled so that more control is provided over topics through explicit topic creation.

For `data durability`, you should also set `min.insync.replicas` in your *topic* configuration and message delivery acknowledgments using `acks=all` in your *producer* configuration.

Use `replica.fetch.max.bytes` to set the maximum size, in bytes, of messages fetched by each follower that replicates the leader partition. Change this value according to the average message size and throughput. When considering the total memory allocation required for read/write buffering, the memory available must also be able to accommodate the maximum replicated message size when multiplied by all followers.

The `delete.topic.enable` property is enabled by default to allow topics to be deleted. In a production environment, you should disable this property to avoid accidental topic deletion, resulting in data loss. You can, however, temporarily enable it and delete topics and then disable it again.

**NOTE**

When running Strimzi on Kubernetes, the Topic Operator can provide operator-style topic management. You can use the `KafkaTopic` resource to create topics. For topics created using the `KafkaTopic` resource, the replication factor is set using `spec.replicas`. If `delete.topic.enable` is enabled, you can also delete topics using the `KafkaTopic` resource.

```
# ...
auto.create.topics.enable=false
delete.topic.enable=true
# ...
```

### 35.3.3. Internal topic settings for transactions and commits

If you are `using transactions` to enable atomic writes to partitions from producers, the state of the transactions is stored in the internal `_transaction_state` topic. By default, the brokers are configured with a replication factor of 3 and a minimum of 2 in-sync replicas for this topic, which means that a minimum of three brokers are required in your Kafka cluster.

```
# ...
transaction.state.log.replication.factor=3
transaction.state.log.min_isr=2
```

```
# ...
```

Similarly, the internal `_consumer_offsets` topic, which stores consumer state, has default settings for the number of partitions and replication factor.

```
# ...
offsets.topic.num.partitions=50
offsets.topic.replication.factor=3
# ...
```

**Do not reduce these settings in production.** You can increase the settings in a *production* environment. As an exception, you might want to reduce the settings in a single-broker *test* environment.

### 35.3.4. Improving request handling throughput by increasing I/O threads

Network threads handle requests to the Kafka cluster, such as produce and fetch requests from client applications. Produce requests are placed in a request queue. Responses are placed in a response queue.

The number of network threads per listener should reflect the replication factor and the levels of activity from client producers and consumers interacting with the Kafka cluster. If you are going to have a lot of requests, you can increase the number of threads, using the amount of time threads are idle to determine when to add more threads.

To reduce congestion and regulate the request traffic, you can limit the number of requests allowed in the request queue. When the request queue is full, all incoming traffic is blocked.

I/O threads pick up requests from the request queue to process them. Adding more threads can improve throughput, but the number of CPU cores and disk bandwidth imposes a practical upper limit. At a minimum, the number of I/O threads should equal the number of storage volumes.

```
# ...
num.network.threads=3 ①
queued.max.requests=500 ②
num.io.threads=8 ③
num.recovery.threads.per.data.dir=4 ④
# ...
```

① The number of network threads for the Kafka cluster.

② The number of requests allowed in the request queue.

③ The number of I/O threads for a Kafka broker.

④ The number of threads used for log loading at startup and flushing at shutdown. Try setting to a value of at least the number of cores.

Configuration updates to the thread pools for all brokers might occur dynamically at the cluster

level. These updates are restricted to between half the current size and twice the current size.

The following Kafka broker metrics can help with working out the number of threads required:

- `kafka.network:type=SocketServer, name=NetworkProcessorAvgIdlePercent` provides metrics on the average time network threads are idle as a percentage.
- `kafka.server:type=KafkaRequestHandlerPool, name=RequestHandlerAvgIdlePercent` provides metrics on the average time I/O threads are idle as a percentage.

If there is 0% idle time, all resources are in use, which means that adding more threads might be beneficial. When idle time goes below 30%, performance may start to suffer.

If threads are slow or limited due to the number of disks, you can try increasing the size of the buffers for network requests to improve throughput:

```
# ...
replica.socket.receive.buffer.bytes=65536
# ...
```

And also increase the maximum number of bytes Kafka can receive:

```
# ...
socket.request.max.bytes=104857600
# ...
```

### 35.3.5. Increasing bandwidth for high latency connections

Kafka batches data to achieve reasonable throughput over high-latency connections from Kafka to clients, such as connections between datacenters. However, if high latency is a problem, you can increase the size of the buffers for sending and receiving messages.

```
# ...
socket.send.buffer.bytes=1048576
socket.receive.buffer.bytes=1048576
# ...
```

You can estimate the optimal size of your buffers using a *bandwidth-delay product* calculation, which multiplies the maximum bandwidth of the link (in bytes/s) with the round-trip delay (in seconds) to give an estimate of how large a buffer is required to sustain maximum throughput.

### 35.3.6. Managing Kafka logs with delete and compact policies

Kafka relies on logs to store message data. A log consists of a series of segments, where each segment is associated with offset-based and timestamp-based indexes. New messages are written to an *active* segment and are never subsequently modified. When serving fetch requests from consumers, the segments are read. Periodically, the active segment is *rolled* to become read-only,

and a new active segment is created to replace it. There is only one active segment per topic-partition per broker. Older segments are retained until they become eligible for deletion.

Configuration at the broker level determines the maximum size in bytes of a log segment and the time in milliseconds before an active segment is rolled:

```
# ...
log.segment.bytes=1073741824
log.roll.ms=604800000
# ...
```

These settings can be overridden at the topic level using `segment.bytes` and `segment.ms`. The choice to lower or raise these values depends on the policy for segment deletion. A larger size means the active segment contains more messages and is rolled less often. Segments also become eligible for deletion less frequently.

In Kafka, log cleanup policies determine how log data is managed. In most cases, you won't need to change the default configuration at the cluster level, which specifies the `delete` cleanup policy and enables the log cleaner used by the `compact` cleanup policy:

```
# ...
log.cleanup.policy=delete
log.cleaner.enable=true
# ...
```

## Delete cleanup policy

Delete cleanup policy is the default cluster-wide policy for all topics. The policy is applied to topics that do not have a specific topic-level policy configured. Kafka removes older segments based on time-based or size-based log retention limits.

## Compact cleanup policy

Compact cleanup policy is generally configured as a topic-level policy (`cleanup.policy=compact`). Kafka's log cleaner applies compaction on specific topics, retaining only the most recent value for a key in the topic. You can also configure topics to use both policies (`cleanup.policy=compact,delete`).

### *Setting up retention limits for the delete policy*

Delete cleanup policy corresponds to managing logs with data retention. The policy is suitable when data does not need to be retained forever. You can establish time-based or size-based log retention and cleanup policies to keep logs bounded.

When log retention policies are employed, non-active log segments are removed when retention limits are reached. Deletion of old segments helps to prevent exceeding disk capacity.

For time-based log retention, you set a retention period based on hours, minutes, or milliseconds:

```
# ...
```

```
log.retention.ms=1680000  
# ...
```

The retention period is based on the time messages were appended to the segment. Kafka uses the timestamp of the latest message within a segment to determine if that segment has expired or not. The milliseconds configuration has priority over minutes, which has priority over hours. The minutes and milliseconds configurations are null by default, but the three options provide a substantial level of control over the data you wish to retain. Preference should be given to the milliseconds configuration, as it is the only one of the three properties that is dynamically updateable.

If `log.retention.ms` is set to -1, no time limit is applied to log retention, and all logs are retained. However, this setting is not generally recommended as it can lead to issues with full disks that are difficult to rectify.

For size-based log retention, you specify a minimum log size (in bytes):

```
# ...  
log.retention.bytes=1073741824  
# ...
```

This means that Kafka will ensure there is always at least the specified amount of log data available.

For example, if you set `log.retention.bytes` to 1000 and `log.segment.bytes` to 300, Kafka will keep 4 segments plus the active segment, ensuring a minimum of 1000 bytes are available. When the active segment becomes full and a new segment is created, the oldest segment is deleted. At this point, the size on disk may exceed the specified 1000 bytes, potentially ranging between 1200 and 1500 bytes (excluding index files).

A potential issue with using a log size is that it does not take into account the time messages were appended to a segment. You can use time-based and size-based log retention for your cleanup policy to get the balance you need. Whichever threshold is reached first triggers the cleanup.

To add a time delay before a segment file is deleted from the system, you can use `log.segment.delete.delay.ms` at the broker level for all topics:

```
# ...  
log.segment.delete.delay.ms=60000  
# ...
```

Or configure `file.delete.delay.ms` at the topic level.

You set the frequency at which the log is checked for cleanup in milliseconds:

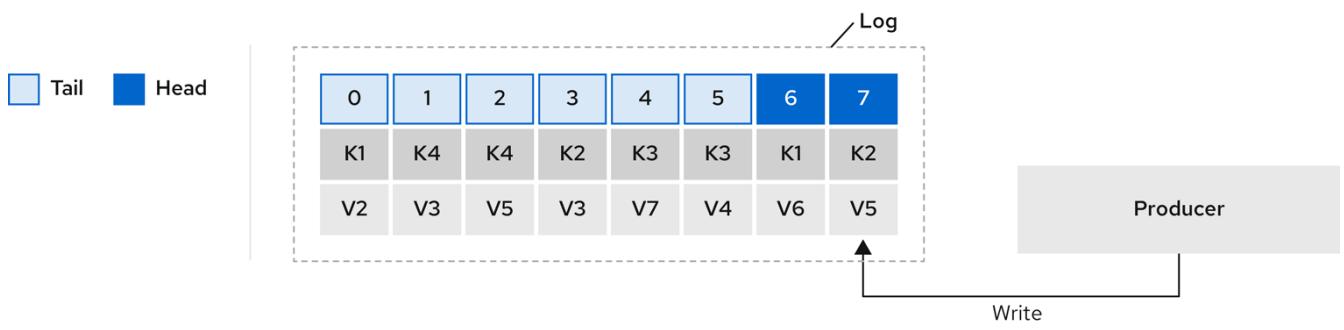
```
# ...  
log.retention.check.interval.ms=300000
```

Adjust the log retention check interval in relation to the log retention settings. Smaller retention sizes might require more frequent checks. The frequency of cleanup should be often enough to manage the disk space but not so often it affects performance on a broker.

#### *Retaining the most recent messages using compact policy*

When you enable log compaction for a topic by setting `cleanup.policy=compact`, Kafka uses the log cleaner as a background thread to perform the compaction. The compact policy guarantees that the most recent message for each message key is retained, effectively cleaning up older versions of records. The policy is suitable when message values are changeable, and you want to retain the latest update.

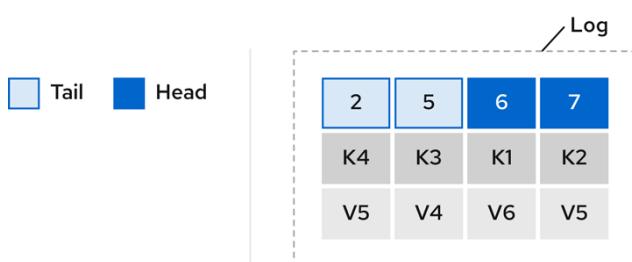
If a cleanup policy is set for log compaction, the *head* of the log operates as a standard Kafka log, with writes for new messages appended in order. In the *tail* of a compacted log, where the log cleaner operates, records are deleted if another record with the same key occurs later in the log. Messages with null values are also deleted. To use compaction, you must have keys to identify related messages because Kafka guarantees that the latest messages for each key will be retained, but it does not guarantee that the whole compacted log will not contain duplicates.



212\_Streams\_0322

Figure 7. Log showing key value writes with offset positions before compaction

Using keys to identify messages, Kafka compaction keeps the latest message (with the highest offset) that is present in the log tail for a specific message key, eventually discarding earlier messages that have the same key. The message in its latest state is always available, and any out-of-date records of that particular message are eventually removed when the log cleaner runs. You can restore a message back to a previous state. Records retain their original offsets even when surrounding records get deleted. Consequently, the tail can have non-contiguous offsets. When consuming an offset that's no longer available in the tail, the record with the next higher offset is found.



212\_Streams\_0322

Figure 8. Log after compaction

If appropriate, you can add a delay to the compaction process:

```
# ...
log.cleaner.delete.retention.ms=86400000
# ...
```

The deleted data retention period gives time to notice the data is gone before it is irretrievably deleted.

To delete all messages related to a specific key, a producer can send a *tombstone* message. A tombstone has a null value and acts as a marker to inform consumers that the corresponding message for that key has been deleted. After some time, only the tombstone marker is retained. Assuming new messages continue to come in, the marker is retained for a duration specified by `log.cleaner.delete.retention.ms` to allow consumers enough time to recognize the deletion.

You can also set a time in milliseconds to put the cleaner on standby if there are no logs to clean:

```
# ...
log.cleaner.backoff.ms=15000
# ...
```

#### *Using combined compact and delete policies*

If you choose only a compact policy, your log can still become arbitrarily large. In such cases, you can set the cleanup policy for a topic to compact and delete logs. Kafka applies log compaction, removing older versions of records and retaining only the latest version of each key. Kafka also deletes records based on the specified time-based or size-based log retention settings.

For example, in the following diagram only the latest message (with the highest offset) for a specific message key is retained up to the compaction point. If there are any records remaining up to the retention point they are deleted. In this case, the compaction process would remove all duplicates.

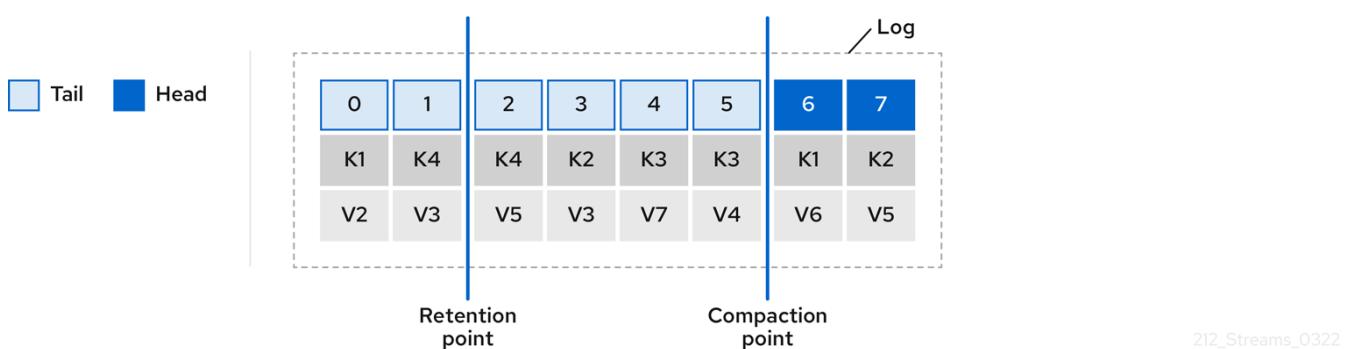


Figure 9. Log retention point and compaction point

### 35.3.7. Managing efficient disk utilization for compaction

When employing the compact policy and log cleaner to handle topic logs in Kafka, consider optimizing memory allocation.

You can fine-tune memory allocation using the deduplication property (`dedupe.buffer.size`), which determines the total memory allocated for cleanup tasks across all log cleaner threads. Additionally, you can establish a maximum memory usage limit by defining a percentage through the `buffer.load.factor` property.

```
# ...
log.cleaner.dedupe.buffer.size=134217728
log.cleaner.io.buffer.load.factor=0.9
# ...
```

Each log entry uses exactly 24 bytes, so you can work out how many log entries the buffer can handle in a single run and adjust the setting accordingly.

If possible, consider increasing the number of log cleaner threads if you are looking to reduce the log cleaning time:

```
# ...
log.cleaner.threads=8
# ...
```

If you are experiencing issues with 100% disk bandwidth usage, you can throttle the log cleaner I/O so that the sum of the read/write operations is less than a specified double value based on the capabilities of the disks performing the operations:

```
# ...
log.cleaner.io.max.bytes.per.second=1.7976931348623157E308
# ...
```

### 35.3.8. Controlling the log flush of message data

Generally, the recommendation is to not set explicit flush thresholds and let the operating system perform background flush using its default settings. Partition replication provides greater data durability than writes to any single disk, as a failed broker can recover from its in-sync replicas.

Log flush properties control the periodic writes of cached message data to disk. The scheduler specifies the frequency of checks on the log cache in milliseconds:

```
# ...
log.flush.scheduler.interval.ms=2000
# ...
```

You can control the frequency of the flush based on the maximum amount of time that a message is kept in-memory and the maximum number of messages in the log before writing to disk:

```
# ...
```

```
log.flush.interval.ms=50000
log.flush.interval.messages=100000
# ...
```

The wait between flushes includes the time to make the check and the specified interval before the flush is carried out. Increasing the frequency of flushes can affect throughput.

If you are using application flush management, setting lower flush thresholds might be appropriate if you are using faster disks.

### 35.3.9. Partition rebalancing for availability

Partitions can be replicated across brokers for fault tolerance. For a given partition, one broker is elected leader and handles all produce requests (writes to the log). Partition followers on other brokers replicate the partition data of the partition leader for data reliability in the event of the leader failing.

Followers do not normally serve clients, though `rack` configuration allows a consumer to consume messages from the closest replica when a Kafka cluster spans multiple datacenters. Followers operate only to replicate messages from the partition leader and allow recovery should the leader fail. Recovery requires an in-sync follower. Followers stay in sync by sending fetch requests to the leader, which returns messages to the follower in order. The follower is considered to be in sync if it has caught up with the most recently committed message on the leader. The leader checks this by looking at the last offset requested by the follower. An out-of-sync follower is usually not eligible as a leader should the current leader fail, unless [unclean leader election is allowed](#).

You can adjust the lag time before a follower is considered out of sync:

```
# ...
replica.lag.time.max.ms=30000
# ...
```

Lag time puts an upper limit on the time to replicate a message to all in-sync replicas and how long a producer has to wait for an acknowledgment. If a follower fails to make a fetch request and catch up with the latest message within the specified lag time, it is removed from in-sync replicas. You can reduce the lag time to detect failed replicas sooner, but by doing so you might increase the number of followers that fall out of sync needlessly. The right lag time value depends on both network latency and broker disk bandwidth.

When a leader partition is no longer available, one of the in-sync replicas is chosen as the new leader. The first broker in a partition's list of replicas is known as the *preferred* leader. By default, Kafka is enabled for automatic partition leader rebalancing based on a periodic check of leader distribution. That is, Kafka checks to see if the preferred leader is the *current* leader. A rebalance ensures that leaders are evenly distributed across brokers and brokers are not overloaded.

You can use Cruise Control for Strimzi to figure out replica assignments to brokers that balance load evenly across the cluster. Its calculation takes into account the differing load experienced by leaders and followers. A failed leader affects the balance of a Kafka cluster because the remaining

brokers get the extra work of leading additional partitions.

For the assignment found by Cruise Control to actually be balanced it is necessary that partitions are lead by the preferred leader. Kafka can automatically ensure that the preferred leader is being used (where possible), changing the current leader if necessary. This ensures that the cluster remains in the balanced state found by Cruise Control.

You can control the frequency, in seconds, of the rebalance check and the maximum percentage of imbalance allowed for a broker before a rebalance is triggered.

```
#...
auto.leader.rebalance.enable=true
leader.imbalance.check.interval.seconds=300
leader.imbalance.per.broker.percentage=10
#...
```

The percentage leader imbalance for a broker is the ratio between the current number of partitions for which the broker is the current leader and the number of partitions for which it is the preferred leader. You can set the percentage to zero to ensure that preferred leaders are always elected, assuming they are in sync.

If the checks for rebalances need more control, you can disable automated rebalances. You can then choose when to trigger a rebalance using the [kafka-leader-election.sh](#) command line tool.

**NOTE**

The Grafana dashboards provided with Strimzi show metrics for under-replicated partitions and partitions that do not have an active leader.

### 35.3.10. Unclean leader election

Leader election to an in-sync replica is considered clean because it guarantees no loss of data. And this is what happens by default. But what if there is no in-sync replica to take on leadership? Perhaps the ISR (in-sync replica) only contained the leader when the leader's disk died. If a minimum number of in-sync replicas is not set, and there are no followers in sync with the partition leader when its hard drive fails irrevocably, data is already lost. Not only that, but *a new leader cannot be elected* because there are no in-sync followers.

You can configure how Kafka handles leader failure:

```
# ...
unclean.leader.election.enable=false
# ...
```

Unclean leader election is disabled by default, which means that out-of-sync replicas cannot become leaders. With clean leader election, if no other broker was in the ISR when the old leader was lost, Kafka waits until that leader is back online before messages can be written or read. Unclean leader election means out-of-sync replicas can become leaders, but you risk losing messages. The choice you make depends on whether your requirements favor availability or

durability.

You can override the default configuration for specific topics at the topic level. If you cannot afford the risk of data loss, then leave the default configuration.

### 35.3.11. Avoiding unnecessary consumer group rebalances

For consumers joining a new consumer group, you can add a delay so that unnecessary rebalances to the broker are avoided:

```
# ...
group.initial.rebalance.delay.ms=3000
# ...
```

The delay is the amount of time that the coordinator waits for members to join. The longer the delay, the more likely it is that all the members will join in time and avoid a rebalance. But the delay also prevents the group from consuming until the period has ended.

## 35.4. Kafka producer configuration tuning

Use configuration properties to optimize the performance of Kafka producers. You can use standard Kafka producer configuration options. Adjusting your configuration to maximize throughput might increase latency or vice versa. You will need to experiment and tune your producer configuration to get the balance you need.

When configuring a producer, consider the following aspects carefully, as they significantly impact its performance and behavior:

### Compression

By compressing messages before they are sent over the network, you can conserve network bandwidth and reduce disk storage requirements, but with the additional cost of increased CPU utilization due to the compression and decompression processes.

### Batching

Adjusting the batch size and time intervals when the producer sends messages can affect throughput and latency.

### Partitioning

Partitioning strategies in the Kafka cluster can support producers through parallelism and load balancing, whereby producers can write to multiple partitions concurrently and each partition receives an equal share of messages. Other strategies might include topic replication for fault tolerance.

### Securing access

Implement security measures for authentication, encryption, and authorization by setting up user accounts to [manage secure access to Kafka](#).

### 35.4.1. Basic producer configuration

Connection and serializer properties are required for every producer. Generally, it is good practice to add a client id for tracking, and use compression on the producer to reduce batch sizes in requests.

In a basic producer configuration:

- The order of messages in a partition is not guaranteed.
- The acknowledgment of messages reaching the broker does not guarantee durability.

*Basic producer configuration properties*

```
# ...
bootstrap.servers=localhost:9092 ①
key.serializer=org.apache.kafka.common.serialization.StringSerializer ②
value.serializer=org.apache.kafka.common.serialization.StringSerializer ③
client.id=my-client ④
compression.type=gzip ⑤
# ...
```

① (Required) Tells the producer to connect to a Kafka cluster using a *host:port* bootstrap server address for a Kafka broker. The producer uses the address to discover and connect to all brokers in the cluster. Use a comma-separated list to specify two or three addresses in case a server is down, but it's not necessary to provide a list of all the brokers in the cluster.

② (Required) Serializer to transform the key of each message to bytes prior to them being sent to a broker.

③ (Required) Serializer to transform the value of each message to bytes prior to them being sent to a broker.

④ (Optional) The logical name for the client, which is used in logs and metrics to identify the source of a request.

⑤ (Optional) The codec for compressing messages, which are sent and might be stored in compressed format and then decompressed when reaching a consumer. Compression is useful for improving throughput and reducing the load on storage, but might not be suitable for low latency applications where the cost of compression or decompression could be prohibitive.

### 35.4.2. Data durability

Message delivery acknowledgments minimize the likelihood that messages are lost. By default, acknowledgments are enabled with the `acks` property set to `acks=all`. To control the maximum time the producer waits for acknowledgments from the broker and handle potential delays in sending messages, you can use the `delivery.timeout.ms` property.

*Acknowledging message delivery*

```
# ...
acks=all ①
delivery.timeout.ms=120000 ②
```

```
# ...
```

① `acks=all` forces a leader replica to replicate messages to a certain number of followers before acknowledging that the message request was successfully received.

② The maximum time in milliseconds to wait for a complete send request. You can set the value to `MAX_LONG` to delegate to Kafka an indefinite number of retries. The default is `120000` or 2 minutes.

The `acks=all` setting offers the strongest guarantee of delivery, but it will increase the latency between the producer sending a message and receiving acknowledgment. If you don't require such strong guarantees, a setting of `acks=0` or `acks=1` provides either no delivery guarantees or only acknowledgment that the leader replica has written the record to its log.

With `acks=all`, the leader waits for all in-sync replicas to acknowledge message delivery. A topic's `min.insync.replicas` configuration sets the minimum required number of in-sync replica acknowledgements. The number of acknowledgements include that of the leader and followers.

A typical starting point is to use the following configuration:

- Producer configuration:
  - `acks=all` (default)
- Broker configuration for topic replication:
  - `default.replication.factor=3` (default = 1)
  - `min.insync.replicas=2` (default = 1)

When you create a topic, you can override the default replication factor. You can also override `min.insync.replicas` at the topic level in the topic configuration.

Strimzi uses this configuration in the example configuration files for multi-node deployment of Kafka.

The following table describes how this configuration operates depending on the availability of followers that replicate the leader replica.

*Table 41. Follower availability*

Number of followers available and in-sync	Acknowledgements	Producer can send messages?
2	The leader waits for 2 follower acknowledgements	Yes
1	The leader waits for 1 follower acknowledgement	Yes
0	The leader raises an exception	No

A topic replication factor of 3 creates one leader replica and two followers. In this configuration, the producer can continue if a single follower is unavailable. Some delay can occur whilst removing a failed broker from the in-sync replicas or a creating a new leader. If the second follower is also unavailable, message delivery will not be successful. Instead of acknowledging successful message

delivery, the leader sends an error (*not enough replicas*) to the producer. The producer raises an equivalent exception. With `retries` configuration, the producer can resend the failed message request.

**NOTE** If the system fails, there is a risk of unsent data in the buffer being lost.

### 35.4.3. Ordered delivery

Idempotent producers avoid duplicates as messages are delivered exactly once. IDs and sequence numbers are assigned to messages to ensure the order of delivery, even in the event of failure. If you are using `acks=all` for data consistency, using idempotency makes sense for ordered delivery. Idempotency is enabled for producers by default. With idempotency enabled, you can set the number of concurrent in-flight requests to a maximum of 5 for message ordering to be preserved.

*Ordered delivery with idempotency*

```
# ...
enable.idempotence=true ①
max.in.flight.requests.per.connection=5 ②
acks=all ③
retries=2147483647 ④
# ...
```

① Set to `true` to enable the idempotent producer.

② With idempotent delivery the number of in-flight requests may be greater than 1 while still providing the message ordering guarantee. The default is 5 in-flight requests.

③ Set `acks` to `all`.

④ Set the number of attempts to resend a failed message request.

If you choose not to use `acks=all` and disable idempotency because of the performance cost, set the number of in-flight (unacknowledged) requests to 1 to preserve ordering. Otherwise, a situation is possible where *Message-A* fails only to succeed after *Message-B* was already written to the broker.

*Ordered delivery without idempotency*

```
# ...
enable.idempotence=false ①
max.in.flight.requests.per.connection=1 ②
retries=2147483647
# ...
```

① Set to `false` to disable the idempotent producer.

② Set the number of in-flight requests to exactly 1.

### 35.4.4. Reliability guarantees

Idempotence is useful for exactly once writes to a single partition. Transactions, when used with idempotence, allow exactly once writes across multiple partitions.

Transactions guarantee that messages using the same transactional ID are produced once, and either *all* are successfully written to the respective logs or *none* of them are.

```
# ...
enable.idempotence=true
max.in.flight.requests.per.connection=5
acks=all
retries=2147483647
transactional.id=UNIQUE-ID ①
transaction.timeout.ms=900000 ②
# ...
```

① Specify a unique transactional ID.

② Set the maximum allowed time for transactions in milliseconds before a timeout error is returned. The default is **900000** or 15 minutes.

The choice of `transactional.id` is important in order that the transactional guarantee is maintained. Each transactional id should be used for a unique set of topic partitions. For example, this can be achieved using an external mapping of topic partition names to transactional ids, or by computing the transactional id from the topic partition names using a function that avoids collisions.

### 35.4.5. Optimizing producers for throughput and latency

Usually, the requirement of a system is to satisfy a particular throughput target for a proportion of messages within a given latency. For example, targeting 500,000 messages per second with 95% of messages being acknowledged within 2 seconds.

It's likely that the messaging semantics (message ordering and durability) of your producer are defined by the requirements for your application. For instance, it's possible that you don't have the option of using `acks=0` or `acks=1` without breaking some important property or guarantee provided by your application.

Broker restarts have a significant impact on high percentile statistics. For example, over a long period the 99th percentile latency is dominated by behavior around broker restarts. This is worth considering when designing benchmarks or comparing performance numbers from benchmarking with performance numbers seen in production.

Depending on your objective, Kafka offers a number of configuration parameters and techniques for tuning producer performance for throughput and latency.

#### Message batching (`linger.ms` and `batch.size`)

Message batching delays sending messages in the hope that more messages destined for the same broker will be sent, allowing them to be batched into a single produce request. Batching is a compromise between higher latency in return for higher throughput. Time-based batching is configured using `linger.ms`, and size-based batching is configured using `batch.size`.

#### Compression (`compression.type`)

Message compression adds latency in the producer (CPU time spent compressing the messages),

but makes requests (and potentially disk writes) smaller, which can increase throughput. Whether compression is worthwhile, and the best compression to use, will depend on the messages being sent. Compression happens on the thread which calls `KafkaProducer.send()`, so if the latency of this method matters for your application you should consider using more threads.

## Pipelining (`max.in.flight.requests.per.connection`)

Pipelining means sending more requests before the response to a previous request has been received. In general more pipelining means better throughput, up to a threshold at which other effects, such as worse batching, start to counteract the effect on throughput.

### *Lowering latency*

When your application calls the `KafkaProducer.send()` method, messages undergo a series of operations before being sent:

- Interception: Messages are processed by any configured interceptors.
- Serialization: Messages are serialized into the appropriate format.
- Partition assignment: Each message is assigned to a specific partition.
- Compression: Messages are compressed to conserve network bandwidth.
- Batching: Compressed messages are added to a batch in a partition-specific queue.

During these operations, the `send()` method is momentarily blocked. It also remains blocked if the `buffer.memory` is full or if metadata is unavailable.

Batches will remain in the queue until one of the following occurs:

- The batch is full (according to `batch.size`).
- The delay introduced by `linger.ms` has passed.
- The sender is ready to dispatch batches for other partitions to the same broker and can include this batch.
- The producer is being flushed or closed.

To minimize the impact of `send()` blocking on latency, optimize batching and buffering configurations. Use the `linger.ms` and `batch.size` properties to batch more messages into a single produce request for higher throughput.

```
# ...
linger.ms=100 ①
batch.size=16384 ②
buffer.memory=33554432 ③
# ...
```

① The `linger.ms` property adds a delay in milliseconds so that larger batches of messages are accumulated and sent in a request. The default is `0`.

② If a maximum `batch.size` in bytes is used, a request is sent when the maximum is reached, or messages have been queued for longer than `linger.ms` (whichever comes sooner). Adding the delay allows batches to accumulate messages up to the batch size.

- ③ The buffer size must be at least as big as the batch size, and be able to accommodate buffering, compression, and in-flight requests.

#### *Increasing throughput*

You can improve throughput of your message requests by directing messages to a specified partition using a custom partitioner to replace the default.

```
# ...
partitioner.class=my-custom-partitioner ①

# ...
```

- ① Specify the class name of your custom partitioner.

## 35.5. Kafka consumer configuration tuning

Use configuration properties to optimize the performance of Kafka consumers. When tuning your consumers your primary concern will be ensuring that they cope efficiently with the amount of data ingested. As with the producer tuning, be prepared to make incremental changes until the consumers operate as expected.

When tuning a consumer, consider the following aspects carefully, as they significantly impact its performance and behavior:

### Scaling

Consumer groups enable parallel processing of messages by distributing the load across multiple consumers, enhancing scalability and throughput. The number of topic partitions determines the maximum level of parallelism that you can achieve, as one partition can only be assigned to one consumer in a consumer group.

### Message ordering

If absolute ordering within a topic is important, use a single-partition topic. A consumer observes messages in a single partition in the same order that they were committed to the broker, which means that Kafka only provides ordering guarantees for messages in a single partition. It is also possible to maintain message ordering for events specific to individual entities, such as users. If a new entity is created, you can create a new topic dedicated to that entity. You can use a unique ID, like a user ID, as the message key and route all messages with the same key to a single partition within the topic.

### Offset reset policy

Setting the appropriate offset policy ensures that the consumer consumes messages from the desired starting point and handles message processing accordingly. The default Kafka reset value is `latest`, which starts at the end of the partition, and consequently means some messages might be missed, depending on the consumer's behavior and the state of the partition. Setting `auto.offset.reset` to `earliest` ensures that when connecting with a new `group.id`, all messages are retrieved from the beginning of the log.

## Securing access

Implement security measures for authentication, encryption, and authorization by setting up user accounts to [manage secure access to Kafka](#).

### 35.5.1. Basic consumer configuration

Connection and deserializer properties are required for every consumer. Generally, it is good practice to add a client id for tracking.

In a consumer configuration, irrespective of any subsequent configuration:

- The consumer fetches from a given offset and consumes the messages in order, unless the offset is changed to skip or re-read messages.
- The broker does not know if the consumer processed the responses, even when committing offsets to Kafka, because the offsets might be sent to a different broker in the cluster.

#### *Basic consumer configuration properties*

```
# ...
bootstrap.servers=localhost:9092 ①
key.deserializer=org.apache.kafka.common.serialization.StringDeserializer ②
value.deserializer=org.apache.kafka.common.serialization.StringDeserializer ③
client.id=my-client ④
group.id=my-group-id ⑤
# ...
```

① (Required) Tells the consumer to connect to a Kafka cluster using a *host:port* bootstrap server address for a Kafka broker. The consumer uses the address to discover and connect to all brokers in the cluster. Use a comma-separated list to specify two or three addresses in case a server is down, but it is not necessary to provide a list of all the brokers in the cluster. If you are using a loadbalancer service to expose the Kafka cluster, you only need the address for the service because the availability is handled by the loadbalancer.

② (Required) Deserializer to transform the bytes fetched from the Kafka broker into message keys.

③ (Required) Deserializer to transform the bytes fetched from the Kafka broker into message values.

④ (Optional) The logical name for the client, which is used in logs and metrics to identify the source of a request. The id can also be used to throttle consumers based on processing time quotas.

⑤ (Conditional) A group id is *required* for a consumer to be able to join a consumer group.

### 35.5.2. Scaling data consumption using consumer groups

Consumer groups share a typically large data stream generated by one or multiple producers from a given topic. Consumers are grouped using a **group.id** property, allowing messages to be spread across the members. One of the consumers in the group is elected leader and decides how the partitions are assigned to the consumers in the group. Each partition can only be assigned to a single consumer.

If you do not already have as many consumers as partitions, you can scale data consumption by adding more consumer instances with the same `group.id`. Adding more consumers to a group than there are partitions will not help throughput, but it does mean that there are consumers on standby should one stop functioning. If you can meet throughput goals with fewer consumers, you save on resources.

Consumers within the same consumer group send offset commits and heartbeats to the same broker. The consumer sends heartbeats to the Kafka broker to indicate its activity within the consumer group. So the greater the number of consumers in the group, the higher the request load on the broker.

```
# ...
group.id=my-group-id ①
# ...
```

① Add a consumer to a consumer group using a group id.

### 35.5.3. Choosing the right partition assignment strategy

Select an appropriate partition assignment strategy, which determines how Kafka topic partitions are distributed among consumer instances in a group.

Partition strategies are supported by the following classes:

- `org.apache.kafka.clients.consumer.RangeAssignor`
- `org.apache.kafka.clients.consumer.RoundRobinAssignor`
- `org.apache.kafka.clients.consumer.StickyAssignor`
- `org.apache.kafka.clients.consumer.CooperativeStickyAssignor`

Specify a class using the `partition.assignment.strategy` consumer configuration property. The **range** assignment strategy assigns a range of partitions to each consumer, and is useful when you want to process related data together.

Alternatively, opt for a **round robin** assignment strategy for equal partition distribution among consumers, which is ideal for high-throughput scenarios requiring parallel processing.

For more stable partition assignments, consider the **sticky** and **cooperative sticky** strategies. Sticky strategies aim to maintain assigned partitions during rebalances, when possible. If a consumer was previously assigned certain partitions, the sticky strategies prioritize retaining those same partitions with the same consumer after a rebalance, while only revoking and reassigning the partitions that are actually moved to another consumer. Leaving partition assignments in place reduces the overhead on partition movements. The cooperative sticky strategy also supports cooperative rebalances, enabling uninterrupted consumption from partitions that are not reassigned.

If none of the available strategies suit your data, you can create a custom strategy tailored to your specific requirements.

### 35.5.4. Message ordering guarantees

Kafka brokers receive fetch requests from consumers that ask the broker to send messages from a list of topics, partitions and offset positions.

A consumer observes messages in a single partition in the same order that they were committed to the broker, which means that Kafka **only** provides ordering guarantees for messages in a single partition. Conversely, if a consumer is consuming messages from multiple partitions, the order of messages in different partitions as observed by the consumer does not necessarily reflect the order in which they were sent.

If you want a strict ordering of messages from one topic, use one partition per consumer.

### 35.5.5. Optimizing consumers for throughput and latency

Control the number of messages returned when your client application calls `KafkaConsumer.poll()`.

Use the `fetch.max.wait.ms` and `fetch.min.bytes` properties to increase the minimum amount of data fetched by the consumer from the Kafka broker. Time-based batching is configured using `fetch.max.wait.ms`, and size-based batching is configured using `fetch.min.bytes`.

If CPU utilization in the consumer or broker is high, it might be because there are too many requests from the consumer. You can adjust `fetch.max.wait.ms` and `fetch.min.bytes` properties higher so that there are fewer requests and messages are delivered in bigger batches. By adjusting higher, throughput is improved with some cost to latency. You can also adjust higher if the amount of data being produced is low.

For example, if you set `fetch.max.wait.ms` to 500ms and `fetch.min.bytes` to 16384 bytes, when Kafka receives a fetch request from the consumer it will respond when the first of either threshold is reached.

Conversely, you can adjust the `fetch.max.wait.ms` and `fetch.min.bytes` properties lower to improve end-to-end latency.

```
# ...
fetch.max.wait.ms=500 ①
fetch.min.bytes=16384 ②
# ...
```

① The maximum time in milliseconds the broker will wait before completing fetch requests. The default is **500** milliseconds.

② If a minimum batch size in bytes is used, a request is sent when the minimum is reached, or messages have been queued for longer than `fetch.max.wait.ms` (whichever comes sooner). Adding the delay allows batches to accumulate messages up to the batch size.

*Lowering latency by increasing the fetch request size*

Use the `fetch.max.bytes` and `max.partition.fetch.bytes` properties to increase the maximum amount of data fetched by the consumer from the Kafka broker.

The `fetch.max.bytes` property sets a maximum limit in bytes on the amount of data fetched from the broker at one time.

The `max.partition.fetch.bytes` sets a maximum limit in bytes on how much data is returned for each partition, which must always be larger than the number of bytes set in the broker or topic configuration for `max.message.bytes`.

The maximum amount of memory a client can consume is calculated approximately as:

```
NUMBER-OF-BROKERS * fetch.max.bytes and NUMBER-OF-PARTITIONS *  
max.partition.fetch.bytes
```

If memory usage can accommodate it, you can increase the values of these two properties. By allowing more data in each request, latency is improved as there are fewer fetch requests.

```
# ...  
fetch.max.bytes=52428800 ①  
max.partition.fetch.bytes=1048576 ②  
# ...
```

① The maximum amount of data in bytes returned for a fetch request.

② The maximum amount of data in bytes returned for each partition.

### 35.5.6. Avoiding data loss or duplication when committing offsets

The Kafka *auto-commit mechanism* allows a consumer to commit the offsets of messages automatically. If enabled, the consumer will commit offsets received from polling the broker at 5000ms intervals.

The auto-commit mechanism is convenient, but it introduces a risk of data loss and duplication. If a consumer has fetched and transformed a number of messages, but the system crashes with processed messages in the consumer buffer when performing an auto-commit, that data is lost. If the system crashes after processing the messages, but before performing the auto-commit, the data is duplicated on another consumer instance after rebalancing.

Auto-committing can avoid data loss only when all messages are processed before the next poll to the broker, or the consumer closes.

To minimize the likelihood of data loss or duplication, you can set `enable.auto.commit` to `false` and develop your client application to have more control over committing offsets. Or you can use `auto.commit.interval.ms` to decrease the intervals between commits.

```
# ...  
enable.auto.commit=false ①  
# ...
```

① Auto commit is set to false to provide more control over committing offsets.

By setting to `enable.auto.commit` to `false`, you can commit offsets after **all** processing has been performed and the message has been consumed. For example, you can set up your application to call the Kafka `commitSync` and `commitAsync` commit APIs.

The `commitSync` API commits the offsets in a message batch returned from polling. You call the API when you are finished processing all the messages in the batch. If you use the `commitSync` API, the application will not poll for new messages until the last offset in the batch is committed. If this negatively affects throughput, you can commit less frequently, or you can use the `commitAsync` API. The `commitAsync` API does not wait for the broker to respond to a commit request, but risks creating more duplicates when rebalancing. A common approach is to combine both commit APIs in an application, with the `commitSync` API used just before shutting the consumer down or rebalancing to make sure the final commit is successful.

### Controlling transactional messages

Consider using transactional ids and enabling idempotence (`enable.idempotence=true`) on the producer side to guarantee exactly-once delivery. On the consumer side, you can then use the `isolation.level` property to control how transactional messages are read by the consumer.

The `isolation.level` property has two valid values:

- `read_committed`
- `read_uncommitted` (default)

Use `read_committed` to ensure that only transactional messages that have been committed are read by the consumer. However, this will cause an increase in end-to-end latency, because the consumer will not be able to return a message until the brokers have written the transaction markers that record the result of the transaction (*committed* or *aborted*).

```
# ...
enable.auto.commit=false
isolation.level=read_committed ①
# ...
```

① Set to `read_committed` so that only committed messages are read by the consumer.

### 35.5.7. Recovering from failure to avoid data loss

In the event of failures within a consumer group, Kafka provides a rebalance protocol designed for effective detection and recovery. To minimize the potential impact of these failures, one key strategy is to adjust the `max.poll.records` property to balance efficient processing with system stability. This property determines the maximum number of records a consumer can fetch in a single poll. Fine-tuning `max.poll.records` helps to maintain a controlled consumption rate, preventing the consumer from overwhelming itself or the Kafka broker.

Additionally, Kafka offers advanced configuration properties like `session.timeout.ms` and `heartbeat.interval.ms`. These settings are typically reserved for more specialized use cases and may not require adjustment in standard scenarios.

The `session.timeout.ms` property specifies the maximum amount of time a consumer can go without sending a heartbeat to the Kafka broker to indicate it is active within the consumer group. If a consumer fails to send a heartbeat within the session timeout, it is considered inactive. A consumer marked as inactive triggers a rebalancing of the partitions for the topic. Setting the `session.timeout.ms` property value too low can result in false-positive outcomes, while setting it too high can lead to delayed recovery from failures.

The `heartbeat.interval.ms` property determines how frequently a consumer sends heartbeats to the Kafka broker. A shorter interval between consecutive heartbeats allows for quicker detection of consumer failures. The heartbeat interval must be lower, usually by a third, than the session timeout. Decreasing the heartbeat interval reduces the chance of accidental rebalancing, but more frequent heartbeats increases the overhead on broker resources.

```
# ...
max.poll.records=100 ①
session.timeout.ms=30000 ②
heartbeat.interval.ms=5000 ③
# ...
```

- ① Set the number records returned to the consumer when calling the `poll()` method.
- ② Set the timeout for detecting client failure. If the broker configuration has a `group.min.session.timeout.ms` and `group.max.session.timeout.ms`, the session timeout value must be within that range.
- ③ Adjust the heartbeat interval according to anticipated rebalances.

### 35.5.8. Managing offset policy

Use the `auto.offset.reset` property to control how a consumer behaves when no offsets have been committed, or a committed offset is no longer valid or deleted.

Suppose you deploy a consumer application for the first time, and it reads messages from an existing topic. Because this is the first time the `group.id` is used, the `__consumer_offsets` topic does not contain any offset information for this application. The new application can start processing all existing messages from the start of the log or only new messages. The default reset value is `latest`, which starts at the end of the partition, and consequently means some messages are missed. To avoid data loss, but increase the amount of processing, set `auto.offset.reset` to `earliest` to start at the beginning of the partition.

Also consider using the `earliest` option to avoid messages being lost when the offsets retention period (`offsets.retention.minutes`) configured for a broker has ended. If a consumer group or standalone consumer is inactive and commits no offsets during the retention period, previously committed offsets are deleted from `__consumer_offsets`.

```
# ...
auto.offset.reset=earliest ①
# ...
```

- ① Set to `earliest` to return to the start of a partition and avoid data loss if offsets were not committed.

If the amount of data returned in a single fetch request is large, a timeout might occur before the consumer has processed it. In this case, you can lower `max.partition.fetch.bytes` or increase `session.timeout.ms`.

### 35.5.9. Minimizing the impact of rebalances

Rebalances in Kafka consumer groups can introduce latency and reduce throughput, impacting overall service performance. The rebalancing of a partition between active consumers in a group is the time it takes for the following to take place:

- Consumers to commit their offsets
- The new consumer group to be formed
- The group leader to assign partitions to group members
- The consumers in the group to receive their assignments and start fetching

Rebalances are triggered by changes in consumer health, network issues, configuration updates, and scaling events. This process can increase service downtime, especially if it occurs frequently, such as during rolling restarts of consumers in a group.

To minimize the impact of rebalances, consider the following strategies and configurations:

#### Assess throughput and parallelism

Assess the expected throughput (bytes and records per second) and parallelism (number of partitions) of the input topics against the number of consumers.

If adjustments are needed, start by setting up static membership, adopting a partition assignment strategy, and setting a limit on the number of records returned using the `max.poll.records` property. Add further configurations for timeouts and intervals, if required and with care, as these can introduce issues related to the handling of failures.

#### Use static membership

Assign a unique identifier (`group.instance.id`) to each consumer instance. Static membership introduces persistence so static consumers retain partition assignments across restarts, reducing unnecessary rebalances.

#### Adopt partition assignment strategies

- Use appropriate partition assignment strategies to reduce the number of partitions that need to be reassigned during a rebalance, minimizing the impact on active consumers.
- The `org.apache.kafka.clients.consumer.CooperativeStickyAssignor` strategy is particularly effective, as it ensures minimal partition movement and better stability during rebalances.

#### Adjust record limits and poll intervals

- Use the `max.poll.records` property to limit the number of records returned during each poll. Processing fewer messages more efficiently can prevent delays.
- Use the `max.poll.interval.ms` property to prevent rebalances caused by prolonged processing

tasks by setting the maximum interval between calls to the `poll()` method.

- Alternatively, consider pausing partitions to retrieve fewer records at a time.

## Adjust session timeout and heartbeat intervals

- Use the `session.timeout.ms` property to set a longer timeout to reduce rebalances caused by temporary network glitches or minor processing delays.
- Adjust the `heartbeat.interval.ms` property to balance failure detection checks with minimizing unnecessary rebalances.

## Monitor consumer health

Instability in consumer applications, such as frequent crashes, can trigger rebalances. Use Kafka consumer metrics to monitor such things as rebalance rates, session timeouts, and failed fetch requests.

### *Example configuration to minimize the impact of rebalances*

```
# ...
group.instance.id=<unique_id>
max.poll.interval.ms=300000
max.poll.records=500
session.timeout.ms=30000
heartbeat.interval.ms=5000
partition.assignment.strategy=org.apache.kafka.clients.consumer.CooperativeStickyAssig
nor
# ...
```

## Scaling strategies

To minimize the impact of rebalances during scaling of consumer groups, consider the following approaches:

### Set a rebalance delay

Use the `group.initial.rebalance.delay.ms` property in the Kafka configuration to delay the time it takes for consumers to join a new consumer group before performing a rebalance. Introducing a delay helps avoid triggering several rebalances when starting multiple consumers near the same time. The appropriate delay depends on the orchestration used and might not be suitable in some circumstances.

### Avoid frequent scaling

- Keep the number of consumers stable, scaling only when necessary and in controlled increments.
- Monitor system performance and adjust your scaling strategy as needed.
  - Lag per partition should be constant and low.
  - Records processed per second by consumers should match the records per second in the input topics.
- Use the Kafka Exporter to check for consumer lag and determine if scaling is required.

## Implement dynamic scaling policies

- If using dynamic or event-driven tools for scaling of consumer applications, set lag thresholds based on the backlog of messages.
- Define maximum and minimum replica counts for consumer groups.
- Set periods between scaling events to prevent rapid scaling.

**NOTE**

In cases where lengthy message processing is unavoidable, consider pausing and resuming partitions as needed. If you pause all partitions, `poll()` returns no records, allowing you to keep calling it without overwhelming the consumers. Alternatively, you can offload the processing tasks to a pool of worker threads. This helps prevents delays and potential rebalances.

## 35.6. Handling high volumes of messages

If your Strimzi deployment needs to handle a high volume of messages, you can use configuration options to optimize for throughput and latency.

Producer and consumer configuration can help control the size and frequency of requests to Kafka brokers. For more information on the configuration options, see the following:

- [Apache Kafka configuration documentation for producers](#)
- [Apache Kafka configuration documentation for consumers](#)

You can also use the same configuration options with the producers and consumers used by the Kafka Connect runtime source connectors (including MirrorMaker 2) and sink connectors.

### Source connectors

- Producers from the Kafka Connect runtime send messages to the Kafka cluster.
- For MirrorMaker 2, since the source system is Kafka, consumers retrieve messages from a source Kafka cluster.

### Sink connectors

- Consumers from the Kafka Connect runtime retrieve messages from the Kafka cluster.

For consumers, you might increase the amount of data fetched in a single fetch request to reduce latency. You increase the fetch request size using the `fetch.max.bytes` and `max.partition.fetch.bytes` properties. You can also set a maximum limit on the number of messages returned from the consumer buffer using the `max.poll.records` property.

For MirrorMaker 2, configure the `fetch.max.bytes`, `max.partition.fetch.bytes`, and `max.poll.records` values at the source connector level (`consumer.*`), as they relate to the specific consumer that fetches messages from the source.

For producers, you might increase the size of the message batches sent in a single produce request. You increase the batch size using the `batch.size` property. A larger batch size reduces the number of outstanding messages ready to be sent and the size of the backlog in the message queue. Messages being sent to the same partition are batched together. A produce request is sent to the target cluster

when the batch size is reached. By increasing the batch size, produce requests are delayed and more messages are added to the batch and sent to brokers at the same time. This can improve throughput when you have just a few topic partitions that handle large numbers of messages.

Consider the number and size of the records that the producer handles for a suitable producer batch size.

Use `linger.ms` to add a wait time in milliseconds to delay produce requests when producer load decreases. The delay means that more records can be added to batches if they are under the maximum batch size.

Configure the `batch.size` and `linger.ms` values at the source connector level (`producer.override.*`), as they relate to the specific producer that sends messages to the target Kafka cluster.

For Kafka Connect source connectors, the data streaming pipeline to the target Kafka cluster is as follows:

*Data streaming pipeline for Kafka Connect source connector*

**external data source → (Kafka Connect tasks) source message queue → producer buffer → target Kafka topic**

For Kafka Connect sink connectors, the data streaming pipeline to the target external data source is as follows:

*Data streaming pipeline for Kafka Connect sink connector*

**source Kafka topic → (Kafka Connect tasks) sink message queue → consumer buffer → external data source**

For MirrorMaker 2, the data mirroring pipeline to the target Kafka cluster is as follows:

*Data mirroring pipeline for MirrorMaker 2*

**source Kafka topic → (Kafka Connect tasks) source message queue → producer buffer → target Kafka topic**

The producer sends messages in its buffer to topics in the target Kafka cluster. While this is happening, Kafka Connect tasks continue to poll the data source to add messages to the source message queue.

The size of the producer buffer for the source connector is set using the `producer.override.buffer.memory` property. Tasks wait for a specified timeout period (`offset.flush.timeout.ms`) before the buffer is flushed. This should be enough time for the sent messages to be acknowledged by the brokers and offset data committed. The source task does not wait for the producer to empty the message queue before committing offsets, except during shutdown.

If the producer is unable to keep up with the throughput of messages in the source message queue, buffering is blocked until there is space available in the buffer within a time period bounded by `max.block.ms`. Any unacknowledged messages still in the buffer are sent during this period. New messages are not added to the buffer until these messages are acknowledged and flushed.

You can try the following configuration changes to keep the underlying source message queue of outstanding messages at a manageable size:

- Increasing the default value in milliseconds of the `offset.flush.timeout.ms`
- Ensuring that there are enough CPU and memory resources
- Increasing the number of tasks that run in parallel by doing the following:
  - Increasing the number of tasks that run in parallel using the `tasksMax` property
  - Increasing the number of worker nodes that run tasks using the `replicas` property

Consider the number of tasks that can run in parallel according to the available CPU and memory resources and number of worker nodes. You might need to keep adjusting the configuration values until they have the desired effect.

### 35.6.1. Configuring Kafka Connect for high-volume messages

Kafka Connect fetches data from the source external data system and hands it to the Kafka Connect runtime producers so that it's replicated to the target cluster.

The following example shows configuration for Kafka Connect using the `KafkaConnect` custom resource.

*Example Kafka Connect configuration for handling high volumes of messages*

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaConnect
metadata:
  name: my-connect-cluster
  annotations:
    strimzi.io/use-connector-resources: "true"
spec:
  replicas: 3
  config:
    offset.flush.timeout.ms: 10000
    # ...
  resources:
    requests:
      cpu: "1"
      memory: 2Gi
    limits:
      cpu: "2"
      memory: 2Gi
  # ...
```

Producer configuration is added for the source connector, which is managed using the `KafkaConnector` custom resource.

### *Example source connector configuration for handling high volumes of messages*

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaConnector
metadata:
  name: my-source-connector
  labels:
    strimzi.io/cluster: my-connect-cluster
spec:
  class: org.apache.kafka.connect.file.FileStreamSourceConnector
  tasksMax: 2
  config:
    producer.override.batch.size: 327680
    producer.override.linger.ms: 100
    # ...
```

**NOTE** [FileStreamSourceConnector](#) and [FileStreamSinkConnector](#) are provided as example connectors. For information on deploying them as [KafkaConnector](#) resources, see [Deploying KafkaConnector resources](#).

Consumer configuration is added for the sink connector.

### *Example sink connector configuration for handling high volumes of messages*

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaConnector
metadata:
  name: my-sink-connector
  labels:
    strimzi.io/cluster: my-connect-cluster
spec:
  class: org.apache.kafka.connect.file.FileStreamSinkConnector
  tasksMax: 2
  config:
    consumer.fetch.max.bytes: 52428800
    consumer.max.partition.fetch.bytes: 1048576
    consumer.max.poll.records: 500
    # ...
```

If you are using the Kafka Connect API instead of the [KafkaConnector](#) custom resource to manage your connectors, you can add the connector configuration as a JSON object.

### *Example curl request to add source connector configuration for handling high volumes of messages*

```
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"
  "producer.override.batch.size": 327680
  "producer.override.linger.ms": 100
}
}'
```

## 35.6.2. Configuring MirrorMaker 2 for high-volume messages

MirrorMaker 2 fetches data from the source cluster and hands it to the Kafka Connect runtime producers so that it's replicated to the target cluster.

The following example shows the configuration for MirrorMaker 2 using the [KafkaMirrorMaker2](#) custom resource.

*Example MirrorMaker 2 configuration for handling high volumes of messages*

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaMirrorMaker2
metadata:
  name: my-mirror-maker2
spec:
  version: 4.1.1
  replicas: 1
  target:
    alias: "my-cluster-target"
    bootstrapServers: my-cluster-target-kafka-bootstrap:9092
    groupId: my-mirror-maker2-group
    configStorageTopic: my-mirror-maker2-config
    offsetStorageTopic: my-mirror-maker2-offset
    statusStorageTopic: my-mirror-maker2-status
    config:
      offset.flush.timeout.ms: 10000
  mirrors:
  - source:
      alias: "my-cluster-source"
      bootstrapServers: my-cluster-source-kafka-bootstrap:9092
      sourceConnector:
        tasksMax: 2
        config:
          producer.override.batch.size: 327680
          producer.override.linger.ms: 100
          consumer.fetch.max.bytes: 52428800
          consumer.max.partition.fetch.bytes: 1048576
          consumer.max.poll.records: 500
# ...
```

```
resources:
  requests:
    cpu: "1"
    memory: Gi
  limits:
    cpu: "2"
    memory: 4Gi
```

### 35.6.3. Checking the MirrorMaker 2 message flow

If you are using Prometheus and Grafana to monitor your deployment, you can check the MirrorMaker 2 message flow.

The example MirrorMaker 2 Grafana dashboards provided with Strimzi show the following metrics related to the flush pipeline.

- The number of messages in Kafka Connect's outstanding messages queue
- The available bytes of the producer buffer
- The offset commit timeout in milliseconds

You can use these metrics to gauge whether or not you need to tune your configuration based on the volume of messages.

## 35.7. Handling large message sizes

Kafka's default batch size for messages is 1MB, which is optimal for maximum throughput in most use cases. Kafka can accommodate larger batches at a reduced throughput, assuming adequate disk capacity.

Large message sizes can be handled in four ways:

1. Brokers, producers, and consumers are configured to accommodate larger message sizes.
2. [Producer-side message compression](#) writes compressed messages to the log.
3. Reference-based messaging sends only a reference to data stored in some other system in the message's payload.
4. Inline messaging splits messages into chunks that use the same key, which are then combined on output using a stream-processor like Kafka Streams.

Unless you are handling very large messages, the configuration approach is recommended. The reference-based messaging and message compression options cover most other situations. With any of these options, care must be taken to avoid introducing performance issues.

### 35.7.1. Configuring Kafka components to handle larger messages

Large messages can impact system performance and introduce complexities in message processing. If they cannot be avoided, there are configuration options available. To handle larger messages efficiently and prevent blocks in the message flow, consider adjusting the following configurations:

- Adjusting the maximum record batch size:
  - Set `message.max.bytes` at the broker level to support larger record batch sizes for all topics.
  - Set `max.message.bytes` at the topic level to support larger record batch sizes for individual topics.
- Increasing the maximum size of messages fetched by each partition follower (`replica.fetch.max.bytes`).
- Increasing the batch size (`batch.size`) for producers to increase the size of message batches sent in a single produce request.
- Configuring a higher maximum request size for producers (`max.request.size`) and consumers (`fetch.max.bytes`) to accommodate larger record batches.
- Setting a higher maximum limit (`max.partition.fetch.bytes`) on how much data is returned to consumers for each partition.

Ensure that the maximum size for batch requests is at least as large as `message.max.bytes` to accommodate the largest record batch size.

#### *Example broker configuration*

```
message.max.bytes: 10000000
replica.fetch.max.bytes: 10485760
```

#### *Example producer configuration*

```
batch.size: 327680
max.request.size: 10000000
```

#### *Example consumer configuration*

```
fetch.max.bytes: 10000000
max.partition.fetch.bytes: 10485760
```

It's also possible to configure the producers and consumers used by other Kafka components like HTTP Bridge, Kafka Connect, and MirrorMaker 2 to handle larger messages more effectively.

## HTTP Bridge

Configure the HTTP Bridge using specific producer and consumer configuration properties:

- `producer.config` for producers
- `consumer.config` for consumers

#### *Example HTTP Bridge configuration*

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaBridge
metadata:
  name: my-bridge
```

```
spec:  
  # ...  
  producer:  
    config:  
      batch.size: 327680  
      max.request.size: 10000000  
  consumer:  
    config:  
      fetch.max.bytes: 10000000  
      max.partition.fetch.bytes: 10485760  
    # ...
```

## Kafka Connect

For Kafka Connect, configure the source and sink connectors responsible for sending and retrieving messages using prefixes for producer and consumer configuration properties:

- **producer.override** for the producer used by the source connector to send messages to a Kafka cluster
- **consumer** for the consumer used by the sink connector to retrieve messages from a Kafka cluster

*Example Kafka Connect source connector configuration*

```
apiVersion: kafka.strimzi.io/v1  
kind: KafkaConnector  
metadata:  
  name: my-source-connector  
  labels:  
    strimzi.io/cluster: my-connect-cluster  
spec:  
  # ...  
  config:  
    producer.override.batch.size: 327680  
    producer.override.max.request.size: 10000000  
  # ...
```

*Example Kafka Connect sink connector configuration*

```
apiVersion: kafka.strimzi.io/v1  
kind: KafkaConnector  
metadata:  
  name: my-sink-connector  
  labels:  
    strimzi.io/cluster: my-connect-cluster  
spec:  
  # ...  
  config:  
    consumer.fetch.max.bytes: 10000000  
    consumer.max.partition.fetch.bytes: 10485760
```

```
# ...
```

## MirrorMaker 2

For MirrorMaker 2, configure the source connector responsible for retrieving messages from the source Kafka cluster using prefixes for producer and consumer configuration properties:

- `producer.override` for the runtime Kafka Connect producer used to replicate data to the target Kafka cluster
- `consumer` for the consumer used by the sink connector to retrieve messages from the source Kafka cluster

*Example MirrorMaker 2 source connector configuration*

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaMirrorMaker2
metadata:
  name: my-mirror-maker2
spec:
  # ...
  mirrors:
  - source:
    # ...
    sourceConnector:
      tasksMax: 2
      config:
        producer.override.batch.size: 327680
        producer.override.max.request.size: 10000000
        consumer.fetch.max.bytes: 10000000
        consumer.max.partition.fetch.bytes: 10485760
    # ...
```

### 35.7.2. Producer-side compression

For producer configuration, you specify a `compression.type`, such as Gzip, which is then applied to batches of data generated by the producer. Using the broker configuration `compression.type=producer` (default), the broker retains whatever compression the producer used. Whenever producer and topic compression do not match, the broker has to compress batches again prior to appending them to the log, which impacts broker performance.

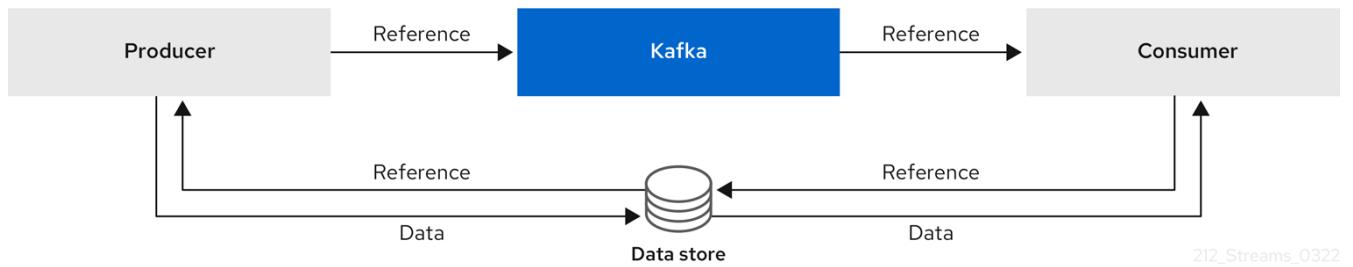
Compression also adds additional processing overhead on the producer and decompression overhead on the consumer, but includes more data in a batch, so is often beneficial to throughput when message data compresses well.

Combine producer-side compression with fine-tuning of the batch size to facilitate optimum throughput. Using metrics helps to gauge the average batch size needed.

### 35.7.3. Reference-based messaging

Reference-based messaging is useful for data replication when you do not know how big a message will be. The external data store must be fast, durable, and highly available for this configuration to work. Data is written to the data store and a reference to the data is returned. The producer sends a message containing the reference to Kafka. The consumer gets the reference from the message and uses it to fetch the data from the data store.

### 35.7.4. Reference-based messaging flow



As the message passing requires more trips, end-to-end latency will increase. Another significant drawback of this approach is there is no automatic clean up of the data in the external system when the Kafka message gets cleaned up. A hybrid approach would be to only send large messages to the data store and process standard-sized messages directly.

#### *Inline messaging*

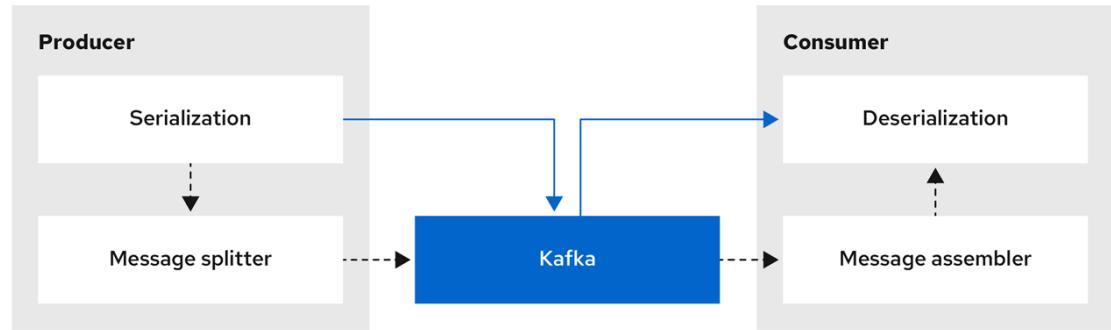
Inline messaging is complex, but it does not have the overhead of depending on external systems like reference-based messaging.

The producing client application has to serialize and then chunk the data if the message is too big. The producer then uses the Kafka `ByteArraySerializer` or similar to serialize each chunk again before sending it. The consumer tracks messages and buffers chunks until it has a complete message. The consuming client application receives the chunks, which are assembled before deserialization. Complete messages are delivered to the rest of the consuming application in order according to the offset of the first or last chunk for each set of chunked messages.

The consumer should commit its offset only after receiving and processing all message chunks to ensure accurate tracking of message delivery and prevent duplicates during rebalancing. Chunks might be spread across segments. Consumer-side handling should cover the possibility that a chunk becomes unavailable if a segment is subsequently deleted.

→ Standard message

→ Large message



2I2\_Streams\_0322

Figure 10. Inline messaging flow

Inline messaging has a performance overhead on the consumer side because of the buffering required, particularly when handling a series of large messages in parallel. The chunks of large messages can become interleaved, so that it is not always possible to commit when all the chunks of a message have been consumed if the chunks of another large message in the buffer are incomplete. For this reason, the buffering is usually supported by persisting message chunks or by implementing commit logic.