Alauda Container Platform

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 Mo

- Access Modes
- Volume Modes
- Storage Featur
- Conclusion

Guides

How To

Troubleshooting

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

Access Mod

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.

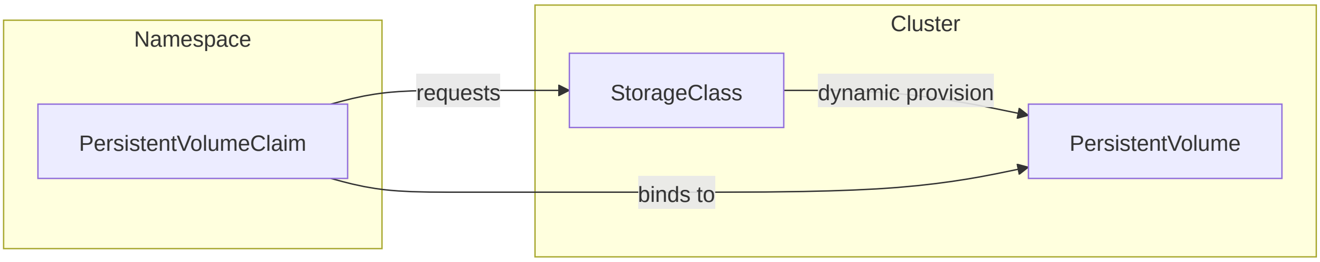

Diagram: Relationship between PVC, PV, and StorageClass.

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.

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

Procedure

Creating an NFS Shared Storage Class

Related Operations

Creating a F

Prerequisites

Example Persis

/ by

/ by

erat

esou

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 CLI

Creating Persistent Volume Claims from Volume Snapshots

Additional resource

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 **Platform Management**.
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 **Platform Management**.

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 **Platform Management**.

2. In the left navigation bar, click **Storage Management** > **Storage Classes**.

3. Click **Create Storage Class**.

4. Select **TopoLVM**, then click **Next**.

5. 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.

Parameter	Description
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.
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**.

6. 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 NFS Shared Storage Plugin

1. Enter **Platform Management**.

2. In the left navigation bar, click **Storage Management** > **Storage Classes**.

3. Click **Create Storage Class**.

4. On the right side of **NFS Shared Storage**, click Deploy to navigate to the **Plugins** page.

5. On the right side of the **NFS** plugin, click ⋮ > **Deploy**.

6. Wait for the deployment status to indicate **Deployment Successful** before completing the deployment.

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 Shared Storage** 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`.

Parameter	Description
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.

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

Related Operations

Setting the naming rules for subdirectories in the NFS Shared Storage Class

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

Procedure

1. Go to **Platform Management**.

2. In the left navigation bar, click on **Storage Management** > **Volume Snapshots**.

3. Click on **Quick Deployment**, which will redirect you to the cluster plugin.

4. Click on the ⋮ > **Deploy** next to **Snapshot Controller**, and wait for a moment until the deployment is successful.

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 **Platform Management**.

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

Parameter	Description
	- Selector: After selecting the static binding method, you can use a 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

Setting the naming rules for subdirectories in the NFS Shared Storage Class

TOC

Feature Overview

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. This document provides configuration methods and best practices for customizing the naming convention.

Use Cases

On the NFS Server side, the subdirectory names can be used to identify their corresponding PersistentVolumeClaims (PVCs) in Kubernetes. This allows administrators to monitor the storage usage of each PVC, thereby simplifying operational management.

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.

Procedure

1 Deploying the NFS Shared Storage Plugin

Refers to Deploying the NFS Shared Storage Plugin.

2 Creating an NFS Shared Storage Class

1. Refers to Creating an NFS Shared Storage Class.

2. Before clicking **Create**, switch to the YAML view and add the subDir configuration under the parameters section to define the naming rules for subdirectories.

   Configuration Example

   ```
   parameters:
     subDir: ${pvc.metadata.namespace}_${pvc.metadata.name}_${pv.metadata.name}
   ```

   Subdirectory Naming Examples

   ```
   default_nfs-pvc-01_pvc-4411db0b-8ec4-461a-8bbd-062d50666249
   ```

`default` is PVC Namespace, `nfs-pvc-01` is PVC Name, `pvc-4411db0b-8ec4-461a-8bbd-062d50666249` is PV Name.

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.
>
> 3. click **Create**.

INFO

Existing StorageClass cannot be modified.

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
No separate PVC	VolumeClaim is embedded directly in the Pod spec

Feature	Description
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

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)

How to Annotate Third-Party Storage Capabilities

With the growing use of both public and private cloud environments, third-party storage integration has become increasingly important. This guide walks you through how to annotate third-party storage capabilities using a ConfigMap so your platform can recognize and surface those capabilities automatically.

TOC

Step 1: Open Storage Class Configuration

1. Go to **Platform Management** in your platform's UI.
2. From the left sidebar, navigate to **Storage Management** > **Storage Classes**.
3. Click **Create Storage Class** to start defining a new storage class.

Step 2: Fill in Storage Class Information

Provide the following details in the form:

Field	Description
Name	Name of your new storage class.
Storage Class	Select or define the storage class identifier.
Provisioner	Enter the provisioner name used by your storage plugin.

Step 3: Annotate Storage Capabilities with a ConfigMap

To enable capability annotations, create a **ConfigMap** in the `kube-public` namespace with the appropriate label and data format.

Example YAML:

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: sd-built-in
  namespace: kube-public
  labels:
    features.alauda.io/type: StorageDescription
data:
  storage.type1.com: |-
    type: Filesystem
    volumeMode:
    - Filesystem
    accessModes:
    - ReadWriteOnce
    - ReadWriteMany
    - ReadWriteOncePod
  storage.type2.com: |-
    type: Filesystem
    snapshot: true
    volumeMode:
    - Filesystem
    - Block
    accessModes:
    - ReadWriteOnce
    - ReadOnlyMany
    - ReadWriteOncePod
```

Key Points:

- **metadata.name**: Must begin with `sd-`, e.g., `sd-configmap1`.
- **metadata.namespace**: Must be `kube-public`.
- **metadata.labels**: Include `features.alauda.io/type = StorageDescription`.
- **data**:
  - Each **key** corresponds to the `provisioner` field in the storage class.
  - Each **value** is a YAML string describing the storage's supported capabilities.

Step 4: Understand Supported Storage Capability Fields

Here are the supported fields you can define in the ConfigMap:

Capability	Field	Options	Default	Notes
Type	`type`	`Filesystem` , `Block`	—	If omitted or inval the type shows a unknown.
Snapshot	`snapshot`	`true` , `false`	`false`	If `false` or inva snapshot creatio via the form UI is disabled.
Volume Mode	`volumeMode`	`Filesystem` , `Block`	`Filesystem`	PVCs with `Bloc` mode don't supp directory mounts.
Access Mode	`accessModes`	`ReadWriteOnce` , `ReadOnlyMany` , `ReadWriteMany` , `ReadWriteOncePod`	—	If omitted or inval no access mode selectable via the `ReadWriteOnceI` isn't currently forı enabled.

Step 5: Finalize the Storage Class

Once the above details are set:

1. Click **Create** to save your storage class.
2. The platform will automatically match the `provisioner` with the ConfigMap and annotate the storage class with the defined capabilities.

Optional: Create a PVC Using the Annotated Storage Class

When you create a Persistent Volume Claim (PVC) via the **form UI**, only the supported capabilities from the annotated ConfigMap will be available. Unsupported options will not appear.

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.