Install

Installation Path Selection

Choose the installation path by the operating system that the `global` cluster nodes run:

- Install on Alauda OS: the nodes run Alauda OS and are provisioned through an infrastructure provider.
- Install on Non-Alauda OS: the nodes run a **non-Alauda OS** such as Ubuntu or RHEL.

Overview

Overview

Plan

Plan

Prepare for Installation

Download | Prepare an External Platform In | Communica

Install on Alauda OS

Installing | Global Cluster Disaster Recovery

Install on Non-Alauda OS

Prerequisites | Node Preprocessing | Installing

Installer Parameters | Global Cluster Disaster Recove | Installation

Validation

Validation

Next Steps

Next Steps

Overview

Use the Install section to plan and complete an ACP 4.5 installation.

The installation deploys ACP Core and the required Aligned Extensions on the `global` cluster. Core provides the platform management plane, cluster management, users and roles, and Extension management frameworks. The required Aligned Extensions provide the Web Console and other platform-facing capabilities. For the platform architecture and definitions of `global` cluster and workload cluster, see Architecture and Glossary.

The Core Package and the required Aligned Extension packages are separate downloads. Before starting the installer, copy the eight required Extension packages listed in Download into the `plugins/` directory of the extracted Core Package. Other Extensions are installed after the platform installation through the Extend workflow. On Alauda OS, the installation also uploads the Cluster Bootstrap Provider and infrastructure provider packages to a temporary bootstrap cluster; see Installing the global Cluster.

INFO

ACP Core installation does not support direct installation of the `global` cluster into an existing Kubernetes environment. On a non-Alauda OS, prepare new nodes that meet the hardware, network, storage, and OS requirements before you start installation. On Alauda OS, the installation provisions the `global` cluster nodes through an infrastructure provider; on Bare Metal you supply the hosts.

TOC

Installation Scope

Choose an Installation Path

Installation Workflow

Related Documentation

Installation Scope

The installation provides the platform management plane and the required Aligned Extensions. After the installation is verified, you can create or connect workload clusters and install additional Extensions according to your platform scenario.

The following work is outside this installation flow:

- Creating workload clusters.
- Connecting existing Kubernetes clusters.
- Uploading and installing Extensions other than the eight required Aligned Extensions.
- Installing optional Operators or Cluster Plugins.
- Configuring optional platform capabilities such as storage, monitoring, backup, registry, or identity integrations.

Choose an Installation Path

Choose the path by the operating system that the `global` cluster nodes run:

Node operating system	Installation path
Alauda OS, provisioned through an infrastructure provider on Huawei DCS, VMware vSphere, Huawei Cloud Stack, or Bare Metal.	Install on Alauda OS
A non-Alauda OS such as Ubuntu or RHEL, on nodes that you prepare.	Install on Non-Alauda OS

Installation Workflow

The installation follows this high-level path:

1. Plan. Choose the deployment architecture and confirm decisions that cannot be changed safely after installation. If required, read the Global Cluster Disaster Recovery page for your path before preparing resources: Alauda OS or non-Alauda OS. DR affects addresses, certificates, load balancing, registry selection, and the consistency of both installations.
2. Prepare for installation. Complete the shared preparation that applies to your path:
   - Download. Download the matching Core Package and the required Aligned Extension packages.
   - If selected, prepare the external platform image registry and upload the complete target-version payload.
   - If you enable a host firewall on non-Alauda OS nodes, use the Communication Matrix to build the allow rules.
3. Install the `global` cluster on one of the two paths:
   - Install on Alauda OS. Run a temporary bootstrap cluster on a bootstrap node, upload the provider packages, apply the provider-specific `global` manifest, and trigger the platform installation. This path also covers the optional primary and standby deployment and the removal of the bootstrap cluster.
   - Install on Non-Alauda OS. Meet the hardware, storage, network, and node requirements in Prerequisites and Node Preprocessing. Then extract the Core Package, stage the required packages, start the installer, and complete the Web UI workflow. Use Installer Parameters while reviewing the values entered in the installer. If the installer stalls or validation fails, use Installation Troubleshooting.
4. Validation. Accept the installation only after the Web UI, Core, required Aligned Extensions, image source, artifacts, and Pods are healthy.
5. Next Steps. Install the Product Docs plugin and continue with scenario-specific configuration.

The Customer Portal is the official source for ACP installation packages. For download requirements and package architecture options, see Download.

Separate Downloads

The Core Package does not contain the required Aligned Extension packages. Download them separately and manually copy only the eight packages listed in Download into the `plugins/` directory of the extracted Core Package before starting the installer.

Related Documentation

- Architecture
- Glossary
- Kubernetes Support Matrix
- Disk Configuration Requirements
- Extend

Plan

Before you start the installation, choose the deployment architecture and confirm the installation settings that affect later operations. The installation paths are Install on Alauda OS and Install on Non-Alauda OS. Where a decision applies to only one path, this page says so.

INFO

The ACP installation deploys the `global` cluster on new nodes: nodes that you prepare on a non-Alauda OS, or Alauda OS nodes that the installation provisions through an infrastructure provider. It does not support direct installation of the `global` cluster into an existing Kubernetes environment.

TOC

Choose a Deployment Architecture

Confirm Install-Time Decisions

Read Before You Install

Choose a Deployment Architecture

Choose the architecture that matches your business scenario, management model, workload placement, resource planning, and isolation requirements.

Architecture	Use when	Planning notes
Multi-Cluster	You need one `global` cluster to manage multiple workload clusters.	Avoid running non-platform workloads on the `global` cluster. Plan resources for the expected number of managed workload clusters.
Single Cluster	You intentionally use one cluster for platform components and application workloads.	Single Cluster supports production use when it fits the business scenario. Because the `global` cluster also runs application workloads, plan resources and platform/workload isolation before installation.
Single Node	You need a test or proof-of-concept environment.	Do not use Single Node for production. On Alauda OS, whether a single control plane node is possible depends on the provider; see the following list.

On Alauda OS, a single control plane node depends on the provider:

- **VMware vSphere**: not supported. A `KubeadmControlPlane` needs at least three replicas. See Provider Requirements.
- **Bare Metal**: not supported. Provision at least three control plane replicas. See Step 3: Create the Control-Plane Cluster Resources.
- **Huawei Cloud Stack**: supported for creation only, not for upgrade. See Create a Cluster on Huawei Cloud Stack.

For resource requirements and sizing guidance on a non-Alauda OS, see Prerequisites. For Alauda OS, see Common Prerequisites and, for `global` cluster sizing, Evaluating Resources for Global Cluster. Bootstrap Node Requirements apply to the temporary bootstrap node only.

Confirm Install-Time Decisions

Confirm the following decisions before you start the installation. The last two columns link to the page that documents each decision on each path.

Decision	What to confirm	Non-Alauda OS	Alauda OS
Installation packages	Choose an x86, ARM, or hybrid Core Package and download the required Aligned Extension packages for the same ACP version and architecture. On Alauda OS, also prepare the provider packages and the Alauda OS image or VM template.	Download	Download and Common Prerequisites
Cluster network protocol	On a non-Alauda OS, choose IPv4, IPv6, or dual stack before starting the installer. On Alauda OS, plan the Pod and Service CIDR ranges.	Installing	Common Prerequisites
Cluster Endpoint and Platform Access Address	Prepare the access addresses that cluster components, administrators, and users will use.	Prerequisites and Installer Parameters	Common Prerequisites and Plan the Control Plane Endpoint
Load balancing	Decide whether to use an external load balancer or Self-built VIP. On Alauda OS, the endpoint modes available depend on the provider.	Prerequisites and LoadBalancer Forwarding Rules	Plan the Control Plane Endpoint
Certificates	Decide whether to use installer-generated self-signed certificates or trusted certificates that you provide.	Prerequisites and Installer Parameters	Trigger the Platform Installation
Platform image source	Decide whether to use the platform built-in Registry or a customer-managed external platform image registry. For new production deployments, use an	Prepare an External Platform Image Registry and Installer Parameters	Prepare an External Platform Image Registry and Common Prerequisites

Decision	What to confirm	Non-Alauda OS	Alauda OS
	external registry when it meets the documented availability and lifecycle requirements.		
Nodes	Confirm node names, node roles, and whether platform node isolation is required.	Node Preprocessing and Installer Parameters	Common Prerequisites
Global Cluster Disaster Recovery	Decide whether your environment requires primary and standby `global` clusters.	Global Cluster Disaster Recovery	Global Cluster Disaster Recovery and Optional Disaster Recovery Deployment

If you plan to use Global Cluster Disaster Recovery, read the Global Cluster Disaster Recovery page for your installation path before installing ACP Core. Disaster recovery planning affects domain names, VIPs, certificates, load balancer rules, image registry choices, package architecture, and installation parameter consistency.

Read Before You Install

Use the following pages as the source of truth before you start the installation.

For both installation paths:

- Disk Configuration Requirements: Confirm disk capacity and performance requirements.
- Download: Download the Core Package and `Common ACP Extensions` separately, and verify that their version and architecture match.
- Prepare an External Platform Image Registry: If selected, upload the complete target-version Core and Extension payload before starting the installer.

For a non-Alauda OS:

- Prerequisites: Prepare resource, network, and storage requirements.

- Node Preprocessing: Check and prepare each node before installation.
- Global Cluster Disaster Recovery: Plan primary and standby `global` clusters if disaster recovery is required.
- Installing: Upload the Core Package, stage required packages, and start the installer.
- Installer Parameters: Configure the installation choices in the Web UI.

For Alauda OS:

- Installing the global Cluster: Meet the common prerequisites and bootstrap node requirements, then install the `global` cluster through the bootstrap cluster for your provider.
- Global Cluster Disaster Recovery: Plan primary and standby `global` clusters if disaster recovery is required.

Prepare for Installation

Download

Prepare an External Platform In

Communica

Download

Before installation, download the **Core Package** and the required Aligned Extension packages separately from the **Alauda Customer Portal**.

TOC

About the Customer Portal
Download the Installation Packages
Download with Violet CLI (Recommended)
Download from the Customer Portal
Required Aligned Extensions
Choose a Package Architecture
Migrating from Single-Architecture to Hybrid

About the Customer Portal

The **Alauda Customer Portal** is the official service and delivery portal for ACP product resources. Use it to obtain verified software packages and official technical guidance for installation, upgrade, and maintenance activities.

The Customer Portal provides access to the following resources:

- Product downloads, including installation, upgrade, and extension packages.

- Product documentation and technical guidance.
- Support resources for submitting, tracking, and managing support requests.
- Marketplace packages for extending platform capabilities.
- License-related resources for deployment and maintenance planning.

Use an authorized account to download the package required for your environment. If you do not have a registered account, contact technical support.

For customer delivery and production environments, use the package versions and documentation published on the Customer Portal as the official deployment and maintenance baseline.

Download the Installation Packages

1. Download the ACP 4.5 **Core Package** that matches your target architecture.
2. Download **Common ACP Extensions** for the same ACP version and architecture. The Violet CLI is the recommended method because it can locate and download the scenario packages directly.
3. If you install the `global` cluster on Alauda OS, also download the Alauda OS image for your node architecture and provider. See Downloading Alauda OS.

Download with Violet CLI (Recommended)

Install **Violet v4.2.13 or later**, run `violet version` to verify the version, and use `violet ac login` to log in to the Alauda Customer Portal. For installation and login instructions, see Download Packages.

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. List the scenarios first, and then download **Common ACP Extensions**:

```
violet ac scenarios --arch=<arch> --platformVersion=<target-platform-vers
ion>
violet ac download-app --arch=<arch> --platformVersion=<target-platform-v
ersion> --scenario="Common ACP Extensions"
```

Confirm that **Common ACP Extensions** appears in the scenario list before downloading it.

Download from the Customer Portal

As an alternative to Violet, in the Customer Portal, go to **Marketplace > Batch Download > Install**. On the page, select the Core Package version from **Your ACP Version**, select the matching **Architecture**, select **Common ACP Extensions** from **Scenarios**, and download the packages.

> **Separate Packages**
>
> The Core Package contains Core artifacts only. It does not contain any of the Extension packages downloaded through **Common ACP Extensions**. After extracting the Core Package, you must manually copy the required Extension packages into its `plugins/` directory before starting the installer.

Required Aligned Extensions

Copy the packages for the following applications into the `plugins/` directory of the extracted Core Package:

- **Alauda Container Platform Web Console**
- **Alauda Container Platform Web Console Central**
- **Alauda Container Platform Marketplace Web Console**
- **Alauda Container Platform Application Catalog**
- **Alauda Container Platform Observability Essentials**
- **Alauda Container Platform Essentials**
- **Alauda Container Platform Cluster Essentials**

- **Alauda Container Platform GatewayAPI Plugin**

Do not copy other packages from **Common ACP Extensions** into the extracted Core Package's `plugins/` directory. Install those Extensions after the platform installation by following Extend. On Alauda OS, the installation also uses the Cluster Bootstrap Provider and infrastructure provider packages listed under **Common ACP Extensions**; Installing the global Cluster uploads them to the bootstrap cluster instead of `plugins/`.

Choose a Package Architecture

Packages are available for **x86**, **ARM**, and **hybrid** architectures. The hybrid package includes images for both x86 and ARM, resulting in a larger package size. Select the package that best matches your environment, and use Extension packages that match the Core Package architecture.

Migrating from Single-Architecture to Hybrid

If you initially installed with an x86 or ARM Core Package but later need to support another architecture, you must re-download the **hybrid Core Package** and perform the following steps:

1. Upload the newly downloaded hybrid Core Package to any control plane node of the global cluster.
2. Extract the package and use the included `upgrade.sh` script to synchronize the multi-architecture images to your image registry:

   ```
   bash upgrade.sh --only-sync-image=true
   ```

3. After the script completes, check the `cluster.platform.tkestack.io` resource to verify whether the label `cpaas.io/node-arch-constraint` exists. If it does, you must remove it:

```
kubectl get cluster.platform.tkestack.io global -oyaml | grep cpaas.io/
node-arch-constraint
# If there is output, edit the resource to remove the label; otherwise,
you can skip this step.
kubectl edit cluster.platform.tkestack.io global   ### Edit the labels
field and delete cpaas.io/node-arch-constraint
```

Prepare an External Platform Image Registry

For new production deployments, you can use a customer-managed external image registry as the central image source for ACP Core, managed clusters, and Extensions. The registry must contain the complete target-version payload before you start the installer.

> **Prepare the Registry Before Installation**
>
> When an external registry is selected, the installer does not deploy the platform built-in Registry and does not copy missing images or packages into the external registry. Complete both upload modes on this page before running `setup.sh`.

TOC

- Understand the Registry Roles
- Registry Requirements
  - Required Capabilities and Access
  - Production Recommendations
- Prepare the Target-Version Payload
- Troubleshooting
- Next Step

Understand the Registry Roles

The following registry roles are independent:

Role	Purpose	Lifecycle owner
Platform built-in Registry	Stores platform Core images and Extension artifacts when the `global` cluster is installed with **Platform Deployment** as its image repository.	ACP deploys the service; the platform administrator operates its capacity and backup.
External platform image registry	Replaces the platform built-in Registry as the image source for Core, managed clusters, and Extensions.	You provide and operate the registry, and you upload every required target-version payload.

This page configures the platform image source used by ACP components and clusters. It does not configure an image service for application workloads.

The platform built-in Registry remains supported for existing environments and non-production evaluation. This page does not describe an online migration from the built-in Registry to an external registry.

Registry Requirements

Required Capabilities and Access

Before uploading the target-version payload, confirm that the registry:

- Supports the standard container image push and pull operations performed by the Core Package tools.
- Uses an address in the form `<host-or-IP>[:port]`, for example `registry.example.com:5000`. Do not include `http://` or `https://` in the installer registry address.
- Is reachable through the required firewalls, proxies, and network routes from the installation node and every cluster node that pulls platform images.

- Provides upload credentials that can create or update the required repository paths and push images. Credentials passed to `setup.sh` must have pull permission. The upload and pull credentials may belong to different accounts; when you authenticate `setup.sh`, provide its username and password together.
- Has enough available storage for the target-version Core and Extension payload that you upload.

Production Recommendations

For a production registry:

- Use HTTPS with a certificate that is valid for the registry address. Ensure that the installation node and every cluster node that pulls platform images trust the issuing CA.
- Use a stable DNS name that resolves from the installation node and all cluster nodes.
- Plan capacity for every required CPU architecture and for the versions retained for upgrade, rollback, and node recovery.
- Provide availability, monitoring, backup, and recovery appropriate to your environment. Registry availability is required when new Pods start, nodes are replaced, clusters are created or upgraded, and Extensions are installed or upgraded.
- Configure retention so it does not delete manifests or digests still referenced by installed platform versions, clusters, or Extensions.

> **Customer-Managed Service**
>
> ACP consumes the external registry but does not install, upgrade, back up, monitor, or repair it. Follow your registry vendor's administration documentation for those operations.

Prepare the Target-Version Payload

Use the Core Package that exactly matches the ACP Distribution Version and CPU architecture that you will install. Do not reuse a payload from another version or architecture.

1. Complete Download, extract the Core Package, and change to its `installer/` directory.

2. Copy the required Aligned Extension packages into `plugins/` as described in Required Aligned Extensions. The `all` upload mode processes the packages currently present in this directory.
3. Set the registry address and upload credentials in the current shell. Read the password without writing it into shell history:

   ```
   export REGISTRY_ADDRESS="registry.example.com"
   export REGISTRY_USERNAME="<registry-username>"
   read -rsp "Registry password: " REGISTRY_PASSWORD
   export REGISTRY_PASSWORD
   printf '\n'
   ```

   For a registry that allows anonymous push, set both `REGISTRY_USERNAME` and `REGISTRY_PASSWORD` to empty values.
4. From the extracted `installer/` directory, run both upload modes. You can run the commands in either order, but both must finish successfully:

   ```
   bash res/upload.sh all \
     "$REGISTRY_ADDRESS" "$REGISTRY_USERNAME" "$REGISTRY_PASSWORD"

   bash res/upload.sh necessary \
     "$REGISTRY_ADDRESS" "$REGISTRY_USERNAME" "$REGISTRY_PASSWORD"
   ```

   The modes overlap, but neither replaces the other:

   - `all` uploads the product image, the cluster version operator image, and every `*.tgz` package currently in `plugins/`. It does not upload the installer artifact.
   - `necessary` uploads the product image, the cluster version operator image, the installer artifact, and the bootstrap plugins required to create the `global` cluster. It does not upload every package in `plugins/`.
5. In the registry administration interface or API, confirm that the newly uploaded repositories and manifests are present and are not immediately eligible for cleanup.

Troubleshooting

Symptom	Likely cause	Action
`x509: certificate signed by unknown authority`	The registry CA or an intermediate certificate is not trusted.	Install the complete CA chain on the installation node and every cluster node that pulls images, then retry. Do not replace HTTPS with an insecure registry as a production workaround.
Certificate name mismatch	The registry address does not match the certificate subject alternative names.	Use the registry DNS name covered by the certificate or replace the certificate.
`unauthorized` or `authentication required`	The upload credentials are incorrect or lack push permission.	Verify the account used by `res/upload.sh` and confirm that it can push to the required repository paths.
`denied` or repository creation fails	The upload account cannot create or update the required repository paths.	Grant the account the required repository permissions and rerun both upload modes.
A required package is reported as not found	The Core Package is incomplete, the wrong version or architecture was extracted, or required Extension packages were not copied into `plugins/`.	Verify the package checksum, version, architecture, and `plugins/` contents before retrying.

Next Step

After both upload modes succeed and the registry contains the target payload, continue with the installation path that you selected:

- Installing on Non-Alauda OS.
- Installing the global Cluster on Alauda OS. Start the bootstrap script with `--registry` in Step 2 and set `registry.externalAddress` in Step 8.

Communication Matrix

This page lists the ports that ACP components listen on for cluster nodes that run a non-Alauda OS. Use it to build the allow rules when you enable a host firewall on those nodes.

TOC

Scope of This Matrix
When to Use This Page
What This Matrix Does Not Cover
Platform Ports
Reachability
NodePort Range
Conditional Ports
Calico

Scope of This Matrix

This matrix covers the ports used by ACP core components and by Aligned extensions. Ports required by Agnostic extensions are maintained by those extensions and are not listed here.

Some of the listed components run only on `global` cluster nodes. A workload cluster node listens on the subset of these ports that applies to the components it runs.

This Page Is Not a Complete Firewall Configuration

The matrix is a component port inventory, not a finished rule set. Several deployment options add requirements that are not expressed as a listening port on a node. See What This Matrix Does Not Cover before you apply any firewall rules.

When to Use This Page

You need this page only when a host firewall, such as `firewalld` or `ufw`, is enabled on the cluster nodes. Keeping the host firewall disabled is the default and recommended node configuration. Before you enable a host firewall, read Hardened Node Security Configuration, which gives the `init.sh` arguments, the additional ports used by the node that runs the installer, and the order in which to apply the configuration.

This page does not apply to clusters running Alauda OS, where the platform maintains the node port configuration.

This page is not the same as the port occupancy check in Node Preprocessing. That check confirms that no other process has already bound the ports the platform needs before the installation starts. This page describes which ports a host firewall must allow. Both apply when a host firewall is enabled.

Allow Cluster-Internal Traffic Before You Apply Firewall Rules

This matrix lists the ports that the nodes listen on. Pod-to-Pod and Pod-to-Service traffic uses arbitrary ports and is not covered by this matrix, so a firewall that allows only the ports listed here breaks cluster networking.

When you enable a host firewall, also treat the Pod CIDR, the Service CIDR, and the network interfaces created by the CNI plugin as trusted sources, so that cluster-internal traffic is not filtered by port. The interface names depend on the CNI plugin and on the environment, so read them from the node instead of assuming a fixed name.

What This Matrix Does Not Cover

Check each of the following against your own deployment. A requirement that applies to your cluster but is not in the matrix still has to be allowed.

- **Pod and Service traffic.** See the warning above.
- **Secondary and non-default container networks.** The `6081/UDP` entry in the table is the port used by the default Kube-OVN Geneve tunnel. If you change the Kube-OVN tunnel type, or attach a secondary network with Multus on top of the default one, those networks carry Pod traffic between node IP addresses using their own encapsulation, which is not listed here. Confirm what your configuration uses and allow it between the nodes.
- **Traffic between the `global` cluster and a workload cluster.** The `global` cluster has to reach the control plane nodes of every workload cluster it manages, and a workload cluster's own load balancer has to reach them as well. See Create a Cluster on Non-Alauda OS for the ports involved.
- **Load balancer forwarding.** When a load balancer fronts the platform access address, its forwarded traffic does not originate from a cluster node IP address. See LoadBalancer Forwarding Rules for the ports involved.
- **`Self-built VIP`.** This option runs keepalived, which uses the VRRP protocol rather than a TCP or UDP port. `firewalld` and `ufw` block VRRP by default, so the VIP never comes up unless you allow it between the nodes that share the VIP.
- **Global disaster recovery.** A DR pair needs several ports open in **both directions** between the Primary and Standby cluster networks. See the network requirements in Global Disaster Recovery.

Platform Ports

Port	Protocol	Component
`22`	TCP	`sshd`
`80`	TCP	`alb2`
`443`	TCP	`global-alb2`
`1936`	TCP	`alb2`

Port	Protocol	Component
`2049`	TCP	`nfs`
`2379`	TCP	`etcd`
`2380`	TCP	`etcd`
`6081`	UDP	`kube-ovn`
`6443`	TCP	`kube-apiserver`
`6641`	TCP	`ovsdb-server`
`6642`	TCP	`ovsdb-server`
`6643`	TCP	`ovsdb-server`
`6644`	TCP	`ovsdb-server`
`7473`	TCP	`frr-metrics`
`9000`	TCP	`minio`
`9100`	TCP	`node_exporter`
`10250`	TCP	`kubelet`
`10257`	TCP	`kube-controller-manager`
`10259`	TCP	`kube-scheduler`
`10660`	TCP	`kube-ovn-controller`
`10661`	TCP	`kube-ovn-monitor`
`10665`	TCP	`kube-ovn-daemon`
`11443`	TCP	`registry`
`11780`	TCP	`global-alb2`

Port	Protocol	Component
`11782`	TCP	`global-alb2`
`17472`	TCP	`speaker`
`17946`	TCP	`speaker`
`30000-32767`	TCP	User application `NodePort` range
`30000-32767`	UDP	User application `NodePort` range

Reachability

- Every port in the table has to be reachable between the cluster nodes and from the node that runs the installer.
- `443/tcp` additionally carries the platform access address, so it has to be reachable by the users and systems that access the platform.
- Specific deployment options require some of these ports to be reachable from outside the cluster network as well. See What This Matrix Does Not Cover.

NodePort Range

`30000-32767` is the port range reserved for `NodePort` Services created by user applications. A port in this range is opened on the nodes when an application that uses it is deployed, and it is closed again when the corresponding Service is deleted. Allow this range on the host firewall only if your applications use `NodePort` Services, and restrict it to the sources those applications are meant to serve.

Conditional Ports

Calico

The following ports are required only when the cluster uses the Calico CNI plugin.

Port	Protocol	Component
`179`	TCP	`calico-node`, BGP
`9091`	TCP	`calico-node`
`9324`	TCP	`bird_exporter`

Install on Alauda OS

The `global` cluster is the platform's control plane: it hosts the registry, the base operator, the cluster registry, and the platform's identity stack, and it orchestrates the lifecycle of every workload cluster. This section describes how to install the `global` cluster when its nodes run Alauda OS, including the optional disaster recovery deployment.

To install the `global` cluster on a non-Alauda OS such as Ubuntu or RHEL, see Install on Non-Alauda OS.

Installing

When to Use This Path
Common Prerequisites
Bootstrap Node Requirements
Compatibility and Version Inputs
Installation Lifecycle
Procedure
Optional Disaster Recovery Deployment
Verification
Decommission the Bootstrap Cluster
Next Steps
Worked Example: Complete `global` Manifest for Huawei DCS

Global Cluster Disaster Recovery

Deployment Procedure
Network Requirements
Observability Placement
Operational Scope
Failover
Bare Metal DR Model
Provider Notes
See Also

TOC

What This Section Does Not Cover

What This Section Does Not Cover

- Uploading and installing the infrastructure provider and the Cluster Bootstrap Provider on the `global` cluster before you create workload clusters: see Prepare.
- Configuring infrastructure resources for workload clusters: see Infrastructure Resources.
- Creating workload clusters: see Create on Alauda OS.
- Upgrading Kubernetes on the `global` cluster: see Upgrade Kubernetes on Alauda OS.
- Upgrading workload clusters: see Upgrade Kubernetes on Alauda OS.

Installing the global Cluster

This document describes how to install the `global` cluster on Alauda OS. The `global` cluster is the platform control plane and is provisioned through Cluster API. Use this path when the platform control plane must run on an immutable operating system such as Alauda OS.

For a `global` cluster on Huawei DCS, create the VM template from the Alauda OS image published for this ACP release. See Downloading Alauda OS.

TOC

- When to Use This Path
- Common Prerequisites
- Bootstrap Node Requirements
  - Hardware
  - Operating System
  - Network
- Compatibility and Version Inputs
- Installation Lifecycle
- Procedure
  - Step 1 — Prepare Common Variables
  - Step 2 — Create the Bootstrap Cluster
  - Step 3 — Upload and Install Provider Packages
  - Step 4 — Configure the Provider-Specific global Manifest

Step 5 — Apply the global Manifest
Step 6 — Wait for the Control Plane
Step 7 — Import Provider Resources
Step 8 — Trigger the Platform Installation
Step 9 — Monitor the Installation
Phases You Will Observe
Signals During Installation
Common Stalls and Where to Look
Optional Disaster Recovery Deployment
Prepare Shared DR Variables
Bare Metal: Prepare Shared ServiceAccount Signing Key
Add DR etcd Server Certificate SANs to KubeadmControlPlane
Add Provider-Specific DR Fields
Install Primary and Standby Clusters
Bare Metal: Verify Before Decommissioning Either Bootstrap Cluster
Bare Metal: Apply the Rule ConfigMap
Install etcd-sync (ACP 4.4.0+)
Bare Metal: Verify the Dynamic Rule Source
Start and verify synchronization
Verification
Decommission the Bootstrap Cluster
Next Steps
Worked Example: Complete `global` Manifest for Huawei DCS
Before You Apply: Prepare These on DCS
Internal Self-built VIP variant
Values to Replace

When to Use This Path

Choose this installation path when all of the following conditions apply:

- You want the `global` cluster to run on an immutable operating system. Alauda OS is the supported image today.
- Your infrastructure is one of the documented providers: Huawei DCS, VMware vSphere, Huawei Cloud Stack, or Bare Metal.
- You can run a temporary bootstrap node that has network access to the target IaaS platform.

For a non-Alauda OS such as Ubuntu or RHEL, use Install on Non-Alauda OS instead.

Common Prerequisites

The following prerequisites apply to every provider:

- A bootstrap node that meets the Bootstrap Node Requirements. A disaster recovery deployment needs two, one for the primary and one for the standby `global` cluster; see Install Primary and Standby Clusters.
- The Core Package from the Customer Portal.
- The eight required Aligned Extension packages from **Common ACP Extensions**, for the same ACP version and architecture as the Core Package. See Required Aligned Extensions.
- The Alauda Container Platform Cluster Bootstrap Provider package.
- The infrastructure provider package for your target platform.
- Network reachability between the bootstrap node and the target IaaS platform API endpoint. See Network.
- IP and hostname planning for the `global` control plane and worker nodes. See Infrastructure Resources for the resource model used by each provider.
- A stable Kubernetes API endpoint for the `global` cluster. Select and validate the provider-specific endpoint mode in Plan the Control Plane Endpoint before you create the cluster.
- A platform access address, registry address, and Pod and Service CIDR ranges. To use a customer-managed external platform image registry, prepare it first; see Prepare an External Platform Image Registry.
- For x86_64 nodes that use ACP-provided Alauda OS images, the underlying CPUs must support the `x86-64-v2` ISA baseline. See OS Support Matrix.

Naming Convention (Required)

This rule applies to **every** infrastructure provider supported by this install path — Huawei DCS, Huawei Cloud Stack, VMware vSphere, and any provider added in the future. Every manifest you author in Step 4 must follow it. Misnaming these resources has two distinct failure modes, both detailed below; one breaks initial provisioning, the other only surfaces during disaster recovery.

- The CAPI `Cluster` and the provider's infrastructure cluster resource (for example, `DCSCluster` for Huawei DCS or `HCSCluster` for Huawei Cloud Stack; each provider has its own equivalent) **must** be named exactly `global`. `cpaas-installer` looks them up by literal name, and the Huawei Cloud Stack provider only allocates the global ELB listener ports (`11443` for the registry and console, `2379` for DR etcd synchronization, `443` for web access) when the infra cluster is named `global`. A different name silently breaks registry pull, DR etcd synchronization, and the web console.
- Every other CAPI resource (`KubeadmControlPlane`, `KubeadmConfigTemplate`, `MachineDeployment`) and every other provider infrastructure resource (machine templates, IP/hostname pools, machine config pools, and any other per-provider resource) **must** use a name with the `global-` prefix. The DR (failover) mechanism uses this prefix to identify resources owned by the `global` cluster. **A `global` cluster resource without the `global-` prefix is invisible to DR and causes the standby cluster's machines to be deleted at failover time** — the cluster will provision and run normally, then lose nodes the first time DR is exercised. This is a hard requirement, not a stylistic convention.
- `Cluster.spec.controlPlaneRef.name` and any other cross-references must match the prefixed names exactly.

Bootstrap Node Requirements

The bootstrap node is a temporary machine that runs `setup.sh` from the Core Package. For the duration of the installation it carries the KIND-based `minialauda` management cluster, the embedded platform registry, the Cluster API bootstrap and infrastructure providers, and `cpaas-installer`. It never joins the `global` cluster, and it is removed after handoff — see Decommission the Bootstrap Cluster.

Size it for the image payload it serves rather than for steady-state load: while the cluster is being provisioned, every `global` node pulls the platform image set from this host.

Hardware

Resource	Minimum (x86_64)
CPU	4 cores
Memory	8 GB
Disk	300 GB or more, available to both the extraction directory (`/root/cpaas-install`) and `/var/cpaas`

The CPU and memory minimums come from standard-delivery practice with this install path. On the non-Alauda OS installation path, the machine that runs `setup.sh` becomes the first control plane node, so it must meet the control-plane minimum in Prerequisites; the bootstrap node here never joins the cluster — it extracts the Core Package, runs the temporary KIND-based bootstrap cluster, and serves the platform registry until handoff. That is a lighter, temporary load, and 4 cores / 8 GB is the figure standard deliveries run and validate against.

On arm64, apply the platform's ARM convention from the same page: at least 1.5x and preferably 2x the x86_64 figures, which gives **6 cores / 12 GB** as the minimum and **8 cores / 16 GB** as the recommended configuration.

Disk is the resource that most often stalls a bootstrap. Budget it in three parts:

Consumer	Space
Extracted Core Package	At least 100 GB after extraction. See Install on Non-Alauda OS
Embedded registry data under `/var/cpaas`	The platform image payload for the target release, plus the Aligned Extension packages you copy into `plugins/`
Provider packages and OS artifacts staged for upload	The Cluster Bootstrap Provider and infrastructure provider packages, plus one Alauda OS image or VM template artifact per node architecture

Operating System

The bootstrap node does not run Alauda OS. It needs a 64-bit Linux distribution with Bash and `root` privilege, so that `setup.sh` can bring up the bootstrap cluster described in Step 2.

The requirements in Node Preprocessing — the validated OS and kernel list, the SSH user, and the `sshd_config` settings — do **not** apply to the bootstrap node on this install path. They govern non-Alauda OS machines that the platform joins to a cluster over SSH, and the bootstrap node never joins the `global` cluster. This differs from the non-Alauda OS installation path, where the machine that runs `setup.sh` becomes the first control plane node and must therefore meet them.

When no other constraint applies, choosing a distribution from that validated list is still a safe default.

> **The x86-64-v2 baseline applies to cluster nodes, not to the bootstrap node**
>
> The `x86-64-v2` ISA baseline in OS Support Matrix applies to cluster nodes created from ACP-provided Alauda OS images. The platform does not impose that baseline on x86_64 machines running a user-provided non-Alauda OS, so it is not a requirement for the bootstrap node itself. See Node Preprocessing.
>
> If the bootstrap node is a virtual machine on the same infrastructure that will back the cluster nodes, inspecting its CPU flags can surface a missing baseline early. It is not a substitute for verifying `x86-64-v2` on the hosts that will actually run the Alauda OS nodes, because the flags a guest sees depend on the hypervisor CPU model.

Network

The bootstrap node needs a **static IP address** that does not change for the lifetime of the installation. `HOST_IP` is written into the `cpaas.io/registry-address` annotation that provisioned nodes use to pull images, and, for Bare Metal, into the registration endpoint and the SAN of its serving certificate. An address change part-way through the installation breaks node provisioning.

From	To	Port	Purpose
Bootstrap node	Target IaaS platform API	Provider-specific: DCS VRM `TCP/7443` and DCS physical host MGMT (typically `TCP/8443`); vCenter `TCP/443`; the HCS API and IAM endpoints	The Cluster API providers run inside `minialauda` and reconcile machines through the IaaS API. For DCS this path also carries the Ignition ISO upload; see Creating a Cluster on Huawei DCS
Bootstrap node	`global` control plane endpoint	`TCP/6443`	Cluster API must reach the endpoint while the control plane is created. See Plan the Control Plane Endpoint
`global` nodes being provisioned	Bootstrap node registry	`TCP/11443`	Nodes pull platform images from `NODE_REGISTRY_ADDRESS` until the `global` cluster's own registry takes over
`global` hosts, Bare Metal only	Bootstrap node	`TCP/12443`	Elemental registration endpoint used during the bootstrap phase

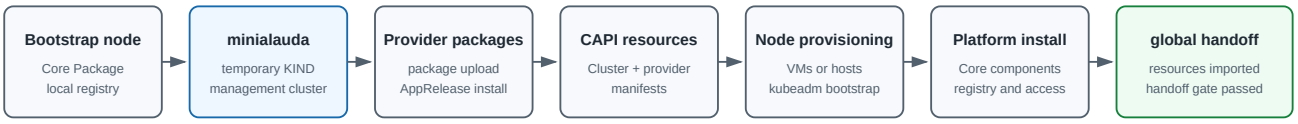
Compatibility and Version Inputs

Before installation, record the supported version set for the delivery package:

Input	Purpose
Core Package version	Provides the installer, local registry, and base platform payload.
Cluster Bootstrap Provider chart version	Must match the Cluster API control plane resources used by the global manifest.
Infrastructure provider chart version	Use the VMware vSphere, DCS, HCS, or Bare Metal provider chart version delivered with the target release.

Input	Purpose
Alauda OS image or VM template	Must contain the Kubernetes version used by `K8S_VERSION`.
`K8S_VERSION`	Use v-prefixed semver that matches the target Alauda OS image, such as `v<major>.<minor>.<patch>`.

Installation Lifecycle



Use the following stage map to identify the input, expected result, and first diagnostic location for each part of the installation.

Stage	Input	Apply	Expected result	Diagnose fi
Bootstrap environment	Core Package and bootstrap node	Run `setup.sh`	The `minialauda` management cluster, local registry, and installer Pods are running.	Bootstrap nc containers a `cpaas-syst`
Provider delivery	Cluster Bootstrap Provider and infrastructure provider packages	Push packages and apply provider `AppRelease` objects.	Provider CRDs and controllers are Ready.	`AppRelease` provider Poc registry acce

Stage	Input	Apply	Expected result	Diagnose fi
Cluster definition	Provider credentials, endpoint, network, node, and image values	Apply the complete Cluster API manifest.	`Cluster`, infrastructure cluster, control plane, and machine resources appear.	Resource co provider-con
Node provisioning	VM template or registered hosts	Let the infrastructure provider reconcile the Machines.	VMs or physical hosts boot and receive kubeadm bootstrap data.	Provider Ma objects, IaaS guest initializ
Control plane	Kubernetes and component versions	Wait for `KubeadmControlPlane`.	The expected control-plane replicas are Ready and the API endpoint responds.	`KubeadmCon` Machines, e endpoint hea
Platform installation	Installer request and import resources	Trigger the installer.	ACP Core components, registry, platform access, and base modules become healthy.	Installer prog and Pods in `system`.
Handoff	Imported provider resources and credentials	Verify the final `global` cluster and every provider-specific handoff gate, then retire only `minialauda`.	The final cluster holds the imported provider resources and credentials and no longer depends on the temporary	Imported res provider cred cluster anno provider-spe Jobs and Co

Stage	Input	Apply	Expected result	Diagnose fi
			management cluster. For Bare Metal, the system-agent handoff ConfigMap matches the current endpoint, authentication profile, and complete target set.	

Procedure

Step 1 — Prepare Common Variables

Set the common variables on the bootstrap node.

```
export HOST_IP="<bootstrap-host-ip>"
export LOCAL_REGISTRY_ADDRESS="127.0.0.1:11443"
export BOOTSTRAP_REGISTRY_ADDRESS="172.18.0.1:11443"
export NODE_REGISTRY_ADDRESS="${HOST_IP}:11443"
export CONTROL_PLANE_VIP="<global-control-plane-vip>"
export PLATFORM_HOST="<platform-access-domain-or-vip>"
export REGISTRY_DOMAIN="<platform-registry-domain-or-vip>:11443"
export CLUSTER_CIDR="100.3.0.0/16"
export SERVICE_CIDR="100.4.0.0/16"
export KUBE_OVN_JOIN_CIDR="<kube-ovn-join-cidr>"
export K8S_VERSION="<target-kubernetes-version>"
export INGRESS_CLASS_NAME="global-alb2"
export PROVIDER_SECRET_NAME="global-secret"
# Use v-prefixed semver that matches the target Alauda OS image.
```

Use `LOCAL_REGISTRY_ADDRESS` when pushing packages from the bootstrap node. Use `BOOTSTRAP_REGISTRY_ADDRESS` in AppRelease chart repository values because provider Pods read the chart repository from inside the bootstrap cluster's network. Use `NODE_REGISTRY_ADDRESS` (the bootstrap node's registry, `<bootstrap-host-ip>:11443`) in the Cluster API registry annotations, because provisioned `global` nodes must pull images through an address reachable from their subnet during provisioning. This is a temporary value: after the `global` cluster's own registry comes up, the installer automatically rewrites the `cpaas.io/registry-address` annotation on the `Cluster` and `DCSCluster` to the permanent platform registry, so later reconciles pull from the `global` cluster instead of the bootstrap node.

Keep the Registry formats distinct:

Use	Format	Example
Registry HTTP API request	URL with `/v2/` path	`http://${LOCAL_REGISTRY_ADDRESS}/v2/acp/chart-cluster-api-provider-kubeadm/tags/list`
Container image reference	Registry, repository, and tag or digest	`${BOOTSTRAP_REGISTRY_ADDRESS}/acp/kubeadm-bootstrap-controller:<tag>`
`cpaas.io/registry-address` annotation	`<host>:<port>` only	`${NODE_REGISTRY_ADDRESS}`
AppRelease `repoURL`	Registry address without `/v2/`	`${BOOTSTRAP_REGISTRY_ADDRESS}`

The `/v2/` segment belongs only to Registry HTTP API requests. Do not add it to image references, the `cpaas.io/registry-address` annotation, or AppRelease `repoURL`.

Step 2 — Create the Bootstrap Cluster

Run the bootstrap script provided by the Core Package with Bash. This brings up a temporary KIND-based bootstrap cluster named `minialauda` on the bootstrap node — the temporary Cluster API management cluster used only to provision the `global` cluster. After it

completes, make the matching `kubectl` client from the bootstrap control-plane container available on the bootstrap node and configure the exported kubeconfig.

```
mkdir -p /root/cpaas-install
tar -xvf <core-package> -C /root/cpaas-install
cd /root/cpaas-install/installer
```

The Core Package does not contain the required Aligned Extension packages. Before you run `setup.sh`, copy each of the eight packages listed in Required Aligned Extensions into the `plugins/` directory of the extracted Core Package. Without them, the required Aligned Extensions cannot be installed. Run the following command once for each package:

```
cp <path-to-extension-package> /root/cpaas-install/installer/plugins/
```

The downloaded files are named `<package-name>.<arch>.<version>.tgz`. Keep the file names unchanged. The eight package names are `console`, `console-central`, `console-marketplace`, `application-catalog`, `aiops-base`, `container-platform`, `acp-business`, and `gatewayapi-plugin`. Keep the files that are already in `plugins/`, and confirm that the eight packages are present:

```
ls -1 /root/cpaas-install/installer/plugins/
```

Do not copy other packages from **Common ACP Extensions** into this directory. The Cluster Bootstrap Provider and infrastructure provider packages are not copied here either; Step 3 uploads them to the bootstrap Registry.

Then run the bootstrap script. To use an external platform image registry, run `bash setup.sh --registry "<registry-address>"` in place of `bash setup.sh`. The registry address uses the form `<host-or-IP>[:port]` without `http://` or `https://`. If the registry requires authentication, also provide the pull `--username` and `--password` together; see Start the Installer for how to enter them without exposing the password. Prepare the registry before this step, as described in Prepare an External Platform Image Registry.

```
bash setup.sh
mkdir -p "${HOME}/.local/bin" ~/.kube

if ! command -v kubectl >/dev/null 2>&1; then
  nerdctl cp minialauda-control-plane:/usr/bin/kubectl \
    "${HOME}/.local/bin/kubectl"
  chmod 0755 "${HOME}/.local/bin/kubectl"
  export PATH="${HOME}/.local/bin:${PATH}"
fi

cp /var/cpaas/data/alauda.kubeconfig ~/.kube/config
kubectl get nodes
```

The bootstrap script provisions an embedded registry, the Cluster API control plane, and the installer components that drive the `global` cluster installation.

By default, the embedded bootstrap Registry is anonymous and no `global-registry-auth` Secret is created. If you configured both a Registry username and password during bootstrap setup, `setup.sh` creates that Secret in `cpaas-system`.

Step 3 — Upload and Install Provider Packages

Upload the Cluster Bootstrap Provider package and the infrastructure provider package to the local registry.

The AppRelease examples below use the default anonymous bootstrap Registry and therefore do not reference `global-registry-auth`.

If the bootstrap Registry requires authentication, verify that the Secret exists before you create any provider AppRelease:

```
kubectl -n cpaas-system get secret global-registry-auth -o name
```

Then add both authentication fields to each provider AppRelease before applying it:

```
spec:
  source:
    chartPullSecret: global-registry-auth
  values:
    global:
      registry:
        imagePullSecrets:
          - global-registry-auth
```

If the Registry is anonymous, do not create a placeholder Secret and do not add these fields. The bootstrap Secret is also not a credential-transfer mechanism: by default, workload clusters receive their Registry pull Secret from the `global` cluster's `public-registry-credential` flow, and a workload cluster can instead be bound to a dedicated registry as described in Choose the Image Registry for a Workload Cluster. Do not copy the bootstrap `global-registry-auth` Secret into a workload cluster.

Why cluster.type is Baremetal for every provider

The AppRelease values in the tabs below all set `global.cluster.type: Baremetal`. This is a chart-internal classifier, not the IaaS provider name. Keep `Baremetal` for the Huawei DCS, VMware vSphere, Huawei Cloud Stack, and Bare Metal `global` installations. The value drives how the platform configures node-level components; it does not select the infrastructure provider.

Huawei DCS

Set the provider package paths and chart versions.

```
export DCS_PROVIDER_PACK="/root/cluster-api-provider-dcs.amd64.<version>.tgz"
export KUBEADM_PROVIDER_PACK="/root/cluster-api-provider-kubeadm.amd64.<version>.tgz"
export DCS_PROVIDER_VERSION="<dcs-provider-chart-version>"
export KUBEADM_PROVIDER_VERSION="<kubeadm-provider-chart-version>"
```

Upload the packages.

```
/root/cpaas-install/installer/res/amd64/packtool pack push \
  -r "${LOCAL_REGISTRY_ADDRESS}" -c "${DCS_PROVIDER_PACK}"

/root/cpaas-install/installer/res/amd64/packtool pack push \
  -r "${LOCAL_REGISTRY_ADDRESS}" -c "${KUBEADM_PROVIDER_PACK}"
```

Create and apply the AppRelease resources for the Cluster Bootstrap Provider and the DCS provider.

```
mkdir -p /root/yamls
export DCS_PROVIDER_APPRELEASES="/root/yamls/dcs-provider-apprelease
s.yaml"

cat > "${DCS_PROVIDER_APPRELEASES}" <<EOF
---
apiVersion: operator.alauda.io/v1alpha1
kind: AppRelease
metadata:
  annotations:
    auto-recycle: "true"
    interval-sync: "true"
  name: cluster-api-provider-kubeadm
  namespace: cpaas-system
spec:
  destination:
    cluster: ""
    namespace: ""
  source:
    charts:
      - name: acp/chart-cluster-api-provider-kubeadm
        releaseName: cluster-api-provider-kubeadm
        targetRevision: ${KUBEADM_PROVIDER_VERSION}
    repoURL: ${BOOTSTRAP_REGISTRY_ADDRESS}
  timeout: 120
  values:
    global:
      albName: ${INGRESS_CLASS_NAME}
      auth:
        default_admin: admin@cpaas.io
      cluster:
        isGlobal: true
        name: global
        networkType: kube-ovn
        type: Baremetal
      host: ${PLATFORM_HOST}
      ingress:
        ingressClassName: ${INGRESS_CLASS_NAME}
      labelBaseDomain: cpaas.io
      namespace: cpaas-system
      platformUrl: https://${PLATFORM_HOST}
      protectSecretFiles:
        enabled: false
```

```
      region: global
      registry:
        address: ${BOOTSTRAP_REGISTRY_ADDRESS}
      replicas: 1
      scheme: https
---
apiVersion: operator.alauda.io/v1alpha1
kind: AppRelease
metadata:
  annotations:
    auto-recycle: "true"
    interval-sync: "true"
  name: cluster-api-provider-dcs
  namespace: cpaas-system
spec:
  destination:
    cluster: ""
    namespace: ""
  source:
    charts:
      - name: acp/chart-cluster-api-provider-dcs
        releaseName: cluster-api-provider-dcs
        targetRevision: ${DCS_PROVIDER_VERSION}
    repoURL: ${BOOTSTRAP_REGISTRY_ADDRESS}
  timeout: 120
  values:
    global:
      albName: ${INGRESS_CLASS_NAME}
      auth:
        default_admin: admin@cpaas.io
      cluster:
        isGlobal: true
        name: global
        networkType: kube-ovn
        type: Baremetal
      host: ${PLATFORM_HOST}
      ingress:
        ingressClassName: ${INGRESS_CLASS_NAME}
      labelBaseDomain: cpaas.io
      namespace: cpaas-system
      platformUrl: https://${PLATFORM_HOST}
      protectSecretFiles:
        enabled: false
      region: global
```

```
      registry:
        address: ${BOOTSTRAP_REGISTRY_ADDRESS}
      replicas: 1
      scheme: https
EOF

kubectl apply -f "${DCS_PROVIDER_APPRELEASES}"

until kubectl get crd kubeadmcontrolplanes.controlplane.cluster.x-k8
s.io --ignore-not-found 2>/dev/null | grep -q kubeadmcontrolplanes.co
ntrolplane.cluster.x-k8s.io; do
  sleep 10
done

until kubectl get crd dcsclusters.infrastructure.cluster.x-k8s.io --i
gnore-not-found 2>/dev/null | grep -q dcsclusters.infrastructure.clus
ter.x-k8s.io; do
  sleep 10
done
```

VMware vSphere

Set the provider package paths and chart versions.

```
export VSPHERE_PROVIDER_PACK="/root/cluster-api-provider-vsphere.amd6
4.<version>.tgz"
export KUBEADM_PROVIDER_PACK="/root/cluster-api-provider-kubeadm.amd6
4.<version>.tgz"
export VSPHERE_PROVIDER_VERSION="<vsphere-provider-chart-version>"
export KUBEADM_PROVIDER_VERSION="<kubeadm-provider-chart-version>"
```

Upload the packages.

```
/root/cpaas-install/installer/res/amd64/packtool pack push \
  -r "${LOCAL_REGISTRY_ADDRESS}" -c "${VSPHERE_PROVIDER_PACK}"

/root/cpaas-install/installer/res/amd64/packtool pack push \
  -r "${LOCAL_REGISTRY_ADDRESS}" -c "${KUBEADM_PROVIDER_PACK}"
```

Create and apply the AppRelease resources for the Cluster Bootstrap Provider and the VMware vSphere provider.

```
mkdir -p /root/yamls
export VSPHERE_PROVIDER_APPRELEASES="/root/yamls/vsphere-provider-app
releases.yaml"

cat > "${VSPHERE_PROVIDER_APPRELEASES}" <<EOF
---
apiVersion: operator.alauda.io/v1alpha1
kind: AppRelease
metadata:
  annotations:
    auto-recycle: "true"
    interval-sync: "true"
  name: cluster-api-provider-kubeadm
  namespace: cpaas-system
spec:
  destination:
    cluster: ""
    namespace: ""
  source:
    charts:
      - name: acp/chart-cluster-api-provider-kubeadm
        releaseName: cluster-api-provider-kubeadm
        targetRevision: ${KUBEADM_PROVIDER_VERSION}
    repoURL: ${BOOTSTRAP_REGISTRY_ADDRESS}
  timeout: 120
  values:
    global:
      albName: ${INGRESS_CLASS_NAME}
      auth:
        default_admin: admin@cpaas.io
      cluster:
        isGlobal: true
        name: global
        networkType: kube-ovn
        type: Baremetal
      host: ${PLATFORM_HOST}
      ingress:
        ingressClassName: ${INGRESS_CLASS_NAME}
      labelBaseDomain: cpaas.io
      namespace: cpaas-system
      platformUrl: https://${PLATFORM_HOST}
      protectSecretFiles:
        enabled: false
```

```
      region: global
      registry:
        address: ${BOOTSTRAP_REGISTRY_ADDRESS}
      replicas: 1
      scheme: https
---
apiVersion: operator.alauda.io/v1alpha1
kind: AppRelease
metadata:
  annotations:
    auto-recycle: "true"
    interval-sync: "true"
  name: cluster-api-provider-vsphere
  namespace: cpaas-system
spec:
  destination:
    cluster: ""
    namespace: ""
  source:
    charts:
      - name: acp/chart-cluster-api-provider-vsphere
        releaseName: cluster-api-provider-vsphere
        targetRevision: ${VSPHERE_PROVIDER_VERSION}
    repoURL: ${BOOTSTRAP_REGISTRY_ADDRESS}
  timeout: 120
  values:
    global:
      albName: ${INGRESS_CLASS_NAME}
      auth:
        default_admin: admin@cpaas.io
      cluster:
        isGlobal: true
        name: global
        networkType: kube-ovn
        type: Baremetal
      host: ${PLATFORM_HOST}
      ingress:
        ingressClassName: ${INGRESS_CLASS_NAME}
      labelBaseDomain: cpaas.io
      namespace: cpaas-system
      platformUrl: https://${PLATFORM_HOST}
      protectSecretFiles:
        enabled: false
      region: global
```

```
      registry:
        address: ${BOOTSTRAP_REGISTRY_ADDRESS}
      replicas: 1
      scheme: https
EOF

kubectl apply -f "${VSPHERE_PROVIDER_APPRELEASES}"

until kubectl get crd kubeadmcontrolplanes.controlplane.cluster.x-k8
s.io --ignore-not-found 2>/dev/null | grep -q kubeadmcontrolplanes.co
ntrolplane.cluster.x-k8s.io; do
  sleep 10
done

until kubectl get crd vsphereclusters.infrastructure.cluster.x-k8s.io
--ignore-not-found 2>/dev/null | grep -q vsphereclusters.infrastructu
re.cluster.x-k8s.io; do
  sleep 10
done
```

Huawei Cloud Stack

Set the provider package paths and chart versions.

```
export HCS_PROVIDER_PACK="/root/cluster-api-provider-hcs.amd64.<versi
on>.tgz"
export KUBEADM_PROVIDER_PACK="/root/cluster-api-provider-kubeadm.amd6
4.<version>.tgz"
export HCS_PROVIDER_VERSION="<hcs-provider-chart-version>"
export KUBEADM_PROVIDER_VERSION="<kubeadm-provider-chart-version>"
```

Upload the packages.

```
/root/cpaas-install/installer/res/amd64/packtool pack push \
  -r "${LOCAL_REGISTRY_ADDRESS}" -c "${HCS_PROVIDER_PACK}"

/root/cpaas-install/installer/res/amd64/packtool pack push \
  -r "${LOCAL_REGISTRY_ADDRESS}" -c "${KUBEADM_PROVIDER_PACK}"
```

Create and apply the AppRelease resources for the Cluster Bootstrap Provider and the HCS provider.

```
mkdir -p /root/yamls
export HCS_PROVIDER_APPRELEASES="/root/yamls/hcs-provider-apprelease
s.yaml"

cat > "${HCS_PROVIDER_APPRELEASES}" <<EOF
---
apiVersion: operator.alauda.io/v1alpha1
kind: AppRelease
metadata:
  annotations:
    auto-recycle: "true"
    interval-sync: "true"
  name: cluster-api-provider-kubeadm
  namespace: cpaas-system
spec:
  destination:
    cluster: ""
    namespace: ""
  source:
    charts:
      - name: acp/chart-cluster-api-provider-kubeadm
        releaseName: cluster-api-provider-kubeadm
        targetRevision: ${KUBEADM_PROVIDER_VERSION}
    repoURL: ${BOOTSTRAP_REGISTRY_ADDRESS}
  timeout: 120
  values:
    global:
      albName: ${INGRESS_CLASS_NAME}
      auth:
        default_admin: admin@cpaas.io
      cluster:
        isGlobal: true
        name: global
        networkType: kube-ovn
        type: Baremetal
      host: ${PLATFORM_HOST}
      ingress:
        ingressClassName: ${INGRESS_CLASS_NAME}
      labelBaseDomain: cpaas.io
      namespace: cpaas-system
      platformUrl: https://${PLATFORM_HOST}
      protectSecretFiles:
        enabled: false
```

```
      region: global
      registry:
        address: ${BOOTSTRAP_REGISTRY_ADDRESS}
      replicas: 1
      scheme: https
---
apiVersion: operator.alauda.io/v1alpha1
kind: AppRelease
metadata:
  annotations:
    auto-recycle: "true"
    interval-sync: "true"
  name: cluster-api-provider-hcs
  namespace: cpaas-system
spec:
  destination:
    cluster: ""
    namespace: ""
  source:
    charts:
      - name: acp/chart-cluster-api-provider-hcs
        releaseName: cluster-api-provider-hcs
        targetRevision: ${HCS_PROVIDER_VERSION}
    repoURL: ${BOOTSTRAP_REGISTRY_ADDRESS}
  timeout: 120
  values:
    global:
      albName: ${INGRESS_CLASS_NAME}
      auth:
        default_admin: admin@cpaas.io
      cluster:
        isGlobal: true
        name: global
        networkType: kube-ovn
        type: Baremetal
      host: ${PLATFORM_HOST}
      ingress:
        ingressClassName: ${INGRESS_CLASS_NAME}
      labelBaseDomain: cpaas.io
      namespace: cpaas-system
      platformUrl: https://${PLATFORM_HOST}
      protectSecretFiles:
        enabled: false
      region: global
```

```
      registry:
        address: ${BOOTSTRAP_REGISTRY_ADDRESS}
      replicas: 1
      scheme: https
EOF

kubectl apply -f "${HCS_PROVIDER_APPRELEASES}"

until kubectl get crd kubeadmcontrolplanes.controlplane.cluster.x-k8
s.io --ignore-not-found 2>/dev/null | grep -q kubeadmcontrolplanes.co
ntrolplane.cluster.x-k8s.io; do
  sleep 10
done

until kubectl get crd hcsclusters.infrastructure.cluster.x-k8s.io --i
gnore-not-found 2>/dev/null | grep -q hcsclusters.infrastructure.clus
ter.x-k8s.io; do
  sleep 10
done
```

Bare Metal

Set the provider package paths and chart versions.

```
export BAREMETAL_PROVIDER_PACK="/root/cluster-api-provider-baremetal.
amd64.<version>.tgz"
export KUBEADM_PROVIDER_PACK="/root/cluster-api-provider-kubeadm.amd6
4.<version>.tgz"
export BAREMETAL_PROVIDER_VERSION="<baremetal-provider-chart-version
>"
export KUBEADM_PROVIDER_VERSION="<kubeadm-provider-chart-version>"
```

Upload the packages.

```
/root/cpaas-install/installer/res/amd64/packtool pack push \
  -r "${LOCAL_REGISTRY_ADDRESS}" -c "${BAREMETAL_PROVIDER_PACK}"

/root/cpaas-install/installer/res/amd64/packtool pack push \
  -r "${LOCAL_REGISTRY_ADDRESS}" -c "${KUBEADM_PROVIDER_PACK}"
```

Import the Bare Metal Alauda OS images into the same local registry. The provider package does not contain them. The Bare Metal OS download is an archive of one OCI image layout that holds two images with the same tag, and both are required — see Image Format by Provider:

Image	Used by
`baremetal-base-image`	`elemental-image-catalog`, when a host is reprovisioned into a `global` node.
`baremetal-base-image-iso`	`SeedImage.spec.baseImage`, when the installation ISO for the `global` hosts is built.

Import the archive into a dedicated containerd namespace on the bootstrap node with `ctr`, and list the image names it creates. `/var/lib/containerd` needs free space of at least the archive size:

```
export OS_IMAGE_ARCHIVE="/root/<bare-metal-os-archive>.tar"

ctr -n bare-metal-os images import --all-platforms --no-unpack "${OS_IMAGE_ARCHIVE}"
ctr -n bare-metal-os images ls -q
```

The output lists both images with the same tag:

```
baremetal-base-image-iso:<os-image-tag>
baremetal-base-image:<os-image-tag>
```

Set `OS_IMAGE_TAG` to that tag, then push both images and confirm that the registry lists the tag for each of them:

```
export OS_IMAGE_TAG="<os-image-tag>"

for repo in baremetal-base-image baremetal-base-image-iso; do
  ctr -n bare-metal-os images push --plain-http \
    "${LOCAL_REGISTRY_ADDRESS}/tkestack/${repo}:${OS_IMAGE_TAG}" "${r
epo}:${OS_IMAGE_TAG}"
  curl -s "http://${LOCAL_REGISTRY_ADDRESS}/v2/tkestack/${repo}/tags/
list"
done
```

If the bootstrap Registry requires authentication, pass its credentials to both commands with `-u <username>:<password>`. Keep the images in the `bare-metal-os` namespace: Upload the base image to the global Registry pushes them again after installation.

Create and apply the AppRelease resources for the Cluster Bootstrap Provider and the Bare Metal provider.

The Bare Metal provider AppRelease also fills the image catalog. The chart creates the `elemental-image-catalog` ConfigMap with no entries, so `provider.imageCatalog.images` must map `${K8S_VERSION}` to the `baremetal-base-image` imported above; the chart prefixes the repository with `global.registry.address`. Without that entry, every `global` `BaremetalMachine` enters `Failed` with `ImageResolved=False / Reason=ImageCatalogMiss` and no node is provisioned.

Bare Metal Bootstrap Endpoint

During bootstrap, the `global` cluster has not handed control to the final VIP yet. Keep the bare-metal registration path on the bootstrap node: `global.platformUrl` points to the bootstrap node, and `elemental.server.url` points to `https://<bootstrap-host-ip>:12443`. Do not set `baremetal.cluster.io/system-agent-server-url` on the `MachineRegistration` used for the `global` machines during this phase. For Bare Metal DR, add `baremetal.cluster.io/system-agent-auth-scope: global` to that bootstrap `MachineRegistration`; this is the only DR-specific system-agent annotation required before

handoff. The DR handoff job later moves those machines to the local direct API endpoint at `https://<CONTROL_PLANE_VIP>:6443`.

Prepare the bootstrap HTTPS certificate used by `global-alb2` before you install the Bare Metal provider. The `elemental-operator` mounts the CA from `cpaas-system/dex.tls`, and `elemental-system-agent` uses that CA when `elemental.tls.agentTLSMode` is `strict`. The Secret must contain `tls.crt`, `tls.key`, and the CA bundle key that you configure in the AppRelease, shown below as `ca.crt`. The serving certificate must include `${HOST_IP}` as an IP SAN because the bootstrap endpoint is `https://${HOST_IP}:12443`.

If cert-manager is available in the bootstrap cluster, create or refresh `dex.tls` with a bootstrap-local CA:

```
kubectl get crd certificates.cert-manager.io issuers.cert-manager.io

kubectl -n cpaas-system apply -f - <<EOF
apiVersion: cert-manager.io/v1
kind: Issuer
metadata:
  name: baremetal-bootstrap-selfsigned
  namespace: cpaas-system
spec:
  selfSigned: {}
---
apiVersion: cert-manager.io/v1
kind: Certificate
metadata:
  name: baremetal-bootstrap-ca
  namespace: cpaas-system
spec:
  secretName: baremetal-bootstrap-ca
  commonName: baremetal-bootstrap-ca
  duration: 87600h
  renewBefore: 720h
  isCA: true
  privateKey:
    algorithm: RSA
    size: 2048
  usages:
    - cert sign
    - crl sign
  issuerRef:
    name: baremetal-bootstrap-selfsigned
    kind: Issuer
---
apiVersion: cert-manager.io/v1
kind: Issuer
metadata:
  name: baremetal-bootstrap-ca
  namespace: cpaas-system
spec:
  ca:
    secretName: baremetal-bootstrap-ca
---
apiVersion: cert-manager.io/v1
kind: Certificate
```

```
  metadata:
    name: dex-tls-bootstrap
    namespace: cpaas-system
  spec:
    secretName: dex.tls
    commonName: ${HOST_IP}
    duration: 87600h
    renewBefore: 720h
    privateKey:
      algorithm: RSA
      size: 2048
    usages:
      - digital signature
      - key encipherment
      - server auth
    ipAddresses:
      - "${HOST_IP}"
      - "127.0.0.1"
    dnsNames:
      - global-alb2
      - global-alb2.cpaas-system.svc
      - global-alb2.cpaas-system.svc.cluster.local
    issuerRef:
      name: baremetal-bootstrap-ca
      kind: Issuer
  EOF

  kubectl -n cpaas-system wait certificate/baremetal-bootstrap-ca \
    --for=condition=Ready \
    --timeout=120s
  kubectl -n cpaas-system wait certificate/dex-tls-bootstrap \
    --for=condition=Ready \
    --timeout=120s
```

Verify that the Secret has the expected keys and that the certificate is valid for the bootstrap endpoint:

```
kubectl -n cpaas-system get secret dex.tls \
  -o jsonpath='{.data.tls\.crt}{" "}{.data.tls\.key}{" "}{.data.ca\.c
rt}{"\n"}'

tmp_dir=$(mktemp -d)
trap 'rm -rf "${tmp_dir}"' EXIT
kubectl -n cpaas-system get secret dex.tls \
  -o jsonpath='{.data.ca\.crt}' | base64 -d > "${tmp_dir}/ca.crt"
kubectl -n cpaas-system get secret dex.tls \
  -o jsonpath='{.data.tls\.crt}' | base64 -d > "${tmp_dir}/tls.crt"

openssl verify -CAfile "${tmp_dir}/ca.crt" "${tmp_dir}/tls.crt"
openssl x509 -in "${tmp_dir}/tls.crt" -noout -text | grep "IP Addres
s:${HOST_IP}"

echo | openssl s_client \
  -connect "${HOST_IP}:12443" \
  -servername "${HOST_IP}" \
  -CAfile "${tmp_dir}/ca.crt" \
  -verify_return_error 2>&1 | grep "Verify return code: 0 (ok)"
```

If the bootstrap ALB keeps serving an older certificate, restart it and verify again:

```
kubectl -n cpaas-system rollout restart deploy/global-alb2
kubectl -n cpaas-system rollout status deploy/global-alb2
```

Do not include this bootstrap `dex.tls` in `dcs-import-extra-resources`. It is only for the temporary bootstrap endpoint. The final `global` cluster's `dex.tls` is created or maintained by the installer platform certificate flow. For DR, use the `thirdParty` console certificate guidance in Step 8 so both sides serve a certificate for the stable platform domain. A control-plane VIP needs to be in that platform certificate only when clients intentionally access platform HTTPS directly through that VIP. The Global registry on port `11443` presents its own `global-registry-server` certificate chain and is not configured by `console.cert` or `dex.tls`.

```
mkdir -p /root/yamls
export BAREMETAL_PROVIDER_APPRELEASES="/root/yamls/baremetal-provider
-appreleases.yaml"

cat > "${BAREMETAL_PROVIDER_APPRELEASES}" <<EOF
---
apiVersion: operator.alauda.io/v1alpha1
kind: AppRelease
metadata:
  annotations:
    auto-recycle: "true"
    interval-sync: "true"
  name: cluster-api-provider-kubeadm
  namespace: cpaas-system
spec:
  destination:
    cluster: ""
    namespace: ""
  source:
    charts:
      - name: acp/chart-cluster-api-provider-kubeadm
        releaseName: cluster-api-provider-kubeadm
        targetRevision: ${KUBEADM_PROVIDER_VERSION}
    repoURL: ${BOOTSTRAP_REGISTRY_ADDRESS}
  timeout: 120
  values:
    global:
      albName: ${INGRESS_CLASS_NAME}
      auth:
        default_admin: admin@cpaas.io
      cluster:
        isGlobal: true
        name: global
        networkType: kube-ovn
        type: Baremetal
      host: ${PLATFORM_HOST}
      ingress:
        ingressClassName: ${INGRESS_CLASS_NAME}
      labelBaseDomain: cpaas.io
      namespace: cpaas-system
      platformUrl: https://${PLATFORM_HOST}
      protectSecretFiles:
        enabled: false
```

```
      region: global
      registry:
        address: ${BOOTSTRAP_REGISTRY_ADDRESS}
      replicas: 1
      scheme: https
---
apiVersion: operator.alauda.io/v1alpha1
kind: AppRelease
metadata:
  annotations:
    auto-recycle: "true"
    interval-sync: "true"
  name: cluster-api-provider-baremetal
  namespace: cpaas-system
spec:
  destination:
    cluster: ""
    namespace: ""
  source:
    charts:
      - name: acp/chart-cluster-api-provider-baremetal
        releaseName: cluster-api-provider-baremetal
        targetRevision: ${BAREMETAL_PROVIDER_VERSION}
    repoURL: ${BOOTSTRAP_REGISTRY_ADDRESS}
  timeout: 120
  values:
    global:
      albName: ${INGRESS_CLASS_NAME}
      auth:
        default_admin: admin@cpaas.io
      cluster:
        isGlobal: true
        name: global
        networkType: kube-ovn
        type: Baremetal
      host: ${PLATFORM_HOST}
      ingress:
        ingressClassName: ${INGRESS_CLASS_NAME}
        tls:
          secretName: dex.tls
      labelBaseDomain: cpaas.io
      namespace: cpaas-system
      platformUrl: https://${HOST_IP}
      protectSecretFiles:
```

```
        enabled: false
      region: global
      registry:
        address: ${BOOTSTRAP_REGISTRY_ADDRESS}
      replicas: 1
      scheme: https
    provider:
      imageCatalog:
        images:
          ${K8S_VERSION}:
            repository: tkestack/baremetal-base-image
            tag: ${OS_IMAGE_TAG}
    handoffHook:
      # Normal non-DR bootstrap baseline. Bare Metal DR must patch th
is to true
      # before calling the installer API, as shown in Optional Disast
er Recovery Deployment.
      directAPIServer: false
      controlPlaneVIP: ${CONTROL_PLANE_VIP}
      delivery:
        enabled: true
        mode: always
    elemental:
      server:
        url: https://${HOST_IP}:12443
      systemAgent:
        authMode: shared
        # Normal non-DR bootstrap baseline. Bare Metal DR must enable
the split
        # and set sharedAuthReadOnly according to this side's active/
standby role.
        splitAuthEnabled: false
        serviceAccountName: baremetal-system-agent
        globalServiceAccountName: baremetal-global-system-agent
        sharedAuthReadOnly: false
      tls:
        agentTLSMode: strict
        caCertSecretName: dex.tls
        caCertSecretKey: ca.crt
EOF

kubectl apply -f "${BAREMETAL_PROVIDER_APPRELEASES}"

until kubectl get crd kubeadmcontrolplanes.controlplane.cluster.x-k8
```

```
s.io --ignore-not-found 2>/dev/null | grep -q kubeadmcontrolplanes.co
ntrolplane.cluster.x-k8s.io; do
  sleep 10
done

until kubectl get crd baremetalclusters.infrastructure.cluster.x-k8s.
io --ignore-not-found 2>/dev/null | grep -q baremetalclusters.infrast
ructure.cluster.x-k8s.io; do
  sleep 10
done

until kubectl get crd machineinventories.elemental.cattle.io --ignore
-not-found 2>/dev/null | grep -q machineinventories.elemental.cattle.
io; do
  sleep 10
done
```

The complete AppRelease above intentionally shows the normal, non-DR baseline. For a Bare Metal DR installation, do not call the installer API with those three baseline values unchanged. Apply the role-specific DR patch in Optional Disaster Recovery Deployment first, and verify the effective AppRelease values on that bootstrap cluster.

After the provider starts, verify that the chart values were accepted. A CrashLoopBackOff with an unknown flag such as `--system-agent-auth-mode` means the AppRelease chart and the elemental-operator image do not match; install a chart and image from the same release payload before continuing.

```
kubectl -n cpaas-system get pods | grep -E 'cluster-api-provider-bare
metal|elemental'
kubectl -n cpaas-system logs deploy/elemental-operator --tail=100
kubectl -n cpaas-system get configmap elemental-image-catalog -o json
path='{.data}'
```

The catalog output must contain `${K8S_VERSION}` as a key. Empty output or `{}` means the `provider.imageCatalog.images` value did not reach the release; fix the AppRelease before you apply the `global` manifest.

Step 4 — Configure the Provider-Specific global Manifest

Create one provider-specific manifest for the `global` cluster. The manifest uses the same provider resources as a workload cluster, but it must also include the global-specific labels, annotations, registry values, installer-compatible kubeadm settings, and persistent data paths required by the platform control plane.

Use the provider creation guides as the detailed resource reference:

- Huawei DCS: Create a Cluster on Huawei DCS
- VMware vSphere: Infrastructure Preparation and Create a Cluster on VMware vSphere
- Huawei Cloud Stack: Create a Cluster on Huawei Cloud Stack

Apply the naming convention from Common Prerequisites to every resource in the manifest you author below.

Set `KubeadmControlPlane.spec.kubeadmConfigSpec.format` to the value that the target provider accepts. This is an API field: enter `cloud-config`, not `cloud-init`. For the providers below, `cloud-init` is the guest operating-system software that receives and applies the generated `cloud-config` data. The provider controllers enforce the field value:

Provider	Bootstrap userdata format
Huawei DCS	`ignition` (provider-enforced; the DCS provider rejects any other format with `invalid format, expected ignition, got <other>`).
VMware vSphere	`cloud-config` (CAPI bootstrap data is processed by the guest `cloud-init` software; setting `ignition` is not supported).
Huawei Cloud Stack	`cloud-config` (the HCS provider rejects `ignition`; the guest processes the generated data with `cloud-init`).
Bare Metal	`cloud-config` (the bare-metal provider consumes CAPI bootstrap data and renders elemental plans that the guest processes with `cloud-init`).

Huawei DCS

Set the output path for the DCS `global` manifest before you render it.

```
export GLOBAL_DCS_YAML="/root/yamls/new-global.yaml"
```

Create the DCS API credential `Secret` before you assemble the manifest, by following Cloud Credentials. Do not author it as YAML inside the manifest: the manifest is applied with `kubectl apply` in Step 5, which records the credential in the Secret's `kubectl.kubernetes.io/last-applied-configuration` annotation. The manifest references it through `DCSCluster.spec.credentialSecretRef`.

The DCS `global` manifest must then contain the following resources in the `cpaas-system` namespace:

Resource	Purpose
`DCSIpHostnamePool` for control plane nodes	Assigns static IPs, hostnames, network settings, and any pool-managed persistent disks.
`DCSMachineTemplate` for control plane nodes	Defines the DCS VM template, folder, CPU, memory, and template-local disks.
`KubeadmControlPlane`	Bootstraps the Kubernetes control plane. Set `spec.version` to `${K8S_VERSION}`.
`DCSCluster`	Defines the DCS infrastructure cluster and control plane endpoint.
`Cluster`	Connects the Cluster API `Cluster` to `DCSCluster` and `KubeadmControlPlane`.
`DCSIpHostnamePool`, `DCSMachineTemplate`, `KubeadmConfigTemplate`, and `MachineDeployment` for workers	Creates worker nodes.

Use the DCS resource fields from Create a Cluster on Huawei DCS and Infrastructure Resources for Huawei DCS. For the `global` cluster, keep these additional requirements:

- Set `Cluster.metadata.name` and `DCSCluster.metadata.name` to `global` (the infra cluster shares the CAPI `Cluster` name). Prefix every other CAPI resource and

provider resource with `global-`; the wiring fragment below uses `KubeadmControlPlane.metadata.name: global-kcp`.

- Set `DCSCluster.spec.credentialSecretRef.name` to `${PROVIDER_SECRET_NAME}`. Step 7 imports this Secret into the final `global` cluster.
- Add `Cluster.metadata.labels.is-global: "true"` and `Cluster.metadata.labels.cluster-type: DCS`.
- Add `Cluster.metadata.annotations["cpaas.io/registry-address"]` with `${NODE_REGISTRY_ADDRESS}`.
- Set `KubeadmControlPlane.spec.kubeadmConfigSpec.format: ignition` for Alauda OS.
- Use an external LoadBalancer and set `DCSCluster.spec.controlPlaneLoadBalancer.type: external`, or use `type: internal` Self-built VIP. For internal mode, reserve an IPv4 VIP in the control-plane Layer 2 network and use the YAML workflow in Create a Cluster on Huawei DCS. For external mode, validate the listener, backends, health check, and reachability in Plan the Control Plane Endpoint.
- Keep the `KubeadmControlPlane.spec.kubeadmConfigSpec.users` entry with a non-empty `sshAuthorizedKeys` list (the `boot` user). The DCS `ignition` format rejects an empty SSH key list, so this field is required even for a `global` cluster you do not plan to access over SSH. See Resolving Placeholder Values for what to supply when no interactive key is needed.
- Keep the non-encryption kubeadm files, kubelet patches, audit policy, and installer RBAC entries. The file contents (the PodSecurity admission config, the kubelet patch, and the audit policy), together with the full `clusterConfiguration`, `preKubeadmCommands`, `postKubeadmCommands`, and the init and join node-registration patches, are identical to a workload cluster. Copy them from the Complete KubeadmControlPlane Configuration appendix, or reference the `dcs-kubernetes-<major.minor>-files` Secret documented there. The wiring fragment below shows only the `global`-specific fields layered on top of that base.
- For a normal non-DR deployment, do not set `DCSCluster.spec.encryptionProviderConfigRef`. Keep `clusterConfiguration.apiServer.extraArgs.encryption-provider-config` as the appendix has it: the DCS provider writes `/etc/kubernetes/encryption-provider.conf` onto every control-plane node itself — a random key for the first

machine, the running API server's file for the rest — so the argument resolves and the `global` cluster runs with etcd secret encryption-at-rest. Do not add that file to `KubeadmControlPlane.spec.kubeadmConfigSpec.files`; the provider drops any entry for that path, so one added there is silently unused. See How the Encryption Provider Configuration Is Delivered.

- Keep `/var/cpaas` as platform state. If you need the disk to survive rolling replacement, declare it in `DCSIpHostnamePool.spec.pool[].persistentDisk`; do not rely on `DCSMachineTemplate` template disks as preserved state.
- Use concrete `datastoreName` values for DCS local storage unless you have verified that the selected datastore cluster can place volumes on hosts that can run the target VM.

> **Fragment Scope**
>
> The following YAML is a differential fragment, not a complete manifest that you can apply directly. Merge these `global`-specific changes into the manifest that you prepare from the DCS create-cluster references, then apply the complete manifest file. If you would rather start from a complete file, adapt the Worked Example: Complete `global` Manifest for Huawei DCS at the end of this page instead of assembling it from fragments.

The following fragment shows the `global`-specific Cluster API wiring. Fill the provider resource fields by using the DCS create-cluster references above.

```
apiVersion: cluster.x-k8s.io/v1beta1
kind: Cluster
metadata:
  name: global
  namespace: cpaas-system
  labels:
    cluster-type: DCS
    is-global: "true"
  annotations:
    capi.cpaas.io/resource-group-version: infrastructure.cluster.x-k8
s.io/v1beta1
    capi.cpaas.io/resource-kind: DCSCluster
    cpaas.io/registry-address: "${NODE_REGISTRY_ADDRESS}"
spec:
  clusterNetwork:
    pods:
      cidrBlocks:
        - ${CLUSTER_CIDR}
    services:
      cidrBlocks:
        - ${SERVICE_CIDR}
  controlPlaneRef:
    apiVersion: controlplane.cluster.x-k8s.io/v1beta1
    kind: KubeadmControlPlane
    name: global-kcp
  infrastructureRef:
    apiVersion: infrastructure.cluster.x-k8s.io/v1beta1
    kind: DCSCluster
    name: global
---
apiVersion: controlplane.cluster.x-k8s.io/v1beta1
kind: KubeadmControlPlane
metadata:
  name: global-kcp
  namespace: cpaas-system
  annotations:
    controlplane.cluster.x-k8s.io/skip-kube-proxy: ""
spec:
  replicas: 3
  version: ${K8S_VERSION}
  rolloutStrategy:
    type: RollingUpdate
    rollingUpdate:
```

```
      maxSurge: 0
  machineTemplate:
    nodeDrainTimeout: 1m
    nodeDeletionTimeout: 5m
    infrastructureRef:
      apiVersion: infrastructure.cluster.x-k8s.io/v1beta1
      kind: DCSMachineTemplate
      name: global-cp-template
  kubeadmConfigSpec:
    format: ignition
    clusterConfiguration:
      etcd:
        local:
          serverCertSANs:
            - "${CONTROL_PLANE_VIP}"
            - "${PLATFORM_HOST}"
```

VMware vSphere

Set the output path for the VMware vSphere `global` manifest before you render it.

```
export GLOBAL_VSPHERE_YAML="/root/yamls/new-global.yaml"
```

Create the two credential Secrets before you assemble the manifest, and keep them out of it. The manifest is applied with `kubectl apply` in Step 5, which would record their contents in each Secret's `kubectl.kubernetes.io/last-applied-configuration` annotation.

- The vCenter credential, which **must** be named `global-vsphere-credentials` — the import ConfigMap in Step 7 and the DR section hardcode this exact name. Create it as described in Create a Cluster on VMware vSphere, using that fixed name, and reference it from `VSphereCluster.spec.identityRef.name`.
- The vSphere CPI credential, referenced by the `ClusterResourceSet`. Create it as described in the same guide.

The VMware vSphere `global` manifest must then contain the following resources in the `cpaas-system` namespace:

Resource	Purpose
`VSphereMachineConfigPool` for control plane nodes	Assigns static hostnames, datacenter values, IP addresses, network settings, and optional data disks.
`VSphereMachineTemplate` for control plane nodes	Defines the VM template, clone mode, CPU, memory, datastore, folder, network devices, and system disk settings.
`KubeadmControlPlane`	Bootstraps the Kubernetes control plane. Set `spec.version` to `${K8S_VERSION}`.
`VSphereCluster`	Defines the vCenter endpoint, thumbprint, credential reference, and control plane endpoint.
`Cluster`	Connects the Cluster API `Cluster` to `VSphereCluster` and `KubeadmControlPlane`.
`ClusterResourceSet` and `ConfigMap` for the vSphere CPI	Delivers the vSphere CPI resources to the `global` cluster after the API server becomes reachable. The `ClusterResourceSet` also references the CPI credential Secret created above.
`VSphereMachineConfigPool`, `VSphereMachineTemplate`, `KubeadmConfigTemplate`, and `MachineDeployment` for workers	Creates worker nodes.

Prepare the vSphere input values by using VMware vSphere Infrastructure Preparation. Prepare the `global` cluster manifest by using Create a Cluster on VMware vSphere and VMware vSphere Provider as the base references. The create-cluster guide creates workload clusters by applying manifests to an existing `global` management cluster; it does not create the `global` cluster itself. Reuse its vSphere resource definitions only after you apply the following `global`-specific requirements:

- Set `Cluster.metadata.name` and `VSphereCluster.metadata.name` to `global` (the infra cluster shares the CAPI `Cluster` name). Prefix every other CAPI resource

and provider resource with `global-`; the wiring fragment below uses `KubeadmControlPlane.metadata.name: global-kcp`.

- Add `Cluster.metadata.labels.is-global: "true"` and `Cluster.metadata.labels.cluster-type: VSphere`.
- Add `Cluster.metadata.annotations["cpaas.io/registry-address"]` with `${NODE_REGISTRY_ADDRESS}`.
- Keep the VMware vSphere annotations required by the platform controllers, including the network and CPI annotations from the VMware vSphere create-cluster guide.
- Set `VSphereMachineTemplate.spec.template.spec.folder` to `/<datacenter>/vm/global` so operators can identify the `global` cluster VMs in vCenter. In a DR deployment, use distinct child folders such as `/<datacenter>/vm/global/primary` and `/<datacenter>/vm/global/standby` for the primary and standby clusters.
- Set `VSphereCluster.spec.identityRef.name` to `global-vsphere-credentials`. This fixed Secret name is required only for the VMware vSphere `global` installation path; non-`global` VMware vSphere clusters follow the generic create-cluster guide.
- Select the control plane endpoint mode before you apply the manifest; it cannot be changed afterwards. For an external LoadBalancer, provision and validate it first by using the contract in Plan the Control Plane Endpoint. The `global` cluster can instead use an `internal` Self-built VIP: complete Self-built VIP prerequisites, then add the `spec.controlPlaneLoadBalancer` block from Enable the Self-built VIP to the `global` `VSphereCluster`.
  The Self-built VIP prerequisite commands read `ModulePlugin/alive` from a running `global` cluster, which does not exist yet during the bootstrap installation. Confirm from the installation package instead that it carries the `alive` plugin.
- Set `KubeadmControlPlane.spec.kubeadmConfigSpec.format: cloud-config`, or leave the field unset when the provider defaults this API field to `cloud-config`. `cloud-init` is the guest software that processes the data; it is not a valid value for this field.
- Keep the release manifest's kubeadm files, including the VMware vSphere `/etc/kubernetes/encryption-provider.conf` file entry, kubelet patches, audit policy, and installer RBAC entries. VMware vSphere delivers this file through `KubeadmControlPlane.spec.kubeadmConfigSpec.files`; do not follow the DCS `DCSCluster.spec.encryptionProviderConfigRef` pattern.

Fragment Scope

The following YAML is a differential fragment, not a complete manifest that you can apply directly. Merge these `global`-specific changes into the manifest that you prepare from the VMware vSphere create-cluster guide, then apply the complete manifest file.

The following fragment shows the `global`-specific Cluster API wiring. Fill the provider resource fields by using the VMware vSphere create-cluster reference above.

```
apiVersion: cluster.x-k8s.io/v1beta1
kind: Cluster
metadata:
  name: global
  namespace: cpaas-system
  labels:
    cluster-type: VSphere
    is-global: "true"
    addons.cluster.x-k8s.io/vsphere-cpi: "enabled"
  annotations:
    capi.cpaas.io/resource-group-version: infrastructure.cluster.x-k8
s.io/v1beta1
    capi.cpaas.io/resource-kind: VSphereCluster
    cpaas.io/alb-address-type: ClusterAddress
    cpaas.io/network-type: kube-ovn
    cpaas.io/registry-address: "${NODE_REGISTRY_ADDRESS}"
spec:
  clusterNetwork:
    pods:
      cidrBlocks:
        - ${CLUSTER_CIDR}
    services:
      cidrBlocks:
        - ${SERVICE_CIDR}
  controlPlaneRef:
    apiVersion: controlplane.cluster.x-k8s.io/v1beta1
    kind: KubeadmControlPlane
    name: global-kcp
  infrastructureRef:
    apiVersion: infrastructure.cluster.x-k8s.io/v1beta1
    kind: VSphereCluster
    name: global
---
apiVersion: controlplane.cluster.x-k8s.io/v1beta1
kind: KubeadmControlPlane
metadata:
  name: global-kcp
  namespace: cpaas-system
spec:
  replicas: 3
  version: "${K8S_VERSION}"
  rolloutStrategy:
    type: RollingUpdate
```

```
    rollingUpdate:
      maxSurge: 0
  machineTemplate:
    nodeDrainTimeout: 1m
    nodeDeletionTimeout: 5m
    infrastructureRef:
      apiVersion: infrastructure.cluster.x-k8s.io/v1beta1
      kind: VSphereMachineTemplate
      name: global-cp-machine-template
  kubeadmConfigSpec:
    format: cloud-config
    clusterConfiguration:
      etcd:
        local:
          serverCertSANs:
            - "${CONTROL_PLANE_VIP}"
            - "${PLATFORM_HOST}"
```

Huawei Cloud Stack

Set the output path for the HCS `global` manifest before you render it.

```
export GLOBAL_HCS_YAML="/root/yamls/new-global.yaml"
```

The HCS `global` manifest must contain the following resources in the `cpaas-system` namespace:

Resource	Purpose
`Secret`	Stores `accessKey`, `secretKey`, `projectID`, `region`, `externalGlobalDomain`, and optional `schema`.
`HCSMachineConfigPool` for control plane nodes	Assigns static hostnames, IP addresses, and any pool-managed persistent disks.
`Cluster`	Connects the Cluster API `Cluster` to `HCSCluster` and `KubeadmControlPlane`.

Resource	Purpose
`HCSCluster`	Defines VPC, subnet, security group, ELB, and identity references.
`KubeadmControlPlane`	Bootstraps the Kubernetes control plane. Set `spec.version` to `${K8S_VERSION}`.
`HCSMachineTemplate` for control plane nodes	Defines image, flavor, availability zone, root volume, and temporary data volumes.
`HCSMachineConfigPool`, `HCSMachineTemplate`, `KubeadmConfigTemplate`, and `MachineDeployment` for workers	Creates worker nodes.

Use the HCS resource fields from Create a Cluster on Huawei Cloud Stack and Infrastructure Resources for Huawei Cloud Stack. For the `global` cluster, keep these additional requirements:

- Set `Cluster.metadata.name` and `HCSCluster.metadata.name` to `global` (the infra cluster shares the CAPI `Cluster` name). Prefix every other CAPI resource and provider resource with `global-`; the wiring fragment below uses `KubeadmControlPlane.metadata.name: global-kcp`.
- Set `HCSCluster.spec.identityRef.name` to `${PROVIDER_SECRET_NAME}`. Step 7 imports this Secret into the final `global` cluster.
- Add `Cluster.metadata.labels.is-global: "true"` and `Cluster.metadata.labels.cluster-type: HCS`.
- Add `Cluster.metadata.annotations["cpaas.io/registry-address"]` with `${NODE_REGISTRY_ADDRESS}`.
- Set `KubeadmControlPlane.spec.kubeadmConfigSpec.format: cloud-config`, or leave the field unset when the provider defaults this API field to `cloud-config`. `cloud-init` is the guest software name, not the field value.
- Keep the release manifest's non-encryption kubeadm files, kubelet patches, audit policy, and installer RBAC entries.

- For a normal non-DR deployment, do not add `/etc/kubernetes/encryption-provider.conf` to `KubeadmControlPlane.spec.kubeadmConfigSpec.files`.
- Keep `/var/cpaas` as platform state. Declare it in `HCSMachineConfigPool.spec.configs[].persistentDisks[]` when it must survive node replacement; do not rely on `HCSMachineTemplate.spec.template.spec.dataVolumes[]` as preserved state.
- Use a highly available control plane for the `global` cluster. Single-control-plane HCS clusters are creation-only topologies and cannot be upgraded.

> **Fragment Scope**
>
> The following YAML is a differential fragment, not a complete manifest that you can apply directly. Merge these `global`-specific changes into the manifest that you prepare from the HCS create-cluster references, then apply the complete manifest file.

The following fragment shows the `global`-specific Cluster API wiring. Fill the provider resource fields by using the HCS create-cluster references above.

```
apiVersion: cluster.x-k8s.io/v1beta1
kind: Cluster
metadata:
  name: global
  namespace: cpaas-system
  labels:
    cluster-type: HCS
    is-global: "true"
  annotations:
    capi.cpaas.io/resource-group-version: infrastructure.cluster.x-k8
s.io/v1beta1
    capi.cpaas.io/resource-kind: HCSCluster
    cpaas.io/registry-address: "${NODE_REGISTRY_ADDRESS}"
spec:
  clusterNetwork:
    pods:
      cidrBlocks:
        - ${CLUSTER_CIDR}
    services:
      cidrBlocks:
        - ${SERVICE_CIDR}
  controlPlaneRef:
    apiVersion: controlplane.cluster.x-k8s.io/v1beta1
    kind: KubeadmControlPlane
    name: global-kcp
  infrastructureRef:
    apiVersion: infrastructure.cluster.x-k8s.io/v1beta1
    kind: HCSCluster
    name: global
---
apiVersion: controlplane.cluster.x-k8s.io/v1beta1
kind: KubeadmControlPlane
metadata:
  name: global-kcp
  namespace: cpaas-system
spec:
  replicas: 3
  version: "${K8S_VERSION}"
  rolloutStrategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 0
  machineTemplate:
```

```
    nodeDrainTimeout: 1m
    nodeDeletionTimeout: 5m
    infrastructureRef:
      apiVersion: infrastructure.cluster.x-k8s.io/v1beta1
      kind: HCSMachineTemplate
      name: global-cp-machine-template
  kubeadmConfigSpec:
    format: cloud-config
    clusterConfiguration:
      etcd:
        local:
          serverCertSANs:
            - "${CONTROL_PLANE_VIP}"
            - "${PLATFORM_HOST}"
```

Bare Metal

Set the output path for the Bare Metal `global` manifest before you render it.

```
export GLOBAL_BAREMETAL_YAML="/root/yamls/new-global.yaml"
```

The Bare Metal `global` manifest must contain the following resources in the `cpaas-system` namespace:

Resource	Purpose
`MachineRegistration`	Provides the live-ISO registration endpoint and the first-install elemental configuration for the `global` hosts.
`SeedImage`	Builds the bootstrap ISO used to install the `global` hosts. This ISO belongs to the bootstrap lifecycle and is not reused after handoff.
`MachineInventory`	Long-lived host identity created by registration. Wait for the expected inventories before applying the CAPI objects.

Resource	Purpose
`MachineInventoryPool` for control plane nodes	Lists the exact `MachineInventory` names that can back the `global` control plane.
`BaremetalMachineTemplate` for control plane nodes	References the control-plane inventory pool.
`KubeadmControlPlane`	Bootstraps the Kubernetes control plane. Set `spec.version` to `${K8S_VERSION}`.
`BaremetalCluster`	Defines the control-plane VIP, VIP port, VRID, endpoint, and optional encryption provider config Secret reference.
`Cluster`	Connects the Cluster API `Cluster` to `BaremetalCluster` and `KubeadmControlPlane`.
Optional worker `MachineInventoryPool`, `BaremetalMachineTemplate`, `KubeadmConfigTemplate`, and `MachineDeployment`	Creates `global` worker nodes when the topology includes them.

Use Create a Cluster on Bare Metal, Manage Nodes on Bare Metal, and Bare Metal Provider as the resource references. For the `global` cluster, keep these additional requirements:

- Set `Cluster.metadata.name` and `BaremetalCluster.metadata.name` to `global`. Prefix every other CAPI, bare-metal, and elemental resource with `global-`.
- Add `Cluster.metadata.labels.cluster-type: ProviderBaremetal`.
- Add `Cluster.metadata.annotations["cpaas.io/registry-address"]` with `${NODE_REGISTRY_ADDRESS}`.
- Set `SeedImage.spec.baseImage` to `${BOOTSTRAP_REGISTRY_ADDRESS}/tkestack/baremetal-base-image-iso:${OS_IMAGE_TAG}`, the ISO image imported in Step 3. The reprovision image comes from the `elemental-image-catalog` entry set in Step 3; the provider

replaces the registry portion of that entry with the `cpaas.io/registry-address` annotation, so the hosts pull it from `${NODE_REGISTRY_ADDRESS}`.

- Add `Cluster.metadata.annotations["cpaas.io/kube-ovn-join-cidr"]`, `Cluster.metadata.annotations["cpaas.io/sentry-deploy-type"]: Baremetal`, and `Cluster.metadata.annotations["cpaas.io/alb-address-type"]: ClusterAddress`.
- Set `KubeadmControlPlane.spec.kubeadmConfigSpec.format: cloud-config`, or leave it unset when the provider defaults this API field to `cloud-config`. The resulting data is processed by `cloud-init` on the node.
- Set `KubeadmControlPlane.spec.rolloutStrategy.rollingUpdate.maxSurge: 0`. Bare-metal pools cannot over-provision physical hosts.
- Keep `controlplane.cluster.x-k8s.io/skip-kube-proxy: ""` on the `KubeadmControlPlane` when the release manifest uses kube-ovn.
- Put `${CONTROL_PLANE_VIP}` and `${PLATFORM_HOST}` in `KubeadmControlPlane.spec.kubeadmConfigSpec.clusterConfiguration.etcd.local.serverCertSANs`.
- Choose the Bare Metal endpoint mode in Plan the Control Plane Endpoint. For `Internal`, set the VIP, port `6443`, and a unique `vrid`. For `External`, set the external load balancer frontend address and port, and omit `vrid`.
- For a normal non-DR deployment, `BaremetalCluster.spec.encryptionProviderConfigRef` can be omitted. For DR, set it as described in Optional Disaster Recovery Deployment; do not deliver `/etc/kubernetes/encryption-provider.conf` by adding it to `KubeadmControlPlane.spec.kubeadmConfigSpec.files`.
- For Bare Metal DR, add `baremetal.cluster.io/system-agent-auth-scope: global` to the `MachineRegistration` used for the bootstrap `global` hosts. Do not set `baremetal.cluster.io/system-agent-server-url` during bootstrap. The bootstrap ISO must register through the bootstrap node; the handoff job later moves the `global` machines to the local direct API endpoint.
- If a `global` VM or physical host does not have DHCP during the live-ISO boot, configure the NIC manually from the host console before waiting for `MachineInventory` registration. Use the same NetworkManager procedure

described in Create a Cluster on Bare Metal, replacing the example address, gateway, DNS, and connection name with the values for that host.

- Set the clock from firmware on every `global` host before the live-ISO boot, and keep the spread across the hosts within the platform's 10-second requirement. See Host Time Synchronization.
- Do not depend on OS hostname side effects. The bare-metal provider normalizes kubeadm node names and provider IDs from the CAPI and inventory objects.
- The `SeedImage` created in the bootstrap cluster is a bootstrap artifact. After handoff, create any new `MachineRegistration` or `SeedImage` on the active `global` cluster.

> **Fragment Scope**
>
> The following YAML is a differential fragment, not a complete manifest that you can apply directly. Merge these `global`-specific changes into the manifest that you prepare from the Bare Metal create-cluster references, then apply the complete manifest file.

The following fragment shows the `global`-specific Cluster API wiring. Fill the inventory names, registration configuration, image references, and optional worker resources by using the Bare Metal create-cluster reference.

Set `<control-plane-load-balancer-type>` to `Internal` for provider-managed Alive or `External` for a user-provisioned LoadBalancer. When you use `External`, remove `vrid`. See Plan the Control Plane Endpoint for the requirements of each mode.

```
apiVersion: cluster.x-k8s.io/v1beta1
kind: Cluster
metadata:
  name: global
  namespace: cpaas-system
  labels:
    cluster-type: ProviderBaremetal
  annotations:
    capi.cpaas.io/resource-group-version: infrastructure.cluster.x-k8
s.io/v1beta1
    capi.cpaas.io/resource-kind: BaremetalCluster
    cpaas.io/kube-ovn-join-cidr: "${KUBE_OVN_JOIN_CIDR}"
    cpaas.io/registry-address: "${NODE_REGISTRY_ADDRESS}"
    cpaas.io/sentry-deploy-type: Baremetal
    cpaas.io/alb-address-type: ClusterAddress
spec:
  clusterNetwork:
    pods:
      cidrBlocks:
        - ${CLUSTER_CIDR}
    services:
      cidrBlocks:
        - ${SERVICE_CIDR}
  controlPlaneRef:
    apiVersion: controlplane.cluster.x-k8s.io/v1beta1
    kind: KubeadmControlPlane
    name: global-kcp
  infrastructureRef:
    apiVersion: infrastructure.cluster.x-k8s.io/v1beta1
    kind: BaremetalCluster
    name: global
---
apiVersion: infrastructure.cluster.x-k8s.io/v1beta1
kind: BaremetalCluster
metadata:
  name: global
  namespace: cpaas-system
spec:
  controlPlaneLoadBalancer:
    type: <control-plane-load-balancer-type>
    host: ${CONTROL_PLANE_VIP}
    port: 6443
    # Required only for Internal. Remove this field for External.
```

```
    vrid: <unique-vrid>
    # vipMode defaults to nic. Set it explicitly only when the enviro
nment
    # requires another supported mode, such as arp or policy_route.
    # vipMode: nic
  # Required for DR. Omit this field for a normal non-DR deployment u
nless
  # you need to provide a pre-existing encryption-provider.conf.
  # encryptionProviderConfigRef:
  #   name: global-encryption-provider-config
---
apiVersion: infrastructure.cluster.x-k8s.io/v1beta1
kind: MachineInventoryPool
metadata:
  name: global-control-plane-pool
  namespace: cpaas-system
spec:
  clusterName: global
  machineInventories:
    - global-cp-1
    - global-cp-2
    - global-cp-3
---
apiVersion: infrastructure.cluster.x-k8s.io/v1beta1
kind: BaremetalMachineTemplate
metadata:
  name: global-control-plane-template
  namespace: cpaas-system
spec:
  template:
    spec:
      machineInventoryPoolRef:
        name: global-control-plane-pool
---
apiVersion: controlplane.cluster.x-k8s.io/v1beta1
kind: KubeadmControlPlane
metadata:
  name: global-kcp
  namespace: cpaas-system
  annotations:
    controlplane.cluster.x-k8s.io/skip-kube-proxy: ""
spec:
  replicas: 3
  version: "${K8S_VERSION}"
```

```
  rolloutStrategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 0
  machineTemplate:
    nodeDrainTimeout: 1m
    nodeDeletionTimeout: 5m
    infrastructureRef:
      apiVersion: infrastructure.cluster.x-k8s.io/v1beta1
      kind: BaremetalMachineTemplate
      name: global-control-plane-template
  kubeadmConfigSpec:
    format: cloud-config
    clusterConfiguration:
      etcd:
        local:
          serverCertSANs:
            - "${CONTROL_PLANE_VIP}"
            - "${PLATFORM_HOST}"
```

Step 5 — Apply the global Manifest

Apply the provider-specific manifest to the bootstrap cluster.

Huawei DCS

```
kubectl apply -f "${GLOBAL_DCS_YAML}"
```

VMware vSphere

```
kubectl apply -f "${GLOBAL_VSPHERE_YAML}"
```

Huawei Cloud Stack

```
kubectl apply -f "${GLOBAL_HCS_YAML}"
```

Bare Metal

```
kubectl apply -f "${GLOBAL_BAREMETAL_YAML}"
```

Wait for the bootstrap registrations to produce the expected inventories before you expect Cluster API reconciliation to progress.

```
kubectl -n cpaas-system get machineinventory.elemental.cattle.io
kubectl -n cpaas-system get machineinventorypool
kubectl -n cpaas-system get baremetalcluster,baremetalmachine
```

Step 6 — Wait for the Control Plane

Wait for the Cluster API provider to provision the machines and bring up the Kubernetes control plane.

```
kubectl get clusters.cluster.x-k8s.io -n cpaas-system
kubectl get kubeadmcontrolplane -n cpaas-system
kubectl get machines -n cpaas-system
```

The control plane is ready when the `KubeadmControlPlane` reports `Ready: True` and the `Cluster` reports `Phase: Provisioned`.

Step 7 — Import Provider Resources

Before triggering the installer, create the `dcs-import-extra-resources` ConfigMap in the `cpaas-system` namespace for providers that require extra resource import. The ConfigMap name keeps the `dcs` prefix for historical installer compatibility, even when the provider is not Huawei DCS.

Every provider uses this ConfigMap to import the IaaS credential `Secret` into the new `global` cluster. For VMware vSphere, Huawei Cloud Stack, and Bare Metal it also imports the provider's Cluster API resources; for Huawei DCS those resources are migrated by the built-in flow, so the DCS ConfigMap only needs the credential `Secret` entry for a normal

installation. VMware vSphere, Huawei Cloud Stack, and Bare Metal require it for both normal and disaster recovery `global` installations.

For a disaster recovery installation, the two providers that deliver the encryption provider configuration through a Secret reference — Huawei DCS (`DCSCluster.spec.encryptionProviderConfigRef`) and Bare Metal (`BaremetalCluster.spec.encryptionProviderConfigRef`) — must also import that Secret through this ConfigMap. The built-in DCS migration flow moves Cluster API resources only; it does not copy the referenced Secret.

Huawei DCS

The DCS provider's Cluster API resources are migrated by the built-in flow, so for a normal installation the ConfigMap only needs to import the credential `Secret` referenced by `DCSCluster.spec.credentialSecretRef.name`. Use the same Secret name you set in the `global` manifest.

```
mkdir -p /root/yamls
cat > /root/yamls/dcs-import-extra-resources.yaml <<EOF
apiVersion: v1
kind: ConfigMap
metadata:
  name: dcs-import-extra-resources
  namespace: cpaas-system
data:
  resources.yaml: |
    resources:
    - resource: "secrets"
      names: ["${PROVIDER_SECRET_NAME}"]
      method: kubectl
EOF

kubectl apply -f /root/yamls/dcs-import-extra-resources.yaml
```

For a DR installation, use the form below instead. It adds the encryption-provider `Secret` referenced by `DCSCluster.spec.encryptionProviderConfigRef.name`, which must reach the final `global` cluster. `DCS_ENCRYPTION_PROVIDER_SECRET` comes from Prepare Shared DR Variables; the first line fails loudly if it is not exported, because the heredoc would otherwise expand it to an empty Secret name.

```
mkdir -p /root/yamls
: "${DCS_ENCRYPTION_PROVIDER_SECRET:?export it first — see Prepare Sh
ared DR Variables}"
cat > /root/yamls/dcs-import-extra-resources.yaml <<EOF
apiVersion: v1
kind: ConfigMap
metadata:
  name: dcs-import-extra-resources
  namespace: cpaas-system
data:
  resources.yaml: |
    resources:
    - resource: "secrets"
      names:
        - "${PROVIDER_SECRET_NAME}"
        # The name must match the Secret referenced by
        # DCSCluster.spec.encryptionProviderConfigRef.
        - "${DCS_ENCRYPTION_PROVIDER_SECRET}"
      method: kubectl
EOF

kubectl apply -f /root/yamls/dcs-import-extra-resources.yaml
kubectl -n cpaas-system get cm dcs-import-extra-resources -o yaml
```

Check the rendered ConfigMap before continuing: both `names` entries must be non-empty, and the second must match `DCSCluster.spec.encryptionProviderConfigRef.name` exactly.

For DR, verify after installation that the final `global` cluster has the imported encryption-provider Secret.

```
kubectl --kubeconfig <global-kubeconfig> -n cpaas-system \
  get secret "${DCS_ENCRYPTION_PROVIDER_SECRET}" \
  -o jsonpath='{.data.encryption-provider\.conf}{"\n"}'
```

VMware vSphere

Create and apply the VMware vSphere import ConfigMap before you trigger the installer. This ConfigMap is required for both normal and disaster recovery `global` installations.

The `global-vsphere-credentials` Secret stores the vCenter username and password and must be the same Secret name referenced by `VSphereCluster.spec.identityRef.name` in the VMware vSphere `global` manifest.

```
mkdir -p /root/yamls
cat > /root/yamls/dcs-import-extra-resources.yaml <<EOF
apiVersion: v1
kind: ConfigMap
metadata:
  name: dcs-import-extra-resources
  namespace: cpaas-system
data:
  resources.yaml: |
    resources:
    - resource: "vsphereclusters.infrastructure.cluster.x-k8s.io"
      names: ["global"]
      etcdKeyBase: "/registry/infrastructure.cluster.x-k8s.io/vsphere
clusters/cpaas-system/"
      method: etcdctl
    - resource: "vspheremachinetemplates.infrastructure.cluster.x-k8
s.io"
      etcdKeyBase: "/registry/infrastructure.cluster.x-k8s.io/vsphere
machinetemplates/cpaas-system/"
      method: etcdctl
    - resource: "vspheremachines.infrastructure.cluster.x-k8s.io"
      etcdKeyBase: "/registry/infrastructure.cluster.x-k8s.io/vsphere
machines/cpaas-system/"
      method: etcdctl
    - resource: "vspherevms.infrastructure.cluster.x-k8s.io"
      etcdKeyBase: "/registry/infrastructure.cluster.x-k8s.io/vsphere
vms/cpaas-system/"
      method: etcdctl
    - resource: "vspheremachineconfigpools.infrastructure.cluster.x-k
8s.io"
      etcdKeyBase: "/registry/infrastructure.cluster.x-k8s.io/vsphere
machineconfigpools/cpaas-system/"
      method: etcdctl
    - resource: "secrets"
      names: ["global-vsphere-credentials"]
      method: kubectl
EOF

kubectl apply -f /root/yamls/dcs-import-extra-resources.yaml
```

Huawei Cloud Stack

Create and apply the HCS import ConfigMap before you trigger the installer. This ConfigMap is required for both normal and disaster recovery `global` installations. Set `PROVIDER_SECRET_NAME` to the same Secret name used by `HCSCluster.spec.identityRef.name`.

```
mkdir -p /root/yamls
cat > /root/yamls/dcs-import-extra-resources.yaml <<EOF
apiVersion: v1
kind: ConfigMap
metadata:
  name: dcs-import-extra-resources
  namespace: cpaas-system
data:
  resources.yaml: |
    resources:
    - resource: "secrets"
      names: ["${PROVIDER_SECRET_NAME}"]
      method: kubectl
    - resource: "hcsclusters.infrastructure.cluster.x-k8s.io"
      names: ["global"]
      etcdKeyBase: "/registry/infrastructure.cluster.x-k8s.io/hcsclus
ters/cpaas-system/"
      method: etcdctl
    - resource: "hcsmachinetemplates.infrastructure.cluster.x-k8s.io"
      etcdKeyBase: "/registry/infrastructure.cluster.x-k8s.io/hcsmach
inetemplates/cpaas-system/"
      method: etcdctl
    - resource: "hcsmachineconfigpools.infrastructure.cluster.x-k8s.i
o"
      etcdKeyBase: "/registry/infrastructure.cluster.x-k8s.io/hcsmach
ineconfigpools/cpaas-system/"
      method: etcdctl
    - resource: "hcsmachines.infrastructure.cluster.x-k8s.io"
      etcdKeyBase: "/registry/infrastructure.cluster.x-k8s.io/hcsmach
ines/cpaas-system/"
      method: etcdctl
EOF

kubectl apply -f /root/yamls/dcs-import-extra-resources.yaml
```

Bare Metal

Create and apply the Bare Metal import ConfigMap before you trigger the installer. This ConfigMap imports the durable bare-metal and elemental owner resources that the fresh-install handoff uses to enumerate the installed Global machines. It must not import Elemental plan Secrets or kubeadm bootstrap data Secrets.

Import Before DCS API

Create `dcs-import-extra-resources` before calling `POST /cpaas-installer/api/config/dcs`. If it is missing, the handoff job can run with an empty target list because the new `global` cluster does not contain the `BaremetalMachine`, `MachineInventory`, or `MachineRegistration` objects that describe the bootstrap `global` machines. Do not compensate by adding their plan Secrets or kubeadm bootstrap data Secrets to the ConfigMap.

Collect the exact Global `MachineInventory` names from the reconciled `BaremetalMachine` objects and identify the bootstrap `MachineRegistration` used by those hosts. For DR, also collect the Secret name referenced by `BaremetalCluster.spec.encryptionProviderConfigRef` so the final `global` cluster contains the same encryption-provider configuration. Do not import arbitrary platform credential Secrets or the MachineRegistration token Secret.

```
kubectl -n cpaas-system get baremetalmachine \
  -l cluster.x-k8s.io/cluster-name=global \
  -o custom-columns='NAME:.metadata.name,INVENTORY:.status.machineInv
entoryRef.name'
kubectl -n cpaas-system get machineregistration.elemental.cattle.io
kubectl -n cpaas-system get baremetalcluster global \
  -o jsonpath='{.spec.encryptionProviderConfigRef.name}{"\n"}'
```

Create the ConfigMap. Replace the inventory and MachineRegistration placeholders with the values from the commands above. For DR, also replace the encryption-provider Secret placeholder; for a non-DR installation that does not set `BaremetalCluster.spec.encryptionProviderConfigRef`, remove that Secret entry.

```
mkdir -p /root/yamls
cat > /root/yamls/dcs-import-extra-resources.yaml <<EOF
apiVersion: v1
kind: ConfigMap
metadata:
  name: dcs-import-extra-resources
  namespace: cpaas-system
data:
  resources.yaml: |
    resources:
    - resource: "customresourcedefinitions.apiextensions.k8s.io"
      names:
        - baremetalclusters.infrastructure.cluster.x-k8s.io
        - baremetalmachines.infrastructure.cluster.x-k8s.io
        - baremetalmachinetemplates.infrastructure.cluster.x-k8s.io
        - machineinventorypools.infrastructure.cluster.x-k8s.io
        - machineinventories.elemental.cattle.io
        - machineregistrations.elemental.cattle.io
        - seedimages.elemental.cattle.io
      method: kubectl
    - resource: "baremetalclusters.infrastructure.cluster.x-k8s.io"
      names: ["global"]
      etcdKeyBase: "/registry/infrastructure.cluster.x-k8s.io/baremet
alclusters/cpaas-system/"
      method: etcdctl
    - resource: "baremetalmachinetemplates.infrastructure.cluster.x-k
8s.io"
      etcdKeyBase: "/registry/infrastructure.cluster.x-k8s.io/baremet
almachinetemplates/cpaas-system/"
      method: etcdctl
    - resource: "baremetalmachines.infrastructure.cluster.x-k8s.io"
      etcdKeyBase: "/registry/infrastructure.cluster.x-k8s.io/baremet
almachines/cpaas-system/"
      method: etcdctl
    - resource: "machineinventorypools.infrastructure.cluster.x-k8s.i
o"
      etcdKeyBase: "/registry/infrastructure.cluster.x-k8s.io/machine
inventorypools/cpaas-system/"
      method: etcdctl
    - resource: "machineinventories.elemental.cattle.io"
      names:
        - "<global-machine-inventory-1>"
        - "<global-machine-inventory-2>"
```

```
        - "<global-machine-inventory-3>"
      etcdKeyBase: "/registry/elemental.cattle.io/machineinventories/
cpaas-system/"
      method: etcdctl
    - resource: "machineregistrations.elemental.cattle.io"
      names: ["<global-machine-registration>"]
      etcdKeyBase: "/registry/elemental.cattle.io/machineregistration
s/cpaas-system/"
      method: etcdctl
    # Required for DR when BaremetalCluster.spec.encryptionProviderCo
nfigRef is set.
    # The name must match the Secret referenced by BaremetalCluster.
    - resource: "secrets"
      names:
        - "<global-encryption-provider-config-secret>"
      method: kubectl
EOF

kubectl apply -f /root/yamls/dcs-import-extra-resources.yaml
kubectl -n cpaas-system get cm dcs-import-extra-resources -o yaml
```

> **Do Not Import Plan or Kubeadm Bootstrap Secrets**
>
> The installer applies `method: kubectl` extras before its batched etcd import has finished creating their owner objects. An Elemental plan Secret imported at that point still references the bootstrap `MachineInventory` UID; that owner UID does not identify the later imported Global `MachineInventory`, so Kubernetes garbage collection can delete the Secret. Do not add Elemental plan Secrets, `KubeadmConfig` objects, or `Machine.spec.bootstrap.dataSecretName` Secrets to this ConfigMap.

The first-version fresh-install handoff handles this ordering explicitly. It verifies that all target Global `MachineInventory` objects exist, reads each source plan Secret from the still-running bootstrap cluster, and creates the Global plan Secret with the live Global `MachineInventory` UID as its controller owner.

`KubeadmConfig` objects and kubeadm bootstrap data Secrets are not used by the first-version fresh-install handoff and must not be added to this ConfigMap.

Before continuing to Step 8, check the rendered ConfigMap:

- It includes the Bare Metal CRDs, `BaremetalCluster/global`, Bare Metal machine resources, the exact Global `MachineInventory` objects, and the Global `MachineRegistration`.
- It does not include `SeedImage` objects, MachineRegistration token Secrets, Elemental plan Secrets, `KubeadmConfig` objects, or kubeadm bootstrap data Secrets.
- For DR, it still includes the encryption-provider Secret whose name exactly matches `BaremetalCluster.spec.encryptionProviderConfigRef.name`.

A bootstrap `SeedImage` produces an ISO that points at the bootstrap environment and is no longer the correct lifecycle object after the `global` cluster has been handed off. The `seedimages.elemental.cattle.io` CRD is imported only so the new `global` cluster understands the API type.

For DR, verify after installation that the final `global` cluster has the imported encryption-provider Secret.

```
kubectl --kubeconfig <global-kubeconfig> -n cpaas-system \
  get secret <global-encryption-provider-config-secret> \
  -o jsonpath='{.data.encryption-provider\.conf}{"\n"}'
```

Step 8 — Trigger the Platform Installation

Submit the platform installation request to the embedded installer REST API. The installer imports the Cluster API resources into the new `global` cluster, deploys the base operator, and installs the selected plugins.

```
export INSTALLER_IP=$(kubectl get pods -n cpaas-system -l service_name=cpaas-installer \
  -o jsonpath='{.items[0].status.podIP}')
```

Network Scope

`INSTALLER_IP` is the Pod IP of the embedded installer in the bootstrap cluster. The endpoint is used only during installation.

Create the provider-specific installer configuration JSON file on the current bootstrap node, then submit it to the installer endpoint. All providers in this install path use the same endpoint path, but their request bodies are different.

Field	Huawei DCS	VMware vSphere
Endpoint path	`/cpaas-installer/api/config/dcs`	`/cpaas-installer/api/config/dcs`
`console.host`	`["${CONTROL_PLANE_VIP}"]`	`["${CONTROL_PLANE_VIP}"]`
`console.globalHost`	Platform access address	Platform access address
`cluster.clusterCIDR` and `cluster.serviceCIDR`	Required	Not set; cluster CIDRs are declared in t vSphere `Cluster` manifest
`cluster.features.ha`	Required, `vip` is `${CONTROL_PLANE_VIP}` with `isThirdParty: true`	Required, `vip` is `${CONTROL_PLANE_` `isThirdParty: true`. The same for l modes; the VIP is also declared in `VSphereCluster.spec.controlPlane`
`hostIP`	Current bootstrap node IP	Current bootstrap node IP

Each `console.host` entry is a bare address: no `https://` prefix and no port. The ports come from `console.httpPort` and `console.httpsPort`.

If you started the bootstrap script with an external platform image registry in Step 2, set `registry.externalAddress` to the registry address in place of `registry.domain` in the `registry` object of the request, and do not set both. Keep `registry.username` and `registry.password` as in the examples: they are the credentials used to pull from the registry. Omit both if the registry allows anonymous pull. The installer does not copy images or packages into an external registry, so the target-version payload must already be uploaded; see Prepare an External Platform Image Registry.

Huawei DCS

The DCS installer request includes the `global` control-plane VIP. With an external LoadBalancer, this is the externally provisioned VIP. With `Self-built VIP`, use the reserved VIP that the provider-managed `alive` runtime owns.

```
mkdir -p /root/yamls
export INSTALLER_CONFIG_JSON="/root/yamls/installer-config-dcs.json"

cat > "${INSTALLER_CONFIG_JSON}" <<EOF
{
  "basic": {
    "username": "admin@cpaas.io",
    "password": "<base64-platform-admin-password>"
  },
  "registry": {
    "domain": "${REGISTRY_DOMAIN}",
    "username": "<registry-username>",
    "password": "<base64-registry-password>"
  },
  "console": {
    "host": [
      "${CONTROL_PLANE_VIP}"
    ],
    "globalHost": "${PLATFORM_HOST}",
    "httpPort": 80,
    "httpsPort": 443,
    "cert": {
      "selfSigned": {}
    }
  },
  "cluster": {
    "clusterCIDR": "${CLUSTER_CIDR}",
    "serviceCIDR": "${SERVICE_CIDR}",
    "features": {
      "ha": {
        "vip": "${CONTROL_PLANE_VIP}",
        "vport": 6443,
        "isThirdParty": true
      }
    }
  },
  "product": [
    "base",
    "acp"
  ],
  "deployMode": "normal",
  "hostIP": "${HOST_IP}"
}
```

```
EOF

curl -k -X POST "http://${INSTALLER_IP}:8080/cpaas-installer/api/conf
ig/dcs" \
  -H 'Content-Type: application/json' \
  -d @"${INSTALLER_CONFIG_JSON}"
```

Set `console.host` to the local `global` control-plane VIP, `${CONTROL_PLANE_VIP}`. Do not use the platform domain in `console.host`; use `console.globalHost` for the platform access address. `cluster.features.ha.vip` uses the same address.

VMware vSphere

VMware vSphere uses the same installer endpoint path as DCS. The request body carries the control plane VIP in `console.host` and `cluster.features.ha`, the same way for an external LoadBalancer and for a Self-built VIP; the same address is also declared in `VSphereCluster.spec.controlPlaneEndpoint.host`. The cluster CIDRs are not sent here — they are declared in the VMware vSphere `Cluster` manifest.

```
mkdir -p /root/yamls
export INSTALLER_CONFIG_JSON="/root/yamls/installer-config-vsphere.json"

cat > "${INSTALLER_CONFIG_JSON}" <<EOF
{
  "basic": {
    "username": "admin@cpaas.io",
    "password": "<base64-platform-admin-password>"
  },
  "registry": {
    "domain": "${REGISTRY_DOMAIN}",
    "username": "<registry-username>",
    "password": "<base64-registry-password>"
  },
  "console": {
    "host": [
      "${CONTROL_PLANE_VIP}"
    ],
    "globalHost": "${PLATFORM_HOST}",
    "httpPort": 80,
    "httpsPort": 443,
    "cert": {
      "selfSigned": {}
    }
  },
  "cluster": {
    "features": {
      "ha": {
        "vip": "${CONTROL_PLANE_VIP}",
        "vport": 6443,
        "isThirdParty": true
      }
    }
  },
  "product": [
    "base",
    "acp"
  ],
  "deployMode": "normal",
  "hostIP": "${HOST_IP}"
}
EOF
```

```
curl -k -X POST "http://${INSTALLER_IP}:8080/cpaas-installer/api/conf
ig/dcs" \
  -H 'Content-Type: application/json' \
  -d @"${INSTALLER_CONFIG_JSON}"
```

Set `console.host` and `cluster.features.ha.vip` to the local `global` control-plane VIP, `${CONTROL_PLANE_VIP}`, whether that address is served by an external LoadBalancer or by the Self-built VIP. Do not use the platform domain in `console.host`; use `console.globalHost` for the platform access address. `VSphereCluster.spec.controlPlaneEndpoint.host` uses the same address.

Huawei Cloud Stack

HCS uses the same installer endpoint path as DCS, but its request body does not include `cluster.features.ha`. The control-plane VIP is owned by the HCS ELB declared in `HCSCluster.spec.controlPlaneLoadBalancer`.

```
mkdir -p /root/yamls
export INSTALLER_CONFIG_JSON="/root/yamls/installer-config-hcs.json"

cat > "${INSTALLER_CONFIG_JSON}" <<EOF
{
  "basic": {
    "username": "admin@cpaas.io",
    "password": "<base64-platform-admin-password>"
  },
  "registry": {
    "domain": "${REGISTRY_DOMAIN}",
    "username": "<registry-username>",
    "password": "<base64-registry-password>"
  },
  "console": {
    "host": [
      "${CONTROL_PLANE_VIP}"
    ],
    "globalHost": "${PLATFORM_HOST}",
    "httpPort": 80,
    "httpsPort": 443,
    "cert": {
      "selfSigned": {}
    }
  },
  "product": [
    "base",
    "acp"
  ],
  "deployMode": "normal",
  "hostIP": "${HOST_IP}"
}
EOF

curl -k -X POST "http://${INSTALLER_IP}:8080/cpaas-installer/api/conf
ig/dcs" \
  -H 'Content-Type: application/json' \
  -d @"${INSTALLER_CONFIG_JSON}"
```

Set `console.host` to the local `global` control-plane VIP, `${CONTROL_PLANE_VIP}`. Do not use the platform domain in `console.host`; use `console.globalHost` for the

platform access address. `HCSCluster.spec.controlPlaneLoadBalancer.vipAddress` uses the same address.

Bare Metal

The Bare Metal installer request includes the control-plane endpoint used after handoff. In `Internal` mode, Alive exposes the self-built VIP. In `External` mode, the user-provisioned load balancer exposes the same endpoint and maintains the control-plane backends. With the platform built-in Registry, set `REGISTRY_DOMAIN` to `${PLATFORM_HOST}:11443`. Keep `registry.externalAddress` unset so the installer deploys and populates the local platform Registry. Do not use the bootstrap Registry address in this field.

In a DR pair that uses the platform built-in Registry, both `global` clusters use the same `${PLATFORM_HOST}:11443` Registry address. This keeps `ProductBase.spec.registry.address` and the two `cpaas.io/registry-address` Cluster annotations identical on both sides, so **Alauda Container Platform Global Cluster Disaster Recovery** replicates them as a no-op instead of overwriting the standby values with the primary ones. Each `global` cluster still runs its own Registry; the platform domain simply resolves to whichever cluster is currently active.

Registry credentials follow the platform domain

Each `global` cluster has its own `cpaas-system/registry-admin` credential, but the platform domain resolves to only one of them at a time. When you push artifacts to `${PLATFORM_HOST}:11443` — for example a provider package or the **Alauda Container Platform Global Cluster Disaster Recovery** package — authenticate with the Registry credential of the cluster the domain currently points to. After a failover the domain points at the other cluster, so the credential to use changes with it. A wrong credential fails with `unable to retrieve auth token: invalid username/password`, which does not indicate that you reached the wrong cluster. Anonymous pull works across sites and is unaffected.

```
mkdir -p /root/yamls
export INSTALLER_CONFIG_JSON="/root/yamls/installer-config-baremetal.
json"

cat > "${INSTALLER_CONFIG_JSON}" <<EOF
{
  "basic": {
    "username": "admin@cpaas.io",
    "password": "<base64-platform-admin-password>"
  },
  "registry": {
    "domain": "${REGISTRY_DOMAIN}",
    "username": "<registry-username>",
    "password": "<base64-registry-password>"
  },
  "console": {
    "host": [
      "${CONTROL_PLANE_VIP}"
    ],
    "globalHost": "${PLATFORM_HOST}",
    "httpPort": 80,
    "httpsPort": 443,
    "cert": {
      "selfSigned": {}
    }
  },
  "cluster": {
    "clusterCIDR": "${CLUSTER_CIDR}",
    "serviceCIDR": "${SERVICE_CIDR}",
    "features": {
      "ha": {
        "vip": "${CONTROL_PLANE_VIP}",
        "vport": 6443,
        "isThirdParty": true
      }
    }
  },
  "product": [
    "base",
    "acp"
  ],
  "deployMode": "normal",
  "hostIP": "${HOST_IP}"
```

```
}
EOF

curl -k -X POST "http://${INSTALLER_IP}:8080/cpaas-installer/api/conf
ig/dcs" \
  -H 'Content-Type: application/json' \
  -d @"${INSTALLER_CONFIG_JSON}"
```

Set `console.host` to the local `global` control-plane VIP, `${CONTROL_PLANE_VIP}`. Do not use the platform domain in `console.host`; use `console.globalHost` for the platform access address. `cluster.features.ha.vip` uses the same address.

Third-Party Console Certificates

The examples use a self-signed console certificate. If the environment requires a third-party certificate, replace `console.cert` with a `thirdParty` block that contains the base64 full certificate chain, private key, and optional PKCS#12 values before you submit the installer request. This certificate is only for platform HTTPS ingress; it does not configure the kube-apiserver or etcd certificates generated from `KubeadmControlPlane`. The Global registry on port `11443` uses an independently managed `global-registry-server` certificate chain, not `console.cert` or `dex.tls`.

DR Certificate Requirement

For a primary/standby Bare Metal `global` DR deployment, do not let each side generate an unrelated self-signed certificate. Configure the same trusted `thirdParty` platform certificate on both sides. Its required SAN is the stable `${PLATFORM_HOST}` domain. Do not add the primary or standby control-plane VIPs or internal Service names by default. Add a VIP SAN only when clients intentionally access platform HTTPS directly through that VIP. Registry access through port `11443` does not use this certificate; registry certificate management is independent and outside this certificate step. Global hosts handed off to `https://<control-plane-vip>:6443` validate the separate kube-apiserver certificate and CA, not `console.cert` or `dex.tls`.

Step 9 — Monitor the Installation

After the installer accepts the request, the install runs through several phases that are observable from the bootstrap node. A typical `global` cluster on Alauda OS takes 30–60 minutes; total time depends on IaaS provisioning speed, image pull time, and the number of plugins selected.

Phases You Will Observe

Phase	What is happening	First place to watch
Bootstrap	The bootstrap cluster, embedded registry, and Cluster API providers are running on the bootstrap node. Completed in Step 2 and Step 3.	bootstrap node terminal; `kubectl get pods -n cpaas-system`
Infrastructure provisioning	The Cluster API provider creates VMs from the Alauda OS template on the target IaaS platform.	`kubectl get machines -n cpaas-system`
Control plane bootstrap	`KubeadmControlPlane` bootstraps the first control plane node, etcd starts, and additional control plane nodes join.	`kubectl get kubeadmcontrolplane -n cpaas-system`
Network and core add-ons	The CAPI provider reconciles Kube-OVN, CoreDNS, and kube-proxy on the new cluster.	`kubectl --kubeconfig <global-kubeconfig> get pods -n kube-system`
Platform installation	The installer imports Cluster API resources into the new `global` cluster, deploys the base operator, and installs the selected plugins.	Installer progress API; installer log
Completion	The installer marks the request as `Success` and writes the final cluster state into `ClusterModule/global`.	Installer progress API; `kubectl --kubeconfig <global-kubeconfig> get clustermodule global`

Signals During Installation

Watch the installer progress API and the installer log together. If one appears stalled, check the underlying Cluster API resources directly on the bootstrap node.

```
# Installer progress and live log
curl "http://${INSTALLER_IP}:8080/cpaas-installer/api/progress"
tail -f /var/cpaas/data/installer.log

# Cluster API resources on the bootstrap node
kubectl get clusters.cluster.x-k8s.io -A
kubectl get kubeadmcontrolplane -A
kubectl get machines -A
```

The installer log records every phase transition. Transient errors retry on a short interval; persistent errors stay visible in the log and surface in the progress API as a stalled stage.

Check the `global` cluster after the installer reports success.

```
kubectl --kubeconfig <global-kubeconfig> get nodes
kubectl --kubeconfig <global-kubeconfig> get pods -n cpaas-system
kubectl --kubeconfig <global-kubeconfig> get clustermodule global
```

Common Stalls and Where to Look

Symptom	First place to look	What you are looking for
Machines stay in `Pending` or do not appear	`kubectl describe machine -n cpaas-system <machine>`	The provider-specific failure reason on the machine `Bootstrap` and `Infrastructure` conditions. IaaS quota, network, and credential issues surface here.
`KubeadmControlPlane` does not reach `Ready`	`kubectl get nodes` with the new cluster kubeconfig and `kubectl describe kubeadmcontrolplane -n cpaas-system`	etcd health on the first control plane node and join progress for the remaining nodes.
Pods in `kube-system` stay `Pending` or fail to pull images	`kubectl --kubeconfig <global-kubeconfig> describe pod -n kube-system <pod>`	Image pull errors usually mean the node-facing registry address is not

Symptom	First place to look	What you are looking for
		reachable from the new cluster's subnet.
Installer progress API shows a stalled stage	`/var/cpaas/data/installer.log`	The most recent phase line and the most recent error message. Retried errors repeat on a short interval; persistent errors do not advance.
`ClusterModule/global` does not reach a healthy phase	`kubectl --kubeconfig <global-kubeconfig> describe clustermodule global`	The `Status.conditions` describe which module is blocking the cluster from completing.

Issues that are not listed here usually point to environment-specific causes. Capture the installer log, the progress API response, and the relevant `kubectl describe` output, then escalate.

Optional Disaster Recovery Deployment

Use this section when you deploy primary and standby `global` clusters for disaster recovery. Complete these additions before you apply the provider-specific manifest for each `global` cluster.

Before you begin, complete the bidirectional DR network requirements. Do not start either installation until both cluster VIPs have the required load-balancer listeners and the inter-cluster network rules allow the required traffic in both directions. The direction used during normal operation is standby to primary, but it reverses after failover.

> **Bare Metal Fresh-install Scope**
>
> The Bare Metal split-auth procedure in this section covers only a fresh installation of the primary and standby `global` clusters. It does not define or validate an in-place upgrade or migration of an

existing Bare Metal DR pair, or the repair or recovery of a failed environment. Use a separately validated operations runbook for those lifecycle tasks.

Not every subsection below applies to every provider. Complete the ones marked for your infrastructure:

Subsection	Huawei DCS	VMware vSphere	Huawei Cloud Stack	Bare Metal
Prepare Shared DR Variables	✓	✓	✓	✓
Bare Metal: Prepare Shared ServiceAccount Signing Key	—	—	—	✓
Add DR etcd Server Certificate SANs	✓	✓	✓	✓
Add Provider-Specific DR Fields	✓	✓	✓	✓
Install Primary and Standby Clusters	✓	✓	✓	✓
Bare Metal: Verify Before Decommissioning	—	—	—	✓
Bare Metal: Apply the Rule ConfigMap	—	—	—	✓
Install etcd-sync	✓	✓	✓	✓
Bare Metal: Verify the Dynamic Rule	—	—	—	✓

Subsection	Huawei DCS	VMware vSphere	Huawei Cloud Stack	Bare Metal
Source				
Start and verify synchronization	✓	✓	✓	✓

Applying a Bare Metal-only step to another provider changes objects that provider does not expect — in particular, do not add the Bare Metal `KubeadmControlPlane` signing-key `files` entries to a DCS, VMware vSphere, or Huawei Cloud Stack manifest.

All providers: primary and standby clusters must use the same encryption provider configuration. For DCS and Bare Metal, the provider-specific cluster resource references a Secret that contains `encryption-provider.conf`; for HCS, normal non-DR deployments do not add `/etc/kubernetes/encryption-provider.conf` to `KubeadmControlPlane.spec.kubeadmConfigSpec.files`. VMware vSphere keeps the release manifest's `/etc/kubernetes/encryption-provider.conf` file entry.

Bare Metal only: primary and standby must additionally use the same Kubernetes ServiceAccount signing key, so that the fixed `baremetal-system-agent` token created on the primary cluster is accepted by the standby API server after failover. The machines that form each `global` cluster still use separate cluster-local identities.

Prepare Shared DR Variables

Set the same encryption key value on both the primary and standby installation environments.

```
export ENCRYPTION_PROVIDER_CONF="/root/yamls/encryption-provider.conf"
export ENCRYPTION_PROVIDER_SECRET_B64="<base64-shared-etcd-encryption-key
>"
export PRIMARY_CLUSTER_VIP="<primary-ha-vip>"
export STANDBY_CLUSTER_VIP="<standby-ha-vip>"
export DCS_ENCRYPTION_PROVIDER_SECRET="encryption-provider-config"
export BAREMETAL_ENCRYPTION_PROVIDER_SECRET="global-encryption-provider-c
onfig"
export SERVICE_ACCOUNT_ISSUER="https://kubernetes.default.svc.cluster.loc
al"
```

Create the encryption provider configuration file on both installation environments.

```
mkdir -p "$(dirname "${ENCRYPTION_PROVIDER_CONF}")"
cat > "${ENCRYPTION_PROVIDER_CONF}" <<EOF_CONF
apiVersion: apiserver.config.k8s.io/v1
kind: EncryptionConfiguration
resources:
- resources:
  - secrets
  providers:
  - aescbc:
      keys:
      - name: key1
        secret: ${ENCRYPTION_PROVIDER_SECRET_B64}
EOF_CONF
```

Bare Metal: Prepare Shared ServiceAccount Signing Key

> **Bare Metal only**
>
> Only Bare Metal needs a shared ServiceAccount signing key. Huawei DCS, VMware vSphere, and Huawei Cloud Stack deployments skip this section — they share the encryption provider configuration, but each side keeps its own signing key. Do not add these `files` entries to a non-Bare Metal `KubeadmControlPlane`.

Generate the ServiceAccount signing key once and use the same files in both the primary and standby `KubeadmControlPlane` manifests. The fixed `baremetal-system-agent` token created on the primary cluster is only accepted by the standby API server when both sides sign with the same key.

```
mkdir -p /root/global-dr-sa
openssl genrsa -out /root/global-dr-sa/sa.key 2048
openssl rsa -in /root/global-dr-sa/sa.key -pubout -out /root/global-dr-s
a/sa.pub
chmod 0600 /root/global-dr-sa/sa.key
chmod 0644 /root/global-dr-sa/sa.pub

kubectl -n cpaas-system create secret generic global-sa-signing-key \
  --from-file=sa.key=/root/global-dr-sa/sa.key \
  --from-file=sa.pub=/root/global-dr-sa/sa.pub \
  --dry-run=client -o yaml | kubectl apply -f -
```

Add the following entries to the primary and standby `KubeadmControlPlane.spec.kubeadmConfigSpec`. The file content and the issuer/audience values must be identical on both sides.

```
files:
  - path: /etc/kubernetes/pki/sa.key
    owner: root:root
    permissions: "0600"
    contentFrom:
      secret:
        name: global-sa-signing-key
        key: sa.key
  - path: /etc/kubernetes/pki/sa.pub
    owner: root:root
    permissions: "0644"
    contentFrom:
      secret:
        name: global-sa-signing-key
        key: sa.pub
clusterConfiguration:
  apiServer:
    extraArgs:
      service-account-key-file: /etc/kubernetes/pki/sa.pub
      service-account-signing-key-file: /etc/kubernetes/pki/sa.key
      service-account-issuer: https://kubernetes.default.svc.cluster.loca
l
      api-audiences: https://kubernetes.default.svc.cluster.local
  controllerManager:
    extraArgs:
      service-account-private-key-file: /etc/kubernetes/pki/sa.key
```

After the clusters are installed, verify the files and kubeadm static pod arguments on one control-plane node from each side.

```
sha256sum /etc/kubernetes/pki/sa.key /etc/kubernetes/pki/sa.pub
grep -E 'service-account-issuer|api-audiences|service-account-key-file|se
rvice-account-signing-key-file' \
  /etc/kubernetes/manifests/kube-apiserver.yaml
grep -E 'service-account-private-key-file' \
  /etc/kubernetes/manifests/kube-controller-manager.yaml
```

Add DR etcd Server Certificate SANs to KubeadmControlPlane

In the manifest generated in Step 4, include the control-plane VIP of the cluster you are installing, the platform access address, and `etcd.kube-system` in `KubeadmControlPlane.spec.kubeadmConfigSpec.clusterConfiguration.etcd.local.serverCertSANs`. Each installation environment uses its own list: the primary lists only `${PRIMARY_CLUSTER_VIP}`, and the standby lists only `${STANDBY_CLUSTER_VIP}`. These values configure the etcd server certificate generated by kubeadm; they are independent of the platform `console.cert` and must not be copied into its `thirdParty` SAN list.

Primary installation environment

```
serverCertSANs:
  - "${PRIMARY_CLUSTER_VIP}"
  - "${PLATFORM_HOST}"
  - "etcd.kube-system"
```

Standby installation environment

```
serverCertSANs:
  - "${STANDBY_CLUSTER_VIP}"
  - "${PLATFORM_HOST}"
  - "etcd.kube-system"
```

Add Provider-Specific DR Fields

Huawei DCS

Create the encryption provider Secret in the bootstrap cluster on both installation environments. The Secret must be in the same namespace as `DCSCluster` and must contain a key named `encryption-provider.conf`.

```
kubectl create secret generic "${DCS_ENCRYPTION_PROVIDER_SECRET}" \
  --from-file=encryption-provider.conf="${ENCRYPTION_PROVIDER_CONF}"
\
  -n cpaas-system \
  --dry-run=client -o yaml | kubectl apply -f -
```

Add the Secret reference to `DCSCluster.spec`.

```
encryptionProviderConfigRef:
  name: encryption-provider-config
```

DCS uses `DCSCluster.spec.encryptionProviderConfigRef` to deliver the disaster recovery encryption provider configuration. The DCS provider reads this Secret every time it builds control-plane bootstrap data. Do not add `/etc/kubernetes/encryption-provider.conf` to `KubeadmControlPlane.spec.kubeadmConfigSpec.files` for the DCS DR path.

This Secret must also be included in the DCS `dcs-import-extra-resources` ConfigMap from Step 7. It cannot remain only in the bootstrap cluster because the handed-off `global` cluster keeps the `DCSCluster` object and its `encryptionProviderConfigRef`, and must also have the referenced Secret for later provider reconciliation. The built-in DCS migration flow moves the Cluster API resources but does not copy this Secret.

Create the DCS `dcs-import-extra-resources` ConfigMap on both installation environments, using the DR form of the ConfigMap in Step 7 rather than the normal one. Set `PROVIDER_SECRET_NAME` to the same Secret name used by `DCSCluster.spec.credentialSecretRef.name`, and keep `DCS_ENCRYPTION_PROVIDER_SECRET` aligned with `DCSCluster.spec.encryptionProviderConfigRef.name`.

A missing Secret only fails later, when a control-plane machine is replaced

When `DCSCluster.spec.encryptionProviderConfigRef` is set, the DCS provider requires the referenced Secret and does not fall back to generating or reading a configuration. Reconciles that do not create a control-plane machine are unaffected, so an installation whose Secret stayed in the bootstrap cluster looks healthy until something needs a new control-plane machine — a Kubernetes upgrade, a `KubeadmControlPlane` rolling update, or a control-plane node replacement. The new `Machine` then stays in `Provisioning` with no VM created, and the provider logs `failed to create vm: failed to resolve user data: get encryption provider config secret`.

Check both sides of an existing DR pair, and backfill the Secret if it is missing:

```
kubectl --kubeconfig <global-kubeconfig> -n cpaas-system get dc
scluster global \
  -o jsonpath='{.spec.encryptionProviderConfigRef.name}{"\n"}'
kubectl --kubeconfig <global-kubeconfig> -n cpaas-system get se
cret <name>
```

Recreate it from the same `encryption-provider.conf` the installation used, so new control-plane machines keep the key the running API servers already use. If that file is no longer on the bootstrap node, copy it from a running control-plane node of the same cluster (`/etc/kubernetes/encryption-provider.conf`). A different key breaks decryption of the etcd data the pair already holds.

VMware vSphere

No `VSphereCluster` encryption Secret reference is required. For VMware vSphere, keep this file entry in `KubeadmControlPlane.spec.kubeadmConfigSpec.files` on both the primary and standby installation environments. The rendered `/etc/kubernetes/encryption-provider.conf` content must be identical on both sides, including the provider order, key name, and base64 key value. Also create the VMware vSphere `dcs-import-extra-resources` ConfigMap from Step 7 on both installation environments so the installer imports the vSphere infrastructure resources and the `global-vsphere-credentials` Secret.

```
- path: /etc/kubernetes/encryption-provider.conf
  owner: "root:root"
  append: false
  permissions: "0644"
  content: |
    apiVersion: apiserver.config.k8s.io/v1
    kind: EncryptionConfiguration
    resources:
    - resources:
      - secrets
      providers:
      - aescbc:
          keys:
          - name: key1
            secret: ${ENCRYPTION_PROVIDER_SECRET_B64}
```

Use each installation environment's own DR `serverCertSANs` list, as described in Add DR etcd Server Certificate SANs to KubeadmControlPlane.

Huawei Cloud Stack

No `HCSCluster` encryption Secret reference is required. For HCS, append this file entry to `KubeadmControlPlane.spec.kubeadmConfigSpec.files` on both the primary and standby installation environments. The rendered `/etc/kubernetes/encryption-provider.conf` content must be identical on both sides, including the provider order, key name, and base64 key value.

```
- path: /etc/kubernetes/encryption-provider.conf
  owner: "root:root"
  append: false
  permissions: "0644"
  content: |
    apiVersion: apiserver.config.k8s.io/v1
    kind: EncryptionConfiguration
    resources:
    - resources:
      - secrets
      providers:
      - aescbc:
          keys:
          - name: key1
            secret: ${ENCRYPTION_PROVIDER_SECRET_B64}
```

Use each installation environment's own DR `serverCertSANs` list, as described in Add DR etcd Server Certificate SANs to KubeadmControlPlane.

Create the HCS `dcs-import-extra-resources` ConfigMap from Step 7 on both installation environments. Set `PROVIDER_SECRET_NAME` to the same Secret name used by `HCSCluster.spec.identityRef.name`.

Bare Metal

Create the encryption provider Secret in `minialauda` on both the primary and standby installation environments. The Secret must be in the same namespace as `BaremetalCluster` and must contain a key named `encryption-provider.conf`.

```
kubectl create secret generic "${BAREMETAL_ENCRYPTION_PROVIDER_SECRE
T}" \
  --from-file=encryption-provider.conf="${ENCRYPTION_PROVIDER_CONF}"
\
  -n cpaas-system \
  --dry-run=client -o yaml | kubectl apply -f -
```

Reference that Secret from `BaremetalCluster.spec.encryptionProviderConfigRef`.

```yaml
apiVersion: infrastructure.cluster.x-k8s.io/v1beta1
kind: BaremetalCluster
metadata:
  name: global
  namespace: cpaas-system
spec:
  encryptionProviderConfigRef:
    name: global-encryption-provider-config
```

The bare-metal provider reads this Secret and injects `/etc/kubernetes/encryption-provider.conf` into the generated control-plane bootstrap data. Do not also add the file manually to `KubeadmControlPlane.spec.kubeadmConfigSpec.files` for Bare Metal DR; the `BaremetalCluster` reference is the source of truth.

This Secret must also be included in the Bare Metal `dcs-import-extra-resources` ConfigMap from Step 7. It cannot remain only in the bootstrap cluster because the handed-off `global` cluster keeps the imported `BaremetalCluster` object and must also have the referenced Secret for later provider reconciliation.

Use each installation environment's own DR `serverCertSANs` list, as described in Add DR etcd Server Certificate SANs to KubeadmControlPlane.

Also keep the shared ServiceAccount signing key configuration from Prepare Shared ServiceAccount Signing Key. Without that key, the standby API server cannot validate the `baremetal-system-agent` token that existing hosts received before failover.

Create the Bare Metal `dcs-import-extra-resources` ConfigMap from Step 7 on both installation environments. The ConfigMap must import the durable bare-metal and elemental owner resources required by handoff, including the encryption-provider Secret referenced by `BaremetalCluster.spec.encryptionProviderConfigRef`. It must not import `SeedImage`, Elemental plan Secrets, `KubeadmConfig` objects, or kubeadm bootstrap data Secrets; the fresh-install handoff creates the initial Global plan Secrets only after all target Global `MachineInventory` objects are present.

The Bare Metal provider AppRelease on both sides must enable the Global-local identity, shared workload identity, and direct kube-apiserver handoff for Global machines. Apply one of the following patches on each side's bootstrap cluster after creating the AppRelease and before calling the installer API.

On the primary bootstrap cluster:

```
kubectl -n cpaas-system patch apprelease cluster-api-provider-baremet
al \
  --type=merge \
  -p '{"spec":{"values":{"handoffHook":{"directAPIServer":true},"elem
ental":{"systemAgent":{"authMode":"shared","splitAuthEnabled":true,"s
erviceAccountName":"baremetal-system-agent","globalServiceAccountNam
e":"baremetal-global-system-agent","sharedAuthReadOnly":false}}}}}'
```

On the standby bootstrap cluster:

```
kubectl -n cpaas-system patch apprelease cluster-api-provider-baremet
al \
  --type=merge \
  -p '{"spec":{"values":{"handoffHook":{"directAPIServer":true},"elem
ental":{"systemAgent":{"authMode":"shared","splitAuthEnabled":true,"s
erviceAccountName":"baremetal-system-agent","globalServiceAccountNam
e":"baremetal-global-system-agent","sharedAuthReadOnly":true}}}}}'
```

Verify the effective values on each bootstrap cluster:

```
export EXPECTED_SHARED_AUTH_READ_ONLY=false # Use true on the standby
bootstrap cluster.

kubectl -n cpaas-system get apprelease cluster-api-provider-baremetal
-o json | \
  jq -e --argjson expected_read_only "${EXPECTED_SHARED_AUTH_READ_ONL
Y}" '
    .spec.values.handoffHook.directAPIServer == true and
    .spec.values.elemental.systemAgent.splitAuthEnabled == true and
    .spec.values.elemental.systemAgent.sharedAuthReadOnly == $expecte
d_read_only
  '
```

Do not continue until this command succeeds. The primary must use `sharedAuthReadOnly: false`; the standby must use `sharedAuthReadOnly: true` while primary-to-standby synchronization is active.

The resulting bootstrap AppRelease values are:

```
handoffHook:
  controlPlaneVIP: <current-side-control-plane-vip>
  directAPIServer: true
  delivery:
    enabled: true
    mode: always

elemental:
  systemAgent:
    authMode: shared
    splitAuthEnabled: true
    serviceAccountName: baremetal-system-agent
    globalServiceAccountName: baremetal-global-system-agent
  tls:
    agentTLSMode: strict
    caCertSecretName: dex.tls
    caCertSecretKey: ca.crt
```

Keep the bootstrap and final certificate inputs separate. The bootstrap `dex.tls` created in Step 3 has a dedicated `ca.crt` key and is used only for `https://<bootstrap-host-ip>:12443`. Do not import it into the final cluster. On each final Global cluster, the installer-managed `dex.tls` contains the platform full certificate chain in `tls.crt`; the Bare Metal provider on the final Global cluster must therefore use `elemental.tls.caCertSecretKey: tls.crt`.

The final Global cluster does not inherit the bootstrap release values. When you install the Bare Metal provider on each final Global cluster, set the following values, together with `sharedAuthReadOnly` below, in `spec.valuesOverride` of the provider's `ModuleInfo` under `acp/chart-cluster-api-provider-baremetal`, as described in Install the cluster plugins. Leave `agentTLSMode` and `caCertSecretName` at their defaults (`strict` and `dex.tls`), and do not copy the other bootstrap values, such as `handoffHook.*`:

```
Final Global provider values
```

```
elemental:
  tls:
    caCertSecretKey: tls.crt
  systemAgent:
    splitAuthEnabled: true
```

The final operator uses this chain for its registration endpoint. Direct-API handoff still obtains the kube-apiserver CA from `Secret/cpaas-system/baremetal-global-system-agent-token` key `ca.crt`; do not replace that CA with `dex.tls/tls.crt` in a Global host's direct `kubeconfig`.

`elemental.systemAgent.splitAuthEnabled` is the single switch for both `elemental-operator` and the handoff job. Do not configure a separate handoff auth-enable switch. Keeping one source of truth prevents a Global host from retaining a shared token after the operator has already narrowed the shared Role to workload plans.

Set the shared bundle ownership separately on each side: on the bootstrap release with the patch above, and on each final Global cluster in the provider's `ModuleInfo` (see Install the cluster plugins). Each final Global cluster keeps its own value; DR synchronization does not copy the provider's `ModuleInfo`. The active primary reconciles workload permissions, while the inactive standby leaves the synchronized shared bundle read-only:

Primary active Global

```
elemental:
  systemAgent:
    sharedAuthReadOnly: false
```

Standby inactive Global

```
elemental:
  systemAgent:
    sharedAuthReadOnly: true
```

The installation uses two non-overlapping system-agent permission sets:

- `Role/cpaas-system/baremetal-global-system-agent` contains only plan Secret names for machines that belong to the local `Cluster/global`. Its ServiceAccount and token Secret are local to that `global` cluster and must not be synchronized to the peer.
- `Role/cpaas-system/baremetal-system-agent` contains only non-`global` workload plan Secret names. Its ServiceAccount, token Secret, Role, and RoleBinding are synchronized from primary to standby.

Use namespaced `Role` and `RoleBinding` objects, not `ClusterRoleBinding`. Do not grant namespace-wide Secret access and do not include registry, bootstrap, cluster access credential, or platform credential Secrets.

> **Use the Standard System-Agent Names**
>
> Keep the Global-local objects named `baremetal-global-system-agent` and `baremetal-global-system-agent-token`, and keep the workload shared objects named `baremetal-system-agent` and `baremetal-system-agent-token`. The `etcd-sync` exact-sync and ignore rules use these literal names. Do not customize them unless you also update and validate every corresponding `etcd-sync` rule before synchronization starts.

For this fresh DR installation, each final Global cluster must report `data.ready: "true"` in `ConfigMap/cpaas-system/baremetal-system-agent-handoff`. This is the bootstrap cleanup signal, not the complete acceptance result: also verify the runtime endpoint, token, CA, plan feedback, and RBAC boundaries below. Fresh installation does not use `RoleBinding/cpaas-system/baremetal-global-system-agent-handoff-bridge`; it must be absent. Do not patch the completion signal manually.

Install Primary and Standby Clusters

Run Steps 1 through 9 for both the primary and standby `global` clusters.

> **Use two separate bootstrap nodes**
>
> Use two independent bootstrap nodes on every provider, one for the primary installation and one for the standby. Do not reuse the same bootstrap cluster for both sides: until handoff, the bootstrap

environment carries the installer, the embedded Registry, and the provider resources of the `global` cluster it installs — on Bare Metal additionally AppRelease objects, Registry Secrets, `MachineRegistration`, `SeedImage`, and handoff state. Sharing one bootstrap environment pollutes the two `global` installations and can make handoff or cleanup act on the wrong side.

Use the provider-specific installer configuration differences for both sides:

Provider	Primary installation	Standby install
Huawei DCS	Set `console.host` to the primary control-plane VIP. `cluster.features.ha.vip` uses the same address. Create the DCS `dcs-import-extra-resources` ConfigMap from Step 7 and keep `PROVIDER_SECRET_NAME` aligned with `DCSCluster.spec.credentialSecretRef.name` and `DCS_ENCRYPTION_PROVIDER_SECRET` aligned with `DCSCluster.spec.encryptionProviderConfigRef.name`.	Set `console.h` `cluster.featu` the DCS `dcs-i` Step 7 and keep `DCSCluster.sp` `DCS_ENCRYPTI` `DCSCluster.sp`
VMware vSphere	Set `console.host` to the primary control-plane VIP. `cluster.features.ha.vip` and `VSphereCluster.spec.controlPlaneEndpoint.host` use the same address. Create the VMware vSphere `dcs-import-extra-resources` ConfigMap from Step 7 and keep `global-vsphere-credentials` aligned with `VSphereCluster.spec.identityRef.name`.	Set `console.h` `cluster.featu` `VSphereCluste` the same addre `extra-resourc` `vsphere-creden` `VSphereCluste`
Huawei Cloud Stack	Set `console.host` to the primary control-plane VIP. `HCSCluster.spec.controlPlaneLoadBalancer.vipAddress` uses the same address. Create the HCS `dcs-import-extra-resources` ConfigMap from Step 7 and keep `HCS_SECRET_NAME` aligned with `HCSCluster.spec.identityRef.name`.	Set `console.h` `HCSCluster.sp` uses the same a `resources` Con `HCS_SECRET_NA` `HCSCluster.sp`

Provider	Primary installation	Standby install
Bare Metal	Set `console.host` to the primary control-plane VIP. `cluster.features.ha.vip`, `BaremetalCluster.spec.controlPlaneLoadBalancer.host`, and `handoffHook.controlPlaneVIP` use the same address. Set `handoffHook.directAPIServer: true`, `elemental.systemAgent.splitAuthEnabled: true`, and `elemental.systemAgent.sharedAuthReadOnly: false`. Create the Bare Metal `dcs-import-extra-resources` ConfigMap from Step 7.	Set `console.h` `cluster.featu` `BaremetalClus` and `handoffHo` Set `handoffHo` `elemental.sys` `elemental.sys` Create the Bare ConfigMap from

For the primary cluster, make sure the platform domain resolves to the primary control-plane VIP. In Step 8, set `hostIP` to the primary bootstrap node IP, and set `console.host` and the provider fields listed in the table above to the primary control-plane VIP.

After the primary cluster installation succeeds, switch the platform domain to the standby control-plane VIP as required by the DR procedure. Then install the standby cluster. This DNS switch before the standby installation is required because several platform resources are rendered with the platform domain and must resolve to the standby entrance while the standby installer runs. In Step 8 on the standby bootstrap node, set `hostIP` to the standby bootstrap node IP. Set `console.host` and the provider fields listed in the table above to the standby control-plane VIP. For Bare Metal, keep `REGISTRY_DOMAIN` at `${PLATFORM_HOST}:11443` — the same value the primary installation used. Get `INSTALLER_IP` from the `cpaas-installer` Pod on the standby bootstrap node; do not reuse the primary bootstrap node value.

Bare Metal: Verify Before Decommissioning Either Bootstrap Cluster

> **Bare Metal only**
>
> Everything from here to the end of this subsection applies to the Bare Metal path only. Huawei DCS, VMware vSphere, and Huawei Cloud Stack deployments continue at Install etcd-sync after both installations report success.

Verify the Registry propagation on each final `global` cluster before installing `etcd-sync`. Keep `ProductBase.spec.registry.preferPlatformURL` at the installer-generated value.

Both sides must report the same `${PLATFORM_HOST}:11443` value in `spec.registry.address` and in the two Cluster annotations; that identical value is what lets **Alauda Container Platform Global Cluster Disaster Recovery** replicate these keys without overwriting anything.

```
verify_baremetal_global_registry() {
  kubeconfig=$1
  expected_registry=$2

  test "$(kubectl --kubeconfig "${kubeconfig}" \
    get productbase.product.alauda.io base \
    -o jsonpath='{.spec.registry.address}')" = "${expected_registry}"
  test "$(kubectl --kubeconfig "${kubeconfig}" \
    get cluster.platform.tkestack.io global \
    -o jsonpath='{.metadata.annotations.cpaas\.io/registry-address}')" =
\
    "${expected_registry}"
  test "$(kubectl --kubeconfig "${kubeconfig}" -n cpaas-system \
    get cluster.cluster.x-k8s.io global \
    -o jsonpath='{.metadata.annotations.cpaas\.io/registry-address}')" =
\
    "${expected_registry}"
  test "$(kubectl --kubeconfig "${kubeconfig}" -n cpaas-system \
    get secret public-registry-credential -o jsonpath='{.data.registry}'
| \
    base64 -d)" = "${expected_registry}"

  # Only deployments that enabled Registry authentication have this Secre
t. On an
  # anonymous Registry it does not exist, and checking it unconditionally
aborts
  # the whole function under `set -e` before the remaining assertions ru
n.
  if kubectl --kubeconfig "${kubeconfig}" -n cpaas-system \
    get secret global-registry-auth >/dev/null 2>&1; then
    kubectl --kubeconfig "${kubeconfig}" -n cpaas-system \
      get secret global-registry-auth -o jsonpath='{.data.\.dockerconfigj
son}' | \
      base64 -d | jq -e --arg registry "${expected_registry}" \
        '.auths[$registry] != null' >/dev/null
  fi

  kubectl --kubeconfig "${kubeconfig}" get apprelease -A -o json | \
    jq -e --arg registry "${expected_registry}" '
      all(.items[];
        .spec.source.repoURL == $registry and
        .spec.values.global.registry.address == $registry)
    ' >/dev/null
```

```
}

export GLOBAL_REGISTRY_ADDRESS="${PLATFORM_HOST}:11443"

verify_baremetal_global_registry \
  "${PRIMARY_GLOBAL_KUBECONFIG}" "${GLOBAL_REGISTRY_ADDRESS}"
verify_baremetal_global_registry \
  "${STANDBY_GLOBAL_KUBECONFIG}" "${GLOBAL_REGISTRY_ADDRESS}"
```

Run the complete handoff gate on both sides before decommissioning either bootstrap KIND cluster. `GLOBAL_HOSTS` is the space-separated management address list for that side's installed Global machines. Run this from a secured host that can use both Global kubeconfigs and SSH to the machines. Keep shell tracing disabled because the runtime connection file contains a bearer token.

```
set -euo pipefail
set +x

export PRIMARY_GLOBAL_KUBECONFIG="<path-to-primary-global-kubeconfig>"
export STANDBY_GLOBAL_KUBECONFIG="<path-to-standby-global-kubeconfig>"
export PRIMARY_BOOTSTRAP_KUBECONFIG="<path-to-primary-bootstrap-kubeconfi
g>"
export STANDBY_BOOTSTRAP_KUBECONFIG="<path-to-standby-bootstrap-kubeconfi
g>"
export PRIMARY_GLOBAL_HOSTS="<primary-global-host-1> <primary-global-host
-2> <primary-global-host-3>"
export STANDBY_GLOBAL_HOSTS="<standby-global-host-1> <standby-global-host
-2> <standby-global-host-3>"
export PRIMARY_GLOBAL_REGISTRATION_NAME="<primary-global-machine-registra
tion-name>"
export STANDBY_GLOBAL_REGISTRATION_NAME="<standby-global-machine-registra
tion-name>"

verify_global_auth_scope() {
  kubeconfig=$1
  registration_name=$2

  kubectl --kubeconfig "${kubeconfig}" -n cpaas-system \
    get machineregistration.elemental.cattle.io "${registration_name}" -o
json | \
  jq -e '
    .metadata.annotations["baremetal.cluster.io/system-agent-auth-scope"]
==
      "global"
  ' >/dev/null

  global_inventory_names="$({
    kubectl --kubeconfig "${kubeconfig}" -n cpaas-system \
      get baremetalmachines.infrastructure.cluster.x-k8s.io -o json | \
    jq -c '[
      .items[]
      | select(
          .metadata.labels["cluster.x-k8s.io/cluster-name"] == "global"
        )
      | .status.machineInventoryRef.name? // empty
    ] | unique | sort'
  })"
  printf '%s\n' "${global_inventory_names}" | \
```

```
    jq -e 'length > 0' >/dev/null

  while IFS= read -r inventory_name; do
    kubectl --kubeconfig "${kubeconfig}" -n cpaas-system \
      get machineinventory.elemental.cattle.io "${inventory_name}" -o jso
n | \
    jq -e '
      .metadata.annotations["baremetal.cluster.io/system-agent-auth-scop
e"] ==
        "global"
    ' >/dev/null
  done < <(printf '%s\n' "${global_inventory_names}" | jq -r '.[]')
}

verify_baremetal_handoff() {
  bootstrap_kubeconfig=$1
  kubeconfig=$2
  control_plane_vip=$3
  global_hosts=$4
  registration_name=$5
  expected_endpoint="https://${control_plane_vip}:6443"

  kubectl --kubeconfig "${bootstrap_kubeconfig}" -n cpaas-system \
    wait --for=condition=complete \
    job/baremetal-system-agent-handoff --timeout=30m

  kubectl --kubeconfig "${kubeconfig}" -n cpaas-system \
    get configmap baremetal-system-agent-handoff -o json | \
    jq -e '.data.ready == "true"' >/dev/null

  verify_global_auth_scope "${kubeconfig}" "${registration_name}"

  if kubectl --kubeconfig "${kubeconfig}" -n cpaas-system \
    get rolebinding baremetal-global-system-agent-handoff-bridge \
    >/dev/null 2>&1; then
    echo "temporary handoff bridge still exists" >&2
    return 1
  fi

  kubectl --kubeconfig "${kubeconfig}" -n cpaas-system get \
    serviceaccount/baremetal-global-system-agent \
    secret/baremetal-global-system-agent-token \
    role/baremetal-global-system-agent \
    rolebinding/baremetal-global-system-agent >/dev/null
```

```
global_plan_names="$({
  kubectl --kubeconfig "${kubeconfig}" -n cpaas-system \
    get role baremetal-global-system-agent -o json | \
  jq -c '[
    .rules[]?
    | select(any(.apiGroups[]?; . == ""))
    | select(any(.resources[]?; . == "secrets"))
    | .resourceNames[]?
  ] | unique | sort'
})"
printf '%s\n' "${global_plan_names}" | jq -e 'length > 0' >/dev/null

while IFS= read -r plan_secret; do
  test "$({
    kubectl --kubeconfig "${kubeconfig}" -n cpaas-system \
      get secret "${plan_secret}" -o jsonpath='{.type}'
  })" = "elemental.cattle.io/plan"
done < <(printf '%s\n' "${global_plan_names}" | jq -r '.[]')

expected_token_sha="$({
  kubectl --kubeconfig "${kubeconfig}" -n cpaas-system \
    get secret baremetal-global-system-agent-token \
    -o jsonpath='{.data.token}' | base64 -d | \
  sha256sum | awk '{print $1}'
})"
expected_ca_sha="$({
  kubectl --kubeconfig "${kubeconfig}" -n cpaas-system \
    get secret baremetal-global-system-agent-token \
    -o jsonpath='{.data.ca\.crt}' | base64 -d | \
  sha256sum | awk '{print $1}'
})"

for host in ${global_hosts}; do
  actual_endpoint="$({
    ssh "root@${host}" \
      'cat /var/lib/elemental/agent/elemental_connection.json' | \
    jq -r '.kubeConfig' | \
    awk '$1 == "server:" {print $2; exit}'
  })"
  test "${actual_endpoint}" = "${expected_endpoint}"

  actual_namespace="$({
    ssh "root@${host}" \
```

```
        'cat /var/lib/elemental/agent/elemental_connection.json' | \
      jq -r '.namespace'
    })"
    test "${actual_namespace}" = "cpaas-system"

    actual_plan_secret="$({
      ssh "root@${host}" \
        'cat /var/lib/elemental/agent/elemental_connection.json' | \
      jq -r '.secretName'
    })"
    printf '%s\n' "${global_plan_names}" | \
      jq -e --arg name "${actual_plan_secret}" 'index($name) != null' \
      >/dev/null

    actual_token_sha="$({
      ssh "root@${host}" \
        'cat /var/lib/elemental/agent/elemental_connection.json' | \
      jq -r '.kubeConfig' | \
      awk '$1 == "token:" {print $2; exit}' | \
      tr -d '\r\n' | sha256sum | awk '{print $1}'
    })"
    test "${actual_token_sha}" = "${expected_token_sha}"

    actual_ca_sha="$({
      ssh "root@${host}" \
        'cat /var/lib/elemental/agent/elemental_connection.json' | \
      jq -r '.kubeConfig' | \
      awk '$1 == "certificate-authority-data:" {print $2; exit}' | \
      base64 -d | sha256sum | awk '{print $1}'
    })"
    test "${actual_ca_sha}" = "${expected_ca_sha}"

    ssh "root@${host}" \
      "grep -Fq -- '${expected_endpoint}' /oem/elemental-system-agent.yam
l"
    ssh "root@${host}" \
      'test "$(stat -c %a /oem/elemental-system-agent.yaml)" = 600'
    ssh "root@${host}" \
      'grep -Fqx -- "CATTLE_AGENT_STRICT_VERIFY=\"true\"" /etc/rancher/el
emental/agent/envs'
    ssh "root@${host}" \
      'systemctl is-active --quiet elemental-system-agent.service'
  done
}
```

```
verify_baremetal_handoff \
  "${PRIMARY_BOOTSTRAP_KUBECONFIG}" "${PRIMARY_GLOBAL_KUBECONFIG}" \
  "${PRIMARY_CLUSTER_VIP}" \
  "${PRIMARY_GLOBAL_HOSTS}" "${PRIMARY_GLOBAL_REGISTRATION_NAME}"
verify_baremetal_handoff \
  "${STANDBY_BOOTSTRAP_KUBECONFIG}" "${STANDBY_GLOBAL_KUBECONFIG}" \
  "${STANDBY_CLUSTER_VIP}" \
  "${STANDBY_GLOBAL_HOSTS}" "${STANDBY_GLOBAL_REGISTRATION_NAME}"
```

Also verify the Global-local permission boundary on both sides. Before the first non-`global` workload plan exists, the shared `baremetal-system-agent` ServiceAccount, token Secret, Role, and RoleBinding may be absent on both clusters. Do not create or copy that bundle by hand merely to satisfy the Global installation gate; the active operator creates it when the first shared-scope registration is reconciled, and `etcd-sync` then copies it to standby.

```
verify_global_local_system_agent_rbac() {
  kubeconfig=$1

  kubectl --kubeconfig "${kubeconfig}" -n cpaas-system get \
    serviceaccount/baremetal-global-system-agent \
    secret/baremetal-global-system-agent-token \
    role/baremetal-global-system-agent \
    rolebinding/baremetal-global-system-agent >/dev/null

  local_plan_names="$({
    kubectl --kubeconfig "${kubeconfig}" -n cpaas-system \
      get role baremetal-global-system-agent -o json | \
    jq -c '[
      .rules[]?
      | select(any(.apiGroups[]?; . == ""))
      | select(any(.resources[]?; . == "secrets"))
      | .resourceNames[]?
    ] | unique | sort'
  })"
  printf '%s\n' "${local_plan_names}" | jq -e 'length > 0' >/dev/null

  expected_local_plan_names="$({
    kubectl --kubeconfig "${kubeconfig}" -n cpaas-system \
      get machineinventories.elemental.cattle.io -o json | \
    jq -c '[
      .items[]
      | select(
          .metadata.annotations["baremetal.cluster.io/system-agent-auth-s
cope"] ==
            "global"
        )
      | .status.plan.secretRef.name? // empty
    ] | unique | sort'
  })"
  test "${local_plan_names}" = "${expected_local_plan_names}"

  while IFS= read -r plan_secret; do
    test "$({
      kubectl --kubeconfig "${kubeconfig}" -n cpaas-system \
        get secret "${plan_secret}" -o jsonpath='{.type}'
    })" = "elemental.cattle.io/plan"

    test "$({
```

```
      kubectl --kubeconfig "${kubeconfig}" \
        --as=system:serviceaccount:cpaas-system:baremetal-global-system-a
gent \
        -n cpaas-system auth can-i get "secret/${plan_secret}"
    })" = yes
    test "$({
      kubectl --kubeconfig "${kubeconfig}" \
        --as=system:serviceaccount:cpaas-system:baremetal-global-system-a
gent \
        -n cpaas-system auth can-i patch "secret/${plan_secret}"
    })" = yes
    test "$({
      kubectl --kubeconfig "${kubeconfig}" \
        --as=system:serviceaccount:cpaas-system:baremetal-system-agent \
        -n cpaas-system auth can-i get "secret/${plan_secret}"
    })" = no
  done < <(printf '%s\n' "${local_plan_names}" | jq -r '.[]')

  for verb in list create delete; do
    test "$({
      kubectl --kubeconfig "${kubeconfig}" \
        --as=system:serviceaccount:cpaas-system:baremetal-global-system-a
gent \
        -n cpaas-system auth can-i "${verb}" secrets
    })" = no
  done

  test "$({
    kubectl --kubeconfig "${kubeconfig}" \
      --as=system:serviceaccount:cpaas-system:baremetal-global-system-age
nt \
      -n cpaas-system auth can-i get secret/global-registry-auth
  })" = no
}

verify_global_local_system_agent_rbac "${PRIMARY_GLOBAL_KUBECONFIG}"
verify_global_local_system_agent_rbac "${STANDBY_GLOBAL_KUBECONFIG}"
```

Do not reduce acceptance to the `ready` signal alone. A runtime token or CA mismatch, a stale bootstrap endpoint in the persistent OEM file, a failed plan probe, or an incorrect permission boundary means fresh handoff is incomplete even when `data.ready` is `true`.

After the standby installation succeeds, switch the platform domain back to the primary entrance. Primary is the active source during the first synchronization.

Bare Metal: Apply the Rule ConfigMap

> **Bare Metal only**
>
> Only Bare Metal needs a custom `etcd-sync` rule source. Huawei DCS, VMware vSphere, and Huawei Cloud Stack deployments skip this section and go straight to Install etcd-sync, where the install order does not matter.

Apply this ConfigMap **before** you install `etcd-sync`, on both the primary and the standby.

Apply the same `baremetal-dr-rules` ConfigMap to the primary and standby. Use the template in Bare Metal etcd-sync Rules. Replace the plan placeholders with every current Global plan Secret name from both clusters. Read the names from `MachineInventory.status.plan.secretRef.name` **after handoff has completed on both sides** — fresh handoff creates the Global plan Secrets, so names collected earlier are stale. The plugin chart supplies the remaining Bare Metal rules. Update this ConfigMap whenever a Global plan Secret is added or replaced.

> **Create the rule source before the mirror can start**
>
> Installing `etcd-sync` with `active_cluster_vip` and the active-cluster token already set opens the sync path immediately. If the Bare Metal rule source does not exist yet, the first synchronization runs without the Global plan Secret exclusions and overwrites the standby plan Secrets with the primary ones — the failure mode listed as `A Global plan is overwritten` in Troubleshooting. The standby Global hosts would then receive plans that belong to the primary cluster.
>
> Creating the ConfigMap first has no side effect: until the plugin is installed it is an ordinary labeled ConfigMap that nothing reads. If you must install the plugin first, leave `active_cluster_vip` unset and set it only after the rule source reports `accepted`.

Install etcd-sync (ACP 4.4.0+)

Install the v4.4.0-or-later plugin on the current standby Global only. Do not install or run the mirror on both sides at the same time.

Before installing, switch the platform domain back to the primary control-plane VIP — it was pointed at the standby for the standby installation. The primary is the active source of the first synchronization, and on startup the plugin resolves the platform domain and refuses to start when the cluster the domain points to runs `etcd-sync` itself; with the domain still on the standby, the installation you are about to perform would trip that check.

The plugin package is published as `etcd-sync`, not `global-etcd-sync`: download `packages/etcd-sync/<minor>/etcd-sync.amd64.<version>.tgz` from the package server. It is not part of the installer bundle and is not present in the bootstrap Registry, so upload it to the target cluster like any other plugin package.

Configure it with:

Parameter	Value
Active Global Cluster VIP	Control-plane VIP of the active `global` cluster
Active Global Cluster ETCD Endpoints	`https://<active-control-plane-vip>:2379`, unless a load balancer forwards port `2379`
Standby Cluster ETCD Endpoints	Default value
Active Global Cluster Token Secret	`etcd-sync-active-cluster-token`
Data Check Interval	Default value
Print detail logs	Disabled unless troubleshooting

Create the token Secret on the standby before installing. The value is the bearer token of the **active** cluster's `cpaas-system/k8sadmin` ServiceAccount:

```
# On the active cluster:
kubectl -n cpaas-system get secret k8sadmin -o jsonpath='{.data.token}' |
base64 -d

# On the standby cluster, with the value copied from above:
ACTIVE_CLUSTER_TOKEN='<paste-the-token-from-the-active-cluster>'
kubectl -n cpaas-system create secret generic etcd-sync-active-cluster-to
ken \
  --from-literal=token="${ACTIVE_CLUSTER_TOKEN}" \
  --dry-run=client -o yaml | kubectl apply -f -
```

During installation the `etcd-sync-bootstrap` Job runs before the `etcd-sync` Deployment starts. The plugin installation continues only after the Job prepares `remote-etcd-ca`, `remote-etcd-issuer`, and `remote-etcd-client`.

Bare Metal: Verify the Dynamic Rule Source

Before opening the sync path, confirm that the rule source reports `accepted` and appears in `ConfigMap/etcd-sync-rule-snapshot`:

```
kubectl --kubeconfig <standby-kubeconfig> -n cpaas-system \
  get configmap baremetal-dr-rules \
  -o jsonpath='{.metadata.annotations.etcd-sync\.cpaas\.io/status}{"\n"}'
kubectl --kubeconfig <standby-kubeconfig> -n cpaas-system \
  get configmap etcd-sync-rule-snapshot \
  -o jsonpath='{.data.snapshot\.json}'
```

A rejected source is not active.

Start and verify synchronization

Trigger one etcd-sync monitor check. Retry HTTP 425 until a check is allowed, then require zero missed and zero surplus keys. Confirm the shared workload system-agent bundle is either absent on both sides or complete on both sides; partial presence is a failure.

Do not use AppRelease readiness alone as the DR acceptance result. For the failover sequence and post-failover tests, continue with Global Cluster Disaster Recovery.

Restart the Pods that must reload DR and endpoint configuration. Run the same commands on a primary control plane node and on a standby control plane node.

```
sudo kubectl delete po -n cpaas-system -l 'service_name in (alertmanager,vmselect,vminsert)'
sudo kubectl delete po -n cpaas-system -l service_name=cpaas-elasticsearch
sudo kubectl delete po -n cpaas-system -l service_name=cluster-transformer
```

For the DR lifecycle after installation, see Global Cluster Disaster Recovery.

Verification

After the installer reports completion, verify that the `global` cluster is healthy as described in Validation, including its checks for Alauda OS.

If the `global` cluster was installed on Bare Metal, complete Prepare the global Cluster as a Management Cluster next. It is required before you decommission the bootstrap cluster, and reprovisioning a `global` node after handoff pulls the base image that it uploads to the `global` Registry.

Decommission the Bootstrap Cluster

After the installer reports success, and, on Bare Metal, after you complete Prepare the global Cluster as a Management Cluster, the `global` cluster no longer depends on the bootstrap cluster. Once the general verification and the provider-specific handoff gate below pass, remove the temporary bootstrap cluster from the bootstrap node — delete only the local bootstrap cluster (`minialauda`) and its KIND container network.

> **Bare Metal: Require the Final Handoff Gate**
>
> Do not remove `minialauda` merely because `KubeadmControlPlane` is Ready, the installer reports success, or the `baremetal-system-agent-handoff` Job is `Complete`. Each imported

Global host must complete the probe from the final `global` cluster before the bootstrap endpoint can be retired.

Read the final handoff record from the new `global` cluster:

```
kubectl --kubeconfig <global-kubeconfig> -n cpaas-system \
  get configmap baremetal-system-agent-handoff \
  -o jsonpath='{.data.ready}{"\n"}'
```

The completion ConfigMap carries exactly one key, `ready`. It is the installer cleanup gate and nothing more: it records that the handoff and the global-side probe finished, and it says nothing about the endpoint, the TLS inputs, or the identity each host ended up with. Do not patch `data.ready` by hand, and do not treat it on its own as proof that the authentication split is correct — the checks below are what establish that.

Continue only when all of the following hold:

- The current installer handoff hook ran the Job with the deployed provider configuration, and the Job completed successfully.
- `data.ready` is `true` on the final `global` cluster.
- Every imported target `MachineInventory` reports `status.plan.state: Applied` on the final `global` cluster. This is the global-side probe result, and it is the check that proves each host now reports to the new cluster.
- Every imported target object, and the plan Secret each one references, exists in the final cluster.
- The system-agent identities on the final cluster match the mode you deployed, and the temporary cross-identity RoleBinding is gone.

Check the last three conditions with:

```
kubectl --kubeconfig <global-kubeconfig> -n cpaas-system \
  get machineinventories.elemental.cattle.io \
  -o jsonpath='{range .items[*]}{.metadata.name}{"\t"}{.status.pla
n.state}{"\n"}{end}'

kubectl --kubeconfig <global-kubeconfig> -n cpaas-system \
  get serviceaccount,secret,role,rolebinding \
  | grep -E 'baremetal-(global-)?system-agent'

kubectl --kubeconfig <global-kubeconfig> -n cpaas-system \
  get rolebinding baremetal-global-system-agent-handoff-bridge
```

With `elemental.systemAgent.splitAuthEnabled: false` the final cluster carries only the shared `baremetal-system-agent` identity. With `true` it also carries the Global-local `baremetal-global-system-agent` identity, and the two Roles must not name the same plan Secret. The last command must report `NotFound`.

If the current Job failed or timed out, the target set is incomplete, a referenced plan Secret is missing, or a probe plan is not `Applied`, keep `minialauda` running and investigate the handoff Job and provider logs.

Verify the DCS Secrets reached the global cluster

The DCS API credential `Secret` is copied to the `global` cluster during installation by the `dcs-import-extra-resources` ConfigMap you create in Step 7 — it imports the Secret named in your `DCSCluster.spec.credentialSecretRef`. Before you remove the bootstrap node, verify it is present: `kubectl --kubeconfig <global-kubeconfig> get secret <name> -n cpaas-system`. If it is missing, copy it over first — without it the DCS provider that you install on the `global` cluster has no DCS API credentials and cannot reconcile (for example, scale-out later fails).

For a DR installation, run the same check for the Secret named in `DCSCluster.spec.encryptionProviderConfigRef`. The `DCSCluster` object survives the handoff with that reference, so once the bootstrap cluster is gone a missing Secret makes every later control-plane machine creation fail — see Add Provider-Specific DR Fields.

Do not delete the Cluster API objects to clean up

Do not run `kubectl delete cluster global`, and do not delete the `Cluster`, `KubeadmControlPlane`, or provider infrastructure objects as a cleanup step. After installation these objects own the live `global` control plane machines, so deleting them cascades into deleting the control plane VMs and destroys the cluster you just installed. Decommissioning is limited to removing the local bootstrap cluster (its KIND container) on the bootstrap node; leave the Cluster API objects in place.

Next Steps

- Before you create workload clusters, upload and install the infrastructure provider for the target IaaS platform and the Cluster Bootstrap Provider on the `global` cluster, regardless of which IaaS platform the `global` cluster runs on: see Prepare. In a DR pair, do this on both `global` clusters.
- Configure infrastructure resources for workload clusters: see Infrastructure Resources.
- Create your first workload cluster: see Create on Alauda OS.
- Plan disaster recovery: see Global Cluster Disaster Recovery.

Worked Example: Complete `global` Manifest for Huawei DCS

This is a complete, single-file manifest for a three-replica control-plane `global` cluster on Huawei DCS. It is the same set of resources described in Step 4, already assembled so you do not have to merge fragments across pages. It uses documentation-only example values: replace every `<placeholder>`, and reuse the `${...}` variables you exported in Step 1. Apply it in Step 5.

This example targets a non-DR cluster. To avoid maintaining two copies of it, the `KubeadmControlPlane` `kubeadmConfigSpec` body is not repeated here — it is identical to a workload cluster and is taken from the Complete KubeadmControlPlane Configuration appendix, with the two `global` / non-DR deltas noted inline in resource 3 below.

The manifest does not create the DCS API credential `Secret`. That Secret is created in the prerequisites above and referenced here by name through `DCSCluster.spec.credentialSecretRef`. Step 7 copies it into the `global` cluster through the `dcs-import-extra-resources` ConfigMap, whose `names` list you must keep aligned with the same Secret name.

Before You Apply: Prepare These on DCS

The manifest references these but does not create them. Prepare them first, in order:

1. **DCS API access, and the credential Secret created from it** — the endpoint (`https://<host>:7443`), an `administrator` user and password, and the site ID. All four go into the credential Secret; the provider reads the site ID from there and writes it to `DCSCluster.spec.site` when that field is left empty, so the manifest below does not set it.
   Create the Secret named `${PROVIDER_SECRET_NAME}` in `cpaas-system` by following Cloud Credentials. Do not author it as YAML and apply it: `kubectl apply` records the password in the Secret's `kubectl.kubernetes.io/last-applied-configuration` annotation.
   The `global` install additionally expects the cluster-name label that every other resource in the manifest carries:

   ```
   kubectl label secret "${PROVIDER_SECRET_NAME}" --namespace cpaas-system
   \
     --overwrite cpaas.io/cluster-name=global
   ```

2. **A VM template** — upload the Alauda OS image and create a VM template from it; record its name for `vmTemplateName`. Use a `4.2.1+` template so the `/var/cpaas` persistent disk can be detached and reattached during node replacement. See Machine Templates.
3. **Compute cluster, distributed virtual switch, port group, and datastore** — pick the target DCS compute cluster, a distributed virtual switch, a port group, and a datastore with enough free capacity. They fill `resource`, `dvSwitchName`, `portGroupName`, and `datastoreName`. See DCS Platform Capacity and Placement.
4. **IPs and the API endpoint** — prepare three free node IPs (with gateway and DNS) for the control plane; these fill the `DCSIpHostnamePool`. Then choose either an external LoadBalancer that follows the control plane endpoint contract and forwards TCP `6443` to

the three nodes, or a separate IPv4 VIP in the same Layer 2 network for Self-built VIP; the selected endpoint host and port fill `DCSCluster.spec.controlPlaneLoadBalancer` and `controlPlaneEndpoint`. See Create a Cluster on Huawei DCS.

5. **Versions and IDs to read** — `${K8S_VERSION}` plus the CoreDNS and etcd image tags from the `cpaas.io/dcs-vm-template` ConfigMap (see Resolving Placeholder Values); the kube-ovn chart version from the OS Support Matrix; and the registry address, VIP, and CIDR values you exported in Step 1.

With those prepared, apply the manifest below. The worked example uses an external LoadBalancer. If you select `Self-built VIP`, apply the internal-mode replacement shown after the manifest before creating the `global` cluster.

```
---
# 1. Control-plane IP / hostname pool (one entry per control-plane replic
a).
apiVersion: infrastructure.cluster.x-k8s.io/v1beta1
kind: DCSIpHostnamePool
metadata:
  name: global-cp-pool
  namespace: cpaas-system
  labels:
    cpaas.io/cluster-name: "global"
spec:
  pool:
  # /var/cpaas holds platform state and must survive node replacement, so
it is
  # declared here as a persistentDisk bound to the IP slot (not as a
  # DCSMachineTemplate disk). Requires a DCS VM template 4.2.1+ and maxSu
rge: 0.
  - ip: "192.0.2.11"
    mask: "24"
    gateway: "192.0.2.1"
    dns: "192.0.2.2"
    hostname: "global-cp-1"
    machineName: "global-cp-1"
    persistentDisk:
    - {slot: 0, quantityGB: 100, datastoreName: <datastore-name>, path: /
var/cpaas, format: xfs, mountOptions: [defaults]}
  - ip: "192.0.2.12"
    mask: "24"
    gateway: "192.0.2.1"
    dns: "192.0.2.2"
    hostname: "global-cp-2"
    machineName: "global-cp-2"
    persistentDisk:
    - {slot: 0, quantityGB: 100, datastoreName: <datastore-name>, path: /
var/cpaas, format: xfs, mountOptions: [defaults]}
  - ip: "192.0.2.13"
    mask: "24"
    gateway: "192.0.2.1"
    dns: "192.0.2.2"
    hostname: "global-cp-3"
    machineName: "global-cp-3"
    persistentDisk:
    - {slot: 0, quantityGB: 100, datastoreName: <datastore-name>, path: /
```

```
var/cpaas, format: xfs, mountOptions: [defaults]}
---
# 2. Control-plane VM spec.
apiVersion: infrastructure.cluster.x-k8s.io/v1beta1
kind: DCSMachineTemplate
metadata:
  name: global-cp-template
  namespace: cpaas-system
  labels:
    cpaas.io/cluster-name: "global"
spec:
  template:
    spec:
      vmTemplateName: <vm-template-name>
      # Places the cloned VMs in a DCS compute cluster.
      resource:
        type: cluster
        name: <dcs-cluster-name>
      vmConfig:
        dvSwitchName: <dvswitch-name>
        portGroupName: <port-group-name>
        dcsMachineCpuSpec: {quantity: 16}
        dcsMachineMemorySpec: {quantity: 32768}   # MB
        dcsMachineDiskSpec:
        - {quantity: 0,   datastoreName: <datastore-name>, systemVolume:
true}
        - {quantity: 10,  datastoreName: <datastore-name>, path: /var/li
b/etcd,       format: xfs}
        - {quantity: 100, datastoreName: <datastore-name>, path: /var/li
b/kubelet,    format: xfs}
        - {quantity: 100, datastoreName: <datastore-name>, path: /var/li
b/containerd, format: xfs}
        # /var/cpaas is intentionally NOT a template disk — it is declare
d as a
        # persistentDisk on the IP pool above so it survives node replace
ment.
      ipHostPoolRef:
        name: global-cp-pool
---
# 3. Control plane.
apiVersion: controlplane.cluster.x-k8s.io/v1beta1
kind: KubeadmControlPlane
metadata:
  name: global-kcp
```

```
  namespace: cpaas-system
  labels:
    cpaas.io/cluster-name: "global"
  annotations:
    controlplane.cluster.x-k8s.io/skip-kube-proxy: ""
spec:
  replicas: 3
  version: ${K8S_VERSION}
  rolloutStrategy:
    type: RollingUpdate
    rollingUpdate: {maxSurge: 0}
  machineTemplate:
    nodeDrainTimeout: 1m
    nodeDeletionTimeout: 5m
    infrastructureRef:
      apiVersion: infrastructure.cluster.x-k8s.io/v1beta1
      kind: DCSMachineTemplate
      name: global-cp-template
  kubeadmConfigSpec:
    format: ignition
    users:
    - name: boot
      sshAuthorizedKeys:
      - "ssh-ed25519 AAAA...replace-with-your-public-key... global-boot"
    # The rest of kubeadmConfigSpec — files, clusterConfiguration,
    # preKubeadmCommands, postKubeadmCommands, initConfiguration,
    # joinConfiguration — is identical to a workload cluster, so it is no
t
    # duplicated here. Take it verbatim from the Complete KubeadmControlP
lane
    # Configuration appendix in the DCS create-cluster guide
    # (#complete-kubeadmcontrolplane-configuration). The three large file
s
    # (psa-config.yaml, control-plane-kubelet-patch.json, audit-policy.ya
ml) may
    # use contentFrom the dcs-kubernetes-<major.minor>-files Secret. Appl
y these
    # global / non-DR deltas to that body:
    #   1. Add clusterConfiguration.etcd.local.serverCertSANs:
    #        ["${CONTROL_PLANE_VIP}", "${PLATFORM_HOST}"]
    #        A DR pair uses a different list — see Optional Disaster Reco
very
    #        Deployment. etcd.kube-system belongs to that list, not this
one.
```

```
    #   2. Omit the /etc/kubernetes/encryption-provider.conf files entry: the
    #        provider writes that file on control-plane nodes and drops any
    #        files entry for it. Keep apiServer.extraArgs.encryption-provider-
    #        config — see the note after this example.
---
# 4. DCS infrastructure cluster.
apiVersion: infrastructure.cluster.x-k8s.io/v1beta1
kind: DCSCluster
metadata:
  name: "global"
  namespace: cpaas-system
  labels:
    cpaas.io/cluster-name: "global"
  annotations:
    cpaas.io/registry-address: "${NODE_REGISTRY_ADDRESS}"
spec:
  controlPlaneLoadBalancer: {host: "${CONTROL_PLANE_VIP}", port: 6443, type: external}
  controlPlaneEndpoint: {host: "${CONTROL_PLANE_VIP}", port: 6443}
  credentialSecretRef: {name: "${PROVIDER_SECRET_NAME}"}   # created in the prerequisites above
  networkType: kube-ovn
  site: <dcs-site-id>
---
# 5. Top-level CAPI Cluster: global wiring, labels, and annotations.
apiVersion: cluster.x-k8s.io/v1beta1
kind: Cluster
metadata:
  name: global
  namespace: cpaas-system
  labels:
    cpaas.io/cluster-name: "global"
    cluster-type: DCS
    is-global: "true"
  annotations:
    capi.cpaas.io/resource-group-version: infrastructure.cluster.x-k8s.io/v1beta1
    capi.cpaas.io/resource-kind: DCSCluster
    capi.cpaas.io/kubernetes: ${K8S_VERSION}   # same value as KubeadmControlPlane.spec.version
    cpaas.io/registry-address: "${NODE_REGISTRY_ADDRESS}"
```

```
    cpaas.io/nodes-mode: self-managed   # node lifecycle managed by CAPI
+ the DCS provider
    cpaas.io/kube-ovn-join-cidr: <kube-ovn-join-cidr>   # a /16 you choos
e; must not overlap the pod / service CIDRs or another cluster's join CID
R
    cpaas.io/kube-ovn-version: <kube-ovn-chart-version>
spec:
  clusterNetwork:
    pods:     {cidrBlocks: ["${CLUSTER_CIDR}"]}
    services: {cidrBlocks: ["${SERVICE_CIDR}"]}
  controlPlaneRef:
    apiVersion: controlplane.cluster.x-k8s.io/v1beta1
    kind: KubeadmControlPlane
    name: global-kcp
  infrastructureRef:
    apiVersion: infrastructure.cluster.x-k8s.io/v1beta1
    kind: DCSCluster
    name: global
```

Internal Self-built VIP variant

The worked example above uses `type: external`. Before applying it, you can replace resource 4's endpoint fields with this internal configuration. The VIP must be reserved in the control-plane Layer 2 network, and the `vrid` must be unique in that network:

```
spec:
  controlPlaneLoadBalancer:
    host: "${CONTROL_PLANE_VIP}"
    port: 6443
    type: internal
    vrid: <unique-vrid-1-255>
    # interface: eth0  # optional; omit for provider/alive auto-detection
  controlPlaneEndpoint: {host: "${CONTROL_PLANE_VIP}", port: 6443}
```

Values to Replace

Placeholder / variable	Where it comes from
`${K8S_VERSION}`, `${CONTROL_PLANE_VIP}`,	Exported in Step 1.

Placeholder / variable	Where it comes from
`${PLATFORM_HOST}`, `${NODE_REGISTRY_ADDRESS}`, `${CLUSTER_CIDR}`, `${SERVICE_CIDR}`, `${PROVIDER_SECRET_NAME}`	
`<dcs-api-user>` / `<dcs-api-password>` / `<dcs-api-host>` / `<dcs-site-id>`	DCS platform credentials and site. They do not appear in the manifest — they go into the credential file used to create `${PROVIDER_SECRET_NAME}` in the prerequisites. See Cloud Credentials.
`<vm-template-name>`, `<dns-image-tag>`, `<etcd-image-tag>`	The `cpaas.io/dcs-vm-template` ConfigMap. See Resolving Placeholder Values.
`<dcs-cluster-name>`, `<dvswitch-name>`, `<port-group-name>`, `<datastore-name>`	DCS platform objects. Confirm with the DCS administrator.
`192.0.2.x`, `<kube-ovn-join-cidr>`, `<kube-ovn-chart-version>`	Node IPs/gateway/DNS for your network; a `/16` Kube-OVN join CIDR that does not overlap the pod / service CIDRs or another cluster's join CIDR; and the kube-ovn chart version from the OS Support Matrix.
`ssh-ed25519 AAAA...`	A real OpenSSH public key. The `ignition` format rejects an empty list; supply any valid key even if you do not plan to SSH in.

Secret encryption and disaster recovery

This example runs with etcd secret encryption-at-rest enabled. `apiServer.extraArgs.encryption-provider-config` stays in the `clusterConfiguration` you copy from the appendix, and the DCS provider supplies the file it points at: a random key generated for the first control-plane machine, then the running API server's own file for every later one. Do not add `/etc/kubernetes/encryption-provider.conf` to `KubeadmControlPlane.spec.kubeadmConfigSpec.files` — the provider drops any entry for that path. See How the Encryption Provider Configuration Is Delivered.

Remove the argument only if you deliberately want to run without at-rest encryption. For a DR pair the key must be identical on both sides, which means pinning it with

`DCSCluster.spec.encryptionProviderConfigRef` as described in Optional Disaster Recovery Deployment.

Global Cluster Disaster Recovery

This page is the disaster recovery entry point for a `global` cluster whose nodes run Alauda OS. Deploy-time disaster recovery configuration is part of the `global` cluster installation procedure.

TOC

- Deployment Procedure
- Network Requirements
- Observability Placement
- Operational Scope
- Failover
- Bare Metal DR Model
  - Endpoint and Identity
  - Synchronization Rules
  - Start and Verify
  - Acceptance
  - Troubleshooting
- Provider Notes
- See Also

Deployment Procedure

Use Optional Disaster Recovery Deployment in the installation guide when you create the primary and standby `global` clusters.

That deployment procedure is the authoritative source for the installation-time DR configuration, including:

- Each side is installed from its own bootstrap node: two bootstrap nodes in total, on every provider.
- Primary and standby clusters use the same Kubernetes API server encryption provider configuration.
- Each cluster's `KubeadmControlPlane` etcd server certificate SAN list includes that cluster's own control plane VIP, the Platform Access Address, and `etcd.kube-system`; the primary and standby lists differ (see Add DR etcd Server Certificate SANs to KubeadmControlPlane). This list is separate from the platform `console.cert` SAN list.
- Huawei DCS deployments reference the shared encryption provider Secret from `DCSCluster.spec.encryptionProviderConfigRef`. The Secret must exist in each final `global` cluster, not only in the bootstrap cluster that installed it.
- VMware vSphere deployments write the same `/etc/kubernetes/encryption-provider.conf` file through `KubeadmControlPlane.spec.kubeadmConfigSpec.files`.
- Huawei Cloud Stack deployments write the same `/etc/kubernetes/encryption-provider.conf` file through `KubeadmControlPlane.spec.kubeadmConfigSpec.files`.
- Bare Metal deployments reference the shared encryption provider Secret from `BaremetalCluster.spec.encryptionProviderConfigRef` and use the same Kubernetes ServiceAccount signing key on both sides.
- Huawei DCS, VMware vSphere, and Huawei Cloud Stack create the `dcs-import-extra-resources` ConfigMap before installer import so the installation can import referenced credential Secrets and, where applicable, provider-specific infrastructure resources. The name keeps the `dcs` prefix for historical installer compatibility.
- Bare Metal creates the `dcs-import-extra-resources` ConfigMap before installer import so the installation can preserve the durable bare-metal and elemental owner resources required by handoff. It does not import Global plan Secrets or kubeadm bootstrap data

Secrets; fresh handoff creates the initial Global plan Secrets after all target Global `MachineInventory` objects exist.

- The standby cluster installs `global-etcd-sync` with connection values that point to the active primary after both installations succeed.

After both clusters are installed, install the infrastructure provider and the Cluster Bootstrap Provider on **both** `global` clusters, so that the standby can reconcile the workload clusters after a failover. See Huawei DCS, VMware vSphere, Huawei Cloud Stack, or Bare Metal.

Network Requirements

A stable Platform Access Address that uses a DNS name is a prerequisite for disaster recovery. During normal operation, the domain resolves only to the primary `global` cluster VIP. The standby `global` cluster uses the domain to access platform services on the primary `global` cluster.

Before installing either cluster, complete both of the following network preparations:

- Configure each cluster's load balancer to forward the required TCP ports on its VIP to the control plane nodes behind that VIP. Configure port `80` only when users access the platform over HTTP.
- Allow the required TCP ports in **both directions** between the primary and standby cluster networks. Apply the rules to firewalls, security groups, router ACLs, and any other inter-site network controls.

TCP port	Service	Why it is required
`443`	ACP platform HTTPS	The standby cluster uses the Platform Access Address to reach the active platform. **Alauda Container Platform Global Cluster Disaster Recovery** uses this address, and logging and monitoring components send alert callbacks to it.
`80`	ACP platform HTTP	Required only when users access the platform over HTTP. The connectivity requirement is otherwise the same as for port `443`.

TCP port	Service	Why it is required
`6443`	Kubernetes API server	During installation or reinstallation of **Alauda Container Platform Global Cluster Disaster Recovery**, the standby cluster connects to the active cluster API server to obtain the etcd CA material required for synchronization.
`11443`	Built-in image registry	The standby cluster pulls platform and plugin images from the registry configured through the Platform Access Address. This applies to the platform built-in Registry only.
`2379`	etcd	**Alauda Container Platform Global Cluster Disaster Recovery** on the standby cluster reads etcd data from the primary cluster and writes it to the local standby etcd.

The network policy must be bidirectional because the cluster roles are reversed after failover. Before failover, the effective service traffic is normally from standby to primary. After failover, the former primary becomes the new standby and must reach the new primary on the same ports. This requirement does not make etcd replication bidirectional: **Alauda Container Platform Global Cluster Disaster Recovery** runs only on the current standby cluster and writes synchronized data to its local etcd.

Observability Placement

Disaster recovery replicates only the `global` cluster's etcd data, and namespaces such as `cpaas-system` are excluded from that replication. Monitoring and logging data is therefore never copied to the standby cluster. Plan observability placement before you install either cluster.

- Each `global` cluster collects, stores, and queries its own metrics and logs. The primary and standby clusters never use each other's monitoring or log storage, so the standby cluster can observe itself immediately after a failover.
- Workload cluster monitoring and logging must not be written to either `global` cluster. Aggregate workload cluster metrics and logs in a dedicated workload cluster that runs the

VictoriaMetrics Center and the log storage, then point the collection agents of each workload cluster at that cluster.

- If a network zone cannot reach that aggregation cluster, give the zone its own aggregation cluster instead of falling back to a `global` cluster.
- Workload cluster observability data that is stored in a `global` cluster becomes unavailable when that cluster's role changes, because the data is not replicated and the other `global` cluster cannot serve it.

For the synchronized and excluded scope, which is shared with the non-Alauda OS path, see Global Cluster Disaster Recovery.

Operational Scope

After the primary and standby clusters are installed, operate DR as a separate lifecycle process. Keep the installation manifests aligned with the installation guide, then use an approved operations runbook for the following tasks:

- Verify the health and replication lag of **Alauda Container Platform Global Cluster Disaster Recovery**.
- Verify that the standby cluster can decrypt Kubernetes Secrets created on the primary cluster.
- Validate the primary and standby control plane VIPs and the platform access path before a planned failover.
- Before a planned failover or failback, verify that the required ports are allowed in both directions between the two cluster networks. After the switch, the former standby becomes the new primary and the former primary becomes the new standby, so the direction of service traffic is reversed.
- Execute failover and failback with an approved operations procedure.
- Reconcile provider-specific resources after a failover.

> **Verify consistency before uninstalling Alauda Container Platform Global Cluster Disaster Recovery**

A disaster recovery switchover and a DR-aware `global` cluster upgrade both uninstall **Alauda Container Platform Global Cluster Disaster Recovery**. Before that uninstall, confirm that the standby `global` cluster data is consistent with the primary. On Alauda OS clusters, workload-cluster nodes are backed by Cluster API `Machine` objects, so an incorrect owner-reference resolution after an inconsistent sync can delete those `Machine` objects and destroy the backing virtual machines. If the consistency check reports missing or surplus keys, do not uninstall the plugin; resolve the inconsistency or contact technical support first. For the detailed switchover and upgrade procedures, which are shared with the non-Alauda OS path, see Global Cluster Disaster Recovery and Upgrade in a DR Environment.

Failover

The sequence below is provider-neutral: it is driven by the platform domain and the `etcd-sync` lifecycle, not by any provider API. Provider-specific synchronization sources and post-failover checks are in Provider Notes and, for Bare Metal, in the Bare Metal DR Model section.

1. Fence the primary so it cannot write workload resources.
2. Wait for the monitor to show no differences, then **uninstall** `etcd-sync` from the old standby. Removing its `ModuleInfo` is what uninstalls the plugin; scaling the Deployments to zero is not enough (see the warning below). On Bare Metal, then set `elemental.systemAgent.sharedAuthReadOnly: false` in the Bare Metal provider's `ModuleInfo` on the old standby, which becomes the active cluster, and wait until the `elemental-operator` Deployment no longer has the `--system-agent-shared-auth-read-only` argument (check with `kubectl -n cpaas-system get deploy elemental-operator -o yaml | grep -- --system-agent-shared-auth-read-only`). Until then, new worker nodes cannot obtain their plans.
3. Switch the platform domain to the standby control-plane entrance.
4. Verify the standby API and platform endpoint.
5. Confirm workload hosts reconnect through the platform domain and can update plan feedback.
6. On Bare Metal, first set `elemental.systemAgent.sharedAuthReadOnly: true` in the provider's `ModuleInfo` on the new standby and confirm with the same command that the `elemental-operator` Deployment has the `--system-agent-shared-auth-read-only`

argument. Then install the `etcd-sync` plugin on the new standby (the fenced former primary) with the reverse active/standby values and, on Bare Metal, the same `baremetal-dr-rules` ConfigMap. Remember that the active-cluster token must now come from the new active cluster, and that pushing the plugin package uses the Registry credential of the cluster the platform domain now points to.

Uninstall the mirror, do not just scale it down

On startup `etcd-sync` resolves the platform domain and checks that the cluster the domain points to does not have `etcd-sync` installed. This is what prevents the mirror from running on both sides at once. If step 2 only scaled the Deployments to zero, the `ModuleInfo` still exists and the new standby refuses to start with:

```
check failed, error: moduleinfo <name> is found, it shouldn't happe
n,
  please remove it or check the DNS resolution of platformURL: <pla
tform-domain>
```

Despite the wording this is the safety check working as intended, not a defect. Delete the `ModuleInfo` on the new active cluster and the new standby starts normally.

Keep the fenced cluster isolated until a separately planned failback. Never run both providers against the same workload inventory.

After the switch, on the new active cluster verify that every workload `Cluster` and `Machine` object is intact — on Alauda OS clusters those objects back real machines, so a resource lost in the switch becomes a deleted node.

Bare Metal DR Model

This section is the Bare Metal DR procedure for ACP 4.4.0 and later. It covers a fresh primary/standby installation; create both clusters with Optional Disaster Recovery Deployment.

Bare Metal keeps physical hosts and their plan Secrets in MachineInventory objects. DR therefore separates Global hosts from workload hosts.

Endpoint and Identity

Host	Endpoint	Identity	Sync
Local `Cluster/global` host	`https://<local-control-plane-vip>:6443`	`baremetal-global-system-agent`	Local only
Non-`global` workload host	`https://<platform-domain>/kubernetes/global`	`baremetal-system-agent`	Primary to standby

The Global row is specific to DR and to the post-handoff Global registration described below. The workload row is the standard Bare Metal registration path; for the stage-by-stage network requirements of the workload-cluster path — including DNS, NTP, the registry, and cluster-internal traffic — see Network Connectivity.

Set `elemental.systemAgent.splitAuthEnabled: true` on the Bare Metal provider of both sides. The active Global uses `sharedAuthReadOnly: false`; the standby uses `sharedAuthReadOnly: true` while it receives data. Set these values on each side's bootstrap release as described in Add Provider-Specific DR Fields, and again on each final `global` cluster in the provider's `ModuleInfo`, together with `elemental.tls.caCertSecretKey`; see Install the cluster plugins. Keep the standard ServiceAccount, Secret, Role, and RoleBinding names. Never grant namespace-wide Secret access.

After handoff, every Global registration must use the local API server:

```
metadata:
  annotations:
    baremetal.cluster.io/system-agent-auth-scope: global
    baremetal.cluster.io/system-agent-server-url: https://<local-control-plane-vip>:6443
    baremetal.cluster.io/system-agent-endpoint-mode: direct-apiserver
```

Workload registrations use the platform domain. Do not reuse a bootstrap Global ISO after handoff; create a new Global registration and SeedImage on the active Global. A workload ISO remains reusable only when its registration URL uses the platform domain.

Synchronization Rules

`etcd-sync` runs on the standby and reads the primary. The DR-capable v4.4 chart already supplies the standard Bare Metal prefixes and Global-resource exclusions, and accepts additional rules from labeled ConfigMaps. The custom rule source must exist on both Global clusters so it is available after a role switch.

Create `baremetal-dr-rules` with weight `100`:

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: baremetal-dr-rules
  namespace: cpaas-system
  labels:
    etcd-sync.cpaas.io/enabled: "true"
  annotations:
    etcd-sync.cpaas.io/weight: "100"
data:
  ignore-equal.txt: |
    # Add one line for every Global plan Secret on both sides.
    /registry/secrets/cpaas-system/<primary-global-plan-secret>
    /registry/secrets/cpaas-system/<standby-global-plan-secret>
```

The built-in chart rules already cover Bare Metal resource prefixes, Global inventory/registration filtering, `SeedImage`, the Global-local system-agent bundle, the monitor, and the local registry credentials. Do not duplicate those entries in this ConfigMap. The installed chart must contain those built-in rules before synchronization starts.

Replace the plan placeholders with live values:

- Read plan names from `MachineInventory.status.plan.secretRef.name`.
- Add one line for every Global plan Secret on both sides.
- Update the ConfigMap on both sides whenever a Global plan Secret is added or replaced.

Before the first sync, require the rule source annotation `etcd-sync.cpaas.io/status: accepted` and confirm that `etcd-sync-rule-snapshot` contains the source:

```
kubectl --kubeconfig <standby-kubeconfig> -n cpaas-system \
  get configmap baremetal-dr-rules \
  -o jsonpath='{.metadata.annotations.etcd-sync\.cpaas\.io/status}{"\n"}'
kubectl --kubeconfig <standby-kubeconfig> -n cpaas-system \
  get configmap etcd-sync-rule-snapshot \
  -o jsonpath='{.data.snapshot\.json}'
```

A rejected source is not active. Fix it before allowing the mirror to write.

The plugin chart owns the standard CAPI, Bare Metal, Elemental, Global-resource, system-agent, SeedImage, registry, and monitor rules. This ConfigMap adds only the runtime Global plan Secret keys that cannot be known when the chart is rendered.

`dcs-import-extra-resources` is an installation-time handoff input. It must contain durable provider resources and, for DCS and Bare Metal, the referenced encryption Secret, but it is not an etcd-sync rule source.

Start and Verify

1. Install `global-etcd-sync` only on the current standby.
2. Set `active_cluster_vip` to the primary VIP.
3. Set `active_cluster_token_secret_ref` to a standby-local Secret containing the primary cluster token.
4. Wait for the AppRelease and both Deployments to become Ready.
5. Run the monitor check and require zero missed and zero surplus keys.
6. Treat the rule snapshot and monitor result as part of the acceptance gate; AppRelease status alone is not sufficient.

Acceptance

After failover, verify:

- Workload `Cluster` and `Machine` objects remain healthy.
- Each `BaremetalMachine` still references the expected `MachineInventory`.
- A workload plan reaches `Applied` and its feedback is updated.

- Worker scale-out creates a plan and updates the shared Role.
- Worker scale-in completes its clean plan and releases inventory.
- A workload ISO whose URL uses the platform domain can register a new host.
- Global hosts still use the local VIP, local token, and local plan Secrets.

Troubleshooting

Symptom	First check
Rule source is `rejected`	Rule data keys, syntax, and the `message` annotation
A Global plan is overwritten	Its exact Secret key is missing from `ignore-equal.txt`; stop sync and update both rule ConfigMaps
A Global inventory is deleted	The installed chart is missing its built-in Global exclusion; stop sync and verify the rendered chart before restoring the local object
A workload host gets `401` or `403`	Signing key, issuer/audience, shared token Secret, and Role `resourceNames`
Monitor reports differences	Active/standby direction, rule snapshot, and fencing state

Provider Notes

Huawei DCS

Follow the DCS steps in Optional Disaster Recovery Deployment. The DCS installation must keep the same encryption provider Secret and `DCSCluster.spec.encryptionProviderConfigRef` on both sides. Do not add the encryption provider file to `KubeadmControlPlane.spec.kubeadmConfigSpec.files` for DCS. Create the DCS `dcs-import-extra-resources` ConfigMap on both sides so the installer imports the Secret referenced by `DCSCluster.spec.credentialSecretRef.name` and the Secret referenced by

`DCSCluster.spec.encryptionProviderConfigRef.name`. DCS provider resources are migrated by the built-in flow, but that flow does not copy either Secret.

The encryption provider Secret must exist in each final `global` cluster. If it stays only in the bootstrap cluster, the installation still completes and the pair looks healthy, but every later control-plane machine creation fails once the bootstrap cluster is gone — the new `Machine` stays in `Provisioning` and the provider logs `get encryption provider config secret ... not found`. Verify it on both sides before decommissioning either bootstrap cluster.

For a planned switch, run the failover sequence. The `baremetal-dr-rules` ConfigMap and the registration checks are Bare Metal-only; every other step applies to DCS as written. After the switch, verify on the new active cluster that the workload `Cluster`, `Machine`, and `DCSMachine` objects are intact and no virtual machine was deleted: during failover the `global-` name prefix from the installation guide's naming convention is what identifies the resources owned by the `global` pair, which is why every DCS resource of both sides must carry it. Also confirm that the DCS credential Secret and the encryption provider Secret exist on whichever side is now active.

VMware vSphere

Follow the VMware vSphere steps in Optional Disaster Recovery Deployment. The VMware vSphere installation uses the same `KubeadmControlPlane` encryption file entry as the primary installation, and its own etcd SAN list. No `VSphereCluster` encryption Secret reference is required. Create the VMware vSphere `dcs-import-extra-resources` ConfigMap so the installer imports vSphere infrastructure resources and the `global-vsphere-credentials` Secret referenced by `VSphereCluster.spec.identityRef.name`.

Huawei Cloud Stack

Follow the HCS steps in Optional Disaster Recovery Deployment. The HCS installation uses the same `KubeadmControlPlane` encryption file entry as the primary installation, and its own etcd SAN list. No `HCSCluster` encryption Secret reference is required. Create the HCS `dcs-import-extra-resources` ConfigMap so the installer imports

HCS infrastructure resources and the Secret referenced by `HCSCluster.spec.identityRef.name`.

Bare Metal

Follow the Bare Metal steps in Optional Disaster Recovery Deployment. Use two bootstrap KIND hosts, one for primary and one for standby. Before installing standby, switch the platform domain to the standby entrance so platform-rendered URLs resolve to the side being installed.

Bare Metal requires all of the following to be aligned:

- Same Kubernetes encryption provider Secret content on both sides, referenced by `BaremetalCluster.spec.encryptionProviderConfigRef`.
- The encryption provider Secret is imported into each final `global` cluster through `dcs-import-extra-resources`; it must not exist only in the bootstrap KIND cluster.
- Same ServiceAccount signing key files and same issuer/audience values on both sides.
- Same trusted platform HTTPS certificate on both sides, with the stable platform domain in its SAN list. Add a control-plane VIP SAN only when platform HTTPS intentionally uses that VIP; do not add internal Service names to `console.cert`. With the platform built-in Registry, the registry on port `11443` uses an independently managed `global-registry-server` certificate chain.
- `dcs-import-extra-resources` created before the installer API call on both sides, containing the durable Global Bare Metal and Elemental owner resources plus the referenced encryption-provider Secret. It must exclude Global plan Secrets, `KubeadmConfig` objects, and kubeadm bootstrap data Secrets; fresh handoff creates the initial Global plan Secrets with the live Global `MachineInventory` UID.
- Bare Metal provider AppRelease values on each bootstrap cluster set with the local `handoffHook.controlPlaneVIP`, `handoffHook.directAPIServer: true`, `elemental.systemAgent.authMode: shared`, `elemental.systemAgent.splitAuthEnabled: true`, and `elemental.systemAgent.globalServiceAccountName: baremetal-global-system-agent`. The active side sets `elemental.systemAgent.sharedAuthReadOnly: false`; the inactive side sets it to `true`. On each final `global` cluster, set only

`elemental.tls.caCertSecretKey`, `elemental.systemAgent.splitAuthEnabled: true`, and that side's `sharedAuthReadOnly` value, in `spec.valuesOverride` of the provider's `ModuleInfo`; see Install the cluster plugins.

- The deployed `elemental-operator` image and every newly produced `baremetal-base-image` and `baremetal-base-image-iso` are built from the same pinned elemental commit. Rebuild the installation media whenever that commit changes.

Do not sync `SeedImage` from primary to standby. Sync non-`global` `MachineRegistration` resources when workload ISOs need to remain valid across failover. Do not sync primary Global `MachineRegistration` resources to standby. Create Global registrations locally with `baremetal.cluster.io/system-agent-auth-scope: global`, `baremetal.cluster.io/system-agent-server-url: https://<local-global-control-plane-vip>:6443`, and `baremetal.cluster.io/system-agent-endpoint-mode: direct-apiserver`.

After failover, keep the primary fenced until a separate failback plan is executed. Do not run primary and standby Bare Metal providers against the same workload inventories at the same time.

See Also

For disaster recovery of a `global` cluster on a non-Alauda OS, see Global Cluster Disaster Recovery.

For installation and upgrade of the `global` cluster on Alauda OS, see:

- Installing the global Cluster
- Upgrade the Distribution Version, then Upgrade Kubernetes on Alauda OS
- Upgrade in a DR Environment

Install on Non-Alauda OS

Prerequisites | Node Preprocessing | Installing

Installer Parameters | Global Cluster Disaster Recove | Installation

Prerequisites

Before installing the `global` cluster, you need to prepare hardware, network, and OS that meet the requirements.

INFO

1. The platform currently does not support direct installation of the `global` cluster in an existing Kubernetes environment. If your environment already has a Kubernetes cluster, please back up your data and clean the environment before installation.
2. If you plan to use global Cluster Disaster Recovery, please first read global Cluster Disaster Recovery.
3. Make sure all of the new nodes meet the Nodes Requirements.
4. The disks' performance and capacity also have to meet the Disk Configuration Requirements.

TOC

Resource Planning

Deployment Architectures

Single Node

Single Cluster

Multi-Cluster

Network

Network Resources

Network Configuration

LoadBalancer Forwarding Rules

External Platform Image Registry

Resource Planning

This section provides guidelines for resource planning before installing ACP. Choose the appropriate deployment scenario based on your environment and usage needs, and prepare resources accordingly.

INFO

The following recommendations only cover the minimum resources required for a successful installation of the **Global cluster**.

They **do not** include the resources required for any **additional extensions or components** deployed on the Global cluster.

For detailed requirements of each extension, refer to the corresponding component documentation.

Deployment Architectures

WARNING

For ARM architectures (such as Kunpeng 920), it is recommended to increase the configuration to **2 times** that of the x86 minimum configuration, but not less than **1.5 times**.

For example: If x86 requires 8 cores 16GB, then ARM should reach at least 12 cores 24GB, and the recommended configuration is 16 cores 32GB.

Before installation, you must determine which deployment architecture best fits your use case. ACP supports the following three common deployment architectures:

- **Multi-Cluster**

Select this architecture if you need to centrally manage multiple Kubernetes clusters. In this mode, ACP consists of one **Global cluster** and multiple **workload clusters**. Running non-platform workloads on the Global cluster may degrade platform stability and performance, and should be avoided.

- **Single Cluster**
  Select this architecture if you plan to install only one cluster and run workloads directly on it. In this mode, the Global cluster also acts as a workload cluster, and therefore requires **more resources** compared with a pure Global-only setup in Multi-Cluster mode.
- **Single Node**

> **WARNING**
>
> This architecture is intended solely for testing or proof-of-concept purposes and should not be used in production.

Single Node

The following table lists the **minimum hardware requirements** for installing ACP in **Single Node** mode.

Resource	Minimum Requirement
CPU	12 cores
Memory	24GB
Storage	Storage Capacity

Single Cluster

In this mode, the Global cluster serves both as the control plane and as the workload cluster. The number of control plane nodes of global cluster MUST be 3.

The total resource requirement consists of two parts:

- **Base resources** for the Global cluster itself

- **Additional resources** for running workloads on the same cluster

The resources required for a **highly available Global cluster** are as follows:

Resource	Minimum Requirement
CPU	8 cores
Memory	16GB
Storage	Storage Capacity

To estimate the additional resources required for your workloads, refer to **Evaluating Resources for Workload Cluster**

Multi-Cluster

When managing multiple workload clusters, the resource usage of the Global cluster increases proportionally with the number of managed clusters. The additional overhead mainly comes from cluster registration, monitoring, and control-plane synchronization.

To estimate the resources required for your Global cluster based on the number of managed clusters, refer to **Evaluating Resources for Global Cluster**

Network

Before installation, ensure that the required network resources are prepared.

If a hardware LoadBalancer is available in your environment, it is **recommended** to use it. If not, you can enable `Self-built VIP`, which provides software load balancing using keepalived.

Note: `Self-built VIP` does not support domain name configuration.

Network Resources

Resource	Mandatory	Quantity	Description
`global` VIP	Mandatory	1	Used for nodes in the cluster to access kube-apiserver, configured in the load balancing device to ensure high availability. This IP can also be used as the access address for the platform Web UI. Workload clusters in the same network as the `global` cluster can also access the `global` cluster through this IP.
External IP	Optional	On Demand	When there are workload clusters that are not in the same network as the `global` cluster, such as a hybrid cloud scenario, it must be provided. Workload clusters in other networks access the `global` cluster through this IP. This IP needs to be configured in the load balancing device to ensure high availability. This IP can also be used as the access address for the platform Web UI.
Domain Name	Recommended	On Demand	Recommended for Cluster Endpoint and Platform Access Address. Please provide it in advance and ensure that the domain name resolution is correct. Using a domain name is recommended because if the VIP needs to be changed after cluster installation, you can easily update the DNS record without impacting cluster operations.

Resource	Mandatory	Quantity	Description
			A domain name is **required** in the following cases: • The `global` cluster needs to support IPv6 access; • A disaster recovery plan for the `global` cluster is planned.
Certificate	Optional	On Demand	It is recommended to use a trusted certificate to avoid browser security warnings; if not provided, the installer will generate a self-signed certificate, but there may be security risks when using HTTPS.

NOTE

If the platform needs to configure multiple access addresses (for example, addresses for internal and external networks), please prepare the corresponding IP addresses or domain names in advance according to the table above. You can configure them in the installation parameters later, or add them according to the product documentation after installation.

Network Configuration

Type	Requirement Description
Network Bandwidth	Intra-cluster bandwidth must be ≥1 Gbps (recommended 10 Gbps). Cross-cluster bandwidth must be ≥100 Mbps (recommended 1 Gbps). Insufficient bandwidth may significantly degrade data query performance.
Network Latency	Intra-cluster latency must be ≤10 ms. Cross-cluster latency must be ≤100 ms (recommended ≤30 ms).
Network Policy	Please refer to LoadBalancer Forwarding Rules to ensure that the necessary ports are open.

Type	Requirement Description
IP Address Range	The `global` cluster nodes should avoid using the 172.17-18 network segment. If it has been used, please adjust the nerdctl configuration (add the bip parameter) to avoid conflicts.

LoadBalancer Forwarding Rules

This rule is designed to ensure that the `global` cluster can receive traffic from the LoadBalancer normally. Please check the network policy according to the following table to ensure that the relevant ports are open.

Source IP	Protocol	Destination IP	Destination Port	Description
`global` VIP, External IP	TCP	All control plane node IPs	443	Provides access services for the platform Web UI, image repository, and Kubernetes API Server through the HTTPS protocol. The default port is `443`. If you need to use a custom HTTPS port, please do the following: • Replace the destination port in the port forwarding

Source IP	Protocol	Destination IP	Destination Port	Description
				rule with your custom port number. • Later, in the Web UI installation parameters, fill in your custom port number.
`global` VIP, External IP	TCP	All control plane node IPs	6443	This port provides access to the Kubernetes API Server for nodes within the cluster.
`global` VIP, External IP	TCP	All control plane node IPs	11443	This port provides access to the image repository for nodes within the cluster. **Note:** If you plan to use an external platform image registry instead of the Registry

Source IP	Protocol	Destination IP	Destination Port	Description
				provided by the `global` cluster, you do not need to configure this port. See Prepare an External Platform Image Registry for the external registry network requirements.

TIP

- It is recommended to configure health checks on the LoadBalancer to monitor the port status.
- Global Cluster Disaster Recovery requires load balancer listeners on both cluster VIPs and bidirectional network access between the Primary and Standby Cluster networks. See Network Requirements.
- The platform only supports HTTPS by default. If HTTP support is required, you need to open the HTTP port for all control plane nodes.

External Platform Image Registry

For new production deployments, prepare a customer-managed external platform image registry when your organization requires independent registry availability, capacity, backup, and lifecycle management.

Before running `setup.sh`, complete Prepare an External Platform Image Registry. That page defines the required address, network access, storage, and push and pull permissions; separates those requirements from production recommendations; and explains how to upload the complete target-version payload.

Node Preprocessing

Before installing the `global` cluster, all nodes (control plane nodes and worker nodes) must complete preprocessing.

> **INFO**
>
> This page applies to nodes running a **non-Alauda OS** such as Kylin Linux Advanced Server, RHEL, or Ubuntu, which ACP provisions over SSH. Several of the checks below — for example the SSH user and the `/etc/ssh/sshd_config` settings — exist to keep the SSH-based node join working. If your cluster uses Alauda OS, the node preprocessing steps on this page do not apply. Follow Installing the global Cluster on Alauda OS instead.

TOC

- Supported OS and Kernel Versions
  - x86
  - ARM
  - Operating System Support Scope Changes
- Execute the Quick Configuration Script
- Node Checks
- Appendix
  - Why SELinux and AppArmor Must Be Disabled
  - Hardened Node Security Configuration
    - Keep the Host Firewall Enabled Instead of Turning It Back On

Order of Operations
If a Node Already Ran the Full `init.sh`
Host Firewall
Remove Conflicting Packages
Configure Search Domain

Supported OS and Kernel Versions

The following table lists the supported operating systems, their validated versions, and the corresponding tested kernel versions.

The platform enforces the version-matching granularity declared in the support list:

- **OS version**: Use the distribution release listed below. A different major or minor release is not officially supported unless it is also listed.
- **Kernel version**: Use an official kernel supplied by the operating system vendor. When the list gives an exact `x.y.z-build` value, the build suffix can vary but `x.y.z` must match. When the list names an operating system release without a kernel value, use the official kernel shipped with that release. Do not assume that an arbitrary newer kernel is supported.

INFO

`x86-64-v2` is a CPU instruction set baseline, but ACP does not impose it as a universal requirement on x86 nodes that run a non-Alauda OS. The CPU must still meet the requirements of the selected operating system and the components to be deployed. If compatibility cannot be confirmed, contact technical support.

x86

Red Hat Enterprise Linux (RHEL)

- RHEL 9.6: `5.14.0-570.12.1`

Ubuntu

- Ubuntu 22.04 LTS: `5.15.0-56-generic`

Note: Ubuntu HWE (Hardware Enablement) versions are not supported.

Kylin Linux Advanced Server

- Kylin Linux Advanced Server V11 2503: official 6.6-series kernel shipped with the release

ARM

Kylin Linux Advanced Server

- Kylin Linux Advanced Server V11 2503: official 6.6-series kernel shipped with the release

Note: ARM architecture only supports `Kunpeng 920`. For other CPU models, please contact technical support.

Operating System Support Scope Changes

ACP 4.4 introduced Kubernetes 1.35, which removed cgroup v1. Nodes in a cluster running Kubernetes 1.35 or later must use cgroup v2 and Linux kernel 5.8 or later.

Because their kernels are earlier than 5.8, the following operating systems were supported through ACP 4.3 and are no longer in the support scope from ACP 4.4 onward:

- Kylin Linux Advanced Server V10 and V10 SP3 (kernel 4.19)
- Ubuntu 20.04 LTS (kernel 5.4)
- RHEL 7.x and RHEL 8.x (kernel 3.10 and kernel 4.18)
- CentOS 7.x (kernel 3.10)

If your nodes run one of these operating systems, choose one of the following paths. Moving to a newer release from the same operating system vendor is usually the cheapest option, because it keeps your existing vendor relationship, tooling, and operational practices:

- Move RHEL 7.x or RHEL 8.x nodes to RHEL 9.6.
- Move Ubuntu 20.04 LTS nodes to Ubuntu 22.04 LTS.
- Move Kylin Linux Advanced Server V10 or V10 SP3 nodes to Kylin Linux Advanced Server V11 2503.
- Move the nodes to Alauda OS. See Install on Alauda OS for the `global` cluster and Create on Alauda OS for workload clusters. Alauda OS is an immutable operating system, so this path changes how the nodes are managed and is usually the most expensive option.
- Stay on ACP 4.3.x until you can complete the operating system change.

CentOS 7.x has no successor in the support list. Move those nodes to one of the supported operating systems above.

> **WARNING**
>
> Change the operating system while the cluster still runs its current ACP version, and upgrade to ACP 4.4 or later only after every node runs a supported operating system. The change usually requires rebuilding the nodes or replacing the machines. Upgrading ACP does not migrate a node to a different operating system.

Execute the Quick Configuration Script

The ACP installation package provides a script for quickly configuring nodes.

Unzip the installation package to obtain the `init.sh` script file in the `res` directory. Copy the script file to the nodes and ensure that you have `root` privileges.

Execute the script:

```
bash init.sh
```

WARNING

`init.sh` cannot guarantee that all of the following checks are properly handled. You still need to continue with the steps below.

Node Checks

The following lists all the checks that must be completed on the nodes. Depending on the node's role, the required checks will vary. For example, some checks apply only to control plane nodes.

Checks are divided into two categories:

- ✅ Indicates a check that must pass.
- ⚠️ Indicates a check that must be met in specific scenarios. Please determine whether the corresponding conditions are met according to the instructions. If they are, you must resolve them.

The following is the list of checks:

- **OS and Kernel**
  - ✅ The machine's grub boot configuration must have the `transparent_hugepage=never` parameter.
  - ✅ Check whether the kernel modules `ip_vs`, `ip_vs_rr`, `ip_vs_wrr`, and `ip_vs_sh` are enabled.
  - ⚠️ If the `global` cluster plans to use `Kube-OVN` CNI, the kernel modules `geneve` and `openvswitch` must be enabled.
  - ✅ SELinux and AppArmor must be disabled. They cannot be kept enabled on a node that runs a non-Alauda OS; see Why SELinux and AppArmor Must Be Disabled.
  - ✅ The host firewall must be disabled, unless your security policy requires it to stay enabled. In that case the node must be configured as described in Hardened Node Security Configuration before you install the platform.

    - `init.sh` disables the host firewall. To keep it enabled, pass the matching `--skip-` argument to `init.sh` instead of turning it back on afterwards.
    - Keeping the host firewall enabled is a non-default configuration. The platform does not manage a non-Alauda OS, so an enabled host firewall can cause unexpected failures on the node or in the cluster. You are responsible for operating and troubleshooting the nodes in this configuration.
    - When the host firewall is enabled, you configure and maintain the port rules on every node. For the ports that the platform requires, see Communication Matrix.
  - ✅ Disable `swap` .

- **Users and Permissions**
  - ✅ The node's SSH user has `root` privileges and can use `sudo` without the password.
  - ✅ The `UseDNS` parameter in `/etc/ssh/sshd_config` must be set to `no` .
  - ✅ Set the `UsePAM` parameter in `/etc/ssh/sshd_config` to `no` before adding the node, then restart `sshd` . PAM session policies (such as a forced password change, `pam_access` , `faillock` , or `pam_limits`) can otherwise block the SSH-based node join. After the node reaches the `Ready` state, you can restore `UsePAM yes` . On SELinux-enforcing systems that use password authentication, `UsePAM no` can itself break SSH login; use key-based authentication in that case.
  - ✅ `systemctl show --property=DefaultTasksMax` must return `infinity` ; a low limit can make busy containers fail to create threads. If it is not `infinity` , set `DefaultTasksMax=infinity` in `/etc/systemd/system.conf` and run `systemctl daemon-reexec` .
- **Node Network**
  - ✅ `hostname` must comply with the following rules:
    - No more than 36 characters.
    - Starts and ends with a letter or number.
    - Contains only lowercase letters, numbers, `-` , and `.` , but cannot contain `.-` , `..` , or `-.` .
  - ✅ `localhost` in `/etc/hosts` must resolve to `127.0.0.1` .

- ✅ The `/etc/resolv.conf` file must exist and contain `nameserver` configurations, but must not contain addresses starting with 172 (disable systemd-resolved).
- ⚠️ The `/etc/resolv.conf` file should not configure search domains (if you must configure them, see Configure Search Domain).
- ✅ The machine's IP address cannot be a loopback, multicast, link-local, all-0, or broadcast address.
- ✅ Executing `ip route` must return a default route or a route pointing to `0.0.0.0`.
- ✅ The nodes must not occupy the following ports. This check confirms that the ports are free before the installation starts; it is not the list of ports that a host firewall must allow. For that list, see Communication Matrix.
  - **Control plane nodes:** `2379`, `2380`, `6443`, `10249` ~ `10256`
  - **Node where the installer is located:** `8080`, `12080`, `12443`, `16443`, `2379`, `2380`, `6443`, `10249` ~ `10256`
  - **Worker nodes:** `10249` ~ `10256`
- ✅ If the cluster uses **Kube-OVN** or **Calico**, ensure that the following ports are not occupied:
  - **Kube-OVN:** `6641`, `6642`
  - **Calico:** `179`
- ⚠️ Ensure that the IP addresses in the network segment `172.17.x.x` ~ `172.18.x.x` required by nerdctl are not occupied. If the IPs in this network segment are occupied and cannot be changed, please contact technical support.

- **Software and Directory Requirements:**
  - ✅ Must have the following installed: `ip`, `ss`, `tar`, `swapoff`, `modprobe`, `sysctl`, `md5sum`, and `scp` or `sftp`.
  - ⚠️ If you plan to use local storage **TopoLVM** or **Rook**, you need to install `lvm2`.
  - ✅ The `/etc/systemd/system/kubelet.service` file is not allowed to exist.
  - ✅ `/tmp` mount parameters must not contain `noexec`.
  - ✅ Remove packages that conflict with `global` cluster components (see Remove Conflicting Packages).

- ✅ The following files must be deleted if they exist:
  - `/var/lib/docker`
  - `/var/lib/nerdctl`
  - `/opt/nerdctl/`
  - `/var/lib/containerd`
  - `/var/log/pods`
  - `/var/lib/kubelet/pki`

- **Cross-Node Checks**
  - ✅ Network communication between nodes in the `global` cluster must not be restricted:
    - In the default configuration, where no host firewall is enabled, there must be no network firewall restrictions between the nodes.
    - When a host firewall is enabled on the nodes, every port in Communication Matrix must be allowed between the nodes, and Pod and Service traffic inside the cluster must not be filtered by port.
  - ✅ The `hostname` of each node in the cluster must be unique.
  - ✅ The time zones of all nodes must be unified, and the time synchronization error must be ≤ 10 seconds.

Appendix

Why SELinux and AppArmor Must Be Disabled

On a node that runs a non-Alauda OS, SELinux and AppArmor must be disabled. This is a platform requirement for this release, not a hardening recommendation. `init.sh` disables both, and the node join procedure disables them again when a node joins a cluster.

SELinux. Platform components mount host paths into containers. The image registry installer bind-mounts the host `/proc/mounts` file into its container, and the registry and object storage components keep their data under host directories such as `/cpaas` and

`/etc/kubernetes`, which carry the operating system's default file labels. On a non-Alauda OS the container runtime is configured without SELinux labeling, so these mounts are evaluated against the host policy, and they are denied when SELinux runs in `enforcing` mode. The container runtime then fails to create the container before its process starts, the image registry component never finishes installing, and the platform installation or upgrade stops.

`permissive` mode is not an alternative: it changes what is logged, not the requirement. A node that is switched back to `enforcing` after the installation is outside the supported configuration.

AppArmor. Node preparation disables AppArmor, and the platform is not validated with AppArmor profiles enforced on cluster nodes. Running a node with AppArmor enabled is outside the supported configuration for this release.

A hardened operating system profile, such as the STIG profile provided by Ubuntu Security Guide, enables these mechanisms. Applying such a profile to a cluster node is outside the supported configuration for the same reason.

If your security policy requires these mechanisms to stay enabled. Use Alauda OS, the immutable operating system delivered with the platform. On Alauda OS, SELinux and the host firewall are enabled by default, and the platform maintains the policy and the port rules for you. See Install on Alauda OS and Create on Alauda OS.

Support for running Red Hat Enterprise Linux 9.6 with SELinux in `enforcing` mode is planned for a future release. Contact technical support for its status.

Hardened Node Security Configuration

Use this section only when a mandatory security policy, such as a STIG baseline, requires a host firewall to stay enabled on cluster nodes. Keeping the host firewall disabled is the default and recommended node configuration, and it is what `init.sh` produces when you run it without arguments. Enabling it is a non-default configuration that you own, so validate it in a non-production environment before you apply it to production nodes.

SELinux and AppArmor are a different case: they must stay disabled. See Why SELinux and AppArmor Must Be Disabled.

Keep the Host Firewall Enabled Instead of Turning It Back On

`init.sh` can skip individual steps. Pass the matching `--skip-` argument so that the script never disables the host firewall your policy requires:

Implementation	Argument
`firewalld`	`--skip-disable_firewalld`
`ufw`	`--skip-disable_ufw`

For example, to keep `firewalld` enabled on a RHEL node:

```
bash init.sh --skip-disable_firewalld
```

This is the recommended path. The node keeps the state it was installed with, and `init.sh` still performs every other node preparation step.

Order of Operations

Configure the node before the platform installation, not after. A firewall problem then surfaces while the installer runs, instead of after the cluster carries workloads.

1. Run `init.sh` with the `--skip-` argument for the host firewall implementation you use.
2. Configure the host firewall rules.
3. Verify the resulting state on the node with the status command of your firewall implementation, for example `firewall-cmd --list-all` or `ufw status verbose`.
4. Install the platform.

If a Node Already Ran the Full `init.sh`

Use this path only when the host firewall was already disabled and you have to turn it back on. It is more work than skipping the step in the first place.

1. Enable and start the service, for example with `systemctl enable --now firewalld`, or with `ufw enable` on Ubuntu.

2. Configure the port rules as described below.
3. Confirm the resulting rule set with `firewall-cmd --list-all` or `ufw status verbose`.

Host Firewall

The platform does not require a specific host firewall implementation. RHEL and Kylin Linux Advanced Server nodes typically use `firewalld`, and Ubuntu nodes typically use `ufw`. Whichever implementation you use, allow the same set of ports. For the port list, see Communication Matrix.

> **DANGER**
>
> The communication matrix lists the ports that the nodes listen on. Pod-to-Pod and Pod-to-Service traffic uses arbitrary ports and is not covered by that list, so a firewall that allows only the listed ports breaks cluster networking. When the host firewall is enabled, also treat the Pod CIDR, the Service CIDR, and the network interfaces created by the CNI plugin as trusted sources, so that cluster-internal traffic is not filtered by port. The interface names depend on the CNI plugin and on the environment, so read them from the node instead of assuming a fixed name.

The node that runs the installer also uses `8080`, `12080`, `12443`, and `16443`. Allow these ports on that node before you start the installation.

Remove Conflicting Packages

Before installation, applications may already be running in the docker/nerdctl/containerd environment on the nodes, or software conflicting with the `global` cluster may have been installed. Therefore, it is necessary to check and uninstall conflicting packages.

> **DANGER**
>
> - To avoid application interruption or data loss, be sure to confirm whether there are conflicting software packages. When a conflict is found, please develop an application switching plan and back up your data before uninstalling.
> - After uninstalling conflicting packages, you still need to check whether there are other potentially conflicting binary files in directories such as `/usr/local/bin/` (such as software related to

docker, nerdctl, containerd, runc, podman, container network, container runtime, or Kubernetes).

The following commands can be used for reference.

RHEL

Check:

```
for x in \
    docker docker-client docker-common docker-latest \
    podman-docker podman \
    runc \
    containernetworking-plugins \
    apptainer \
    kubernetes kubernetes-master kubernetes-node kubernetes-client \
    ; do
    rpm -qa | grep -F "$x"
done
```

Uninstall:

```
for x in \
    docker docker-client docker-common docker-latest \
    podman-docker podman \
    runc \
    containernetworking-plugins \
    apptainer \
    kubernetes kubernetes-master kubernetes-node kubernetes-client \
    ; do
    yum remove "$x"
done
```

Ubuntu

Check:

```
for x in \
    docker.io \
    podman-docker \
    containerd \
    rootlesskit \
    rkt \
    containernetworking-plugins \
    kubernetes \
    ; do
    dpkg-query -l | grep -F "$x"
done

for x in \
    kubernetes-worker \
    kubectl kube-proxy kube-scheduler kube-controller-manager kube-ap
iserver \
    k8s microk8s \
    kubeadm kubelet \
    ; do
    snap list | grep -F "$x"
done
```

Uninstall:

```
for x in \
    docker.io \
    podman-docker \
    containerd \
    rootlesskit \
    rkt \
    containernetworking-plugins \
    kubernetes \
    ; do
    apt-get purge "$x"
done

for x in \
    kubernetes-worker \
    kubectl kube-proxy kube-scheduler kube-controller-manager kube-ap
iserver \
    k8s microk8s \
    kubeadm kubelet \
    ; do
    snap remove --purge "$x"
done
```

Kylin

Check:

```
for x in \
    docker docker-client docker-common \
    docker-engine docker-proxy docker-runc \
    podman-docker podman \
    containernetworking-plugins \
    apptainer \
    containerd \
    kubernetes kubernetes-master kubernetes-node kubernetes-client ku
bernetes-kubeadm \
    ; do
    rpm -qa | grep -F "$x"
done
```

Uninstall:

```
for x in \
    docker docker-client docker-common \
    docker-engine docker-proxy docker-runc \
    podman-docker podman \
    containernetworking-plugins \
    apptainer \
    containerd \
    kubernetes kubernetes-master kubernetes-node kubernetes-client ku
bernetes-kubeadm \
    ; do
    yum remove "$x"
done
```

Configure Search Domain

In Linux OS, the `/etc/resolv.conf` file is used to configure DNS client domain name resolution settings. The `search` line specifies the domain search path for DNS queries.

Configuration Requirements

- **Number of Domains:** The number of domains in the `search` line should be less than `domainCountLimit - 3` (default `domainCountLimit` is 32).
- **Length of Single Domain:** Each domain name must not exceed 253 characters.
- **Total Character Length:** The total character count of all domain names and spaces must not exceed `MaxDNSSearchListChar` (default is 2048).

Example

```
search domain1.com domain2.com domain3.com
```

- The total number of domains is 3.
- The length of a single domain, such as `domain1.com`, is 11.
- The total character length is 35, i.e., 11 + 11 + 11 + 2 (two spaces).

WARNING

- If the `search` line in the `/etc/resolv.conf` file does not meet the above limitations, it may cause DNS query failures or performance degradation.
- Before modifying the `/etc/resolv.conf` file, it is recommended to back up the file.

Installing

This section describes how to install ACP Core and the required Aligned Extensions, and deploy the `global` cluster.

> **Installation Path**
>
> This installation path applies to clusters running on a **non-Alauda OS**. If your environment uses Alauda OS, see Installing the global Cluster on Alauda OS instead.

Before starting the installation, ensure that you have completed the prerequisite checks, downloaded and verified the Core Package and required Aligned Extension packages, completed node preprocessing, and finished the other preparation work. If you selected an external platform image registry, its complete target-version payload must already be available before you run `setup.sh`.

TOC

- Process
  - Upload and Extract Installation Package
  - Prepare the Required Aligned Extension Packages
  - Start the Installer
    - IP Family
  - Parameter Configuration
- Validate the Installation
- Installer Parameter Reference

Installation Troubleshooting

Additional Resources

Process

1 Upload and Extract Installation Package

Upload the Core Package installation package to any machine of the `global` cluster control plane nodes, and extract it according to the following command:

```
# Assume that the /root/cpaas-install folder already exists on the machine
tar -xvf {Path to Core Package File}/{Core Package File Name} -C /root/cpaas-install
cd /root/cpaas-install/installer || exit 1
```

INFO

- This machine will become the first control plane node after the `global` cluster installation is complete.
- After the Core Package is extracted, at least **100GB** of disk space is required. Please ensure sufficient storage resources.
- The Core Package does not contain the required Aligned Extension packages. Prepare them separately as described in Download.

2 Prepare the Required Aligned Extension Packages

Copy each of the eight packages listed in Required Aligned Extensions into the `plugins/` directory of the extracted Core Package. Run the following command once for each package:

```
cp <path-to-extension-package> /root/cpaas-install/installer/plugins/
```

Confirm that the directory contains the eight required packages before starting the installer:

```
ls -1 /root/cpaas-install/installer/plugins/
```

Do not copy other packages from **Common ACP Extensions** into this directory. The installer processes only the packages that you manually place in `plugins/`; it does not download packages from the Customer Portal.

3 Start the Installer

Choose the command that matches the platform image source. After the installer starts successfully, the terminal outputs the web console access address.

After waiting for about 5 minutes, you can use a browser on your PC to access the web console provided by the installer.

For the platform built-in Registry:

```
bash setup.sh
```

For an external registry that permits anonymous pull:

```
bash setup.sh --registry "<registry-address>"
```

For an authenticated external registry, provide the pull username and password together. The pull account used by `setup.sh` can be different from the push account used to upload the payload:

```
export REGISTRY_ADDRESS="registry.example.com"
export REGISTRY_USERNAME="<registry-pull-username>"
read -rsp "Registry password: " REGISTRY_PASSWORD
export REGISTRY_PASSWORD
printf '\n'

bash setup.sh \
  --registry "$REGISTRY_ADDRESS" \
  --username "$REGISTRY_USERNAME" \
  --password "$REGISTRY_PASSWORD"

unset REGISTRY_PASSWORD
```

The registry address must use the form `<host-or-IP>[:port]` without `http://` or `https://`. Do not provide only one of `--username` and `--password`.

> **WARNING**
>
> Ensure that the IP address and port 8080 of the node where the installer is located can be accessed normally, so that the web console provided by the installer can be accessed smoothly after the installer starts successfully.

IP Family

```
bash setup.sh --ip-family ipv6
```

If you plan to create a `global` cluster with Single-stack Network IPv6, you must explicitly specify `--ip-family ipv6` when starting the installer. Without this parameter, the `global` cluster created by the installer will support Single-stack Network IPv4 and Dual-stack Network by default.

4 Parameter Configuration

Configure the installation in the Web UI, review every value, and then confirm the installation. Use Installer Parameters for the field-by-field reference.

Validate the Installation

After the installer reports that the installation is complete, validate Core and the required Aligned Extensions before continuing with post-install tasks. For the checklist and commands, see Validation.

Installer Parameter Reference

For the detailed field reference, see Installer Parameters.

Installation Troubleshooting

For installer stalls, failed resources, image-pull failures, missing artifacts, and cleanup, see Installation Troubleshooting.

Additional Resources

- Upload and Install Extensions
- Installer Parameters
- Validation
- Installation Troubleshooting
- Next Steps

Installer Parameters

Use this reference while configuring the `global` cluster in the installer Web UI. Confirm the network, access, image source, and node decisions before you submit the installation.

Parameter	Description
Kubernetes Version	All optional versions are rigorously tested for stability and compatibility. **Recommendation:** Choose the latest version for optimal features and support.
Cluster Network Protocol	Supports three modes: IPv4 single stack, IPv6 single stack, IPv4/IPv6 dual stack. **Note:** If you select dual stack mode, ensure all nodes have correctly configured IPv6 addresses; the network protocol cannot be changed after setting.
Cluster Endpoint	Enter the pre-prepared domain name. If no domain name is available, enter the pre-prepared `global VIP`. `Self-built VIP` is disabled by default, only enable it if you have not provided a LoadBalancer. The following conditions must be met when using `Self-built VIP` : • A usable VRID is available;

	• The host network supports the VRRP protocol; • All control plane nodes and the VIP must be on the same subnet. **Tip:** For single-node deployments in feature experience scenarios, you can directly enter the node IP. There is no need to enable `Self-built VIP` or prepare network resources such as `global VIP`.
Platform Access Address	If you do not need to distinguish between **Cluster Endpoint** and **Platform Access Address**, enter the same address as the **Cluster Endpoint**. If you need to distinguish, for example, if the `global` cluster is only for internal network access and the platform needs to provide external network access, enter the pre-prepared domain name or `External IP`. The platform uses HTTPS access by default and does not enable HTTP. If you need to enable HTTP access, enable it in **Advanced Settings** (not recommended). **Note:** A domain name must be entered in the following cases, • A disaster recovery plan for the `global` cluster is planned; • The platform needs to support IPv6 access. **Tip:** If you need to configure more platform access addresses, you can add them in **Other Settings > Other Platform Access Addresses** in the next step. Or, after installation, add them in platform management according to the user manual.
Certificate	The platform provides self-signed certificates to support HTTPS access by default. If you need to use a custom certificate, you can upload an existing certificate.

Image Repository	`Platform Deployment` uses the Registry deployed with the `global` cluster. `External` uses a customer-managed platform image registry. Its address and pull credentials must match the values passed to `setup.sh`, and the registry must already contain the complete target-version Core and Extension payload.
Container Network	The default subnet and Service network segment of the cluster cannot overlap. When using the Kube-OVN Overlay network, ensure that the container network and the host network are not in the same network segment, otherwise it may cause network exceptions.
Node Name	If you select `Host Name as Node Name`, ensure that the host names of all nodes are unique.
`global` **Cluster Platform Node Isolation**	Enable only when you plan to run application workloads in the `global` cluster. **After enabling:** • Nodes can be set to `Platform Exclusive`, i.e., only run platform components, ensuring platform and application workloads are isolated; • Workloads of the DaemonSet type are excluded.
Add Node	**Control Plane Node:** • Supports adding 1 or 3 control plane nodes (3 for high availability configuration); • If `Platform Exclusive` is enabled, `Deployable Applications` is forced to be disabled, and control plane nodes

only run platform components;

- If `Platform Exclusive` is disabled, you can choose whether to enable `Deployable Applications`, allowing control plane nodes to run application workloads.

Worker Node:

- If `Platform Exclusive` is enabled, `Deployable Applications` is forced to be disabled;
- If `Platform Exclusive` is disabled, `Deployable Applications` is forced to be enabled.

When using Kube-OVN, you can specify the node network card by entering the gateway name.

If the node availability check fails, adjust it according to the page prompt and add it again.

After reviewing the parameters, return to Installing to submit the installation.

Global Cluster Disaster Recovery

Disaster Recovery Path

This page documents disaster recovery for `global` clusters running on a **non-Alauda OS**. If the `global` cluster uses Alauda OS, follow Global Cluster Disaster Recovery on Alauda OS instead.

TOC

- Overview
  - Supported Disaster Scenarios
  - Unsupported Disaster Scenarios
- Notes
- Process Overview
- Required Resources
  - Network Requirements
- Procedure
  - Step 1: Install the Primary Cluster
  - Step 2: Install the Standby Cluster
  - Step 3: Enable etcd Synchronization
- Disaster Recovery Process
- Routine Checks
- Uploading Packages

Overview

This solution is designed for disaster recovery scenarios involving the `global` cluster. The `global` cluster serves as the control plane of the platform and is responsible for managing other clusters. To ensure continuous platform service availability when the `global` cluster fails, this solution deploys two `global` clusters: a Primary Cluster and a Standby Cluster.

The disaster recovery mechanism is based on real-time synchronization of etcd data from the Primary Cluster to the Standby Cluster. If the Primary Cluster becomes unavailable due to a failure, services can quickly switch to the Standby Cluster.

Supported Disaster Scenarios

- Irrecoverable system-level failure of the Primary Cluster rendering it inoperable;
- Failure of physical or virtual machines hosting the Primary Cluster, making it inaccessible;
- Network failure at the Primary Cluster location resulting in service interruption;

Unsupported Disaster Scenarios

- Failures of applications deployed within the `global` cluster;
- Data loss caused by storage system failures (outside the scope of etcd synchronization);

The roles of **Primary Cluster** and **Standby Cluster** are relative: the cluster currently serving the platform is the Primary Cluster (DNS points to it), while the standby cluster is the Standby Cluster. After a failover, these roles are swapped.

Notes

- This solution only synchronizes etcd data of the `global` cluster; it does not include data from registry, chartmuseum, or other components;
- Monitoring and logging data is not synchronized either. Do not store workload cluster monitoring or logging data in the Primary Cluster or the Standby Cluster: install the VictoriaMetrics Center and the log storage that aggregate workload cluster data in a dedicated workload cluster, and let each `global` cluster collect, store, and query only its

own metrics and logs. Observability data held in the Primary Cluster is unavailable after a failover, so a Standby Cluster that depends on it cannot observe the platform it has just taken over;

- In favor of facilitating troubleshooting and management, it is recommended to name nodes in a style like `standby-global-m1`, to indicates which cluster the node belongs to (Primary or Standby).
- Disaster recovery of application data within the cluster is not supported;
- Stable network connectivity is required between the two clusters to ensure reliable etcd synchronization;
- If the clusters are based on heterogeneous architectures (e.g., x86 and ARM), use a dual-architecture installation package;
- The following namespaces are excluded from etcd synchronization. If resources are created in these namespaces, users must back them up manually:

```
cpaas-system
cert-manager
default
global-credentials
cpaas-system-global-credentials
kube-ovn
kube-public
kube-system
nsx-system
cpaas-solution
kube-node-lease
kubevirt
nativestor-system
operators
```

- If both clusters are set to use built-in image registries, container images must be uploaded separately to each;
- If the Primary Cluster deploys **Alauda DevOps Eventing v3** (knative-operator) and instances thereof, the same components must be pre-deployed in the standby cluster.

Process Overview

1. Prepare a unified domain name for platform access;
2. Point the domain to the **Primary Cluster's** VIP and install the **Primary Cluster**;
3. Temporarily switch DNS resolution to the standby VIP to install the Standby Cluster;
4. Copy the ETCD encryption key of the **Primary Cluster** to the nodes that will later be the control plane nodes of Standby Cluster;
5. Install and enable **Alauda Container Platform Global Cluster Disaster Recovery**;
6. Verify sync status and perform regular checks;
7. In case of failure, switch DNS to the standby cluster to complete disaster recovery.

Required Resources

- A unified domain which will be the `Platform Access Address`, and the TLS certificate plus private key for serving HTTPS on that domain;
- A dedicated virtual IP address for each cluster — one for the **Primary Cluster** and another for the Standby Cluster;

Network Requirements

The unified domain is a prerequisite for disaster recovery. During normal operation, the domain resolves only to the **Primary Cluster** VIP. The Standby Cluster uses the domain to access platform services on the Primary Cluster.

Before installing either cluster, complete both of the following network preparations:

- Configure each cluster's load balancer to forward the required TCP ports on its VIP to the control plane nodes behind that VIP. Configure port `80` only when users access the platform over HTTP.
- Allow the required TCP ports in **both directions** between the Primary Cluster and Standby Cluster networks. Apply the rules to firewalls, security groups, router ACLs, and any other inter-site network controls.

TCP port	Service	Why it is required
`443`	ACP platform HTTPS	The Standby Cluster uses the Platform Access Address to reach the active platform. **Alauda Container Platform Global Cluster Disaster Recovery** uses this address, and logging and monitoring components send alert callbacks to it.
`80`	ACP platform HTTP	Required only when users access the platform over HTTP. The connectivity requirement is otherwise the same as for port `443`.
`6443`	Kubernetes API server	During installation or reinstallation of **Alauda Container Platform Global Cluster Disaster Recovery**, the Standby Cluster connects to the active cluster API server to obtain the etcd CA material required for synchronization.
`11443`	Built-in image registry	The Standby Cluster pulls platform and plugin images from the registry configured through the Platform Access Address.
`2379`	etcd	**Alauda Container Platform Global Cluster Disaster Recovery** on the Standby Cluster reads etcd data from the Primary Cluster and writes it to the local standby etcd.

The network policy must be bidirectional because the cluster roles are reversed after failover. Before failover, the effective service traffic is normally from Standby to Primary. After failover, the former Primary becomes the new Standby and must reach the new Primary on the same ports. This requirement does not make etcd replication bidirectional: **Alauda Container Platform Global Cluster Disaster Recovery** runs only on the current Standby Cluster and writes synchronized data to its local etcd.

Procedure

Step 1: Install the Primary Cluster

NOTES OF DR (Disaster Recovery Environment) INSTALLING

While installing the primary cluster of the DR Environment,

- First of all, documenting all of the parameters set while following the guide of the installation web UI. It is necessary to keep some options the same while installing the standby cluster.
- A **User-provisioned** Load Balancer MUST be preconfigured to route traffic sent to the virtual IP. The `Self-built VIP` option is NOT available.
- The `Platform Access Address` field MUST be a domain, while the `Cluster Endpoint` MUST be the virtual IP address.
- Both clusters MUST be configured to use `An Existing Certificate` (has be the same one), request a legit certificate if necessary. The `Self-signed Certificate` option is NOT available.
- When `Image Repository` is set to `Platform Deployment`, both `Username` and `Password` fields MUST NOT be empty; The `IP/Domain` field MUST be set to the domain used as the `Platform Access Address`.
- Both `HTTP Port` and `HTTPS Port` fields of `Platform Access Address` MUST be 80 and 443.
- When coming to the second page the of the installation guide (Step: `Advanced`), the `Other Platform Access Addresses` field MUST include the virtual IP of current Cluster.

Refer to the following documentation to complete installation:

- Prepare for Installation
- Installing

Step 2: Install the Standby Cluster

1. Temporarily point the domain name to the standby cluster's VIP;
2. Log into the first control plane node of the **Primary Cluster** and copy the etcd encryption config to all standby cluster control plane nodes:

```
# Assume the primary cluster control plane nodes are 1.1.1.1, 2.2.2.2 &
3.3.3.3
# and the standby cluster control plane nodes are 4.4.4.4, 5.5.5.5 & 6.
6.6.6
for i in 4.4.4.4 5.5.5.5 6.6.6.6  # Replace with standby cluster contro
l plane node IPs
do
  ssh "<user>@$i" "sudo mkdir -p /etc/kubernetes/"
  scp /etc/kubernetes/encryption-provider.conf "<user>@$i:/tmp/encrypti
on-provider.conf"
  ssh "<user>@$i" "sudo install -o root -g root -m 600 /tmp/encryption-
provider.conf /etc/kubernetes/encryption-provider.conf && rm -f /tmp/en
cryption-provider.conf"
done
```

3. Install the standby cluster in the same way as the primary cluster

> **NOTES FOR INSTALLING STANDBY CLUSTER**
>
> While installing the standby cluster of the DR Environment, the following options MUST be set to the same as the **primary cluster**:
>
> - The `Platform Access Address` field.
> - All fields of `Certificate`.
> - All fields of `Image Repository`
> - Important: ensure the credentials of image repository and the ACP admin user match those set on the **Primary Cluster**.
>
> and MAKE SURE you followed the `NOTES OF DR (Disaster Recovery Environment) INSTALLING` in Step 1.

Refer to the following documentation to complete installation:

- Prepare for Installation
- Installing

Step 3: Enable etcd Synchronization

1. Before installing the plugin, get the bearer token for accessing the active global cluster API server from the active global cluster. Copy the command output and use it when creating the `etcd-sync-active-cluster-token` Secret in the standby global cluster under the `cpaas-system` namespace. The Secret stores the token under the data key `token`.

   ```
   # Run this command on the active cluster.
   kubectl -n cpaas-system get secret k8sadmin -o jsonpath='{.data.token}'
   | base64 -d
   ```

   Copy the output value, then run this command on the standby cluster:

   ```
   ACTIVE_CLUSTER_TOKEN='<paste-the-token-from-the-active-cluster>'
   kubectl -n cpaas-system create secret generic etcd-sync-active-cluster-
   token \
     --from-literal=token="${ACTIVE_CLUSTER_TOKEN}" \
     --dry-run=client -o yaml | kubectl apply -f -
   ```

2. When applicable, configure the load balancer to forward port `2379` to control plane nodes of the corresponding cluster. ONLY TCP mode is supported; forwarding on L7 is not supported.

   > **INFO**
   >
   > Port forwarding through a load balancer is not required. If direct access from the standby cluster to the active global cluster is available, specify the etcd addresses via **Active Global Cluster ETCD Endpoints**.

3. Access the **standby global cluster** Web Console using its VIP, and switch to **Administrator** view.
4. Navigate to **Marketplace > Cluster Plugins**, select the `global` cluster.
5. Find **Alauda Container Platform Global Cluster Disaster Recovery**, click **Install**, and configure these parameters:

   - Set **Active Global Cluster VIP** to the VIP of the active global cluster.
   - When not forwarding port `2379` through the load balancer, configure **Active Global Cluster ETCD Endpoints** correctly.

- Set **Standby Cluster ETCD Endpoints** to the standby cluster etcd address. Use the default value unless the local etcd service is exposed through a different endpoint.
- Set **Active Global Cluster Token Secret** to `etcd-sync-active-cluster-token`.
- Use the default value of **Data Check Interval**.
- Leave **Print detail logs** disabled unless troubleshooting.

During installation, the system runs the `etcd-sync-bootstrap` Job before the `etcd-sync` Deployment starts. The plugin installation continues only after the Job prepares `remote-etcd-ca`, `remote-etcd-issuer`, and `remote-etcd-client`.

Verify the bootstrap Job and runtime resources:

```
kubectl get job -n cpaas-system etcd-sync-bootstrap
kubectl logs -n cpaas-system job/etcd-sync-bootstrap
kubectl get secret -n cpaas-system remote-etcd-ca
kubectl get issuer -n cpaas-system remote-etcd-issuer
kubectl get certificate -n cpaas-system remote-etcd-client
kubectl get secret -n cpaas-system remote-etcd-client
```

Verify the sync Pods are running on the standby cluster and identify the current leader:

```
kubectl get po -n cpaas-system -l app=etcd-sync
kubectl get lease -n cpaas-system etcd-sync-mirror
leader_pod=$(kubectl get lease -n cpaas-system etcd-sync-mirror -o jsonpa
th='{.spec.holderIdentity}')
kubectl logs -n cpaas-system "$leader_pod" | grep -E "Acquired leader lea
se|Start Sync update"
```

If resources with `ownerReference` dependencies need to be resynchronized, recreate the current leader Pod after `Start Sync update` appears:

```
leader_pod=$(kubectl get lease -n cpaas-system etcd-sync-mirror -o jsonpa
th='{.spec.holderIdentity}')
kubectl delete po -n cpaas-system "$leader_pod"
```

Check sync status:

```
mirror_svc=$(kubectl get svc -n cpaas-system etcd-sync-monitor -o jsonpat
h='{.spec.clusterIP}')
ipv6_regex="^[0-9a-fA-F:]+$"
if [[ $mirror_svc =~ $ipv6_regex ]]; then
  mirror_host="[$mirror_svc]"
else
  mirror_host="$mirror_svc"
fi
curl -g "http://${mirror_host}/check"
```

Output explanation:

- `LOCAL ETCD missed keys`: Keys exist in the Primary but are missing from the standby. Often caused by GC due to resource order during sync. Restart the current etcd-sync leader Pod to fix;
- `LOCAL ETCD surplus keys`: Extra keys exist only in the standby cluster. Confirm with ops team before deleting these keys from the standby.

If the following components are installed, restart their services:

- Alauda Container Platform Log Storage for Elasticsearch:

  ```
  kubectl delete po -n cpaas-system -l service_name=cpaas-elasticsearch
  ```

- Alauda Container Platform Advanced Monitoring:

  ```
  kubectl delete po -n cpaas-system -l 'service_name in (alertmanager,vms
  elect,vminsert)'
  ```

Disaster Recovery Process

> **Verify primary/standby consistency before uninstalling Alauda Container Platform Global Cluster Disaster Recovery**
>
> This procedure uninstalls **Alauda Container Platform Global Cluster Disaster Recovery**. Before you uninstall it, confirm that the standby cluster data is consistent with the primary cluster.

Uninstalling the plugin while the standby is missing data that the primary holds can cause owner references to resolve incorrectly, and workload-cluster node `Machine` objects — including Alauda OS clusters, where this destroys the backing virtual machine — may be deleted. If the consistency check reports missing or surplus keys, do not uninstall the plugin; resolve the inconsistency or contact technical support first.

1. Restart Elasticsearch on the standby cluster in case it is necessary:

```
# Copy installer/res/packaged-scripts/for-upgrade/ensure-asm-template.s
h to /root:
# DO NOT skip this step

# switch to the root user if necessary
sudo -i

# check whether the Log Storage for Elasticsearch is installed on globa
l cluster
_es_pods=$(kubectl get po -n cpaas-system | grep cpaas-elasticsearch |
awk '{print $1}')
if [[ -n "${_es_pods}" ]]; then
    # In case the script returned the 401 error, restart Elasticsearch
    # then execute the script to check the cluster again
    bash /root/ensure-asm-template.sh

    # Restart Elasticsearch
    xargs -r -t -- kubectl delete po -n cpaas-system <<< "${_es_pods}"
fi
```

2. Verify data consistency in the standby cluster (same check as in Step 3). If the check reports missing or surplus keys, the standby is not consistent with the primary: do not continue to the next step. Resolve the inconsistency or contact technical support first. On **both** clusters, also confirm that no `Machine` nodes are in a non-running state, and resolve any that are before continuing:

```
kubectl get machines.platform.tkestack.io
```

3. Uninstall **Alauda Container Platform Global Cluster Disaster Recovery**;
4. Remove port forwarding for `2379` from both VIPs;

5. Switch the platform domain DNS to the standby VIP, which now becomes the Primary Cluster;
6. Verify DNS resolution:

```
kubectl exec -it -n cpaas-system deployments/sentry -- nslookup <platfo
rm access domain>
# If not resolved correctly, restart coredns Pods and retry until succe
ss
```

7. Clear browser cache and access the platform page to confirm it reflects the former standby cluster;
8. Restart the following services (if installed):
   - Alauda Container Platform Log Storage for Elasticsearch:

```
kubectl delete po -n cpaas-system -l service_name=cpaas-elasticsearch
```

   - Alauda Container Platform Advanced Monitoring:

```
kubectl delete po -n cpaas-system -l 'service_name in (alertmanager,v
mselect,vminsert)'
```

   - cluster-transformer:

```
kubectl delete po -n cpaas-system -l service_name=cluster-transformer
```

9. If workload clusters send monitoring data to the Primary, restart warlock in the workload cluster:

```
kubectl delete po -n cpaas-system -l service_name=warlock
```

10. On the original Primary Cluster, repeat the Enable etcd Synchronization steps to convert it into the new standby cluster.

Routine Checks

Regularly check sync status on the standby cluster:

```
mirror_svc=$(kubectl get svc -n cpaas-system etcd-sync-monitor -o jsonpat
h='{.spec.clusterIP}')
ipv6_regex="^[0-9a-fA-F:]+$"
if [[ $mirror_svc =~ $ipv6_regex ]]; then
  mirror_host="[$mirror_svc]"
else
  mirror_host="$mirror_svc"
fi
curl -g "http://${mirror_host}/check"
```

If any keys are missing or surplus, follow the instructions in the output to resolve them.

Uploading Packages

For details of the `violet push` subcommand, please refer to Upload Packages.

Installation Troubleshooting

Use this page when the installer does not progress, an installation resource fails, or post-install validation reports an image or artifact problem.

TOC

Common Stalls and Where to Look

Image Pull and Artifact Failures

Installer Cleanup

Collect Evidence Before Escalation

Common Stalls and Where to Look

Collect the signal in the second column before changing the environment or opening a support ticket.

Symptom	First place to look	What you are looking for
Web UI does not load	Terminal output of `setup.sh` and `nerdctl ps` on the installation node	The `minialauda-control-plane` container is Running and port `8080` is reachable from the browser.
Installer log stops advancing	`tail -f /var/cpaas/data/installer.log`	The last completed phase and the first repeated or failed operation.
Stuck on control plane provisioning	`kubectl get machines -A` and `kubectl describe` on a machine that is not `Ready`	The `Bootstrap` and `Infrastructure` conditions and the node address in `status.addresses`.
Stuck on network and Core add-ons	`kubectl -n kube-system get pods` on the new cluster	Kube-OVN, CoreDNS, and kube-proxy Pods are `Running`; image-pull failures usually identify a registry path problem.
AppRelease shows `Failed`	`kubectl describe apprelease <name> -n <namespace>`	Status conditions and recent Events that identify the chart or dependency failure.
`ClusterModule/global` does not become healthy	`kubectl describe clustermodule global`	The status condition that identifies the blocking Core or Aligned module.

Image Pull and Artifact Failures

Symptom	Checks	Recovery
Pod is `ImagePullBackOff` or `ErrImagePull`	Run `kubectl describe pod <pod> -n <namespace>` and test the configured registry from the failing node.	Correct the reported DNS, route, firewall, CA trust, pull credential, registry availability, or missing-manifest problem.
A required Core or Aligned application is unavailable after installation	Inspect the corresponding `AppRelease`, `ModuleInfo`, `ModulePlugin`, and Pod Events. If package availability is still unclear, inspect the matching entry in `kubectl get productbase base -o yaml` and the exact `repository:tag` in the registry.	If the registry confirms that the required manifest is missing, upload the matching installation-version package. For authorization, CA, DNS, routing, or connectivity errors, correct the reported registry problem instead of republishing the package. Ignore unrelated `Absent` entries for optional or uninstalled catalog packages.
A recovered or replacement node cannot pull an installed-version image	Inspect the failing Pod and registry retention records.	Restore the referenced digest, then correct registry availability, backup, or retention policy before retrying recovery.

For upload-command failures that occur before the installer starts, use External Platform Image Registry Troubleshooting.

Installer Cleanup

The installer normally removes itself after installation. If the installer container remains 30 minutes after installation completes, run this command on the installation node:

```
nerdctl rm -f minialauda-control-plane
```

Collect Evidence Before Escalation

Capture `/var/cpaas/data/installer.log`, the relevant `kubectl describe` output, failing resource conditions, and the exact Core Package version and architecture. Do not include registry passwords or Secret contents.

Validation

Validate ACP Core and the required Aligned Extensions after the installation completes and before you continue with post-install tasks. These checks apply to both installation paths. If the `global` cluster runs Alauda OS, also complete Additional Checks for Alauda OS.

TOC

Prerequisites

Validate Web UI Access

Validate Core Applications, Extensions, and Pods

Validate the Platform Image Source

Additional Checks for Alauda OS

Next Step

Prerequisites

- The ACP installation completed.
- You can access the platform Web UI address: the address shown by the installer on a non-Alauda OS, or the platform access address set in `console.globalHost` on Alauda OS.
- You can run `kubectl` against the new `global` cluster: from the installation node on a non-Alauda OS, or with a kubeconfig for the new `global` cluster on Alauda OS. On Alauda OS, add `--kubeconfig <global-kubeconfig>` to the commands on this page.

Validate Web UI Access

After the installation is complete, open the platform Web UI and verify that the platform login page is reachable. On a non-Alauda OS, the installer displays the platform access URL; click **Access** to open it. On Alauda OS, open `https://<platform-access-address>`, the address set in `console.globalHost`.

Validate Core Applications, Extensions, and Pods

Run the following commands against the new `global` cluster to confirm the installed state. On a non-Alauda OS, run them on a control plane node.

```
# All nodes are Ready
kubectl get nodes

# ClusterModule/global reports the base module as healthy
kubectl get clustermodule global -o jsonpath='{.status.phase}'

# No AppRelease is in a failed state
kubectl get apprelease -A

# Core and Aligned modules report their current phase and version
kubectl get moduleinfo -l cpaas.io/cluster-name=global

# No Pod is stuck outside Running or Completed
kubectl get pod --all-namespaces | awk '{if ($4 != "Running" && $4 != "Co
mpleted")print}' | awk -F'[/ ]+' '{if ($3 != $4)print}'
```

The installation is healthy when:

- All `global` cluster nodes are `Ready`.
- `ClusterModule/global` reports a healthy phase.
- Every AppRelease is in a non-failed state.
- The following applications are installed, and their corresponding `ModuleInfo` resources report `Running` at the target version:

- **Alauda Container Platform Web Console**
- **Alauda Container Platform Web Console Central**
- **Alauda Container Platform Marketplace Web Console**
- **Alauda Container Platform Application Catalog**
- **Alauda Container Platform Observability Essentials**
- **Alauda Container Platform Essentials**
- **Alauda Container Platform Cluster Essentials**
- **Alauda Container Platform GatewayAPI Plugin**

- Critical Pods in `cpaas-system` are `Running` or `Completed`.

In the Web Console, open **Administrator > Marketplace > Cluster Plugins** and confirm that the required applications are available without an installation failure.

Validate the Platform Image Source

Confirm that `ProductBase` records the expected platform image source:

```
kubectl get productbase base \
  -o jsonpath='{.spec.registry.address}{"\t"}{.spec.registry.external}
{"\n"}'
```

For an external registry, the address must match the prepared registry and the second value must be `true`.

For an authenticated external registry, confirm that the pull Secret exists without printing its contents:

```
kubectl -n cpaas-system get secret global-registry-auth
```

Confirm that the `global` cluster records the same registry address:

```
kubectl -n cpaas-system get cluster global \
  -o jsonpath='{.metadata.annotations.cpaas\.io/registry-address}{"\n"}'
```

`ProductBase` also contains catalog entries for optional or uninstalled Extensions. Do not require every entry in `ProductBase.status.artifacts` to be `Ready`; an unrelated `Absent` entry does not by itself indicate an installation failure.

Confirm that the required Extension catalog entries exist and that Pods are not blocked by image pulls:

```
kubectl get moduleplugins

kubectl get pods --all-namespaces \
  | awk '$4 ~ /ImagePullBackOff|ErrImagePull/'
```

The Pod command must produce no output. Installation acceptance is based on the required applications, their `ModuleInfo` and `AppRelease` state, and their running Pods. If a required installed application is unavailable, follow Installation Troubleshooting to distinguish a missing manifest from DNS, routing, firewall, CA trust, pull credential, or registry availability problems.

Additional Checks for Alauda OS

If the `global` cluster runs Alauda OS, also verify the following with a kubeconfig for the new `global` cluster. Complete these checks before you decommission the bootstrap cluster.

```
kubectl --kubeconfig <global-kubeconfig> get nodes
kubectl --kubeconfig <global-kubeconfig> get clusters.platform.tkestack.i
o global \
  -o jsonpath='{.status.phase}'
kubectl --kubeconfig <global-kubeconfig> get pods -n cpaas-system
kubectl --kubeconfig <global-kubeconfig> get clustermodule global
```

The installation is successful when all of the following conditions are true:

- The installer progress API reports `status: Success` and `type: Complete`. See Signals During Installation.

- All `global` cluster nodes are `Ready`.
- Critical Pods in `cpaas-system` are `Running` or `Completed`.
- `ClusterModule/global` reports the base module as healthy.

Next Step

After validation succeeds, continue with Next Steps. On Alauda OS, first decommission the bootstrap cluster.

Next Steps

After Core and the required Aligned Extensions pass validation, complete the following post-install tasks according to your scenario.

Priority	Task	When to use	Continue with
Recommended immediately	Install the Product Docs plugin.	Install this plugin so in-console help links can open product documentation.	Install Product Docs Plugin
Conditional	Create workload clusters or connect existing clusters.	Use this path when your architecture requires workload clusters separate from the `global` cluster. To create workload clusters on Alauda OS, first upload and install the infrastructure provider for the target IaaS platform and the Cluster Bootstrap Provider on the `global` cluster, regardless of which IaaS platform the `global` cluster runs on.	Clusters
Conditional	Upload and install additional Extensions Packages.	Use this path for packages from **Common ACP Extensions** that were not part of the eight required installation packages, or when you need other platform capabilities.	Extend

Priority	Task	When to use	Continue with
Recommended before broad user onboarding	Configure identity providers, users, roles, projects, and project membership.	Use this path before onboarding administrators, project users, or application teams.	Users and Roles and Projects
Recommended before application workloads	Prepare storage, networking, registry access, and workload prerequisites.	Use this path before application teams deploy workloads.	Storage, Networking, Registry, and Workload Clusters
Recommended before production operation	Review backup, upgrade, monitoring, and operational documentation.	Use this path before operating the platform for production workloads.	Backup, Upgrade, and Observability

These tasks do not all apply to every installation. Follow the paths that match your deployment architecture and operational requirements.

TOC

Install Product Docs Plugin

- Open Cluster Plugins
- Install the Plugin

Install Product Docs Plugin

The **Alauda Container Platform Product Docs** plugin provides access to product documentation within the platform. All help links throughout the platform direct users to this

documentation. If this plugin is not installed, clicking help links in the platform results in 404 access errors.

1 Open Cluster Plugins

Navigate to **Administrator**. In the left sidebar, click **Marketplace** > **Cluster Plugins** and select the `global` cluster.

2 Install the Plugin

Locate the **Alauda Container Platform Product Docs** plugin and click **Install**.