etcd

Overview

etcd stores Kubernetes control-plane state for a cluster. Platform administrators use etcd guidance to understand encryption, operational health, capacity considerations, and the handoff to backup and restore procedures.

Need	Continue with
Enable and understand etcd encryption key rotation.	etcd encryption
Review operational practices and route to health, capacity, and backup topics.	etcd practices
Back up or restore etcd data.	etcd backup and restore

Use these etcd pages to plan control-plane state encryption, health, capacity, and maintenance. When you need to protect or recover etcd data, continue with the backup and restore procedure.

etcd Practices

Use etcd practices to plan control-plane state operations before making changes that affect cluster recovery or availability.

Practice area	Guidance
Backup before risky maintenance	Back up etcd before control-plane maintenance, node operating system changes, or other operations that can affect cluster state. Follow etcd backup and restore.
Encryption	Use etcd encryption when sensitive Kubernetes resources need encryption at rest through the documented encryption manager.
Capacity and health	Monitor etcd health, disk latency, and available capacity through the observability and cluster operation tools available in your environment.
Recovery scope	Use etcd backup and restore for control-plane state recovery. Registry data, monitoring data, logging data, and application data need their own backup and recovery paths.

Do not use this page as a production hardening checklist. Follow topic-specific support guidance for validated recovery procedures, scale limits, and maintenance windows.

etcd Encryption

Install and operate etcd Encryption Manager in ACP to automate etcd data encryption key rotation within your clusters.

It ensures that sensitive data stored in etcd, such as secrets and configmaps, is encrypted using a secure algorithm, enhancing your cluster's security.

TOC

Key Strategies

The plugin supports two key strategies. Pick the one that matches your cluster role.

Strategy	When to use	How keys are produced
`random`	Default for workload and DCS clusters running standalone (no etcd synchronization).	Each rotation generates a fresh random key locally.
`deterministic`	**Required** for `global` clusters, especially when paired in a DR setup via the etcd Synchronizer.	Each key is derived deterministically from a shared root Secret and a shared SeedBundle, so the Active and Standby clusters always reach the same key for the same revision.

In `deterministic` mode, the runtime Active / Standby role is detected automatically; it is not a manual install parameter.

Installation

The plugin supports two installation paths depending on the target cluster:

- **Workload / DCS clusters** — use the default `random` key strategy.
- **Global clusters** — must use the `deterministic` key strategy (see Global Clusters (Deterministic Mode) below).

See Cluster Plugin for the general plugin installation workflow.

Workload / DCS Clusters (random strategy)

- Supported cluster types: On-Premises, DCS.
- No additional installation parameters are required — install via the standard Cluster Plugin workflow.

Global Clusters (Deterministic Mode)

> **DR pair requirements**
>
> For a Global DR pair, install the plugin **on both the Active and Standby clusters** in `deterministic` mode, and configure **identical** `replication_group_id` and `root_secret_name` (with matching root key material) on both sides. Asymmetric or mismatched configuration causes the Standby to derive different per-revision keys than the Active and breaks failover.

Prerequisites

- The cluster must be a `global` cluster. When `global.cluster.name` is `global`, the chart enforces `etcdEncryption.keyStrategy=deterministic`; the plugin UI also defaults the field to `deterministic` for clusters labeled `is-global=true`.

> **etcd Synchronizer version**
>
> The **etcd Synchronizer** plugin must be installed at version **v4.3.7 or later** before installing the etcd Encryption Manager in `deterministic` mode. Earlier versions do not replicate the shared SeedBundle, so the Standby cluster cannot derive matching keys.

Plugin Parameters

The plugin form exposes the following parameters for deterministic installation. Defaults reflect the current `plugin-config.yaml` shipped with the chart.

Plugin parameter	Required	Default	Purpose
`key_strategy`	Yes	`random`; dynamically defaults to `deterministic` for `global` clusters.	Selects between random per-rotation key generation and deterministic derivation.
`replication_group_id`	Yes	The plugin UI prefills the current cluster	Identifies the Active / Standby DR group

Plugin parameter	Required	Default	Purpose
		name (not recommended — see below).	and participates in HKDF domain separation.
`root_secret_name`	Yes	`etcd-derivation-root`	Name of the Secret in `kube-system` that holds the deterministic root key material.

The `rootSecretRef.namespace` is fixed to `kube-system` and is not exposed in the form.

`key_strategy`

Value	Meaning
`random`	Single-cluster random rotation.
`deterministic`	Derived keys based on a shared root Secret and SeedBundle.

For `global` clusters this must be `deterministic`.

`replication_group_id`

Property	Value
Purpose	Identifies one Active / Standby DR replication group and participates in HKDF domain separation.
Consistency requirement	The Active and Standby clusters in the same DR group **must** use exactly the same value.
Default in UI	Current cluster name.
Recommended to keep the default	**No.**

Recommendations:

- For DR clusters built with the etcd Synchronizer, both sides of the same DR group must share the same `replication_group_id`.
- To avoid leaking real business, environment, or cluster identity, do **not** reuse the cluster name even when the UI prefills it. Plan a separate opaque identifier ahead of installation.
- A UUID-style identifier may be a good fit, for example `6f1b9e2c-7c3a-4a9c-8b72-2fd0f8f3c1ab`.

To generate a UUID-style value from the command line:

```
openssl rand -hex 16 | sed -E 's/^(.{8})(.{4})(.{4})(.{4})(.{12})$/\1-\2-\3-\4-\5/'
```

> **Use the same value on both clusters**
>
> Generate the value **once**, then configure the same `replication_group_id` on the Active and Standby clusters of the DR group. Generating different values on each side will produce divergent encryption keys.

`root_secret_name`

Property	Value
Purpose	Refers to the deterministic root key Secret.
Default	`etcd-derivation-root`
Fixed namespace	`kube-system`
Cross-cluster requirement	All clusters in the same replication group must use the same root key material.

The plugin form exposes only the Secret name; the namespace is always `kube-system` at render time.

Preparing the Root Secret

The deterministic root Secret must exist in `kube-system` before installing the plugin. Prepare it in two steps:

1. Generate 32 bytes of cryptographically strong random material into `./root-key.bin`. The bytes must come from a high-entropy source — typically the operating system CSPRNG. `openssl rand` is a safe default; equivalent OS-CSPRNG-backed commands such as `head -c 32 /dev/urandom > ./root-key.bin` also work. Do **not** use shell `$RANDOM`, language-level `rand()`, or any time-seeded generator.

   ```
   openssl rand -out ./root-key.bin 32
   ```

2. Create the Secret in `kube-system` from that file:

   ```
   kubectl -n kube-system create secret generic etcd-derivation-root \
     --from-file=root-key=./root-key.bin
   ```

If you use a different Secret name, replace `etcd-derivation-root` above with the value you will enter as `root_secret_name`.

> **Use the same key material on both clusters**
>
> Generate `./root-key.bin` **once**, then transport the **same file** to every cluster in the replication group and run the `kubectl create secret` step on each. Do **not** regenerate the key material per cluster — different root key bytes will produce divergent encryption keys.

Advanced (chart values, not exposed in the plugin form)

▶ Show advanced chart values

How it Works

Upon installation, an `etcd-encryption-manager` controller is deployed in the `kube-system` namespace, which:

- Periodically rotates etcd data encryption keys.
- Retains the 8 most recent keys for rollback compatibility.
- Updates encryption configurations on all control nodes.
- Triggers `kube-apiserver` to hot reload new keys.
- Automatically migrates resources to re-encrypt data with new keys.

Cluster stability is maintained throughout these operations.

When the plugin is installed in `deterministic` mode on a `global` DR pair, the controller additionally:

- Detects whether it is acting as **Active** or **Standby** at runtime — this is not a manual configuration.
- On the Active side, generates the SeedBundle and lets the etcd Synchronizer replicate it to the Standby.
- On the Standby side, derives the same per-revision keys from the shared SeedBundle and the root Secret, so a failover produces identical encryption keys without manual key copy.

Default Configuration

Parameter	Value
Encrypted resources	secrets, configmaps
Encryption algorithm	256-bit AES-GCM
Rotation interval	168 hours (7 days)

Operations Guide

Configuration Files

Path	Content
`/etc/kubernetes/encryption-provider.conf`	Current encryption configuration
`/etc/kubernetes/encryption-provider-history.bak`	Historical key records (for recovery)
`/etc/kubernetes/encryption-provider-bak/`	Expired encryption configuration versions

Checking Status

Run the following command to check the current rotation status:

```
kubectl get EtcdEncryptionConfig default -o yaml
```

Example output:

```yaml
apiVersion: cluster.alauda.io/v1alpha1
kind: EtcdEncryptionConfig
metadata:
  name: default
spec:
  resources:
    - secrets
    - configmaps
  rotationInterval: 168h0m0s
  type: aesgcm
status:
  deployStatus:
    192.168.100.1:
      revision: 3
      state: Success
    192.168.100.2:
      revision: 3
      state: Success
    192.168.100.3:
      revision: 3
      state: Success
  migration:
    completeTimestamp: "2025-05-27T05:47:01Z"
    resources:
      - secrets
      - configmaps
    revision: 3
    state: Success
  revision: 3
```