Menu

Storage

Introduction

Concepts

Guides

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

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. Under the hood, **CSI** (Container Storage Interface) drivers frequently handle the actual provisioning and attachment of storage. Let's briefly look at each component and then highlight the CSI Driver's role.

TOC

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 more usage, please refer to 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 more usage, please refer to 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 (e.g., Docker socket)
  - 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.

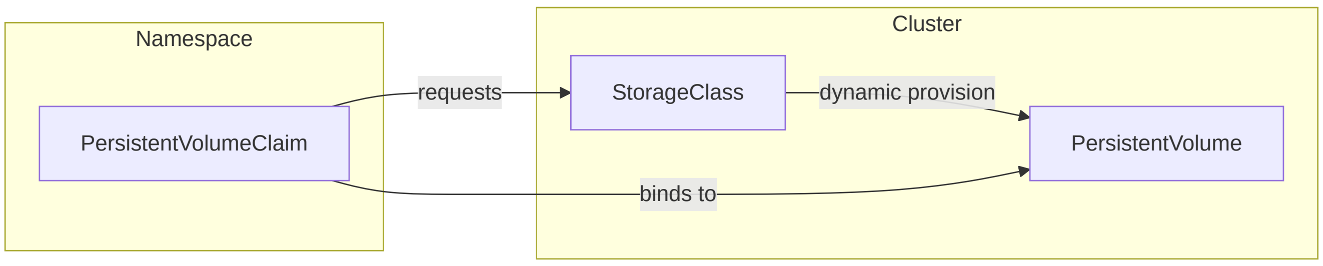

Diagram: Relationship between PVC, PV, and StorageClass.

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

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 compute 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 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
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.

Menu

Guides

Using Volume Snapshots

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

After clicking **Deploy**, on the **Distributed Storage** page, Create Storage Service or Access Storage Service.

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	Please allocate projects that can use this type of storage. If there are currently no projects that need to use this type of storage, you may choose not to allocate them for now and update 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

After clicking **Deploy**, on the **Distributed Storage** page, create a storage service or access a storage service.

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	Please assign projects that can use this type of storage. If there are no projects currently needing this type of storage, you can choose not to assign one 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

- 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

Please try to use local storage only for applications where data replication and backup at the application layer can be realized, such as MySQL. Avoid data loss due to the lack of data persistence guarantee from local storage.

Learn more ↗

Deploy Volume Plugin

After clicking deploy, on the newly opened page configure local 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.

Parameter	Description
	• 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.
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. • Please ensure that the number of storage nodes is greater than the number of application instance replicas, or it will affect PVC reconstruction.
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

- 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.

Deploying the Alauda Container Platform NFS CSI plugin

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

Refs to 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. Refer to the following instructions to configure the relevant parameters.

Parameter	Description
Name	The name of the storage class. It must be unique within the current cluster.
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	Please allocate the projects that can use this type of storage. If there are currently no projects needing this type of storage, you may choose not to allocate any projects at this time and update them 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

Parameter	Description
	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 refers to a snapshot of a persistent volume, which is a copy of the persistent volume at a specific point in time. If the cluster uses persistent volumes that support snapshot functionality, the volume snapshot component can be deployed to enable this feature.

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
CephRBD Block Storage	Supported	Not Supported	Not Supported
CephFS File Storage	Supported	**Supported**	Supported

TOC

Deploying via Web Console

1. Go to **Administrator**.
2. Click **Marketplace** > **Cluster Plugins** to access the **Cluster Plugins** list page.
3. Locate the `Alauda Container Platform Snapshot Management` cluster plugin, click **Install**, and wait for a moment until the deployment is successful.

Deploying via YAML

Refs to Installing via YAML

`Alauda Container Platform Snapshot Management` is a **Non-config plugin**, and the module-name is `snapshot`

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

- 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. Refer to the instructions below and configure the parameters before clicking **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.
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

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.
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

Parameter	Description
	selector to target PersistentVolumes that are labeled with specific tags. 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**: This helps to back up application data and restore it as needed, ensuring the reliability of business application data. Please refer to Using Volume Snapshots.

- **Delete PersistentVolumeClaim**: You can click the **Actions** button in the top right corner of the details page to delete the PersistentVolumeClaim as needed. Before deleting, please ensure that the PersistentVolumeClaim is not bound to any applications or workloads and that it does not contain any volume snapshots. After deleting the PersistentVolumeClaim, the platform will process the PersistentVolume according to the reclamation policy, which may clear data in the PersistentVolume and free storage resources. Please proceed with caution based on data security considerations.

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 may take some time, please be patient.

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

- 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.

```yaml
# 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**. The time taken for the snapshot depends on network conditions and data volume; please be patient.

   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**. The time taken for the snapshot depends on network conditions and data volume; please be patient.

   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**. The time taken for the snapshot depends on network conditions and data volume; please be patient.

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

How To

Configuring Persistent Storage Using NFS

Third-Party Storage Capability Annotation Guide

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

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
Dynamic provisioning	Backed by any CSI driver with dynamic provisioning

Feature	Description
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.

Feature	`emptyDir`	Generic Ephemeral Volume
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

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)

Configuring Persistent Storage Using Local volumes

Alauda Container Platform can be provisioned with persistent storage by using local volumes. Local persistent volumes allow you to access local storage devices, such as a disk or partition, by using the standard persistent volume claim interface.

Local volumes can be used without manually scheduling pods to nodes because the system is aware of the volume node constraints. However, local volumes are still subject to the availability of the underlying node and are not suitable for all applications.

NOTE

Local volumes can only be used as a statically created persistent volume.

TOC

Prerequisites

- **Download** the **Alauda Build of LocalStorage** installation package corresponding to your platform architecture.
- **Upload** the **Alauda Build of LocalStorage** installation package using the Upload Packages mechanism.
- You have a local disk that meets the following conditions:
    - It is attached to a node.
    - It is not mounted.
    - It does not contain partitions.

Procedure

1 Installing the Local Storage Operator

1. Login, go to the **Administrator** page.
2. Click **Marketplace** > **OperatorHub** to enter the **OperatorHub** page.
3. Find the **Alauda Build of LocalStorage**, click **Install**, and navigate to the **Install Alauda Build of LocalStorage** page.

   Configuration Parameters:

Parameter	Recommended Configuration
Channel	The default channel is `stable`.
Installation Mode	`Cluster`: All namespaces in the cluster share a single Operator instance for creation and management, resulting in lower resource usage.

Parameter	Recommended Configuration
Installation Place	Select `Recommended`, Namespace only support **acp-storage**.
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.

2 Provisioning local volumes by using the Local Storage Operator

Local volumes cannot be created by dynamic provisioning. Instead, persistent volumes can be created by the Local Storage Operator. The local volume provisioner looks for any file system or block volume devices at the paths specified in the defined resource.

1. Create the local volume resource. This resource must define the nodes and paths to the local volumes.

   NOTE

   Do not use different storage class names for the same device. Doing so will create multiple persistent volumes (PVs).

   Execute commands on the **control node** of the cluster.

```
cat << EOF | kubectl create -f -
apiVersion: "local.storage.openshift.io/v1"
kind: "LocalVolume"
metadata:
  name: "local-disks"
  namespace: "acp-storage" 1
spec:
  nodeSelector: 2
    nodeSelectorTerms:
    - matchExpressions:
        - key: kubernetes.io/hostname
          operator: In
          values:
          - worker-01
          - worker-02
          - worker-03
  storageClassDevices:
    - storageClassName: "local-sc" 3
      forceWipeDevicesAndDestroyAllData: false 4
      volumeMode: Filesystem 5
      fsType: xfs 6
      devicePaths: 7
        - /path/to/device 8
EOF
```

1. The namespace where the Local Storage Operator is installed, default is `acp-storage`.
2. Optional: A node selector containing a list of nodes where the local storage volumes are attached. This example uses the node hostnames, obtained from `kubectl get node`. If a value is not defined, then the Local Storage Operator will attempt to find matching disks on all available nodes.
3. The name of the storage class to use when creating persistent volume objects. The Local Storage Operator automatically creates the storage class if it does not exist. Be sure to use a storage class that uniquely identifies this set of local volumes.
4. Controls whether the operator wipes the listed devices before use. The default is "false". WARNING: Setting `forceWipeDevicesAndDestroyAllData: true` is destructive and will erase existing partition tables/filesystem signatures and data on those devices. Use only when you are certain the devices can be safely wiped.

5. The volume mode, either `Filesystem` or `Block`, that defines the type of local volumes.
6. Optional: The file system that is created when the local volume is mounted for the first time. This parameter must be configured only if `volumeMode` set to `Filesystem`.
7. The path containing a list of local storage devices to choose from.
8. Replace this value with your actual local disks filepath to the `LocalVolume` resource `by-id`, such as `/dev/disk/by-id/wwn`. PVs are created for these local disks when the provisioner is deployed successfully.

2. Verify that the persistent volumes were created:

   Execute commands on the **control node** of the cluster.

   ```
   kubectl get pv
   ```

   Example output:

   ```
   NAME                CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS      CLAIM
   STORAGECLASS   REASON   AGE
   local-pv-n8xlrd2a   100Gi      RWO            Delete           Available
   local-sc                88m
   local-pv-tc78vc73   100Gi      RWO            Delete           Available
   local-sc                82m
   local-pv-q86px4df   100Gi      RWO            Delete           Available
   local-sc                48m
   ```

   NOTE

   Editing the LocalVolume object does not change the fsType or volumeMode of existing persistent volumes because doing so might result in a destructive operation.

3. Creating the local volume persistent volume claim

   Execute commands on the **control node** of the cluster.

```
cat << EOF | kubectl create -f -
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: local-pvc-name ①
spec:
  accessModes:
  - ReadWriteOnce
  volumeMode: Filesystem ②
  resources:
    requests:
      storage: 100Gi ③
  storageClassName: local-sc ④
EOF
```

1. Name of the PVC.
2. The type of the PVC. Defaults to `Filesystem`.
3. The amount of storage available to the PVC.
4. Name of the storage class required by the claim.

Automating discovery local storage devices

The Local Storage Operator automates local storage discovery.

Prerequisites

- You have installed the Local Storage Operator.

Procedure

1. Create LocalVolumeDiscovery object

   Execute commands on the **control node** of the cluster.

```
cat << EOF | kubectl create -f -
apiVersion: local.storage.openshift.io/v1alpha1
kind: LocalVolumeDiscovery
metadata:
  name: auto-discover-devices
  namespace: acp-storage
spec:
  nodeSelector: 1
    nodeSelectorTerms:
      - matchExpressions:
          - key: kubernetes.io/hostname
            operator: In
            values:
              - worker-01
              - worker-02
              - worker-03
  tolerations: 2
    - operator: Exists
EOF
```

1. Optional: Nodes on which the automatic detection policies must run. If a value is not defined, then the Local Storage Operator will attempt to automatic detection on all available nodes.
2. Optional: If specified tolerations is the list of toleration that is passed to the LocalVolumeDiscovery Daemon

2. Verify discover result

Execute commands on the **control node** of the cluster.

```
kubectl -n acp-storage get localvolumediscoveryresult
```

Example output:

```
NAME                              AGE
discovery-result-worker-01        21m
discovery-result-worker-02        21m
discovery-result-worker-03        21m
```

A dedicated LocalVolumeDiscoveryResult object is generated for selected nodes in LocalVolumeDiscovery Object, from which you can inspect the block devices discovered on that node.

Configuring Persistent Storage Using NFS

Alauda Container Platform clusters support persistent storage using NFS. Persistent Volumes (PVs) and Persistent Volume Claims (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

- Storage must exist in the underlying infrastructure before it can be mounted as a volume in Alauda Container Platform.
- 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    STATUS    CLAIM
STORAGECLASS    REASON    AGE
pv-nfs-example    1Gi         RWO             Retain            Available
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 MODES   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 Alauda Container Platform mandates unique PV names, it remains the administrator's responsibility to ensure the uniqueness of the NFS volume's server and path for each export.

This partitioned approach enables precise capacity management. Developers request persistent storage specifying a required amount (e.g., 10Gi), and ACP matches the request to a PV backed by a partition/export offering at least that specified capacity. Please note: The quota enforcement applies to the usable storage space within the assigned partition/export.

NFS volume security

This section details NFS volume security mechanisms with a focus on permission matching. Readers are assumed to possess fundamental knowledge of POSIX permissions, process UIDs, and supplemental groups.

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 Alauda Container Platform's 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 Alauda Container Platform 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:

```yaml
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:

```yaml
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.

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

> **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

```yaml
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

Menu

Troubleshooting

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

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.

Menu

Object Storage

Introduction

Concepts

Installing

Guides

Creating a BucketClass for Ceph RGW

Creating a BucketClass for MinIO

Create a Bucket Request

How To

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

Introduction

The Container Object Storage Interface (COSI) is a Kubernetes-native framework designed to provide a standardized and declarative approach for managing object storage services, such as AWS S3, MinIO, and Ceph RGW, within Kubernetes clusters. COSI extends Kubernetes' storage model to support object storage resources in a way that is portable, scalable, and consistent with the principles of Kubernetes.

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

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

Concepts

TOC

Overview

This document introduces Kubernetes administrators familiar with persistent storage concepts to the core resources and principles of the Container Object Storage Interface (COSI). COSI provides a declarative mechanism for managing object storage (such as AWS S3, MinIO, and Ceph RGW), similar to existing Kubernetes persistent storage management approaches.

We will cover the three primary resources in COSI—**BucketClass, Bucket, and BucketClaim**—drawing analogies with Kubernetes storage resources to clarify their relationships and functionalities.

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:

```yaml
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 an abstraction of an actual bucket present in an external object storage system (such as AWS S3, MinIO, Ceph RGW) within Kubernetes.

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

1. **Download** the **Alauda Container Platform COSI** cluster plugin package corresponding to your platform architecture.
2. **Download** either the **Alauda Container Platform COSI for Ceph** or the **Alauda Container Platform COSI for MinIO** cluster plugin package, depending on which object storage solution you plan to use.
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 to get instructions on how to publish the cluster plugin to the cluster which you want to use. For more details, please refer to CLI.

Installing 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 for Ceph** and **Alauda Container Platform COSI for MinIO** both depend 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 for Ceph** or **Alauda Container Platform COSI for MinIO** plugin, based on your chosen backend.
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 for Ceph** and/or **Alauda Container Platform COSI for MinIO**.

Guides

Creating a BucketClass for Ceph RGW

Ceph Object Storage can be exposed to Kubernetes workloads via the **Container Object Storage Interface (COSI)**, providing highly scalable and elastic storage for big-data analytics, backup & 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

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 for Ceph** 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 for Ceph**

> Refer to *Installing* for exact commands.

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:

```yaml
# 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:

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

Verification & Next Steps

Verify the BucketClass:

```
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.

Creating a BucketClass for MinIO

MinIO integrates with Kubernetes through the **Container Object Storage Interface (COSI)** to provide scalable, S3-compatible object storage for analytics, backup & restore, and ML/AI workloads. Before provisioning buckets, define a *BucketClass*.

A **BucketClass** is a template resource that sets the storage driver, the authentication Secret, and the deletion policy applied to all buckets created from it.

TOC

Prerequisites

Requirement	Notes
MinIO cluster ready for use	Prepare MinIO by following the installation guide.

Requirement	Notes
Alauda Container Platform COSI plug-ins	**acp-cosi** and **acp-cosi-minio** must be installed. See Installing COSI Plug-ins for installation steps.
Kubernetes Secret containing MinIO credentials	Prepared in **Step 2**.

Step 1 - Prepare a MinIO Cluster

Ensure that a MinIO cluster is installed and accessible. Follow the MinIO installation documentation to deploy and configure your MinIO environment.

Step 2 - Prepare the Credential Secret

COSI retrieves MinIO credentials from a Kubernetes **Secret**. Collect the following values:

- `Endpoint` - e.g. `http://minio.minio-system.svc` or `https://minio.example.com:9000`
- `AccessKey`
- `SecretKey`

Create the Secret in the target namespace and label it for UI discovery:

```
kubectl create secret generic minio-credentials -n <YOUR_NAMESPACE> \
  --from-literal=Endpoint=http://<YOUR_MINIO_ENDPOINT> \
  --from-literal=AccessKey=<YOUR_ACCESS_KEY> \
  --from-literal=SecretKey=<YOUR_SECRET_KEY>

kubectl label secret minio-credentials -n <YOUR_NAMESPACE> app=minio
```

Important: The label `app=minio` is required for the platform UI to list the Secret.

Note: Key names are **case-sensitive** and must be exactly `Endpoint`, `AccessKey`, and `SecretKey`.

If you prefer GitOps, you can define the Secret declaratively:

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: minio-credentials
  namespace: <YOUR_NAMESPACE>
  labels:
    app: minio
type: Opaque
stringData:
  Endpoint: http://<YOUR_MINIO_ENDPOINT>
  AccessKey: <YOUR_ACCESS_KEY>
  SecretKey: <YOUR_SECRET_KEY>
```

Step 3 - Create the BucketClass

Option 1 - UI Workflow

1. Navigate to **Storage** → **Object StorageClass** and click **Create Object StorageClass**.
2. Select **MinIO 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** - Choose the Secret created in Step 2 (only Secrets with `app=minio` are shown).
   - **Allocate Projects** - Optional: restrict usage to specific projects.
4. Click **Create**.

Option 2 - YAML (GitOps-friendly)

Create `minio-bucketclass.yaml`. The example below uses the MinIO COSI driver and points to a Secret with the correct Secret references.

```yaml
apiVersion: objectstorage.k8s.io/v1alpha1
kind: BucketClass
driverName: minio.objectstorage.k8s.io
metadata:
  labels:
    project.cpaas.io/name: null
    project.cpaas.io/ALL_ALL: "true"
  name: minio-bucket-class
  annotations:
    cpaas.io/display-name: BucketClass for MinIO
    cpaas.io/access-mode: ""
    cpaas.io/features: ""
parameters:
  providerSecretName: <your-secret-name>
  providerSecretNamespace: <your-secret-namespace>
deletionPolicy: Delete
```

Apply the manifest:

```
kubectl apply -f minio-bucketclass.yaml
```

Verification and Next Steps

Verify the BucketClass:

```
kubectl get bucketclass
```

Once the BucketClass is ready, create **Bucket** or **BucketClaim** resources referencing it to provision S3-compatible object storage backed by MinIO.

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

- **Install** the **Alauda Container Platform COSI** cluster.
- **Install** either the **Alauda Container Platform COSI for Ceph** or the **Alauda Container Platform COSI for MinIO** cluster plugin package, depending on which object storage solution you plan to use.

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.

Menu

How To

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

This guide shows Kubernetes administrators how to 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

- 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
```

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:

```yaml
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` .
- **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`).