etcd

Overview

etcd Practices

etcd Encryp

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 Control Plane Database Encryption 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
- Installation
  - Workload / DCS Clusters (random strategy)
  - Global Clusters (Deterministic Mode)
    - Prerequisites
    - Plugin Parameters
    - Preparing the Root Secret
    - Advanced (chart values, not exposed in the plugin form)
- How it Works
  - Default Configuration
- Operations Guide
  - Configuration Files
  - Checking Status
  - Approving a Deterministic Key Rotation
    - Manual Approval (Recommended)

Timed Approval (Not Recommended)

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 Global Cluster Disaster Recovery plugin.	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 plugin enforces `etcdEncryption.keyStrategy=deterministic`; the plugin configuration UI also defaults the field to `deterministic` for clusters labeled `is-global=true`.

> **Global Cluster Disaster Recovery version**
>
> The **Global Cluster Disaster Recovery** plugin must be installed at version **v4.3.7 or later** before installing Control Plane Database Encryption 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.

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.

Plugin parameter	Required	Default	Purpose
`replication_group_id`	Yes	The plugin UI prefills the current cluster name (not recommended — see below).	Identifies the Active / Standby DR group 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 Global Cluster Disaster Recovery plugin, 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

> **Set activationPolicy to manual for Global clusters**
>
> For a Global Active/Standby pair, explicitly set `activationPolicy` to `manual`. A new key revision must not become active on the Active cluster until an operator has confirmed that the

matching revision is ready on the Standby cluster. If `activationPolicy` is omitted, the plugin defaults to `timed-approval` with a `30m` delay — see Timed Approval for the risks this carries.

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 plane 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 Global Cluster Disaster Recovery plugin 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

Parameter	Value
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
```

Approving a Deterministic Key Rotation

When the encryption manager rotates a key in `deterministic` mode, the new revision enters a `WaitingApproval` phase on the Active cluster. The key does not take effect until the operator approves it. This gate exists because the Active cluster must not encrypt data with a key that the Standby cluster has not yet loaded — otherwise a failover would leave Standby unable to decrypt the new ciphertext.

Approval is always performed on the **Active cluster only**. Do not patch the Standby's `EtcdKeySeedBundleState`, and do not edit any resource's `status` field directly.

Manual Approval (Recommended)

Step 1 — Identify the pending revision.

Log in to a control-plane node of the **Active** cluster and list all bundle states:

```
kubectl get eksbs \
  -o custom-columns='NAME:.metadata.name,REVISION:.spec.revision,PHASE:.status.phase,APPROVED:.spec.approved'
```

Find the row with `PHASE=WaitingApproval` and `APPROVED=false`. Record its revision number:

```
REVISION=<revision-number>
STATE="seed-bundle-state-${REVISION}"
BUNDLE="seed-bundle-${REVISION}"
```

Step 2 — Verify Standby readiness.

Log in to a control-plane node of the **Standby** cluster and check the same revision:

```
kubectl get eksbs "$STATE" \
  -o custom-columns='REVISION:.spec.revision,PHASE:.status.phase,PREPARED_AT:.status.preparedAt'
```

The output must show `PHASE=Prepared` for the expected revision. **Stop** if the object is missing, still `Preparing`, or `Failed` — investigate the Global Cluster Disaster Recovery replication path before proceeding.

Step 3 — Cross-check derivation inputs.

Run the following command on **both the Active and Standby clusters** and compare the output. The `replicationGroupID`, `dimension`, and `derivationAlgorithm` must match exactly:

```
kubectl get eksb "$BUNDLE" \
  -o custom-columns='REVISION:.spec.revision,GROUP:.spec.replicationGroup
ID,DIMENSION:.spec.dimension,ALGORITHM:.spec.derivationAlgorithm'
```

Step 4 — Approve.

On the **Active** cluster, approve the pending revision:

```
kubectl patch eksbs "$STATE" \
  --type=merge -p '{"spec":{"approved":true}}'
```

The patch records the approval. Activation, deployment to control-plane nodes, and data migration happen asynchronously after this point.

Step 5 — Confirm activation completes.

On the **Active** cluster, check the bundle state and migration progress:

```
kubectl get eksbs "$STATE" \
  -o custom-columns='REVISION:.spec.revision,PHASE:.status.phase,MESSAG
E:.status.message'
kubectl get eec default \
  -o custom-columns='REVISION:.status.revision,MIGRATION_STATE:.status.mi
gration.state,MIGRATION_REVISION:.status.migration.revision'
```

The rotation is complete when:

- The bundle state reaches `PHASE=Active`.
- `eec/default` shows the new `REVISION` with `MIGRATION_STATE=Success`.

To check per-node deployment progress on the **Active** cluster:

```
kubectl get eec default \
  -o go-template='{{range $node, $status := .status.deployStatus}}{{print
f "%s\t%s\t%d\t%s\n" $node $status.state $status.revision $status.messag
e}}{{end}}'
```

Every control-plane node must report `state=Success` for the new revision. A bundle that remains `Activating` indicates deployment or migration is still in progress (or retrying after a failure).

Timed Approval (Not Recommended)

> **Timed approval risks data loss during failover**
>
> `timed-approval` is **not recommended** for Global Active/Standby pairs. The `approvalDelay` timer (default `30m`) is a local countdown on the Active cluster — it does not check whether the Standby has actually loaded the new key. If etcd synchronization is slow, the Standby network is partitioned, or the Standby control plane is under maintenance, the deadline can expire while the Standby is still unprepared. A failover during this window leaves Standby unable to decrypt data encrypted with the new key.

When `activationPolicy` is set to `timed-approval`, the controller automatically approves the pending revision once `approvalDelay` elapses after the `EtcdKeySeedBundleState` is created. The delay must be long enough for the full Standby preparation pipeline to complete:

Phase	What happens	Typical duration
Sync	The Global Cluster Disaster Recovery plugin replicates the SeedBundle to Standby	Seconds (depends on cross-region latency)
Reconcile	Standby controller derives the key and updates local EncryptionConfig	Seconds
Deploy	Deployer writes the new provider config to all Standby control-plane nodes	Seconds to minutes (scales with node count)
Reload	Standby kube-apiserver loads the new provider	Seconds to minutes (depends on whether automatic reload is enabled for the encryption provider config)

The default `30m` provides margin for typical single-region deployments with a small control plane. Increase it if your setup involves cross-region replication, large control planes, or clusters without automatic provider reload.

Required monitoring

If you choose `timed-approval`, you **must** verify that monitoring and alerting are working on **both** the Active and Standby clusters before enabling this policy.

The encryption manager exposes the following Prometheus metrics:

Metrics

Metric	Type	Descriptio
`etcd_encryption_manager_bundle_waiting_approval_seconds`	Gauge	Seconds tl bundle has in `WaitingA` Labeled by `activati` (`manual` `approval`
`etcd_encryption_manager_timed_approval_remaining_seconds`	Gauge	Seconds r before the approval d expires. N values me deadline h Only emitt `timed-ap` bundle is i `WaitingA`
`etcd_encryption_manager_bundle_activating`	Gauge	`1` when a is currentl `Activati`
`etcd_encryption_manager_bundle_activation_failed`	Gauge	`1` when a `Activati` has an exp

Metric	Type	Descriptio
		deploymer migration f

Built-in alert rules

The plugin includes the following alerts:

Alert	Severity	Condition
`EtcdEncryptionManagerApprovalRequired`	warning	A `manual` bu has been in `WaitingAppr` for ≥ 5 minute
`EtcdEncryptionManagerApprovalOverdue`	critical	A `manual` bu has been in `WaitingAppr` for ≥ 30 minut
`EtcdEncryptionManagerTimedApprovalDeadlineApproaching`	warning	A `timed-approval` bu has ≤ 10 minu remaining bef auto-approval
`EtcdEncryptionManagerTimedApprovalDeadlinePassed`	critical	A `timed-approval` bu deadline has passed while `WaitingAppr`
`EtcdEncryptionManagerActivationFailed`	critical	An `Activati` bundle has a deployment o migration failu

Alert	Severity	Condition
`EtcdEncryptionManagerActivationStuck`	critical	A bundle has remained `Activating` 30 minutes.

These alerts fire on each cluster independently. For Global clusters, the critical alert to act on is `EtcdEncryptionManagerTimedApprovalDeadlineApproaching` on the **Active** cluster — it gives you a 10-minute window to check whether the Standby is `Prepared` and switch to `manual` if it is not.

> **Cross-cluster correlation**
>
> Each cluster runs its own metrics and alerting independently. The built-in alerts detect local issues — a stuck `WaitingApproval` on Active or a failed activation on Standby. To correlate the two (e.g., "Active is about to auto-approve but Standby is not yet `Prepared`"), aggregate metrics from both clusters into a central Prometheus / Thanos / Victoria Metrics instance.

Emergency response

If the `EtcdEncryptionManagerTimedApprovalDeadlineApproaching` alert fires and the Standby is not ready, switch the Active to `manual` to halt auto-approval. On the **Active** cluster:

```
kubectl patch eec default \
  --type=merge -p '{"spec":{"deterministic":{"activationPolicy":"manua
l"}}}'
```

Verify the current pending bundle has switched:

```
kubectl get eksbs \
  -o custom-columns='NAME:.metadata.name,REVISION:.spec.revision,PHASE:.s
tatus.phase,APPROVED:.spec.approved'
```

The bundle must still show `APPROVED=false`. Switching back to `timed-approval` later affects only future bundles — the current bundle remains under manual control. If the bundle has already moved to `Activating`, the deadline has already taken effect; follow the activation checks above and do not assume that changing the policy rolls it back.