Extend

Overview | Operator | Cluster Plu

Chart Repository | Download Packages | Upload Pac

Overview

The platform provides a comprehensive extension system that allows users to enhance the functionality of their Kubernetes clusters. This system is designed to be flexible and user-friendly, enabling users to easily add new features and capabilities to their clusters.

This system consists of two main extension types:

- Operators: Operators are built on the Operator Lifecycle Manager (OLM) v0 framework, providing specialized operational capabilities for the platform. These extensions enable automated management of complex applications and services within your cluster.
- Cluster Plugins: The platform features a proprietary cluster plugin system specifically designed for Chart-type plugins. This system delivers an improved installation and management experience compared to standard methods, with a user-friendly interface for handling Chart-based extensions.

With support for numerous Operators and cluster plugins, users can significantly expand the platform's capabilities to meet specific operational requirements and use cases.

Operator

TOC

- Overview
- Operator Sources
- Pre-installation Preparation
  - Installation Mode
  - Update Channel
  - Approval Strategy
  - Installation Location
- Installing via Web Console
- Installing via YAML
  - Manual
    1. Check available versions
    2. Confirm catalogSource
    3. Create a namespace
    4. Create a Subscription
    5. Check Subscription status
    6. Approve InstallPlan
  - Automatic
    1. Check available versions
    2. Confirm catalogSource
    3. Create a namespace

4. Create a Subscription

5. Check Subscription status

6. Verify CSV

Advanced Subscription Configuration

Configure Operator Pod Scheduling

Upgrade Process

Overview

Based on the **OLM (Operator Lifecycle Manager)** framework, **OperatorHub** provides a unified interface for managing the installation, upgrade, and lifecycle of Operators. Administrators can use OperatorHub to install and manage Operators, enabling full lifecycle automation for Kubernetes applications, including creation, updates, and deletion.

OLM mainly consists of the following components and CRDs:

- **OLM (olm-operator)**: Manages the complete lifecycle of Operators, including installation, upgrades, and version conflict detection.
- **Catalog Operator**: Manages Operator catalogs and generates corresponding InstallPlans.
- **CatalogSource**: A namespace-scoped CRD that manages the Operator catalog source and provides Operator metadata (e.g., version info, managed CRDs). The platform provides 3 default CatalogSources: **system**, **platform**, and **custom**. Operators in **system** are not displayed in OperatorHub.
- **ClusterServiceVersion (CSV)**: A namespace-scoped CRD that describes a specific version of an Operator, including the resources, CRDs, and permissions it requires.
- **Subscription**: A namespace-scoped CRD that describes the subscribed Operator, its source, acquisition channel, and upgrade strategy.
- **InstallPlan**: A namespace-scoped CRD that describes the actual installation operations to be performed (e.g., creating Deployments, CRDs, RBAC). An Operator will only be installed or upgraded once the InstallPlan is approved.

Operator Sources

To clarify the lifecycle strategy of different Operators in OperatorHub, the platform provides 5 source types:

1. **Alauda** Provided and maintained by Alauda, including full lifecycle management, security updates, technical support, and SLA commitments.
2. **Curated** Selected from the open-source community, consistent with community versions, without code modifications or recompilation. Alauda provides guidance and security updates but does not guarantee SLA or lifecycle management.
3. **Community** Provided by the open-source community, updated periodically to ensure installability, but functional completeness is not guaranteed; no SLA or Alauda support is provided.
4. **Marketplace** Provided and maintained by third-party vendors certified by Alauda. Alauda provides platform integration support, while the vendor is responsible for core maintenance.
5. **Custom** Developed and uploaded by the user to meet custom use-case requirements.

Pre-installation Preparation

Before installing an Operator, you need to understand the following key parameters:

Installation Mode

OLM provides three installation modes:

- **Single Namespace**
- **Multi Namespace**
- **Cluster**

Cluster mode (AllNamespaces) is recommended. The platform will eventually be upgraded to OLM v1, which only supports the AllNamespaces install mode. Therefore, SingleNamespace and MultiNamespace should be strongly avoided.

Update Channel

If an Operator provides multiple update channels, you can choose which channel to subscribe to, e.g., **stable**.

Approval Strategy

Options: **Automatic** or **Manual**.

- **Automatic**: OLM will automatically upgrade the Operator when a new version is released in the selected channel.
- **Manual**: When a new version is available, OLM creates an upgrade request that must be manually approved by the cluster administrator before the upgrade occurs.

Note: Operators from Alauda only support **Manual** mode; otherwise, installation will fail.

Installation Location

It is recommended to create a separate namespace for each Operator.

If multiple Operators share the same namespace, their Subscriptions may be resolved into a single InstallPlan:

- If an InstallPlan in that namespace requires Manual approval and remains pending, it can block automatic upgrades for other Subscriptions included in the same InstallPlan.

Installing via Web Console

1. Log in to the web console and switch to the **Administrator** view.
2. Navigate to **Marketplace > OperatorHub**.
3. If the status is **Absent**:
   - Download the Operator package from the Alauda Customer Portal or contact support.
   - Upload the package to the target cluster using `violet` (see CLI).

- On the **Marketplace > Upload Packages** page, switch to the **Operator** tab and confirm the upload.

4. If the status is **Ready**, click **Install** and follow the Operator's user guide.

Installing via YAML

The following examples demonstrate installation methods for Operators from Alauda (Manual only) and non-Alauda sources (Manual or Automatic).

> **INFO**
>
> Unlike cluster plugins (which must always be installed in the **global cluster** when using YAML), Operators are installed in the **target cluster** where you want them to run. Make sure you are connected to the intended cluster before executing any YAML manifests.

Manual

The `harbor-ce-operator` is from Alauda and supports **Manual** approval only. In Manual mode, even if a new version is released, the Operator will not upgrade automatically. You must **Approve** manually before OLM executes the upgrade.

1. Check available versions

```
(
  echo -e "CHANNEL\tNAME\tVERSION"
  kubectl get packagemanifest harbor-ce-operator -o json | jq -r '
    .status.channels[] |
    .name as $channel |
    .entries[] |
    [$channel, .name, .version] | @tsv
  '
) | column -t -s $'\t'
```

Example output:

```
CHANNEL   NAME                          VERSION
harbor-2  harbor-ce-operator.v2.12.11  2.12.11
harbor-2  harbor-ce-operator.v2.12.10  2.12.10
stable    harbor-ce-operator.v2.12.11  2.12.11
stable    harbor-ce-operator.v2.12.10  2.12.10
```

Fields:

- **CHANNEL**: Operator channel name
- **NAME**: CSV resource name
- **VERSION**: Operator version

2. Confirm catalogSource

```
kubectl get packagemanifests harbor-ce-operator -ojsonpath='{.status.catalogSource}'
```

Example output:

```
platform
```

This indicates the `harbor-ce-operator` comes from the `platform` catalogSource.

3. Create a namespace

```
kubectl create namespace harbor-ce-operator
```

4. Create a Subscription

```yaml
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  annotations:
    cpaas.io/target-namespaces: ""
  name: harbor-ce-operator-subs
  namespace: harbor-ce-operator
spec:
  channel: stable
  installPlanApproval: Manual
  name: harbor-ce-operator
  source: platform
  sourceNamespace: cpaas-system
  startingCSV: harbor-ce-operator.v2.12.11
```

Field explanations:

- **annotation** `cpaas.io/target-namespaces` : It is recommended to set this to empty; empty indicates cluster-wide installation.
- **.metadata.name**: Subscription name (DNS-compliant, max 253 characters).
- **.metadata.namespace**: Namespace where the Operator will be installed.
- **.spec.channel**: Subscribed Operator channel.
- **.spec.installPlanApproval**: Approval strategy (`Manual` or `Automatic`). Here, `Manual` requires manual approval for install/upgrade.
- **.spec.source**: Operator catalogSource.
- **.spec.sourceNamespace**: Must be set to cpaas-system because all catalogSources provided by the platform are located in this namespace.
- **.spec.startingCSV**: Specifies the version to install for Manual approval; defaults to the latest in the channel if empty. Not required for Automatic.

5. Check Subscription status

```bash
kubectl -n harbor-ce-operator get subscriptions harbor-ce-operator-subs -o yaml
```

Key output:

- **.status.state**: `UpgradePending` indicates the Operator is awaiting installation or upgrade.
- **Condition InstallPlanPending = True**: Waiting for manual approval.
- **.status.currentCSV**: Latest subscribed CSV.
- **.status.installPlanRef**: Associated InstallPlan; must be approved before installation proceeds.

6. Approve InstallPlan

```
kubectl -n harbor-ce-operator get installplan \
  "$(kubectl -n harbor-ce-operator get subscriptions harbor-ce-operator-subs -o jsonpath='{.status.installPlanRef.name}')"
```

Example output:

```
NAME            CSV                              APPROVAL   APPROVED
install-27t29   harbor-ce-operator.v2.12.11   Manual     false
```

Approve manually:

```
PLAN="$(kubectl -n harbor-ce-operator get subscription harbor-ce-operator-subs -o jsonpath='{.status.installPlanRef.name}')"
kubectl -n harbor-ce-operator patch installplan "$PLAN" --type=json -p='[{"op": "replace", "path": "/spec/approved", "value": true}]'
```

Wait for CSV creation; Phase changes to `Succeeded`:

```
kubectl -n harbor-ce-operator get csv
```

Example output:

```
NAME                           DISPLAY                  VERSION   REPLACES
PHASE
harbor-ce-operator.v2.12.11   Alauda Build of Harbor   2.12.11   harbor-ce-operator.v2.12.10   Succeeded
```

Fields:

- **NAME**: Installed CSV name
- **DISPLAY**: Operator display name
- **VERSION**: Operator version
- **REPLACES**: CSV replaced during upgrade
- **PHASE**: Installation status (`Succeeded` indicates success)

Automatic

The `clickhouse-operator` comes from a non-Alauda source, and its Approval Strategy can be set to **Automatic**. In Automatic mode, the Operator upgrades automatically when a new version is released, without manual approval.

1. Check available versions

```
(
  echo -e "CHANNEL\tNAME\tVERSION"
  kubectl get packagemanifest clickhouse-operator -o json | jq -r '
    .status.channels[] |
    .name as $channel |
    .entries[] |
    [$channel, .name, .version] | @tsv
  '
) | column -t -s $'\t'
```

Example output:

```
CHANNEL   NAME                           VERSION
stable    clickhouse-operator.v0.18.2    0.18.2
```

2. Confirm catalogSource

```
kubectl get packagemanifests clickhouse-operator -ojsonpath='{.status.cat
alogSource}'
```

Example output:

```
platform
```

This indicates the `clickhouse-operator` comes from the `platform` catalogSource.

3. Create a namespace

```
kubectl create namespace clickhouse-operator
```

4. Create a Subscription

```
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  annotations:
    cpaas.io/target-namespaces: ""
  name: clickhouse-operator-subs
  namespace: clickhouse-operator
spec:
  channel: stable
  installPlanApproval: Automatic
  name: clickhouse-operator
  source: platform
  sourceNamespace: cpaas-system
```

Field explanations are the same as in Manual.

5. Check Subscription status

```
kubectl -n clickhouse-operator get subscriptions clickhouse-operator-subs
-oyaml
```

6. Verify CSV

```
kubectl -n clickhouse-operator get csv
```

Example output:

```
NAME                              DISPLAY                VERSION   PHASE
clickhouse-operator.v0.18.2       ClickHouse Operator    0.18.2    Succeeded
```

Installation is successful.

Advanced Subscription Configuration

The following configuration applies to Subscriptions with either `Manual` or `Automatic` approval.

Configure Operator Pod Scheduling

To control where the Operator Pod runs, configure scheduling settings in `Subscription.spec.config`. This configuration affects only the Operator deployment managed by OLM. It does not control the scheduling of instances or custom resources created by the Operator after installation. To control those workloads, follow the specific Operator's documentation.

Recommendation: Use this configuration when you want Operators to run on infra nodes or on other dedicated nodes.

Prerequisites: The target nodes must already have the required labels. If the target nodes are tainted, configure matching tolerations.

Available fields:

- **.spec.config.nodeSelector**: Optional. Selects the nodes where the Operator Pod can run.
- **.spec.config.tolerations**: Optional. Allows the Operator Pod to run on tainted nodes when the taints match.

Example `Subscription` fragment:

```
spec:
  config:
    nodeSelector:
      node-role.kubernetes.io/infra: ""
    tolerations:
      - key: node-role.kubernetes.io/infra
        operator: Equal
        value: reserved
        effect: NoSchedule
```

If the Operator is already installed, patch the existing `Subscription`:

```
kubectl patch subscription <subscription-name> -n <operator-namespace> --
type='merge' -p '
{
  "spec": {
    "config": {
      "nodeSelector": {
        "node-role.kubernetes.io/infra": ""
      },
      "tolerations": [
        {
          "effect": "NoSchedule",
          "key": "node-role.kubernetes.io/infra",
          "operator": "Equal",
          "value": "reserved"
        }
      ]
    }
  }
}'
```

Upgrade Process

The upgrade process starts by uploading the new **Operator** version.

After the upload is completed, wait approximately **10–15 minutes** for the platform to synchronize the new version information.

Once synchronization is complete, upgrades follow the strategy configured in the **Subscription**:

- If the Operator **Approval Strategy** is set to **Automatic**, the Operator is upgraded automatically.
- If the strategy is set to **Manual**, the upgrade request must be approved manually. You can choose one of the following upgrade methods:
  - **Batch Upgrade**: Execute the upgrade on the **Platform Management > Cluster Management > Cluster > Features** page.
  - **Individual Upgrade**: Manually approve upgrade requests in **OperatorHub**.

Note: Only Operators provided by Alauda support batch upgrades.

Cluster Plugin

TOC

Overview
Viewing Available Plugins
Installing via Web Console
Installing via YAML
non-config
1. Check available versions
2. Create a ModuleInfo
3. Verify installation
with-config
1. Check available versions
2. Create a ModuleInfo
3. Verify installation
Cluster plugin life cycle values
Upgrade Process

Overview

A cluster plugin is a tool for extending the platform's functionality. Each plugin is managed through three cluster-level CRDs: **ModulePlugin**, **ModuleConfig**, and **ModuleInfo**.

- **ModulePlugin**: Defines the basic information of the cluster plugin.
- **ModuleConfig**: Defines the version information of the plugin. Each ModulePlugin can correspond to one or more ModuleConfigs.
- **ModuleInfo**: Records the installed plugin's version and status information.

Cluster plugins support dynamic form configuration. Dynamic forms are simple UI forms that provide customizable configuration options or parameter combinations for plugins. For example, when installing the Alauda Container Platform Log Collector, you can select the log storage plugin as ElasticSearch or ClickHouse via the dynamic form. The dynamic form definition is located in the `.spec.config` field of the ModuleConfig; if the plugin does not require a dynamic form, this field is empty.

Plugins are published via the **violet** tool. Note:

- Plugins can only be published to the **global cluster**, but can be installed on either the global or workload cluster depending on the configuration.
- In the same cluster, a plugin can only be installed once.
- Once published successfully, the platform will automatically create the corresponding ModulePlugin and ModuleConfig in the global cluster—no manual modifications are required.
- Creating a ModuleInfo resource installs the plugin and allows selecting the version, target cluster, and dynamic form parameters. Refer to the ModuleConfig of the selected version for the dynamic form definition. For more usage instructions, refer to the plugin-specific documentation.

Viewing Available Plugins

To view all plugins provided by the platform:

1. Navigate to the platform management view.
2. Click the left navigation menu: **Administrator** > **Marketplace** > **Cluster Plugins**

This page lists all available plugins along with their current status.

Installing via Web Console

If a plugin shows an "absent" status, follow these steps to install it:

1. Download the plugin package:
   - Visit the Alauda Customer Portal to download the corresponding plugin package.
   - If you don't have access to the Alauda Customer Portal, contact technical support.
2. Upload the package to the platform:
   - Use the `violet` tool to publish the package to the platform.
   - For detailed instructions on using this tool, refer to the CLI.
3. Verify the upload:
   - Navigate to **Administrator** > **Marketplace** > **Upload Packages**
   - Switch to the **Cluster Plugin** tab
   - Locate the uploaded plugin name
   - The plugin details will show the version(s) of the uploaded package
4. Install the plugin:
   - If the plugin shows a "ready" status, click **Install**
   - Some plugins require installation parameters; refer to the plugin-specific documentation
   - Plugins without installation parameters will start installation immediately after clicking Install

Installing via YAML

The installation method differs by plugin type:

- **Non-config plugin**: No additional parameters required; installation is straightforward.
- **Config plugin**: Requires filling in configuration parameters; refer to the plugin documentation for details.

> **INFO**
>
> YAML-based installation **must always be performed in the global cluster**.
>
> Although the plugin itself can target either the global cluster or a workload cluster (depending on the affinity settings in the ModuleConfig), the `ModuleInfo` resource can only be created in the **global cluster**.

The following examples demonstrate YAML-based installation.

non-config

Example: Alauda Container Platform Web Terminal

1. Check available versions

Ensure the plugin has been published by checking for ModulePlugin and ModuleConfig resources in the **global cluster**:

```
# kubectl get moduleplugins web-cli
NAME      AGE
web-cli   4d20h

# kubectl get moduleconfigs -l cpaas.io/module-name=web-cli
NAME             AGE
web-cli-v4.0.4   4d21h
```

This indicates that the ModulePlugin `web-cli` exists in the global cluster and version `v4.0.4` is published.

Check the ModuleConfig for version v4.0.4:

```
# kubectl get moduleconfigs web-cli-v4.0.4 -oyaml
apiVersion: cluster.alauda.io/v1alpha1
kind: ModuleConfig
metadata:
  ...
  name: web-cli-v4.0.4
spec:
  affinity:
    clusterAffinity:
      matchLabels:
        is-global: "true"
  version: v4.0.4
  config: {}
  ...
```

The `.spec.affinity` defines cluster affinity, indicating that `web-cli` can only be installed on the global cluster. `.spec.config` is empty, meaning the plugin requires no configuration and can be installed directly.

2. Create a ModuleInfo

Create a ModuleInfo resource in the global cluster to install the plugin without any configuration parameters:

```
apiVersion: cluster.alauda.io/v1alpha1
kind: ModuleInfo
metadata:
  labels:
    cpaas.io/cluster-name: global
    cpaas.io/module-name: web-cli
    cpaas.io/module-type: plugin
  name: global-temporary-name
spec:
  config: {}
  version: v4.0.4
```

Field explanations:

- `name`: Temporary name for the cluster plugin. The platform will rename it after creation based on the content, in the format `<cluster-name>-<hash of content>`, e.g., `global-ee98c9991ea1464aaa8054bdacbab313`.
- `label cpaas.io/cluster-name`: Specifies the target cluster where the plugin should be installed. If it conflicts with the ModuleConfig's affinity, installation will fail.
  Note: This label does not change where the YAML is applied—the YAML **must still be applied in the global cluster**.
- `label cpaas.io/module-name`: Plugin name, must match the ModulePlugin resource.
- `label cpaas.io/module-type`: Fixed field, must be `plugin`; missing this field causes installation failure.
- `.spec.config`: If the corresponding ModuleConfig is empty, this field can be left empty.
- `.spec.version`: Specifies the plugin version to install, must match `.spec.version` in ModuleConfig.

3. Verify installation

Since the ModuleInfo name changes upon creation, locate the resource via label in the global cluster to check the plugin status and version:

```
kubectl get moduleinfo -l cpaas.io/module-name=web-cli
NAME                                        CLUSTER   MODULE    DISPLAY_NAM
E   STATUS    TARGET_VERSION   CURRENT_VERSION   NEW_VERSION
global-ee98c9991ea1464aaa8054bdacbab313   global    web-cli   web-cli
Running   v4.0.4           v4.0.4            v4.0.4
```

Field explanations:

- `NAME`: ModuleInfo resource name
- `CLUSTER`: Cluster where the plugin is installed
- `MODULE`: Plugin name
- `DISPLAY_NAME`: Display name of the plugin
- `STATUS`: Installation status; `Running` means successfully installed and running
- `TARGET_VERSION`: Intended installation version
- `CURRENT_VERSION`: Version before installation

- `NEW_VERSION`: Latest available version for installation

with-config

Example: Alauda Container Platform GPU Device Plugin

1. Check available versions

Ensure the plugin has been published by checking ModulePlugin and ModuleConfig resources in the **global cluster**:

```
# kubectl get moduleplugins gpu-device-plugin
NAME                AGE
gpu-device-plugin   4d23h

# kubectl get moduleconfigs -l cpaas.io/module-name=gpu-device-plugin
NAME                          AGE
gpu-device-plugin-v4.0.15   4d23h
```

This indicates that ModulePlugin `gpu-device-plugin` in the global cluster exists and version `v4.0.15` is published.

Check the ModuleConfig for v4.0.15:

```
# kubectl get moduleconfigs gpu-device-plugin-v4.0.15 -oyaml
apiVersion: cluster.alauda.io/v1alpha1
kind: ModuleConfig
metadata:
  ...
  name: gpu-device-plugin-v4.0.15
spec:
  affinity:
    clusterAffinity:
      matchExpressions:
      - key: cpaas.io/os-linux
        operator: Exists
      matchLabels:
        cpaas.io/arch-amd64: "true"
  config:
    custom:
      mps_enable: false
      pgpu_enable: false
      vgpu_enable: false
  version: v4.0.15
  ...
```

Notes:

- This plugin can only be installed on clusters with Linux OS and amd64 architecture.
- The dynamic form includes three device driver switches: `custom.mps_enable`, `custom.pgpu_enable`, and `custom.vgpu_enable`. Only when set to `true` will the corresponding driver be installed.

2. Create a ModuleInfo

Create a ModuleInfo resource **in the global cluster** to install the plugin, filling in dynamic form parameters as needed (e.g., enabling pgpu and vgpu drivers):

```yaml
apiVersion: cluster.alauda.io/v1alpha1
kind: ModuleInfo
metadata:
  labels:
    cpaas.io/cluster-name: business
    cpaas.io/module-name: gpu-device-plugin
    cpaas.io/module-type: plugin
  name: business-temporary-name
spec:
  config:
    custom:
      mps_enable: false
      pgpu_enable: true
      vgpu_enable: true
  version: v4.0.15
```

Field explanations are the same as non-config. Refer to the plugin documentation for config details.

3. Verify installation

Locate the ModuleInfo via label in the global cluster to check status and version:

```
# kubectl get moduleinfo -l cpaas.io/module-name=gpu-device-plugin
NAME                                        CLUSTER   MODULE              D
ISPLAY_NAME        STATUS    TARGET_VERSION   CURRENT_VERSION   NEW_VERSI
ON
business-7ebb241b4f77471235e57dd1ec7fbd0d business  gpu-device-plugin   g
pu-device-plugin   Running   v4.0.15          v4.0.15           v4.0.15
```

Field explanations are the same as non-config.

Cluster plugin life cycle values

The **Cluster Plugins** page uses the `cpaas.io/lifecycle-type` label to indicate how a cluster plugin is maintained and upgraded. If the label is absent, the plugin is treated as `Agnostic`.

In this context, `Core` means the plugin follows the ACP Core release and upgrade cycle together with the cluster Distribution Version.

The **Cluster Plugins** list page shows a **Life cycle** column and provides a **Life cycle** filter so you can quickly identify each plugin type.

Life cycle	Description	Upgrade path
`Core`	The plugin follows the ACP Core release and upgrade cycle together with the cluster Distribution Version and does not support standalone upgrade.	Upgrade the cluster to update the plugin.
`Aligned`	The plugin follows the ACP release stream but can still be updated independently when a newer plugin version is published.	Upgrade the plugin from the Cluster Plugins list page or details page.
`Agnostic`	The plugin is released independently from ACP.	Upgrade the plugin from the Cluster Plugins list page or details page.

Upgrade Process

To upgrade an existing plugin to a newer version:

1. Upload the new version:
   - Follow the same process to upload the new version to the platform.
   - After the upload is completed, wait approximately **10–15 minutes** for the platform to synchronize the new version information.
2. Verify the new version:
   - Navigate to **Administrator** > **Marketplace** > **Upload Packages**
   - Switch to the **Cluster Plugin** tab
   - The plugin details will show the newly uploaded version
3. Review the plugin:

- Navigate to **Administrator** > **Marketplace** > **Cluster Plugins**
- Select the target cluster
- Review the plugin from the list page or open the plugin details page

4. Perform the upgrade:

- If the plugin uses the `Aligned` or `Agnostic` life cycle, start the standalone upgrade from the list page or details page
- The platform upgrades the installed plugin to a newer published version

5. Handle `Core` plugins:

- `Core` plugins do not support standalone upgrades
- When a newer version is detected, the plugin details page shows a reminder to upgrade the cluster instead

Note: `Upgrade` changes the installed plugin version. `Update` remains a plugin-specific action for changing configuration or running plugin-defined update logic.

Chart Repository

For information about Chart repositories and Helm charts, see Working with Helm Charts.

Download Packages

The platform provides a command-line tool `violet`, which is used to download packages from the platform.

TOC

- Download the Tool
  - For Linux or macOS
  - For Windows
- Prerequisites
- Usage
  - violet ac login
    - Optional Flags
  - violet ac scenarios
    - Optional Flags
    - ACP 4.5 Installation and Upgrade Scenarios
  - violet ac packages
    - Optional Flags
  - violet ac download-pkg
    - Optional Flags
  - violet ac download-app
    - Optional Flags
  - violet ac download-os
    - Flags

violet ac import-yaml

Optional Flags

Example workflows

Download the Tool

Log in to the Alauda Customer Portal, navigate to the **Downloads** page, and click **CLI Tools**. Download the binary that matches your operating system and architecture.

After downloading, install the tool on your server or PC.

For Linux or macOS

For non-root users:

```
# Linux x86
sudo mv -f violet_linux_amd64 /usr/local/bin/violet && sudo chmod +x /us
r/local/bin/violet
# Linux ARM
sudo mv -f violet_linux_arm64 /usr/local/bin/violet && sudo chmod +x /us
r/local/bin/violet
# macOS x86
sudo mv -f violet_darwin_amd64 /usr/local/bin/violet && sudo chmod +x /us
r/local/bin/violet
# macOS ARM
sudo mv -f violet_darwin_arm64 /usr/local/bin/violet && sudo chmod +x /us
r/local/bin/violet
```

For root users:

```
# Linux x86
mv -f violet_linux_amd64 /usr/bin/violet && chmod +x /usr/bin/violet
# Linux ARM
mv -f violet_linux_arm64 /usr/bin/violet && chmod +x /usr/bin/violet
# macOS x86
mv -f violet_darwin_amd64 /usr/bin/violet && chmod +x /usr/bin/violet
# macOS ARM
mv -f violet_darwin_arm64 /usr/bin/violet && chmod +x /usr/bin/violet
```

For Windows

1. Download the file and rename it to `violet.exe`, or use PowerShell to rename it:

   ```
   # Windows x86
   mv -Force violet_windows_amd64.exe violet.exe
   ```

2. Run the tool in PowerShell.

> **Note**: If the tool path is not added to your environment variables, you must specify the full path when running commands.

Use **Violet v4.2.13 or later** for the workflows documented on this page. After installing the binary, verify its version:

```
violet version
```

Prerequisites

Permission requirements

- You must provide a valid platform user account (account, username and password).

Usage

violet ac login

Before downloading packages, use the `violet ac login` command to log in to the platform.

```
violet ac login --account=<account> --username=<username> --password=<pas
sword> --ac-url=<url>
# or provide an existing token:
violet ac login --access-token=<token> --ac-url=<url>
```

Optional Flags

```
--account         account / tenant name
--username        username
--password        password
--ac-url          AC system URL (default: `https://cloud.alauda.io`)
--access-token    directly provide an access token to skip username/passw
ord login
```

> **Note**
>
> You can export an access token from the `Customer Portal - Settings`. The validity period of an access token is within 24 hours from successful login.

violet ac scenarios

List available scenarios

Output: A formatted table listing `ID`, `Name` and `Description`.

```
violet ac scenarios --arch=amd64 --platformVersion=v4.5.0
```

Optional Flags

```
--arch                target architecture (`amd64` , `arm64` , `hybrid`, d
efault: `amd64`)
--platformVersion     target platform version
```

ACP 4.5 Installation and Upgrade Scenarios

Use the following scenarios when preparing ACP 4.5 packages:

Scenario	Use
`Common ACP Extensions`	Download the package set that contains the eight Aligned applications required for a new ACP 4.5 installation, together with other Extensions.
`ACP Upgrade to v4.5`	Download the Aligned Extension packages required when upgrading an existing environment to ACP 4.5.

Set `<target-platform-version>` to the exact ACP Distribution Version of the Core Package, such as `v4.5.0`. Set `<arch>` to `amd64`, `arm64`, or `hybrid` to match the Core Package.

First list the available scenarios to confirm that the expected scenario is published for the target version and architecture. Then download the required scenario:

```
violet ac scenarios --arch=<arch> --platformVersion=<target-platform-vers
ion>

# New installation
violet ac download-app --arch=<arch> --platformVersion=<target-platform-v
ersion> --scenario="Common ACP Extensions"

# Upgrade to ACP 4.5
violet ac download-app --arch=<arch> --platformVersion=<target-platform-v
ersion> --scenario="ACP Upgrade to v4.5"
```

Downloading a scenario does not add its packages to the Core Package. Extract the Core Package and manually copy the packages required by the installation or upgrade procedure into its `plugins/` directory. For the exact application names and copy steps, see Download and Pre-Upgrade Preparation.

violet ac packages

List available packages.

Output: A formatted table listing `APP ID`, `APP Name`, `Channel And Version` and `Package`.

```
# download all packages for the specified architecture, platform version, and scenario
violet ac packages --arch=<arch> --platformVersion=<version> --scenario=<scenario>
# download all packages for the specified architecture and platform version
violet ac packages --arch=<arch> --platformVersion=<version>
# download single package for a specific application ID
violet ac packages --appID=my-app
```

Optional Flags

```
--arch              target architecture (`amd64` , `arm64` , `hybrid`, default: `amd64`)
--platformVersion   target platform version
--appID             download single app for a specific application ID
--scenario          scenario filter (optional)
```

> **Note**
>
> About `scenario`, if not configured, all packages are displayed.

violet ac download-pkg

Download packages and their signature files for a specific architecture and platform version.

```
violet ac download-pkg --arch=x86 --platformVersion=v4.1.0 --type=core
```

Optional Flags

```
--arch                target architecture (`x86` , `arm` , `hybrid`, defau
lt: `x86`)
--platformVersion     target platform version
--type                package type (`core`, `extensions`, `standard`, defa
ult: `core`)
```

> **Note**: About `type`, For versions v4.0.5 and later, the tool downloads the core package by default — you do not need to set this option.
> For v4.0.0 through v4.0.4, the default is `core` to download the core package; you can set the option to `extensions` to download extension packages.

violet ac download-app

Download application packages either by scenario (batch) or by specifying `appID` and `appVersion`. The command obtains download URLs and then fetches the package and checksum files.

```
violet ac download-app --arch=amd64 --platformVersion=<version> --scenari
o=<scenario>
# List available packages status
violet ac download-app --arch=amd64 --platformVersion=<version> --scenari
o=<scenario> --check=true
# or download a specific app version
violet ac download-app --arch=amd64 --appID=<app-id> --appVersion=<app-ve
rsion1>,<app-version2>
```

Optional Flags

```
--arch              target architecture (`amd64` , `arm64` , `hybrid`, d
efault: `amd64`)
--platformVersion   target platform version
--appID             specific application ID
--appVersion        application version (supports multiple versions sepa
rated by commas)
--check             If true, do not download; only check available packa
ges status (default: `false`)
--scenario          scenario name (optional, download latest versions fo
r the scenario)
```

> **Note**
>
> About `scenario`, if not configured, all packages are downloaded.

violet ac download-os

Download OS packages from the AC Marketplace OS page. Use `--list` first to inspect available platform versions, architectures, and operating systems, then download the matching OS package and checksum file.

```
# List all available OS package filters
violet ac download-os --list
# List OS packages for a specific platform version
violet ac download-os --platformVersion=v4.1.0 --list
# Download matching OS packages
violet ac download-os --platformVersion=v4.1.0 --arch=x86_64 --operatingS
ystem=ubuntu
```

Flags

```
--platformVersion   target platform version (required when downloading)
--arch              target architecture (required when downloading)
--operatingSystem   target operating system (required when downloading)
--list              list matching OS packages without downloading (defau
lt: `false`)
```

Note

Use `--list` with fewer filters to discover valid `platformVersion`, `arch`, and `operatingSystem` values before downloading.

violet ac import-yaml

Read a local YAML file (default `./apps.yaml`, `violet list` export) containing an `applications` map, send it to the AC scenario-check endpoint, and display validation results. Optionally download validated packages.

```
violet ac import-yaml --arch=amd64 --platformVersion=v4.1.3 --download=tr
ue
```

Optional Flags

```
--arch               target architecture (`amd64` , `arm64` , `hybrid`, d
efault: `amd64`)
--platformVersion    target platform version
--download           boolean; if true, attempt to download validated pack
ages after the check
--path               path to the YAML file (default `./apps.yaml`)
```

Example workflows

- Login and save token:

```
violet ac login --account=tenantA --username=admin --password=password
--ac-url=https://ac.example.com
```

- List available scenarios:

```
violet ac scenarios --arch=amd64 --platformVersion=v4.1
```

- Download the target version core packages:

```
violet ac download-pkg --arch=x86 --platformVersion=v4.1.0
```

- List available packages and download the latest version for a scenario:

```
violet ac download-app --arch=amd64 --platformVersion=v4.1.0 --scenario
=my-scenario
```

- List available packages status('Downloaded' or 'Download Failed'):

```
violet ac download-app --arch=amd64 --platformVersion=v4.1.0 --scenario
=my-scenario --check=true
```

- List and download OS packages:

```
violet ac download-os --platformVersion=v4.1.0 --list
violet ac download-os --platformVersion=v4.1.0 --arch=x86_64 --operatin
gSystem=ubuntu
```

- Validate an applications YAML and download validated packages:

```
violet ac import-yaml --arch=amd64 --platformVersion=v4.1.0 --download=
true --path=./apps.yaml
```

Upload Packages

The platform provides a command-line tool `violet`, which is used to upload packages downloaded from the Marketplace in the Alauda Customer Portal to the platform.

`violet` supports uploading the following types of packages:

- **Operator**
- **Cluster Plugin**
- **Helm Chart**

When the status of a package in **Cluster Plugins** or **OperatorHub** is shown as `Absent`, you need to use this tool to upload the corresponding package.

The upload process of `violet` mainly includes the following steps:

1. Extract and retrieve information from the package
2. Push images to the image registry
3. Create **Artifact** and **ArtifactVersion** resources on the platform

---

TOC

Download the Tool

For Linux or macOS

For Windows

Prerequisites

Usage

Common Parameters
Platform Connection Parameters
Image Registry Parameters
violet show
violet list
Optional Flags
violet gc
Optional Flags
violet verify
Optional Flags
violet push
Optional Flags
Upload an Operator to Multiple Clusters
Upload an Operator to a Standby Global Cluster
Upload a Cluster Plugin
Upload a Helm Chart to the chart repository
Push all packages at once

Download the Tool

Log in to the Alauda Customer Portal, navigate to the **Downloads** page, and click **CLI Tools**. Download the binary that matches your operating system and architecture.

After downloading, install the tool on your server or PC.

For Linux or macOS

For non-root users:

```
# Linux x86
sudo mv -f violet_linux_amd64 /usr/local/bin/violet && sudo chmod +x /us
r/local/bin/violet
# Linux ARM
sudo mv -f violet_linux_arm64 /usr/local/bin/violet && sudo chmod +x /us
r/local/bin/violet
# macOS x86
sudo mv -f violet_darwin_amd64 /usr/local/bin/violet && sudo chmod +x /us
r/local/bin/violet
# macOS ARM
sudo mv -f violet_darwin_arm64 /usr/local/bin/violet && sudo chmod +x /us
r/local/bin/violet
```

For root users:

```
# Linux x86
mv -f violet_linux_amd64 /usr/bin/violet && chmod +x /usr/bin/violet
# Linux ARM
mv -f violet_linux_arm64 /usr/bin/violet && chmod +x /usr/bin/violet
# macOS x86
mv -f violet_darwin_amd64 /usr/bin/violet && chmod +x /usr/bin/violet
# macOS ARM
mv -f violet_darwin_arm64 /usr/bin/violet && chmod +x /usr/bin/violet
```

For Windows

1. Download the file and rename it to `violet.exe`, or use PowerShell to rename it:

   ```
   # Windows x86
   mv -Force violet_windows_amd64.exe violet.exe
   ```

2. Run the tool in PowerShell.

> **Note**: If the tool path is not added to your environment variables, you must specify the full path when running commands.

Use **Violet v4.2.13 or later** for the workflows documented on this page. After installing the binary, verify its version:

```
violet version
```

Prerequisites

Permission requirements

- You must provide valid platform credentials, either a username and password or a platform token.
- The account must have the role property set to `System` and the role name must be `platform-admin-system`.

> **Note:** If the role property of your account is set to `Custom`, you cannot use this tool.

Usage

Common Parameters

Several `violet` commands accept the following parameters. Refer to individual command sections for specific usage.

Platform Connection Parameters

```
--platform-address <platform access URL>     # The access URL of the plat
form, e.g., "https://example.com"
--platform-username <platform user>          # The username of the platfo
rm user
--platform-password <platform password>      # The password of the platfo
rm user
--platform-token <platform token>            # The platform access token;
cannot be used together with username/password
```

To create or manage platform tokens, sign in to the platform Web Console, open **Profile** > **API Tokens**, and create a token with an appropriate expiration. For details, see API

Authentication.

> **WARNING**
>
> **Identity provider (IdP) users must use a platform token.** Username and password login authenticates only against the platform's local user store. If your account comes from an external identity provider (for example, LDAP or OIDC), username and password login fails — use `--platform-token` instead.

> **Violet versions earlier than v4.2.13**
>
> Earlier versions can fail when `--platform-token` is used and report `failed to get user info from platform server`. Upgrade to Violet v4.2.13 or later before using the token-based workflows documented on this page.

Image Registry Parameters

```
--dest-repo <image repository address>       # Specify the destination im
age repository address, e.g., "harbor.demo.io"
--username <registry user>                   # The username of the specif
ied image registry
--password <registry password>               # The password of the specif
ied image registry
--no-auth                                    # Specify if the image regis
try does not require authentication
--plain                                      # Specify if the image regis
try uses HTTP instead of HTTPS
```

> **WARNING**
>
> **IPv6 Address Limitation**
>
> For `--platform-address` and `--dest-repo` parameters:
>
> - If using an IP address (rather than a domain name), **IPv6 format is NOT supported**
> - Only IPv4 addresses or domain names are supported

violet show

Before uploading a package, use the `violet show` command to preview its details.

```
violet show topolvm-operator.v2.3.0.tgz
Name: NativeStor
Type: bundle
Arch: [linux/amd64]
Version: 2.3.0

violet show topolvm-operator.v2.3.0.tgz --all
Name: NativeStor
Type: bundle
Arch: []
Version: 2.3.0
Artifact: harbor.demo.io/acp/topolvm-operator-bundle:v3.11.0
RelateImages: [harbor.demo.io/acp/topolvm-operator:v3.11.0 harbor.demo.i
o/acp/topolvm:v3.11.0 harbor.demo.io/3rdparty/k8scsi/csi-provisioner:v3.0
0 ...]
```

violet list

When upgrading the platform, use `violet list` to list installed extensions and export the result to a YAML file. The generated file can then be uploaded to Alauda Cloud so that the required extension packages can be downloaded.

`violet list` covers extensions with the **Aligned** or **Agnostic** life cycle. **Core** plugins follow the cluster upgrade and are not managed through `violet list`.

The command lists the following application types:

- **ModulePlugin** — cluster plugins whose lifecycle type is `aligned` or `agnostic`.
- **OperatorBundle** — installed operators.
- **Chart** — Helm chart applications created from chart references.

By default, the command prints the YAML content to stdout and writes it to `apps.yaml` in the current directory.

```
violet list \
  --platform-address "https://example.com" \
  --platform-username "<platform_user>" \
  --platform-password "<platform_password>"
```

You can also authenticate with a platform token instead of username and password:

```
violet list \
  --platform-address "https://example.com" \
  --platform-token "<platform_token>"
```

To export only applications installed in specific clusters, specify multiple cluster names separated by commas:

```
violet list \
  --platform-address "https://example.com" \
  --platform-token "<platform_token>" \
  --clusters "global,region1" \
  --output-file "./apps.yaml"
```

Example output:

```
applications:
  topolvm-operator:
    displayName: NativeStor
    type: OperatorBundle
    installed:
      - clusterName: global
        version: 2.3.0
  log-center:
    displayName: Log Center
    type: ModulePlugin
    installed:
      - clusterName: region1
        version: v1.2.0
```

Optional Flags

```
--clusters <cluster names>                    # List applications installe
d in the specified clusters, separated by commas
--output-file <output file>                   # File path for the output a
pplication list (default: "apps.yaml")
--debug                                       # Use debug log level
```

For platform connection parameters (`--platform-address` , `--platform-username` , `--platform-password` , `--platform-token`), see Common Parameters.

violet gc

Use `violet gc` to clean up uninstalled Operator and Cluster Plugin resources whose life cycle is **Aligned** or **Agnostic**. The command deletes resources that are no longer installed or no longer match the installed version.

The command can delete the following resource types:

- **Artifact** — artifacts for uninstalled operators.
- **ArtifactVersion** — old artifact versions for installed operators. The currently installed CSV or version is kept.
- **ModulePlugin** — cluster plugins that are not installed in any cluster.
- **ModuleConfig** — obsolete cluster plugin configuration records.

`violet gc` determines cleanup candidates by checking `OperatorView` and `ModulePluginView` records in the global cluster.

> **DANGER**
>
> `violet gc` deletes resources from the global cluster and does not provide a dry-run mode or undo operation. Run it only during a documented maintenance window and verify the target clusters before using `--clusters` .

By default, the command scans all clusters visible to the platform user and prints a summary of deleted resources.

```
violet gc \
  --platform-address "https://example.com" \
  --platform-username "<platform_user>" \
  --platform-password "<platform_password>"
```

You can also authenticate with a platform token instead of username and password:

```
violet gc \
  --platform-address "https://example.com" \
  --platform-token "<platform_token>"
```

To clean up resources related to specific clusters only, specify multiple cluster names separated by commas:

```
violet gc \
  --platform-address "https://example.com" \
  --platform-token "<platform_token>" \
  --clusters "global,region1"
```

Example output:

```
GC Summary:
TYPE                 NAME                                                C
LUSTER
-------------------------------------------------------------------------
-----------------
Artifact             opensearch-operator                                 r
egion1
ArtifactVersion      opensearch-operator.v2.8.0                          r
egion1
ModuleConfig         log-center-default                                  g
lobal
```

If no resources are deleted, the command prints:

```
No resources were deleted.
```

Optional Flags

```
--clusters <cluster names>                    # Limit cleanup to the speci
fied clusters, separated by commas
--debug                                       # Use debug log level
```

For platform connection parameters (`--platform-address` , `--platform-username` , `--platform-password` , `--platform-token`), see Common Parameters.

violet verify

Use the `violet verify` command to verify the signature of one or more packages before uploading them. Two verification methods are supported: **checksum** and **GPG**. The package (`.tgz`) and its corresponding signature file must be located in the same directory.

```
violet verify example.tgz
# or verify all packages within a directory
violet verify packages_dir_name
```

Example output:

```
verify path: /path/to/packages
====== Verification Summary ======
Verified successfully with GPG: 2 file(s)
  - /path/to/packages/redis-operator.tgz
  - /path/to/packages/mysql-operator.tgz

Verified successfully with checksum: 1 file(s)
  - /path/to/packages/nginx-controller.tgz

Verification failed: 1 file(s)
  - /path/to/packages/etcd-operator.tgz

No verification file found: 1 file(s)
  - /path/to/packages/demo-plugin.tgz
```

Explanation:

- **Verified successfully with GPG** — The listed files have been successfully verified using **GPG signature files** (with `.sig` extension).
- **Verified successfully with checksum** — Files verified using checksum files (e.g., `.sha256`) passed the integrity check.
- **Verification failed** — The listed files failed verification due to mismatched or invalid signatures.
- **No verification file found** — No corresponding `.sig` (GPG) or checksum file was found in the directory.

Optional Flags

```
--debug       Use debug log level.
-h, --help    Display help information for the verify command.
```

violet push

The following examples illustrate common usage scenarios.

For platform connection and image registry parameters, see Common Parameters.

For non-interactive use, prefer `--platform-token` over `--platform-password`. See Common Parameters.

Optional Flags

```
--clusters <cluster names>                    # Specify target clusters, s
eparated by commas (e.g., region1,region2)
--target-catalog-source <catalog source>      # Target CatalogSource for o
perators: "platform", "system", or a custom CatalogSource name (default:
"platform")
--target-chartrepo <chart repository>         # Target chart repository fo
r Helm charts (default: "public-charts", the platform-managed Helm reposi
tory)
--skip-crs                                    # Skip creating custom resou
rces in the cluster
--skip-push                                   # Skip pushing images and ar
tifacts to the registry
```

`--skip-crs` and `--skip-push` cannot be specified together.

When `--dest-repo` is specified, either the authentication info of the image registry or `--no-auth` MUST be provided.

Upload an Operator to Multiple Clusters

```
violet push opensearch-operator.v3.14.2.tgz \
  --platform-address "https://example.com" \
  --platform-username "<platform_user>" \
  --platform-password "<platform_password>" \
  --clusters region1,region2
```

INFO

- If `--clusters` is not specified, the Operator is uploaded to the **global cluster** by default.

Upload an Operator to a Standby Global Cluster

```
violet push opensearch-operator.v3.14.2.tgz \
  --platform-address "https://<standby-platform-address>" \
  --platform-username "<platform_user>" \
  --platform-password "<platform_password>" \
  --dest-repo "<standby-cluster-VIP>:11443" --username "<registry-usernam
e>" --password "<registry-password>"
```

WARNING

When using `violet` to upload packages to a standby cluster:

- The parameter `--dest-repo <VIP addr of standby cluster>` **MUST** be specified
- The parameter `--platform-address` **MUST** be set to the **standby cluster's** platform access address
- Either the authentication info of the standby cluster's image registry or `--no-auth` parameter MUST be provided

Otherwise, the packages will be uploaded to the image repository of the **primary cluster**, preventing the standby cluster from installing or upgrading extensions.

Upload a Cluster Plugin

```
violet push plugins-cloudedge-v0.3.16-hybrid.tgz \
  --platform-address "https://example.com" \
  --platform-username "<platform_user>" \
  --platform-password "<platform_password>"
```

INFO

- You do not need to specify the `--clusters` parameter when uploading a Cluster Plugin, as the platform will automatically distribute it based on its affinity configuration. If you specify `--clusters`, the parameter will be ignored.

Upload a Helm Chart to the chart repository

```
violet push plugins-cloudedge-v0.3.16-hybrid.tgz \
  --platform-address "https://example.com" \
  --platform-username "<platform_user>" \
  --platform-password "<platform_password>"
```

INFO

- Helm Charts can only be uploaded to the default `public-charts` repository provided by the platform.

Push all packages at once

When multiple packages are downloaded from the Marketplace, you can place them in the same directory and upload them all at once:

```
violet push <packages_dir_name> \
  --platform-address "https://example.com" \
  --platform-username "<platform_user>" \
  --platform-password "<platform_password>" \
  --clusters "<cluster_name>"
```

WARNING

When the upgrade target is the **global cluster** , you can omit the `--clusters` parameter, as it defaults to uploading to the global cluster.

However, when the upgrade target is a workload cluster, you **must** specify the `--clusters <workload_cluster_name>` parameter.