

# Deploying and Upgrading Strimzi

# Table of Contents

1. Deployment overview	1
1.1. How Strimzi supports Kafka	1
1.2. Strimzi Operators	1
1.2.1. Cluster Operator	2
1.2.2. Topic Operator	3
1.2.3. User Operator	4
1.3. Strimzi custom resources	5
1.3.1. Strimzi custom resource example	5
2. What is deployed with Strimzi	9
2.1. Order of deployment	9
2.2. Additional deployment configuration options	9
2.2.1. Securing Kafka	10
2.2.2. Monitoring your deployment	10
3. Preparing for your Strimzi deployment	11
3.1. Deployment prerequisites	11
3.2. Downloading Strimzi release artifacts	11
3.3. Pushing container images to your own registry	12
3.4. Designating Strimzi administrators	13
3.5. Alternative cluster deployment options	13
3.5.1. Installing a local Kubernetes cluster	14
3.5.2. Installing a local OpenShift cluster	14
4. Deploying Strimzi	16
4.1. Create the Kafka cluster	16
4.1.1. Deploying the Cluster Operator	17
4.1.2. Deploying Kafka	22
4.1.3. Alternative standalone deployment options for Strimzi Operators	26
4.2. Deploy Kafka Connect	29
4.2.1. Deploying Kafka Connect to your Kubernetes cluster	30
4.2.2. Kafka Connect configuration for multiple instances	31
4.2.3. Extending Kafka Connect with connector plug-ins	31
4.2.4. Creating and managing connectors	38
4.2.5. Deploying a <code>KafkaConnector</code> resource to Kafka Connect	39
4.3. Deploy Kafka MirrorMaker	40
4.3.1. Deploying Kafka MirrorMaker to your Kubernetes cluster	40
4.4. Deploy Kafka Bridge	41
4.4.1. Deploying Kafka Bridge to your Kubernetes cluster	41
5. Setting up client access to the Kafka cluster	42
5.1. Deploying example clients	42

5.2. Setting up access for clients outside of Kubernetes .....	42
6. Introducing Metrics to Kafka .....	50
6.1. Example metrics files .....	50
6.1.1. Example Grafana dashboards .....	52
6.1.2. Example Prometheus metrics configuration .....	52
6.2. Add Prometheus and Grafana .....	53
6.2.1. Deploying Prometheus metrics configuration .....	53
6.2.2. Setting up Prometheus .....	54
6.2.3. Setting up Prometheus Alertmanager .....	57
6.2.4. Setting up Grafana .....	60
6.2.5. Using metrics with Minikube or Minishift .....	66
6.3. Add Kafka Exporter .....	66
6.3.1. Monitoring Consumer lag .....	66
6.3.2. Example Kafka Exporter alerting rules .....	67
6.3.3. Exposing Kafka Exporter metrics .....	68
6.3.4. Configuring Kafka Exporter .....	69
6.3.5. Enabling the Kafka Exporter Grafana dashboard .....	71
6.4. Monitor Kafka Bridge .....	72
6.4.1. Configuring Kafka Bridge .....	72
6.4.2. Enabling the Kafka Bridge Grafana dashboard .....	73
6.5. Monitor Cruise Control .....	74
6.5.1. Configuring Cruise Control .....	74
6.5.2. Enabling the Cruise Control Grafana dashboard .....	74
7. Upgrading Strimzi .....	76
7.1. Strimzi and Kafka upgrades .....	76
7.1.1. Kafka versions .....	76
7.1.2. Upgrading the Cluster Operator .....	77
7.1.3. Upgrading Kafka .....	78
7.2. Strimzi resource upgrades .....	87
7.2.1. Upgrading Kafka resources .....	88
7.2.2. Upgrading Kafka Connect resources .....	91
7.2.3. Upgrading Kafka Connect S2I resources .....	92
7.2.4. Upgrading Kafka MirrorMaker resources .....	94
7.2.5. Upgrading Kafka Topic resources .....	95
7.2.6. Upgrading Kafka User resources .....	95
8. Downgrading Strimzi .....	97
8.1. Downgrading the Cluster Operator to a previous version .....	97
8.2. Downgrading Kafka .....	98
8.2.1. Kafka version compatibility for downgrades .....	98
8.2.2. Downgrading Kafka brokers and client applications .....	99

# Chapter 1. Deployment overview

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

This guide provides instructions on all the options available for deploying and upgrading Strimzi, describing what is deployed, and the order of deployment required to run Apache Kafka in a Kubernetes cluster.

As well as describing the deployment steps, the guide also provides pre- and post-deployment instructions to prepare for and verify a deployment. Additional deployment options described include the steps to introduce metrics. Upgrade instructions are provided for Strimzi and Kafka upgrades.

Strimzi is designed to work on all types of Kubernetes cluster regardless of distribution, from public and private clouds to local deployments intended for development.

## 1.1. How Strimzi supports Kafka

Strimzi provides container images and Operators for running Kafka on Kubernetes. Strimzi Operators are fundamental to the running of Strimzi. The Operators provided with Strimzi are purpose-built with specialist operational knowledge to effectively manage Kafka.

Operators simplify the process of:

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

## 1.2. Strimzi Operators

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

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

### Operators

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

## Cluster Operator

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

## Entity Operator

Comprises the Topic Operator and User Operator

## Topic Operator

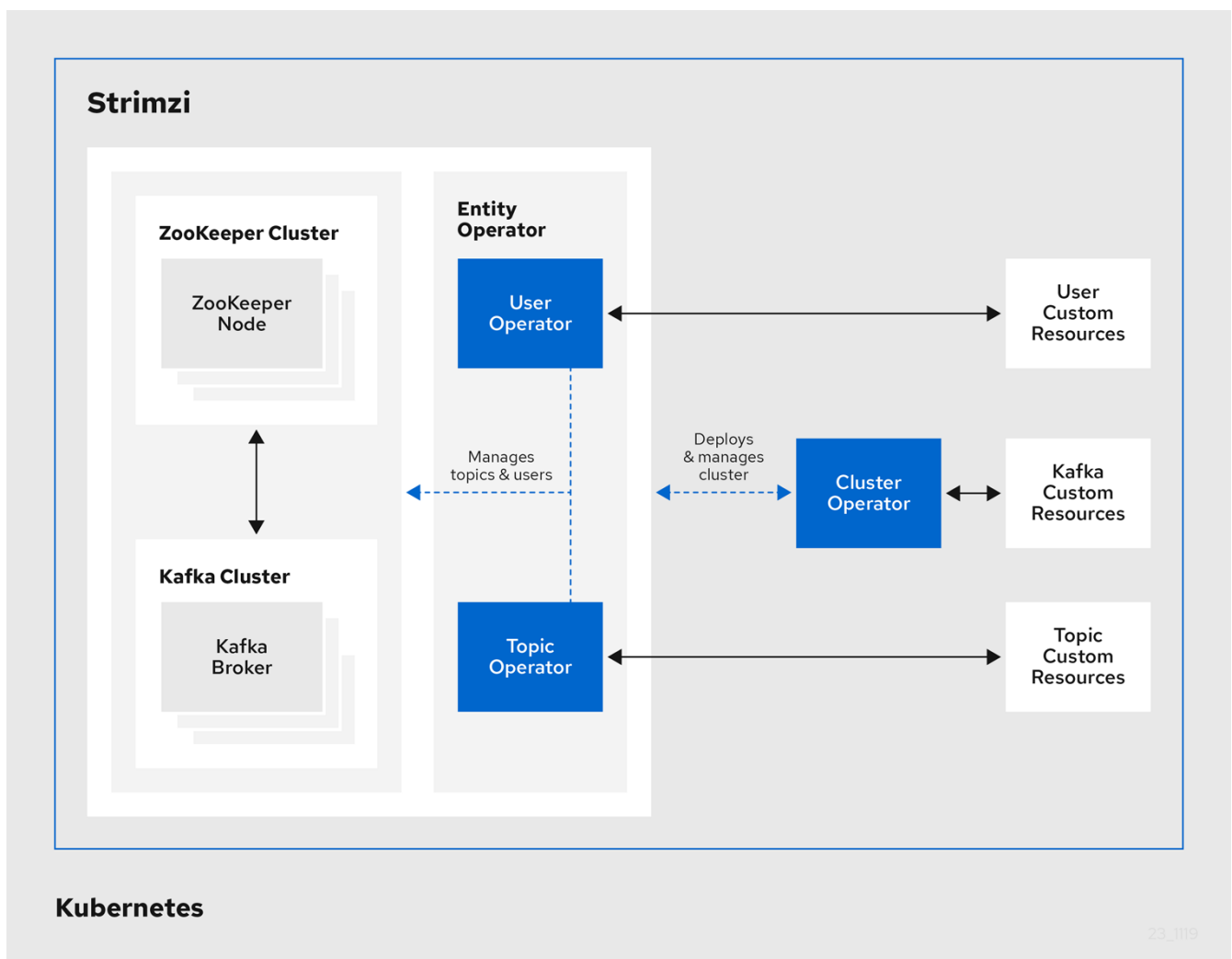
Manages Kafka topics

## User Operator

Manages Kafka users

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

*Operators within the Strimzi architecture*



### 1.2.1. Cluster Operator

Strimzi uses the Cluster Operator to deploy and manage clusters for:

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

Custom resources are used to deploy the clusters.

For example, to deploy a Kafka cluster:

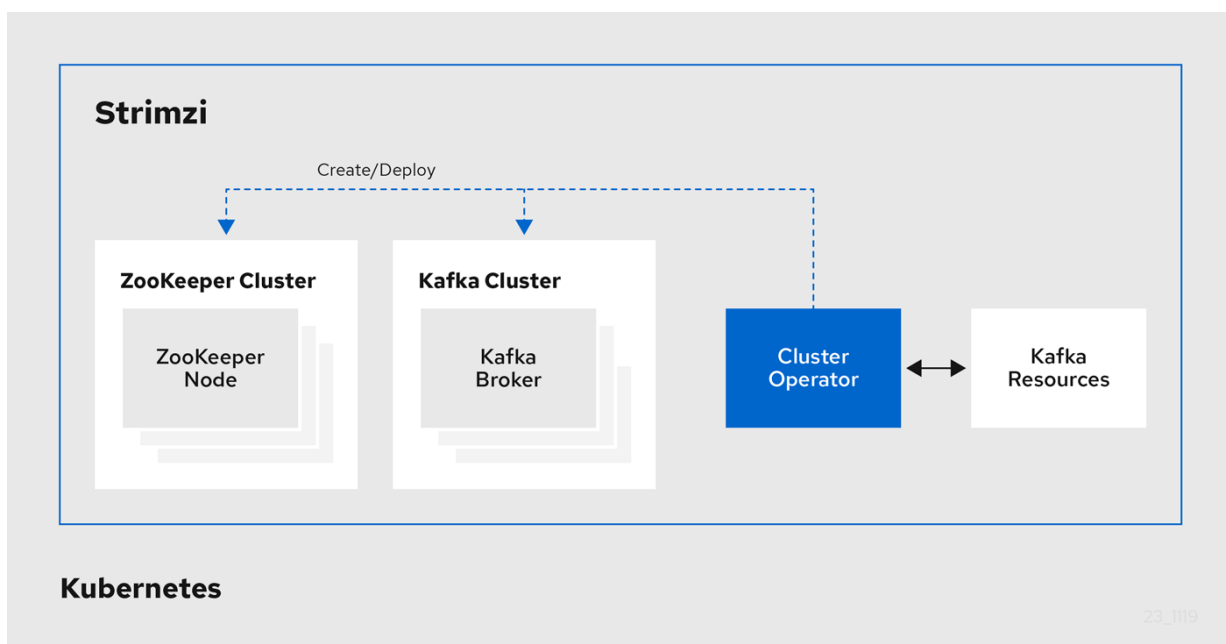
- A **Kafka** resource with the cluster configuration is created within the Kubernetes cluster.
- The Cluster Operator deploys a corresponding Kafka cluster, based on what is declared in the **Kafka** resource.

The Cluster Operator can also deploy (through configuration of the **Kafka** resource):

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

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

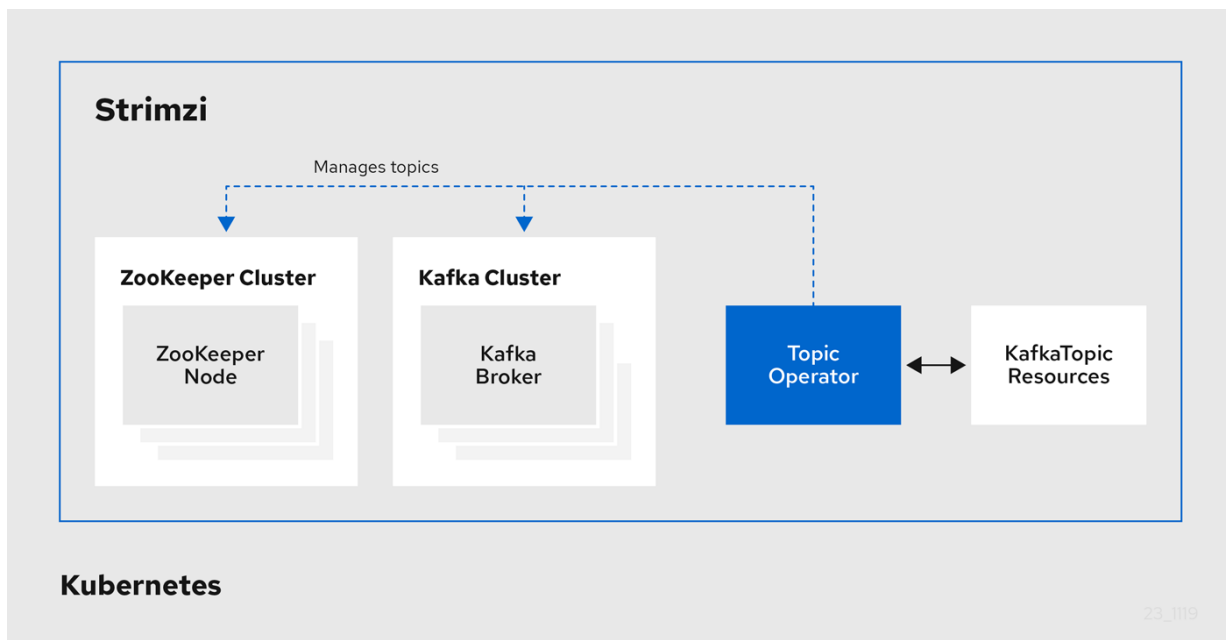
*Example architecture for the Cluster Operator*



### 1.2.2. Topic Operator

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

*Example architecture for the Topic Operator*



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

Specifically, if a **KafkaTopic** is:

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

Working in the other direction, if a topic is:

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

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

If the topic is reconfigured or reassigned to different Kafka nodes, the **KafkaTopic** will always be up to date.

### 1.2.3. User Operator

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

For example, if a **KafkaUser** is:

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

- Changed, the User Operator updates the user it describes

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

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

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

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

## 1.3. Strimzi custom resources

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

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

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

*Additional resources*

- [Extend the Kubernetes API with CustomResourceDefinitions](#)

### 1.3.1. 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](#) in the *Deploying and Upgrading Strimzi* guide.



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.

**WARNING**

When CRDs are deleted, custom resources of that type are also deleted. Additionally, the resources created by the custom resource, such as pods and statefulsets are also deleted.

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.

```

apiVersion: kafka.strimzi.io/v1beta1
kind: CustomResourceDefinition
metadata: ❶
  name: kafkatopics.kafka.strimzi.io
  labels:
    app: strimzi
spec: ❷
  group: kafka.strimzi.io
  versions:
    v1beta1
  scope: Namespaced
  names:
    # ...
    singular: kafkatopic
    plural: kafkatopics
    shortNames:
      - kt ❸
  additionalPrinterColumns: ❹
    # ...
  subresources:
    status: {} ❺
  validation: ❻
    openAPIV3Schema:
      properties:
        spec:
          type: object
          properties:
            partitions:
              type: integer
              minimum: 1
            replicas:
              type: integer
              minimum: 1
              maximum: 32767
    # ...

```

- ❶ The metadata for the topic CRD, its name and a label to identify the CRD.
- ❷ 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`.
- ❸ The shortname can be used in CLI commands. For example, `kubectl get kt` can be used as an abbreviation instead of `kubectl get kafkatopic`.
- ❹ The information presented when using a `get` command on the custom resource.
- ❺ The current status of the CRD as described in the [schema reference](#) for the resource.
- ❻ openAPIV3Schema 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/v1beta1
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.

# Chapter 2. What is deployed with Strimzi

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

A typical deployment incorporating Kafka components might include:

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

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

## 2.1. Order of deployment

The required order of deployment to a Kubernetes cluster is as follows:

1. Deploy the Cluster operator to manage your Kafka cluster
2. Deploy the Kafka cluster with the ZooKeeper cluster, and include the Topic Operator and User Operator in the deployment
3. Optionally deploy:
  - The Topic Operator and User Operator standalone if you did not deploy them with the Kafka cluster
  - Kafka Connect
  - Kafka MirrorMaker
  - Kafka Bridge
  - Components for the monitoring of metrics

## 2.2. Additional deployment configuration options

The deployment procedures in this guide describe a deployment using the example installation YAML files provided with Strimzi. The procedures highlight any important configuration considerations, but they do not describe all the configuration options available.

You can use custom resources to refine your deployment.

You may wish to review the configuration options available for Kafka components before you deploy Strimzi. For more information on the configuration through custom resources, see [Deployment configuration](#) in the *Using Strimzi* guide.

### 2.2.1. Securing Kafka

On deployment, the Cluster Operator automatically sets up TLS certificates for data encryption and authentication within your cluster.

Strimzi provides additional configuration options for *encryption*, *authentication* and *authorization*, which are described in the *Using Strimzi* guide:

- Secure data exchange between the Kafka cluster and clients by [Managing secure access to Kafka](#).
- Configure your deployment to use an authorization server to provide [OAuth 2.0 authentication](#) and [OAuth 2.0 authorization](#).
- [Secure Kafka using your own certificates](#).

### 2.2.2. Monitoring your deployment

Strimzi supports additional deployment options to monitor your deployment.

- Extract metrics and monitor Kafka components by [deploying Prometheus and Grafana with your Kafka cluster](#).
- Extract additional metrics, particularly related to monitoring consumer lag, by [deploying Kafka Exporter with your Kafka cluster](#).
- Track messages end-to-end by [setting up distributed tracing](#), as described in the *Using Strimzi* guide.

# Chapter 3. Preparing for your Strimzi deployment

This section shows how you prepare for a Strimzi deployment, describing:

- [The prerequisites you need before you can deploy Strimzi](#)
- [How to download the Strimzi release artifacts to use in your deployment](#)
- [How to push the Strimzi container images into your own registry \(if required\)](#)
- [How to set up \*admin\* roles for configuration of custom resources used in deployment](#)
- [Alternative deployment options to Kubernetes using \*Minikube\* or \*Minishift\*](#)

## NOTE

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

## 3.1. Deployment prerequisites

To deploy Strimzi, make sure:

- A Kubernetes 1.16 and later cluster is available
- The `kubectl` command-line tool is installed and configured to connect to the running cluster.

## 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.

## Alternatives if a Kubernetes cluster is not available

If you do not have access to a Kubernetes cluster, as an alternative you can try installing Strimzi with:

- [Minikube](#)
- [Minishift](#)

## 3.2. Downloading Strimzi release artifacts

To install Strimzi, download the release artifacts from [GitHub](#).

Strimzi release artifacts include sample YAML files to help you deploy the components of Strimzi to Kubernetes, perform common operations, and configure your Kafka cluster.

Use `kubectl` to deploy the Cluster Operator from the `install/cluster-operator` folder of the downloaded ZIP file. For more information about deploying and configuring the Cluster Operator, see [Deploying the Cluster Operator](#).

In addition, if you want to use standalone installations of the Topic and User Operators with a Kafka cluster that is not managed by the Strimzi Cluster Operator, you can deploy them from the [install/topic-operator](#) and [install/user-operator](#) folders.

**NOTE**

Additionally, Strimzi container images are available through the [Container Registry](#). However, we recommend that you use the YAML files provided to deploy Strimzi.

### 3.3. 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>• <a href="#">quay.io/strimzi/kafka:0.21.1-kafka-2.5.0</a></li><li>• <a href="#">quay.io/strimzi/kafka:0.21.1-kafka-2.5.1</a></li><li>• <a href="#">quay.io/strimzi/kafka:0.21.1-kafka-2.6.0</a></li><li>• <a href="#">quay.io/strimzi/kafka:0.21.1-kafka-2.7.0</a></li></ul>	Strimzi image for running Kafka, including: <ul style="list-style-type: none"><li>• Kafka Broker</li><li>• Kafka Connect / S2I</li><li>• Kafka MirrorMaker</li><li>• ZooKeeper</li><li>• TLS Sidecars</li></ul>
Operator	<ul style="list-style-type: none"><li>• <a href="#">quay.io/strimzi/operator:0.21.1</a></li></ul>	Strimzi 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>
Kafka Bridge	<ul style="list-style-type: none"><li>• <a href="#">quay.io/strimzi/kafka-bridge:0.19.0</a></li></ul>	Strimzi image for running the Strimzi kafka Bridge

Container image	Namespace/Repository	Description
JmxTrans	<ul style="list-style-type: none"> <li>quay.io/strimzi/jmxtrans:0.2.1.1</li> </ul>	Strimzi image for running the Strimzi JmxTrans

## 3.4. Designating Strimzi administrators

Strimzi 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. Strimzi provides two cluster roles that you can use to assign these rights to other users:

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

When you install these roles, they will automatically aggregate (add) these rights to the default Kubernetes cluster roles. `strimzi-view` aggregates to the `view` role, and `strimzi-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 `strimzi-admin` role that allows non-cluster administrators to manage Strimzi resources.

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

### Prerequisites

- The Strimzi 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 `strimzi-view` and `strimzi-admin` cluster roles in Kubernetes.

```
kubectl apply -f install/strimzi-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
```

## 3.5. Alternative cluster deployment options

This section suggests alternatives to using a Kubernetes cluster.

If a Kubernetes cluster is unavailable, you can use:



- *Minikube* to create a local cluster
- *Minishift* to create a local OpenShift cluster and use OpenShift-specific features

### 3.5.1. Installing a local Kubernetes cluster

The easiest way to get started with Kubernetes is using Minikube. This section provides basic guidance on how to use it. For more information on the tools, refer to the documentation available online.

In order to interact with a Kubernetes cluster the `kubectl` utility needs to be installed.

You can download and install Minikube from the [Kubernetes website](#). Depending on the number of brokers you want to deploy inside the cluster, and if you need Kafka Connect running as well, try running Minikube with at least with 4 GB of RAM instead of the default 2 GB.

Once installed, start Minikube using:

```
minikube start --memory 4096
```

### 3.5.2. Installing a local OpenShift cluster

The easiest way to get started with OpenShift is using Minishift or `oc cluster up`. This section provides basic guidance on how to use them. For more information on the tools, refer to the documentation available online.

#### **oc cluster up**

The `oc` utility is one of the main tools for interacting with OpenShift. It provides a simple way of starting a local cluster using the command:

```
oc cluster up
```

This command requires Docker to be installed. You can find more information on [here](#).

#### **Minishift**

Minishift is an OpenShift installation within a VM. It can be downloaded and installed from the [Minishift website](#). Depending on the number of brokers you want to deploy inside the cluster, and if you need Kafka Connect running as well, try running Minishift with at least 4 GB of RAM instead of the default 2 GB.

Once installed, start Minishift using:

```
minishift start --memory 4GB
```

If you want to use `kubectl` with either an `oc cluster up` or `minishift` cluster, you will need to

[configure it](#), as unlike with Minikube this won't be done automatically.

### oc and kubectl commands

The `oc` command functions as an alternative to `kubectl`. In almost all cases the example `kubectl` commands given 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*, while `kubectl` does not make such a distinction. For example, the `oc` equivalent of `kubectl taint` is `oc adm taint`.

# Chapter 4. Deploying Strimzi

Having [prepared your environment for a deployment of Strimzi](#), this section shows:

- [How to create the Kafka cluster](#)
- Optional procedures to deploy other Kafka components according to your requirements:
  - [Kafka Connect](#)
  - [Kafka MirrorMaker](#)
  - [Kafka Bridge](#)

The procedures assume a Kubernetes cluster is available and running.

This section describes the procedures to deploy Strimzi on Kubernetes 1.16 and later.

## NOTE

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

## 4.1. Create the Kafka cluster

In order to create your Kafka cluster, you deploy the Cluster Operator to manage the Kafka cluster, then deploy the Kafka cluster.

When deploying the Kafka cluster using the [Kafka](#) resource, you can deploy the Topic Operator and User Operator at the same time. Alternatively, if you are using a non-Strimzi Kafka cluster, you can deploy the Topic Operator and User Operator as standalone components.

### Deploying a Kafka cluster with the Topic Operator and User Operator

Perform these deployment steps if you want to use the Topic Operator and User Operator with a Kafka cluster managed by Strimzi.

1. [Deploy the Cluster Operator](#)
2. Use the Cluster Operator to deploy the:
  - a. [Kafka cluster](#)
  - b. [Topic Operator](#)
  - c. [User Operator](#)

### Deploying a standalone Topic Operator and User Operator

Perform these deployment steps if you want to use the Topic Operator and User Operator with a Kafka cluster that is **not managed** by Strimzi.

1. [Deploy the standalone Topic Operator](#)
2. [Deploy the standalone User Operator](#)

### 4.1.1. Deploying the Cluster Operator

The Cluster Operator is responsible for deploying and managing Apache Kafka clusters within a Kubernetes cluster.

The procedures in this section show:

- How to deploy the Cluster Operator to *watch*:
  - [A single namespace](#)
  - [Multiple namespaces](#)
  - [All namespaces](#)
- Alternative deployment options:
  - [How to deploy the Cluster Operator using a Helm chart](#)
  - [How to deploy the Cluster Operator from \*OperatorHub.io\*](#)

#### Watch options for a Cluster Operator deployment

When the Cluster Operator is running, it starts to *watch* for updates of Kafka resources.

You can choose to deploy the Cluster Operator to watch Kafka resources from:

- A single namespace (the same namespace containing the Cluster Operator)
- Multiple namespaces
- All namespaces

**NOTE** | Strimzi provides example YAML files to make the deployment process easier.

The Cluster Operator watches for changes to the following resources:

- **Kafka** for the Kafka cluster.
- **KafkaConnect** for the Kafka Connect cluster.
- **KafkaConnectS2I** for the Kafka Connect cluster with Source2Image support.
- **KafkaConnector** for creating and managing connectors in a Kafka Connect cluster.
- **KafkaMirrorMaker** for the Kafka MirrorMaker instance.
- **KafkaBridge** for the Kafka Bridge instance

When one of these resources is created in the Kubernetes cluster, the operator gets the cluster description from the resource and starts creating a new cluster for the resource by creating the necessary Kubernetes resources, such as StatefulSets, Services and ConfigMaps.

Each time a Kafka resource is updated, the operator performs corresponding updates on the Kubernetes resources that make up the cluster for the resource.

Resources are either patched or deleted, and then recreated in order to make the cluster for the resource reflect the desired state of the cluster. This operation might cause a rolling update that

might lead to service disruption.

When a resource is deleted, the operator undeploys the cluster and deletes all related Kubernetes resources.

## 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

- This procedure requires use of a Kubernetes user account which is able to create `CustomResourceDefinitions`, `ClusterRoles` and `ClusterRoleBindings`. Use of Role Base Access Control (RBAC) in the Kubernetes cluster usually means that permission to create, edit, and delete these resources is limited to Kubernetes cluster administrators, such as `system:admin`.

### 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 apply -f install/cluster-operator -n my-cluster-operator-namespace
```

3. Verify that the Cluster Operator was successfully deployed:

```
kubectl get deployments
```

## 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

- This procedure requires use of a Kubernetes user account which is able to create `CustomResourceDefinitions`, `ClusterRoles` and `ClusterRoleBindings`. Use of Role Base Access Control (RBAC) in the Kubernetes cluster usually means that permission to create, edit, and delete these resources is limited to Kubernetes cluster administrators, such as `system:admin`.

## 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:0.21.1
          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 apply -f install/cluster-operator/020-RoleBinding-strimzi-cluster-operator.yaml -n watched-namespace
kubectl apply -f install/cluster-operator/031-RoleBinding-strimzi-cluster-operator-entity-operator-delegation.yaml -n watched-namespace
kubectl apply -f install/cluster-operator/032-RoleBinding-strimzi-cluster-operator-topic-operator-delegation.yaml -n watched-namespace
```

#### 4. Deploy the Cluster Operator:

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

#### 5. Verify that the Cluster Operator was successfully deployed:

```
kubectl get deployments
```

### 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

- This procedure requires use of a Kubernetes user account which is able to create `CustomResourceDefinitions`, `ClusterRoles` and `ClusterRoleBindings`. Use of Role Base Access Control (RBAC) in the Kubernetes cluster usually means that permission to create, edit, and delete these resources is limited to Kubernetes cluster administrators, such as `system:admin`.

#### 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 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/strimzi/operator:0.21.1
          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-entity-operator-
delegation --clusterrole=strimzi-entity-operator --serviceaccount my-cluster-
operator-namespace:strimzi-cluster-operator
kubectl create clusterrolebinding strimzi-cluster-operator-topic-operator-
delegation --clusterrole=strimzi-topic-operator --serviceaccount my-cluster-
operator-namespace:strimzi-cluster-operator
```

Replace `my-cluster-operator-namespace` with the namespace you want to install the Cluster Operator into.

4. Deploy the Cluster Operator to your Kubernetes cluster.

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

5. Verify that the Cluster Operator was successfully deployed:



```
kubectl get deployments
```

## Deploying the Cluster Operator using a Helm Chart

As an alternative to using the YAML deployment files, this procedure shows how to deploy the Cluster Operator using a Helm chart provided with Strimzi.

### Prerequisites

- The Helm client must be installed on a local machine.
- Helm must be installed to the Kubernetes cluster.

For more information about Helm, see the [Helm website](#).

### Procedure

1. Add the Strimzi Helm Chart repository:

```
helm repo add strimzi https://strimzi.io/charts/
```

2. Deploy the Cluster Operator using the Helm command line tool:

```
helm install strimzi/strimzi-kafka-operator
```

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

```
helm ls
```

## Deploying the Cluster Operator from OperatorHub.io

[OperatorHub.io](#) is a catalog of Kubernetes Operators sourced from multiple providers. It offers you an alternative way to install stable versions of Strimzi using the Strimzi Kafka Operator.

The [Operator Lifecycle Manager](#) is used for the installation and management of all Operators published on [OperatorHub.io](#).

To install Strimzi from [OperatorHub.io](#), locate the *Strimzi Kafka Operator* and follow the instructions provided.

### 4.1.2. Deploying Kafka

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

The procedures in this section show:

- How to use the Cluster Operator to deploy:
  - [An ephemeral or persistent Kafka cluster](#)
  - The Topic Operator and User Operator by configuring the **Kafka** custom resource:
    - [Topic Operator](#)
    - [User Operator](#)
- Alternative standalone deployment procedures for the Topic Operator and User Operator:
  - [Deploy the standalone Topic Operator](#)
  - [Deploy the standalone User Operator](#)

When installing Kafka, Strimzi also installs a ZooKeeper cluster and adds the necessary configuration to connect Kafka with ZooKeeper.

## Deploying the Kafka cluster

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

The deployment uses a YAML file to provide the specification to create a **Kafka** resource.

Strimzi provides example YAMLs files for deployment in [examples/kafka/](#):

### **kafka-persistent.yaml**

Deploys a persistent cluster with three ZooKeeper and three Kafka nodes.

### **kafka-jbod.yaml**

Deploys a persistent cluster with three ZooKeeper and three Kafka nodes (each using multiple persistent volumes).

### **kafka-persistent-single.yaml**

Deploys a persistent cluster with a single ZooKeeper node and a single Kafka node.

### **kafka-ephemeral.yaml**

Deploys an ephemeral cluster with three ZooKeeper and three Kafka nodes.

### **kafka-ephemeral-single.yaml**

Deploys an ephemeral cluster with three ZooKeeper nodes and a single Kafka node.

In this procedure, we use the examples for an *ephemeral* and *persistent* Kafka cluster deployment:

## Ephemeral cluster

In general, an ephemeral (or temporary) Kafka cluster is suitable for development and testing purposes, not for production. This deployment uses **emptyDir** volumes for storing broker information (for ZooKeeper) and topics or partitions (for Kafka). Using an **emptyDir** volume means that its content is strictly related to the pod life cycle and is deleted when the pod goes down.

## Persistent cluster

A persistent Kafka cluster uses `PersistentVolumes` to store ZooKeeper and Kafka data. The `PersistentVolume` is acquired using a `PersistentVolumeClaim` to make it independent of the actual type of the `PersistentVolume`. For example, it can use Amazon EBS volumes in Amazon AWS deployments without any changes in the YAML files. The `PersistentVolumeClaim` can use a `StorageClass` to trigger automatic volume provisioning.

The example clusters are named `my-cluster` by default. The cluster name is defined by the name of the resource and 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.

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
metadata:
  name: my-cluster
# ...
```

For more information about configuring the `Kafka` resource, see [Kafka cluster configuration](#) in the *Using Strimzi* guide.

### Prerequisites

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

### Procedure

1. Create and deploy an *ephemeral* or *persistent* cluster.

For development or testing, you might prefer to use an ephemeral cluster. You can use a persistent cluster in any situation.

- To create and deploy an *ephemeral* cluster:

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

- To create and deploy a *persistent* cluster:

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

2. Verify that the Kafka cluster was successfully deployed:

```
kubectl get deployments
```

## 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`.

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](#) in the *Using Strimzi* guide.

#### Prerequisites

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

#### Procedure

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

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
metadata:
  name: my-cluster
spec:
  #...
  entityOperator:
    topicOperator: {}
    userOperator: {}
```

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

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

3. Create or update the resource:

Use `kubectl apply`:

```
kubectl apply -f <your-file>
```

## 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`.

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](#) in the *Using Strimzi* guide.

#### Prerequisites

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

#### Procedure

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

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
metadata:
  name: my-cluster
spec:
  #...
  entityOperator:
    topicOperator: {}
    userOperator: {}
```

2. Configure the User Operator `spec` using the properties described in [EntityUserOperatorSpec schema reference](#) in the *Using Strimzi* guide.

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

3. Create or update the resource:

```
kubectl apply -f <your-file>
```

### 4.1.3. Alternative standalone deployment options for Strimzi Operators

When deploying a Kafka cluster using the Cluster Operator, you can also deploy the Topic Operator and User Operator. Alternatively, you can perform a standalone deployment.

A standalone deployment means the Topic Operator and User Operator can operate with a Kafka cluster that is not managed by Strimzi.

#### Deploying the standalone Topic Operator

This procedure shows how to deploy the Topic Operator as a standalone component.

A standalone deployment requires configuration of environment variables, and is more complicated than [deploying the Topic Operator using the Cluster Operator](#). However, a standalone deployment is more flexible as the Topic Operator can operate with *any* Kafka cluster, not necessarily one deployed by the Cluster Operator.

#### Prerequisites

- You need an existing Kafka cluster for the Topic Operator to connect to.

#### Procedure

1. Edit the `Deployment.spec.template.spec.containers[0].env` properties in the `install/topic-operator/05-Deployment-strimzi-topic-operator.yaml` file by setting:

- a. `STRIMZI_KAFKA_BOOTSTRAP_SERVERS` to list the bootstrap brokers in your Kafka cluster, given as a comma-separated list of `hostname:port` pairs.
- b. `STRIMZI_ZOOKEEPER_CONNECT` to list the ZooKeeper nodes, given as a comma-separated list of `hostname:port` pairs. This should be the same ZooKeeper cluster that your Kafka cluster is using.
- c. `STRIMZI_NAMESPACE` to the Kubernetes namespace in which you want the operator to watch for `KafkaTopic` resources.
- d. `STRIMZI_RESOURCE_LABELS` to the label selector used to identify the `KafkaTopic` resources managed by the operator.
- e. `STRIMZI_FULL_RECONCILIATION_INTERVAL_MS` to specify the interval between periodic reconciliations, in milliseconds.
- f. `STRIMZI_TOPIC_METADATA_MAX_ATTEMPTS` to specify the number of attempts at getting topic metadata from Kafka. The time between each attempt is defined as an exponential back-off. Consider increasing this value when topic creation could take more time due to the number of partitions or replicas. Default `6`.
- g. `STRIMZI_ZOOKEEPER_SESSION_TIMEOUT_MS` to the ZooKeeper session timeout, in milliseconds. For example, `10000`. Default `20000` (20 seconds).
- h. `STRIMZI_TOPICS_PATH` to the Zookeeper node path where the Topic Operator stores its metadata. Default `/strimzi/topics`.
- i. `STRIMZI_TLS_ENABLED` to enable TLS support for encrypting the communication with Kafka brokers. Default `true`.
- j. `STRIMZI_TRUSTSTORE_LOCATION` to the path to the truststore containing certificates for enabling TLS based communication. Mandatory only if TLS is enabled through `STRIMZI_TLS_ENABLED`.
- k. `STRIMZI_TRUSTSTORE_PASSWORD` to the password for accessing the truststore defined by `STRIMZI_TRUSTSTORE_LOCATION`. Mandatory only if TLS is enabled through `STRIMZI_TLS_ENABLED`.
- l. `STRIMZI_KEYSTORE_LOCATION` to the path to the keystore containing private keys for enabling TLS based communication. Mandatory only if TLS is enabled through `STRIMZI_TLS_ENABLED`.
- m. `STRIMZI_KEYSTORE_PASSWORD` to the password for accessing the keystore defined by `STRIMZI_KEYSTORE_LOCATION`. Mandatory only if TLS is enabled through `STRIMZI_TLS_ENABLED`.
- n. `STRIMZI_LOG_LEVEL` to the level for printing logging messages. The value can be set to: `ERROR`, `WARNING`, `INFO`, `DEBUG`, and `TRACE`. Default `INFO`.
- o. `STRIMZI_JAVA_OPTS` (*optional*) to the Java options used for the JVM running the Topic Operator. An example is `-Xmx=512M -Xms=256M`.
- p. `STRIMZI_JAVA_SYSTEM_PROPERTIES` (*optional*) to list the `-D` options which are set to the Topic Operator. An example is `-Djavax.net.debug=verbose -DpropertyName=value`.

## 2. Deploy the Topic Operator:

```
kubectl apply -f install/topic-operator
```

## 3. Verify that the Topic Operator has been deployed successfully:

```
kubectl describe deployment strimzi-topic-operator
```

The Topic Operator is deployed when the `Replicas:` entry shows `1 available`.

#### NOTE

You may experience a delay with the deployment if you have a slow connection to the Kubernetes cluster and the images have not been downloaded before.

## Deploying the standalone User Operator

This procedure shows how to deploy the User Operator as a standalone component.

A standalone deployment requires configuration of environment variables, and is more complicated than [deploying the User Operator using the Cluster Operator](#). However, a standalone deployment is more flexible as the User Operator can operate with *any* Kafka cluster, not necessarily one deployed by the Cluster Operator.

### Prerequisites

- You need an existing Kafka cluster for the User Operator to connect to.

### Procedure

1. Edit the following `Deployment.spec.template.spec.containers[0].env` properties in the `install/user-operator/05-Deployment-strimzi-user-operator.yaml` file by setting:
  - a. `STRIMZI_KAFKA_BOOTSTRAP_SERVERS` to list the Kafka brokers, given as a comma-separated list of `hostname:port` pairs.
  - b. `STRIMZI_ZOOKEEPER_CONNECT` to list the ZooKeeper nodes, given as a comma-separated list of `hostname:port` pairs. This must be the same ZooKeeper cluster that your Kafka cluster is using. Connecting to ZooKeeper nodes with TLS encryption is not supported.
  - c. `STRIMZI_NAMESPACE` to the Kubernetes namespace in which you want the operator to watch for `KafkaUser` resources.
  - d. `STRIMZI_LABELS` to the label selector used to identify the `KafkaUser` resources managed by the operator.
  - e. `STRIMZI_FULL_RECONCILIATION_INTERVAL_MS` to specify the interval between periodic reconciliations, in milliseconds.
  - f. `STRIMZI_ZOOKEEPER_SESSION_TIMEOUT_MS` to the ZooKeeper session timeout, in milliseconds. For example, `10000`. Default `20000` (20 seconds).
  - g. `STRIMZI_CA_CERT_NAME` to point to a Kubernetes `Secret` that contains the public key of the Certificate Authority for signing new user certificates for TLS client authentication. The `Secret` must contain the public key of the Certificate Authority under the key `ca.crt`.
  - h. `STRIMZI_CA_KEY_NAME` to point to a Kubernetes `Secret` that contains the private key of the Certificate Authority for signing new user certificates for TLS client authentication. The `Secret` must contain the private key of the Certificate Authority under the key `ca.key`.
  - i. `STRIMZI_CLUSTER_CA_CERT_SECRET_NAME` to point to a Kubernetes `Secret` containing the public key of the Certificate Authority used for signing Kafka brokers certificates for enabling TLS-

based communication. The **Secret** must contain the public key of the Certificate Authority under the key `ca.crt`. This environment variable is optional and should be set only if the communication with the Kafka cluster is TLS based.

- j. **STRIMZI\_EO\_KEY\_SECRET\_NAME** to point to a Kubernetes **Secret** containing the private key and related certificate for TLS client authentication against the Kafka cluster. The **Secret** must contain the keystore with the private key and certificate under the key `entity-operator.p12`, and the related password under the key `entity-operator.password`. This environment variable is optional and should be set only if TLS client authentication is needed when the communication with the Kafka cluster is TLS based.
- k. **STRIMZI\_CA\_VALIDITY** the validity period for the Certificate Authority. Default is **365** days.
- l. **STRIMZI\_CA\_RENEWAL** the renewal period for the Certificate Authority.
- m. **STRIMZI\_LOG\_LEVEL** to the level for printing logging messages. The value can be set to: **ERROR**, **WARNING**, **INFO**, **DEBUG**, and **TRACE**. Default **INFO**.
- n. **STRIMZI\_GC\_LOG\_ENABLED** to enable garbage collection (GC) logging. Default **true**. Default is **30** days to initiate certificate renewal before the old certificates expire.
- o. **STRIMZI\_JAVA\_OPTS** (*optional*) to the Java options used for the JVM running User Operator. An example is `-Xmx=512M -Xms=256M`.
- p. **STRIMZI\_JAVA\_SYSTEM\_PROPERTIES** (*optional*) to list the `-D` options which are set to the User Operator. An example is `-Djavax.net.debug=verbose -DpropertyName=value`.

## 2. Deploy the User Operator:

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

## 3. Verify that the User Operator has been deployed successfully:

```
kubectl describe deployment strimzi-user-operator
```

The User Operator is deployed when the **Replicas:** entry shows **1 available**.

### NOTE

You may experience a delay with the deployment if you have a slow connection to the Kubernetes cluster and the images have not been downloaded before.

## 4.2. Deploy Kafka Connect

**Kafka Connect** is a tool for streaming data between Apache Kafka and external systems.

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.

Kafka Connect is typically used to integrate Kafka with external databases and storage and



messaging systems.

The procedures in this section show how to:

- [Deploy a Kafka Connect cluster using a `KafkaConnect` resource](#)
- [Run multiple Kafka Connect instances](#)
- [Create a Kafka Connect image containing the connectors you need to make your connection](#)
- [Create and manage connectors using a `KafkaConnector` resource or the Kafka Connect REST API](#)
- [Deploy a `KafkaConnector` resource to Kafka Connect](#)

**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.

### 4.2.1. Deploying Kafka Connect to your Kubernetes cluster

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

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

The deployment uses a YAML file to provide the specification to create a `KafkaConnect` resource.

In this procedure, we use the example file provided with Strimzi:

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

For information about configuring the `KafkaConnect` resource (or the `KafkaConnectS2I` resource with Source-to-Image (S2I) support), see [Kafka Connect cluster configuration](#) in the *Using Strimzi* guide.

#### Prerequisites

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

#### Procedure

1. Deploy Kafka Connect to your Kubernetes cluster. For a Kafka cluster with 3 or more brokers, use the `examples/connect/kafka-connect.yaml` file. For a Kafka cluster with less than 3 brokers, use the `examples/connect/kafka-connect-single-node-kafka.yaml` file.

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

2. Verify that Kafka Connect was successfully deployed:

```
kubectl get deployments
```

### 4.2.2. Kafka Connect configuration for multiple instances

If you are running multiple instances of Kafka Connect, you have to change the default configuration of the following `config` properties:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaConnect
metadata:
  name: my-connect
spec:
  # ...
  config:
    group.id: connect-cluster ①
    offset.storage.topic: connect-cluster-offsets ②
    config.storage.topic: connect-cluster-configs ③
    status.storage.topic: connect-cluster-status ④
    # ...
  # ...
```

- ① Kafka Connect cluster group that the instance belongs to.
- 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.

#### NOTE

Values for the three topics must be the same for all Kafka Connect instances with the same `group.id`.

Unless you change the default settings, each Kafka Connect instance connecting to the same Kafka cluster is deployed with the same values. What happens, in effect, is all instances are coupled to run in a cluster and use the same topics.

If multiple Kafka Connect clusters try to use the same topics, Kafka Connect will not work as expected and generate errors.

If you wish to run multiple Kafka Connect instances, change the values of these properties for each instance.

### 4.2.3. Extending Kafka Connect with connector plug-ins

The Strimzi container images for Kafka Connect include two built-in file connectors for moving file-based data into and out of your Kafka cluster.

*Table 1. File connectors*

File Connector	Description
<code>FileStreamSourceConnector</code>	Transfers data to your Kafka cluster from a file (the source).
<code>FileStreamSinkConnector</code>	Transfers data from your Kafka cluster to a file (the sink).

The procedures in this section show how to add your own connector classes to connector images by:

- [Creating a new container image automatically using Strimzi](#)
- [Creating a container image from the Kafka Connect base image \(manually or using continuous integration\)](#)
- [Creating a container image using OpenShift builds and Source-to-Image \(S2I\) \(available only on OpenShift\)](#)

#### IMPORTANT

You create the configuration for connectors directly [using the Kafka Connect REST API or KafkaConnector custom resources](#).

### Creating a new container image automatically using Strimzi

This procedure shows how to 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 will automatically download and add 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/v1beta1
kind: KafkaConnect
metadata:
  name: my-connect-cluster
spec: ❶
  #...
  build:
    output: ❷
    type: docker
    image: my-registry.io/my-org/my-connect-cluster:latest
    pushSecret: my-registry-credentials
    plugins: ❸
      - name: debezium-postgres-connector
        artifacts:
          - type: tgz
            url: https://repo1.maven.org/maven2/io/debezium/debezium-connector-
postgres/1.3.1.Final/debezium-connector-postgres-1.3.1.Final-plugin.tar.gz
            sha512sum:
962a12151bdf9a5a30627eebac739955a4fd95a08d373b86bdcea2b4d0c27dd6e1edd5cb548045e115e
33a9e69b1b2a352bee24df035a0447cb820077af00c03
          - name: camel-telegram
            artifacts:
              - type: tgz
                url:
https://repo.maven.apache.org/maven2/org/apache/camel/kafkaconnector/camel-
telegram-kafka-connector/0.7.0/camel-telegram-kafka-connector-0.7.0-package.tar.gz
                sha512sum:
a9b1ac63e3284bea7836d7d24d84208c49cdf5600070e6bd1535de654f6920b74ad950d51733e8020bf
4187870699819f54ef5859c7846ee4081507f48873479
              #...

```

❶ [The specification for the Kafka Connect cluster.](#)

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

❸ (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-CONFIG-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 the **KafkaConnector** custom resources to use the connector plugins you added.

#### *Additional resources*

See the *Using Strimzi* guide for more information on:

- [Kafka Connect Build schema reference](#)

## Creating a Docker image from the Kafka Connect base image

This procedure shows how to create a custom image and add it 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 plug-ins.

At startup, the Strimzi version of Kafka Connect loads any third-party connector plug-ins contained in the `/opt/kafka/plugins` directory.

### Prerequisites

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

### Procedure

1. Create a new `Dockerfile` using `quay.io/strimzi/kafka:0.21.1-kafka-2.7.0` as the base image:

```
FROM quay.io/strimzi/kafka:0.21.1-kafka-2.7.0
USER root:root
COPY ./my-plugins/ /opt/kafka/plugins/
USER 1001
```

### Example plug-in file

```
$ tree ./my-plugins/
./my-plugins/
├── debezium-connector-mongodb
│   ├── bson-3.4.2.jar
│   ├── CHANGELOG.md
│   ├── CONTRIBUTE.md
│   ├── COPYRIGHT.txt
│   ├── debezium-connector-mongodb-0.7.1.jar
│   ├── debezium-core-0.7.1.jar
│   ├── LICENSE.txt
│   ├── mongodb-driver-3.4.2.jar
│   ├── mongodb-driver-core-3.4.2.jar
│   └── README.md
├── debezium-connector-mysql
│   ├── CHANGELOG.md
│   ├── CONTRIBUTE.md
│   ├── COPYRIGHT.txt
│   ├── debezium-connector-mysql-0.7.1.jar
│   ├── debezium-core-0.7.1.jar
│   ├── LICENSE.txt
│   ├── mysql-binlog-connector-java-0.13.0.jar
│   ├── mysql-connector-java-5.1.40.jar
│   ├── README.md
│   └── wkb-1.0.2.jar
└── debezium-connector-postgres
    ├── CHANGELOG.md
    ├── CONTRIBUTE.md
    ├── COPYRIGHT.txt
    ├── debezium-connector-postgres-0.7.1.jar
    ├── debezium-core-0.7.1.jar
    ├── LICENSE.txt
    ├── postgresql-42.0.0.jar
    ├── protobuf-java-2.6.1.jar
    └── README.md
```

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

You can either:

- Edit the `KafkaConnect.spec.image` property of the `KafkaConnect` custom resource.

If set, this property overrides the `STRIMZI_KAFKA_CONNECT_IMAGES` variable in the Cluster Operator.

```

apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaConnect
metadata:
  name: my-connect-cluster
spec: ❶
  #...
  image: my-new-container-image ❷
  config: ❸
  #...

```

❶ The specification for the Kafka Connect cluster.

❷ The docker image for the pods.

❸ Configuration of the Kafka Connect *workers* (not connectors).

or

- In the `install/cluster-operator/060-Deployment-strimzi-cluster-operator.yaml` file, edit the `STRIMZI_KAFKA_CONNECT_IMAGES` variable to point to the new container image, and then reinstall the Cluster Operator.

#### Additional resources

See the *Using Strimzi* guide for more information on:

- [Container image configuration and the `KafkaConnect.spec.image` property](#)
- [Cluster Operator configuration and the `STRIMZI\_KAFKA\_CONNECT\_IMAGES` variable](#)

## Creating a container image using OpenShift builds and Source-to-Image

This procedure shows how to use OpenShift [builds](#) and the [Source-to-Image \(S2I\)](#) framework to create a new container image.

An OpenShift build takes a builder image with S2I support, together with source code and binaries provided by the user, and uses them to build a new container image. Once built, container images are stored in OpenShift's local container image repository and are available for use in deployments.

A Kafka Connect builder image with S2I support is provided on the [Container Registry](#) as part of the `quay.io/strimzi/kafka:0.21.1-kafka-2.7.0` image. This S2I image takes your binaries (with plug-ins and connectors) and stores them in the `/tmp/kafka-plugins/s2i` directory. It creates a new Kafka Connect image from this directory, which can then be used with the Kafka Connect deployment. When started using the enhanced image, Kafka Connect loads any third-party plug-ins from the `/tmp/kafka-plugins/s2i` directory.

#### Procedure

1. On the command line, use the `oc apply` command to create and deploy a Kafka Connect S2I cluster:

```
oc apply -f examples/connect/kafka-connect-s2i.yaml
```

2. Create a directory with Kafka Connect plug-ins:

```
$ tree ./my-plugins/
./my-plugins/
├── debezium-connector-mongodb
│   ├── bson-3.4.2.jar
│   ├── CHANGELOG.md
│   ├── CONTRIBUTE.md
│   ├── COPYRIGHT.txt
│   ├── debezium-connector-mongodb-0.7.1.jar
│   ├── debezium-core-0.7.1.jar
│   ├── LICENSE.txt
│   ├── mongodb-driver-3.4.2.jar
│   ├── mongodb-driver-core-3.4.2.jar
│   └── README.md
├── debezium-connector-mysql
│   ├── CHANGELOG.md
│   ├── CONTRIBUTE.md
│   ├── COPYRIGHT.txt
│   ├── debezium-connector-mysql-0.7.1.jar
│   ├── debezium-core-0.7.1.jar
│   ├── LICENSE.txt
│   ├── mysql-binlog-connector-java-0.13.0.jar
│   ├── mysql-connector-java-5.1.40.jar
│   ├── README.md
│   └── wkb-1.0.2.jar
└── debezium-connector-postgres
    ├── CHANGELOG.md
    ├── CONTRIBUTE.md
    ├── COPYRIGHT.txt
    ├── debezium-connector-postgres-0.7.1.jar
    ├── debezium-core-0.7.1.jar
    ├── LICENSE.txt
    ├── postgresql-42.0.0.jar
    ├── protobuf-java-2.6.1.jar
    └── README.md
```

3. Use the `oc start-build` command to start a new build of the image using the prepared directory:

```
oc start-build my-connect-cluster-connect --from-dir ./my-plugins/
```

**NOTE**

The name of the build is the same as the name of the deployed Kafka Connect cluster.

4. When the build has finished, the new image is used automatically by the Kafka Connect deployment.



## 4.2.4. Creating and managing connectors

When you have created a container image for your connector plug-in, you need to create a connector instance in your Kafka Connect cluster. You can then configure, monitor, and manage a running connector instance.

A connector is an instance of a particular *connector class* that knows how to communicate with the relevant external system in terms of messages. Connectors are available for many external systems, or you can create your own.

You can create *source* and *sink* types of connector.

### 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.

Strimzi provides two APIs for creating and managing connectors:

- `KafkaConnector` resources (referred to as `KafkaConnectors`)
- Kafka Connect REST API

Using the APIs, you can:

- Check the status of a connector instance
- Reconfigure a running connector
- Increase or decrease the number of tasks for a connector instance
- Restart failed tasks (not supported by `KafkaConnector` resource)
- Pause a connector instance
- Resume a previously paused connector instance
- Delete a connector instance

### `KafkaConnector` resources

`KafkaConnectors` allow you to create and manage connector instances for Kafka Connect in a Kubernetes-native way, so an HTTP client such as cURL is not required. Like other Kafka resources, you declare a connector's desired state in a `KafkaConnector` YAML file that is deployed to your Kubernetes cluster to create the connector instance.

You manage a running connector instance by updating its corresponding `KafkaConnector`, and then applying the updates. You remove a connector by deleting its corresponding `KafkaConnector`.

To ensure compatibility with earlier versions of Strimzi, `KafkaConnectors` are disabled by default. To enable them for a Kafka Connect cluster, you must use annotations on the `KafkaConnect` resource.

For instructions, see [Configuring Kafka Connect](#) in the *Using Strimzi* guide.

When `KafkaConnectors` are enabled, the Cluster Operator begins to watch for them. It updates the configurations of running connector instances to match the configurations defined in their `KafkaConnectors`.

Strimzi includes an example `KafkaConnector`, named `examples/connect/source-connector.yaml`. You can use this example to create and manage a `FileStreamSourceConnector`.

### Availability of the Kafka Connect REST API

The Kafka Connect REST API is available on port 8083 as the `<connect-cluster-name>-connect-api` service.

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

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

## 4.2.5. Deploying a `KafkaConnector` resource to Kafka Connect

This procedure describes how to deploy the example `KafkaConnector` to a Kafka Connect cluster.

The example YAML will create a `FileStreamSourceConnector` to send each line of the license file to Kafka as a message in a topic named `my-topic`.

### Prerequisites

- A Kafka Connect deployment in which `KafkaConnectors` are [enabled](#)
- A running Cluster Operator

### Procedure

1. Edit the `examples/connect/source-connector.yaml` file:

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: KafkaConnector
metadata:
  name: my-source-connector ①
  labels:
    strimzi.io/cluster: my-connect-cluster ②
spec:
  class: org.apache.kafka.connect.file.FileStreamSourceConnector ③
  tasksMax: 2 ④
  config: ⑤
    file: "/opt/kafka/LICENSE"
    topic: my-topic
    # ...
```

- ① Enter a name for the `KafkaConnector` resource. This will be used as the name of the connector within Kafka Connect. You can choose any name that is valid for a Kubernetes resource.

- ② Enter the name of the Kafka Connect cluster in which to create the connector.
- ③ The name or alias of the connector class. This should be present in the image being used by the Kafka Connect cluster.
- ④ The maximum number of tasks that the connector can create.
- ⑤ Configuration settings for the connector. Available configuration options depend on the connector class.

2. Create the `KafkaConnector` in your Kubernetes cluster:

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

3. Check that the resource was created:

```
kubectl get kctr --selector strimzi.io/cluster=my-connect-cluster -o name
```

## 4.3. Deploy Kafka MirrorMaker

The Cluster Operator deploys one or more Kafka MirrorMaker replicas to replicate data between Kafka clusters. This process is called mirroring to avoid confusion with the Kafka partitions replication concept. MirrorMaker consumes messages from the source cluster and republishes those messages to the target cluster.

### 4.3.1. Deploying Kafka MirrorMaker to your Kubernetes cluster

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

The deployment uses a YAML file to provide the specification to create a `KafkaMirrorMaker` or `KafkaMirrorMaker2` resource depending on the version of MirrorMaker deployed.

In this procedure, we use the example files provided with Strimzi:

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

For information about configuring `KafkaMirrorMaker` or `KafkaMirrorMaker2` resources, see [Kafka MirrorMaker cluster configuration](#) in the *Using Strimzi* guide.

#### Prerequisites

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

#### Procedure

1. Deploy Kafka MirrorMaker to your Kubernetes cluster:

For MirrorMaker:

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

For MirrorMaker 2.0:

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

2. Verify that MirrorMaker was successfully deployed:

```
kubectl get deployments
```

## 4.4. Deploy Kafka Bridge

The Cluster Operator deploys one or more Kafka bridge replicas to send data between Kafka clusters and clients via HTTP API.

### 4.4.1. Deploying Kafka Bridge to your Kubernetes cluster

This procedure shows how to deploy a Kafka 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.

In this procedure, we use the example file provided with Strimzi:

- `examples/bridge/kafka-bridge.yaml`

For information about configuring the `KafkaBridge` resource, see [Kafka Bridge cluster configuration](#) in the *Using Strimzi* guide.

#### *Prerequisites*

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

#### *Procedure*

1. Deploy Kafka Bridge to your Kubernetes cluster:

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

2. Verify that Kafka Bridge was successfully deployed:

```
kubectl get deployments
```

# Chapter 5. Setting up client access to the Kafka cluster

After you have [deployed Strimzi](#), the procedures in this section explain how to:

- Deploy example producer and consumer clients, which you can use to verify your deployment
- Set up external client access to the Kafka cluster

The steps to set up access to the Kafka cluster for a client outside Kubernetes are more complex, and require familiarity with the [Kafka component configuration procedures](#) described in the *Using Strimzi* guide.

## 5.1. Deploying example clients

This procedure shows how to deploy example producer and consumer clients that use the Kafka cluster you created to send and receive messages.

### Prerequisites

- The Kafka cluster is available for the clients.

### Procedure

1. Deploy a Kafka producer.

```
kubectl run kafka-producer -ti --image=quay.io/strimzi/kafka:0.21.1-kafka-2.7.0
--rm=true --restart=Never -- bin/kafka-console-producer.sh --broker-list cluster-
name-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.

```
kubectl run kafka-consumer -ti --image=quay.io/strimzi/kafka:0.21.1-kafka-2.7.0
--rm=true --restart=Never -- bin/kafka-console-consumer.sh --bootstrap-server
cluster-name-kafka-bootstrap:9092 --topic my-topic --from-beginning
```

5. Confirm that you see the incoming messages in the consumer console.

## 5.2. Setting up access for clients outside of Kubernetes

This procedure shows how to configure client access to a Kafka cluster from outside Kubernetes.

Using the address of the Kafka cluster, you can provide external access to a client on a different Kubernetes namespace or outside Kubernetes entirely.

You configure an external Kafka listener to provide the access.

The following external listener types are supported:

- `route` to use OpenShift `Route` and the default HAProxy router
- `loadbalancer` to use loadbalancer services
- `nodeport` to use ports on Kubernetes nodes
- `ingress` to use Kubernetes *Ingress* and the [NGINX Ingress Controller for Kubernetes](#)

The type chosen depends on your requirements, and your environment and infrastructure. For example, loadbalancers might not be suitable for certain infrastructure, such as bare metal, where node ports provide a better option.

In this procedure:

1. An external listener is configured for the Kafka cluster, with TLS encryption and authentication, and Kafka *simple authorization* is enabled.
2. A `KafkaUser` is created for the client, with TLS authentication and Access Control Lists (ACLs) defined for *simple authorization*.

You can configure your listener to use TLS or SCRAM-SHA-512 authentication, both of which can be used with TLS encryption. If you are using an authorization server, you can use token-based [OAuth 2.0 authentication](#) and [OAuth 2.0 authorization](#). Open Policy Agent (OPA) authorization is also supported as a [Kafka authorization](#) option.

When you configure the `KafkaUser` authentication and authorization mechanisms, ensure they match the equivalent Kafka configuration:

- `KafkaUser.spec.authentication` matches `Kafka.spec.kafka.listeners[*].authentication`
- `KafkaUser.spec.authorization` matches `Kafka.spec.kafka.authorization`

You should have at least one listener supporting the authentication you want to use for the `KafkaUser`.

**NOTE**

Authentication between Kafka users and Kafka brokers depends on the authentication settings for each. For example, it is not possible to authenticate a user with TLS if it is not also enabled in the Kafka configuration.

Strimzi operators automate the configuration process:

- 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.

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 Certificate Authority](#).

Certificates are available in PKCS #12 format (.p12) and PEM (.crt) formats.

### Prerequisites

- The Kafka cluster is available for the client
- The Cluster Operator and User Operator are running in the cluster
- A client outside the Kubernetes cluster to connect to the Kafka cluster

### Procedure

1. Configure the Kafka cluster with an **external** Kafka listener.
  - Define the authentication required to access the Kafka broker through the listener
  - Enable authorization on the Kafka broker

For example:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
metadata:
  name: my-cluster
  namespace: myproject
spec:
  kafka:
    # ...
    listeners: ①
    - name: external ②
      port: 9094 ③
      type: LISTENER-TYPE ④
      tls: true ⑤
      authentication:
        type: tls ⑥
      configuration:
        preferredNodePortAddressType: InternalDNS ⑦
        bootstrap and broker service overrides ⑧
      #...
    authorization: ⑨
      type: simple
      superUsers:
        - super-user-name ⑩
    # ...
```

- 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`, `loadbalancer`, `nodeport` or `ingress`. An internal listener is specified as `internal`.
- ⑤ Enables TLS encryption on the listener. Default is `false`. TLS encryption is not required for `route` listeners.
- ⑥ Authentication specified as `tls`.
- ⑦ (Optional, for `nodeport` listeners only) Configuration to [specify a preference for the first address type used by Strimzi as the node address](#).
- ⑧ (Optional) Strimzi automatically determines the addresses to advertise to clients. The addresses are automatically assigned by Kubernetes. You can override [bootstrap and broker service addresses](#) if the infrastructure on which you are running Strimzi does not provide the right address. Validation is not performed on the overrides. The override configuration differs according to the listener type. For example, you can override hosts for `route`, DNS names or IP addresses for `loadbalancer`, and node ports for `nodeport`.
- ⑨ Authoization specified as `simple`, which uses the `AclAuthorizer` Kafka plugin.
- ⑩ (Optional) Super users can access all brokers regardless of any access restrictions defined in ACLs.

#### WARNING

An OpenShift Route address comprises the name of the Kafka cluster, the name of the listener, and the name of the namespace it is created in. For example, `my-cluster-kafka-listener1-bootstrap-myproject` (`CLUSTER-NAME-kafka-LISTENER-NAME-bootstrap-NAMESPACE`). If you are using a `route` listener type, be careful that the whole length of the address does not exceed a maximum limit of 63 characters.

## 2. Create or update the `Kafka` resource.

```
kubectl apply -f KAFKA-CONFIG-FILE
```

The Kafka cluster is configured with a Kafka broker listener using TLS 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 with the same name as the `Kafka` resource.

## 3. Find the bootstrap address and port from the status of the `Kafka` resource.

```
kubectl get kafka KAFKA-CLUSTER-NAME -o
jsonpath='{.status.listeners[?(@.type=="external")].bootstrapServers}'
```



Use the bootstrap address in your Kafka client to connect to the Kafka cluster.

4. Extract the public cluster CA certificate and password from the generated `KAFKA-CLUSTER-NAME-cluster-ca-cert` Secret.

```
kubect1 get secret KAFKA-CLUSTER-NAME-cluster-ca-cert -o jsonpath='{.data.ca\.p12}'  
| base64 -d > ca.p12
```

```
kubect1 get secret KAFKA-CLUSTER-NAME-cluster-ca-cert -o  
jsonpath='{.data.ca\.password}' | base64 -d > ca.password
```

Use the certificate and password in your Kafka client to connect to the Kafka cluster with TLS encryption.

**NOTE**

Cluster CA certificates renew automatically by default. If you are using your own Kafka listener certificates, you will need to [renew the certificates manually](#).

5. 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.

For example:

```

apiVersion: kafka.strimzi.io/v1beta1
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
          operation: Read
      - resource:
          type: topic
          name: my-topic
          patternType: literal
          operation: Describe
      - resource:
          type: group
          name: my-group
          patternType: literal
          operation: Read

```

① The label must match the label of the Kafka cluster for the user to be created.

② Authentication specified as `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`).

6. Create or modify the `KafkaUser` resource.

```
kubectl apply -f USER-CONFIG-FILE
```

The user is created, as well as a Secret with the same name as the `KafkaUser` resource. The Secret contains a private and public key for TLS client authentication.

For example:

```

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-OF-THE-CLIENT-CA
  user.crt: USER-CERTIFICATE-CONTAINING-PUBLIC-KEY-OF-USER
  user.key: PRIVATE-KEY-OF-USER
  user.p12: P12-ARCHIVE-FILE-STORING-CERTIFICATES-AND-KEYS
  user.password: PASSWORD-PROTECTING-P12-ARCHIVE

```

7. Configure your client to connect to the Kafka cluster with the properties required to make a secure connection to the Kafka cluster.

a. Add the authentication details for the public cluster certificates:

```

security.protocol: SSL ❶
ssl.truststore.location: PATH-TO/ssl/keys/truststore ❷
ssl.truststore.password: CLUSTER-CA-CERT-PASSWORD ❸
ssl.truststore.type=PKCS12 ❹

```

❶ Enables TLS encryption (with or without TLS client authentication).

❷ Specifies the truststore location where the certificates were imported.

❸ Specifies the password for accessing the truststore. This property can be omitted if it is not needed by the truststore.

❹ Identifies the truststore type.

#### NOTE

Use `security.protocol: SASL_SSL` when using SCRAM-SHA authentication over TLS.

b. Add the bootstrap address and port for connecting to the Kafka cluster:

```
bootstrap.servers: BOOTSTRAP-ADDRESS:PORT
```

c. Add the authentication details for the public user certificates:

```

ssl.keystore.location: PATH-TO/ssl/keys/user1.keystore ❶
ssl.keystore.password: USER-CERT-PASSWORD ❷

```

❶ Specifies the keystore location where the certificates were imported.

❷ Specifies the password for accessing the keystore. This property can be omitted if it is not

needed by the keystore.

The public user certificate is signed by the client CA when it is created.

# Chapter 6. Introducing Metrics to Kafka

This section describes how to monitor your Strimzi deployment.

Depending on your requirements, you can:

- [Set up and deploy Prometheus and Grafana](#)
- [Configure the `Kafka` resource to deploy Kafka Exporter with your Kafka cluster](#)

Kafka Exporter provides additional monitoring related to consumer lag.

With Prometheus and Grafana set up, you can use the example Grafana dashboards provided by Strimzi for monitoring.

Additionally, you can configure your deployment to track messages end-to-end by [setting up distributed tracing](#), as described in the *Using Strimzi* guide.

*Additional resources*

- [Prometheus documentation](#)
- [Grafana documentation](#)
- [Apache Kafka Monitoring](#) describes JMX metrics exposed by Apache Kafka
- [ZooKeeper JMX](#) describes JMX metrics exposed by Apache ZooKeeper

## 6.1. Example metrics files

You can find example Grafana dashboards and other metrics configuration files in the [examples/metrics](#) directory.

```
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.json
│   ├── strimzi-operators.json
│   └── strimzi-zookeeper.json
├── grafana-install
│   └── grafana.yaml ②
├── prometheus-additional-properties
│   └── prometheus-additional.yaml ③
├── prometheus-alertmanager-config
│   └── alert-manager-config.yaml ④
├── prometheus-install
│   ├── alert-manager.yaml ⑤
│   ├── prometheus-rules.yaml ⑥
│   ├── prometheus.yaml ⑦
│   └── strimzi-pod-monitor.yaml ⑧
├── kafka-bridge-metrics.yaml ⑨
├── kafka-connect-metrics.yaml ⑩
├── kafka-cruise-control-metrics.yaml ⑪
├── kafka-metrics.yaml ⑫
└── kafka-mirror-maker-2-metrics.yaml ⑬
```

- ① Grafana dashboards.
- ② Installation file for the Grafana image.
- ③ 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.
- ④ Hook definitions for sending notifications through Alertmanager.
- ⑤ Resources for deploying and configuring Alertmanager.
- ⑥ Alerting rules examples for use with Prometheus Alertmanager (deployed with Prometheus).
- ⑦ Installation resource file for the Prometheus image.
- ⑧ PodMonitor definitions translated by the Prometheus Operator into jobs for the Prometheus server to be able to scrape metrics data directly from pods.
- ⑨ Kafka Bridge resource with metrics enabled.
- ⑩ Metrics configuration that defines Prometheus JMX Exporter relabeling rules for Kafka Connect.
- ⑪ Metrics configuration that defines Prometheus JMX Exporter relabeling rules for Cruise Control.
- ⑫ Metrics configuration that defines Prometheus JMX Exporter relabeling rules for Kafka and ZooKeeper.
- ⑬ Metrics configuration that defines Prometheus JMX Exporter relabeling rules for Kafka Mirror

### 6.1.1. Example Grafana dashboards

Example Grafana dashboards are provided for monitoring:

- Strimzi Operators
- Kafka
- Kafka ZooKeeper
- Kafka Connect
- Kafka MirrorMaker 2.0
- [Kafka Bridge](#)
- [Cruise Control](#)

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.

### 6.1.2. Example Prometheus metrics configuration

Strimzi uses the [Prometheus JMX Exporter](#) to expose JMX metrics using an HTTP endpoint, which is then scraped by the Prometheus server.

Grafana dashboards are dependent on Prometheus JMX Exporter relabeling rules, which are defined for Strimzi components as 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.

Strimzi provides example custom resource configuration YAML files with relabeling rules. When deploying Prometheus metrics configuration, you can can deploy the example custom resource or copy the metrics configuration to your own custom resource definition.

*Table 2. Example custom resources with metrics configuration*

Component	Custom resource	Example YAML file
Kafka and ZooKeeper	<a href="#">Kafka</a>	<a href="#">kafka-metrics.yaml</a>
Kafka Connect	<a href="#">KafkaConnect</a> and <a href="#">KafkaConnectS2I</a>	<a href="#">kafka-connect-metrics.yaml</a>
Kafka MirrorMaker 2.0	<a href="#">KafkaMirrorMaker2</a>	<a href="#">kafka-mirror-maker-2-metrics.yaml</a>
Kafka Bridge	<a href="#">KafkaBridge</a>	<a href="#">kafka-bridge-metrics.yaml</a>
Cruise Control	<a href="#">Kafka</a>	<a href="#">kafka-cruise-control-metrics.yaml</a>

#### *Additional resources*

For more information on the use of relabeling, see [Configuration](#) in the Prometheus

documentation.

## 6.2. Add Prometheus and Grafana

You can use Prometheus to provide monitoring data for the example Grafana dashboards provided with Strimzi.

In order to run the example Grafana dashboards, you must:

1. [Add metrics configuration to your Kafka cluster resource](#)
2. [Deploy Prometheus and Prometheus Alertmanager](#)
3. [Deploy Grafana](#)

### NOTE

The resources referenced in this section are intended as a starting point for setting up monitoring, but they are provided as examples only. If you require further support on configuring and running Prometheus or Grafana in production, try reaching out to their respective communities.

### 6.2.1. Deploying Prometheus metrics configuration

Strimzi provides [example custom resource configuration YAML files](#) with relabeling rules.

To apply metrics configuration of relabeling rules, do one of the following:

- [Copy the example configuration to your own custom resource definition](#)
- [Deploy the custom resource with the metrics configuration](#)

#### Copying Prometheus metrics configuration to a custom resource

To use Grafana dashboards for monitoring, copy [the example metrics configuration to a custom resource](#).

In this procedure, the **Kafka** resource is updated, but the procedure is the same for all components that support monitoring.

#### Procedure

Perform the following steps for each **Kafka** resource in your deployment.

1. Update the **Kafka** resource in an editor.

```
kubectl edit kafka KAFKA-CONFIG-FILE
```

2. Copy the [example configuration in kafka-metrics.yaml](#) to your own **Kafka** resource definition.
3. Save the file, and wait for the updated resource to be reconciled.



## Deploying a Kafka cluster with Prometheus metrics configuration

To use Grafana dashboards for monitoring, you can deploy [an example Kafka cluster with metrics configuration](#).

In this procedure, The `kafka-metrics.yaml` file is used for the `Kafka` resource.

### Procedure

- Deploy the Kafka cluster with the [example metrics configuration](#).

```
kubectl apply -f kafka-metrics.yaml
```

## 6.2.2. 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 [Discovering services](#).

### Prometheus configuration

Strimzi provides [example configuration files for the Prometheus server](#).

A Prometheus image is provided for deployment:

- `prometheus.yaml`

Additional Prometheus-related configuration is also provided in the following files:

- `prometheus-additional.yaml`
- `prometheus-rules.yaml`
- `strimzi-pod-monitor.yaml`

For Prometheus to obtain monitoring data:

- [Deploy the Prometheus Operator](#)

Then use the configuration files to:

- [Deploy Prometheus](#)

## Alerting rules

The `prometheus-rules.yaml` file provides [example alerting rule 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` that grants permissions to Prometheus to read the health endpoints exposed by the Kafka and ZooKeeper pods, cAdvisor and the kubelet for container metrics.
- 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.

On Linux, use:

```
curl -s https://raw.githubusercontent.com/coreos/prometheus-operator/master/bundle.yaml | sed -e '/[[:space:]]*namespace: [a-zA-Z0-9-]*$/s/namespace:[[:space:]]*[a-zA-Z0-9-]*$/namespace: my-namespace/' > prometheus-operator-deployment.yaml
```

On MacOS, use:

```
curl -s https://raw.githubusercontent.com/coreos/prometheus-operator/master/bundle.yaml | sed -e ' ' '/[[:space:]]*namespace: [a-zA-Z0-9-]*$/s/namespace:[[:space:]]*[a-zA-Z0-9-]*$/namespace: my-namespace/' > prometheus-operator-deployment.yaml
```

- Replace the example `namespace` with your own.
- 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+.

#### 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 apply -f prometheus-operator-deployment.yaml
```

## Deploying Prometheus

To obtain monitoring data in your Kafka cluster, you can use your own Prometheus deployment or deploy Prometheus by applying the [example installation resource file for the Prometheus docker image and the YAML files for Prometheus-related resources](#).

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 `strimzi-pod-monitor.yaml` 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, ZooKeeper, Operators, the Kafka 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** and the **prometheus-additional.yaml** file.

4. Deploy the Prometheus resources:

```
kubectl apply -f strimzi-pod-monitor.yaml  
kubectl apply -f prometheus-rules.yaml  
kubectl apply -f prometheus.yaml
```

### 6.2.3. Setting up Prometheus Alertmanager

**Prometheus Alertmanager** is a plugin 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.

#### Alertmanager configuration

Strimzi provides [example configuration files for Prometheus Alertmanager](#).

A configuration file defines the resources for deploying Alertmanager:

- **alert-manager.yaml**

An additional configuration file provides the hook definitions for sending notifications from your Kafka cluster.

- **alert-manager-config.yaml**

For Alertmanger to handle Prometheus alerts, use the configuration files to:

- [Deploy Alertmanager](#)

## Alerting rules

Alerting rules provide notifications about specific conditions observed in the 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.

Alertmanager can be configured to use email, chat messages or other notification methods.

### *Additional resources*

For more information about setting up alerting rules, see [Configuration](#) in the Prometheus documentation.

## Alerting rule examples

Example alerting rules for Kafka and ZooKeeper metrics are provided with Strimzi for use in a [Prometheus 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.
- A tick is a basic ZooKeeper time unit, which is measured in milliseconds and configured using the `tickTime` parameter of `Kafka.spec.zookeeper.config`. For example, if `ZooKeeper tickTime=3000`, 3 ticks (3 x 3000) equals 9000 milliseconds.
- The availability of the `ZookeeperRunningOutOfSpace` 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.

### *Kafka alerting rules*

#### **UnderReplicatedPartitions**

Gives the number of partitions for which the current broker is the lead replica but which have fewer replicas than the `min.insync.replicas` configured for their topic. This metric provides insights about brokers that host the follower replicas. Those followers are not keeping up with the leader. Reasons for this could include being (or having been) offline, and over-throttled interbroker replication. An alert is raised when this value is greater than zero, providing information on the under-replicated partitions for each broker.

#### **AbnormalControllerState**

Indicates whether the current broker is the controller for the cluster. The metric can be 0 or 1.

During the life of a cluster, only one broker should be the controller and the cluster always needs to have an active controller. Having two or more brokers saying that they are controllers indicates a problem. If the condition persists, an alert is raised when the sum of all the values for this metric on all brokers is not equal to 1, meaning that there is no active controller (the sum is 0) or more than one controller (the sum is greater than 1).

### **UnderMinIsrPartitionCount**

Indicates that the minimum number of in-sync replicas (ISRs) for a lead Kafka broker, specified using `min.insync.replicas`, that must acknowledge a write operation has not been reached. The metric defines the number of partitions that the broker leads for which the in-sync replicas count is less than the minimum in-sync. An alert is raised when this value is greater than zero, providing information on the partition count for each broker that did not achieve the minimum number of acknowledgments.

### **OfflineLogDirectoryCount**

Indicates the number of log directories which are offline (for example, due to a hardware failure) so that the broker cannot store incoming messages anymore. An alert is raised when this value is greater than zero, providing information on the number of offline log directories for each broker.

### **KafkaRunningOutOfSpace**

Indicates the remaining amount of disk space that can be used for writing data. An alert is raised when this value is lower than 5GiB, providing information on the disk that is running out of space for each persistent volume claim. The threshold value may be changed in `prometheus-rules.yaml`.

### *ZooKeeper alerting rules*

#### **AvgRequestLatency**

Indicates the amount of time it takes for the server to respond to a client request. An alert is raised when this value is greater than 10 (ticks), providing the actual value of the average request latency for each server.

#### **OutstandingRequests**

Indicates the number of queued requests in the server. This value goes up when the server receives more requests than it can process. An alert is raised when this value is greater than 10, providing the actual number of outstanding requests for each server.

#### **ZookeeperRunningOutOfSpace**

Indicates the remaining amount of disk space that can be used for writing data to ZooKeeper. An alert is raised when this value is lower than 5GiB., providing information on the disk that is running out of space for each persistent volume claim.

## **Deploying Alertmanager**

To deploy Alertmanager, apply the [example configuration files](#).

The sample configuration provided with Strimzi configures the Alertmanager to send notifications to a Slack channel.

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 configured for the Kafka cluster resource](#)
- [Prometheus is deployed](#)

#### Procedure

1. Create a **Secret** resource from the Alertmanager configuration file (**alert-manager-config.yaml** in the **examples/metrics/prometheus-alertmanager-config** directory):

```
kubectl apply -f alert-manager-config.yaml
```

2. Update the **alert-manager-config.yaml** file to replace the:
  - **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
3. Deploy Alertmanager:

```
kubectl apply -f alert-manager.yaml
```

## 6.2.4. Setting up Grafana

Grafana provides visualizations of Prometheus metrics.

You can deploy and enable the example Grafana dashboards provided with Strimzi.

### Deploying Grafana

To provide visualizations of Prometheus metrics, you can use your own Grafana installation or deploy Grafana by applying the **grafana.yaml** file provided in the **examples/metrics** directory.

#### Prerequisites

- [Metrics are configured for the Kafka cluster resource](#)
- [Prometheus and Prometheus Alertmanager are deployed](#)

#### Procedure

1. Deploy Grafana:

```
kubectl apply -f grafana.yaml
```

## 2. Enable the Grafana dashboards.

### Enabling the example Grafana dashboards

Strimzi provides [example dashboard configuration files for Grafana](#). Example dashboards are provided in the [examples/metrics](#) directory as JSON files:

- [strimzi-kafka.json](#)
- [strimzi-zookeeper.json](#)
- [strimzi-kafka-connect.json](#)
- [strimzi-kafka-mirror-maker-2.json](#)
- [strimzi-operators.json](#)
- [strimzi-kafka-bridge.json](#)
- [strimzi-cruise-control.json](#)

The example dashboards are a good starting point for monitoring key metrics, but they do not represent all available metrics. You can modify the example dashboards or add other metrics, depending on your infrastructure.

After setting up Prometheus and Grafana, you can visualize the Strimzi data on the Grafana dashboards.

**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.

**NOTE** The name of the Grafana pod is different for each user.

#### Procedure

1. 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.

2. Use [port-forward](#) to redirect the Grafana user interface to [localhost:3000](#):



```
kubectl port-forward svc/grafana 3000:3000
```

3. Point a web browser to <http://localhost:3000>.

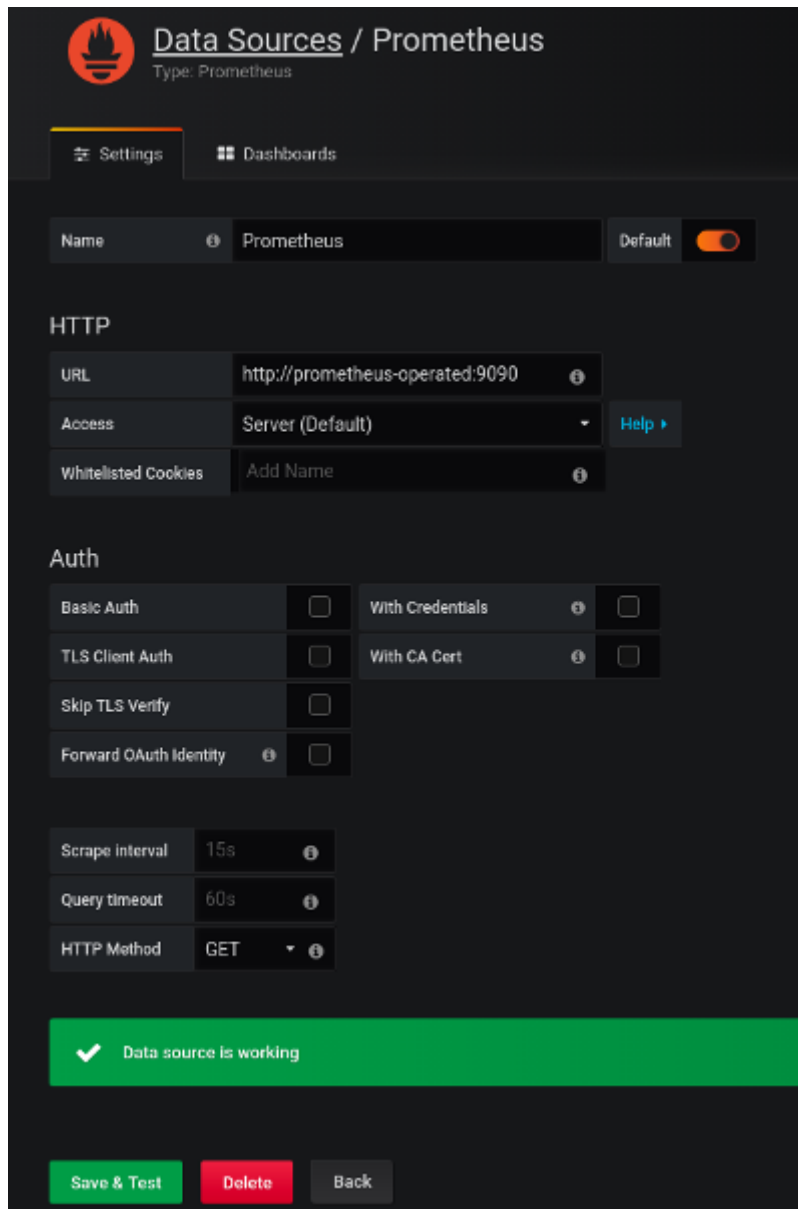
The Grafana Log In page appears.

4. 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.

5. Add Prometheus as a *data source*.
  - Specify a name
  - Add *Prometheus* as the type
  - Specify a Prometheus server URL (<http://prometheus-operated:9090>)

Save and test the connection when you have added the details.



**Data Sources / Prometheus**  
Type: Prometheus

Settings Dashboards

Name Prometheus Default ☒

**HTTP**

URL <http://prometheus-operated:9090>

Access Server (Default) [Help](#)

Whitelisted Cookies Add Name

**Auth**

Basic Auth ☒ With Credentials ☒

TLS Client Auth ☒ With CA Cert ☒

Skip TLS Verify ☒

Forward OAuth Identity ☒

Scrape Interval 15s

Query timeout 60s

HTTP Method GET

✓ Data source is working

Save & Test Delete Back

6. From **Dashboards** › **Import**, upload the example dashboards or paste the JSON directly.
7. On the top header, click the dashboard drop-down menu, and then select the dashboard you want to view.

When the Prometheus server has been collecting metrics for a Strimzi cluster for some time, the dashboards are populated.

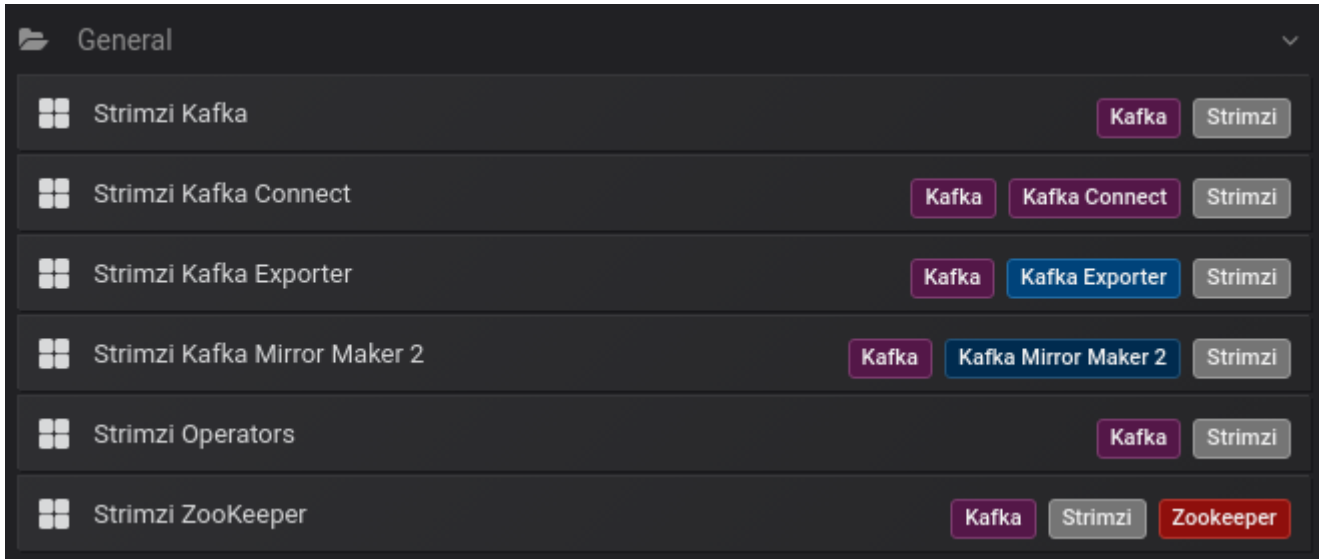


Figure 1. Dashboard selection options

### Strimzi Kafka

Shows metrics for:

- Brokers online count
- Active controllers in the cluster count
- Unclean leader election rate
- Replicas that are online
- Under-replicated partitions count
- Partitions which are at their minimum in sync replica count
- Partitions which are under their minimum in sync replica count
- Partitions that do not have an active leader and are hence not writable or readable
- Kafka broker pods memory usage
- Aggregated Kafka broker pods CPU usage
- Kafka broker pods disk usage
- JVM memory used
- JVM garbage collection time
- JVM garbage collection count
- Total incoming byte rate
- Total outgoing byte rate

- Incoming messages rate
- Total produce request rate
- Byte rate
- Produce request rate
- Fetch request rate
- Network processor average time idle percentage
- Request handler average time idle percentage
- Log size

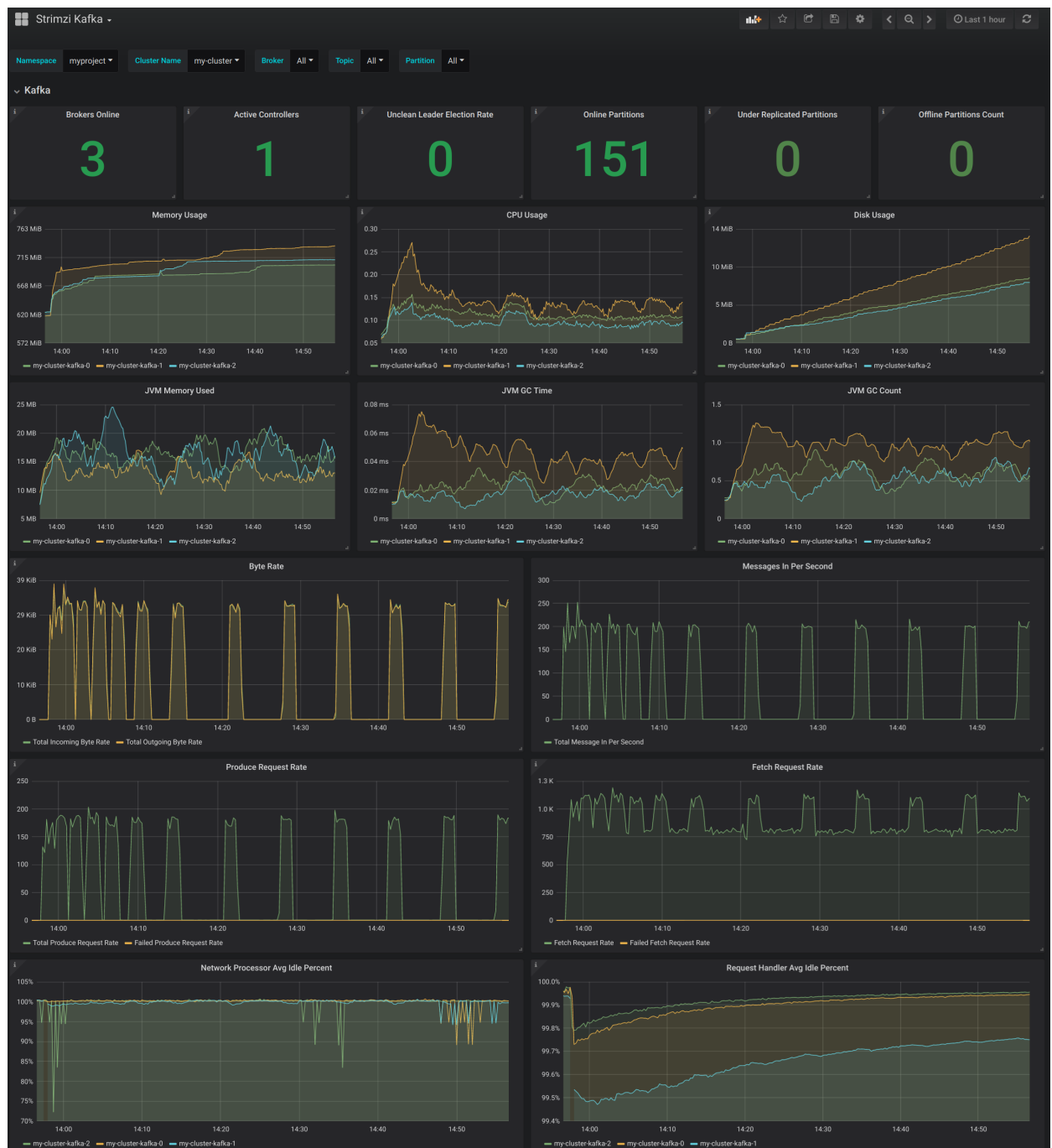


Figure 2. Strimzi Kafka dashboard

## Strimzi ZooKeeper

Shows metrics for:

- Quorum Size of Zookeeper ensemble
- Number of *alive* connections
- Queued requests in the server count
- Watchers count
- ZooKeeper pods memory usage
- Aggregated ZooKeeper pods CPU usage
- ZooKeeper pods disk usage
- JVM memory used
- JVM garbage collection time
- JVM garbage collection count
- Amount of time it takes for the server to respond to a client request (maximum, minimum and average)

## Strimzi Kafka Connect

Shows metrics for:

- Total incoming byte rate
- Total outgoing byte rate
- Disk usage
- JVM memory used
- JVM garbage collection time

## Strimzi Kafka MirrorMaker 2

Shows metrics for:

- Number of connectors
- Number of tasks
- Total incoming byte rate
- Total outgoing byte rate
- Disk usage
- JVM memory used
- JVM garbage collection time

## Strimzi Operators

Shows metrics for:

- Custom resources

- Successful custom resource reconciliations per hour
- Failed custom resource reconciliations per hour
- Reconciliations without locks per hour
- Reconciliations started hour
- Periodical reconciliations per hour
- Maximum reconciliation time
- Average reconciliation time
- JVM memory used
- JVM garbage collection time
- JVM garbage collection count

### 6.2.5. Using metrics with Minikube or Minishift

When adding Prometheus and Grafana servers to an Apache Kafka deployment using Minikube or Minishift, the memory available to the virtual machine should be increased (to 4 GB of RAM, for example, instead of the default 2 GB).

For information on how to increase the default amount of memory, see:

- [Installing a Kubernetes cluster](#)
- [Installing an OpenShift cluster](#)

#### *Additional resources*

- [Prometheus - Monitoring Docker Container Metrics using cAdvisor](#) describes how to use cAdvisor (short for container Advisor) metrics with Prometheus to analyze and expose resource usage (CPU, Memory, and Disk) and performance data from running containers within pods on Kubernetes.

## 6.3. Add Kafka Exporter

[Kafka Exporter](#) is an open source project to enhance monitoring of Apache Kafka brokers and clients. Kafka Exporter is provided with Strimzi for deployment with a Kafka cluster to extract additional metrics data from Kafka brokers related to offsets, consumer groups, consumer lag, and topics.

The metrics data is used, for example, to help identify slow consumers.

Lag data is exposed as Prometheus metrics, which can then be presented in Grafana for analysis.

If you are already using Prometheus and Grafana for monitoring of built-in Kafka metrics, you can configure Prometheus to also scrape the Kafka Exporter Prometheus endpoint.

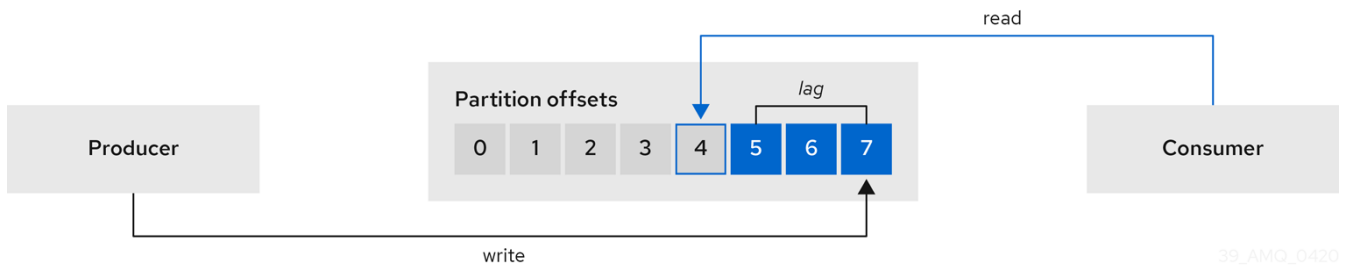
### 6.3.1. Monitoring Consumer lag

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*



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.

### The importance of monitoring 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

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.

### 6.3.2. Example Kafka Exporter alerting rules

If you performed the steps to introduce metrics to your deployment, you will already have your Kafka cluster configured to use the alert notification rules that support Kafka Exporter.

The rules for Kafka Exporter are defined in `prometheus-rules.yaml`, and are deployed with Prometheus. For more information, see [Prometheus](#).

The sample alert notification rules specific to Kafka Exporter are as follows:

#### UnderReplicatedPartition

An alert to warn that a topic is under-replicated and the broker is not replicating to enough partitions. The default configuration is for an alert if there are one or more under-replicated partitions for a topic. The alert might signify that a Kafka instance is down or the Kafka cluster is overloaded. A planned restart of the Kafka broker may be required to restart the replication process.

#### TooLargeConsumerGroupLag

An alert to warn that the lag on a consumer group is too large for a specific topic partition. The default configuration is 1000 records. A large lag might indicate that consumers are too slow and are falling behind the producers.

#### NoMessageForTooLong

An alert to warn that a topic has not received messages for a period of time. The default configuration for the time period is 10 minutes. The delay might be a result of a configuration issue preventing a producer from publishing messages to the topic.

Adapt the default configuration of these rules according to your specific needs.

#### Additional resources

- [Add Prometheus and Grafana](#)
- [Example metrics files](#)
- [Alerting rules](#)

### 6.3.3. Exposing Kafka Exporter metrics

Lag information is exposed by Kafka Exporter as Prometheus metrics for presentation in Grafana.

Kafka Exporter exposes metrics data for brokers, topics and consumer groups.

The data extracted is described here.

Table 3. Broker metrics output

Name	Information
kafka_brokers	Number of brokers in the Kafka cluster

Table 4. Topic metrics output

Name	Information
kafka_topic_partitions	Number of partitions for a topic
kafka_topic_partition_current_offset	Current topic partition offset for a broker
kafka_topic_partition_oldest_offset	Oldest topic partition offset for a broker
kafka_topic_partition_in_sync_replica	Number of in-sync replicas for a topic partition
kafka_topic_partition_leader	Leader broker ID of a topic partition

Name	Information
<code>kafka_topic_partition_leader_is_preferred</code>	Shows <b>1</b> if a topic partition is using the preferred broker
<code>kafka_topic_partition_replicas</code>	Number of replicas for this topic partition
<code>kafka_topic_partition_under_replicated_partition</code>	Shows <b>1</b> if a topic partition is under-replicated

Table 5. Consumer group metrics output

Name	Information
<code>kafka_consumergroup_current_offset</code>	Current topic partition offset for a consumer group
<code>kafka_consumergroup_lag</code>	Current approximate lag for a consumer group at a topic partition

### 6.3.4. Configuring Kafka Exporter

This procedure shows how to configure Kafka Exporter in the `Kafka` resource through `KafkaExporter` properties.

For more information about configuring the `Kafka` resource, see [Kafka cluster configuration](#) in the *Using Strimzi* guide.

The properties relevant to the Kafka Exporter configuration are shown in this procedure.

You can configure these properties as part of a deployment or redeployment of the Kafka cluster.

#### Prerequisites

- A Kubernetes cluster
- A running Cluster Operator

#### Procedure

1. Edit the `KafkaExporter` properties for the `Kafka` resource.

The properties you can configure are shown in this example configuration:



```

apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
metadata:
  name: my-cluster
spec:
  # ...
  kafkaExporter:
    image: my-org/my-image:latest ❶
    groupRegex: ".*" ❷
    topicRegex: ".*" ❸
    resources: ❹
      requests:
        cpu: 200m
        memory: 64Mi
      limits:
        cpu: 500m
        memory: 128Mi
    logging: debug ❺
    enableSaramaLogging: true ❻
    template: ❼
      pod:
        metadata:
          labels:
            label1: value1
        imagePullSecrets:
          - name: my-docker-credentials
        securityContext:
          runAsUser: 1000001
          fsGroup: 0
          terminationGracePeriodSeconds: 120
    readinessProbe: ❽
      initialDelaySeconds: 15
      timeoutSeconds: 5
    livenessProbe: ❾
      initialDelaySeconds: 15
      timeoutSeconds: 5
  # ...

```

- ❶ ADVANCED OPTION: Container image configuration, which is [recommended only in special situations](#).
- ❷ A regular expression to specify the consumer groups to include in the metrics.
- ❸ A regular expression to specify the topics to include in the metrics.
- ❹ [CPU and memory resources to reserve](#).
- ❺ Logging configuration, to log messages with a given severity (debug, info, warn, error, fatal) or above.
- ❻ Boolean to enable Sarama logging, a Go client library used by Kafka Exporter.
- ❼ [Customization of deployment templates and pods](#).

⑧ [Healthcheck readiness probes](#).

⑨ [Healthcheck liveness probes](#).

2. Create or update the resource:

```
kubectl apply -f kafka.yaml
```

#### *What to do next*

After configuring and deploying Kafka Exporter, you can [enable Grafana to present the Kafka Exporter dashboards](#).

#### *Additional resources*

[KafkaExporterTemplate](#) [schema reference](#).

### 6.3.5. Enabling the Kafka Exporter Grafana dashboard

Strimzi provides [example dashboard configuration files for Grafana](#). The Kafka Exporter dashboard is provided in the [examples/metrics](#) directory as a JSON file:

- [strimzi-kafka-exporter.json](#)

If you deployed Kafka Exporter with your Kafka cluster, you can visualize the metrics data it exposes on the Grafana dashboard.

#### *Prerequisites*

- Kafka is deployed with [Kafka Exporter metrics configuration](#)
- [Prometheus and Prometheus Alertmanager](#) are deployed to the Kafka cluster
- [Grafana is deployed to the Kafka cluster](#)

This procedure assumes you already have access to the Grafana user interface and Prometheus has been added as a data source. If you are accessing the user interface for the first time, see [Grafana](#).

#### *Procedure*

1. [Access the Grafana user interface](#).
2. Select the *Strimzi Kafka Exporter* dashboard.

When metrics data has been collected for some time, the Kafka Exporter charts are populated.

#### **Strimzi Kafka Exporter**

Shows metrics for:

- Topic count
- Partition count
- Replicas count
- In-sync replicas count

- Under-replicated partitions count
- Partitions which are at their minimum in sync replica count
- Partitions which are under their minimum in sync replica count
- Partitions not on a preferred node
- Messages in per second from topics
- Messages consumed per second from topics
- Messages consumed per minute by consumer groups
- Lag by consumer group
- Number of partitions
- Latest offsets
- Oldest offsets

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.

## 6.4. Monitor Kafka Bridge

If you are already using Prometheus and Grafana for monitoring of built-in Kafka metrics, you can configure Prometheus to also scrape the Kafka Bridge Prometheus endpoint.

The example Grafana dashboard for the Kafka Bridge provides:

- Information about HTTP connections and related requests to the different endpoints
- Information about the Kafka consumers and producers used by the bridge
- JVM metrics from the bridge itself

### 6.4.1. Configuring Kafka Bridge

You can enable the Kafka Bridge metrics in the `KafkaBridge` resource using the `enableMetrics` property.

You can configure this property as part of a deployment or redeployment of the Kafka Bridge.

For example:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaBridge
metadata:
  name: my-bridge
spec:
  # ...
  bootstrapServers: my-cluster-kafka:9092
  http:
    # ...
  enableMetrics: true
  # ...
```

### 6.4.2. Enabling the Kafka Bridge Grafana dashboard

If you deployed Kafka Bridge with your Kafka cluster, you can enable Grafana to present the metrics data it exposes.

A Kafka Bridge dashboard is provided in the `examples/metrics` directory as a JSON file:

- `strimzi-kafka-bridge.json`

When metrics data has been collected for some time, the Kafka Bridge charts are populated.

#### Kafka Bridge

Shows metrics for:

- HTTP connections to the Kafka Bridge count
- HTTP requests being processed count
- Requests processed per second grouped by HTTP method
- The total request rate grouped by response codes (2XX, 4XX, 5XX)
- Bytes received and sent per second
- Requests for each Kafka Bridge endpoint
- Number of Kafka consumers, producers, and related opened connections used by the Kafka Bridge itself
- Kafka producer:
  - The average number of records sent per second (grouped by topic)
  - The number of outgoing bytes sent to all brokers per second (grouped by topic)
  - The average number of records per second that resulted in errors (grouped by topic)
- Kafka consumer:
  - The average number of records consumed per second (grouped by clientId-topic)
  - The average number of bytes consumed per second (grouped by clientId-topic)
  - Partitions assigned (grouped by clientId)

- JVM memory used
- JVM garbage collection time
- JVM garbage collection count

## 6.5. Monitor Cruise Control

If you are already using Prometheus and Grafana for monitoring of built-in Kafka metrics, you can configure Prometheus to also scrape the Cruise Control Prometheus endpoint.

The example Grafana dashboard for Cruise Control provides:

- Information about optimization proposals computation, goals violation, cluster balancedness, and more
- Information about REST API calls for rebalance proposals and actual rebalance operations
- JVM metrics from Cruise Control itself

### 6.5.1. Configuring Cruise Control

Enable Cruise Control metrics using the `cruiseControl.metricsConfig` property in the `Kafka` resource to provide a reference to a ConfigMap that contains JMX exporter configuration for the metrics to expose.

For example:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
metadata:
  name: my-cluster
spec:
  # ...
  kafka:
    # ...
  zookeeper:
    # ...
  cruiseControl:
    metricsConfig:
      type: jmxPrometheusExporter
      valueFrom:
        configMapKeyRef:
          name: my-config-map
          key: my-key
```

### 6.5.2. Enabling the Cruise Control Grafana dashboard

If you deployed Cruise Control with your Kafka cluster with the metrics enabled, you can enable Grafana to present the metrics data it exposes.

A Cruise Control dashboard is provided in the `examples/metrics` directory as a JSON file:

- `strimzi-cruise-control.json`

When metrics data has been collected for some time, the Cruise Control charts are populated.

## Cruise Control

Shows metrics for:

- Number of snapshot windows that are monitored by Cruise Control
- Number of time windows considered valid because they contain enough samples to compute an optimization proposal
- Number of ongoing executions running for proposals or rebalances
- Current balancedness score of the Kafka cluster as calculated by the anomaly detector component of Cruise Control (every 5 minutes by default)
- Percentage of monitored partitions
- Number of goal violations reported by the anomaly detector (every 5 minutes by default)
- How often a disk read failure happens on the brokers
- Rate of metric sample fetch failures
- Time needed to compute an optimization proposal
- Time needed to create the cluster model
- How often a proposal request or an actual rebalance request is made through the Cruise Control REST API
- How often the overall cluster state and the user tasks state are requested through the Cruise Control REST API
- JVM memory used
- JVM garbage collection time
- JVM garbage collection count

# Chapter 7. Upgrading Strimzi

Strimzi can be upgraded with no cluster downtime. Each version of Strimzi supports one or more versions of Apache Kafka. You can upgrade to a higher Kafka version as long as it is supported by your version of Strimzi. In some cases, you can also downgrade to a lower supported Kafka version.

Newer versions of Strimzi may support newer versions of Kafka, but you need to upgrade Strimzi *before* you can upgrade to a higher supported Kafka version.

## IMPORTANT

If applicable, [Resource upgrades](#) *must* be performed after upgrading Strimzi and Kafka.

## 7.1. Strimzi and Kafka upgrades

Upgrading Strimzi is a two-stage process. To upgrade brokers and clients without downtime, you *must* complete the upgrade procedures in the following order:

1. Update your Cluster Operator to the latest Strimzi version.
  - [Upgrading the Cluster Operator](#)
2. Upgrade all Kafka brokers and client applications to the latest Kafka version.
  - [Upgrading Kafka](#)

### 7.1.1. Kafka versions

Kafka's log message format version and inter-broker protocol version specify the log format version appended to messages and the version of protocol used in a cluster. As a result, the upgrade process involves making configuration changes to existing Kafka brokers and code changes to client applications (consumers and producers) to ensure the correct versions are used.

The following table shows the differences between Kafka versions:

Kafka version	Interbroker protocol version	Log message format version	ZooKeeper version
2.5.0	2.5	2.5	3.5.7
2.5.1	2.5	2.5	3.5.8
2.6.0	2.6	2.6	3.5.8
2.7.0	2.7	2.7	3.5.8

#### *Message format version*

When a producer sends a message to a Kafka broker, the message is encoded using a specific format. The format can change between Kafka releases, so messages include a version identifying which version of the format they were encoded with. You can configure a Kafka broker to convert messages from newer format versions to a given older format version before the broker appends the message to the log.

In Kafka, there are two different methods for setting the message format version:

- The `message.format.version` property is set on topics.
- The `log.message.format.version` property is set on Kafka brokers.

The default value of `message.format.version` for a topic is defined by the `log.message.format.version` that is set on the Kafka broker. You can manually set the `message.format.version` of a topic by modifying its topic configuration.

The upgrade tasks in this section assume that the message format version is defined by the `log.message.format.version`.

### 7.1.2. Upgrading the Cluster Operator

The steps to upgrade your Cluster Operator deployment to use Strimzi 0.21.1 are outlined in this section.

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.

#### Upgrading the Cluster Operator to a later version

This procedure describes how to upgrade a Cluster Operator deployment to a later version.

##### Prerequisites

- An existing Cluster Operator deployment is available.
- You have [downloaded the installation files for the new version](#).

##### Procedure

1. Take note of any configuration changes made to the existing Cluster Operator resources (in the `/install/cluster-operator` directory). 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 the version of Strimzi you are upgrading to.
3. Update the Cluster Operator.
  - a. Modify the installation files for the new 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-strimzi-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 apply -f install/cluster-operator
```

Wait for the rolling updates to complete.

5. Get the image for the Kafka pod to ensure the upgrade 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, **NEW-STRIMZI-VERSION-kafka-CURRENT-KAFKA-VERSION**.

6. Update existing resources to handle deprecated custom resource properties.
  - [Strimzi resource upgrades](#)

Your Cluster Operator was upgraded to the later version., but the version of Kafka running in the cluster it manages is unchanged.

Following the Cluster Operator upgrade, you can perform a [Kafka upgrade](#).

### 7.1.3. Upgrading Kafka

After you have upgraded your Cluster Operator, you can upgrade your brokers to a higher supported version of Kafka.

Kafka upgrades are performed using the Cluster Operator. How the Cluster Operator performs an upgrade depends on the differences between versions of:

- Inter-broker protocol
- Log message format
- ZooKeeper

When the versions are the same for the current and target Kafka version, as is typically the case for a patch level upgrade, the Cluster Operator can upgrade through a single rolling update of the Kafka brokers.

When one or more of these versions differ, the Cluster Operator requires two or three rolling updates of the Kafka brokers to perform the upgrade.

#### *Additional resources*

- [Upgrading the Cluster Operator](#)

### **Kafka version and image mappings**

When upgrading Kafka, consider your settings for the `STRIMZI_KAFKA_IMAGES` and `Kafka.spec.kafka.version` properties.

- Each `Kafka` resource can be configured with a `Kafka.spec.kafka.version`.
- The Cluster Operator's `STRIMZI_KAFKA_IMAGES` environment variable provides a mapping between the Kafka version and the image to be used when that version is requested in a given `Kafka` resource.
  - 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.

### **Updating listener configuration**

Strimzi provides a `GenericKafkaListener` schema for the configuration of Kafka listeners in a `Kafka` resource.

`GenericKafkaListener` replaces the `KafkaListeners` schema, which is deprecated.

With the `GenericKafkaListener` schema, you can configure as many listeners as required, as long as their names and ports are unique. The `listeners` configuration is defined as an array, but the deprecated format is also supported.

For clients inside the Kubernetes cluster, you can create `plain` (without encryption) or `tls internal` listeners.

For clients outside the Kubernetes cluster, you create `external` listeners and specify a connection mechanism, which can be `nodeport`, `loadbalancer`, `ingress` or `route`.

The `KafkaListeners` schema uses sub-properties for `plain`, `tls` and `external` listeners, with fixed ports for each. After the Kafka upgrade, you can convert listeners configured using the `KafkaListeners` schema into the format of the `GenericKafkaListener` schema.

For example, if you are currently using the following configuration in your `Kafka` configuration:

### Old listener configuration

```
listeners:
  plain:
    # ...
  tls:
    # ...
  external:
    type: loadbalancer
    # ...
```

Convert the listeners into the new format using:

### New listener configuration

```
listeners:
  #...
  - name: plain
    port: 9092
    type: internal
    tls: false ①
  - name: tls
    port: 9093
    type: internal
    tls: true
  - name: external
    port: 9094
    type: EXTERNAL-LISTENER-TYPE ②
    tls: true
```

① The TLS property is now required for all listeners.

② Options: `ingress`, `loadbalancer`, `nodeport`, `route`.

Make sure to use the **exact** names and port numbers shown.

For any additional `configuration` or `overrides` properties used with the old format, you need to update them to the new format.

Changes introduced to the listener `configuration`:

- `overrides` is merged with the `configuration` section
- `dnsAnnotations` has been renamed `annotations`
- `preferredAddressType` has been renamed `preferredNodePortAddressType`
- `address` has been renamed `alternativeNames`
- `loadBalancerSourceRanges` and `externalTrafficPolicy` move to the listener configuration from the now deprecated `template`

For example, this configuration:

### Old additional listener configuration

```
listeners:
  external:
    type: loadbalancer
    authentication:
      type: tls
    overrides:
      bootstrap:
        dnsAnnotations:
          #...
```

Changes to:

### New additional listener configuration

```
listeners:
  #...
- name: external
  port: 9094
  type: loadbalancer
  tls: true
  authentication:
    type: tls
  configuration:
    bootstrap:
      annotations:
        #...
```

#### IMPORTANT

The name and port numbers shown in the new listener configuration **must** be used for backwards compatibility. Using any other values will cause renaming of the Kafka listeners and Kubernetes services.

For more information on the configuration options available for each type of listener, see the [GenericKafkaListener schema reference](#).

### Strategies for upgrading clients

The best approach to upgrading your client applications (including Kafka Connect connectors) depends on your particular circumstances.

Consuming applications need to receive messages in a message format that they understand. You can ensure that this is the case in one of two ways:

- By upgrading all the consumers for a topic *before* upgrading any of the producers.
- By having the brokers down-convert messages to an older format.

Using broker down-conversion puts extra load on the brokers, so it is not ideal to rely on down-conversion for all topics for a prolonged period of time. For brokers to perform optimally they

should not be down converting messages at all.

Broker down-conversion is configured in two ways:

- The topic-level `message.format.version` configures it for a single topic.
- The broker-level `log.message.format.version` is the default for topics that do not have the topic-level `message.format.version` configured.

Messages published to a topic in a new-version format will be visible to consumers, because brokers perform down-conversion when they receive messages from producers, not when they are sent to consumers.

There are a number of strategies you can use to upgrade your clients:

### Consumers first

1. Upgrade all the consuming applications.
2. Change the broker-level `log.message.format.version` to the new version.
3. Upgrade all the producing applications.

This strategy is straightforward, and avoids any broker down-conversion. However, it assumes that all consumers in your organization can be upgraded in a coordinated way, and it does not work for applications that are both consumers and producers. There is also a risk that, if there is a problem with the upgraded clients, new-format messages might get added to the message log so that you cannot revert to the previous consumer version.

### Per-topic consumers first

For each topic:

1. Upgrade all the consuming applications.
2. Change the topic-level `message.format.version` to the new version.
3. Upgrade all the producing applications.

This strategy avoids any broker down-conversion, and means you can proceed on a topic-by-topic basis. It does not work for applications that are both consumers and producers of the same topic. Again, it has the risk that, if there is a problem with the upgraded clients, new-format messages might get added to the message log.

### Per-topic consumers first, with down conversion

For each topic:

1. Change the topic-level `message.format.version` to the old version (or rely on the topic defaulting to the broker-level `log.message.format.version`).
2. Upgrade all the consuming and producing applications.
3. Verify that the upgraded applications function correctly.
4. Change the topic-level `message.format.version` to the new version.

This strategy requires broker down-conversion, but the load on the brokers is minimized because it is only required for a single topic (or small group of topics) at a time. It also works for applications that are both consumers and producers of the same topic. This approach ensures that the upgraded producers and consumers are working correctly before you commit to using the new message format version.

The main drawback of this approach is that it can be complicated to manage in a cluster with many topics and applications.

Other strategies for upgrading client applications are also possible.

#### NOTE

It is also possible to apply multiple strategies. For example, for the first few applications and topics the "per-topic consumers first, with down conversion" strategy can be used. When this has proved successful another, more efficient strategy can be considered acceptable to use instead.

## Upgrading Kafka brokers and client applications

This procedure describes how to upgrade a Strimzi Kafka cluster to a higher version of Kafka.

### Prerequisites

For the **Kafka** resource to be upgraded, check:

- The Cluster Operator, which supports both versions of Kafka, is up and running.
- The `Kafka.spec.kafka.config` does not contain options that are not supported in the version of Kafka that you are upgrading to.
- Whether the `log.message.format.version` and `inter.broker.protocol.version` for the current Kafka version needs to be updated for the new version. [Consult the Kafka versions table](#).

### Procedure

1. Update the Kafka cluster configuration in an editor, as required:

```
kubectl edit kafka my-cluster
```

2. If the `log.message.format.version` and `inter.broker.protocol.version` of the current Kafka version is the same as that of the new Kafka version, proceed to the next step.

Otherwise, ensure that `Kafka.spec.kafka.config` has the `log.message.format.version` and `inter.broker.protocol.version` configured to the default for the *current* version.

For example, if upgrading from Kafka 2.6.0:

```

kind: Kafka
spec:
  # ...
  kafka:
    version: 2.6.0
    config:
      log.message.format.version: "2.6"
      inter.broker.protocol.version: "2.6"
  # ...

```

If the `log.message.format.version` and `inter.broker.protocol.version` are unset, set them to the current version.

#### NOTE

The value of `log.message.format.version` and `inter.broker.protocol.version` 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 version (leaving the `log.message.format.version` and `inter.broker.protocol.version` at the current version).

For example, if upgrading from Kafka 2.6.0 to 2.7.0:

```

apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
spec:
  # ...
  kafka:
    version: 2.7.0 ①
    config:
      log.message.format.version: "2.6" ②
      inter.broker.protocol.version: "2.6" ③
  # ...

```

① Kafka version is changed to the new version.

② Message format version is unchanged.

③ Inter-broker protocol version is unchanged.

#### WARNING

You cannot downgrade Kafka if the `inter.broker.protocol.version` for the new Kafka version changes. The inter-broker protocol version determines the schemas used for persistent metadata stored by the broker, including messages written to `__consumer_offsets`. The downgraded cluster will not understand the messages.

4. 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](#)

5. Save and exit the editor, then wait for rolling updates to complete.

**NOTE**

Additional rolling updates occur if the new version of Kafka has a new ZooKeeper version.

Check the update in the logs or by watching the pod state transitions:

```
kubectl logs -f CLUSTER-OPERATOR-POD-NAME | grep -E "Kafka version upgrade from KafkaVersion.[0-9.]+. to KafkaVersion.*[0-9.]+.*completed"
```

```
kubectl get pod my-cluster-kafka-0 -o jsonpath='{.spec.containers[0].image}'
```

If the current and new versions of Kafka have different inter-broker protocol versions, check the Cluster Operator logs for an **INFO** level message:

```
Reconciliation #NUM(watch) Kafka(NAMESPACE/NAME): Kafka version upgrade from FROM-VERSION to TO-VERSION, phase 2 of 2 completed
```

Alternatively, if the current and new versions of Kafka have the same interbroker protocol version, check for:

```
Reconciliation #NUM(watch) Kafka(NAMESPACE/NAME): Kafka version upgrade from FROM-VERSION to TO-VERSION, phase 1 of 1 completed
```

The rolling updates:

- Ensure each pod is using the broker binaries for the new version of Kafka
- Configure the brokers to send messages using the inter-broker protocol of the new version of Kafka

**NOTE**

Clients are still using the old version, so brokers will convert messages to the old version before sending them to the clients. To minimize this additional load, update the clients as quickly as possible.

6. Depending on your chosen strategy for upgrading clients, upgrade all client applications to use the new version of the client binaries.

See [Strategies for upgrading clients](#)

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, update `KafkaMirrorMaker.spec.version`



7. If the `log.message.format.version` and `inter.broker.protocol.version` identified in step 1 are the same as the new version proceed to the next step.

Otherwise change the `log.message.format.version` and `inter.broker.protocol.version` in `Kafka.spec.kafka.config` to the default version for the new version of Kafka now being used.

For example, if upgrading to 2.7.0:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
spec:
  # ...
  kafka:
    version: 2.7.0
    config:
      log.message.format.version: "2.7"
      # ...
```

8. Wait for the Cluster Operator to update the cluster.

The Kafka cluster and clients are now using the new Kafka version.

## Upgrading consumers and Kafka Streams applications to cooperative rebalancing

You can upgrade Kafka consumers and Kafka Streams applications to use the *incremental cooperative rebalance* protocol for partition rebalances instead of the default *eager rebalance* protocol. The new protocol was added in Kafka 2.4.0.

Consumers keep their partition assignments in a cooperative rebalance and only revoke them at the end of the process, if needed to achieve a balanced cluster. This reduces the unavailability of the consumer group or Kafka Streams application.

<b>NOTE</b>	Upgrading to the incremental cooperative rebalance protocol is optional. The eager rebalance protocol is still supported.
-------------	---

### Prerequisites

- You have [upgraded Kafka brokers and client applications](#) to Kafka 2.7.0.

### Procedure

#### To upgrade a Kafka consumer to use the incremental cooperative rebalance protocol:

1. Replace the Kafka clients `.jar` file with the new version.
2. In the consumer configuration, append `cooperative-sticky` to the `partition.assignment.strategy`. For example, if the `range` strategy is set, change the configuration to `range, cooperative-sticky`.
3. Restart each consumer in the group in turn, waiting for the consumer to rejoin the group after each restart.

4. Reconfigure each consumer in the group by removing the earlier `partition.assignment.strategy` from the consumer configuration, leaving only the `cooperative-sticky` strategy.
5. Restart each consumer in the group in turn, waiting for the consumer to rejoin the group after each restart.

**To upgrade a Kafka Streams application to use the incremental cooperative rebalance protocol:**

1. Replace the Kafka Streams `.jar` file with the new version.
2. In the Kafka Streams configuration, set the `upgrade.from` configuration parameter to the Kafka version you are upgrading from (for example, 2.3).
3. Restart each of the stream processors (nodes) in turn.
4. Remove the `upgrade.from` configuration parameter from the Kafka Streams configuration.
5. Restart each consumer in the group in turn.

*Additional resources*

- [Notable changes in 2.4.0](#) in the Apache Kafka documentation.

## 7.2. Strimzi resource upgrades

The `kafka.strimzi.io/v1alpha1` API version is deprecated for the following Strimzi resources:

- `Kafka`
- `KafkaConnect`
- `KafkaConnectS2I`
- `KafkaMirrorMaker`
- `KafkaTopic`
- `KafkaUser`

Update these resources to use the `kafka.strimzi.io/v1beta1` API version.

This section describes the upgrade steps for the resources.

### IMPORTANT

The upgrade of resources *must* be performed after [upgrading the Cluster Operator](#), so the Cluster Operator can understand the resources.

*What if the resource upgrade does not take effect?*

If the upgrade does not take effect, a warning is given in the logs on reconciliation to indicate that the resource cannot be updated until the `apiVersion` is updated.

To trigger the update, make a cosmetic change to the custom resource, such as adding an annotation.

Example annotation:

```
metadata:
  # ...
  annotations:
    upgrade: "Upgraded to kafka.strimzi.io/v1beta1"
```

The following procedures describe the steps to update specific resources to use the `kafka.strimzi.io/v1beta1` API version:

- [Upgrading Kafka resources](#)
- [Upgrading Kafka Connect resources](#)
- [Upgrading Kafka Connect S2I resources](#)
- [Upgrading Kafka MirrorMaker resources](#)
- [Upgrading Kafka Topic resources](#)
- [Upgrading Kafka User resources](#)

### 7.2.1. Upgrading Kafka resources

#### *Prerequisites*

- A Cluster Operator supporting the `v1beta1` API version is up and running.

#### *Procedure*

Execute the following steps for each `Kafka` resource in your deployment.

1. Update the `Kafka` resource in an editor.

```
kubectl edit kafka my-cluster
```

2. Replace:

```
apiVersion: kafka.strimzi.io/v1alpha1
```

with:

```
apiVersion: kafka.strimzi.io/v1beta1
```

3. If the `Kafka` resource has:

```
Kafka.spec.topicOperator
```

Replace it with:

```
Kafka.spec.entityOperator.topicOperator
```

For example, replace:

```
spec:
  # ...
  topicOperator: {}
```

with:

```
spec:
  # ...
  entityOperator:
    topicOperator: {}
```

4. If present, move:

```
Kafka.spec.entityOperator.affinity
```

```
Kafka.spec.entityOperator.tolerations
```

to:

```
Kafka.spec.entityOperator.template.pod.affinity
```

```
Kafka.spec.entityOperator.template.pod.tolerations
```

For example, move:

```
spec:
  # ...
  entityOperator:
    affinity {}
    tolerations {}
```

to:

```
spec:
  # ...
  entityOperator:
    template:
      pod:
        affinity {}
        tolerations {}
```

5. If present, move:

```
Kafka.spec.kafka.affinity
```

```
Kafka.spec.kafka.tolerations
```

to:

```
Kafka.spec.kafka.template.pod.affinity
```

```
Kafka.spec.kafka.template.pod.tolerations
```

For example, move:

```
spec:
  # ...
  kafka:
    affinity {}
    tolerations {}
```

to:

```
spec:
  # ...
  kafka:
    template:
      pod:
        affinity {}
        tolerations {}
```

6. If present, move:

```
Kafka.spec.zookeeper.affinity
```

```
Kafka.spec.zookeeper.tolerations
```

to:

```
Kafka.spec.zookeeper.template.pod.affinity
```

```
Kafka.spec.zookeeper.template.pod.tolerations
```

For example, move:

```
spec:
  # ...
  zookeeper:
    affinity {}
    tolerations {}
```

to:

```
spec:
  # ...
  zookeeper:
    template:
      pod:
        affinity {}
        tolerations {}
```

7. Save the file, exit the editor and wait for the updated resource to be reconciled.

## 7.2.2. Upgrading Kafka Connect resources

### Prerequisites

- A Cluster Operator supporting the **v1beta1** API version is up and running.

### Procedure

Execute the following steps for each **KafkaConnect** resource in your deployment.

1. Update the **KafkaConnect** resource in an editor.

```
kubectl edit kafkaconnect my-connect
```

2. Replace:

```
apiVersion: kafka.strimzi.io/v1alpha1
```

with:

```
apiVersion: kafka.strimzi.io/v1beta1
```

3. If present, move:

```
KafkaConnect.spec.affinity
```

```
KafkaConnect.spec.tolerations
```

to:

```
KafkaConnect.spec.template.pod.affinity
```

```
KafkaConnect.spec.template.pod.tolerations
```

For example, move:

```
spec:
  # ...
  affinity {}
  tolerations {}
```

to:

```
spec:
  # ...
  template:
    pod:
      affinity {}
      tolerations {}
```

4. Save the file, exit the editor and wait for the updated resource to be reconciled.

### 7.2.3. Upgrading Kafka Connect S2I resources

#### *Prerequisites*

- A Cluster Operator supporting the **v1beta1** API version is up and running.

## Procedure

Execute the following steps for each **KafkaConnectS2I** resource in your deployment.

1. Update the **KafkaConnectS2I** resource in an editor.

```
kubectl edit kafkaconnects2i my-connect
```

2. Replace:

```
apiVersion: kafka.strimzi.io/v1alpha1
```

with:

```
apiVersion: kafka.strimzi.io/v1beta1
```

3. If present, move:

```
KafkaConnectS2I.spec.affinity
```

```
KafkaConnectS2I.spec.tolerations
```

to:

```
KafkaConnectS2I.spec.template.pod.affinity
```

```
KafkaConnectS2I.spec.template.pod.tolerations
```

For example, move:

```
spec:
  # ...
  affinity {}
  tolerations {}
```

to:



```
spec:
  # ...
  template:
    pod:
      affinity {}
      tolerations {}
```

4. Save the file, exit the editor and wait for the updated resource to be reconciled.

## 7.2.4. Upgrading Kafka MirrorMaker resources

### Prerequisites

- A Cluster Operator supporting the **v1beta1** API version is up and running.

### Procedure

Execute the following steps for each **KafkaMirrorMaker** resource in your deployment.

1. Update the **KafkaMirrorMaker** resource in an editor.

```
kubectl edit kafkamirrormaker my-connect
```

2. Replace:

```
apiVersion: kafka.strimzi.io/v1alpha1
```

with:

```
apiVersion: kafka.strimzi.io/v1beta1
```

3. If present, move:

```
KafkaConnectMirrorMaker.spec.affinity
```

```
KafkaConnectMirrorMaker.spec.tolerations
```

to:

```
KafkaConnectMirrorMaker.spec.template.pod.affinity
```

```
KafkaConnectMirrorMaker.spec.template.pod.tolerations
```

For example, move:

```
spec:
  # ...
  affinity {}
  tolerations {}
```

to:

```
spec:
  # ...
  template:
    pod:
      affinity {}
      tolerations {}
```

4. Save the file, exit the editor and wait for the updated resource to be reconciled.

## 7.2.5. Upgrading Kafka Topic resources

### Prerequisites

- A Topic Operator supporting the **v1beta1** API version is up and running.

### Procedure

Execute the following steps for each **KafkaTopic** resource in your deployment.

1. Update the **KafkaTopic** resource in an editor.

```
kubectl edit kafkatopic my-topic
```

2. Replace:

```
apiVersion: kafka.strimzi.io/v1alpha1
```

with:

```
apiVersion: kafka.strimzi.io/v1beta1
```

3. Save the file, exit the editor and wait for the updated resource to be reconciled.

## 7.2.6. Upgrading Kafka User resources

### Prerequisites

- A User Operator supporting the **v1beta1** API version is up and running.

### Procedure

Execute the following steps for each **KafkaUser** resource in your deployment.

1. Update the **KafkaUser** resource in an editor.

```
kubectl edit kafkauser my-user
```

2. Replace:

```
apiVersion: kafka.strimzi.io/v1alpha1
```

with:

```
apiVersion: kafka.strimzi.io/v1beta1
```

3. Save the file, exit the editor and wait for the updated resource to be reconciled.

# Chapter 8. Downgrading Strimzi

If you are encountering issues with the version of Strimzi you upgraded to, you can revert your installation to the previous version.

You can perform a downgrade to:

1. Revert your Cluster Operator to the previous Strimzi version.
  - [Downgrading the Cluster Operator to a previous version](#)
2. Downgrade all Kafka brokers and client applications to the previous Kafka version.
  - [Downgrading Kafka](#)

If the previous version of Strimzi does not support the version of Kafka you are using, you can also downgrade Kafka as long as the log message format versions appended to messages match.

## 8.1. Downgrading the Cluster Operator to a previous version

If you are encountering issues with Strimzi, you can revert your installation.

This procedure describes how to downgrade a Cluster Operator deployment to a previous version.

### *Prerequisites*

- An existing Cluster Operator deployment is available.
- You have [downloaded the installation files for the previous version](#).

### *Procedure*

1. Take note of any configuration changes made to the existing Cluster Operator resources (in the `/install/cluster-operator` directory). 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-strimzi-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 apply -f install/cluster-operator
```

Wait for the rolling updates to complete.

5. 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, `NEW-STRIMZI-VERSION-kafka-CURRENT-KAFKA-VERSION`.

Your Cluster Operator was downgraded to the previous version.

## 8.2. Downgrading Kafka

Kafka version downgrades are performed by the Cluster Operator.

### 8.2.1. Kafka version compatibility for downgrades

Kafka downgrades are dependent on compatible current and target [Kafka versions](#), and the state at which messages have been logged.

You cannot revert to the previous Kafka version if that version does not support any of the `inter.broker.protocol.version` settings which have *ever been used* in that cluster, or messages have been added to message logs that use a newer `log.message.format.version`.

The `inter.broker.protocol.version` determines the schemas used for persistent metadata stored by the broker, such as the schema for messages written to `__consumer_offsets`. If you downgrade to a version of Kafka that does not understand an `inter.broker.protocol.version` that has (ever) been previously used in the cluster the broker will encounter data they cannot understand.

If the target downgrade version of Kafka has:

- The *same* `log.message.format.version` as the current version, the Cluster Operator downgrades by performing a single rolling restart of the brokers.
- A *different* `log.message.format.version`, downgrading is only possible if the running cluster has

always had `log.message.format.version` set to the version used by the downgraded version. This is typically only the case if the upgrade procedure was aborted before the `log.message.format.version` was changed. In this case, the downgrade requires:

- Two rolling restarts of the brokers if the interbroker protocol of the two versions is different
- A single rolling restart if they are the same

Downgrading is *not possible* if the new version has ever used a `log.message.format.version` that is not supported by the previous version, including when the default value for `log.message.format.version` is used. For example, this resource can be downgraded to Kafka version 2.6.0 because the `log.message.format.version` has not been changed:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
spec:
  # ...
  kafka:
    version: 2.7.0
    config:
      log.message.format.version: "2.6"
      # ...
```

The downgrade would not be possible if the `log.message.format.version` was set at `"2.7"` or a value was absent (so that the parameter took the default value for a 2.7.0 broker of 2.7).

### 8.2.2. Downgrading Kafka brokers and client applications

This procedure describes how you can downgrade a Strimzi Kafka cluster to a lower (previous) version of Kafka, such as downgrading from 2.7.0 to 2.6.0.

#### Prerequisites

For the `Kafka` resource to be downgraded, check:

- **IMPORTANT:** [Compatibility of Kafka versions](#).
- The Cluster Operator, which supports both versions of Kafka, is up and running.
- The `Kafka.spec.kafka.config` does not contain options that are not supported by the Kafka version being downgraded to.
- The `Kafka.spec.kafka.config` has a `log.message.format.version` and `inter.broker.protocol.version` that is supported by the Kafka version being downgraded to.

#### Procedure

1. Update the Kafka cluster configuration in an editor, as required.

```
kubectl edit kafka my-cluster
```

2. Change the `Kafka.spec.kafka.version` to specify the previous version.

For example, if downgrading from Kafka 2.7.0 to 2.6.0:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
spec:
  # ...
  kafka:
    version: 2.6.0 ①
    config:
      log.message.format.version: "2.6" ②
      inter.broker.protocol.version: "2.6" ③
    # ...
```

① Kafka version is changed to the previous version.

② Message format version is unchanged.

③ Inter-broker protocol version is unchanged.

**NOTE**

You must format the value of `log.message.format.version` and `inter.broker.protocol.version` as a string to prevent it from being interpreted as a floating point number.

3. 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](#)

4. Save and exit the editor, then wait for rolling updates to complete.

Check the update in the logs or by watching the pod state transitions:

```
kubectl logs -f CLUSTER-OPERATOR-POD-NAME | grep -E "Kafka version downgrade from
[0-9.]+ to [0-9.]+, phase ([0-9]+) of \1 completed"
```

```
kubectl get pod -w
```

Check the Cluster Operator logs for an **INFO** level message:

```
Reconciliation #NUM(watch) Kafka(NAMESPACE/NAME): Kafka version downgrade from
FROM-VERSION to TO-VERSION, phase 1 of 1 completed
```

5. 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.