OpenShift Container Platform 4.18

Extensions

Working with extensions in OpenShift Container Platform using Operator Lifecycle Manager (OLM) v1.

Last Updated: 2025-06-04

OpenShift Container Platform 4.18 Extensions

Working with extensions in OpenShift Container Platform using Operator Lifecycle Manager (OLM) v1.

Abstract

This document provides information about installing, managing, and configuring extensions and Operators on OpenShift Container Platform.

Table of Contents

CHAPTER 1. EXTENSIONS OVERVIEW

Extensions enable cluster administrators to extend capabilities for users on their OpenShift Container Platform cluster.

Operator Lifecycle Manager (OLM) has been included with OpenShift Container Platform 4 since its initial release. OpenShift Container Platform 4.18 includes components for a next-generation iteration of OLM as a Generally Available (GA) feature, known during this phase as *OLM v1*. This updated framework evolves many of the concepts that have been part of previous versions of OLM and adds new capabilities.

1.1. HIGHLIGHTS

Administrators can explore the following highlights:

Fully declarative model that supports GitOps workflows

OLM v1 simplifies extension management through two key APIs:

- A new **ClusterExtension** API streamlines management of installed extensions, which includes Operators via the **registry+v1** bundle format, by consolidating user-facing APIs into a single object. This API is provided as **clusterextension.olm.operatorframework.io** by the new Operator Controller component. Administrators and SREs can use the API to automate processes and define desired states by using GitOps principles.

  NOTE

  Earlier Technology Preview phases of OLM v1 introduced a new **Operator** API; this API is renamed **ClusterExtension** in OpenShift Container Platform 4.16 to address the following improvements:

  - More accurately reflects the simplified functionality of extending a cluster’s capabilities
  - Better represents a more flexible packaging format
  - **Cluster** prefix clearly indicates that **ClusterExtension** objects are cluster-scoped, a change from OLM (Classic) where Operators could be either namespace-scoped or cluster-scoped

- The **Catalog** API, provided by the new catalogd component, serves as the foundation for OLM v1, unpacking catalogs for on-cluster clients so that users can discover installable content, such as Kubernetes extensions and Operators. This provides increased visibility into all available Operator bundle versions, including their details, channels, and update edges.

For more information, see Operator Controller and Catalogd.

Improved control over extension updates

With improved insight into catalog content, administrators can specify target versions for installation and updates. This grants administrators more control over the target version of extension updates. For more information, see Updating an cluster extension.

Flexible extension packaging format

Administrators can use file-based catalogs to install and manage extensions, such as OLM-based Operators, similar to the OLM (Classic) experience.

In addition, bundle size is no longer constrained by the etcd value size limit. For more information, see Installing extensions.

Secure catalog communication

OLM v1 uses HTTPS encryption for catalogd server responses.

1.2. PURPOSE

The mission of Operator Lifecycle Manager (OLM) has been to manage the lifecycle of cluster extensions centrally and declaratively on Kubernetes clusters. Its purpose has always been to make installing, running, and updating functional extensions to the cluster easy, safe, and reproducible for cluster and platform-as-a-service (PaaS) administrators throughout the lifecycle of the underlying cluster.

The initial version of OLM, which launched with OpenShift Container Platform 4 and is included by default, focused on providing unique support for these specific needs for a particular type of cluster extension, known as Operators. Operators are classified as one or more Kubernetes controllers, shipping with one or more API extensions, as **CustomResourceDefinition** (CRD) objects, to provide additional functionality to the cluster.

After running in production clusters for many releases, the next-generation of OLM aims to encompass lifecycles for cluster extensions that are not just Operators.

CHAPTER 2. ARCHITECTURE

2.1. OLM V1 COMPONENTS OVERVIEW

Operator Lifecycle Manager (OLM) v1 comprises the following component projects:

Operator Controller

Operator Controller is the central component of OLM v1 that extends Kubernetes with an API through which users can install and manage the lifecycle of Operators and extensions. It consumes information from catalogd.

Catalogd

Catalogd is a Kubernetes extension that unpacks file-based catalog (FBC) content packaged and shipped in container images for consumption by on-cluster clients. As a component of the OLM v1 microservices architecture, catalogd hosts metadata for Kubernetes extensions packaged by the authors of the extensions, and as a result helps users discover installable content.

2.2. OPERATOR CONTROLLER

Operator Controller is the central component of Operator Lifecycle Manager (OLM) v1 and consumes the other OLM v1 component, catalogd. It extends Kubernetes with an API through which users can install Operators and extensions.

2.2.1. ClusterExtension API

Operator Controller provides a new **ClusterExtension** API object that is a single resource representing an instance of an installed extension, which includes Operators via the **registry+v1** bundle format. This **clusterextension.olm.operatorframework.io** API streamlines management of installed extensions by consolidating user-facing APIs into a single object.

IMPORTANT

In OLM v1, **ClusterExtension** objects are cluster-scoped. This differs from OLM (Classic) where Operators could be either namespace-scoped or cluster-scoped, depending on the configuration of their related **Subscription** and **OperatorGroup** objects.

For more information about the earlier behavior, see *Multitenancy and Operator colocation*.

Example ClusterExtension object

```
apiVersion: olm.operatorframework.io/v1
kind: ClusterExtension
metadata:
  name: <extension_name>
spec:
  namespace: <namespace_name>
  serviceAccount:
    name: <service_account_name>
  source:
    sourceType: Catalog
    catalog:
      packageName: <package_name>
      channels:
        - <channel>
      version: "<version>"
```

Additional resources

- Operator Lifecycle Manager (OLM) → Multitenancy and Operator colocation

2.2.1.1. Example custom resources (CRs) that specify a target version

In Operator Lifecycle Manager (OLM) v1, cluster administrators can declaratively set the target version of an Operator or extension in the custom resource (CR).

You can define a target version by specifying any of the following fields:

- Channel
- Version number
- Version range

If you specify a channel in the CR, OLM v1 installs the latest version of the Operator or extension that can be resolved within the specified channel. When updates are published to the specified channel, OLM v1 automatically updates to the latest release that can be resolved from the channel.

Example CR with a specified channel

```
apiVersion: olm.operatorframework.io/v1
  kind: ClusterExtension
  metadata:
    name: <clusterextension_name>
  spec:
    namespace: <installed_namespace>
    serviceAccount:
      name: <service_account_installer_name>
    source:
      sourceType: Catalog
      catalog:
        packageName: <package_name>
        channels:
          - latest 1
```

1 Optional: Installs the latest release that can be resolved from the specified channel. Updates to the channel are automatically installed. Specify the value of the **channels** parameter as an array.

If you specify the Operator or extension's target version in the CR, OLM v1 installs the specified version. When the target version is specified in the CR, OLM v1 does not change the target version when updates are published to the catalog.

If you want to update the version of the Operator that is installed on the cluster, you must manually edit the Operator's CR. Specifying an Operator's target version pins the Operator's version to the specified release.

Example CR with the target version specified

```
apiVersion: olm.operatorframework.io/v1
  kind: ClusterExtension
  metadata:
    name: <clusterextension_name>
  spec:
    namespace: <installed_namespace>
    serviceAccount:
      name: <service_account_installer_name>
    source:
      sourceType: Catalog
      catalog:
        packageName: <package_name>
        version: "1.11.1" 1
```

1 Optional: Specifies the target version. If you want to update the version of the Operator or extension that is installed, you must manually update this field the CR to the desired target version.

If you want to define a range of acceptable versions for an Operator or extension, you can specify a version range by using a comparison string. When you specify a version range, OLM v1 installs the latest version of an Operator or extension that can be resolved by the Operator Controller.

Example CR with a version range specified

```
apiVersion: olm.operatorframework.io/v1
  kind: ClusterExtension
  metadata:
    name: <clusterextension_name>
  spec:
    namespace: <installed_namespace>
    serviceAccount:
      name: <service_account_installer_name>
    source:
      sourceType: Catalog
      catalog:
        packageName: <package_name>
        version: ">1.11.1" 1
```

1 Optional: Specifies that the desired version range is greater than version **1.11.1**. For more information, see "Support for version ranges".

After you create or update a CR, apply the configuration file by running the following command:

Command syntax

```
$ oc apply -f <extension_name>.yaml
```

2.2.2. Object ownership for cluster extensions

In Operator Lifecycle Manager (OLM) v1, a Kubernetes object can only be owned by a single **ClusterExtension** object at a time. This ensures that objects within an OpenShift Container Platform cluster are managed consistently and prevents conflicts between multiple cluster extensions attempting to control the same object.

2.2.2.1. Single ownership

The core ownership principle enforced by OLM v1 is that each object can only have one cluster extension as its owner. This prevents overlapping or conflicting management by multiple cluster extensions, ensuring that each object is uniquely associated with only one bundle.

Implications of single ownership

- Bundles that provide a **CustomResourceDefinition** (CRD) object can only be installed once.
  Bundles provide CRDs, which are part of a **ClusterExtension** object. This means you can install a bundle only once in a cluster. Attempting to install another bundle that provides the same CRD results in failure, as each custom resource can have only one cluster extension as its owner.

- Cluster extensions cannot share objects.
  The single-owner policy of OLM v1 means that cluster extensions cannot share ownership of any objects. If one cluster extension manages a specific object, such as a **Deployment**, **CustomResourceDefinition**, or **Service** object, another cluster extension cannot claim ownership of the same object. Any attempt to do so is blocked by OLM v1.

2.2.2.2. Error messages

When a conflict occurs due to multiple cluster extensions attempting to manage the same object, Operator Controller returns an error message indicating the ownership conflict, such as the following:

Example error message

```
CustomResourceDefinition 'logfilemetricexporters.logging.kubernetes.io' already exists in namespace
'kubernetes-logging' and cannot be managed by operator-controller
```

This error message signals that the object is already being managed by another cluster extension and cannot be reassigned or shared.

2.2.2.3. Considerations

As a cluster or extension administrator, review the following considerations:

Uniqueness of bundles

Ensure that Operator bundles providing the same CRDs are not installed more than once. This can prevent potential installation failures due to ownership conflicts.

Avoid object sharing

If you need different cluster extensions to interact with similar resources, ensure they are managing separate objects. Cluster extensions cannot jointly manage the same object due to the single-owner enforcement.

2.3. CATALOGD

Operator Lifecycle Manager (OLM) v1 uses the catalogd component and its resources to manage Operator and extension catalogs.

2.3.1. About catalogs in OLM v1

You can discover installable content by querying a catalog for Kubernetes extensions, such as Operators and controllers, by using the catalogd component. Catalogd is a Kubernetes extension that unpacks catalog content for on-cluster clients and is part of the Operator Lifecycle Manager (OLM) v1 suite of microservices. Currently, catalogd unpacks catalog content that is packaged and distributed as container images.

Additional resources

- File-based catalogs
- Adding a catalog to a cluster
- Red Hat-provided catalogs

CHAPTER 3. OPERATOR FRAMEWORK GLOSSARY OF COMMON TERMS

The following terms are related to the Operator Framework, including Operator Lifecycle Manager (OLM) v1.

3.1. BUNDLE

In the bundle format, a *bundle* is a collection of an Operator CSV, manifests, and metadata. Together, they form a unique version of an Operator that can be installed onto the cluster.

3.2. BUNDLE IMAGE

In the bundle format, a *bundle image* is a container image that is built from Operator manifests and that contains one bundle. Bundle images are stored and distributed by Open Container Initiative (OCI) spec container registries, such as Quay.io or DockerHub.

3.3. CATALOG SOURCE

A *catalog source* represents a store of metadata that OLM can query to discover and install Operators and their dependencies.

3.4. CHANNEL

A *channel* defines a stream of updates for an Operator and is used to roll out updates for subscribers. The head points to the latest version of that channel. For example, a **stable** channel would have all stable versions of an Operator arranged from the earliest to the latest.

An Operator can have several channels, and a subscription binding to a certain channel would only look for updates in that channel.

3.5. CHANNEL HEAD

A *channel head* refers to the latest known update in a particular channel.

3.6. CLUSTER SERVICE VERSION

A *cluster service version (CSV)* is a YAML manifest created from Operator metadata that assists OLM in running the Operator in a cluster. It is the metadata that accompanies an Operator container image, used to populate user interfaces with information such as its logo, description, and version.

It is also a source of technical information that is required to run the Operator, like the RBAC rules it requires and which custom resources (CRs) it manages or depends on.

3.7. DEPENDENCY

An Operator may have a *dependency* on another Operator being present in the cluster. For example, the Vault Operator has a dependency on the etcd Operator for its data persistence layer.

OLM resolves dependencies by ensuring that all specified versions of Operators and CRDs are installed on the cluster during the installation phase. This dependency is resolved by finding and installing an Operator in a catalog that satisfies the required CRD API, and is not related to packages or bundles.

3.8. EXTENSION

Extensions enable cluster administrators to extend capabilities for users on their OpenShift Container Platform cluster. Extensions are managed by Operator Lifecycle Manager (OLM) v1.

The **ClusterExtension** API streamlines management of installed extensions, which includes Operators via the **registry+v1** bundle format, by consolidating user-facing APIs into a single object. Administrators and SREs can use the API to automate processes and define desired states by using GitOps principles.

3.9. INDEX IMAGE

In the bundle format, an *index image* refers to an image of a database (a database snapshot) that contains information about Operator bundles including CSVs and CRDs of all versions. This index can host a history of Operators on a cluster and be maintained by adding or removing Operators using the **opm** CLI tool.

3.10. INSTALL PLAN

An *install plan* is a calculated list of resources to be created to automatically install or upgrade a CSV.

3.11. MULTITENANCY

A *tenant* in OpenShift Container Platform is a user or group of users that share common access and privileges for a set of deployed workloads, typically represented by a namespace or project. You can use tenants to provide a level of isolation between different groups or teams.

When a cluster is shared by multiple users or groups, it is considered a *multitenant* cluster.

3.12. OPERATOR

Operators are a method of packaging, deploying, and managing a Kubernetes application. A Kubernetes application is an app that is both deployed on Kubernetes and managed using the Kubernetes APIs and **kubectl** or **oc** tooling.

In Operator Lifecycle Manager (OLM) v1, the **ClusterExtension** API streamlines management of installed extensions, which includes Operators via the **registry+v1** bundle format.

3.13. OPERATOR GROUP

An *Operator group* configures all Operators deployed in the same namespace as the **OperatorGroup** object to watch for their CR in a list of namespaces or cluster-wide.

3.14. PACKAGE

In the bundle format, a *package* is a directory that encloses all released history of an Operator with each version. A released version of an Operator is described in a CSV manifest alongside the CRDs.

3.15. REGISTRY

A *registry* is a database that stores bundle images of Operators, each with all of its latest and historical versions in all channels.

3.16. SUBSCRIPTION

A *subscription* keeps CSVs up to date by tracking a channel in a package.

3.17. UPDATE GRAPH

An *update graph* links versions of CSVs together, similar to the update graph of any other packaged software. Operators can be installed sequentially, or certain versions can be skipped. The update graph is expected to grow only at the head with newer versions being added.

Also known as *update edges* or *update paths*.

CHAPTER 4. CATALOGS

4.1. FILE-BASED CATALOGS

Operator Lifecycle Manager (OLM) v1 in OpenShift Container Platform supports *file-based catalogs* for discovering and sourcing cluster extensions, including Operators, on a cluster.

4.1.1. Highlights

File-based catalogs are the latest iteration of the catalog format in Operator Lifecycle Manager (OLM). It is a plain text-based (JSON or YAML) and declarative config evolution of the earlier SQLite database format, and it is fully backwards compatible. The goal of this format is to enable Operator catalog editing, composability, and extensibility.

Editing

With file-based catalogs, users interacting with the contents of a catalog are able to make direct changes to the format and verify that their changes are valid. Because this format is plain text JSON or YAML, catalog maintainers can easily manipulate catalog metadata by hand or with widely known and supported JSON or YAML tooling, such as the **jq** CLI.
This editability enables the following features and user-defined extensions:

- Promoting an existing bundle to a new channel
- Changing the default channel of a package
- Custom algorithms for adding, updating, and removing upgrade paths

Composability

File-based catalogs are stored in an arbitrary directory hierarchy, which enables catalog composition. For example, consider two separate file-based catalog directories: **catalogA** and **catalogB**. A catalog maintainer can create a new combined catalog by making a new directory **catalogC** and copying **catalogA** and **catalogB** into it.
This composability enables decentralized catalogs. The format permits Operator authors to maintain Operator-specific catalogs, and it permits maintainers to trivially build a catalog composed of individual Operator catalogs. File-based catalogs can be composed by combining multiple other catalogs, by extracting subsets of one catalog, or a combination of both of these.

NOTE

Duplicate packages and duplicate bundles within a package are not permitted. The **opm validate** command returns an error if any duplicates are found.

Because Operator authors are most familiar with their Operator, its dependencies, and its upgrade compatibility, they are able to maintain their own Operator-specific catalog and have direct control over its contents. With file-based catalogs, Operator authors own the task of building and maintaining their packages in a catalog. Composite catalog maintainers, however, only own the task of curating the packages in their catalog and publishing the catalog to users.

Extensibility

The file-based catalog specification is a low-level representation of a catalog. While it can be maintained directly in its low-level form, catalog maintainers can build interesting extensions on top that can be used by their own custom tooling to make any number of mutations.

For example, a tool could translate a high-level API, such as **(mode=semver)**, down to the low-level, file-based catalog format for upgrade paths. Or a catalog maintainer might need to customize all of the bundle metadata by adding a new property to bundles that meet a certain criteria.

While this extensibility allows for additional official tooling to be developed on top of the low-level APIs for future OpenShift Container Platform releases, the major benefit is that catalog maintainers have this capability as well.

4.1.2. Directory structure

File-based catalogs can be stored and loaded from directory-based file systems. The **opm** CLI loads the catalog by walking the root directory and recursing into subdirectories. The CLI attempts to load every file it finds and fails if any errors occur.

Non-catalog files can be ignored using **.indexignore** files, which have the same rules for patterns and precedence as **.gitignore** files.

Example .indexignore file

```
# Ignore everything except non-object .json and .yaml files
**/*
!*.json
!*.yaml
**/objects/*.json
**/objects/*.yaml
```

Catalog maintainers have the flexibility to choose their desired layout, but it is recommended to store each package's file-based catalog blobs in separate subdirectories. Each individual file can be either JSON or YAML; it is not necessary for every file in a catalog to use the same format.

Basic recommended structure

```
catalog
├── packageA
│   └── index.yaml
├── packageB
│   ├── .indexignore
│   ├── index.yaml
│   └── objects
│       └── packageB.v0.1.0.clusterserviceversion.yaml
└── packageC
    └── index.json
    └── deprecations.yaml
```

This recommended structure has the property that each subdirectory in the directory hierarchy is a self-contained catalog, which makes catalog composition, discovery, and navigation trivial file system operations. The catalog can also be included in a parent catalog by copying it into the parent catalog's root directory.

4.1.3. Schemas

File-based catalogs use a format, based on the CUE language specification, that can be extended with arbitrary schemas. The following **_Meta** CUE schema defines the format that all file-based catalog blobs must adhere to:

_Meta schema

```
_Meta: {
  // schema is required and must be a non-empty string
  schema: string & !=""

  // package is optional, but if it's defined, it must be a non-empty string
  package?: string & !=""

  // properties is optional, but if it's defined, it must be a list of 0 or more properties
  properties?: [... #Property]
}

#Property: {
  // type is required
  type: string & !=""

  // value is required, and it must not be null
  value: !=null
}
```

NOTE

No CUE schemas listed in this specification should be considered exhaustive. The **opm validate** command has additional validations that are difficult or impossible to express concisely in CUE.

An Operator Lifecycle Manager (OLM) catalog currently uses three schemas (**olm.package**, **olm.channel**, and **olm.bundle**), which correspond to OLM's existing package and bundle concepts.

Each Operator package in a catalog requires exactly one **olm.package** blob, at least one **olm.channel** blob, and one or more **olm.bundle** blobs.

NOTE

All **olm.*** schemas are reserved for OLM-defined schemas. Custom schemas must use a unique prefix, such as a domain that you own.

4.1.3.1. olm.package schema

The **olm.package** schema defines package-level metadata for an Operator. This includes its name, description, default channel, and icon.

Example 4.1. olm.package schema

```
#Package: {
  schema: "olm.package"

  // Package name
  name: string & !=""

  // A description of the package
  description?: string
  // The package's default channel
  defaultChannel: string & !=""

  // An optional icon
  icon?: {
    base64data: string
    mediatype:  string
  }
}
```

4.1.3.2. olm.channel schema

The **olm.channel** schema defines a channel within a package, the bundle entries that are members of the channel, and the upgrade paths for those bundles.

If a bundle entry represents an edge in multiple **olm.channel** blobs, it can only appear once per channel.

It is valid for an entry's **replaces** value to reference another bundle name that cannot be found in this catalog or another catalog. However, all other channel invariants must hold true, such as a channel not having multiple heads.

Example 4.2. olm.channel schema

```
#Channel: {
  schema: "olm.channel"
  package: string & !=""
  name: string & !=""
  entries: [...#ChannelEntry]
}

#ChannelEntry: {
  // name is required. It is the name of an `olm.bundle` that
  // is present in the channel.
  name: string & !=""

  // replaces is optional. It is the name of bundle that is replaced
  // by this entry. It does not have to be present in the entry list.
  replaces?: string & !=""

  // skips is optional. It is a list of bundle names that are skipped by
  // this entry. The skipped bundles do not have to be present in the
  // entry list.
  skips?: [...string & !=""]

  // skipRange is optional. It is the semver range of bundle versions
  // that are skipped by this entry.
  skipRange?: string & !=""
}
```

WARNING

When using the **skipRange** field, the skipped Operator versions are pruned from the update graph and are longer installable by users with the **spec.startingCSV** property of **Subscription** objects.

You can update an Operator incrementally while keeping previously installed versions available to users for future installation by using both the **skipRange** and **replaces** field. Ensure that the **replaces** field points to the immediate previous version of the Operator version in question.

4.1.3.3. olm.bundle schema

Example 4.3. olm.bundle schema

```
#Bundle: {
  schema: "olm.bundle"
  package: string & !=""
  name: string & !=""
  image: string & !=""
  properties: [...#Property]
  relatedImages?: [...#RelatedImage]
}

#Property: {
  // type is required
  type: string & !=""

  // value is required, and it must not be null
  value: !=null
}

#RelatedImage: {
  // image is the image reference
  image: string & !=""

  // name is an optional descriptive name for an image that
  // helps identify its purpose in the context of the bundle
  name?: string & !=""
}
```

4.1.3.4. olm.deprecations schema

The optional **olm.deprecations** schema defines deprecation information for packages, bundles, and channels in a catalog. Operator authors can use this schema to provide relevant messages about their Operators, such as support status and recommended upgrade paths, to users running those Operators from a catalog.

When this schema is defined, the OpenShift Container Platform web console displays warning badges for the affected elements of the Operator, including any custom deprecation messages, on both the pre- and post-installation pages of the OperatorHub.

An **olm.deprecations** schema entry contains one or more of the following **reference** types, which indicates the deprecation scope. After the Operator is installed, any specified messages can be viewed as status conditions on the related **Subscription** object.

Table 4.1. Deprecation **reference** types

Type	Scope	Status condition
olm.package	Represents the entire package	**PackageDeprecated**
olm.channel	Represents one channel	**ChannelDeprecated**
olm.bundle	Represents one bundle version	**BundleDeprecated**

Each **reference** type has their own requirements, as detailed in the following example.

Example 4.4. Example olm.deprecations schema with eachreference type

```
schema: olm.deprecations
package: my-operator (1)
entries:
  - reference:
      schema: olm.package (2)
    message: | (3)
    The 'my-operator' package is end of life. Please use the
    'my-operator-new' package for support.
  - reference:
      schema: olm.channel
      name: alpha (4)
    message: |
    The 'alpha' channel is no longer supported. Please switch to the
    'stable' channel.
  - reference:
      schema: olm.bundle
      name: my-operator.v1.68.0 (5)
    message: |
    my-operator.v1.68.0 is deprecated. Uninstall my-operator.v1.68.0 and
    install my-operator.v1.72.0 for support.
```

(1) Each deprecation schema must have a **package** value, and that package reference must be unique across the catalog. There must not be an associated **name** field.

(2) The **olm.package** schema must not include a **name** field, because it is determined by the **package** field defined earlier in the schema.

(3) All **message** fields, for any **reference** type, must be a non-zero length and represented as an opaque text blob.

(4) The **name** field for the **olm.channel** schema is required.

5 The **name** field for the **olm.bundle** schema is required.

NOTE

The deprecation feature does not consider overlapping deprecation, for example package versus channel versus bundle.

Operator authors can save **olm.deprecations** schema entries as a **deprecations.yaml** file in the same directory as the package's **index.yaml** file:

Example directory structure for a catalog with deprecations

```
my-catalog
└── my-operator
    ├── index.yaml
    └── deprecations.yaml
```

Additional resources

- Updating or filtering a file-based catalog image

4.1.4. Properties

Properties are arbitrary pieces of metadata that can be attached to file-based catalog schemas. The **type** field is a string that effectively specifies the semantic and syntactic meaning of the **value** field. The value can be any arbitrary JSON or YAML.

OLM defines a handful of property types, again using the reserved **olm.*** prefix.

4.1.4.1. olm.package property

The **olm.package** property defines the package name and version. This is a required property on bundles, and there must be exactly one of these properties. The **packageName** field must match the bundle's first-class **package** field, and the **version** field must be a valid semantic version.

Example 4.5. olm.package property

```
#PropertyPackage: {
  type: "olm.package"
  value: {
    packageName: string & !=""
    version: string & !=""
  }
}
```

4.1.4.2. olm.gvk property

The **olm.gvk** property defines the group/version/kind (GVK) of a Kubernetes API that is provided by this bundle. This property is used by OLM to resolve a bundle with this property as a dependency for other bundles that list the same GVK as a required API. The GVK must adhere to Kubernetes GVK validations.

Example 4.6. olm.gvk property

```
#PropertyGVK: {
  type: "olm.gvk"
  value: {
    group: string & !=""
    version: string & !=""
    kind: string & !=""
  }
}
```

4.1.4.3. olm.package.required

The **olm.package.required** property defines the package name and version range of another package that this bundle requires. For every required package property a bundle lists, OLM ensures there is an Operator installed on the cluster for the listed package and in the required version range. The **versionRange** field must be a valid semantic version (semver) range.

Example 4.7. olm.package.required property

```
#PropertyPackageRequired: {
  type: "olm.package.required"
  value: {
    packageName: string & !=""
    versionRange: string & !=""
  }
}
```

4.1.4.4. olm.gvk.required

The **olm.gvk.required** property defines the group/version/kind (GVK) of a Kubernetes API that this bundle requires. For every required GVK property a bundle lists, OLM ensures there is an Operator installed on the cluster that provides it. The GVK must adhere to Kubernetes GVK validations.

Example 4.8. olm.gvk.required property

```
#PropertyGVKRequired: {
  type: "olm.gvk.required"
  value: {
    group: string & !=""
    version: string & !=""
    kind: string & !=""
  }
}
```

4.1.5. Example catalog

With file-based catalogs, catalog maintainers can focus on Operator curation and compatibility. Because Operator authors have already produced Operator-specific catalogs for their Operators, catalog maintainers can build their catalog by rendering each Operator catalog into a subdirectory of the catalog's root directory.

There are many possible ways to build a file-based catalog; the following steps outline a simple approach:

1. Maintain a single configuration file for the catalog, containing image references for each Operator in the catalog:

   Example catalog configuration file

   ```
   name: community-operators
   repo: quay.io/community-operators/catalog
   tag: latest
   references:
   - name: etcd-operator
     image: quay.io/etcd-
   operator/index@sha256:5891b5b522d5df086d0ff0b110fbd9d21bb4fc7163af34d08286a2e846f
   6be03
   - name: prometheus-operator
     image: quay.io/prometheus-
   operator/index@sha256:e258d248fda94c63753607f7c4494ee0fcbe92f1a76bfdac795c9d84101
   eb317
   ```

2. Run a script that parses the configuration file and creates a new catalog from its references:

   Example script

   ```
   name=$(yq eval '.name' catalog.yaml)
   mkdir "$name"
   yq eval '.name + "/" + .references[].name' catalog.yaml | xargs mkdir
   for l in $(yq e '.name as $catalog | .references[] | .image + "|" + $catalog + "/" + .name +
   "/index.yaml"' catalog.yaml); do
     image=$(echo $l | cut -d'|' -f1)
     file=$(echo $l | cut -d'|' -f2)
     opm render "$image" > "$file"
   done
   opm generate dockerfile "$name"
   indexImage=$(yq eval '.repo + ":" + .tag' catalog.yaml)
   docker build -t "$indexImage" -f "$name.Dockerfile" .
   docker push "$indexImage"
   ```

4.1.6. Guidelines

Consider the following guidelines when maintaining file-based catalogs.

4.1.6.1. Immutable bundles

The general advice with Operator Lifecycle Manager (OLM) is that bundle images and their metadata should be treated as immutable.

If a broken bundle has been pushed to a catalog, you must assume that at least one of your users has upgraded to that bundle. Based on that assumption, you must release another bundle with an upgrade path from the broken bundle to ensure users with the broken bundle installed receive an upgrade. OLM will not reinstall an installed bundle if the contents of that bundle are updated in the catalog.

However, there are some cases where a change in the catalog metadata is preferred:

- Channel promotion: If you already released a bundle and later decide that you would like to add it to another channel, you can add an entry for your bundle in another **olm.channel** blob.
- New upgrade paths: If you release a new **1.2.z** bundle version, for example **1.2.4**, but **1.3.0** is already released, you can update the catalog metadata for **1.3.0** to skip **1.2.4**.

4.1.6.2. Source control

Catalog metadata should be stored in source control and treated as the source of truth. Updates to catalog images should include the following steps:

1. Update the source-controlled catalog directory with a new commit.
2. Build and push the catalog image. Use a consistent tagging taxonomy, such as **:latest** or **:<target_cluster_version>**, so that users can receive updates to a catalog as they become available.

4.1.7. CLI usage

For instructions about creating file-based catalogs by using the **opm** CLI, see Managing custom catalogs.

For reference documentation about the **opm** CLI commands related to managing file-based catalogs, see CLI tools.

4.1.8. Automation

Operator authors and catalog maintainers are encouraged to automate their catalog maintenance with CI/CD workflows. Catalog maintainers can further improve on this by building GitOps automation to accomplish the following tasks:

- Check that pull request (PR) authors are permitted to make the requested changes, for example by updating their package's image reference.
- Check that the catalog updates pass the **opm validate** command.
- Check that the updated bundle or catalog image references exist, the catalog images run successfully in a cluster, and Operators from that package can be successfully installed.
- Automatically merge PRs that pass the previous checks.
- Automatically rebuild and republish the catalog image.

4.2. RED HAT-PROVIDED CATALOGS

Red Hat provides several Operator catalogs that are included with OpenShift Container Platform by default.

4.2.1. About Red Hat-provided Operator catalogs

The Red Hat-provided catalog sources are installed by default in the **openshift-marketplace** namespace, which makes the catalogs available cluster-wide in all namespaces.

The following Operator catalogs are distributed by Red Hat:

Catalog	Index image	Description
redhat-operators	**registry.redhat.io/redhat/redhat-operator-index:v4.18**	Red Hat products packaged and shipped by Red Hat. Supported by Red Hat.
certified-operators	**registry.redhat.io/redhat/certified-operator-index:v4.18**	Products from leading independent software vendors (ISVs). Red Hat partners with ISVs to package and ship. Supported by the ISV.
redhat-marketplace	**registry.redhat.io/redhat/redhat-marketplace-index:v4.18**	Certified software that can be purchased from Red Hat Marketplace.
community-operators	**registry.redhat.io/redhat/community-operator-index:v4.18**	Software maintained by relevant representatives in the redhat-openshift-ecosystem/community-operators-prod/operators GitHub repository. No official support.

During a cluster upgrade, the index image tag for the default Red Hat-provided catalog sources are updated automatically by the Cluster Version Operator (CVO) so that Operator Lifecycle Manager (OLM) pulls the updated version of the catalog. For example during an upgrade from OpenShift Container Platform 4.8 to 4.9, the **spec.image** field in the **CatalogSource** object for the **redhat-operators** catalog is updated from:

```
registry.redhat.io/redhat/redhat-operator-index:v4.8
```

to:

```
registry.redhat.io/redhat/redhat-operator-index:v4.9
```

4.3. MANAGING CATALOGS

Cluster administrators can add *catalogs*, or curated collections of Operators and Kubernetes extensions, to their clusters. Operator authors publish their products to these catalogs. When you add a catalog to your cluster, you have access to the versions, patches, and over-the-air updates of the Operators and extensions that are published to the catalog.

You can manage catalogs and extensions declaratively from the CLI by using custom resources (CRs).

File-based catalogs are the latest iteration of the catalog format in Operator Lifecycle Manager (OLM). It is a plain text-based (JSON or YAML) and declarative config evolution of the earlier SQLite database format, and it is fully backwards compatible.

IMPORTANT

Kubernetes periodically deprecates certain APIs that are removed in subsequent releases. As a result, Operators are unable to use removed APIs starting with the version of OpenShift Container Platform that uses the Kubernetes version that removed the API.

If your cluster is using custom catalogs, see Controlling Operator compatibility with OpenShift Container Platform versions for more details about how Operator authors can update their projects to help avoid workload issues and prevent incompatible upgrades.

4.3.1. About catalogs in OLM v1

You can discover installable content by querying a catalog for Kubernetes extensions, such as Operators and controllers, by using the catalogd component. Catalogd is a Kubernetes extension that unpacks catalog content for on-cluster clients and is part of the Operator Lifecycle Manager (OLM) v1 suite of microservices. Currently, catalogd unpacks catalog content that is packaged and distributed as container images.

Additional resources

- File-based catalogs

4.3.2. Red Hat-provided Operator catalogs in OLM v1

Operator Lifecycle Manager (OLM) v1 includes the following Red Hat-provided Operator catalogs on the cluster by default. If you want to add an additional catalog to your cluster, create a custom resource (CR) for the catalog and apply it to the cluster. The following custom resource (CR) examples show the default catalogs installed on the cluster.

Red Hat Operators catalog

```
apiVersion: olm.operatorframework.io/v1
kind: ClusterCatalog
metadata:
  name: openshift-redhat-operators
spec:
  priority: -100
  source:
    image:
      pollIntervalMinutes: <poll_interval_duration> (1)
      ref: registry.redhat.io/redhat/redhat-operator-index:v4.18
    type: Image
```

(1) Specify the interval in minutes for polling the remote registry for newer image digests. To disable polling, do not set the field.

Certified Operators catalog

```
apiVersion: olm.operatorframework.io/v1
```

```
kind: ClusterCatalog
metadata:
  name: openshift-certified-operators
spec:
priority: -200
  source:
    type: image
    image:
      pollIntervalMinutes: 10
      ref: registry.redhat.io/redhat/certified-operator-index:v4.18
    type: Image
```

Red Hat Marketplace catalog

```
apiVersion: olm.operatorframework.io/v1
kind: ClusterCatalog
metadata:
  name: openshift-redhat-marketplace
spec:
  priority: -300
  source:
    image:
      pollIntervalMinutes: 10
      ref: registry.redhat.io/redhat/redhat-marketplace-index:v4.18
    type: Image
```

Community Operators catalog

```
apiVersion: olm.operatorframework.io/v1
kind: ClusterCatalog
metadata:
  name: openshift-community-operators
spec:
  priority: -400
  source:
    image:
      pollIntervalMinutes: 10
      ref: registry.redhat.io/redhat/community-operator-index:v4.18
    type: Image
```

The following command adds a catalog to your cluster:

Command syntax

```
$ oc apply -f <catalog_name>.yaml 1
```

1 Specifies the catalog CR, such as **my-catalog.yaml**.

4.3.3. Adding a catalog to a cluster

To add a catalog to a cluster for Operator Lifecycle Manager (OLM) v1 usage, create a **ClusterCatalog** custom resource (CR) and apply it to the cluster.

Procedure

1. Create a catalog custom resource (CR), similar to the following example:

   Example my-redhat-operators.yaml file

   ```
   apiVersion: olm.operatorframework.io/v1
   kind: ClusterCatalog
   metadata:
     name: my-redhat-operators 1
   spec:
     priority: 1000 2
     source:
       image:
         pollIntervalMinutes: 10 3
         ref: registry.redhat.io/redhat/community-operator-index:v4.18 4
       type: Image
   ```

   1 The catalog is automatically labeled with the value of the **metadata.name** field when it is applied to the cluster. For more information about labels and catalog selection, see "Catalog content resolution".

   2 Optional: Specify the priority of the catalog in relation to the other catalogs on the cluster. For more information, see "Catalog selection by priority".

   3 Specify the interval in minutes for polling the remote registry for newer image digests. To disable polling, do not set the field.

   4 Specify the catalog image in the **spec.source.image.ref** field.

2. Add the catalog to your cluster by running the following command:

   ```
   $ oc apply -f my-redhat-operators.yaml
   ```

   Example output

   ```
   clustercatalog.olm.operatorframework.io/my-redhat-operators created
   ```

Verification

- Run the following commands to verify the status of your catalog:

  a. Check if you catalog is available by running the following command:

     ```
     $ oc get clustercatalog
     ```

     Example output

     ```
     NAME                          LASTUNPACKED   SERVING   AGE
     my-redhat-operators           55s            True      64s
     openshift-certified-operators 83m            True      84m
     ```

```
openshift-community-operators   43m           True      84m
openshift-redhat-marketplace    83m           True      84m
openshift-redhat-operators      54m           True      84m
```

b. Check the status of your catalog by running the following command:

```
$ oc describe clustercatalog my-redhat-operators
```

Example output

```
Name:         my-redhat-operators
Namespace:
Labels:       olm.operatorframework.io/metadata.name=my-redhat-operators
Annotations:  <none>
API Version:  olm.operatorframework.io/v1
Kind:         ClusterCatalog
Metadata:
  Creation Timestamp:  2025-02-18T20:28:50Z
  Finalizers:
    olm.operatorframework.io/delete-server-cache
  Generation:        1
  Resource Version:  50248
  UID:               86adf94f-d2a8-4e70-895b-31139f2eeab7
Spec:
  Availability Mode:  Available
  Priority:           1000
  Source:
    Image:
      Poll Interval Minutes:  10
      Ref:                    registry.redhat.io/redhat/community-operator-index:v4.18
    Type:                     Image
Status: (1)
  Conditions:
    Last Transition Time:  2025-02-18T20:29:00Z
    Message:               Successfully unpacked and stored content from resolved source
    Observed Generation:   1
    Reason:                Succeeded (2)
    Status:                True
    Type:                  Progressing
    Last Transition Time:  2025-02-18T20:29:00Z
    Message:               Serving desired content from resolved source
    Observed Generation:   1
    Reason:                Available
    Status:                True
    Type:                  Serving
  Last Unpacked:           2025-02-18T20:28:59Z
  Resolved Source:
    Image:
      Ref:  registry.redhat.io/redhat/community-operator-
index@sha256:11627ea6fdd06b8092df815076e03cae9b7cede8b353c0b461328842d0289
6c5 (3)
    Type:  Image
  Urls:
    Base:  https://catalogd-service.openshift-catalogd.svc/catalogs/my-redhat-operators
Events:    <none>
```

```
-
```

1. Describes the status of the catalog.
2. Displays the reason the catalog is in the current state.
3. Displays the image reference of the catalog.

4.3.4. Deleting a catalog

You can delete a catalog by deleting its custom resource (CR).

Prerequisites

- You have a catalog installed.

Procedure

- Delete a catalog by running the following command:

  ```
  $ oc delete clustercatalog <catalog_name>
  ```

  Example output

  ```
  clustercatalog.olm.operatorframework.io "my-redhat-operators" deleted
  ```

Verification

- Verify the catalog is deleted by running the following command:

  ```
  $ oc get clustercatalog
  ```

4.3.5. Disabling a default catalog

You can disable the Red Hat-provided catalogs that are included with OpenShift Container Platform by default.

Procedure

- Disable a default catalog by running the following command:

  ```
  $ oc patch clustercatalog openshift-certified-operators -p \
    '{"spec": {"availabilityMode": "Unavailable"}}' --type=merge
  ```

  Example output

  ```
  clustercatalog.olm.operatorframework.io/openshift-certified-operators patched
  ```

Verification

- Verify the catalog is disabled by running the following command:

```
$ oc get clustercatalog openshift-certified-operators
```

Example output

```
NAME                           LASTUNPACKED   SERVING   AGE
openshift-certified-operators                 False     6h54m
```

4.4. CATALOG CONTENT RESOLUTION

When you specify the cluster extension you want to install in a custom resource (CR), Operator Lifecycle Manager (OLM) v1 uses catalog selection to resolve what content is installed.

You can perform the following actions to control the selection of catalog content:

- Specify labels to select the catalog.
- Use match expressions to perform complex filtering across catalogs.
- Set catalog priority.

If you do not specify any catalog selection criteria, Operator Lifecycle Manager (OLM) v1 selects an extension from any available catalog on the cluster that provides the requested package.

During resolution, bundles that are not deprecated are preferred over deprecated bundles by default.

CHAPTER 5. CATALOG SELECTION BY NAME

When a catalog is added to a cluster, a label is created by using the value of the **metadata.name** field of the catalog custom resource (CR). In the CR of an extension, you can specify the catalog name by using the **spec.source.catalog.selector.matchLabels** field. The value of the **matchLabels** field uses the following format:

Example label derived from the metadata.name field

```
apiVersion: olm.operatorframework.io/v1
kind: ClusterExtension
metadata:
  name: <example_extension>
  labels:
    olm.operatorframework.io/metadata.name: <example_extension> (1)
...
```

(1) A label derived from the **metadata.name** field and automatically added when the catalog is applied.

The following example resolves the **<example_extension>-operator** package from a catalog with the **openshift-redhat-operators** label:

Example extension CR

```
apiVersion: olm.operatorframework.io/v1
kind: ClusterExtension
metadata:
  name: <example_extension>
spec:
  namespace: <example_namespace>
  serviceAccount:
    name: <example_extension>-installer
  source:
    sourceType: Catalog
    catalog:
      packageName: <example_extension>-operator
      selector:
        matchLabels:
          olm.operatorframework.io/metadata.name: openshift-redhat-operators
```

CHAPTER 6. CATALOG SELECTION BY LABELS OR EXPRESSIONS

You can add metadata to a catalog by using labels in the custom resource (CR) of a cluster catalog. You can then filter catalog selection by specifying the assigned labels or using expressions in the CR of the cluster extension.

The following cluster catalog CR adds the **example.com/support** label with the value of **true** to the **catalog-a** cluster catalog:

Example cluster catalog CR with labels

```
apiVersion: olm.operatorframework.io/v1
kind: ClusterCatalog
metadata:
  name: catalog-a
  labels:
    example.com/support: "true"
spec:
  source:
    type: Image
    image:
      ref: quay.io/example/content-management-a:latest
```

The following cluster extension CR uses the **matchLabels** selector to select catalogs with the **example.com/support** label and the value of **true**:

Example cluster extension CR with matchLabels selector

```
apiVersion: olm.operatorframework.io/v1
kind: ClusterExtension
metadata:
  name: <example_extension>
spec:
  namespace: <example_namespace>
  serviceAccount:
    name: <example_extension>-installer
  source:
    sourceType: Catalog
    catalog:
      packageName: <example_extension>-operator
      selector:
        matchLabels:
          example.com/support: "true"
```

You can use the **matchExpressions** field to perform more complex filtering for labels. The following cluster extension CR selects catalogs with the **example.com/support** label and a value of **production** or **supported**:

Example cluster extension CR with matchExpression selector

```
apiVersion: olm.operatorframework.io/v1
kind: ClusterExtension
metadata:
  name: <example_extension>
spec:
  namespace: <example_namespace>
  serviceAccount:
    name: <example_extension>-installer
  source:
    sourceType: Catalog
    catalog:
      packageName: <example_extension>-operator
      selector:
        matchExpressions:
          - key: example.com/support
            operator: In
            values:
              - "production"
              - "supported"
```

NOTE

If you use both the **matchLabels** and **matchExpressions** fields, the selected catalog must satisfy all specified criteria.

CHAPTER 7. CATALOG EXCLUSION BY LABELS OR EXPRESSIONS

You can exclude catalogs by using match expressions on metadata with the **NotIn** or **DoesNotExist** operators.

The following CRs add an **example.com/testing** label to the **unwanted-catalog-1** and **unwanted-catalog-2** cluster catalogs:

Example cluster catalog CR

```
apiVersion: olm.operatorframework.io/v1
kind: ClusterCatalog
metadata:
  name: unwanted-catalog-1
  labels:
    example.com/testing: "true"
spec:
  source:
    type: Image
    image:
      ref: quay.io/example/content-management-a:latest
```

Example cluster catalog CR

```
apiVersion: olm.operatorframework.io/v1
kind: ClusterCatalog
metadata:
  name: unwanted-catalog-2
  labels:
    example.com/testing: "true"
spec:
  source:
    type: Image
    image:
      ref: quay.io/example/content-management-b:latest
```

The following cluster extension CR excludes selection from the **unwanted-catalog-1** catalog:

Example cluster extension CR that excludes a specific catalog

```
apiVersion: olm.operatorframework.io/v1
kind: ClusterExtension
metadata:
  name: <example_extension>
spec:
  namespace: <example_namespace>
  serviceAccount:
    name: <example_extension>-installer
  source:
    sourceType: Catalog
    catalog:
      packageName: <example_extension>-operator
      selector:
        matchExpressions:
          - key: olm.operatorframework.io/metadata.name
            operator: NotIn
            values:
              - unwanted-catalog-1
```

The following cluster extension CR selects from catalogs that do not have the **example.com/testing** label. As a result, both **unwanted-catalog-1** and **unwanted-catalog-2** are excluded from catalog selection.

Example cluster extension CR that excludes catalogs with a specific label

```
apiVersion: olm.operatorframework.io/v1
kind: ClusterExtension
metadata:
  name: <example_extension>
spec:
  namespace: <example_namespace>
  serviceAccount:
    name: <example_extension>-installer
  source:
    sourceType: Catalog
    catalog:
      packageName: <example_extension>-operator
      selector:
        matchExpressions:
          - key: example.com/testing
            operator: DoesNotExist
```

CHAPTER 8. CATALOG SELECTION BY PRIORITY

When multiple catalogs provide the same package, you can resolve ambiguities by specifying the priority in the custom resource (CR) of each catalog. If unspecified, catalogs have a default priority value of **0**. The priority can be any positive or negative 32-bit integer.

NOTE

- During bundle resolution, catalogs with higher priority values are selected over catalogs with lower priority values.
- Bundles that are not deprecated are prioritized over bundles that are deprecated.
- If multiple bundles exist in catalogs with the same priority and the catalog selection is ambiguous, an error is printed.

Example cluster catalog CR with a higher priority

```yaml
apiVersion: olm.operatorframework.io/v1
kind: ClusterCatalog
metadata:
  name: high-priority-catalog
spec:
  priority: 1000
  source:
    type: Image
    image:
      ref: quay.io/example/higher-priority-catalog:latest
```

Example cluster catalog CR with a lower priority

```yaml
apiVersion: olm.operatorframework.io/v1
kind: ClusterCatalog
metadata:
  name: lower-priority-catalog
spec:
  priority: 10
  source:
    type: Image
    image:
      ref: quay.io/example/lower-priority-catalog:latest
```

CHAPTER 9. TROUBLESHOOTING CATALOG SELECTION ERRORS

If bundle resolution fails because of ambiguity or because no catalog is selected, an error message is printed in the **status.conditions** field of the cluster extension.

Perform the following actions to troubleshoot catalog selection errors:

- Refine your selection criteria using labels or expressions.
- Adjust your catalog priorities.
- Ensure that only one bundle matches your package name and version requirements.

9.1. CREATING CATALOGS

Catalog maintainers can create new catalogs in the file-based catalog format for use with Operator Lifecycle Manager (OLM) v1 on OpenShift Container Platform.

9.1.1. Creating a file-based catalog image

You can use the **opm** CLI to create a catalog image that uses the plain text *file-based catalog* format (JSON or YAML), which replaces the deprecated SQLite database format.

Prerequisites

- You have installed the **opm** CLI.
- You have **podman** version 1.9.3+.
- A bundle image is built and pushed to a registry that supports Docker v2-2.

Procedure

1. Initialize the catalog:
   a. Create a directory for the catalog by running the following command:

```
$ mkdir <catalog_dir>
```

   b. Generate a Dockerfile that can build a catalog image by running the **opm generate dockerfile** command:

```
$ opm generate dockerfile <catalog_dir> \
    -i registry.redhat.io/openshift4/ose-operator-registry-rhel9:v4.18 1
```

1 Specify the official Red Hat base image by using the **-i** flag, otherwise the Dockerfile uses the default upstream image.

The Dockerfile must be in the same parent directory as the catalog directory that you created in the previous step:

Example directory structure

```
. 1
├── <catalog_dir> 2
└── <catalog_dir>.Dockerfile 3
```

1. Parent directory
2. Catalog directory
3. Dockerfile generated by the **opm generate dockerfile** command

c. Populate the catalog with the package definition for your Operator by running the **opm init** command:

```
$ opm init <operator_name> \ 1
    --default-channel=preview \ 2
    --description=./README.md \ 3
    --icon=./operator-icon.svg \ 4
    --output yaml \ 5
    > <catalog_dir>/index.yaml 6
```

1. Operator, or package, name
2. Channel that subscriptions default to if unspecified
3. Path to the Operator's **README.md** or other documentation
4. Path to the Operator's icon
5. Output format: JSON or YAML
6. Path for creating the catalog configuration file

This command generates an **olm.package** declarative config blob in the specified catalog configuration file.

2. Add a bundle to the catalog by running the **opm render** command:

```
$ opm render <registry>/<namespace>/<bundle_image_name>:<tag> \ 1
    --output=yaml \
    >> <catalog_dir>/index.yaml 2
```

1. Pull spec for the bundle image
2. Path to the catalog configuration file

NOTE

Channels must contain at least one bundle.

3. Add a channel entry for the bundle. For example, modify the following example to your specifications, and add it to your **<catalog_dir>/index.yaml** file:

Example channel entry

```
---
schema: olm.channel
package: <operator_name>
name: preview
entries:
 - name: <operator_name>.v0.1.0 (1)
```

(1) Ensure that you include the period (**.**) after **<operator_name>** but before the **v** in the version. Otherwise, the entry fails to pass the **opm validate** command.

4. Validate the file-based catalog:

   a. Run the **opm validate** command against the catalog directory:

   ```
   $ opm validate <catalog_dir>
   ```

   b. Check that the error code is **0**:

   ```
   $ echo $?
   ```

   #### Example output

   ```
   0
   ```

5. Build the catalog image by running the **podman build** command:

   ```
   $ podman build . \
       -f <catalog_dir>.Dockerfile \
       -t <registry>/<namespace>/<catalog_image_name>:<tag>
   ```

6. Push the catalog image to a registry:

   a. If required, authenticate with your target registry by running the **podman login** command:

   ```
   $ podman login <registry>
   ```

   b. Push the catalog image by running the **podman push** command:

   ```
   $ podman push <registry>/<namespace>/<catalog_image_name>:<tag>
   ```

Additional resources

- **opm** CLI reference

9.1.2. Updating or filtering a file-based catalog image

You can use the **opm** CLI to update or filter a catalog image that uses the file-based catalog format. By extracting the contents of an existing catalog image, you can modify the catalog as needed, for example:

- Adding packages

- Removing packages
- Updating existing package entries
- Detailing deprecation messages per package, channel, and bundle

You can then rebuild the image as an updated version of the catalog.

> **NOTE**
>
> Alternatively, if you already have a catalog image on a mirror registry, you can use the oc-mirror CLI plugin to automatically prune any removed images from an updated source version of that catalog image while mirroring it to the target registry.
>
> For more information about the oc-mirror plugin and this use case, see the "Keeping your mirror registry content updated" section, and specifically the "Pruning images" subsection, of "Mirroring images for a disconnected installation using the oc-mirror plugin".

Prerequisites

- You have the following on your workstation:
  - The **opm** CLI.
  - **podman** version 1.9.3+.
  - A file-based catalog image.
  - A catalog directory structure recently initialized on your workstation related to this catalog. If you do not have an initialized catalog directory, create the directory and generate the Dockerfile. For more information, see the "Initialize the catalog" step from the "Creating a file-based catalog image" procedure.

Procedure

1. Extract the contents of the catalog image in YAML format to an **index.yaml** file in your catalog directory:

   ```
   $ opm render <registry>/<namespace>/<catalog_image_name>:<tag> \
       -o yaml > <catalog_dir>/index.yaml
   ```

   > **NOTE**
   >
   > Alternatively, you can use the **-o json** flag to output in JSON format.

2. Modify the contents of the resulting **index.yaml** file to your specifications:

   > **IMPORTANT**
   >
   > After a bundle has been published in a catalog, assume that one of your users has installed it. Ensure that all previously published bundles in a catalog have an update path to the current or newer channel head to avoid stranding users that have that version installed.

- To add an Operator, follow the steps for creating package, bundle, and channel entries in the "Creating a file-based catalog image" procedure.
- To remove an Operator, delete the set of **olm.package**, **olm.channel**, and **olm.bundle** blobs that relate to the package. The following example shows a set that must be deleted to remove the **example-operator** package from the catalog:

Example 9.1. Example removed entries

```
---
defaultChannel: release-2.7
icon:
  base64data: <base64_string>
  mediatype: image/svg+xml
name: example-operator
schema: olm.package
---
entries:
- name: example-operator.v2.7.0
  skipRange: '>=2.6.0 <2.7.0'
- name: example-operator.v2.7.1
  replaces: example-operator.v2.7.0
  skipRange: '>=2.6.0 <2.7.1'
- name: example-operator.v2.7.2
  replaces: example-operator.v2.7.1
  skipRange: '>=2.6.0 <2.7.2'
- name: example-operator.v2.7.3
  replaces: example-operator.v2.7.2
  skipRange: '>=2.6.0 <2.7.3'
- name: example-operator.v2.7.4
  replaces: example-operator.v2.7.3
  skipRange: '>=2.6.0 <2.7.4'
name: release-2.7
package: example-operator
schema: olm.channel
---
image: example.com/example-inc/example-operator-bundle@sha256:<digest>
name: example-operator.v2.7.0
package: example-operator
properties:
- type: olm.gvk
  value:
    group: example-group.example.io
    kind: MyObject
    version: v1alpha1
- type: olm.gvk
  value:
    group: example-group.example.io
    kind: MyOtherObject
    version: v1beta1
- type: olm.package
  value:
    packageName: example-operator
    version: 2.7.0
- type: olm.bundle.object
  value:
    data: <base64_string>
```

```
- type: olm.bundle.object
  value:
    data: <base64_string>
relatedImages:
- image: example.com/example-inc/example-related-image@sha256:<digest>
  name: example-related-image
schema: olm.bundle
---
```

- To add or update deprecation messages for an Operator, ensure there is a **deprecations.yaml** file in the same directory as the package's **index.yaml** file. For information on the **deprecations.yaml** file format, see "olm.deprecations schema".

3. Save your changes.
4. Validate the catalog:

```
$ opm validate <catalog_dir>
```

5. Rebuild the catalog:

```
$ podman build . \
    -f <catalog_dir>.Dockerfile \
    -t <registry>/<namespace>/<catalog_image_name>:<tag>
```

6. Push the updated catalog image to a registry:

```
$ podman push <registry>/<namespace>/<catalog_image_name>:<tag>
```

Verification

1. In the web console, navigate to the OperatorHub configuration resource in the **Administration → Cluster Settings → Configuration** page.
2. Add the catalog source or update the existing catalog source to use the pull spec for your updated catalog image.
   For more information, see "Adding a catalog source to a cluster" in the "Additional resources" of this section.
3. After the catalog source is in a **READY** state, navigate to the **Operators → OperatorHub** page and check that the changes you made are reflected in the list of Operators.

Additional resources

- Packaging format → Schemas → olm.deprecations schema
- Mirroring images for a disconnected installation using the oc-mirror plugin → Keeping your mirror registry content updated
- Adding a catalog source to a cluster

9.2. DISCONNECTED ENVIRONMENT SUPPORT IN OLM V1

To support cluster administrators that prioritize high security by running their clusters in internet-disconnected environments, especially for mission-critical production workloads, Operator Lifecycle Manager (OLM) v1 includes cluster extension lifecycle management functionality that works within these disconnected environments, starting in OpenShift Container Platform 4.18.

9.2.1. About disconnected support and the oc-mirror plugin in OLM v1

Operator Lifecycle Manager (OLM) v1 supports disconnected environments starting in OpenShift Container Platform 4.18. After using the oc-mirror plugin for the OpenShift CLI (**oc**) to mirror the images required for your cluster to a mirror registry in your fully or partially disconnected environments, OLM v1 can function properly in these environments by utilizing either of the following sets of resources, depending on which oc-mirror plugin version you are using:

- **ImageContentSourcePolicy** resources, which are automatically generated by oc-mirror plugin v1, and **ClusterCatalog** resources, which must be manually created after using oc-mirror plugin v1
- **ImageDigestMirrorSet**, **ImageTagMirrorSet**, and **ClusterCatalog** resources, which are all automatically generated by oc-mirror plugin v2

NOTE

Starting in OpenShift Container Platform 4.18, oc-mirror plugin v2 is the recommended version for mirroring.

For more information and procedures, see the *Disconnected environments* guide for the oc-mirror plugin version you plan to use.

Additional resources

- Mirroring images for a disconnected installation using the oc-mirror plugin v1
- Mirroring images for a disconnected installation using the oc-mirror plugin v2

CHAPTER 10. CLUSTER EXTENSIONS

10.1. MANAGING CLUSTER EXTENSIONS

After a catalog has been added to your cluster, you have access to the versions, patches, and over-the-air updates of the extensions and Operators that are published to the catalog.

You can use custom resources (CRs) to manage extensions declaratively from the CLI.

10.1.1. Supported extensions

Currently, Operator Lifecycle Manager (OLM) v1 supports installing cluster extensions that meet all of the following criteria:

- The extension must use the **registry+v1** bundle format introduced in OLM (Classic).
- The extension must support installation via the **AllNamespaces** install mode.
- The extension must not use webhooks.
- The extension must not declare dependencies by using any of the following file-based catalog properties:
  - **olm.gvk.required**
  - **olm.package.required**
  - **olm.constraint**

OLM v1 checks that the extension you want to install meets these constraints. If the extension that you want to install does not meet these constraints, an error message is printed in the cluster extension's conditions.

IMPORTANT

Operator Lifecycle Manager (OLM) v1 does not support the **OperatorConditions** API introduced in OLM (Classic).

If an extension relies on only the **OperatorConditions** API to manage updates, the extension might not install correctly. Most extensions that rely on this API fail at start time, but some might fail during reconciliation.

As a workaround, you can pin your extension to a specific version. When you want to update your extension, consult the extension's documentation to find out when it is safe to pin the extension to a new version.

Additional resources

- Operator conditions

10.1.2. Finding Operators to install from a catalog

After you add a catalog to your cluster, you can query the catalog to find Operators and extensions to install.

Currently in Operator Lifecycle Manager (OLM) v1, you cannot query on-cluster catalogs managed by catalogd. In OLM v1, you must use the **opm** and **jq** CLI tools to query the catalog registry.

Prerequisites

- You have added a catalog to your cluster.
- You have installed the **jq** CLI tool.
- You have installed the **opm** CLI tool.

Procedure

1. To return a list of extensions that support the **AllNamespaces** install mode and do not use webhooks, enter the following command:

```
$ opm render <catalog_registry_url>:<tag> \
  | jq -cs '[.[] | select(.schema == "olm.bundle" \
  and (.properties[] | select(.type == "olm.csv.metadata").value.installModes[] \
  | select(.type == "AllNamespaces" and .supported == true)) \
  and .spec.webhookdefinitions == null) | .package] | unique[]'
```

where:

catalog_registry_url

Specifies the URL of the catalog registry, such as **registry.redhat.io/redhat/redhat-operator-index**.

tag

Specifies the tag or version of the catalog, such as **v4.18** or **latest**.

Example 10.1. Example command

```
$ opm render \
  registry.redhat.io/redhat/redhat-operator-index:v4.18 \
  | jq -cs '[.[] | select(.schema == "olm.bundle" \
  and (.properties[] | select(.type == "olm.csv.metadata").value.installModes[] \
  | select(.type == "AllNamespaces" and .supported == true)) \
  and .spec.webhookdefinitions == null) | .package] | unique[]'
```

Example 10.2. Example output

```
"3scale-operator"
"amq-broker-rhel8"
"amq-online"
"amq-streams"
"amq-streams-console"
"ansible-automation-platform-operator"
"ansible-cloud-addons-operator"
"apicast-operator"
"authorino-operator"
"aws-load-balancer-operator"
"bamoe-kogito-operator"
"cephcsi-operator"
```

```
"cincinnati-operator"
"cluster-logging"
"cluster-observability-operator"
"compliance-operator"
"container-security-operator"
"cryostat-operator"
"datagrid"
"devspaces"
...
```

2. Inspect the contents of an extension's metadata by running the following command:

```
$ opm render <catalog_registry_url>:<tag> \
  | jq -s '.[] | select( .schema == "olm.package") \
  | select( .name == "<package_name>")'
```

Example 10.3. Example command

```
$ opm render \
  registry.redhat.io/redhat/redhat-operator-index:v4.18 \
  | jq -s '.[] | select( .schema == "olm.package") \
  | select( .name == "openshift-pipelines-operator-rh")'
```

Example 10.4. Example output

```
{
  "schema": "olm.package",
  "name": "openshift-pipelines-operator-rh",
  "defaultChannel": "latest",
  "icon": {
    "base64data": "iVBORw0KGgoAAAANSUhE...",
    "mediatype": "image/png"
  }
}
```

10.1.2.1. Common catalog queries

You can query catalogs by using the **opm** and **jq** CLI tools. The following tables show common catalog queries that you can use when installing, updating, and managing the lifecycle of extensions.

Command syntax

```
$ opm render <catalog_registry_url>:<tag> | <jq_request>
```

where:

catalog_registry_url

Specifies the URL of the catalog registry, such as **registry.redhat.io/redhat/redhat-operator-index**.

tag
: Specifies the tag or version of the catalog, such as **v4.18** or **latest**.

jq_request
: Specifies the query you want to run on the catalog.

Example 10.5. Example command

```
$ opm render \
  registry.redhat.io/redhat/redhat-operator-index:v4.18 \
  | jq -cs '[.[] | select(.schema == "olm.bundle" and (.properties[] \
  | select(.type == "olm.csv.metadata").value.installModes[] \
  | select(.type == "AllNamespaces" and .supported == true)) \
  and .spec.webhookdefinitions == null) \
  | .package] | unique[]'
```

Table 10.1. Common package queries

Query	Request

Available packages in a catalog	`$ opm render <catalog_registry_url>:<tag> \`
	`  | jq -s '.[] | select( .schema == "olm.package")'`

Packages that support **AllNamespaces** install mode and do not use webhooks	`$ opm render <catalog_registry_url>:<tag> \`
	`  | jq -cs '[.[] | select(.schema == "olm.bundle" and (.properties[] \`
	`  | select(.type == "olm.csv.metadata").value.installModes[] \`
	`  | select(.type == "AllNamespaces" and .supported == true)) \`
	`  and .spec.webhookdefinitions == null) \`
	`  | .package] | unique[]'`

Package metadata	`$ opm render <catalog_registry_url>:<tag> \`
	`  | jq -s '.[] | select( .schema == "olm.package") \`
	`  | select( .name == "<package_name>")'`

Catalog blobs in a package	`$ opm render <catalog_registry_url>:<tag> \`
	`  | jq -s '.[] | select( .package == "<package_name>")'`

Table 10.2. Common channel queries

Query	Request

Query	Request
Channels in a package	`$ opm render <catalog_registry_url>:<tag> \	jq -s '.[]	select(.schema == "olm.channel") \	select(.package == "<package_name>")	.name'`
Versions in a channel	`$ opm render <catalog_registry_url>:<tag> \	jq -s '.[]	select(.package == "<package_name>") \	select(.schema == "olm.channel") \	select(.name == "<channel_name>") .entries \	.[]	.name'`
● Latest version in a channel ● Upgrade path	`$ opm render <catalog_registry_url>:<tag> \	jq -s '.[]	select(.schema == "olm.channel") \	select (.name == "<channel_name>") \	select(.package == "<package_name>")'`

Table 10.3. Common bundle queries

Query	Request
Bundles in a package	`$ opm render <catalog_registry_url>:<tag> \	jq -s '.[]	select(.schema == "olm.bundle") \	select(.package == "<package_name>")	.name'`
● Bundle dependencies ● Available APIs	`$ opm render <catalog_registry_url>:<tag> \	jq -s '.[]	select(.schema == "olm.bundle") \	select (.name == "<bundle_name>") \	select(.package == "<package_name>")'`

10.1.3. Cluster extension permissions

In Operator Lifecycle Manager (OLM) Classic, a single service account with cluster administrator privileges manages all cluster extensions.

OLM v1 is designed to be more secure than OLM (Classic) by default. OLM v1 manages a cluster extension by using the service account specified in an extension's custom resource (CR). Cluster administrators can create a service account for each cluster extension. As a result, administrators can follow the principle of least privilege and assign only the role-based access controls (RBAC) to install and manage that extension.

You must add each permission to either a cluster role or role. Then you must bind the cluster role or role to the service account with a cluster role binding or role binding.

You can scope the RBAC to either the cluster or to a namespace. Use cluster roles and cluster role bindings to scope permissions to the cluster. Use roles and role bindings to scope permissions to a namespace. Whether you scope the permissions to the cluster or to a namespace depends on the design of the extension you want to install and manage.

IMPORTANT

To simply the following procedure and improve readability, the following example manifest uses permissions that are scoped to the cluster. You can further restrict some of the permissions by scoping them to the namespace of the extension instead of the cluster.

If a new version of an installed extension requires additional permissions, OLM v1 halts the update process until a cluster administrator grants those permissions.

10.1.3.1. Creating a namespace

Before you create a service account to install and manage your cluster extension, you must create a namespace.

Prerequisites

- Access to an OpenShift Container Platform cluster using an account with **cluster-admin** permissions.

Procedure

- Create a new namespace for the service account of the extension that you want to install by running the following command:

```
$ oc adm new-project <new_namespace>
```

10.1.3.2. Creating a service account for an extension

You must create a service account to install, manage, and update a cluster extension.

Prerequisites

- Access to an OpenShift Container Platform cluster using an account with **cluster-admin** permissions.

Procedure

1. Create a service account, similar to the following example:

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: <extension>-installer
  namespace: <namespace>
```

Example 10.6. Example extension-service-account.yaml file

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: pipelines-installer
  namespace: pipelines
```

2. Apply the service account by running the following command:

```
$ oc apply -f extension-service-account.yaml
```

10.1.3.3. Downloading the bundle manifests of an extension

Use the **opm** CLI tool to download the bundle manifests of the extension that you want to install. Use the CLI tool or text editor of your choice to view the manifests and find the required permissions to install and manage the extension.

Prerequisites

- You have access to an OpenShift Container Platform cluster using an account with **cluster-admin** permissions.
- You have decided which extension you want to install.
- You have installed the **opm** CLI tool.

Procedure

1. Inspect the available versions and images of the extension you want to install by running the following command:

```
$ opm render <registry_url>:<tag_or_version> | \
  jq -cs '.[] | select( .schema == "olm.bundle" ) | \
  select( .package == "<extension_name>") | \
  {"name":.name, "image":.image}'
```

Example 10.7. Example command

```
$ opm render registry.redhat.io/redhat/redhat-operator-index:v4.18 | \
  jq -cs '.[] | select( .schema == "olm.bundle" ) | \
  select( .package == "openshift-pipelines-operator-rh") | \
  {"name":.name, "image":.image}'
```

Example 10.8. Example output

```
{"name":"openshift-pipelines-operator-rh.v1.14.3","image":"registry.redhat.io/openshift-
pipelines/pipelines-operator-
bundle@sha256:3f64b29f6903981470d0917b2557f49d84067bccdba0544bfe874ec4412f45
b0"}
{"name":"openshift-pipelines-operator-rh.v1.14.4","image":"registry.redhat.io/openshift-
pipelines/pipelines-operator-
bundle@sha256:dd3d18367da2be42539e5dde8e484dac3df33ba3ce1d5bcf896838954f386
4ec"}
{"name":"openshift-pipelines-operator-rh.v1.14.5","image":"registry.redhat.io/openshift-
pipelines/pipelines-operator-
bundle@sha256:f7b19ce26be742c4aaa458d37bc5ad373b5b29b20aaa7d308349687d3cbd
8838"}
{"name":"openshift-pipelines-operator-rh.v1.15.0","image":"registry.redhat.io/openshift-
pipelines/pipelines-operator-
bundle@sha256:22be152950501a933fe6e1df0e663c8056ca910a89dab3ea801c3bb2dc2bf
1e6"}
{"name":"openshift-pipelines-operator-rh.v1.15.1","image":"registry.redhat.io/openshift-
pipelines/pipelines-operator-
bundle@sha256:64afb32e3640bb5968904b3d1a317e9dfb307970f6fda0243e2018417207f
d75"}
{"name":"openshift-pipelines-operator-rh.v1.15.2","image":"registry.redhat.io/openshift-
pipelines/pipelines-operator-
bundle@sha256:8a593c1144709c9aeffbeb68d0b4b08368f528e7bb6f595884b2474bcfbcafc
d"}
{"name":"openshift-pipelines-operator-rh.v1.16.0","image":"registry.redhat.io/openshift-
pipelines/pipelines-operator-
bundle@sha256:a46b7990c0ad07dae78f43334c9bd5e6cba7b50ca60d3f880099b71e77bed
214"}
{"name":"openshift-pipelines-operator-rh.v1.16.1","image":"registry.redhat.io/openshift-
pipelines/pipelines-operator-
bundle@sha256:29f27245e93b3f605647993884751c490c4a44070d3857a878d2aee87d43f
85b"}
{"name":"openshift-pipelines-operator-rh.v1.16.2","image":"registry.redhat.io/openshift-
pipelines/pipelines-operator-
bundle@sha256:2037004666526c90329f4791f14cb6cc06e8775cb84ba107a24cc4c2cf9446
49"}
{"name":"openshift-pipelines-operator-rh.v1.17.0","image":"registry.redhat.io/openshift-
pipelines/pipelines-operator-
bundle@sha256:d75065e999826d38408049aa1fde674cd1e45e384bfdc96523f6bad58a0e0
dbc"}
```

2. Make a directory to extract the image of the bundle that you want to install by running the following command:

```
$ mkdir <new_dir>
```

3. Change into the directory by running the following command:

```
$ cd <new_dir>
```

4. Find the image reference of the version that you want to install and run the following command:

```
$ oc image extract <full_path_to_registry_image>@sha256:<sha>
```

Example command

```
$ oc image extract registry.redhat.io/openshift-pipelines/pipelines-operator-
bundle@sha256:f7b19ce26be742c4aaa458d37bc5ad373b5b29b20aaa7d308349687d3cbd883
8
```

5. Change into the **manifests** directory by running the following command:

```
$ cd manifests
```

6. View the contents of the manifests directory by entering the following command. The output lists the manifests of the resources required to install, manage, and operate your extension.

```
$ tree
```

Example 10.9. Example output

```
.
├── manifests
│   ├── config-logging_v1_configmap.yaml
│   ├── openshift-pipelines-operator-
monitor_monitoring.coreos.com_v1_servicemonitor.yaml
│   ├── openshift-pipelines-operator-prometheus-k8s-read-
binding_rbac.authorization.k8s.io_v1_rolebinding.yaml
│   ├── openshift-pipelines-operator-read_rbac.authorization.k8s.io_v1_role.yaml
│   ├── openshift-pipelines-operator-rh.clusterserviceversion.yaml
│   ├── operator.tekton.dev_manualapprovalgates.yaml
│   ├── operator.tekton.dev_openshiftpipelinesascodes.yaml
│   ├── operator.tekton.dev_tektonaddons.yaml
│   ├── operator.tekton.dev_tektonchains.yaml
│   ├── operator.tekton.dev_tektonconfigs.yaml
│   ├── operator.tekton.dev_tektonhubs.yaml
│   ├── operator.tekton.dev_tektoninstallersets.yaml
│   ├── operator.tekton.dev_tektonpipelines.yaml
│   ├── operator.tekton.dev_tektonresults.yaml
│   ├── operator.tekton.dev_tektontriggers.yaml
│   ├── tekton-config-defaults_v1_configmap.yaml
│   ├── tekton-config-observability_v1_configmap.yaml
│   ├── tekton-config-read-
rolebinding_rbac.authorization.k8s.io_v1_clusterrolebinding.yaml
│   ├── tekton-config-read-role_rbac.authorization.k8s.io_v1_clusterrole.yaml
│   ├── tekton-operator-controller-config-leader-election_v1_configmap.yaml
│   ├── tekton-operator-info_rbac.authorization.k8s.io_v1_rolebinding.yaml
│   ├── tekton-operator-info_rbac.authorization.k8s.io_v1_role.yaml
│   ├── tekton-operator-info_v1_configmap.yaml
│   ├── tekton-operator_v1_service.yaml
│   ├── tekton-operator-webhook-certs_v1_secret.yaml
│   ├── tekton-operator-webhook-config-leader-election_v1_configmap.yaml
│   ├── tekton-operator-webhook_v1_service.yaml
│   ├── tekton-result-read-
rolebinding_rbac.authorization.k8s.io_v1_clusterrolebinding.yaml
│   └── tekton-result-read-role_rbac.authorization.k8s.io_v1_clusterrole.yaml
├── metadata
│   ├── annotations.yaml
│   └── properties.yaml
└── root
    └── buildinfo
        ├── content_manifests
        │   └── openshift-pipelines-operator-bundle-container-v1.16.2-3.json
        └── Dockerfile-openshift-pipelines-pipelines-operator-bundle-container-v1.16.2-3
```

Next steps

- View the contents of the **install.spec.clusterpermissions** stanza of cluster service version (CSV) file in the **manifests** directory using your preferred CLI tool or text editor. The following examples reference the **openshift-pipelines-operator-rh.clusterserviceversion.yaml** file of the Red Hat OpenShift Pipelines Operator.
- Keep this file open as a reference while assigning permissions to the cluster role file in the following procedure.

10.1.3.4. Required permissions to install and manage a cluster extension

You must inspect the manifests included in the bundle image of a cluster extension to assign the necessary permissions. The service account requires enough role-based access controls (RBAC) to create and manage the following resources.

IMPORTANT

Follow the principle of least privilege and scope permissions to specific resource names with the least RBAC required to run.

Admission plugins

Because OpenShift Container Platform clusters use the **OwnerReferencesPermissionEnforcement** admission plugin, cluster extensions must have permissions to update the **blockOwnerDeletion** and **ownerReferences** finalizers.

Cluster role and cluster role bindings for the controllers of the extension

You must define RBAC so that the installation service account can create and manage cluster roles and cluster role bindings for the extension controllers.

Cluster service version (CSV)

You must define RBAC for the resources defined in the CSV of the cluster extension.

Cluster-scoped bundle resources

You must define RBAC to create and manage any cluster-scoped resources included in the bundle. If the cluster-scoped resources matches another resource type, such as a **ClusterRole**, you can add the resource to the pre-existing rule under the **resources** or **resourceNames** field.

Custom resource definitions (CRDs)

You must define RBAC so that the installation service account can create and manage the CRDs for the extension. Also, you must grant the service account for the controller of the extension the RBAC to manage its CRDs.

Deployments

You must define RBAC for the installation service account to create and manage the deployments needed by the extension controller, such as services and config maps.

Extension permissions

You must include RBAC for the permissions and cluster permissions defined in the CSV. The installation service account needs the ability to grant these permissions to the extension controller, which needs these permissions to run.

Namespace-scoped bundle resources

You must define RBAC for any namespace-scoped bundle resources. The installation service account requires permission to create and manage resources, such as config maps or services.

Roles and role bindings

You must define RBAC for any roles or role bindings defined in the CSV. The installation service account needs permission to create and manage those roles and role bindings.

Service accounts

You must define RBAC so that the installation service account can create and manage the service accounts for the extension controllers.

10.1.3.5. Creating a cluster role for an extension

You must review the **install.spec.clusterpermissions** stanza of the cluster service version (CSV) and the manifests of an extension carefully to define the required role-based access controls (RBAC) of the extension that you want to install. You must create a cluster role by copying the required RBAC from the CSV to the new manifest.

TIP

If you want to test the process for installing and updating an extension in OLM v1, you can use the following cluster role to grant cluster administrator permissions. This manifest is for testing purposes only. It should not be used in production clusters.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: <extension>-installer-clusterrole
rules:
- apiGroups: ["*"]
  resources: ["*"]
  verbs: ["*"]
```

The following procedure uses the **openshift-pipelines-operator-rh.clusterserviceversion.yaml** file of the Red Hat OpenShift Pipelines Operator as an example. The examples include excerpts of the RBAC required to install and manage the OpenShift Pipelines Operator. For a complete manifest, see "Example cluster role for the Red Hat OpenShift Pipelines Operator".

IMPORTANT

To simply the following procedure and improve readability, the following example manifest uses permissions that are scoped to the cluster. You can further restrict some of the permissions by scoping them to the namespace of the extension instead of the cluster.

Prerequisites

- Access to an OpenShift Container Platform cluster using an account with **cluster-admin** permissions.
- You have downloaded the manifests in the image reference of the extension that you want to install.

Procedure

1. Create a new cluster role manifest, similar to the following example:

Example <extension>-cluster-role.yaml file

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: <extension>-installer-clusterrole
```

2. Edit your cluster role manifest to include permission to update finalizers on the extension, similar to the following example:

Example <extension>-cluster-role.yaml

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: pipelines-installer-clusterrole
rules:
- apiGroups:
  - olm.operatorframework.io
  resources:
  - clusterextensions/finalizers
  verbs:
  - update
  # Scoped to the name of the ClusterExtension
  resourceNames:
  - <metadata_name> (1)
```

(1) Specifies the value from the **metadata.name** field from the custom resource (CR) of the extension.

3. Search for the **clusterrole** and **clusterrolebindings** values in the **rules.resources** field in the extension's CSV file.

- Copy the API groups, resources, verbs, and resource names to your manifest, similar to the following example:

Example cluster role manifest

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: pipelines-installer-clusterrole
rules:
# ...
# ClusterRoles and ClusterRoleBindings for the controllers of the extension
- apiGroups:
  - rbac.authorization.k8s.io
  resources:
  - clusterroles
  verbs:
  - create (1)
  - list
  - watch
- apiGroups:
  - rbac.authorization.k8s.io
  resources:
  - clusterroles
  verbs:
  - get
  - update
  - patch
  - delete
  resourceNames: (2)
  - "*"
- apiGroups:
  - rbac.authorization.k8s.io
  resources:
  - clusterrolebindings
  verbs:
  - create
  - list
  - watch
- apiGroups:
  - rbac.authorization.k8s.io
  resources:
  - clusterrolebindings
  verbs:
  - get
  - update
  - patch
  - delete
  resourceNames:
  - "*"
# ...
```

1. You cannot scope **create**, **list**, and **watch** permissions to specific resource names (the **resourceNames** field). You must scope these permissions to their resources (the **resources** field).

2. Some resource names are generated by using the following format: **<package_name>.<hash>**. After you install the extension, look up the resource names for the cluster roles and cluster role bindings for the controller of the extension. Replace the wildcard characters in this example with the generated names and follow the principle of least privilege.

4. Search for the **customresourcedefinitions** value in the **rules.resources** field in the extension's CSV file.

   - Copy the API groups, resources, verbs, and resource names to your manifest, similar to the following example:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: pipelines-installer-clusterrole
rules:
# ...
# Custom resource definitions of the extension
- apiGroups:
  - apiextensions.k8s.io
  resources:
  - customresourcedefinitions
  verbs:
  - create
  - list
  - watch
- apiGroups:
  - apiextensions.k8s.io
  resources:
  - customresourcedefinitions
  verbs:
  - get
  - update
  - patch
  - delete
  resourceNames:
  - manualapprovalgates.operator.tekton.dev
  - openshiftpipelinesascodes.operator.tekton.dev
  - tektonaddons.operator.tekton.dev
  - tektonchains.operator.tekton.dev
  - tektonconfigs.operator.tekton.dev
  - tektonhubs.operator.tekton.dev
  - tektoninstallersets.operator.tekton.dev
  - tektonpipelines.operator.tekton.dev
  - tektonresults.operator.tekton.dev
  - tektontriggers.operator.tekton.dev
# ...
```

5. Search the CSV file for stanzas with the **permissions** and **clusterPermissions** values in the **rules.resources** spec.

   - Copy the API groups, resources, verbs, and resource names to your manifest, similar to the following example:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: pipelines-installer-clusterrole
rules:
# ...
# Excerpt from install.spec.clusterPermissions
- apiGroups:
  - ''
  resources:
  - nodes
  - pods
  - services
  - endpoints
  - persistentvolumeclaims
  - events
  - configmaps
  - secrets
  - pods/log
  - limitranges
  verbs:
```

```
  - create
  - list
  - watch
  - delete
  - deletecollection
  - patch
  - get
  - update
- apiGroups:
  - extensions
  - apps
  resources:
  - ingresses
  - ingresses/status
  verbs:
  - create
  - list
  - watch
  - delete
  - patch
  - get
  - update
# ...
```

6. Search the CSV file for resources under the **install.spec.deployments** stanza.

   - Copy the API groups, resources, verbs, and resource names to your manifest, similar to the following example:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: pipelines-installer-clusterrole
rules:
# ...
# Excerpt from install.spec.deployments
- apiGroups:
  - apps
  resources:
  - deployments
  verbs:
  - create
  - list
  - watch
- apiGroups:
  - apps
  resources:
  - deployments
  verbs:
  - get
  - update
  - patch
  - delete
  # scoped to the extension controller deployment name
  resourceNames:
```

```
  - openshift-pipelines-operator
  - tekton-operator-webhook
# ...
```

7. Search for the **services** and **configmaps** values in the **rules.resources** field in the extension’s CSV file.

   - Copy the API groups, resources, verbs, and resource names to your manifest, similar to the following example:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: pipelines-installer-clusterrole
rules:
# ...
# Services
- apiGroups:
  - ""
  resources:
  - services
  verbs:
  - create
- apiGroups:
  - ""
  resources:
  - services
  verbs:
  - get
  - list
  - watch
  - update
  - patch
  - delete
  # scoped to the service name
  resourceNames:
  - openshift-pipelines-operator-monitor
  - tekton-operator
  - tekton-operator-webhook
# configmaps
- apiGroups:
  - ""
  resources:
  - configmaps
  verbs:
  - create
- apiGroups:
  - ""
  resources:
  - configmaps
  verbs:
  - get
  - list
  - watch
  - update
  - patch
  - delete
  # scoped to the configmap name
  resourceNames:
  - config-logging
  - tekton-config-defaults
  - tekton-config-observability
  - tekton-operator-controller-config-leader-election
  - tekton-operator-info
  - tekton-operator-webhook-config-leader-election
- apiGroups:
  - operator.tekton.dev
  resources:
  - tekton-config-read-role
  - tekton-result-read-role
  verbs:
  - get
  - watch
  - list
```

8. Add the cluster role manifest to the cluster by running the following command:

```
$ oc apply -f <extension>-installer-clusterrole.yaml
```

Example command

```
$ oc apply -f pipelines-installer-clusterrole.yaml
```

10.1.3.6. Example cluster role for the Red Hat OpenShift Pipelines Operator

See the following example for a complete cluster role manifest for the OpenShift Pipelines Operator.

```
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: pipelines-installer-clusterrole
rules:
- apiGroups:
  - olm.operatorframework.io
  resources:
  - clusterextensions/finalizers
  verbs:
  - update
  # Scoped to the name of the ClusterExtension
  resourceNames:
  - pipes # the value from <metadata.name> from the extension's custom resource (CR)
# ClusterRoles and ClusterRoleBindings for the controllers of the extension
- apiGroups:
  - rbac.authorization.k8s.io
  resources:
  - clusterroles
  verbs:
  - create
  - list
  - watch
- apiGroups:
  - rbac.authorization.k8s.io
  resources:
  - clusterroles
  verbs:
  - get
  - update
  - patch
  - delete
  resourceNames:
  - "*"
- apiGroups:
  - rbac.authorization.k8s.io
  resources:
  - clusterrolebindings
  verbs:
  - create
  - list
  - watch
- apiGroups:
  - rbac.authorization.k8s.io
  resources:
  - clusterrolebindings
  verbs:
  - get
  - update
  - patch
  - delete
  resourceNames:
  - "*"
# Extension's custom resource definitions
- apiGroups:
  - apiextensions.k8s.io
  resources:
  - customresourcedefinitions
  verbs:
  - create
  - list
  - watch
- apiGroups:
  - apiextensions.k8s.io
  resources:
  - customresourcedefinitions
  verbs:
  - get
  - update
  - patch
  - delete
  resourceNames:
  - manualapprovalgates.operator.tekton.dev
  - openshiftpipelinesascodes.operator.tekton.dev
  - tektonaddons.operator.tekton.dev
  - tektonchains.operator.tekton.dev
  - tektonconfigs.operator.tekton.dev
  - tektonhubs.operator.tekton.dev
  - tektoninstallersets.operator.tekton.dev
  - tektonpipelines.operator.tekton.dev
  - tektonresults.operator.tekton.dev
  - tektontriggers.operator.tekton.dev
- apiGroups:
  - ''
  resources:
  - nodes
  - pods
  - services
  - endpoints
  - persistentvolumeclaims
  - events
  - configmaps
  - secrets
  - pods/log
  - limitranges
  verbs:
  - create
  - list
  - watch
  - delete
  - deletecollection
  - patch
  - get
  - update
- apiGroups:
  - extensions
  - apps
  resources:
  - ingresses
  - ingresses/status
  verbs:
  - create
  - list
  - watch
  - delete
  - patch
  - get
  - update
- apiGroups:
  - ''
  resources:
  - namespaces
  verbs:
  - get
  - list
  - create
  - update
  - delete
  - patch
  - watch
- apiGroups:
  - apps
  resources:
  - deployments
  - daemonsets
  - replicasets
  - statefulsets
  - deployments/finalizers
  verbs:
  - delete
  - deletecollection
  - create
  - patch
  - get
  - list
  - update
  - watch
- apiGroups:
  - monitoring.coreos.com
  resources:
  - servicemonitors
  verbs:
  - get
  - create
  - delete
- apiGroups:
  - rbac.authorization.k8s.io
  resources:
  - clusterroles
  - roles
  verbs:
  - delete
  - deletecollection
  - create
  - patch
  - get
  - list
  - update
  - watch
  - bind
  - escalate
- apiGroups:
  - ''
  resources:
  - serviceaccounts
  verbs:
  - get
  - list
  - create
  - update
  - delete
  - patch
  - watch
  - impersonate
- apiGroups:
  - rbac.authorization.k8s.io
  resources:
  - clusterrolebindings
  - rolebindings
  verbs:
```

```
  - get
  - update
  - delete
  - patch
  - create
  - list
  - watch
- apiGroups:
  - apiextensions.k8s.io
  resources:
  - customresourcedefinitions
  - customresourcedefinitions/status
  verbs:
  - get
  - create
  - update
  - delete
  - list
  - patch
  - watch
- apiGroups:
  - admissionregistration.k8s.io
  resources:
  - mutatingwebhookconfigurations
  - validatingwebhookconfigurations
  verbs:
  - get
  - list
  - create
  - update
  - delete
  - patch
  - watch
- apiGroups:
  - build.knative.dev
  resources:
  - builds
  - buildtemplates
  - clusterbuildtemplates
  verbs:
  - get
  - list
  - create
  - update
  - delete
  - patch
  - watch
- apiGroups:
  - extensions
  resources:
  - deployments
  verbs:
  - get
  - list
  - create
  - update
  - delete
  - patch
  - watch
- apiGroups:
  - extensions
  resources:
  - deployments/finalizers
  verbs:
  - get
  - list
  - create
  - update
  - delete
  - patch
  - watch
- apiGroups:
  - operator.tekton.dev
  resources:
  - '*'
  - tektonaddons
  verbs:
  - delete
  - deletecollection
  - create
  - patch
  - get
  - list
  - update
  - watch
- apiGroups:
  - tekton.dev
  - triggers.tekton.dev
  - operator.tekton.dev
  - pipelinesascode.tekton.dev
  resources:
  - '*'
  verbs:
  - add
  - delete
  - deletecollection
  - create
  - patch
  - get
  - list
  - update
  - watch
- apiGroups:
  - dashboard.tekton.dev
  resources:
  - '*'
  - tektonaddons
  verbs:
  - delete
  - deletecollection
  - create
  - patch
  - get
  - list
  - update
  - watch
- apiGroups:
  - security.openshift.io
  resources:
  - securitycontextconstraints
  verbs:
  - use
  - get
  - list
  - create
  - update
  - delete
- apiGroups:
  - events.k8s.io
  resources:
  - events
  verbs:
  - create
- apiGroups:
  - route.openshift.io
  resources:
  - routes
  verbs:
  - delete
  - deletecollection
  - create
  - patch
  - get
  - list
  - update
  - watch
- apiGroups:
  - coordination.k8s.io
  resources:
  - leases
  verbs:
  - get
  - list
  - create
  - update
  - delete
  - patch
  - watch
- apiGroups:
  - console.openshift.io
  resources:
  - consoleyamlsamples
  - consoleclidownloads
  - consolequickstarts
  - consolelinks
  verbs:
  - delete
  - deletecollection
  - create
  - patch
  - get
  - list
  - update
  - watch
- apiGroups:
  - autoscaling
  resources:
  - horizontalpodautoscalers
  verbs:
  - delete
  - create
  - patch
  - get
  - list
  - update
  - watch
- apiGroups:
  - policy
  resources:
  - poddisruptionbudgets
  verbs:
  - delete
  - deletecollection
  - create
  - patch
  - get
  - list
  - update
  - watch
- apiGroups:
  - monitoring.coreos.com
  resources:
  - servicemonitors
  verbs:
  - delete
  - deletecollection
  - create
  - patch
  - get
  - list
  - update
  - watch
- apiGroups:
  - batch
  resources:
  - jobs
  - cronjobs
  verbs:
  - delete
  - deletecollection
  - create
  - patch
  - get
  - list
  - update
  - watch
- apiGroups:
  - ''
  resources:
  - namespaces/finalizers
  verbs:
  - update
- apiGroups:
  - resolution.tekton.dev
  resources:
  - resolutionrequests
  - resolutionrequests/status
  verbs:
  - get
  - list
  - watch
  - create
  - delete
  - update
  - patch
- apiGroups:
  - console.openshift.io
  resources:
  - consoleplugins
  verbs:
  - get
  - list
  - watch
  - create
  - delete
  - update
  - patch
# Deployments specified in install.spec.deployments
- apiGroups:
  - apps
  resources:
  - deployments
  verbs:
  - create
  - list
  - watch
- apiGroups:
  - apps
  resources:
  - deployments
  verbs:
  - get
  - update
  - patch
  - delete
  # scoped to the extension controller deployment name
  resourceNames:
  - openshift-pipelines-operator
  - tekton-operator-webhook
# Service accounts in the CSV
```

```
- apiGroups:
  - ""
  resources:
  - serviceaccounts
  verbs:
  - create
  - list
  - watch
- apiGroups:
  - ""
  resources:
  - serviceaccounts
  verbs:
  - get
  - update
  - patch
  - delete
  # scoped to the extension controller's deployment service account
  resourceNames:
  - openshift-pipelines-operator
# Services
- apiGroups:
  - ""
  resources:
  - services
  verbs:
  - create
- apiGroups:
  - ""
  resources:
  - services
  verbs:
  - get
  - list
  - watch
  - update
  - patch
  - delete
  # scoped to the service name
  resourceNames:
  - openshift-pipelines-operator-monitor
  - tekton-operator
  - tekton-operator-webhook
# configmaps
- apiGroups:
  - ""
  resources:
  - configmaps
  verbs:
  - create
- apiGroups:
  - ""
  resources:
  - configmaps
  verbs:
  - get
  - list
  - watch
  - update
  - patch
  - delete
  # scoped to the configmap name
  resourceNames:
  - config-logging
  - tekton-config-defaults
  - tekton-config-observability
  - tekton-operator-controller-config-leader-election
  - tekton-operator-info
  - tekton-operator-webhook-config-leader-election
- apiGroups:
  - operator.tekton.dev
  resources:
  - tekton-config-read-role
  - tekton-result-read-role
  verbs:
  - get
  - watch
  - list
---
```

10.1.3.7. Creating a cluster role binding for an extension

After you have created a service account and cluster role, you must bind the cluster role to the service account with a cluster role binding manifest.

Prerequisites

- Access to an OpenShift Container Platform cluster using an account with **cluster-admin** permissions.
- You have created and applied the following resources for the extension you want to install:
  - Namespace
  - Service account
  - Cluster role

Procedure

1. Create a cluster role binding to bind the cluster role to the service account, similar to the following example:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: <extension>-installer-binding
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: <extension>-installer-clusterrole
```

```
subjects:
- kind: ServiceAccount
  name: <extension>-installer
  namespace: <namespace>
```

Example 10.10. Example **pipelines-cluster-role-binding.yaml** file

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: pipelines-installer-binding
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: pipelines-installer-clusterrole
subjects:
- kind: ServiceAccount
  name: pipelines-installer
  namespace: pipelines
```

2. Apply the cluster role binding by running the following command:

```
$ oc apply -f pipelines-cluster-role-binding.yaml
```

10.1.4. Installing a cluster extension from a catalog

You can install an extension from a catalog by creating a custom resource (CR) and applying it to the cluster. Operator Lifecycle Manager (OLM) v1 supports installing cluster extensions, including OLM (Classic) Operators in the **registry+v1** bundle format, that are scoped to the cluster. For more information, see *Supported extensions*.

Prerequisites

- You have created a service account and assigned enough role-based access controls (RBAC) to install, update, and manage the extension that you want to install. For more information, see "Cluster extension permissions".

Procedure

1. Create a CR, similar to the following example:

```
apiVersion: olm.operatorframework.io/v1
  kind: ClusterExtension
  metadata:
    name: <clusterextension_name>
  spec:
    namespace: <installed_namespace> 1
    serviceAccount:
      name: <service_account_installer_name> 2
    source:
      sourceType: Catalog
      catalog:
        packageName: <package_name>
        channels:
          - <channel_name> (3)
        version: <version_or_version_range> (4)
        upgradeConstraintPolicy: CatalogProvided (5)
```

(1) Specifies the namespace where you want the bundle installed, such as **pipelines** or **my-extension**. Extensions are still cluster-scoped and might contain resources that are installed in different namespaces.

(2) Specifies the name of the service account you created to install, update, and manage your extension.

(3) Optional: Specifies channel names as an array, such as **pipelines-1.14** or **latest**.

(4) Optional: Specifies the version or version range, such as **1.14.0**, **1.14.x**, or **>=1.16**, of the package you want to install or update. For more information, see "Example custom resources (CRs) that specify a target version" and "Support for version ranges".

(5) Optional: Specifies the upgrade constraint policy. If unspecified, the default setting is **CatalogProvided**. The **CatalogProvided** setting only updates if the new version satisfies the upgrade constraints set by the package author. To force an update or rollback, set the field to **SelfCertified**. For more information, see "Forcing an update or rollback".

Example pipelines-operator.yaml CR

```
apiVersion: olm.operatorframework.io/v1
kind: ClusterExtension
metadata:
  name: pipelines-operator
spec:
  namespace: pipelines
  serviceAccount:
    name: pipelines-installer
  source:
    sourceType: Catalog
    catalog:
      packageName: openshift-pipelines-operator-rh
      version: "1.14.x"
```

1. Apply the CR to the cluster by running the following command:

   ```
   $ oc apply -f pipeline-operator.yaml
   ```

   Example output

   ```
   clusterextension.olm.operatorframework.io/pipelines-operator created
   ```

Verification

1. View the Operator or extension’s CR in the YAML format by running the following command:

   ```
   $ oc get clusterextension pipelines-operator -o yaml
   ```

Example 10.11. Example output

```
apiVersion: v1
items:
- apiVersion: olm.operatorframework.io/v1
  kind: ClusterExtension
  metadata:
    annotations:
      kubectl.kubernetes.io/last-applied-configuration: |
        {"apiVersion":"olm.operatorframework.io/v1","kind":"ClusterExtension","metadata":
{"annotations":{},"name":"pipes"},"spec":{"namespace":"pipelines","serviceAccount":
{"name":"pipelines-installer"},"source":{"catalog":{"packageName":"openshift-pipelines-
operator-rh","version":"1.14.x"},"sourceType":"Catalog"}}}
    creationTimestamp: "2025-02-18T21:48:13Z"
    finalizers:
    - olm.operatorframework.io/cleanup-unpack-cache
    - olm.operatorframework.io/cleanup-contentmanager-cache
    generation: 1
    name: pipelines-operator
    resourceVersion: "72725"
    uid: e18b13fb-a96d-436f-be75-a9a0f2b07993
  spec:
    namespace: pipelines
    serviceAccount:
      name: pipelines-installer
    source:
      catalog:
        packageName: openshift-pipelines-operator-rh
        upgradeConstraintPolicy: CatalogProvided
        version: 1.14.x
      sourceType: Catalog
  status:
    conditions:
    - lastTransitionTime: "2025-02-18T21:48:13Z"
      message: ""
      observedGeneration: 1
      reason: Deprecated
      status: "False"
      type: Deprecated
    - lastTransitionTime: "2025-02-18T21:48:13Z"
      message: ""
      observedGeneration: 1
      reason: Deprecated
      status: "False"
      type: PackageDeprecated
    - lastTransitionTime: "2025-02-18T21:48:13Z"
      message: ""
      observedGeneration: 1
      reason: Deprecated
      status: "False"
      type: ChannelDeprecated
    - lastTransitionTime: "2025-02-18T21:48:13Z"
      message: ""
      observedGeneration: 1
      reason: Deprecated
      status: "False"
      type: BundleDeprecated
    - lastTransitionTime: "2025-02-18T21:48:16Z"
      message: Installed bundle registry.redhat.io/openshift-pipelines/pipelines-operator-
bundle@sha256:f7b19ce26be742c4aaa458d37bc5ad373b5b29b20aaa7d308349687d3cbd
8838
        successfully
      observedGeneration: 1
      reason: Succeeded
      status: "True"
      type: Installed
    - lastTransitionTime: "2025-02-18T21:48:16Z"
      message: desired state reached
      observedGeneration: 1
      reason: Succeeded
      status: "True"
      type: Progressing
    install:
      bundle:
        name: openshift-pipelines-operator-rh.v1.14.5
        version: 1.14.5
kind: List
metadata:
  resourceVersion: ""
```

where:

spec.channel

Displays the channel defined in the CR of the extension.

spec.version

Displays the version or version range defined in the CR of the extension.

status.conditions

Displays information about the status and health of the extension.

type: Deprecated

Displays whether one or more of following are deprecated:

type: PackageDeprecated

Displays whether the resolved package is deprecated.

type: ChannelDeprecated

Displays whether the resolved channel is deprecated.

type: BundleDeprecated

Displays whether the resolved bundle is deprecated.

The value of **False** in the **status** field indicates that the **reason: Deprecated** condition is not deprecated. The value of **True** in the **status** field indicates that the **reason: Deprecated** condition is deprecated.

installedBundle.name

Displays the name of the bundle installed.

installedBundle.version

Displays the version of the bundle installed.

Additional resources

- Supported extensions
- Creating a service account
- Example custom resources (CRs) that specify a target version
- Support for version ranges

10.1.5. Updating a cluster extension

You can update your cluster extension or Operator by manually editing the custom resource (CR) and applying the changes.

Prerequisites

- You have an Operator or extension installed.
- You have installed the **jq** CLI tool.
- You have installed the **opm** CLI tool.

Procedure

1. Inspect a package for channel and version information from a local copy of your catalog file by completing the following steps:

   a. Get a list of channels from a selected package by running the following command:

   ```
   $ opm render <catalog_registry_url>:<tag> \
     | jq -s '.[] | select( .schema == "olm.channel" ) \
     | select( .package == "openshift-pipelines-operator-rh") | .name'
   ```

   Example 10.12. Example command

   ```
   $ opm render registry.redhat.io/redhat/redhat-operator-index:v4.18 \
     | jq -s '.[] | select( .schema == "olm.channel" ) \
     | select( .package == "openshift-pipelines-operator-rh") | .name'
   ```

   Example 10.13. Example output

   ```
   "latest"
   "pipelines-1.14"
   "pipelines-1.15"
   "pipelines-1.16"
   "pipelines-1.17"
   ```

   b. Get a list of the versions published in a channel by running the following command:

   ```
   $ opm render <catalog_registry_url>:<tag> \
     | jq -s '.[] | select( .package == "<package_name>" ) \
   ```

```
  | select( .schema == "olm.channel" ) \
  | select( .name == "<channel_name>" ) | .entries \
  | .[] | .name'
```

Example 10.14. Example command

```
$ opm render registry.redhat.io/redhat/redhat-operator-index:v4.18 \
  | jq -s '.[] | select( .package == "openshift-pipelines-operator-rh" ) \
  | select( .schema == "olm.channel" ) | select( .name == "latest" ) \
  | .entries | .[] | .name'
```

Example 10.15. Example output

```
"openshift-pipelines-operator-rh.v1.15.0"
"openshift-pipelines-operator-rh.v1.16.0"
"openshift-pipelines-operator-rh.v1.17.0"
"openshift-pipelines-operator-rh.v1.17.1"
```

2. Find out what version or channel is specified in your Operator or extension's CR by running the following command:

```
$ oc get clusterextension <operator_name> -o yaml
```

Example command

```
$ oc get clusterextension pipelines-operator -o yaml
```

Example 10.16. Example output

```
apiVersion: v1
items:
- apiVersion: olm.operatorframework.io/v1
  kind: ClusterExtension
  metadata:
    annotations:
      kubectl.kubernetes.io/last-applied-configuration: |
        {"apiVersion":"olm.operatorframework.io/v1","kind":"ClusterExtension","metadata":
{"annotations":{},"name":"pipes"},"spec":{"namespace":"pipelines","serviceAccount":
{"name":"pipelines-installer"},"source":{"catalog":{"packageName":"openshift-pipelines-
operator-rh","version":"1.14.x"},"sourceType":"Catalog"}}}
    creationTimestamp: "2025-02-18T21:48:13Z"
    finalizers:
    - olm.operatorframework.io/cleanup-unpack-cache
    - olm.operatorframework.io/cleanup-contentmanager-cache
    generation: 1
    name: pipelines-operator
    resourceVersion: "72725"
    uid: e18b13fb-a96d-436f-be75-a9a0f2b07993
  spec:
    namespace: pipelines
    serviceAccount:
      name: pipelines-installer
    source:
      catalog:
        packageName: openshift-pipelines-operator-rh
        upgradeConstraintPolicy: CatalogProvided
        version: 1.14.x
      sourceType: Catalog
  status:
    conditions:
    - lastTransitionTime: "2025-02-18T21:48:13Z"
      message: ""
      observedGeneration: 1
      reason: Deprecated
      status: "False"
      type: Deprecated
    - lastTransitionTime: "2025-02-18T21:48:13Z"
      message: ""
      observedGeneration: 1
      reason: Deprecated
      status: "False"
      type: PackageDeprecated
    - lastTransitionTime: "2025-02-18T21:48:13Z"
      message: ""
      observedGeneration: 1
      reason: Deprecated
      status: "False"
      type: ChannelDeprecated
    - lastTransitionTime: "2025-02-18T21:48:13Z"
      message: ""
      observedGeneration: 1
      reason: Deprecated
      status: "False"
      type: BundleDeprecated
    - lastTransitionTime: "2025-02-18T21:48:16Z"
      message: Installed bundle registry.redhat.io/openshift-pipelines/pipelines-operator-
bundle@sha256:f7b19ce26be742c4aaa458d37bc5ad373b5b29b20aaa7d308349687d3cbd
8838
        successfully
      observedGeneration: 1
      reason: Succeeded
      status: "True"
      type: Installed
    - lastTransitionTime: "2025-02-18T21:48:16Z"
      message: desired state reached
      observedGeneration: 1
      reason: Succeeded
      status: "True"
      type: Progressing
    install:
      bundle:
        name: openshift-pipelines-operator-rh.v1.14.5
        version: 1.14.5
kind: List
metadata:
  resourceVersion: ""
```

3. Edit your CR by using one of the following methods:

   - If you want to pin your Operator or extension to specific version, such as **1.15.0**, edit your CR similar to the following example:

     Example pipelines-operator.yaml CR

     ```
     apiVersion: olm.operatorframework.io/v1
     kind: ClusterExtension
     metadata:
       name: pipelines-operator
     spec:
       namespace: pipelines
       serviceAccount:
         name: pipelines-installer
       source:
         sourceType: Catalog
         catalog:
           packageName: openshift-pipelines-operator-rh
           version: "1.15.0" (1)
     ```

     (1) Update the version from **1.14.x** to **1.15.0**

   - If you want to define a range of acceptable update versions, edit your CR similar to the following example:

     Example CR with a version range specified

     ```
     apiVersion: olm.operatorframework.io/v1
     kind: ClusterExtension
     metadata:
       name: pipelines-operator
     spec:
       namespace: pipelines
       serviceAccount:
         name: pipelines-installer
       source:
         sourceType: Catalog
         catalog:
           packageName: openshift-pipelines-operator-rh
           version: ">1.15, <1.17" (1)
     ```

     (1) Specifies that the desired version range is greater than version **1.15** and less than **1.17**. For more information, see "Support for version ranges" and "Version comparison strings".

   - If you want to update to the latest version that can be resolved from a channel, edit your CR similar to the following example:

     Example CR with a specified channel

```
apiVersion: olm.operatorframework.io/v1
kind: ClusterExtension
metadata:
  name: pipelines-operator
spec:
  namespace: pipelines
  serviceAccount:
    name: pipelines-installer
  source:
    sourceType: Catalog
    catalog:
      packageName: openshift-pipelines-operator-rh
      channels:
        - latest (1)
```

(1) Installs the latest release that can be resolved from the specified channel. Updates to the channel are automatically installed. Enter values as an array.

- If you want to specify a channel and version or version range, edit your CR similar to the following example:

  Example CR with a specified channel and version range

  ```
  apiVersion: olm.operatorframework.io/v1
  kind: ClusterExtension
  metadata:
    name: pipelines-operator
  spec:
    namespace: pipelines
    serviceAccount:
      name: pipelines-installer
    source:
      sourceType: Catalog
      catalog:
        packageName: openshift-pipelines-operator-rh
        channels:
          - latest
        version: "<1.16"
  ```

  For more information, see "Example custom resources (CRs) that specify a target version".

4. Apply the update to the cluster by running the following command:

   ```
   $ oc apply -f pipelines-operator.yaml
   ```

   Example output

   ```
   clusterextension.olm.operatorframework.io/pipelines-operator configured
   ```

Verification

- Verify that the channel and version updates have been applied by running the following command:

```
$ oc get clusterextension pipelines-operator -o yaml
```

Example 10.17. Example output

```
apiVersion: olm.operatorframework.io/v1
kind: ClusterExtension
metadata:
  annotations:
    kubectl.kubernetes.io/last-applied-configuration: |
      {"apiVersion":"olm.operatorframework.io/v1","kind":"ClusterExtension","metadata":
{"annotations":{},"name":"pipes"},"spec":{"namespace":"pipelines","serviceAccount":
{"name":"pipelines-installer"},"source":{"catalog":{"packageName":"openshift-pipelines-
operator-rh","version":"\u003c1.16"},"sourceType":"Catalog"}}}
  creationTimestamp: "2025-02-18T21:48:13Z"
  finalizers:
  - olm.operatorframework.io/cleanup-unpack-cache
  - olm.operatorframework.io/cleanup-contentmanager-cache
  generation: 2
  name: pipes
  resourceVersion: "90693"
  uid: e18b13fb-a96d-436f-be75-a9a0f2b07993
spec:
  namespace: pipelines
  serviceAccount:
    name: pipelines-installer
  source:
    catalog:
      packageName: openshift-pipelines-operator-rh
      upgradeConstraintPolicy: CatalogProvided
      version: <1.16
    sourceType: Catalog
status:
  conditions:
  - lastTransitionTime: "2025-02-18T21:48:13Z"
    message: ""
    observedGeneration: 2
    reason: Deprecated
    status: "False"
    type: Deprecated
  - lastTransitionTime: "2025-02-18T21:48:13Z"
    message: ""
    observedGeneration: 2
    reason: Deprecated
    status: "False"
    type: PackageDeprecated
  - lastTransitionTime: "2025-02-18T21:48:13Z"
    message: ""
    observedGeneration: 2
    reason: Deprecated
    status: "False"
    type: ChannelDeprecated
  - lastTransitionTime: "2025-02-18T21:48:13Z"
    message: ""
    observedGeneration: 2
    reason: Deprecated
    status: "False"
    type: BundleDeprecated
  - lastTransitionTime: "2025-02-18T21:48:16Z"
    message: Installed bundle registry.redhat.io/openshift-pipelines/pipelines-operator-
bundle@sha256:8a593c1144709c9aeffbeb68d0b4b08368f528e7bb6f595884b2474bcfbcaf
cd
      successfully
    observedGeneration: 2
    reason: Succeeded
    status: "True"
    type: Installed
  - lastTransitionTime: "2025-02-18T21:48:16Z"
    message: desired state reached
    observedGeneration: 2
    reason: Succeeded
    status: "True"
    type: Progressing
  install:
    bundle:
      name: openshift-pipelines-operator-rh.v1.15.2
      version: 1.15.2
```

Troubleshooting

- If you specify a target version or channel that is deprecated or does not exist, you can run the following command to check the status of your extension:

```
$ oc get clusterextension <operator_name> -o yaml
```

Example 10.18. Example output for a version that does not exist

```
apiVersion: olm.operatorframework.io/v1
kind: ClusterExtension
metadata:
  annotations:
    kubectl.kubernetes.io/last-applied-configuration: |
      {"apiVersion":"olm.operatorframework.io/v1","kind":"ClusterExtension","metadata":
{"annotations":{},"name":"pipes"},"spec":{"namespace":"pipelines","serviceAccount":
{"name":"pipelines-installer"},"source":{"catalog":{"packageName":"openshift-pipelines-
operator-rh","version":"9.x"},"sourceType":"Catalog"}}}
  creationTimestamp: "2025-02-18T21:48:13Z"
  finalizers:
  - olm.operatorframework.io/cleanup-unpack-cache
  - olm.operatorframework.io/cleanup-contentmanager-cache
  generation: 3
  name: pipes
  resourceVersion: "93334"
  uid: e18b13fb-a96d-436f-be75-a9a0f2b07993
spec:
  namespace: pipelines
  serviceAccount:
    name: pipelines-installer
  source:
    catalog:
      packageName: openshift-pipelines-operator-rh
      upgradeConstraintPolicy: CatalogProvided
      version: 9.x
    sourceType: Catalog
status:
  conditions:
  - lastTransitionTime: "2025-02-18T21:48:13Z"
    message: ""
    observedGeneration: 2
    reason: Deprecated
    status: "False"
    type: Deprecated
  - lastTransitionTime: "2025-02-18T21:48:13Z"
    message: ""
    observedGeneration: 2
    reason: Deprecated
    status: "False"
    type: PackageDeprecated
  - lastTransitionTime: "2025-02-18T21:48:13Z"
    message: ""
    observedGeneration: 2
    reason: Deprecated
    status: "False"
    type: ChannelDeprecated
  - lastTransitionTime: "2025-02-18T21:48:13Z"
    message: ""
    observedGeneration: 2
    reason: Deprecated
    status: "False"
    type: BundleDeprecated
  - lastTransitionTime: "2025-02-18T21:48:16Z"
    message: Installed bundle registry.redhat.io/openshift-pipelines/pipelines-operator-
bundle@sha256:8a593c1144709c9aeffbeb68d0b4b08368f528e7bb6f595884b2474bcfbcafc
d
      successfully
    observedGeneration: 3
    reason: Succeeded
    status: "True"
    type: Installed
  - lastTransitionTime: "2025-02-18T21:48:16Z"
    message: 'error upgrading from currently installed version "1.15.2": no bundles
      found for package "openshift-pipelines-operator-rh" matching version "9.x"'
    observedGeneration: 3
    reason: Retrying
    status: "True"
    type: Progressing
  install:
    bundle:
      name: openshift-pipelines-operator-rh.v1.15.2
      version: 1.15.2
```

Additional resources

- Update paths

10.1.6. Deleting an Operator

You can delete an Operator and its custom resource definitions (CRDs) by deleting the **ClusterExtension** custom resource (CR).

Prerequisites

- You have a catalog installed.
- You have an Operator installed.

Procedure

- Delete an Operator and its CRDs by running the following command:

  ```
  $ oc delete clusterextension <operator_name>
  ```

  Example output

  ```
  clusterextension.olm.operatorframework.io "<operator_name>" deleted
  ```

Verification

- Run the following commands to verify that your Operator and its resources were deleted:
  - Verify the Operator is deleted by running the following command:

    ```
    $ oc get clusterextensions
    ```

    Example output

    ```
    No resources found
    ```

  - Verify that the Operator’s system namespace is deleted by running the following command:

    ```
    $ oc get ns <operator_name>-system
    ```

    Example output

    ```
    Error from server (NotFound): namespaces "<operator_name>-system" not found
    ```

10.2. USER ACCESS TO EXTENSION RESOURCES

After a cluster extension has been installed and is being managed by Operator Lifecycle Manager (OLM) v1, the extension can often provide **CustomResourceDefinition** objects (CRDs) that expose new API resources on the cluster. Cluster administrators typically have full management access to these resources by default, whereas non-cluster administrator users, or *regular users*, might lack sufficient permissions.

OLM v1 does not automatically configure or manage role-based access control (RBAC) for regular users to interact with the APIs provided by installed extensions. Cluster administrators must define the required RBAC policy to create, view, or edit these custom resources (CRs) for such users.

NOTE

The RBAC permissions described for user access to extension resources are different from the permissions that must be added to a service account to enable OLM v1-based initial installation of a cluster extension itself. For more on RBAC requirements while installing an extension, see "Cluster extension permissions" in "Managing extensions".

Additional resources

- "Managing extensions" → "Cluster extension permissions"

10.2.1. Common default cluster roles for users

An installed cluster extension might include default cluster roles to determine role-based access control (RBAC) for regular users to API resources provided by the extension. A common set of cluster roles can resemble the following policies:

view cluster role

Grants read-only access to all custom resource (CR) objects of specified API resources across the cluster. Intended for regular users who require visibility into the resources without any permissions to modify them. Ideal for monitoring purposes and limited access viewing.

edit cluster role

Allows users to modify all CR objects within the cluster. Enables users to create, update, and delete resources, making it suitable for team members who must manage resources but should not control RBAC or manage permissions for others.

admin cluster role

Provides full permissions, including **create**, **update**, and **delete** verbs, over all custom resource objects for the specified API resources across the cluster.

Additional resources

- User-facing roles (Kubernetes documentation)

10.2.2. Finding API groups and resources exposed by a cluster extension

To create appropriate RBAC policies for granting user access to cluster extension resources, you must know which API groups and resources are exposed by the installed extension. As an administrator, you can inspect custom resource definitions (CRDs) installed on the cluster by using OpenShift CLI (**oc**).

Prerequisites

- A cluster extension has been installed on your cluster.

Procedure

- To list installed CRDs while specifying a label selector targeting a specific cluster extension by name to find only CRDs owned by that extension, run the following command:

```
$ oc get crds -l 'olm.operatorframework.io/owner-
kind=ClusterExtension,olm.operatorframework.io/owner-name=<cluster_extension_name>'
```

- Alternatively, you can search through all installed CRDs and individually inspect them by CRD name:

    a. List all available custom resource definitions (CRDs) currently installed on the cluster by running the following command:

    ```
    $ oc get crds
    ```

    Find the CRD you are looking for in the output.

    b. Inspect the individual CRD further to find its API groups by running the following command:

    ```
    $ oc get crd <crd_name> -o yaml
    ```

10.2.3. Granting user access to extension resources by using custom role bindings

As a cluster administrator, you can manually create and configure role-based access control (RBAC) policies to grant user access to extension resources by using custom role bindings.

Prerequisites

- A cluster extension has been installed on your cluster.
- You have a list of API groups and resource names, as described in "Finding API groups and resources exposed by a cluster extension".

Procedure

1. If the installed cluster extension does not provide default cluster roles, manually create one or more roles:

    a. Consider the use cases for the set of roles described in "Common default cluster roles for users".
    For example, create one or more of the following **ClusterRole** object definitions, replacing **<cluster_extension_api_group>** and **<cluster_extension_custom_resource>** with the actual API group and resource names provided by the installed cluster extension:

    Example view-custom-resource.yaml file

    ```
    apiVersion: rbac.authorization.k8s.io/v1
    kind: ClusterRole
    metadata:
      name: view-custom-resource
    rules:
    - apiGroups:
      - <cluster_extension_api_group>
      resources:
      - <cluster_extension_custom_resources>
      verbs:
      - get
      - list
      - watch
    ```

    Example edit-custom-resource.yaml file

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: edit-custom-resource
rules:
- apiGroups:
  - <cluster_extension_api_group>
  resources:
  - <cluster_extension_custom_resources>
  verbs:
  - get
  - list
  - watch
  - create
  - update
  - patch
  - delete
```

Example admin-custom-resource.yaml file

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: admin-custom-resource
rules:
- apiGroups:
  - <cluster_extension_api_group>
  resources:
  - <cluster_extension_custom_resources>
  verbs:
  - '*' 1
```

1 Setting a wildcard (*) in **verbs** allows all actions on the specified resources.

b. Create the cluster roles by running the following command for any YAML files you created:

```
$ oc create -f <filename>.yaml
```

2. Associate a cluster role to specific users or groups to grant them the necessary permissions for the resource by binding the cluster roles to individual user or group names:

   a. Create an object definition for either a *cluster role binding* to grant access across all namespaces or a *role binding* to grant access within a specific namespace:

      - The following example cluster role bindings grant read-only **view** access to the custom resource across all namespaces:

        Example ClusterRoleBinding object for a user

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: view-custom-resource-binding
subjects:
```

```
- kind: User
  name: <user_name>
roleRef:
  kind: ClusterRole
  name: view-custom-resource
  apiGroup: rbac.authorization.k8s.io
```

Example ClusterRoleBinding object for a user

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: view-custom-resource-binding
subjects:
- kind: Group
  name: <group_name>
roleRef:
  kind: ClusterRole
  name: view-custom-resource
  apiGroup: rbac.authorization.k8s.io
```

- The following role binding restricts **edit** permissions to a specific namespace:

Example RoleBinding object for a user

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: edit-custom-resource-edit-binding
  namespace: <namespace>
subjects:
- kind: User
  name: <username>
roleRef:
  kind: Role
  name: custom-resource-edit
  apiGroup: rbac.authorization.k8s.io
```

b. Save your object definition to a YAML file.

c. Create the object by running the following command:

```
$ oc create -f <filename>.yaml
```

10.2.4. Granting user access to extension resources by using aggregated cluster roles

As a cluster administrator, you can configure role-based access control (RBAC) policies to grant user access to extension resources by using aggregated cluster roles.

To automatically extend existing default cluster roles, you can add *aggregation labels* by adding one or more of the following labels to a **ClusterRole** object:

Aggregation labels in a ClusterRole object

```
# ..
metadata:
  labels:
    rbac.authorization.k8s.io/aggregate-to-admin: "true"
    rbac.authorization.k8s.io/aggregate-to-edit: "true"
    rbac.authorization.k8s.io/aggregate-to-view: "true"
# ..
```

This allows users who already have **view**, **edit**, or **admin** roles to interact with the custom resource specified by the **ClusterRole** object without requiring additional role or cluster role bindings to specific users or groups.

Prerequisites

- A cluster extension has been installed on your cluster.
- You have a list of API groups and resource names, as described in "Finding API groups and resources exposed by a cluster extension".

Procedure

1. Create an object definition for a cluster role that specifies the API groups and resources provided by the cluster extension and add an aggregation label to extend one or more existing default cluster roles:

   Example ClusterRole object with an aggregation label

   ```
   apiVersion: rbac.authorization.k8s.io/v1
   kind: ClusterRole
   metadata:
     name: view-custom-resource-aggregated
     labels:
       rbac.authorization.k8s.io/aggregate-to-view: "true"
   rules:
     - apiGroups:
         - <cluster_extension_api_group>
       resources:
         - <cluster_extension_custom_resource>
       verbs:
         - get
         - list
         - watch
   ```

   You can create similar **ClusterRole** objects for **edit** and **admin** with appropriate verbs, such as **create**, **update**, and **delete**. By using aggregation labels, the permissions for the custom resources are added to the default roles.

2. Save your object definition to a YAML file.

3. Create the object by running the following command:

   ```
   $ oc create -f <filename>.yaml
   ```

Additional resources

- Aggregated ClusterRoles (Kubernetes documentation)

10.3. UPDATE PATHS

When determining *update paths*, also known as upgrade edges or upgrade constraints, for an installed cluster extension, Operator Lifecycle Manager (OLM) v1 supports OLM (Classic) semantics starting in OpenShift Container Platform 4.16. This support follows the behavior from OLM (Classic), including **replaces**, **skips**, and **skipRange** directives, with a few noted differences.

By supporting OLM (Classic) semantics, OLM v1 accurately reflects the update graph from catalogs.

Differences from original OLM (Classic) implementation

- If there are multiple possible successors, OLM v1 behavior differs in the following ways:
  - In OLM (Classic), the successor closest to the channel head is chosen.
  - In OLM v1, the successor with the highest semantic version (semver) is chosen.
- Consider the following set of file-based catalog (FBC) channel entries:

  ```
  # ...
  - name: example.v3.0.0
    skips: ["example.v2.0.0"]
  - name: example.v2.0.0
    skipRange: >=1.0.0 <2.0.0
  ```

  If **1.0.0** is installed, OLM v1 behavior differs in the following ways:

  - OLM (Classic) will not detect an update path to **v2.0.0** because **v2.0.0** is skipped and not on the **replaces** chain.
  - OLM v1 will detect the update path because OLM v1 does not have a concept of a **replaces** chain. OLM v1 finds all entries that have a **replace**, **skip**, or **skipRange** value that covers the currently installed version.

Additional resources

- OLM (Classic) upgrade semantics

10.3.1. Support for version ranges

In Operator Lifecycle Manager (OLM) v1, you can specify a version range by using a comparison string in an Operator or extension's custom resource (CR). If you specify a version range in the CR, OLM v1 installs or updates to the latest version of the Operator that can be resolved within the version range.

Resolved version workflow

- The resolved version is the latest version of the Operator that satisfies the constraints of the Operator and the environment.
- An Operator update within the specified range is automatically installed if it is resolved successfully.

- An update is not installed if it is outside of the specified range or if it cannot be resolved successfully.

10.3.2. Version comparison strings

You can define a version range by adding a comparison string to the **spec.version** field in an Operator or extension's custom resource (CR). A comparison string is a list of space- or comma-separated values and one or more comparison operators enclosed in double quotation marks (**"**). You can add another comparison string by including an **OR**, or double vertical bar (**||**), comparison operator between the strings.

Table 10.4. Basic comparisons

Comparison operator	Definition
=	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

You can specify a version range in an Operator or extension's CR by using a range comparison similar to the following example:

Example version range comparison

```
apiVersion: olm.operatorframework.io/v1
  kind: ClusterExtension
  metadata:
    name: <clusterextension_name>
  spec:
    namespace: <installed_namespace>
    serviceAccount:
      name: <service_account_installer_name>
    source:
      sourceType: Catalog
      catalog:
        packageName: <package_name>
        version: ">=1.11, <1.13"
```

You can use wildcard characters in all types of comparison strings. OLM v1 accepts **x**, **X**, and asterisks (*) as wildcard characters. When you use a wildcard character with the equal sign (**=**) comparison operator, you define a comparison at the patch or minor version level.

Table 10.5. Example wildcard characters in comparison strings

Wildcard comparison	Matching string
1.11.x	**>=1.11.0, <1.12.0**
>=1.12.X	**>=1.12.0**
<=2.x	**<3**
*****	**>=0.0.0**

You can make patch release comparisons by using the tilde (**~**) comparison operator. Patch release comparisons specify a minor version up to the next major version.

Table 10.6. Example patch release comparisons

Patch release comparison	Matching string
~1.11.0	**>=1.11.0, <1.12.0**
~1	**>=1, <2**
~1.12	**>=1.12, <1.13**
~1.12.x	**>=1.12.0, <1.13.0**
~1.x	**>=1, <2**

You can use the caret (**^**) comparison operator to make a comparison for a major release. If you make a major release comparison before the first stable release is published, the minor versions define the API's level of stability. In the semantic versioning (semver) specification, the first stable release is published as the **1.0.0** version.

Table 10.7. Example major release comparisons

Major release comparison	Matching string
^0	**>=0.0.0, <1.0.0**
^0.0	**>=0.0.0, <0.1.0**
^0.0.3	**>=0.0.3, <0.0.4**
^0.2	**>=0.2.0, <0.3.0**
^0.2.3	**>=0.2.3, <0.3.0**
^1.2.x	**>= 1.2.0, < 2.0.0**

Major release comparison	Matching string
^1.2.3	**>= 1.2.3, < 2.0.0**
^2.x	**>= 2.0.0, < 3**
^2.3	**>= 2.3, < 3**

10.3.3. Example custom resources (CRs) that specify a target version

In Operator Lifecycle Manager (OLM) v1, cluster administrators can declaratively set the target version of an Operator or extension in the custom resource (CR).

You can define a target version by specifying any of the following fields:

- Channel
- Version number
- Version range

If you specify a channel in the CR, OLM v1 installs the latest version of the Operator or extension that can be resolved within the specified channel. When updates are published to the specified channel, OLM v1 automatically updates to the latest release that can be resolved from the channel.

Example CR with a specified channel

```
apiVersion: olm.operatorframework.io/v1
  kind: ClusterExtension
  metadata:
    name: <clusterextension_name>
  spec:
    namespace: <installed_namespace>
    serviceAccount:
      name: <service_account_installer_name>
    source:
      sourceType: Catalog
      catalog:
        packageName: <package_name>
        channels:
          - latest (1)
```

(1) Optional: Installs the latest release that can be resolved from the specified channel. Updates to the channel are automatically installed. Specify the value of the **channels** parameter as an array.

If you specify the Operator or extension’s target version in the CR, OLM v1 installs the specified version. When the target version is specified in the CR, OLM v1 does not change the target version when updates are published to the catalog.

If you want to update the version of the Operator that is installed on the cluster, you must manually edit the Operator’s CR. Specifying an Operator’s target version pins the Operator’s version to the specified release.

Example CR with the target version specified

```
apiVersion: olm.operatorframework.io/v1
 kind: ClusterExtension
 metadata:
   name: <clusterextension_name>
 spec:
   namespace: <installed_namespace>
   serviceAccount:
     name: <service_account_installer_name>
   source:
     sourceType: Catalog
     catalog:
       packageName: <package_name>
       version: "1.11.1" 1
```

1 Optional: Specifies the target version. If you want to update the version of the Operator or extension that is installed, you must manually update this field the CR to the desired target version.

If you want to define a range of acceptable versions for an Operator or extension, you can specify a version range by using a comparison string. When you specify a version range, OLM v1 installs the latest version of an Operator or extension that can be resolved by the Operator Controller.

Example CR with a version range specified

```
apiVersion: olm.operatorframework.io/v1
 kind: ClusterExtension
 metadata:
   name: <clusterextension_name>
 spec:
   namespace: <installed_namespace>
   serviceAccount:
     name: <service_account_installer_name>
   source:
     sourceType: Catalog
     catalog:
       packageName: <package_name>
       version: ">1.11.1" 1
```

1 Optional: Specifies that the desired version range is greater than version **1.11.1**. For more information, see "Support for version ranges".

After you create or update a CR, apply the configuration file by running the following command:

Command syntax

```
$ oc apply -f <extension_name>.yaml
```

10.3.4. Forcing an update or rollback

OLM v1 does not support automatic updates to the next major version or rollbacks to an earlier version. If you want to perform a major version update or rollback, you must verify and force the update manually.

WARNING

You must verify the consequences of forcing a manual update or rollback. Failure to verify a forced update or rollback might have catastrophic consequences such as data loss.

Prerequisites

- You have a catalog installed.
- You have an Operator or extension installed.
- You have created a service account and assigned enough role-based access controls (RBAC) to install, update, and manage the extension you want to install. For more information, see *Creating a service account* .

Procedure

1. Edit the custom resource (CR) of your Operator or extension as shown in the following example:

   Example CR

   ```
   apiVersion: olm.operatorframework.io/v1
    kind: ClusterExtension
    metadata:
     name: <clusterextension_name>
    spec:
     namespace: <installed_namespace> (1)
     serviceAccount:
      name: <service_account_installer_name> (2)
     source:
      sourceType: Catalog
      catalog:
       packageName: <package_name>
       channels:
        - <channel_name> (3)
       version: <version_or_version_range> (4)
       upgradeConstraintPolicy: SelfCertified (5)
   ```

   (1) Specifies the namespace where you want the bundle installed, such as **pipelines** or **my-extension**. Extensions are still cluster-scoped and might contain resources that are installed in different namespaces.

   (2) Specifies the name of the service account you created to install, update, and manage your extension.

   (3) Optional: Specifies channel names as an array, such as **pipelines-1.14** or **latest**.

   (4) Optional: Specifies the version or version range, such as **1.14.0**, **1.14.x**, or **>=1.16**, of the package you want to install or update. For more information, see "Example custom resources (CRs) that specify a target version" and "Support for version ranges".

5 Optional: Specifies the upgrade constraint policy. To force an update or rollback, set the field to **SelfCertified**. If unspecified, the default setting is **CatalogProvided**. The **CatalogProvided** setting only updates if the new version satisfies the upgrade constraints set by the package author.

2. Apply the changes to your Operator or extensions CR by running the following command:

```
$ oc apply -f <extension_name>.yaml
```

Additional resources

- Support for version ranges

10.3.5. Compatibility with OpenShift Container Platform versions

Before cluster administrators can update their OpenShift Container Platform cluster to its next minor version, they must ensure that all installed Operators are updated to a bundle version that is compatible with the cluster's next minor version (4.y+1).

For example, Kubernetes periodically deprecates certain APIs that are removed in subsequent releases. If an extension is using a deprecated API, it might no longer work after the OpenShift Container Platform cluster is updated to the Kubernetes version where the API has been removed.

If an Operator author knows that a specific bundle version is not supported and will not work correctly, for any reason, on OpenShift Container Platform later than a certain cluster minor version, they can configure the maximum version of OpenShift Container Platform that their Operator is compatible with.

In the Operator project's cluster service version (CSV), authors can set the **olm.maxOpenShiftVersion** annotation to prevent administrators from updating the cluster before updating the installed Operator to a compatible version.

Example CSV with **olm.maxOpenShiftVersion** annotation

```
apiVersion: operators.coreos.com/v1alpha1
kind: ClusterServiceVersion
metadata:
  annotations:
    "olm.properties": '[{"type": "olm.maxOpenShiftVersion", "value": "<cluster_version>"}]' 1
```

1 Specifies the latest minor version of OpenShift Container Platform (4.y) that an Operator is compatible with. For example, setting **value** to **4.18** prevents cluster updates to minor versions later than 4.18 when this bundle is installed on a cluster.

If the **olm.maxOpenShiftVersion** field is omitted, cluster updates are not blocked by this Operator.

NOTE

When determining a cluster's next minor version (4.y+1), OLM v1 only considers major and minor versions (x and y) for comparisons. It ignores any *z-stream* versions (4.y.z), also known as patch releases, or pre-release versions.

For example, if the cluster's current version is **4.18.0**, the next minor version is **4.19**. If the current version is **4.18.0-rc1**, the next minor version is still **4.19**.

Additional resources

- Deprecated API Migration Guide (Kubernetes documentation)

10.3.5.1. Cluster updates blocked by olm cluster Operator

If an installed Operator's **olm.maxOpenShiftVersion** field is set and a cluster administrator attempts to update their cluster to a version that the Operator does not provide a valid update path for, the cluster update fails and the **Upgradeable** status for the **olm** cluster Operator is set to **False**.

To resolve the issue, the cluster administrator must either update the installed Operator to a version with a valid update path, if one is available, or they must uninstall the Operator. Then, they can attempt the cluster update again.

Additional resources

- Understanding cluster Operator condition types
- Upgrading installed Operators
- Deleting Operators from a cluster
- Cluster Operators reference → Operator Lifecycle Manager (OLM) v1 Operator

10.4. CUSTOM RESOURCE DEFINITION (CRD) UPGRADE SAFETY

When you update a custom resource definition (CRD) that is provided by a cluster extension, Operator Lifecycle Manager (OLM) v1 runs a CRD upgrade safety preflight check to ensure backwards compatibility with previous versions of that CRD. The CRD update must pass the validation checks before the change is allowed to progress on a cluster.

Additional resources

- Updating a cluster extension

10.4.1. Prohibited CRD upgrade changes

The following changes to an existing custom resource definition (CRD) are caught by the CRD upgrade safety preflight check and prevent the upgrade:

- A new required field is added to an existing version of the CRD
- An existing field is removed from an existing version of the CRD
- An existing field type is changed in an existing version of the CRD

- A new default value is added to a field that did not previously have a default value
- The default value of a field is changed
- An existing default value of a field is removed
- New enum restrictions are added to an existing field which did not previously have enum restrictions
- Existing enum values from an existing field are removed
- The minimum value of an existing field is increased in an existing version
- The maximum value of an existing field is decreased in an existing version
- Minimum or maximum field constraints are added to a field that did not previously have constraints

NOTE

The rules for changes to minimum and maximum values apply to **minimum**, **minLength**, **minProperties**, **minItems**, **maximum**, **maxLength**, **maxProperties**, and **maxItems** constraints.

The following changes to an existing CRD are reported by the CRD upgrade safety preflight check and prevent the upgrade, though the operations are technically handled by the Kubernetes API server:

- The scope changes from **Cluster** to **Namespace** or from **Namespace** to **Cluster**
- An existing stored version of the CRD is removed

If the CRD upgrade safety preflight check encounters one of the prohibited upgrade changes, it logs an error for each prohibited change detected in the CRD upgrade.

TIP

In cases where a change to the CRD does not fall into one of the prohibited change categories, but is also unable to be properly detected as allowed, the CRD upgrade safety preflight check will prevent the upgrade and log an error for an "unknown change".

10.4.2. Allowed CRD upgrade changes

The following changes to an existing custom resource definition (CRD) are safe for backwards compatibility and will not cause the CRD upgrade safety preflight check to halt the upgrade:

- Adding new enum values to the list of allowed enum values in a field
- An existing required field is changed to optional in an existing version
- The minimum value of an existing field is decreased in an existing version
- The maximum value of an existing field is increased in an existing version
- A new version of the CRD is added with no modifications to existing versions

10.4.3. Disabling CRD upgrade safety preflight check

The custom resource definition (CRD) upgrade safety preflight check can be disabled by adding the **preflight.crdUpgradeSafety.disabled** field with a value of **true** to the **ClusterExtension** object that provides the CRD.

WARNING

Disabling the CRD upgrade safety preflight check could break backwards compatibility with stored versions of the CRD and cause other unintended consequences on the cluster.

You cannot disable individual field validators. If you disable the CRD upgrade safety preflight check, all field validators are disabled.

NOTE

The following checks are handled by the Kubernetes API server:

- The scope changes from **Cluster** to **Namespace** or from **Namespace** to **Cluster**
- An existing stored version of the CRD is removed

After disabling the CRD upgrade safety preflight check via Operator Lifecycle Manager (OLM) v1, these two operations are still prevented by Kubernetes.

Prerequisites

- You have a cluster extension installed.

Procedure

1. Edit the **ClusterExtension** object of the CRD:

```
$ oc edit clusterextension <clusterextension_name>
```

2. Set the **preflight.crdUpgradeSafety.disabled** field to **true**:

Example 10.19. Example ClusterExtension object

```
apiVersion: olm.operatorframework.io/v1alpha1
kind: ClusterExtension
metadata:
  name: clusterextension-sample
spec:
  installNamespace: default
  packageName: argocd-operator
  version: 0.6.0
```

```
preflight:
  crdUpgradeSafety:
    disabled: true (1)
```

(1) Set to **true**.

10.4.4. Examples of unsafe CRD changes

The following examples demonstrate specific changes to sections of an example custom resource definition (CRD) that would be caught by the CRD upgrade safety preflight check.

For the following examples, consider a CRD object in the following starting state:

Example 10.20. Example CRD object

```
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  annotations:
    controller-gen.kubebuilder.io/version: v0.13.0
  name: example.test.example.com
spec:
  group: test.example.com
  names:
    kind: Sample
    listKind: SampleList
    plural: samples
    singular: sample
  scope: Namespaced
  versions:
  - name: v1alpha1
    schema:
      openAPIV3Schema:
        properties:
          apiVersion:
            type: string
          kind:
            type: string
          metadata:
            type: object
          spec:
            type: object
          status:
            type: object
          pollInterval:
            type: string
        type: object
    served: true
    storage: true
    subresources:
      status: {}
```

10.4.4.1. Scope change

In the following custom resource definition (CRD) example, the **scope** field is changed from **Namespaced** to **Cluster**:

Example 10.21. Example scope change in a CRD

```
    spec:
      group: test.example.com
      names:
        kind: Sample
        listKind: SampleList
        plural: samples
        singular: sample
      scope: Cluster
      versions:
      - name: v1alpha1
```

Example 10.22. Example error output

```
validating upgrade for CRD "test.example.com" failed: CustomResourceDefinition
test.example.com failed upgrade safety validation. "NoScopeChange" validation failed: scope
changed from "Namespaced" to "Cluster"
```

10.4.4.2. Removal of a stored version

In the following custom resource definition (CRD) example, the existing stored version, **v1alpha1**, is removed:

Example 10.23. Example removal of a stored version in a CRD

```
      versions:
      - name: v1alpha2
        schema:
          openAPIV3Schema:
            properties:
              apiVersion:
                type: string
              kind:
                type: string
              metadata:
                type: object
              spec:
                type: object
              status:
                type: object
              pollInterval:
                type: string
            type: object
```

Example 10.24. Example error output

```
validating upgrade for CRD "test.example.com" failed: CustomResourceDefinition
test.example.com failed upgrade safety validation. "NoStoredVersionRemoved" validation failed:
stored version "v1alpha1" removed
```

10.4.4.3. Removal of an existing field

In the following custom resource definition (CRD) example, the **pollInterval** property field is removed from the **v1alpha1** schema:

Example 10.25. Example removal of an existing field in a CRD

```
  versions:
  - name: v1alpha1
    schema:
      openAPIV3Schema:
        properties:
          apiVersion:
            type: string
          kind:
            type: string
          metadata:
            type: object
          spec:
            type: object
          status:
            type: object
        type: object
```

Example 10.26. Example error output

```
validating upgrade for CRD "test.example.com" failed: CustomResourceDefinition
test.example.com failed upgrade safety validation. "NoExistingFieldRemoved" validation failed:
crd/test.example.com version/v1alpha1 field/^.spec.pollInterval may not be removed
```

10.4.4.4. Addition of a required field

In the following custom resource definition (CRD) example, the **pollInterval** property has been changed to a required field:

Example 10.27. Example addition of a required field in a CRD

```
  versions:
  - name: v1alpha2
    schema:
      openAPIV3Schema:
        properties:
          apiVersion:
```

```
          type: string
        kind:
          type: string
        metadata:
          type: object
        spec:
          type: object
        status:
          type: object
        pollInterval:
          type: string
      type: object
      required:
      - pollInterval
```

Example 10.28. Example error output

```
validating upgrade for CRD "test.example.com" failed: CustomResourceDefinition
test.example.com failed upgrade safety validation. "ChangeValidator" validation failed: version
"v1alpha1", field "^": new required fields added: [pollInterval]
```