Storage

Introduction

Introduction

Concepts

Core Concepts

Persistent Volume (PV)

Persistent Volume Claim (PVC)

Generic Ephemeral Volumes

emptyDir

hostPath

ConfigMap

Secret

StorageClass

Container Storage Interface (CSI)

Persistent Volume

Dynamic Persistent Volumes vs. Static Pe

Lifecycle of Persistent Volumes

Access Mod

Access Modes

Volume Modes

Storage Featur

Conclusion

Guides

Creating CephFS File Storage T

Deploy Volume Plugin

Create Storage Class

Creating CephRBD Block Stora

Deploy Volume Plugin

Create Storage Class

Create Topo

Background Inf

Deploy Volume

Create Storage

Follow-up Actio

Deploy Volume Snapshot Component

Next steps

Creating an NFS Shared Storage Class

Creating a F

Prerequisites

Example Persis

/ by

/ by

Creating PVCs

Prerequisites

Example PersistentVolumeClaim:

Creating a Persistent Volume Claim by usi

Creating a Persistent Volume Claim by usi

Operations

Expanding PersistentVolumeClaim Storag

Expanding Persistent Volume Claim Stora

Additional resources

Using Volume Snapshots

Prerequisites

Example VolumeSnapshot custom resource (CR)

Creating Volume Snapshots by using th web console

Creating Volume Snapshots by using the C

Creating Persistent Volume Claims from V

Additional resource

Cloning a P

Prerequisites

ACP storage su

Create a cloned

Verify the clone

How To

Generic ephemeral volumes

- Example ephemeral volumes
- Key features
- When to Use Generic Ephemeral Volumes
- How Are They Different from emptyDir?

Using an emptyDir

- Example emptyDir
- Optional Medium Setting
- Key Characteristics
- Common Use Cases

Migrate the

- Scope
- Impact
- Prerequisites
- Procedure
- Rollback

Configuring Persistent Storage Using NFS

Configuring Disaster Recovery For Persistent Volume Claims

- Overview
- Terminology
- Prerequisites
- Deploy Alauda Storage Volume Replication
- Configuring a Scheduled Synchronization
- Configuring a One-Time Synchronization
- Enable Disaster Recovery for the Application PVC
- Planned Migration
- Failover
- Failback (post-disaster recovery)

Third-Party

1. Getting Start
2. Sample Conf
3. Update Exist
4. Compatibility
5. Frequently A

Troubleshooting

Recover From PVC Expansion F

Procedure

Additional Tips

Detach CSI Volumes After Non-Graceful Node

When to use this procedure

Prerequisites

Procedure

Verify the taint

Remove the taint after node recovery

Object Storage

Introduction

Limitations

Concepts

Overview

Core Resources

Installing

Prerequisites

Install COSI

on

Guides

How To

Introduction

Kubernetes offers a flexible and scalable storage mechanism for managing data persistence in containerized environments. By abstracting storage resources such as Volumes, PersistentVolumes, and PersistentVolumeClaims, Kubernetes decouples applications from underlying storage systems, enabling dynamic provisioning, automatic mounting, and persistent data across nodes.

Key features include support for multiple backend storage systems (e.g., local disks, NFS, cloud storage services), dynamic provisioning, access mode control (such as read/write permissions), and lifecycle management—meeting the storage needs of stateful applications. For enterprise-level workloads requiring high availability, data persistence, and multi-tenant isolation, Kubernetes storage is an essential foundational capability.

Kubernetes storage is designed for developers, operations engineers, and platform teams, helping them efficiently and securely manage data in containerized workloads.

Concepts

Core Concepts

Persistent Volume (PV)
Persistent Volume Claim (PVC)
Generic Ephemeral Volumes
emptyDir
hostPath
ConfigMap
Secret
StorageClass
Container Storage Interface (CSI)

Persistent Volume

Dynamic Persistent Volumes vs. Static Pe
Lifecycle of Persistent Volumes

Access Mod

Access Modes
Volume Modes
Storage Feature
Conclusion

Core Concepts

Kubernetes storage is centered on three key concepts: **PersistentVolume (PV)**, **PersistentVolumeClaim (PVC)**, and **StorageClass**. These define how storage is requested, allocated, and configured within a cluster. **CSI** (Container Storage Interface) drivers often handle the actual provisioning and attachment of storage.

TOC

Persistent Volume (PV)
Persistent Volume Claim (PVC)
Generic Ephemeral Volumes
emptyDir
hostPath
ConfigMap
Secret
StorageClass
Container Storage Interface (CSI)

Persistent Volume (PV)

A **PersistentVolume (PV)** is a piece of storage in the cluster that has been provisioned (either statically by an administrator or dynamically through a StorageClass). It represents the

underlying storage—such as a disk on a cloud provider or a network-attached filesystem—and is treated as a resource in the cluster, similar to a node.

Persistent Volume Claim (PVC)

A **PersistentVolumeClaim (PVC)** is a request for storage. Users define how much storage they need and the access mode (e.g., read-write). If an appropriate PV is available or can be dynamically provisioned (via a StorageClass), the PVC becomes “bound” to that PV. Once bound, Pods can reference the PVC to persist or share data.

Generic Ephemeral Volumes

Generic Ephemeral Volumes for Kubernetes is a feature introduced in Kubernetes that allows you to use CSI-driven `temporary` volumes during the Pod lifecycle, similar to the This is similar to `emptyDir`, but is more powerful and allows you to mount any type of CSI volume (with support for snapshots, scaling, etc.).

For usage details, see Generic ephemeral volumes.

emptyDir

1. emptyDir is a temporary storage volume of the empty directory type.
2. It is created when a Pod is dispatched to a node, and the storage is located on that node's local filesystem (node disk by default).
3. When a Pod is deleted, the data in emptyDir is also erased.

For usage details, see Using an emptyDir.

hostPath

In Kubernetes, a hostPath volume is a special type of volume that maps a file or directory from the host node's filesystem directly into a Pod's container.

- It allows a pod to access files or directories on the host node.
- Useful for:
  - Accessing host-level resources
  - Debugging
  - Using pre-existing data on the node

ConfigMap

A ConfigMap in Kubernetes is an API object used to store non-sensitive configuration data in the form of key-value pairs. It allows you to decouple configuration from application code, making your applications more portable and easier to manage.

Secret

In Kubernetes, a Secret is an API object that stores sensitive data such as:

- passwords
- OAuth tokens
- SSH keys
- TLS certificates
- database credentials

Secrets help protect this data by avoiding storing it directly in Pod specifications or container images.

StorageClass

A **StorageClass** describes *how* volumes should be dynamically provisioned. It maps to a specific provisioner (often a CSI driver) and can include parameters such as storage tiers, performance characteristics, or other backend configurations. By creating multiple StorageClasses, you can offer various types of storage to developers.

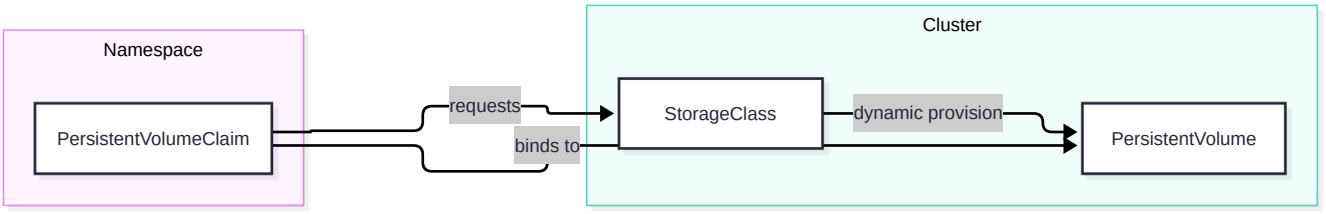


Diagram: Relationship between PVC, PV, and StorageClass.

Container Storage Interface (CSI)

The **Container Storage Interface (CSI)** is a standard API that Kubernetes uses to integrate with storage drivers. It allows third-party storage providers to build out-of-tree plugins, meaning you can install or update a storage driver without modifying Kubernetes itself.

A CSI **driver** typically has two components:

1. **Controller component**: Runs in the cluster (often as a Deployment) and handles high-level operations, such as **creating** or **deleting** volumes. For networked storage, it may also handle attaching and detaching volumes to nodes.
2. **Node component**: Runs on each node (often as a DaemonSet) and is responsible for **mounting** and **unmounting** the volume on that particular node. It communicates with the kubelet to ensure the volume is accessible to Pods.

When a user creates a PVC referring to a StorageClass that uses a CSI driver, the CSI driver observes that request and provisions storage accordingly (if dynamic provisioning is required). Once the storage is created, the driver notifies Kubernetes, which creates a corresponding PV and binds it to the PVC. Whenever a Pod uses that PVC, the node component of the driver handles the volume mount, making the storage available inside the container.

By leveraging **PV**, **PVC**, **StorageClass**, and **CSI**, Kubernetes enables a powerful, declarative approach to storage management. Administrators can define one or more StorageClasses to represent different storage backends or performance tiers, while developers simply request storage using PVCs—without worrying about the underlying infrastructure.

Persistent Volume

A PersistentVolume (PV) represents the mapping relationship with backend storage volumes in a Kubernetes cluster, functioning as a Kubernetes API resource. It is a cluster resource created and configured uniformly by administrators, responsible for abstracting the actual storage resources and forming the storage infrastructure of the cluster.

PersistentVolumes possess a lifecycle independent of Pods, enabling the persistent storage of Pod data.

Administrators may manually create static PersistentVolumes or generate dynamic PersistentVolumes based on storage classes. If developers need to obtain storage resources for applications, they can request them via PersistentVolumeClaims (PVC), which match and bind to suitable PersistentVolumes.

TOC

Dynamic Persistent Volumes vs. Static Persistent Volumes

Lifecycle of Persistent Volumes

Dynamic Persistent Volumes vs. Static Persistent Volumes

The platform supports management of two types of PersistentVolumes by administrators, namely dynamic and static Persistent Volumes.

- **Dynamic Persistent Volumes**: Implemented based on storage classes. Storage classes are created by administrators and define a Kubernetes resource that describes the category of storage resources. Once a developer creates a PersistentVolumeClaim associated with a storage class, the platform will dynamically create a suitable PersistentVolume according to the parameters configured in the PersistentVolumeClaim and storage class, binding it to the PersistentVolumeClaim for dynamic allocation of storage resources.
- **Static Persistent Volumes**: Persistent Volumes created manually by the administrator. Currently, it supports the creation of **HostPath** or **NFS shared storage** type static Persistent Volumes. When developers create a PersistentVolumeClaim without using a storage class, the platform will match and bind a suitable static PersistentVolume according to the parameters configured in the PersistentVolumeClaim.
  - **HostPath**: Uses a file directory on the node host (local storage is not supported) as backend storage, such as: `/etc/kubernetes`. It generally applies only to testing scenarios within a single worker node cluster.
  - **NFS Shared Storage**: Refers to the Network File System, a common type of backend storage for Persistent Volumes. Users and programs can access files on remote systems as if they were local files.

Lifecycle of Persistent Volumes

1. **Provisioning**: Administrators manually create static Persistent Volumes. After creation, the Persistent Volume enters an **Available** state; alternatively, the platform creates suitable Persistent Volumes dynamically based on PersistentVolumeClaims associated with storage classes.
2. **Binding**: Once a static Persistent Volume is matched and bound to a PersistentVolumeClaim, it enters a **Bound** state; dynamic Persistent Volumes are created dynamically based on requests matching PersistentVolumeClaims and also enter a **Bound** state once created successfully.
3. **Using**: Developers associate PersistentVolumeClaims with container instances of compute components, utilizing the backend storage resources mapped by the Persistent Volumes.

4. **Releasing**: After developers delete the PersistentVolumeClaim, the Persistent Volume is released.
5. **Reclaiming**: Once the Persistent Volume is released, reclamation operations are performed on it according to the reclamation policy parameters of the Persistent Volume or storage class.

Access Modes and Volume Modes

In Kubernetes, PersistentVolumeClaims (PVCs) and StorageClasses work together to manage how storage is provisioned and accessed by workloads. Two essential concepts in this domain are **Access Modes** and **Volume Modes**. This article explores these concepts and highlights how different storage systems support them.

TOC

- Access Modes in Kubernetes
  - Access Modes by Storage Class
- Volume Modes in Kubernetes
  - Volume Modes by Storage Class
- Storage Features: Snapshots and Expansion
- Conclusion

Access Modes in Kubernetes

Access Modes define how a volume can be mounted and used by pods. The key access modes are:

- **ReadWriteOnce (RWO)**: The volume can be mounted as read-write by a single node.
- **ReadOnlyMany (ROX)**: The volume can be mounted as read-only by multiple nodes.

- **ReadWriteMany (RWX)**: The volume can be mounted as read-write by multiple nodes.

Access Modes by Storage Class

Storage Class	RWO Supported	ROX Supported	RWX Supported
CephFS File Storage	Yes	No	Yes
CephRBD Block Storage	Yes	No	No
TopoLVM	Yes	No	No
NFS Shared Storage	Yes	No	Yes

As shown above, file-based storage systems like **CephFS** and **NFS** support multiple concurrent write or read operations, making them suitable for shared-access scenarios. On the other hand, block storage systems like **CephRBD** and **TopoLVM** provide exclusive access to a single node at a time.

Volume Modes in Kubernetes

Volume Modes define how the data is exposed to the pod:

- **Filesystem**: The volume is mounted into the pod as a filesystem.
- **Block**: The volume is presented as a raw block device.

Volume Modes by Storage Class

Storage Class	Type	Supported Volume Modes
CephFS File Storage	File Storage	Filesystem
CephRBD Block Storage	Block Storage	Filesystem, Block
TopoLVM	Block Storage	Filesystem, Block

Storage Class	Type	Supported Volume Modes
NFS Shared Storage	File Storage	Filesystem

Block storage systems like **CephRBD** and **TopoLVM** offer both filesystem and raw block access, providing flexibility for different application needs. File storage systems such as **CephFS** and **NFS**, in contrast, only support the filesystem mode.

Storage Features: Snapshots and Expansion

Kubernetes also supports advanced features like volume snapshots and dynamic expansion of PVCs, depending on the storage class used.

Storage Class	Volume Snapshot	Expansion
CephFS File Storage	Supported	Supported
CephRBD Block Storage	Supported	Supported
TopoLVM	Supported	Supported
NFS Shared Storage	Not Supported	Not Supported

Only dynamically provisioned PVCs using a StorageClass support volume snapshots. This feature is useful for backups and cloning environments.

Conclusion

When configuring storage in Kubernetes, understanding the **Access Modes** and **Volume Modes** of PVCs and their backing **StorageClasses** is critical for choosing the right solution for your workload. File storage solutions such as CephFS and NFS are ideal for shared access scenarios, while block storage like CephRBD and TopoLVM excel in high-performance, single-node deployments. Furthermore, support for features like snapshots and expansion can greatly enhance storage flexibility and data management strategies.

Guides

Creating CephFS File Storage T

Deploy Volume Plugin

Create Storage Class

Creating CephRBD Block Stora

Deploy Volume Plugin

Create Storage Class

Create Topo

Background Inf

Deploy Volume

Create Storage

Follow-up Actio

Deploy Volume Snapshot Component

Next steps

Creating an NFS Shared Storage Class

Creating a P

Prerequisites

Example Persis

/ by

/ by

Creating PVCs

Prerequisites

Example PersistentVolumeClaim:

Creating a Persistent Volume Claim by usi

Creating a Persistent Volume Claim by usi

Operations

Expanding PersistentVolumeClaim Storag

Expanding Persistent Volume Claim Stora

Additional resources

Using Volume Snapshots

Prerequisites

Example VolumeSnapshot custom resource (CR)

Creating Volume Snapshots by using th web console

Creating Volume Snapshots by using the C

Creating Persistent Volume Claims from V

Additional resource

Cloning a P

Prerequisites

ACP storage su

Create a cloned

Verify the clone

Creating CephFS File Storage Type Storage Class

CephFS file storage is a built-in Ceph file storage system that provides the platform with a Container Storage Interface (CSI)-based storage access method, offering a secure, reliable, and scalable shared file storage service suitable for scenarios such as file sharing and data backup. Before proceeding, you must first create a CephFS file storage class.

After binding the storage class in a Persistent Volume Claim (PVC), the platform will dynamically create persistent volumes on the nodes according to the persistent volume claim for business applications.

TOC

Deploy Volume Plugin

Create Storage Class

Deploy Volume Plugin

After clicking **Deploy** on the **Distributed Storage** page, choose either Deploying in Internal Mode or Deploying in External Mode.

Create Storage Class

1. Go to **Administrator**.
2. In the left navigation bar, click **Storage Management** > **Storage Classes**.
3. Click **Create Storage Class**.
   Note: The following content is provided as an example in form format; you may also choose to create it using YAML.
4. Select **CephFS File Storage** and click **Next**.
5. Configure the relevant parameters according to the following instructions.

Parameter	Description
Reclaim Policy	The reclaim policy for persistent volumes. - Delete: When the persistent volume claim is deleted, the bound persistent volume will also be deleted. - Retain: The bound persistent volume will remain, even if the persistent volume claim is deleted.
Access Modes	All access modes supported by the current storage. Only one of these modes can be selected when declaring persistent volumes later. - ReadWriteOnce (RWO): Can be mounted as read-write by a single node. - ReadWriteMany (RWX): Can be mounted as read-write by multiple nodes.
Allocate Project	Allocate the projects that can use this type of storage. If no project currently needs this type of storage, leave the allocation empty and update it later.

   Tip: The following parameters need to be set in the distributed storage and will be applied directly here.

   - Storage Cluster: The built-in Ceph storage cluster in the current cluster.
   - Storage Pool: The logical partition used for data storage in the storage cluster.

6. Click **Create**.

Creating CephRBD Block Storage Class

CephRBD block storage is a built-in Ceph block storage for the platform, providing a Container Storage Interface (CSI) based storage access method that can deliver high IOPS and low-latency storage services, suitable for scenarios such as databases and virtualization. Before using this, you need to create a CephRBD block storage class.

Once a Persistent Volume Claim (PVC) is bound to the storage class, the platform will dynamically create a Persistent Volume based on the Persistent Volume Claim for business applications to use.

TOC

Deploy Volume Plugin

Create Storage Class

Deploy Volume Plugin

After clicking **Deploy** on the **Distributed Storage** page, choose either Deploying in Internal Mode or Deploying in External Mode.

Create Storage Class

1. Go to **Administrator**.
2. In the left navigation bar, click **Storage Management** > **Storage Classes**.
3. Click **Create Storage Class**.
   Note: The following content is an example in form format, you can also choose YAML to complete the operation.
4. Select **CephRBD Block Storage**, and click **Next**.
5. Configure the parameters as required.

Parameter	Description
File System	Defaults to **EXT4**, which is a journaling file system for Linux, capable of providing extent file storage and processing large files. The filesystem capacity can reach 1 EiB, with supported file sizes up to 16 TiB.
Reclaim Policy	The reclaim policy for persistent volumes. - Delete: The bound persistent volume will be deleted along with the persistent volume claim. - Retain: The bound persistent volume will be retained even if the persistent volume claim is deleted.
Access Modes	Only supports ReadWriteOnce (RWO): it can be mounted by a single node in read-write mode.
Assign Project	Assign the projects that can use this type of storage. If no project currently needs this type of storage, leave the assignment empty and update it later.

   Tip: The following parameters need to be set in distributed storage and will be directly applied here.

   - Storage Cluster: The built-in Ceph storage cluster in the current cluster.
   - Storage Pool: The logical partition used for storing data within the storage cluster.

6. Click **Create**.

Create TopoLVM Local Storage Class

TopoLVM is an LVM-based local storage solution that provides simple, easy-to-maintain, and high-performance local storage services suitable for scenarios such as databases and middleware. Before using it, you need to create a TopoLVM storage class.

Once the Persistent Volume Claim (PVC) is bound to the storage class, the platform dynamically creates persistent volumes on the nodes based on the Persistent Volume Claim for business applications to use.

TOC

- Background Information
  - Advantages of Use
  - Use Cases
  - Constraints and Limitations
- Deploy Volume Plugin
- Create Storage Class
- Follow-up Actions

Background Information

Advantages of Use

- Compared to remote storage (e.g., **NFS shared storage**): TopoLVM-type storage is located locally on the node, offering better IOPS and throughput performance, as well as lower latency.
- Compared to hostPath (e.g., **local-path**): Although both are local storage on the node, TopoLVM allows for flexible scheduling of container groups to nodes with sufficient available resources, avoiding issues where container groups cannot start due to insufficient resources.
- TopoLVM supports automatic volume expansion by default. After modifying the required storage quota in the Persistent Volume Claim, the expansion can be completed automatically without restarting the container group.

Use Cases

- When only temporary storage is needed, such as for development and debugging.
- When there are high storage I/O requirements, such as real-time indexing.

Constraints and Limitations

Use local storage only for applications that provide application-layer data replication and backup. Local storage does not provide an independent data persistence guarantee when the underlying node fails.

For TopoLVM behavior, see the TopoLVM user manual ↗.

Deploy Volume Plugin

After clicking deploy, on the newly opened page Install LVM Storage.

Create Storage Class

1. Go to **Administrator**.
2. In the left navigation bar, click **Storage Management** > **Storage Classes**.
3. Click **Create Storage Class**.

4. Select **Block Storage**.
5. Select **TopoLVM**, then click **Next**.
6. Configure the storage class parameters as described below.
   Note: The following content is presented as a form example; you may also choose to create it using YAML.

Parameter	Description
Name	The name of the storage class, which must be unique within the current cluster.
Display Name	A name that can help you identify or filter it, such as a Chinese description of the storage class.
Device Class	The device class is a way to categorize storage devices in TopoLVM, with each device class corresponding to a group of storage devices with similar characteristics. If there are no special requirements, use the **Automatically Assigned** device class.
File System	• **XFS** is a high-performance journaling file system well-suited for handling parallel I/O workloads, supporting large file handling and smooth data transfer. • **EXT4** is a journaling file system under Linux that provides extent file storage and supports large file handling, with a maximum file system capacity of 1 EiB and a maximum file size of 16 TiB.
Reclamation Policy	The reclamation policy for persistent volumes. • Delete: The bound persistent volume will also be deleted along with the PVC. • Retain: The bound persistent volume will remain even if the PVC is deleted.
Access Mode	ReadWriteOnce (RWO): Can be mounted as read-write by a single node.

Parameter	Description
PVC Reconstruction	Supports PVC reconstruction across nodes. When enabled, the **Reconstruction Wait Time** must be configured. When the node hosting the PVC created using this storage class fails, the PVC will be automatically rebuilt on other nodes after the wait time to ensure business continuity. **Note**: • The rebuilt PVC does not contain the original data. • Ensure that the number of storage nodes is greater than the number of application instance replicas. Otherwise, PVC reconstruction can be affected.
Allocated Projects	Persistent volume claims of this type can only be created in specific projects. If no project is currently allocated, the project can also be **updated later**.

7. After confirming that the configuration information is correct, click the **Create** button.

Follow-up Actions

Once everything is ready, you can notify the developers to use the TopoLVM features. For example, create a Persistent Volume Claim and bind it to the TopoLVM storage class in the **Storage** > **Persistent Volume Claims** page of the container platform.

Creating an NFS Shared Storage Class

Based on the community NFS CSI (Container Storage Interface) storage driver, it provides the capability to access multiple NFS storage systems or accounts.

Unlike the traditional client-server model of NFS access, NFS shared storage utilizes the community NFS CSI (Container Storage Interface) storage plugin, which is more aligned with Kubernetes design principles and allows client access to multiple servers.

TOC

Prerequisites

Deploy Alauda Container Platform NFS CSI

Deploying via Web Console

Deploying via YAML

Creating an NFS Shared Storage Class

Prerequisites

- An NFS server must be configured, and its access methods must be obtained. Currently, the platform supports three NFS protocol versions: `v3`, `v4.0`, and `v4.1`. You can execute `nfsstat -s` on the server side to check the version information.

Deploy Alauda Container Platform NFS CSI

Deploying via Web Console

1. Enter **Administrator**.
2. In the left navigation bar, click **Storage** > **StorageClasses**.
3. Click **Create StorageClass**.
4. On the right side of **NFS CSI**, click Deploy to navigate to the **Plugins** page.
5. On the right side of the **Alauda Container Platform NFS CSI** plugin, click ⋮ > **Install**.
6. Wait for the deployment status to indicate **Deployment Successful** before completing the deployment.

Deploying via YAML

For YAML installation, see Installing via YAML.

Alauda Container Platform NFS CSI is a **Non-config plugin**, and the module name is `nfs`.

Creating an NFS Shared Storage Class

1. Click **Create Storage Class**.
   Note: The following content is presented in a form, but you may also choose to complete the operation using YAML.
2. Select **NFS CSI** and click **Next**.
3. Configure the following parameters.

Parameter	Description
Name	The name of the storage class. It must be unique within the current cluster.

Parameter	Description
Service Address	The access address of the NFS server. For example: `192.168.2.11`.
Path	The mount path of the NFS file system on the server node. For example: `/nfs/data`.
NFS Protocol Version	Currently supports three versions: `v3`, `v4.0`, and `v4.1`.
Reclaim Policy	The reclaim policy for the persistent volume. - Delete: When the persistent volume claim is deleted, the bound persistent volume will also be deleted. - Retain: Even if the persistent volume claim is deleted, the bound persistent volume will still be retained.
Access Modes	All access modes supported by the current storage. During the subsequent declaration of persistent volumes, only one of these modes can be selected for mounting persistent volumes. - ReadWriteOnce (RWO): Can be mounted as read-write by a single node. - ReadWriteMany (RWX): Can be mounted as read-write by multiple nodes. - ReadOnlyMany (ROX): Can be mounted as read-only by multiple nodes.
Allocated Projects	Allocate the projects that can use this type of storage. If no projects currently need this type of storage, leave the allocation empty and update it later.
subDir	Each PersistentVolumeClaim (PVC) created using the NFS Shared Storage Class corresponds to a subdirectory within the NFS share. By default, subdirectories are named using the pattern `${pv.metadata.name}` (i.e., the PersistentVolume name). If the default generated name does not meet your requirements, you can customize the subdirectory naming rules.

NOTE

The `subDir` field supports only the following three variables, which the NFS CSI Driver automatically resolves:

- `${pvc.metadata.namespace}`: PVC Namespace.
- `${pvc.metadata.name}`: PVC Name.
- `${pv.metadata.name}`: PV Name.

The `subDir` naming rule **MUST** guarantee unique subdirectory names. Otherwise, multiple PVCs may share the same subdirectory, causing data conflicts.

Recommended Configurations:

- `${pvc.metadata.namespace}_${pvc.metadata.name}_${pv.metadata.name}`
- `<cluster-identifier>_${pvc.metadata.namespace}_${pvc.metadata.name}_${pv.metadata.name}`

Designed for multiple Kubernetes clusters sharing the same NFS Server, this configuration ensures clear cluster differentiation by incorporating a cluster-specific identifier (e.g., the cluster name) into the subdirectory naming rules.

Not Recommended Configurations:

- `${pvc.metadata.namespace}-${pvc.metadata.name}-${pv.metadata.name}` Avoid - as separators, may lead to ambiguous subdirectory names. For example: If two PVCs are named `ns-1/test` and `ns/1-test`, both could generate the same subdirectory `ns-1-test`.
- `${pvc.metadata.namespace}/${pvc.metadata.name}/${pv.metadata.name}` Do NOT configure subDir to create nested directories. The NFS CSI Driver only deletes the last-level directory `${pv.metadata.name}` when a PVC is removed, leaving orphaned parent directories on the NFS Server.

4. Once you have confirmed that the configuration information is correct, click **Create**.

Deploy Volume Snapshot Component

A volume snapshot is a point-in-time copy of a persistent volume. If the cluster uses persistent volumes that support snapshot functionality, the volume snapshot component must be available before users can create and restore volume snapshots.

In ACP 4.4 and later, you do not need to install the **Alauda Container Platform Snapshot Management** cluster plugin for new clusters. ACP prepares the cluster-level snapshot component automatically when ACP Storage is available.

If the cluster already has an external Snapshot Controller or existing snapshot CRDs, ACP keeps the existing snapshot capability and does not replace it automatically.

If the cluster was installed with the legacy **Alauda Container Platform Snapshot Management** cluster plugin, migrate it by following Migrate the Volume Snapshot Component from the Legacy Plugin.

Currently, the platform only supports creating volume snapshots for PVCs that are **dynamically created** using storage classes. You can create new PVC bindings based on these snapshots.

Tip: The access modes supported when creating PVCs from snapshots differ from those supported when creating PVCs using storage classes, which are indicated in **bold** in the table below.

Storage Class Used to Create Volume Snapshots	Single Node Read-Write (RWO)	Multi-Node Read-Only (ROX)	Multi-Node Read-Write (RWX)
TopoLVM	Supported	Not Supported	Not Supported

Storage Class Used to Create Volume Snapshots	Single Node Read-Write (RWO)	Multi-Node Read-Only (ROX)	Multi-Node Read-Write (RWX)
CephRBD Block Storage	Supported	Not Supported	Not Supported
CephFS File Storage	Supported	**Supported**	Supported

TOC

Next steps

Next steps

After the snapshot component is available, users can create volume snapshots. For details, see Using Volume Snapshots.

If snapshot operations do not work after removing the legacy plugin, follow Migrate the Volume Snapshot Component from the Legacy Plugin.

Creating a PV

Manually create a static persistent volume of type **HostPath** or **NFS Shared Storage**.

- **HostPath**: Mounts the file directory from the host where the container resides to a specified path in the container (corresponding to Kubernetes' HostPath), allowing the container to use the host's file system for persistent storage. If the host becomes inaccessible, the HostPath may not be accessible.
- **NFS Shared Storage**: NFS Shared Storage uses the community NFS CSI (Container Storage Interface) storage plugin, which aligns more closely with Kubernetes design principles, providing client access capabilities for multiple services. Ensure that the current cluster has deployed the **NFS storage plugin** before use.

TOC

Prerequisites
Example PersistentVolume
Creating PV by using the web console
Storage Information
Creating PV by using the CLI
Access Modes
Reclaim Policies
Related Operations
Additional resource

Prerequisites

- Confirm the size of the persistent volume to be created and ensure that the backend storage system currently has the capacity to provide the corresponding storage.
- Obtain the backend storage access address, the file path to be mounted, credential access (if required), and other relevant information.

Example PersistentVolume

```yaml
# example-pv.yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: example-pv
spec:
  capacity:
    storage: 5Gi (1)
  accessModes:
    - ReadWriteOnce (2)
  persistentVolumeReclaimPolicy: Retain (3)
  storageClassName: manual (4)
  hostPath: (5)
    path: "/mnt/data"
```

(1) Amount of storage.

(2) How the volume can be mounted.

(3) What happens after PVC is deleted (Retain, Delete, Recycle).

(4) Name of the StorageClass (for dynamic binding).

(5) Storage backend type.

Creating PV by using the web console

1. Navigate to **Administrator**.
2. In the left navigation bar, click on **Storage Management** > **Persistent Volumes (PV)**.

3. Click on **Create Persistent Volume**.
4. Configure the parameters below, and then click **Create**.

Storage Information

Type	Parameter	Description
HostPath	Path	The path to the directory of files on the node backing the storage volume. For example: `/etc/kubernetes`.
NFS Shared Storage	Server Address	The access address of the NFS server.
	Path	The mount path of the NFS file system on the server node, such as `/nfs/data`.
	NFS Protocol Version	The currently supported NFS protocol versions on the platform are `v3`, `v4.0`, and `v4.1`. You can execute `nfsstat -s` on the server side to view version information.

Creating PV by using the CLI

```
kubectl apply -f example-pv.yaml
```

Access Modes

Access modes of the persistent volume influenced by the relevant parameters set by the backend storage.

Access Mode	Meaning
ReadWriteOnce (RWO)	Can be mounted as read-write by a single node.
ReadWriteMany (RWX)	Can be mounted as read-write by multiple nodes.

Access Mode	Meaning
ReadOnlyMany (ROX)	Can be mounted as read-only by multiple nodes.

Reclaim Policies

Reclaim Policy	Meaning
Delete	Deletes the persistent volume claim at the same time deletes the bound persistent volume, as well as the backend storage volume resource. **Note**: The reclaim policy for PV of type NFS Shared Storage does not support **Delete**.
Retain	Even when the persistent volume claim is deleted, the bound persistent volume and storage data will still be retained. Manual handling of the storage data and deletion of the persistent volume will be required thereafter.

Related Operations

You can click the ⋮ on the right of the list page or click the **Operations** in the upper right corner of the details page to update or delete the persistent volume as needed.

Deleting a persistent volume is applicable in the following two scenarios:

- Deleting an unbound persistent volume: Has not been written to and is no longer required for writing, thus freeing up corresponding storage space upon deletion.
- Deleting a **Retained** persistent volume: The persistent volume claim has been deleted, but due to the retain reclaim policy, it has not been deleted simultaneously. If the data in the persistent volume has been backed up to other storage or is no longer needed, deleting it can also free up corresponding storage space.

Additional resource

- Creating PVCs

Creating PVCs

Create a PersistentVolumeClaim (PVC) and set the parameters for the requested PersistentVolume (PV) as needed.

You can create a PersistentVolumeClaim either through a visual UI form or by using a custom YAML orchestration file.

TOC

Prerequisites
Example PersistentVolumeClaim:
Creating a Persistent Volume Claim by using the web console
Creating a Persistent Volume Claim by using the CLI
Operations
Expanding PersistentVolumeClaim Storage Capacity by using the web console
Expanding Persistent Volume Claim Storage Capacity by using the CLI
Additional resources

Prerequisites

Ensure that there is enough remaining **storage** quota in the namespace to satisfy the required storage size for this creation operation.

Example PersistentVolumeClaim:

```yaml
# example-pvc.yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: example-pvc
  namespace: k-1
  annotations: {}
  labels: {}
spec:
  storageClassName: cephfs
  accessModes:
    - ReadWriteOnce
  volumeMode: Filesystem
  resources:
    requests:
      storage: 4Gi
```

Creating a Persistent Volume Claim by using the web console

1. Go to **Container Platform**.
2. Click on **Storage > PersistentVolumeClaims (PVC)** in the left sidebar.
3. Click on **Create PVC**.
4. Configure the parameters as required.
   Note: The following content is provided as an example using the form method; you can also switch to YAML mode to complete the operation.

Parameter	Description
Name	The name of the PersistentVolumeClaim, which must be unique within the current namespace.

Parameter	Description
Creation Method	- Dynamic Creation: Dynamically generates a PersistentVolume based on the storage class and binds it. - Static Binding: Matches and binds based on configured parameters and existing PersistentVolumes.
Storage Class	After selecting the dynamic creation method, the platform will dynamically create the PersistentVolume as per the description in the specified storage class.
Access Mode	- ReadWriteOnce (RWO): Can be mounted by a single node in read-write mode. - ReadWriteMany (RWX): Can be mounted by multiple nodes in read-write mode. - ReadOnlyMany (ROX): Can be mounted by multiple nodes in read-only mode. **Tip**: It's recommended to consider the number of workload instances that are planned to bind to the current PersistentVolumeClaim and the type of deployment controller. For example, when creating a multi-instance deployment (Deployment), since all instances use the same PersistentVolumeClaim, it is not advisable to choose the RWO access mode, which can only attach to a single node.
Capacity	The size of the requested PersistentVolume.
Volume Mode	- Filesystem: Binds the PersistentVolume as a file directory mounted into the Pod. This mode is available for any type of workload. - Block Device: Binds the PersistentVolume as a raw block device mounted into the Pod. This mode is available only for virtual machines.
More	- Labels - Annotations - Selector: After selecting the static binding method, you can use a selector to target PersistentVolumes that are labeled with specific tags.

Parameter	Description
	PersistentVolume labels can be used to denote special attributes of the storage, such as disk type or geographic location.

5. Click on **Create**. Wait for the PersistentVolumeClaim to change to `Bound` status, indicating that the PersistentVolume has been successfully matched.

Creating a Persistent Volume Claim by using the CLI

```
kubectl apply -f example-pvc.yaml
```

Operations

- **Bind PersistentVolumeClaim**: When creating applications or workloads that require persistent data storage, bind the PersistentVolumeClaim to request a compliant PersistentVolume.
- **Create a PersistentVolumeClaim using Volume Snapshots**: Use volume snapshots to back up application data and restore it when needed. For details, see Using Volume Snapshots.
- **Delete PersistentVolumeClaim**: Click the **Actions** button in the upper-right corner of the details page to delete the PersistentVolumeClaim. Before deleting it, ensure that the PersistentVolumeClaim is not bound to applications or workloads and does not contain volume snapshots. After deletion, the platform processes the PersistentVolume according to the reclamation policy, which might clear data in the PersistentVolume and free storage resources. Confirm data security requirements before deleting a PersistentVolumeClaim.

Expanding PersistentVolumeClaim Storage Capacity by using the web console

1. In the left navigation bar, click Storage > Persistent Volume Claims (PVC).
2. Find the persistent volume claim and click ⋮ > Expand.
3. Fill in the new capacity.
4. Click **Expand**. The expansion process might take some time.

Expanding Persistent Volume Claim Storage Capacity by using the CLI

```
kubectl patch pvc example-pvc -n k-1 --type='merge' -p '{
  "spec": {
    "resources": {
      "requests": {
        "storage": "6Gi"
      }
    }
  }
}'
```

INFO

When PVC expansion fails in Kubernetes, administrators can manually recover the Persistent Volume Claim (PVC) state and cancel the expansion request. See Recover From PVC Expansion Failure

Additional resources

- How to Annotate Third-Party Storage Capabilities

Using Volume Snapshots

A volume snapshot is a point-in-time copy of a persistent volume claim (PVC) that can be used to configure new persistent volume claims (pre-filling with snapshot data) or to roll back existing persistent volume claims to a previous state, achieving the effect of backing up application data and restoring it as needed, thereby ensuring the reliability of application data.

TOC

- Prerequisites
- Example VolumeSnapshot custom resource (CR)
- Creating Volume Snapshots by using th web console
  - Creating a Volume Snapshot Based on a Specified Persistent Volume Claim (PVC)
  - Creating Volume Snapshots in a Custom Way
- Creating Volume Snapshots by using the CLI
- Creating Persistent Volume Claims from Volume Snapshots
  - Method One
  - Method Two
- Additional resource

Prerequisites

- The administrator has deployed the volume snapshot component **Snapshot Controller** for the current cluster and enabled snapshot-related features in the storage cluster.
- The persistent volume claim must be created dynamically and its status must be **Bound**.
- The storage class bound to the persistent volume claim must support snapshot functionality, such as **CephRBD Built-in Storage**, **CephFS Built-in Storage**, or **TopoLVM**.

Example VolumeSnapshot custom resource (CR)

This creates a snapshot of the example-pvc `PVC` using a CSI snapshot class.

```
# example-snapshot.yaml
apiVersion: snapshot.storage.k8s.io/v1
kind: VolumeSnapshot
metadata:
  name: example-pvc-20250527-111124
  namespace: k-1
  labels:
    snapshot.cpaas.io/sourcepvc: example-pvc
  annotations:
    cpaas.io/description: demo
spec:
  volumeSnapshotClassName: csi-cephfs-snapshotclass
  source:
    persistentVolumeClaimName: example-pvc
```

Creating Volume Snapshots by using th web console

Creating a Volume Snapshot Based on a Specified Persistent Volume Claim (PVC)

Method One

1. Enter the **Container Platform**.

2. In the left navigation bar, click **Storage > Persistent Volume Claims (PVC)**.
3. Click the ⋮ next to the corresponding persistent volume claim in the list and select **Create Volume Snapshot**.
4. Fill in the snapshot description. This description can help you record the current state of the persistent volume, such as *Before Application Upgrade*.
5. Click **Create**. Snapshot creation time depends on network conditions and data volume. When the snapshot changes to `Available` status, it indicates that the creation was successful.

Method Two

1. Enter the **Container Platform**.
2. In the left navigation bar, click **Storage > Persistent Volume Claims (PVC)**.
3. Click on the name of the persistent volume claim in the list.
4. Switch to the **Volume Snapshots** tab.
5. Click **Create Volume Snapshot**, and configure the relevant parameters as needed.
6. Click **Create**. Snapshot creation time depends on network conditions and data volume. When the snapshot changes to `Available` status, it indicates that the creation was successful.

Creating Volume Snapshots in a Custom Way

1. Enter the **Container Platform**.
2. In the left navigation bar, click **Storage > Volume Snapshots**.
3. Click **Create Volume Snapshot**, and configure the relevant parameters as needed.
4. Click **Create**. Snapshot creation time depends on network conditions and data volume. When the snapshot changes to `Available` status, it indicates that the creation was successful.

Creating Volume Snapshots by using the CLI

```
kubectl apply -f example-snapshot.yaml
```

Creating Persistent Volume Claims from Volume Snapshots

Currently, the platform only supports creating volume snapshots using PVCs created from storage classes with **Dynamic Provisioning**. You can create new PVCs based on that snapshot and bind them.

Note: The access modes supported when creating a PVC from a snapshot differ from those supported when creating a PVC from a storage class, as highlighted in **bold** in the table.

Storage Class Used for Creating Volume Snapshots	Single Node Read-Write (RWO)	Multi-Node Read-Only (ROX)	Multi-Node Read-Write (RWX)
TopoLVM	Supported	Not Supported	Not Supported
CephRBD Block Storage	Supported	Not Supported	Not Supported
CephFS File Storage	Supported	**Supported**	Supported

Method One

1. Enter the **Container Platform**.
2. In the left navigation bar, click **Storage > Persistent Volume Claims (PVC)**.
3. Click on the name of the persistent volume claim in the list.
4. Switch to the **Volume Snapshots** tab.
5. Click the ⋮ next to the corresponding volume snapshot in the list and select **Create Persistent Volume Claim**, configuring the relevant parameters.
6. Click **Create**.

Method Two

1. Enter the **Container Platform**.

2. In the left navigation bar, click **Storage > Volume Snapshots**.
3. Click the ⋮ next to the corresponding volume snapshot in the list and select **Create Persistent Volume Claim**, configuring the relevant parameters.
4. Click **Create**.

Additional resource

- Creating PVCs

Cloning a PVC

PVC cloning creates a new PersistentVolumeClaim (PVC) with a copy of the data in an existing PVC. Use a clone when an application needs an independent writable copy of data, for example, to prepare a test environment from existing application data.

A clone is created directly from a source PVC. In contrast, a volume snapshot captures the PVC at a point in time and can be retained or used later to create a PVC. Choose a clone for a direct copy and a volume snapshot when you need a recoverable point-in-time copy.

> **NOTE**
>
> ACP does not currently provide a UI for cloning PVCs. Create the cloned PVC by applying a YAML manifest with `kubectl`.

TOC

- Prerequisites
- ACP storage support
- Create a cloned PVC
- Verify the clone

Prerequisites

- PVC cloning is available only for dynamically provisioned PVCs whose CSI driver supports cloning. Static provisioning does not support PVC cloning.
- The source PVC is in the `Bound` phase and is not in use.
- The source and target PVCs are in the same namespace.
- The target PVC requests storage capacity that is at least the source PVC capacity.
- The source and target PVCs use the same volume mode. The target access mode must be supported by its storage class and CSI driver.

ACP storage support

The target PVC can use a different storage class only when the CSI driver and storage backend support cloning from the source volume to that storage class. Kubernetes does not validate this backend compatibility.

Storage type	PVC clone support	Storage class requirements
CephRBD Block Storage	Supported	The source and target must use compatible RBD storage classes in the same Ceph cluster. A different RBD storage class, including one that uses a different pool, can be used when Ceph CSI supports the selected backend configuration.
CephFS File Storage	Supported	The source and target must use compatible storage classes for the same Ceph cluster and CephFS file system.
TopoLVM	Supported for thin-provisioned volumes	Use the same storage class for the source and target. A clone created with a different storage class can fail to schedule when its device class differs from the source.
NFS Shared Storage	Not supported	NFS Shared Storage does not support PVC cloning.

For information about creating a source PVC, see Creating PVCs.

Create a cloned PVC

Create a manifest named `clone-pvc.yaml`. Replace the namespace, storage class, access mode, volume mode, capacity, and PVC names with values supported by your storage driver. See ACP storage support before choosing a target storage class.

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: cloned-pvc
  namespace: application-space
spec:
  storageClassName: clone-capable-csi
  accessModes:
    - ReadWriteOnce
  volumeMode: Filesystem
  resources:
    requests:
      storage: 10Gi
  dataSource:
    apiGroup: ""
    kind: PersistentVolumeClaim
    name: source-pvc
```

Apply the manifest:

```
kubectl apply -f clone-pvc.yaml
```

Verify the clone

Check the status of the target PVC:

```
kubectl get pvc cloned-pvc -n application-space
```

When the `STATUS` column shows `Bound`, the storage driver has provisioned and bound the cloned volume. You can then mount `cloned-pvc` in a workload and verify that the expected

source data is available.

Inspect the PVC events for provisioning or cloning errors:

```
kubectl describe pvc cloned-pvc -n application-space
```

If the PVC remains in `Pending` or creation fails, review the events shown by this command. Confirm that the selected storage class supports PVC cloning and that the source PVC, namespace, requested capacity, volume mode, and access mode meet the prerequisites.

How To

Generic ephemeral volumes

- Example ephemeral volumes
- Key features
- When to Use Generic Ephemeral Volumes
- How Are They Different from emptyDir?

Using an emptyDir

- Example emptyDir
- Optional Medium Setting
- Key Characteristics
- Common Use Cases

Migrate the

- Scope
- Impact
- Prerequisites
- Procedure
- Rollback

Configuring Persistent Storage Using NFS

Configuring Disaster Recovery For Persistent Volume Claims

- Overview
- Terminology
- Prerequisites
- Deploy Alauda Storage Volume Replication
- Configuring a Scheduled Synchronization
- Configuring a One-Time Synchronization
- Enable Disaster Recovery for the Application PVC
- Planned Migration
- Failover
- Failback (post-disaster recovery)

Third-Party

1. Getting Start
2. Sample Conf
3. Update Exist
4. Compatibility
5. Frequently A

Generic ephemeral volumes

Generic Ephemeral Volumes in Kubernetes are a feature that allows you to provision ephemeral (temporary), per-pod volumes using existing StorageClasses and CSI drivers, without needing to predefine PersistentVolumeClaims (PVCs).

They combine the flexibility of dynamic provisioning with the simplicity of pod-level volume declaration.

- They are temporary volumes that are automatically:
  - created when the Pod starts
  - deleted when the Pod terminates
- Use the same underlying mechanisms as PersistentVolumeClaim
- Require a CSI (Container Storage Interface) driver that supports dynamic provisioning

TOC

Example ephemeral volumes

Key features

When to Use Generic Ephemeral Volumes

How Are They Different from emptyDir?

Example ephemeral volumes

This automatically creates a temporary PVC for the Pod using the specified `StorageClass`.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: ephemeral-demo
spec:
  containers:
    - name: app
      image: busybox
      command: ["sh", "-c", "echo hello > /data/hello.txt && sleep 3600"]
      volumeMounts:
        - mountPath: /data
          name: ephemeral-volume
  volumes:
    - name: ephemeral-volume
      ephemeral: (1)
        volumeClaimTemplate:
          metadata:
            labels:
              type: temporary
          spec:
            accessModes: [ "ReadWriteOnce" ]
            resources:
              requests:
                storage: 1Gi
            storageClassName: standard
```

(1) `Pod` will create a `PVC` by using this template.

Key features

Feature	Description
Ephemeral	Volume is deleted when the Pod is deleted

Feature	Description
Dynamic provisioning	Backed by any CSI driver with dynamic provisioning
No separate PVC	VolumeClaim is embedded directly in the Pod spec
CSI-powered	Works with any compatible CSI driver (EBS, RBD, Longhorn, etc.)

When to Use Generic Ephemeral Volumes

- When you need temporary storage with features like:
  - Resizable volumes
  - Snapshots
  - Encryption
  - Non-node-local storage (e.g., cloud block storage)
- Ideal for:
  - Caching intermediate data
  - Temporary working directories
  - Pipelines, AI/ML workflows

How Are They Different from emptyDir?

Feature	`emptyDir`	Generic Ephemeral Volume
Backing storage	Node's local disk or memory	Any CSI-supported backend
Storage features	Basic	Supports snapshots, encryption, etc.
Use case	Simple temporary storage	Advanced ephemeral storage needs
Reschedulable	No (tied to node)	Yes (if CSI volume is attachable)

Using an emptyDir

In Kubernetes, an emptyDir is a simple ephemeral volume type that provides temporary storage to a pod during its lifetime. It is created when a pod is assigned to a node, and deleted when the pod is removed from that node.

TOC

Example emptyDir
Optional Medium Setting
Key Characteristics
Common Use Cases

Example emptyDir

This Pod creates a temporary volume mounted at /data, which is shared with the container.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: emptydir-demo
spec:
  containers:
    - name: app
      image: busybox
      command: ["sh", "-c", "echo hello > /data/hello.txt && sleep 3600"]
      volumeMounts:
        - mountPath: /data
          name: cache-volume
  volumes:
    - name: cache-volume
      emptyDir: {}
```

Optional Medium Setting

You can choose where the data is stored:

```yaml
emptyDir:
  medium: "Memory"
```

Medium	Description
(default)	Uses node's disk, SSD or network storage, depending on your environment
`Memory`	Uses RAM (`tmpfs`) for faster access (but volatile)

Key Characteristics

Feature	Description
Starts empty	No data when created

Feature	Description
Shared across containers	Same volume can be used by multiple containers in the pod
Deleted with pod	Volume is destroyed when the pod is removed
Node-local	Volume is stored on the node's local disk or memory
Fast	Ideal for performance-sensitive scratch space

Common Use Cases

- Caching intermediate build artifacts
- Buffering logs
- Temporary work directories
- Sharing data between containers in the same pod (like sidecars)

Migrate the Volume Snapshot Component from the Legacy Plugin

In ACP 4.4, new clusters no longer need to install the **Alauda Container Platform Snapshot Management** cluster plugin separately. If an existing workload cluster already installed the legacy plugin, use this runbook to migrate the cluster to the ACP 4.4 snapshot component without deleting existing snapshot resources.

TOC

- Scope
- Impact
- Prerequisites
- Procedure
  - Record current snapshot resources
  - Confirm the legacy plugin is installed
  - Uninstall the legacy plugin
  - Verify the legacy controller is gone
  - Enable ACP snapshot management
  - Wait for the snapshot component to become ready
  - Verify snapshot CRDs are present
  - Verify existing snapshot resources are retained
- Rollback
- Troubleshooting

Snapshot status reports `Skipped`

Snapshot operations remain pending

Snapshot management was enabled while another controller is still running

Related Documentation

Scope

This procedure applies only when all of the following are true:

- The workload cluster uses the legacy **Alauda Container Platform Snapshot Management** cluster plugin.
- You are moving the workload cluster to ACP 4.4 or later.
- You want ACP to manage the cluster-level snapshot component after the legacy plugin is removed.

Do not use this procedure for clusters where another platform or an administrator-managed component still provides the Snapshot Controller. Only one Snapshot Controller should manage a cluster.

Impact

There is a short expected gap between uninstalling the legacy plugin and the new snapshot component becoming ready. During this gap, the cluster has no running Snapshot Controller. New snapshot create or delete requests can remain pending until the new component is ready.

Existing `VolumeSnapshot`, `VolumeSnapshotContent`, `VolumeSnapshotClass`, `VolumeGroupSnapshot`, `VolumeGroupSnapshotContent`, and `VolumeGroupSnapshotClass` resources are retained.

WARNING

Do not delete snapshot CRDs during this migration. Deleting snapshot CRDs can delete all cluster-wide snapshot objects and cause data loss.

WARNING

Do not use cleanup actions that delete snapshot CRDs or snapshot custom resources.

Prerequisites

- You have administrator access to the global cluster and the target workload cluster.
- The target workload cluster is on ACP 4.4 or later, or is being upgraded to ACP 4.4.
- The ACP Storage component is installed in the target workload cluster.
- The `jq` command-line tool is available on the workstation where you run the `kubectl patch` commands.
- You have a maintenance window for snapshot operations.

Procedure

1 Record current snapshot resources

Run these commands in the target workload cluster before making changes:

```
kubectl get crd | grep -E 'snapshot.storage.k8s.io|groupsnapshot.storage.k8s.io'
kubectl get volumesnapshots.snapshot.storage.k8s.io -A --ignore-not-found
kubectl get volumesnapshotcontents.snapshot.storage.k8s.io --ignore-not-found
kubectl get volumesnapshotclasses.snapshot.storage.k8s.io --ignore-not-found
kubectl get volumegroupsnapshots.groupsnapshot.storage.k8s.io -A --ignore-not-found
kubectl get volumegroupsnapshotcontents.groupsnapshot.storage.k8s.io --ignore-not-found
kubectl get volumegroupsnapshotclasses.groupsnapshot.storage.k8s.io --ignore-not-found
```

Keep the output for post-migration comparison.

2 Confirm the legacy plugin is installed

Run this command in the global cluster. Replace `<workload-cluster-name>` with the target workload cluster name:

```
kubectl get moduleinfo \
  -l cpaas.io/module-name=snapshot,cpaas.io/cluster-name=<workload-cluster-name>
```

The legacy Snapshot Management plugin is installed when this command returns a resource.

3 Uninstall the legacy plugin

Uninstall the legacy plugin from the global cluster. You can use the web console or YAML.

To use the web console:

1. Go to **Administrator** > **Marketplace** > **Cluster Plugins**.
2. Select the target workload cluster.

3. Find **Alauda Container Platform Snapshot Management**.
4. Uninstall the plugin.

To use YAML, delete the plugin installation resource from the global cluster:

```
kubectl delete moduleinfo \
  -l cpaas.io/module-name=snapshot,cpaas.io/cluster-name=<workload-cluster-name>
```

The uninstall removes the legacy Snapshot Controller workload, but it must not delete snapshot CRDs or existing snapshot resources.

4 Verify the legacy controller is gone

Run this command in the target workload cluster:

```
kubectl get deployment -A | grep -E 'snapshot-controller|snapshot' || true
```

Continue only when there is no legacy Snapshot Controller deployment.

If another Snapshot Controller is still running, stop here. Enabling ACP to take over snapshot management while another controller is active can create two Snapshot Controllers in the same cluster.

5 Enable ACP snapshot management

Run this command in the target workload cluster:

```
kubectl patch storagefoundation default --type=json -p "$(
  kubectl get storagefoundation default -o json | jq -c '
    (.spec.operators // []) as $operators |
    (
      if any($operators[]?; .name == "snapshot") then
        $operators | map(
          if .name == "snapshot" then
            .enabled = true | .upgradePolicy.mode = "Automatic"
          else
            .
          end
        )
      else
        $operators + [{
          "name": "snapshot",
          "enabled": true,
          "upgradePolicy": {"mode": "Automatic"}
        }]
      end
    ) as $updated |
    [{
      "op": (if .spec.operators == null then "add" else "replace" en
d),
      "path": "/spec/operators",
      "value": $updated
    }]
  '
)"
```

This patch adds or updates the `snapshot` entry and preserves all existing entries. Use this setting only after the legacy controller has been removed.

6 Wait for the snapshot component to become ready

Run this command in the target workload cluster:

```
kubectl get storagefoundation default \
  -o jsonpath='{range .status.operators[?(@.name=="snapshot")]}phase=
{.phase}{"\n"}{range .conditions[*]}{.type}={.status} reason={.reaso
n} message={.message}{"\n"}{end}{end}'
```

The migration succeeds when the `snapshot` entry reports `phase=Ready`.

7 Verify snapshot CRDs are present

Run this command in the target workload cluster:

```
for crd in \
  volumesnapshots.snapshot.storage.k8s.io \
  volumesnapshotcontents.snapshot.storage.k8s.io \
  volumesnapshotclasses.snapshot.storage.k8s.io \
  volumegroupsnapshots.groupsnapshot.storage.k8s.io \
  volumegroupsnapshotcontents.groupsnapshot.storage.k8s.io \
  volumegroupsnapshotclasses.groupsnapshot.storage.k8s.io; do
  kubectl get crd "$crd"
done
```

All six CRDs should exist after the snapshot component is ready.

8 Verify existing snapshot resources are retained

Compare the following output with the output recorded before the migration:

```
kubectl get volumesnapshots.snapshot.storage.k8s.io -A --ignore-not-found
kubectl get volumesnapshotcontents.snapshot.storage.k8s.io --ignore-not-found
kubectl get volumesnapshotclasses.snapshot.storage.k8s.io --ignore-not-found
kubectl get volumegroupsnapshots.groupsnapshot.storage.k8s.io -A --ignore-not-found
kubectl get volumegroupsnapshotcontents.groupsnapshot.storage.k8s.io --ignore-not-found
kubectl get volumegroupsnapshotclasses.groupsnapshot.storage.k8s.io --ignore-not-found
```

Existing snapshot resources should still be present.

Rollback

If the snapshot component does not become `Ready`, inspect the snapshot status:

```
kubectl get storagefoundation default \
  -o jsonpath='{range .status.operators[?(@.name=="snapshot")]}phase={.ph
ase}{"\n"}{range .conditions[*]}{.type}={.status} reason={.reason} messag
e={.message}{"\n"}{end}{end}'
```

If you need to stop ACP snapshot management while keeping snapshot CRDs and snapshot resources, set the `snapshot` entry to `enabled: false`:

```
kubectl patch storagefoundation default --type=json -p "$(
  kubectl get storagefoundation default -o json | jq -c '
    (.spec.operators // []) as $operators |
    (
      if any($operators[]?; .name == "snapshot") then
        $operators | map(
          if .name == "snapshot" then
            .enabled = false | .upgradePolicy.mode = "Automatic"
          else
            .
          end
        )
      else
        $operators + [{
          "name": "snapshot",
          "enabled": false,
          "upgradePolicy": {"mode": "Automatic"}
        }]
      end
    ) as $updated |
    [{
      "op": (if .spec.operators == null then "add" else "replace" end),
      "path": "/spec/operators",
      "value": $updated
    }]
  '
)"
```

To restore the legacy plugin, install **Alauda Container Platform Snapshot Management** again from **Administrator** > **Marketplace** > **Cluster Plugins** in the global cluster. Do this only if ACP snapshot management has been disabled and its controller is no longer running.

Troubleshooting

Snapshot status reports `Skipped`

`Skipped` means the cluster already has one or more snapshot CRDs and ACP did not automatically take over snapshot management. During migration from the legacy plugin, this is expected until you explicitly enable ACP snapshot management:

```
name: snapshot
enabled: true
```

Snapshot operations remain pending

Check whether a Snapshot Controller is running:

```
kubectl get deployment -A | grep -E 'snapshot-controller|snapshot'
```

If no controller is running and the snapshot status reports `Skipped`, confirm that the legacy plugin has been uninstalled, then enable ACP snapshot management.

Snapshot management was enabled while another controller is still running

This can create two Snapshot Controllers in the same cluster. Disable ACP snapshot management first:

```
kubectl patch storagefoundation default --type=json -p "$(
  kubectl get storagefoundation default -o json | jq -c '
    (.spec.operators // []) as $operators |
    (
      if any($operators[]?; .name == "snapshot") then
        $operators | map(
          if .name == "snapshot" then
            .enabled = false | .upgradePolicy.mode = "Automatic"
          else
            .
          end
        )
      else
        $operators + [{
          "name": "snapshot",
          "enabled": false,
          "upgradePolicy": {"mode": "Automatic"}
        }]
      end
    ) as $updated |
    [{
      "op": (if .spec.operators == null then "add" else "replace" end),
      "path": "/spec/operators",
      "value": $updated
    }]
  '
)"
```

After the ACP snapshot component is removed, decide which component should provide the cluster-level snapshot capability. Only one Snapshot Controller should remain.

Related Documentation

- Deploy Volume Snapshot Component
- Using Volume Snapshots
- Cluster Plugin

Configuring Persistent Storage Using NFS

ACP clusters support persistent storage using NFS. PersistentVolumes (PVs) and PersistentVolumeClaims (PVCs) provide an abstraction layer for provisioning and consuming storage volumes within a project. While NFS configuration details can be embedded directly in a Pod definition, this approach does not create the volume as a distinct, isolated cluster resource, increasing the risk of conflicts.

TOC

- Prerequisites
- Procedure
  - Create an object definition for the PV
  - Verify that the PV was created
  - Create a PVC that references the PV
  - Verify that the persistent volume claim was created
- Enforcing Disk Quotas via Partitioned Exports
- NFS volume security
  - Group IDs
  - User IDs
  - Export settings
- Reclaiming resources

Prerequisites

- Storage must exist in the underlying infrastructure before it can be mounted as a volume in ACP.
- To provision NFS volumes, a list of NFS servers and export paths are all that is required.

Procedure

1 Create an object definition for the PV

```
cat << EOF | kubectl create -f -
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-nfs-example 1
spec:
  capacity:
    storage: 1Gi 2
  accessModes:
  - ReadWriteOnce 3
  nfs: 4
    path: /tmp 5
    server: 10.0.0.3 6
  persistentVolumeReclaimPolicy: Retain 7
EOF
```

1 The name of the volume.

2 Amount of storage.

3 Though this appears to be related to controlling access to the volume, it is actually used similarly to labels and used to match a PVC to a PV. Currently, no access rules are enforced based on the accessModes.

4 The volume type being used, in this case the nfs plugin.

5 The NFS server address.

6 The NFS export path.

7 What happens after PVC is deleted (Retain, Delete, Recycle).

2 Verify that the PV was created

Command

```
kubectl get pv
```

Output Example

```
NAME             CAPACITY   ACCESS MODES   RECLAIM POLICY   STATU
S   CLAIM   STORAGECLASS   REASON   AGE
pv-nfs-example   1Gi         RWO            Retain           Avail
able                                 10s
```

3 Create a PVC that references the PV

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: nfs-claim1
spec:
  accessModes:
  - ReadWriteOnce 1
  resources:
    requests:
      storage: 1Gi 2
  volumeName: pv-nfs-example 3
  storageClassName: ""
```

1 The access modes do not enforce security, but rather act as labels to match a PV to a PVC.

2 This claim looks for PVs offering 1Gi or greater capacity.

3 The name of the PV to be used.

4 Verify that the persistent volume claim was created

Command

```
kubectl get pvc
```

Output Example

```
NAME            STATUS   VOLUME           CAPACITY   ACCESS MODE
S   STORAGECLASS   AGE
nfs-claim1      Bound    pv-nfs-example   1Gi        RWO
10s
```

Enforcing Disk Quotas via Partitioned Exports

To enforce disk quotas and size constraints, you can utilize disk partitions. Assign each partition as a dedicated export point, with each export corresponding to a distinct PersistentVolume (PV).

While ACP requires unique PV names, administrators must also ensure that the NFS server and path combination is unique for each export.

This partitioned approach enables precise capacity management. Developers request persistent storage by specifying a required amount, such as `10Gi`, and ACP matches the request to a PV backed by a partition or export that offers at least that capacity. Quota enforcement applies to the usable storage space within the assigned partition or export.

NFS volume security

NFS volume security depends on permission matching. Review POSIX permissions, process UIDs, and supplemental groups before configuring NFS volumes for shared use.

Developers request NFS storage through either:

- A PersistentVolumeClaim (PVC) reference by name, or

- Direct configuration of the NFS volume plugin in the volumes section of their Pod specification.

On the NFS server, the /etc/exports file defines export rules for accessible directories. Each exported directory retains its native POSIX owner/group IDs.

Key behavior of the ACP NFS plugin:

1. Mounts volumes to containers while preserving exact POSIX ownership and permissions from the source directory
2. Executes containers without forcing process UIDs to match the mount ownership - an intentional security measure

For example, consider an NFS directory with these server-side attributes:

Command

```
ls -l /share/nfs -d
```

Output Example

```
drwxrws---. nfsnobody 5555   /share/nfs
```

Command

```
id nfsnobody
```

Output Example

```
uid=65534(nfsnobody) gid=65534(nfsnobody) groups=65534(nfsnobody)
```

Then the container must either run with a UID of 65534, the nfsnobody owner, or with 5555 in its supplemental groups to access the directory.

> **NOTE**
>
> Note The owner ID of 65534 is used as an example. Even though NFS's root_squash maps root, uid 0, to nfsnobody, uid 65534, NFS exports can have arbitrary owner IDs. Owner 65534 is not required for NFS exports.

Group IDs

Recommended NFS Access Management (When Export Permissions Are Fixed) When modifying permissions on the NFS export is not feasible, the recommended approach for managing access is through supplemental groups.

Supplemental groups in ACP are a common mechanism for controlling access to shared file storage, such as NFS.

Contrast with Block Storage: Access to block storage volumes (e.g., iSCSI) is typically managed by setting the fsGroup value within the pod's securityContext. This approach leverages filesystem group ownership change upon mount.

> **NOTE**
>
> To gain access to persistent storage, it is generally preferable to use supplemental group IDs versus user IDs.

Because the group ID on the example target NFS directory is 5555, the pod can define that group ID using supplementalGroups under the securityContext definition of the pod. For example:

```
spec:
  containers:
    - name:
    ...
  securityContext: 1
    supplementalGroups: [5555] 2
```

1 securityContext must be defined at the pod level, not under a specific container.

(2) An array of GIDs defined for the pod. In this case, there is one element in the array. Additional GIDs would be comma-separated.

User IDs

User IDs can be defined in the container image or in the Pod definition.

> **NOTE**
>
> It is generally preferable to use supplemental group IDs to gain access to persistent storage versus using user IDs.

In the example target NFS directory shown above, the container needs its UID set to 65534, ignoring group IDs for the moment, so the following can be added to the Pod definition:

```
spec:
  containers: (1)
  - name:
  ...
    securityContext:
      runAsUser: 65534 (2)
```

(1) Pods contain a securityContext definition specific to each container and a pod's securityContext which applies to all containers defined in the pod.

(2) 65534 is the nfsnobody user.

Export settings

To enable arbitrary container users to read and write the volume, each exported volume on the NFS server should conform to the following conditions:

- Every export must be exported using the following format:

```
# replace 10.0.0.0/24 to trusted CIDRs/hosts
/<example_fs> 10.0.0.0/24(rw,sync,root_squash,no_subtree_check)
```

- The firewall must be configured to allow traffic to the mount point.
  - For NFSv4, configure the default port 2049 (nfs).

```
iptables -I INPUT 1 -p tcp --dport 2049 -j ACCEPT
```

  - For NFSv3, there are three ports to configure: 2049 (nfs), 20048 (mountd), and 111 (portmapper).

```
iptables -I INPUT 1 -p tcp --dport 2049 -j ACCEPT
iptables -I INPUT 1 -p tcp --dport 20048 -j ACCEPT
iptables -I INPUT 1 -p tcp --dport 111 -j ACCEPT
```

- The NFS export and directory must be set up so that they are accessible by the target pods. Either set the export to be owned by the container's primary UID, or supply the pod group access using supplementalGroups, as shown in the group IDs above.

Reclaiming resources

NFS implements the Alauda Container Platform Recyclable plugin interface. Automatic processes handle reclamation tasks based on policies set on each persistent volume.

By default, PVs are set to Retain.

Once claim to a PVC is deleted, and the PV is released, the PV object should not be reused. Instead, a new PV should be created with the same basic volume details as the original.

For example, the administrator creates a PV named nfs1:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: nfs1
spec:
  capacity:
    storage: 1Mi
  accessModes:
    - ReadWriteMany
  nfs:
    server: 192.168.1.1
    path: "/"
```

The user creates PVC1, which binds to nfs1. The user then deletes PVC1, releasing claim to nfs1. This results in nfs1 being Released. If the administrator wants to make the same NFS share available, they should create a new PV with the same NFS server details, but a different PV name:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: nfs2
spec:
  capacity:
    storage: 1Mi
  accessModes:
    - ReadWriteMany
  nfs:
    server: 192.168.1.1
    path: "/"
```

Deleting the original PV and re-creating it with the same name is discouraged. Attempting to manually change the status of a PV from Released to Available causes errors and potential data loss.

Configuring Disaster Recovery For Persistent Volume Claims

To achieve cross-cluster disaster recovery for Application PVCs, use **Alauda Storage Volume Replication**.

TOC

- Overview
- Terminology
- Prerequisites
- Deploy Alauda Storage Volume Replication
- Configuring a Scheduled Synchronization
  - Create a rsync-tls Data Mover Secret
  - Create ReplicationDestination Resource
  - Create ReplicationSource Resource
  - Check Synchronization Status
- Configuring a One-Time Synchronization
  - Create One-Time ReplicationSource Resource
  - Check Synchronization Status
- Enable Disaster Recovery for the Application PVC
  - Deploy stateful application
  - Configuring PVC Disaster Recovery
- Planned Migration

Procedures
Failover
Procedures
Failback (post-disaster recovery)
Procedures

Overview

Alauda Storage Volume Replication is an operator that performs asynchronous replication of persistent volumes within or across clusters. The replication it provides, based on VolSync, is independent of the storage system. This allows replication to and from storage types that don't normally support remote replication. Additionally, it can replicate across different types (and vendors) of storage.

Terminology

Term	Explanation
Primary Cluster	Active production site.
Secondary Cluster	Standby recovery site; remains on standby, ready to take over during a disaster.
Stateful Application	Applications that use PVCs for data persistence.
ReplicationSource	A ReplicationSource is a resource that you can use to define the source PVC and replication mover type, enabling you to replicate or synchronize PVC data to a remote location
ReplicationDestination	A ReplicationDestination is a resource that you can use to define the destination of a volume replication or synchronization.
Data Movers	The data movers are responsible for copying the data from one location to the other.

Term	Explanation
	Supported movers: • Rclone • Restic • Rsync

Prerequisites

- **Download** the **Alauda Storage Volume Replication** installation package corresponding to your platform architecture.
- **Upload** the **Alauda Storage Volume Replication** installation package using the Upload Packages mechanism to both Primary and Secondary clusters.
- **Alauda Container Platform Snapshot Management** has been deployed on both the Primary and Secondary clusters.
- The storage used by the PVC must be provisioned by the **CSI** and support **snapshot** functionality.

Deploy Alauda Storage Volume Replication

1. Login, go to the **Administrator** page.
2. Click **Marketplace** > **OperatorHub** to enter the **OperatorHub** page.
3. Find the **Alauda Storage Volume Replication**, click **Install**, and navigate to the **Install Alauda Storage Volume Replication** page.
   Configuration Parameters:

Parameter	Recommended Configuration
Channel	The default channel is `stable`.

Parameter	Recommended Configuration
Installation Mode	`Cluster`: All namespaces in the cluster share a single Operator instance for creation and management, resulting in lower resource usage.
Installation Place	Select `Recommended`, Namespace only support **volsync-system**.
Upgrade Strategy	`Manual`: When there is a new version in the Operator Hub, manual confirmation is required to upgrade the Operator to the latest version.

Configuring a Scheduled Synchronization

After configuring Scheduled Synchronization for a PVC, the operator will automatically synchronize the data from the `ReplicationSource` to the `ReplicationDestination` at the specified interval.

Configure synchronization from the Primary cluster to the Secondary cluster. For synchronization from the Secondary cluster to the Primary cluster, adapt the example below by swapping the cluster roles.

1 Create a rsync-tls Data Mover Secret

Create the Secret on both **Primary** and **Secondary** clusters; skip this step if the Secret already exists.

Command

```yaml
apiVersion: v1
data:
  psk.txt: <psk>
kind: Secret
metadata:
  name: <name>
  namespace: <namespace>
type: Opaque
```

Example

```yaml
apiVersion: v1
data:
  # echo -n 1:23b7395fafc3e842bd8ac0fe142e6ad1 | base64
  psk.txt: MToyM2I3Mzk1ZmFmYzNlODQyYmQ4YWMwZmUxNDJlNmFkMQ==
kind: Secret
metadata:
  name: volsync-rsync-tls
  namespace: default
type: Opaque
```

Parameters:

Parameter	Explanation
name	The name of secret
namespace	The namespace of secret, should same as application
psk	This field adheres to the format expected by stunnel: `<id>:<at least 32 hex digits>`. for example, `1:23b7395fafc3e842bd8ac0fe142e6ad1`.

2 Create ReplicationDestination Resource

Create `ReplicationDestination` on **Secondary** cluster

Command

```
cat << EOF | kubectl create -f -
apiVersion: volsync.backube/v1alpha1
kind: ReplicationDestination
metadata:
  name: rd-<pvc-name>
  namespace: <namespace>
spec:
  rsyncTLS:
    copyMethod: Snapshot
    destinationPVC: <pvc-name>
    keySecret: <key-secret>
    serviceType: <service-type>
    storageClassName: <storageclass-name>
    volumeSnapshotClassName: <volumesnapshotclass-name>
    moverSecurityContext:
      fsGroup: 65534
      runAsGroup: 65534
      runAsNonRoot: true
      runAsUser: 65534
      seccompProfile:
        type: RuntimeDefault
EOF
```

Example

```
cat << EOF | kubectl create -f -
apiVersion: volsync.backube/v1alpha1
kind: ReplicationDestination
metadata:
  name: rd-pvc-01
  namespace: default
spec:
  rsyncTLS:
    copyMethod: Snapshot
    destinationPVC: pvc-01
    keySecret: volsync-rsync-tls
    serviceType: NodePort
    storageClassName: sc-cephfs
    volumeSnapshotClassName: csi-cephfs-snapshotclass
    moverSecurityContext:
      fsGroup: 65534
      runAsGroup: 65534
      runAsNonRoot: true
      runAsUser: 65534
      seccompProfile:
        type: RuntimeDefault
EOF
```

Parameters:

Parameter	Explanation
namespace	Namespace same as application
pvc-name	The name of a **pre-existing** PVC, which application used
key-secret	This is the name of a Secret that contains the TLS-PSK key for authenticating the connection with the source, create at Step 1
service-type	The operator creates a Service to allow the source to connect to the destination. This field determines the type of that Service. Allowed values are `ClusterIP` or `LoadBalancer` or `NodePort`.
storageclass-name	The name of storageclass which application pvc used

Parameter	Explanation
volumesnapshotclass-name	The name of volumesnapshotclass, corresponding to the application pvc

NOTE

About service type

If `ClusterIP` is specified, the Service will receive an IP address allocated from the “cluster network” address pool. By default, this collection of addresses are not accessible from outside the cluster, making it a poor choice for cross-cluster replication. However, various networking addons such as Submariner bridge the cluster networks, making this a good option.

If `LoadBalancer` is specified, an externally accessible IP address will be allocated. This requires cluster support for load balancers such as those provided by the various cloud providers or MetalLB in the case of physical clusters. While this is the easiest method for allocating an accessible address in cloud environments, load balancers tend to incur additional costs and be limited in number.

3 Create ReplicationSource Resource

Create `ReplicationSource` on **Primary** cluster

Command

```
cat << EOF | kubectl create -f -
apiVersion: volsync.backube/v1alpha1
kind: ReplicationSource
metadata:
  name: rs-<pvc-name>
  namespace: <namespace>
spec:
  rsyncTLS:
    address: <address>
    copyMethod: Snapshot
    keySecret: <key-secret>
    port: <port>
    storageClassName: <storageclass-name>
    volumeSnapshotClassName: <volumesnapshotclass-name>
    moverSecurityContext:
      fsGroup: 65534
      runAsGroup: 65534
      runAsNonRoot: true
      runAsUser: 65534
      seccompProfile:
        type: RuntimeDefault
  sourcePVC: <pvc-name>
  trigger:
    schedule: <schedule>
EOF
```

Example

```
cat << EOF | kubectl create -f -
apiVersion: volsync.backube/v1alpha1
kind: ReplicationSource
metadata:
  name: rs-pvc-01
  namespace: default
spec:
  rsyncTLS:
    address: 192.168.129.201
    copyMethod: Snapshot
    keySecret: volsync-rsync-tls
    port: 30532
    storageClassName: sc-cephfs
    volumeSnapshotClassName: csi-cephfs-snapshotclass
    moverSecurityContext:
      fsGroup: 65534
      runAsGroup: 65534
      runAsNonRoot: true
      runAsUser: 65534
      seccompProfile:
        type: RuntimeDefault
  sourcePVC: pvc-01
  trigger:
    schedule: "*/10 * * * *"
EOF
```

Parameters:

Parameter	Explanation
namespace	The name of Namespace, same as application
pvc-name	The name of the application pvc
key-secret	The name of the Secret created at Step 1
address	This specifies the address of the replication destination's ssh server. It can be taken directly from the ReplicationDestination's `.status.rsync.address` field.

Parameter	Explanation
port	service port to connect to the destination
storageclass-name	The name of storageclass which application pvc used
volumesnapshotclass-name	The name of volumesnapshotclass, corresponding to the application pvc
schedule	The synchronization schedule, is defined by a cronspec, making the schedule very flexible. Both intervals as well as specific times and/or days can be specified.

4 Check Synchronization Status

Check synchronization from `ReplicationSource`

Command

```
kubectl -n <namespace> get ReplicationSource <rs-name> -o jsonpath='{.status}'
```

Example

```
kubectl -n default get ReplicationSource rs-pvc-01 -o jsonpath='{.status}'
```

The last synchronization was completed at `.status.lastSyncTime` and took `.status.lastSyncDuration` seconds.

The next scheduled synchronization is at `.status.nextSyncTime`.

Configuring a One-Time Synchronization

One-Time Synchronization is initiated manually. This is controlled by setting a unique string for the manual field under the trigger specification in a ReplicationSource resource. The synchronization job runs once immediately upon applying the configuration.

1 Create One-Time ReplicationSource Resource

Command

```
cat << EOF | kubectl create -f -
apiVersion: volsync.backube/v1alpha1
kind: ReplicationSource
metadata:
  name: rs-<pvc-name>-latest
  namespace: <namespace>
spec:
  rsyncTLS:
    address: <address>
    copyMethod: Snapshot
    keySecret: <key-secret>
    port: <port>
    storageClassName: <storageclass-name>
    volumeSnapshotClassName: <volumesnapshotclass-name>
    moverSecurityContext:
      fsGroup: 65534
      runAsGroup: 65534
      runAsNonRoot: true
      runAsUser: 65534
      seccompProfile:
        type: RuntimeDefault
  sourcePVC: <pvc-name>
  trigger:
    manual: <manual-id>
EOF
```

Example

```
cat << EOF | kubectl create -f -
apiVersion: volsync.backube/v1alpha1
kind: ReplicationSource
metadata:
  name: rs-pvc-01-latest
  namespace: default
spec:
  rsyncTLS:
    address: 192.168.129.201
    copyMethod: Snapshot
    keySecret: volsync-rsync-tls
    port: 30532
    storageClassName: sc-cephfs
    volumeSnapshotClassName: csi-cephfs-snapshotclass
    moverSecurityContext:
      fsGroup: 65534
      runAsGroup: 65534
      runAsNonRoot: true
      runAsUser: 65534
      seccompProfile:
        type: RuntimeDefault
  sourcePVC: pvc-01
  trigger:
    manual: latest
```

The only difference from Scheduled Synchronization is `.spec.trigger` should set to **manual**.

2 Check Synchronization Status

Command

```
kubectl -n <namespace> get ReplicationSource <rs-name> -o jsonpath='{.status.lastManualSync}'
```

Example

```
kubectl -n default get ReplicationSource rs-pvc-01-latest -o jso
npath='{.status.lastManualSync}'
```

If the output matches `<manual-id>`, the synchronization is complete.

Enable Disaster Recovery for the Application PVC

1 Deploy stateful application

1. Deploy stateful applications on **Primary** cluster

▶ Click to view

2. Create application pvc on **Secondary** cluster

```
cat << EOF | kubectl create -f -
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-01
  namespace: default
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 10Gi
  storageClassName: sc-cephfs
  volumeMode: Filesystem
EOF
```

2 Configuring PVC Disaster Recovery

Set up `Primary-to-Secondary` Synchronization

refers to Configuring a Scheduled Synchronization

Planned Migration

User Scenario:

Relocate business services from the Primary cluster to the Secondary cluster while both clusters are operating normally.

Procedures

1. **Scale down application pods**
   Scale down all the application pods which are using the dr PVC on the **Primary** cluster.
2. **Delete ReplicationSource Resource**
   Delete `ReplicationSource` on **Primary** cluster

   Command

   ```
   kubectl -n <namespace> delete ReplicationSource <rs-name>
   ```

   Example

   ```
   kubectl -n default delete ReplicationSource rs-pvc-01
   ```

3. **Create One-Time Synchronization**
   Initiate a synchronization task from the **Primary** cluster to guarantee that the data in the **Secondary** cluster is `up-to-date`.
   Create `ReplicationSource` on **Primary** cluster
   Refers to Configuring a One-Time Synchronization
4. **Delete One-Time Synchronization**
   After One-Time synchronization completed, delete One-Time `ReplicationSource` resource

   Command

```
kubectl -n <namespace> delete ReplicationSource <rs-name>
```

Example

```
kubectl -n default delete ReplicationSource rs-pvc-01-latest
```

5. **Delete ReplicationDestination Resource**

   Delete `ReplicationDestination` on **Secondary** cluster

   Command

   ```
   kubectl -n <namespace> delete ReplicationDestination <rd-name>
   ```

   Example

   ```
   kubectl -n default delete ReplicationDestination rd-pvc-01
   ```

6. **Scale up application pods**

   Scale up all the application pods which are using the dr PVC on the **Secondary** cluster.

7. **Set up secondary-to-primary Synchronization**

   Set up the `Secondary-to-Primary` cluster synchronization for PVC disaster recovery by creating a `ReplicationDestination` on the **Primary** cluster and a `ReplicationSource` on the **Secondary** cluster.

   Refers to Configuring a Scheduled Synchronization

Failover

User Scenario:

Switching services to the Secondary cluster after Primary cluster abrupt shutdown.

Procedures

To ensure data integrity (in case the primary cluster experiences failures during synchronization), do a local synchronization on the **Secondary** cluster. Use the PVC restored from the last snapshot of the application's PVC as the source, and the application's current PVC as the destination to perform a data synchronization.

1. **Restore PVC**

   Restore PVC from `ReplicationDestination` on **Secondary** cluster

   Command

   ```
   cat << EOF | kubectl create -f -
   apiVersion: v1
   kind: PersistentVolumeClaim
   metadata:
     name: restored-<pvc-name>
     namespace: <namespace>
   spec:
     accessModes: [<access-modes>]
     dataSourceRef:
       kind: ReplicationDestination
       apiGroup: volsync.backube
       name: <rd-name>
     resources:
       requests:
         storage: <pvc-size>
     storageClassName: <storageclass-name>
   EOF
   ```

   Example

```
cat << EOF | kubectl create -f -
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: restored-pvc-01
  namespace: default
spec:
  accessModes: [ReadWriteMany]
  dataSourceRef:
    kind: ReplicationDestination
    apiGroup: volsync.backube
    name: rd-pvc-01
  resources:
    requests:
      storage: 10Gi
  storageClassName: sc-cephfs
EOF
```

2. **Create local ReplicationSource Resource**

   Create `ReplicationSource` Resource on **Secondary** cluster

   Command

```
cat << EOF | kubectl create -f -
apiVersion: volsync.backube/v1alpha1
kind: ReplicationSource
metadata:
  name: rs-<pvc-name>-local
  namespace: <namespace>
spec:
  rsyncTLS:
    address: <address>
    copyMethod: Snapshot
    keySecret: <key-secret>
    port: <port>
    storageClassName: <storageclass-name>
    volumeSnapshotClassName: <volumesnapshotclass-name>
    moverSecurityContext:
      fsGroup: 65534
      runAsGroup: 65534
      runAsNonRoot: true
      runAsUser: 65534
      seccompProfile:
        type: RuntimeDefault
  sourcePVC: restored-<pvc-name>
  trigger:
    manual: <manual-id>
EOF
```

Example

```
cat << EOF | kubectl create -f -
apiVersion: volsync.backube/v1alpha1
kind: ReplicationSource
metadata:
  name: rs-pvc-01-local
  namespace: default
spec:
  rsyncTLS:
    address: 192.168.129.201
    copyMethod: Snapshot
    keySecret: volsync-rsync-tls
    port: 30532
    storageClassName: sc-cephfs
    volumeSnapshotClassName: csi-cephfs-snapshotclass
    moverSecurityContext:
      fsGroup: 65534
      runAsGroup: 65534
      runAsNonRoot: true
      runAsUser: 65534
      seccompProfile:
        type: RuntimeDefault
  sourcePVC: restored-pvc-01
  trigger:
    manual: latest
EOF
```

Parameters refers to Configuring a One-Time Synchronization

3. **Waiting for synchronization to complete**

Command

```
kubectl -n <namespace> get ReplicationSource <rs-name> -o jsonpath='{.status.lastManualSync}'
```

Example

```
kubectl -n default get ReplicationSource rs-pvc-01-local -o jsonpath='{.status.lastManualSync}'
```

If the output matches `<manual-id>`, the synchronization is complete.

4. **Delete local ReplicationSource**

   Delete local `ReplicationSource` on **Secondary** cluster

   Command

   ```
   kubectl -n <namespace> delete ReplicationSource <rs-name>
   ```

   Example

   ```
   kubectl -n default delete ReplicationSource rs-pvc-01-local
   ```

5. **Delete ReplicationDestination**

   Delete `ReplicationDestination` on **Secondary** cluster

   Command

   ```
   kubectl -n <namespace> delete ReplicationDestination <rd-name>
   ```

   Example

   ```
   kubectl -n default delete ReplicationDestination rd-pvc-01
   ```

6. **Scale up application pods**

   Scale up all the application pods on **Secondary** cluster.

Failback (post-disaster recovery)

User Scenario:

The primary cluster has now been restored and is operational, necessitating a switchback of services to it.

Procedures

1. **Scale down application pods on Primary cluster**
   When the primary cluster is back online, application pods will recover automatically. However, the service must first be scaled down to halt traffic. After synchronizing the latest data from the secondary cluster to the primary cluster, the application can then be scaled up to resume normal operation.
2. **Delete ReplicationSource on Primary cluster**
   The `ReplicationSource` created before the **Primary** cluster failed needs to be deleted first.

   Command

   ```
   kubectl -n <namespace> delete ReplicationSource <rs-name>
   ```

   Example

   ```
   kubectl -n default delete ReplicationSource rs-pvc-01
   ```

3. **Syncing latest data from Secondary cluster**
   Set up a `Secondary-to-Primary` One-Time Synchronization.
   Create a `ReplicationDestination` on **Primary** cluster, and then create a one-time `ReplicationSource` on **Secondary** cluster
   Refers to Configuring a One-Time Synchronization
4. **Delete ReplicationDestination and ReplicationSource**
   After data synchronization, delete one-time resources

Delete `ReplicationSource` on **Secondary** cluster

Command

```
kubectl -n <namespace> delete ReplicationSource <rs-name>
```

Example

```
kubectl -n default delete ReplicationSource rs-pvc-01-latest
```

Delete `ReplicationDestination` on **Primary** cluster

Command

```
kubectl -n <namespace> delete ReplicationDestination <rd-name>
```

Example

```
kubectl -n default delete ReplicationDestination rd-pvc-01
```

5. **Migrate application**

   Scale down application pods on **Secondary** cluster

   Scale up application pods on **Primary** cluster

6. **Set up Primary-to-Secondary Synchronization**

   Refers to Configuring a Scheduled Synchronization

VolSync is a project of the backube community. Alauda is an independent vendor. This product is not affiliated with, endorsed by, or sponsored by the backube project. All trademarks are the property of their respective owners and are used here for identification purposes only.

Third-Party Storage Capability Annotation Guide

> **Feature Overview:** By adding a **StorageDescription** ConfigMap in the `kube-public` namespace, the platform automatically detects each third-party StorageClass's snapshot support as well as supported volume modes and access modes (including block-specific access modes). The PVC creation screen will then display only the valid options, helping you choose and use the right storage features with ease.

TOC

1. Getting Started
   1.1 Create or Update the ConfigMap
   1.2 Populate the `data` field
   1.3 Apply the configuration
2. Sample ConfigMap
3. Update Existing Capability Descriptions
4. Compatibility with the Legacy Format
5. Frequently Asked Questions

---

1. Getting Started

1.1 Create or Update the ConfigMap

> **Important:** Perform the following operation **in the `kube-public` namespace**, otherwise the platform will not recognize the storage capabilities.

Edit or create a ConfigMap whose name starts with `sd-`, for example `sd-capabilities-enhanced`:

```
kubectl -n kube-public edit configmap sd-capabilities-enhanced
```

Required label

```
metadata:
  labels:
    features.alauda.io/type: StorageDescription
```

1.2 Populate the `data` field

Each `key` corresponds to a StorageClass `provisioner`; the value is a YAML string that describes its capabilities. Key fields:

Field	Type	Description
`snapshot`	`Boolean`	Indicates whether volume snapshots are supported
`volumeMode`	`List[String]`	Supported volume modes; at least one of `Filesystem`, `Block`
`accessModes`	`List[String]`	Access modes available when `volumeMode` is `Filesystem`
`blockAccessModes`	`List[String]`	Access modes specific to `Block` volumes (optional)

> If `blockAccessModes` is omitted, the platform will fall back to `accessModes` for Block volumes.

1.3 Apply the configuration

```
kubectl apply -f sd-capabilities-enhanced.yaml
```

Once applied, the UI automatically adjusts available options, for example:

- When **Block** volume mode is selected, the access-mode dropdown is populated with `blockAccessModes`.
- If `snapshot: true`, snapshot-related operations become available on the PVC page.

2. Sample ConfigMap

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: sd-capabilities-enhanced
  namespace: kube-public
  labels:
    features.alauda.io/type: StorageDescription
data:
  storage.advanced-block-fs.com: |-
    snapshot: true
    volumeMode:
      - Filesystem
      - Block
    accessModes:
      - ReadWriteOnce
      - ReadOnlyMany
    blockAccessModes:
      - ReadWriteOnce
  storage.filesystem-basic.com: |-
    snapshot: false
    volumeMode:
      - Filesystem
    accessModes:
      - ReadWriteOnce
      - ReadWriteMany
```

3. Update Existing Capability Descriptions

1. Locate the `provisioner` key you want to modify.
2. Adjust the field values to reflect the actual capabilities.
3. Re-apply the ConfigMap with `kubectl apply -f ...`. The platform polls for updates and refreshes the UI automatically; you can also refresh the browser to see the changes immediately.

4. Compatibility with the Legacy Format

- If `blockAccessModes` is missing, Block volumes will inherit `accessModes`.
- You do not need to delete old ConfigMaps; simply add the new fields for a smooth upgrade.

5. Frequently Asked Questions

Symptom	Possible Cause	Resolution
Access-mode list empty for Block volumes	`blockAccessModes` is empty and `accessModes` is also empty	Provide at least one of the two
UI still shows outdated capabilities	ConfigMap not saved or browser cache	Verify with `kubectl get cm`, reload

Troubleshooting

Recover From PVC Expansion I

Procedure

Additional Tips

Detach CSI Volumes After Non-Graceful Node

When to use this procedure

Prerequisites

Procedure

Verify the taint

Remove the taint after node recovery

Recover From PVC Expansion Failure

When PVC expansion fails in Kubernetes, administrators can manually recover the Persistent Volume Claim (PVC) state and cancel the expansion request.

TOC

Procedure

Additional Tips

Procedure

1. Modify the reclaim policy of the Persistent Volume (PV) bound to the PVC to `Retain`. To do this, edit the corresponding PV and set the `persistentVolumeReclaimPolicy` field to `Retain`.
2. Delete the original PVC.
3. Manually edit the PV to remove the `claimRef` entry from its specifications. This ensures that the new PVC can bind to this PV, changing the PV's status to `Available`.
4. Recreate a new PVC with a smaller size or a size supported by the underlying storage provider.
5. Explicitly specify the `volumeName` field in the new PVC to match the original PV name. This ensures that the new PVC accurately binds to the specified PV.
6. Finally, restore the original reclaim policy of the PV.

Additional Tips

- Ensure that the `StorageClass` in use has volume expansion enabled by setting `allowVolumeExpansion` to `true`.
- Perform these actions carefully to avoid the risk of data loss.

Detach CSI Volumes After Non-Graceful Node Shutdown

If a node shuts down unexpectedly, the kubelet might not report the shutdown event to the control plane. In this case, Pods that use CSI-backed persistent volumes can remain stuck on the failed node, and the volumes might not detach automatically. In ACP, you can manually add an `out-of-service` taint to trigger Pod eviction and volume detachment.

TOC

When to use this procedure

Prerequisites

Procedure

Verify the taint

Remove the taint after node recovery

When to use this procedure

Use this procedure when all of the following conditions are true:

- A worker node has stopped because of a power loss, operating system failure, or hardware fault.
- The node did not complete a graceful shutdown.

- Workloads that use CSI-backed persistent volumes cannot restart on another node because the volumes are still attached to the failed node.

Prerequisites

- You know the name of the affected node.
- The affected node is completely powered off or otherwise confirmed to be unavailable.

> **WARNING**
>
> Do not apply the `out-of-service` taint to a node that is still running or still recovering. If the node is not fully shut down, forcing volume detachment can cause file system corruption or application-level data loss.

Procedure

1. Check the status of the affected node.

   ```
   kubectl get node <node_name>
   ```

2. If the node is still partially reachable, shut it down completely at the infrastructure layer before you continue.
3. Add the `out-of-service` taint to the node.

   ```
   kubectl taint node <node_name> node.kubernetes.io/out-of-service=nodeshutdown:NoExecute
   ```

   After the taint is applied, Kubernetes starts evicting Pods from the failed node. The attach-detach controller can then detach CSI volumes so that replacement Pods can attach the same volumes on another node.

> **WARNING**

The `NoExecute` effect can cause Pods to be deleted and recreated on other nodes. If the workload is managed by a StatefulSet, the replacement Pod starts only after the original volume is detached successfully.

Verify the taint

Run the following command:

```
kubectl describe node <node_name>
```

Verify that the node includes the following taint:

```
node.kubernetes.io/out-of-service=nodeshutdown:NoExecute
```

You can also check whether the affected Pods are being rescheduled and whether the CSI volumes are detached from the failed node.

Remove the taint after node recovery

After the node is repaired and is ready to return to service, remove the taint:

```
kubectl taint node <node_name> node.kubernetes.io/out-of-service=nodeshut
down:NoExecute-
```

If the node was replaced instead of recovered, you can remove the old Node object from the cluster after confirming that workloads are healthy on the replacement node.

Object Storage

Introduction

Introduction

Limitations

Concepts

Concepts

Overview

Core Resources

Resource Interaction Workflow

Summary

Installing

Installing

Prerequisites

Install COSI

Uninstallation

Guides

Creating a BucketClass for Cep

Prerequisites

Step 1 – Prepare a Ceph Cluster

Step 2 – Install the COSI Plug-in

Step 3 – Prepare the Credential Secret

Step 4 – Create the BucketClass

Verification & Next Steps

Create a Bucket Request

Prerequisites

Procedure

Related

How To

Control Access & Quotas for COSI Buckets with CephObjectStoreUser (Ceph Driver)

Prerequisites

Step 1 — Create a CephObjectStoreUser (with capabilities & quotas)

Step 2 — Define a BucketClass bound to the CephObjectStoreUser

Step 3 — Provision a bucket with BucketClaim

Step 4 — Issue least-privileged credentials with BucketAccessClass/BucketAccess

Step 5 — Anonymous public read (optional)

Step 6 — Quota control: where to enforce and how to change

Operations & troubleshooting

Cleanup

Introduction

The Container Object Storage Interface (COSI) is a Kubernetes-native framework for managing object storage services through standardized and declarative APIs. COSI extends the Kubernetes storage model so workloads can request object storage resources through Kubernetes-style objects.

COSI enables administrators to define, provision, and consume object storage buckets through familiar Kubernetes-style APIs. It simplifies integration between applications and backend object storage systems, automating the lifecycle of buckets and their access credentials. With COSI, Kubernetes users can request object storage resources dynamically, reducing manual configuration overhead and improving operational efficiency.

By adopting COSI, enterprises can:

- Standardize object storage provisioning across multiple cloud and on-premises environments.
- Dynamically create and manage buckets through declarative resource definitions.
- Seamlessly distribute access credentials to workloads via Kubernetes Secrets.
- Align object storage management with Kubernetes persistent storage patterns for a unified experience.

TOC

Limitations

Limitations

- COSI is currently in alpha.
- At present, COSI only supports the Ceph RGW driver.
- Integration with legacy object storage buckets might require additional manual configurations.

Concepts

TOC

Overview

Core Resources

1. BucketClass
2. Bucket
3. BucketClaim

Resource Interaction Workflow

Summary

Overview

Kubernetes administrators who are familiar with persistent storage concepts can use Container Object Storage Interface (COSI) resources to manage object storage through declarative Kubernetes APIs. COSI provides a declarative mechanism for managing object storage, similar to existing Kubernetes persistent storage management approaches.

COSI uses three primary resources: **BucketClass**, **Bucket**, and **BucketClaim**.

Core Resources

COSI defines three essential resources:

1. BucketClass

Scope: Cluster-scoped **Analogous Kubernetes Concept:** Similar to StorageClass

BucketClass is created by cluster administrators to define specific types or service levels of buckets, including region location, redundancy policies, and performance tiers.

Key functions:

- Specifies bucket deletion policies (e.g., whether to delete the underlying bucket upon BucketClaim deletion)
- Specifies the COSI driver (driverName)
- Defines vendor-specific parameters

YAML Example:

```
apiVersion: objectstorage.k8s.io/v1alpha1
kind: BucketClass
metadata:
  name: ceph-cosi-driver-class
deletionPolicy: Delete
driverName: ceph.objectstorage.k8s.io
parameters:
  objectStoreUserSecretName: rook-ceph-object-user-object-store-user-for-
cosi
  objectStoreUserSecretNamespace: rook-ceph
```

2. Bucket

Scope: Cluster-scoped **Analogous Kubernetes Concept:** Similar to PersistentVolume (PV)

Bucket represents a Kubernetes abstraction of an actual bucket in an external object storage system.

Lifecycle management:

- **Dynamic creation**: Automatically created by the COSI controller upon receiving a BucketClaim request.

3. BucketClaim

Scope: Namespace-scoped **Analogous Kubernetes Concept:** Similar to PersistentVolumeClaim (PVC)

BucketClaim resources are created by application developers within their namespaces to request object storage buckets.

Workflow:

1. User creates a BucketClaim specifying a BucketClass.
2. The COSI controller detects the request and dynamically creates the bucket in the object storage backend based on the BucketClass definition.
3. A corresponding Bucket resource is created and bound to the BucketClaim.
4. A Secret containing bucket access credentials is generated and automatically mounted into Pods requesting the bucket.

YAML Example:

```yaml
apiVersion: objectstorage.k8s.io/v1alpha1
kind: BucketClaim
metadata:
  name: my-app-bucket-claim
  namespace: my-app-ns
spec:
  bucketClassName: ceph-standard-replicated
  protocol:
    s3: {} # Defaults populated by the driver
```

Resource Interaction Workflow

The following process demonstrates the dynamic creation flow of COSI resources in practice:

1. **Cluster administrator** creates and maintains BucketClass.
2. **Namespace user** creates a BucketClaim referencing the BucketClass.
3. **COSI controller** observes the BucketClaim, dynamically creates the bucket based on the BucketClass definition.
4. The controller generates a corresponding Bucket resource within Kubernetes.
5. BucketClaim and Bucket are bound together.
6. A Secret containing storage credentials is created for Pod use.
7. Pods mount the Secret and access the object storage.

Summary

COSI Resource	Scope	Kubernetes Analogy	Purpose
BucketClass	Cluster-scoped	StorageClass	Defines bucket types and policies
Bucket	Cluster-scoped	PersistentVolume (PV)	Kubernetes abstraction for actual bucket
BucketClaim	Namespace-scoped	PersistentVolumeClaim (PVC)	User request for bucket resources

By leveraging standardized APIs provided by COSI, Kubernetes administrators can declaratively and portably manage object storage resources, greatly enhancing integration efficiency between applications and object storage within Kubernetes clusters.

Installing

TOC

Prerequisites

Install Alauda Container Platform COSI

Constraints and Limitations

Procedure

Uninstallation

Prerequisites

1. **Download** the **Alauda Container Platform COSI** Cluster Plugin package corresponding to your platform architecture.
2. **Download** the **Alauda Container Platform COSI Driver** Cluster Plugin package.
3. **Upload** the downloaded plugin packages using the **Upload Packages** feature.
4. **Install** the plugin packages to the target cluster using the **Cluster Plugins** installation mechanism.

INFO

Upload Packages: **Platform Management** > **Marketplace** > **Upload Packages** page. Click **Help Document** on the right for instructions about publishing the Cluster Plugin to the target cluster. For

CLI guidance, see CLI.

Install Alauda Container Platform COSI

Constraints and Limitations

- Plugins are scoped to the current cluster only. You must install the required plugins individually on each cluster where you want to enable COSI features.
- The **Alauda Container Platform COSI Driver** plugin depends on the **Alauda Container Platform COSI** plugin. Ensure that the **Alauda Container Platform COSI** plugin is installed first.

Procedure

1. Log in to the platform and navigate to the **Platform Management** page.
2. Go to **Marketplace** > **Cluster Plugins** to access the list of available cluster plugins.
3. Select the target cluster where you want to install the plugins.
4. Locate the **Alauda Container Platform COSI** plugin and select **Install** from the ⋮ menu to start the installation.
5. Locate and install the **Alauda Container Platform COSI Driver** plugin.
6. Wait until the status of all plugins changes to **Installed**.

Uninstallation

1. Log in to the platform and navigate to the **Platform Management** page.
2. Go to **Marketplace** > **Cluster Plugins** to access the list of installed cluster plugins.
3. Select the target cluster where you want to remove plugins.
4. Locate the plugin you want to uninstall and select **Uninstall** from the ⋮ menu to start the uninstallation process.

5. Wait until the plugin status changes to **Ready**, indicating that it can be reinstalled if required.

Important: Before uninstalling the **Alauda Container Platform COSI** plugin, you must first uninstall the **Alauda Container Platform COSI Driver** plugin.

Guides

Creating a BucketClass for Cep

Prerequisites

Step 1 – Prepare a Ceph Cluster

Step 2 – Install the COSI Plug-in

Step 3 – Prepare the Credential Secret

Step 4 – Create the BucketClass

Verification & Next Steps

Create a Bucket Request

Prerequisites

Procedure

Related

Creating a BucketClass for Ceph RGW

Ceph Object Storage can be exposed to Kubernetes workloads through the **Container Object Storage Interface (COSI)**, providing object storage for big-data analytics, backup and restore, and machine-learning scenarios. A *BucketClass* is required before users can provision buckets.

A **BucketClass** is a template resource that specifies the storage driver, authentication secret, and the deletion policy that will be applied to every bucket created from it.

TOC

- Prerequisites
- Step 1 – Prepare a Ceph Cluster
- Step 2 – Install the COSI Plug-in
- Step 3 – Prepare the Credential Secret
  - Method A – Auto-generate (Rook-managed Ceph)
  - Method B – Manual (External Ceph)
- Step 4 – Create the BucketClass
  - Option 1 – UI Workflow
  - Option 2 – YAML (GitOps-friendly)
- Verification & Next Steps

Prerequisites

Requirement	Notes
Running Ceph cluster with RGW (S3) enabled	Internal (Rook-managed) or external cluster is acceptable.
Alauda Container Platform COSI plug-ins	Both **Alauda Container Platform COSI** and **Alauda Container Platform COSI Driver** must be installed.
Kubernetes Secret containing Ceph RGW credentials	Prepared in **Step 3** below.

Step 1 – Prepare a Ceph Cluster

Choose **one** of the following:

Option	Description
Internal Ceph	Ceph cluster deployed and managed **inside** the platform by the Rook Operator.See *create a storage service* for details.
External Ceph	Stand-alone Ceph cluster reachable from the platform network.

Step 2 – Install the COSI Plug-in

Install the following cluster plug-ins:

1. **Alauda Container Platform COSI**
2. **Alauda Container Platform COSI Driver**

> For exact commands, see *Installing*.

Step 3 – Prepare the Credential Secret

COSI retrieves RGW credentials from a Kubernetes **Secret**. Pick **one** method depending on your Ceph deployment.

Method A – Auto-generate (Rook-managed Ceph)

1. Create a **CephObjectStoreUser** in the *rook-ceph* namespace:

```
# ceph-object-store-user.yaml
apiVersion: ceph.rook.io/v1
kind: CephObjectStoreUser
metadata:
  name: user-for-cosi
  namespace: rook-ceph
spec:
  store: object-store               # name of your CephObjectStore
  capabilities:
    bucket: ["read", "write"]
    user:   ["read", "write"]
```

2. Apply the manifest:

```
kubectl apply -f ceph-object-store-user.yaml
```

3. Retrieve the autogenerated Secret name (used later):

```
kubectl get cephobjectstoreuser user-for-cosi -n rook-ceph \
  -o jsonpath='{.status.info.secretName}'
```

Method B – Manual (External Ceph)

1. Obtain **AccessKey**, **SecretKey**, and **RGW Endpoint**.
2. Create a Secret in the target project/namespace and label it so the UI can discover it:

```
kubectl create secret generic ceph-external-creds -n <YOUR_NAMESPACE> \
  --from-literal=AccessKey=<YOUR_ACCESS_KEY> \
  --from-literal=SecretKey=<YOUR_SECRET_KEY> \
  --from-literal=Endpoint=http://<YOUR_RGW_ENDPOINT>

kubectl label secret ceph-external-creds -n <YOUR_NAMESPACE> app=rook-ceph-rgw
```

> **Important:** The label `app=rook-ceph-rgw` is mandatory for the platform UI to list the Secret.

Step 4 – Create the BucketClass

Option 1 – UI Workflow

1. Navigate to **Storage** → **Object StorageClass** and click **Create Object StorageClass**.
2. Select **Ceph Object Storage** as the driver.
3. Configure the following fields:
   - **Deletion Policy** – How the underlying bucket is handled when its BucketClaim is deleted (default: `Delete`).
   - **Secret** – Pick the Secret prepared in *Step 3* (only Secrets with `app=rook-ceph-rgw` are shown).
   - **Allocate Projects** – *(Optional)* Restrict usage to specific projects.
4. Click **Create**.

Option 2 – YAML (GitOps-friendly)

Create `ceph-bucketclass.yaml` with the correct Secret references:

```yaml
apiVersion: objectstorage.k8s.io/v1alpha1
kind: BucketClass
metadata:
  name: ceph-cosi-driver
  labels:
    project.cpaas.io/ALL_ALL: "true"
driverName: ceph.objectstorage.k8s.io
deletionPolicy: Delete
parameters:
  objectStoreUserSecretName: <your-secret-name>
  objectStoreUserSecretNamespace: <your-secret-namespace>
```

Apply the manifest:

```bash
kubectl apply -f ceph-bucketclass.yaml
```

Verification & Next Steps

Verify the BucketClass:

```bash
kubectl get bucketclass
```

Once the BucketClass is ready, you can create **Bucket** or **BucketClaim** resources referencing it, thereby provisioning S3-compatible object storage for your applications.

Create a Bucket Request

Use a **Bucket Request** to dynamically create a bucket based on an **Object Storage Class** and automatically bind the two.

TOC

Prerequisites

Procedure

Related

Prerequisites

- **Install** the **Alauda Container Platform COSI** cluster plugin.
- **Install** the **Alauda Container Platform COSI Driver** cluster plugin package.

For detailed installation steps, see Installing.

Procedure

1. Switch to the **Container Platform** view.
2. In the left navigation, click **Storage > Bucket Claims**.

3. Click **Create a Bucket Request**.
4. Configure the parameters as follows.

Parameter	Description
Name	The name of the bucket request.
Object Storage Class	The Object Storage Class used to dynamically create the bucket and establish the binding.

5. Click **Create**. Wait until the status becomes `Available`, which indicates the request has been fulfilled and the binding is complete.

Related

From the bucket request details page, click **Actions** (upper-right) to **Delete bucket policy** if needed. **Warning:** deleting the bucket policy clears all data in the bucket. Proceed with caution and confirm data backup and security requirements before deletion.

How To

Control Access & Quotas for COSI Buckets with CephObjectStoreUser (Ceph Driver)

Prerequisites

Step 1 — Create a CephObjectStoreUser (with capabilities & quotas)

Step 2 — Define a BucketClass bound to the CephObjectStoreUser

Step 3 — Provision a bucket with BucketClaim

Step 4 — Issue least-privileged credentials with BucketAccessClass/BucketAccess

Step 5 — Anonymous public read (optional)

Step 6 — Quota control: where to enforce and how to change

Operations & troubleshooting

Cleanup

Control Access & Quotas for COSI Buckets with CephObjectStoreUser (Ceph Driver)

Kubernetes administrators can combine **CephObjectStoreUser (COSU)**, **BucketClass/BucketClaim**, and **BucketAccessClass/BucketAccess** to implement **least-privileged access** and **quota enforcement** for Ceph RGW-backed COSI buckets.

> **What you'll build**
>
> 1. A CephObjectStoreUser with explicit capabilities and optional per-user quotas;
> 2. A BucketClass that tells the Ceph COSI driver which COSU credentials to use;
> 3. One or more BucketClaims to provision buckets;
> 4. Fine-grained, per-workload credentials using BucketAccessClass/BucketAccess (read-only, write-only, read-write), with optional anonymous read.

TOC

Prerequisites

Step 1 — Create a CephObjectStoreUser (with capabilities & quotas)

Step 2 — Define a BucketClass bound to the CephObjectStoreUser

Step 3 — Provision a bucket with BucketClaim

Step 4 — Issue least-privileged credentials with BucketAccessClass/BucketAccess

Example `BucketAccessClass` (read-only)

Mint credentials with `BucketAccess`

Step 5 — Anonymous public read (optional)

Step 6 — Quota control: where to enforce and how to change

Operations & troubleshooting

Cleanup

Prerequisites

- A healthy Ceph cluster with RGW and Rook installed.
- COSI plugins installed.
- Cluster admin privileges (to create cluster-scoped resources).

Step 1 — Create a CephObjectStoreUser (with capabilities & quotas)

`CephObjectStoreUser` is the service account the driver uses to perform bucket operations in RGW. It must live in the `rook-ceph` namespace and target your `CephObjectStore`.

```yaml
apiVersion: ceph.rook.io/v1
kind: CephObjectStoreUser
metadata:
  name: user-for-cosi
  namespace: rook-ceph
spec:
  # Target CephObjectStore
  store: object-store
  # Required capabilities for COSI bucket lifecycle
  capabilities:
    bucket: read, write
    user: read, write
  # Optional per-user quotas enforced by RGW
  quotas:
    maxBuckets: 50         # limit number of buckets this user can own
    maxObjects: 500         # total objects across buckets (example)
    maxSize: 100Gi        # total logical size across buckets
  displayName: "User for COSI driver"
```

Get the generated access keys (Rook creates a Secret for this user):

```
kubectl get cephobjectstoreuser user-for-cosi -n rook-ceph -o yaml
# Read status.info.secretName -> the Secret holding AccessKey/SecretKey
```

> These COSU credentials are consumed by the **driver** (not by your apps) to create/delete buckets on your behalf.

Step 2 — Define a BucketClass bound to the CephObjectStoreUser

`BucketClass` instructs the driver which COSU Secret to use when provisioning buckets.

```yaml
apiVersion: objectstorage.k8s.io/v1alpha1
kind: BucketClass
metadata:
  name: ceph-cosi-driver-class
deletionPolicy: Delete
# Must match the driver name
driverName: ceph.objectstorage.k8s.io
parameters:
  # Reference the Secret created for the CephObjectStoreUser
  objectStoreUserSecretName: <secret-name-from-step-1>
  objectStoreUserSecretNamespace: rook-ceph
```

> `deletionPolicy` controls the bucket's physical lifecycle when a `BucketClaim` is deleted (`Delete` vs `Retain`).

Step 3 — Provision a bucket with BucketClaim

Create the bucket in your application namespace by referencing the `BucketClass` .

```yaml
apiVersion: objectstorage.k8s.io/v1alpha1
kind: BucketClaim
metadata:
  name: my-bucket-claim
  namespace: app-a
spec:
  bucketClassName: ceph-cosi-driver-class
  # Required by the COSI API (choose the data API you need)
  protocols:
  - S3
```

Wait until `status.bucketReady: true` and note `status.bucketName` for the actual RGW bucket name.

Step 4 — Issue least-privileged credentials with BucketAccessClass/BucketAccess

Define policy templates with `BucketAccessClass` and mint credentials per workload via `BucketAccess`. Supported policies: **`readonly`**, **`writeonly`**, **`readwrite`**. Anonymous read is available by setting `parameters.anonymous: "true"` (string).

Example `BucketAccessClass` (read-only)

```yaml
apiVersion: objectstorage.k8s.io/v1alpha1
kind: BucketAccessClass
metadata:
  name: ceph-readonly-access-class
authenticationType: KEY
driverName: ceph.objectstorage.k8s.io
parameters:
  policy: readonly
  anonymous: "false"
```

Mint credentials with `BucketAccess`

```yaml
apiVersion: objectstorage.k8s.io/v1alpha1
kind: BucketAccess
metadata:
  name: my-bucket-readonly-access
  namespace: app-a
spec:
  bucketAccessClassName: ceph-readonly-access-class
  bucketClaimName: my-bucket-claim
  credentialsSecretName: my-bucket-readonly-credentials
```

The driver writes a Secret named in `credentialsSecretName`. Decode `.data.BucketInfo` (base64) to get `secretS3.endpoint`, `accessKeyID`, and `accessSecretKey` for your S3 client.

> **Tip**: Issue **distinct** credentials per Deployment/Job to simplify rotation and revocation without disrupting other workloads.

Step 5 — Anonymous public read (optional)

If you need public static asset hosting:

```
apiVersion: objectstorage.k8s.io/v1alpha1
kind: BucketAccessClass
metadata:
  name: ceph-anonymous-readonly-class
authenticationType: KEY
driverName: ceph.objectstorage.k8s.io
parameters:
  policy: readonly
  anonymous: "true"
```

Bind it with a `BucketAccess` to your `BucketClaim`. Once granted, objects are retrievable via unauthenticated HTTP `GET` (ensure network exposure as appropriate).

Step 6 — Quota control: where to enforce and how to change

Scope: The `quotas` block on **CephObjectStoreUser** applies **per user** and is enforced by RGW across all buckets owned by that user.

- `maxBuckets`: upper bound on buckets the user can create/own.
- `maxObjects`: maximum number of objects the user can store (across buckets).
- `maxSize`: total logical size permitted for the user.

Update quotas by editing the COSU resource:

```
kubectl -n rook-ceph edit cephobjectstoreuser user-for-cosi
# Modify spec.quotas.{maxBuckets,maxObjects,maxSize}; save and exit.
# Rook reconciles and applies the new RGW user quotas.
```

> **Design choice:** Keep **COSU quotas** relatively tight to bound blast radius. Use **least-privilege policies** via BAC/BA to limit what a given set of application credentials can do

> **within** a bucket.

Operations & troubleshooting

- **Namespace placement**: `CephObjectStoreUser` and its Secret must be in `rook-ceph`. App-level resources (`BucketClaim` , `BucketAccess`) should live in the app namespace.
- **Policy not applied**: confirm `bucketAccessClassName` and `parameters.policy` in BAC (`readonly|writeonly|readwrite`).
- **Anonymous read fails**: ensure `anonymous: "true"` is a **string**, not boolean; verify endpoint exposure and HTTP access path (`/<bucket>/<object>`).
- **Can't find keys**: check the `BucketAccess` Secret and decode `.data.BucketInfo` .
- **Bucket stays not-ready**: check controller/driver logs (e.g. `kubectl -n cpaas-system logs deploy/ceph-cosi-driver -c ceph-cosi-driver`).
- **Rotation**: create a new `BucketAccess` , roll your workloads to the new Secret, then delete the old Secret/BA.

Cleanup

- Remove a workload credential by deleting its `BucketAccess` and the referenced Secret.
- Removing a bucket: delete the `BucketClaim` (behavior follows `BucketClass.deletionPolicy`). If the backend refuses deletion because the bucket is not empty, delete objects first.