Menu

# Extend

## Overview

## Operator

## Cluster Plugin

## Chart Repository

## Upload Packages

Menu

# 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

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

```
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

```
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.catalogSource}'
```

Example output:

```
community-operators
```

This indicates the `clickhouse-operator` comes from the `community-operators` catalogSource.

## 3. Create a namespace

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

## 4. Create a Subscription

```yaml
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: community-operators
  sourceNamespace: openshift-marketplace
```

Field explanations are the same as in Manual.

### 5. Check Subscription status

```
kubectl -n clickhouse-operator get subscriptions clickhouse-operator -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.

## Upgrade Process

1. Upload the new Operator version.

2. Upgrades follow the strategy configured in the Subscription:

- **Automatic Upgrade**: Upgrades automatically upon upload.
- **Manual Upgrade**:
  - **Batch Upgrade**: Execute on **Platform Management > Cluster Management > Cluster > Features** page.
  - **Individual Upgrade**: Manually approve upgrade requests in OperatorHub.

**Note**: Only Operators from Alauda support batch upgrades.

# Cluster Plugin

## TOC



## 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 Plugin**

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_NAME   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              DISPLAY_NAME
STATUS    TARGET_VERSION   CURRENT_VERSION   NEW_VERSION
business-7ebb241b4f77471235e57dd1ec7fbd0d business  gpu-device-plugin   gpu-device-plugin
Running   v4.0.15          v4.0.15           v4.0.15
```

Field explanations are the same as non-config.

## 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.

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. Perform the upgrade:

   - Navigate to **Administrator** > **Clusters** > **Clusters**
   - Clusters with upgradable plugins will display an upgrade icon
   - Enter the cluster details and switch to the **Features** tab
   - The upgrade button will be enabled under the features component
   - Click **Upgrade** to complete the plugin upgrade

# Chart Repository

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

# 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

**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
/usr/local/bin/violet
# Linux ARM
sudo mv -f violet_linux_arm64 /usr/local/bin/violet && sudo chmod +x
/usr/local/bin/violet
# macOS x86
sudo mv -f violet_darwin_amd64 /usr/local/bin/violet && sudo chmod +x
/usr/local/bin/violet
# macOS ARM
sudo mv -f violet_darwin_arm64 /usr/local/bin/violet && sudo chmod +x
/usr/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.

# Prerequisites

### Permission requirements

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

## 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.io/acp/topolvm:v3.11.0 harbor.demo.io/3rdparty/k8scsi/csi-provisioner:v3.00
...]
```

## 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.
Before digging into the examples, here are some common OPTIONAL parameters used in the commands:

```
--platform-address <platform access URL>     # The access URL of the platform, e.g.,
"https://example.com"
--platform-username <platform user>          # The username of the platform user
--platform-password <platform password>      # The password of the platform user
--clusters <cluster names>                   # Specify target clusters, separated by
commas (e.g., region1,region2)

--dest-repo <image repository URL>           # Specify the destination image repository
URL. MUST be specified when uploading extensions to a standby cluster.
                                                 When `--dest-repo` is specified, either
the authentication info of the image registry or `--no-auth` MUST be provided.
--username <registry user>                   # The username of the specified image
registry.
--password <registry password>               # The password of the specified image
registry.
--no-auth                                    # Specify if the image registry does not
require authentication.
--plain                                      # Specify if the image registry uses HTTP
instead of HTTPS.
```

## 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://example.com" \
  --platform-username "<platform_user>" \
  --platform-password "<platform_password>" \
  --dest-repo "<standby-cluster-VIP>:11443" --username "<registry-username>" --password "
<registry-password>"
```

## 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.