



Provision and manage volumes

Trident

NetApp
October 30, 2026

Table of Contents

Provision and manage volumes	1
Provision a Trident volume	1
Overview	1
Create the PVC	1
Move a volume	3
Requirements	4
How it works	4
Move a volume	4
Expand Trident volumes	7
Expand an iSCSI volume	7
Expand an FC volume	10
Expand an NFS volume	14
Understand RWX NVMe subsystem limits	17
Understand the 64-node limit	17
Understand NVMe subsystem models	17
Identify error symptoms	17
Resolve subsystem limit errors	17
Upgrade Trident to apply the super-subsystem model	18
Controller scalability	18
Key concepts and definitions	18
Default behavior (serial mode)	18
Controller scalability support	18
How enableConcurrency works	19
Before you enable	19
Enable controller scalability	20
Concurrency behavior	22
Caveats and limitations	22
Recommendations	22
Automatic volume expansion	23
Requirements	23
Limitations	23
Provision Volumes with Autogrow Policy	24
Create an Autogrow Policy	24
Create an Autogrow Policy	25
Policy states	25
Associate a policy with a StorageClass	25
Policy precedence	26
Configuration examples	27
Manage Autogrow Policies	29
View Autogrow Policies	29
Update an Autogrow Policy	30
Delete an Autogrow Policy	31
Monitor Autogrow Policy usage	31

Supported protocols	32
Supported backends	32
Known limitations	32
Frequently asked questions	32
Import Trident volumes	38
Overview and considerations	38
Import a volume	39
Examples	41
ONTAP SAN-economy examples	46
Customize Trident volume names and labels	49
Before you begin	49
Limitations	50
Key behaviors of customizable volume names	50
Backend configuration examples with name template and labels	50
Name template examples	51
Points to consider	51
Share a Trident NFS volume across namespaces	52
Features	52
Quick start	53
Configure the source and destination namespaces	53
Delete a shared volume	55
Use <code>tridentctl get</code> to query subordinate volumes	55
Limitations	55
For more information	55
Clone Trident volumes across namespaces	55
Prerequisites	55
Quick start	56
Configure the source and destination namespaces	56
Limitations	58
Replicate volumes using SnapMirror	58
Replication prerequisites	58
Create a mirrored PVC	59
Volume Replication States	62
Promote secondary PVC during an unplanned failover	62
Promote secondary PVC during a planned failover	62
Restore a mirror relationship after a failover	63
Additional operations	63
Update mirror relationships when ONTAP is online	64
Update mirror relationships when ONTAP is offline	64
Use CSI Topology	64
Overview	64
Step 1: Create a topology-aware backend	66
Step 2: Define StorageClasses that are topology aware	67
Step 3: Create and use a PVC	68
Update backends to include <code>supportedTopologies</code>	70

Find more information	71
Work with Trident snapshots	71
Overview	71
Create a volume snapshot	71
Create a PVC from a volume snapshot	73
Import a volume snapshot	73
Recover volume data using snapshots	75
In-place volume restoration from a snapshot	76
Delete a PV with associated snapshots	77
Deploy a volume snapshot controller	77
Related links	78
Work with Trident volume group snapshots	78
Create volume group snapshots	79
Recover volume data using a group snapshot	80
In-place volume restoration from a snapshot	81
Delete a PV with associated group snapshots	81
Deploy a volume snapshot controller	81
Related links	82

Provision and manage volumes

Provision a Trident volume

Create a PersistentVolumeClaim (PVC) that uses the configured Kubernetes StorageClass to request access to the PV. You can then mount the PV to a pod.

Overview

A [PersistentVolumeClaim](#) (PVC) is a request for access to the PersistentVolume on the cluster.

The PVC can be configured to request storage of a certain size or access mode. Using the associated StorageClass, the cluster administrator can control more than PersistentVolume size and access mode—such as performance or service level.

After you create the PVC you can mount the volume in a pod.

Create the PVC

Steps

1. Create the PVC.

```
kubectl create -f pvc.yaml
```

2. Verify the PVC status.

```
kubectl get pvc
```

NAME	STATUS	VOLUME	CAPACITY	ACCESS MODES	STORAGECLASS	AGE
pvc-storage	Bound	pv-name	1Gi	RWO		5m

1. Mount the volume in a pod.

```
kubectl create -f pv-pod.yaml
```



You can monitor the progress using `kubectl get pod --watch`.

2. Verify that the volume is mounted on `/my/mount/path`.

```
kubectl exec -it task-pv-pod -- df -h /my/mount/path
```

3. You can now delete the Pod. The Pod application will no longer exist, but the volume will remain.

```
kubectl delete pod pv-pod
```

Sample manifests

▼ PersistentVolumeClaim sample manifests

These examples show basic PVC configuration options.

PVC with RWO access

This example shows a basic PVC with RWO access that is associated with a StorageClass named `basic-csi`.

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: pvc-storage
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
  storageClassName: basic-csi
```

PVC with NVMe/TCP

This example shows a basic PVC for NVMe/TCP with RWO access that is associated with a StorageClass named `protection-gold`.

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: pvc-san-nvme
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 300Mi
  storageClassName: protection-gold
```

▼ Pod manifest samples

These examples show basic configurations to attach the PVC to a pod.

Basic configuration

```
kind: Pod
apiVersion: v1
metadata:
  name: pv-pod
spec:
  volumes:
    - name: storage
      persistentVolumeClaim:
        claimName: pvc-storage
  containers:
    - name: pv-container
      image: nginx
      ports:
        - containerPort: 80
          name: "http-server"
      volumeMounts:
        - mountPath: "/my/mount/path"
          name: storage
```

Basic NVMe/TCP configuration

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-nginx
spec:
  volumes:
    - name: basic-pvc
      persistentVolumeClaim:
        claimName: pvc-san-nvme
  containers:
    - name: task-pv-container
      image: nginx
      volumeMounts:
        - mountPath: "/my/mount/path"
          name: basic-pvc
```

Refer to [Kubernetes and Trident objects](#) for details on how storage classes interact with the PersistentVolumeClaim and parameters for controlling how Trident provisions volumes.

Move a volume

Beginning with Trident 26.06, a storage administrator can non-disruptively move an iSCSI

volume to a different ONTAP node and aggregate within the same ONTAP cluster while the workload that uses it keeps running. You request the move declaratively with a `TridentVolumeMove` custom resource or the `tridentctl` equivalent, and Trident coordinates ONTAP and the host so that I/O is never interrupted.

Requirements

Volume move applies to the `ontap-san` driver for iSCSI volumes only. NVMe and NAS protocols are not supported.

Before you move a volume, ensure that the following requirements are met:

- The backend uses the `ontap-san` driver with the ONTAP REST API.
- The ONTAP cluster has at least two nodes.
- You have one of the following sets of credentials:
 - ONTAP 9.19.1 or later if using the dedicated custom ONTAP role.
 - Cluster-admin credentials on earlier ONTAP releases.

How it works

Trident moves the volume (LUN) between two locations inside the same ONTAP cluster. Before the data moves, Trident adds new iSCSI paths to the target alongside the existing paths, so that the host multipath device never loses access. After ONTAP completes the move, Trident prunes the stale source paths.

Trident drives the move through an ordered set of states that are visible in the `TridentVolumeMove` resource, so that you can follow the move progress.

Move a volume

Identify the volume and locations

1. Find the name of the PV to move. The PV name is also the name of the move resource.
2. From the ONTAP cluster, note the current (source) node and aggregate for that volume, and choose the destination (target) node and aggregate.

Start the move

Use either method. Both create the same `TridentVolumeMove` resource, whose name is the PV name.

Using `tridentctl`

```
tridentctl create volume-move \  
  --volume      pvc-8a814d62-bd58-4253-b0d1-82f2885db671 \  
  --source-node ontap-node-01 \  
  --source-pool aggr1_node01 \  
  --target-node ontap-node-02 \  
  --target-pool aggr1_node02 \  

```

```
-n trident
```

`tvm` is an alias for `volume-move`. Add `--delete-after-success 10m` to remove the resource automatically after the move succeeds.

Using a TridentVolumeMove resource

```
apiVersion: trident.netapp.io/v1
kind: TridentVolumeMove
metadata:
  name: pvc-8a814d62-bd58-4253-b0d1-82f2885db671    # the PV to move
  namespace: trident
spec:
  sourceNode: ontap-node-01
  sourcePool: aggr1_node01
  targetNode: ontap-node-02
  targetPool: aggr1_node02
  # deleteAfterSuccess: 10m    # optional
```

Apply the resource:

```
kubectl apply -f tridentvolumemove.yaml
```

Spec fields

sourceNode / targetNode

ONTAP node names (not UUIDs).

sourcePool / targetPool

ONTAP aggregate names.

deleteAfterSuccess

Optional duration, such as 30s or 10m. Set to 0 to delete the resource immediately, or omit it to retain the resource.

Monitor the move

Track progress using the `tvm` short name or `tridentvolumemove`:

```
kubectl get tridentvolumemove -n trident
```

To see the current state and per-node progress, describe the resource:

```
kubectl describe tridentvolumemove <pv-name> -n trident
```

The volume can have concurrent attachments during the `Moving` state, because ONTAP can take a while to complete the move for a large volume. To ensure a seamless move, Trident coordinates between the Trident controller and the Trident node. Each state performs the following steps in order, ending in either `Succeeded` or `Failed`.

State	What is happening
Pending	Trident accepts and validates the move request and gathers the information it needs before any storage changes begin.
ControllerStaging	Trident prepares the target location. It maps the target node's storage interfaces to the LUN and updates Kubernetes so that a future node restart reconnects to the correct location.
NodeStaging	On each host that has the volume mounted, Trident adds the new storage paths from the target node alongside the existing paths, so that the pod's I/O continues across both path sets.
Moving	ONTAP copies and cuts over the volume data to the target aggregate. This is the longest phase, and ONTAP handles it entirely while Trident polls until it finishes. New publishes are allowed during this state.
ControllerUnstaging	The data move is complete. Trident updates Kubernetes to drop the source paths from the desired state, and then unmaps the source node's storage interfaces from the LUN.
NodeUnstaging	On each host, Trident removes the stale source paths. The host is left with only the target paths, and the multipath device continues serving I/O without interruption.
Succeeded	The volume is fully moved and online at the target. Trident retains the <code>TridentVolumeMove</code> resource, or deletes it automatically if you set <code>deleteAfterSuccess</code> .
Failed	A non-recoverable error stopped the move or prevented it from starting. Trident resets the volume state so that normal operations can resume, and keeps the resource with an error message. Review the message, fix the root cause, delete the resource, and try again.

The `.status.attachments` list reports each attached node's progress in its `state` field:

Attachment state	Meaning
Pending	Node identified as attached; work has not yet started.
Bridged	New target paths added alongside the existing paths.
Migrated	Traffic is now served over the new paths.
Cleaned	Stale source paths removed from the node.
Detached	Node detached mid-move and excluded from further consideration while Trident assesses whether the attachment is ready to proceed to the next step.
Failed	Terminal error on this node. See the message.

Clean up the move resource

If you set `deleteAfterSuccess`, Trident removes the resource automatically. Otherwise, you can delete it manually. Before you request another move on the same volume, ensure that no `TridentVolumeMove` resource exists for that volume in either a `Succeeded` or `Failed` state.



Trident never deletes failed moves automatically. Trident keeps a `Failed` resource as an audit record. After you fix the root cause, delete the resource manually before you retry.

Expand Trident volumes

Trident provides Kubernetes users the ability to expand their volumes after they are created. Find information about the configurations required to expand iSCSI, NFS, SMB, NVMe/TCP, and FC volumes.

Expand an iSCSI volume

You can expand an iSCSI Persistent Volume (PV) by using the CSI provisioner.



iSCSI volume expansion is supported by the `ontap-san`, `ontap-san-economy`, `solidfire-san` drivers and requires Kubernetes 1.16 and later.

Step 1: Configure the StorageClass to support volume expansion

Edit the `StorageClass` definition to set the `allowVolumeExpansion` field to `true`.

```
cat storageclass-ontapsan.yaml
```

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: ontap-san
provisioner: csi.trident.netapp.io
parameters:
  backendType: "ontap-san"
allowVolumeExpansion: True
```

For an already existing `StorageClass`, edit it to include the `allowVolumeExpansion` parameter.

Step 2: Create a PVC with the StorageClass you created

Edit the PVC definition and update the `spec.resources.requests.storage` to reflect the newly desired size, which must be greater than the original size.

```
cat pvc-ontapsan.yaml
```

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: san-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
  storageClassName: ontap-san
```

Trident creates a Persistent Volume (PV) and associates it with this Persistent Volume Claim (PVC).

```
kubectl get pvc
NAME          STATUS    VOLUME                                     CAPACITY
ACCESS MODES  STORAGECLASS  AGE
san-pvc       Bound       pvc-8a814d62-bd58-4253-b0d1-82f2885db671  1Gi
RWO           ontap-san    8s

kubectl get pv
NAME          CAPACITY  ACCESS MODES
RECLAIM POLICY STATUS    CLAIM                                STORAGECLASS  REASON    AGE
pvc-8a814d62-bd58-4253-b0d1-82f2885db671  1Gi       RWO
Delete       Bound     default/san-pvc                     ontap-san    10s
```

Step 3: Define a pod that attaches the PVC

Attach the PV to a pod for it to be resized. There are two scenarios when resizing an iSCSI PV:

- If the PV is attached to a pod, Trident expands the volume on the storage backend, rescans the device, and resizes the filesystem.
- When attempting to resize an unattached PV, Trident expands the volume on the storage backend. After the PVC is bound to a pod, Trident rescans the device and resizes the filesystem. Kubernetes then updates the PVC size after the expand operation has successfully completed.

In this example, a pod is created that uses the `san-pvc`.

```
kubectl get pod
NAME          READY    STATUS    RESTARTS   AGE
ubuntu-pod    1/1     Running   0           65s
```

```
kubectl describe pvc san-pvc
Name:                san-pvc
Namespace:           default
StorageClass:        ontap-san
Status:              Bound
Volume:              pvc-8a814d62-bd58-4253-b0d1-82f2885db671
Labels:              <none>
Annotations:         pv.kubernetes.io/bind-completed: yes
                    pv.kubernetes.io/bound-by-controller: yes
                    volume.beta.kubernetes.io/storage-provisioner:
csi.trident.netapp.io
Finalizers:          [kubernetes.io/pvc-protection]
Capacity:            1Gi
Access Modes:        RWO
VolumeMode:          Filesystem
Mounted By:          ubuntu-pod
```

Step 4: Expand the PV

To resize the PV that has been created from 1Gi to 2Gi, edit the PVC definition and update the `spec.resources.requests.storage` to 2Gi.

```
kubectl edit pvc san-pvc
```

```
# Please edit the object below. Lines beginning with a '#' will be
ignored,
# and an empty file will abort the edit. If an error occurs while saving
this file will be
# reopened with the relevant failures.
#
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  annotations:
    pv.kubernetes.io/bind-completed: "yes"
    pv.kubernetes.io/bound-by-controller: "yes"
    volume.beta.kubernetes.io/storage-provisioner: csi.trident.netapp.io
  creationTimestamp: "2019-10-10T17:32:29Z"
  finalizers:
  - kubernetes.io/pvc-protection
  name: san-pvc
  namespace: default
  resourceVersion: "16609"
  selfLink: /api/v1/namespaces/default/persistentvolumeclaims/san-pvc
```

```
uid: 8a814d62-bd58-4253-b0d1-82f2885db671
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 2Gi
# ...
```

Step 5: Validate the expansion

You can validate the expansion worked correctly by checking the size of the PVC, PV, and the Trident volume:

```
kubectl get pvc san-pvc
NAME          STATUS    VOLUME                                     CAPACITY
ACCESS MODES  STORAGECLASS  AGE
san-pvc      Bound       pvc-8a814d62-bd58-4253-b0d1-82f2885db671  2Gi
RWO           ontap-san    11m

kubectl get pv
NAME          CAPACITY  ACCESS MODES
RECLAIM POLICY STATUS  CLAIM          STORAGECLASS  REASON  AGE
pvc-8a814d62-bd58-4253-b0d1-82f2885db671  2Gi      RWO
Delete          Bound    default/san-pvc  ontap-san    12m

tridentctl get volumes -n trident
+-----+-----+-----+-----+
+-----+-----+-----+-----+
|          NAME          | SIZE | STORAGE CLASS |
+-----+-----+-----+-----+
|          BACKEND UUID   | STATE | MANAGED |
+-----+-----+-----+-----+
+-----+-----+-----+-----+
| pvc-8a814d62-bd58-4253-b0d1-82f2885db671 | 2.0 GiB | ontap-san |
block      | a9b7bfff-0505-4e31-b6c5-59f492e02d33 | online | true      |
+-----+-----+-----+-----+
+-----+-----+-----+-----+
```

Expand an FC volume

You can expand an FC Persistent Volume (PV) by using the CSI provisioner.



FC volume expansion is supported by the `ontap-san` driver and requires Kubernetes 1.16 and later.

Step 1: Configure the StorageClass to support volume expansion

Edit the StorageClass definition to set the `allowVolumeExpansion` field to `true`.

```
cat storageclass-ontapsan.yaml
```

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: ontap-san
provisioner: csi.trident.netapp.io
parameters:
  backendType: "ontap-san"
allowVolumeExpansion: True
```

For an already existing StorageClass, edit it to include the `allowVolumeExpansion` parameter.

Step 2: Create a PVC with the StorageClass you created

Edit the PVC definition and update the `spec.resources.requests.storage` to reflect the newly desired size, which must be greater than the original size.

```
cat pvc-ontapsan.yaml
```

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: san-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
  storageClassName: ontap-san
```

Trident creates a Persistent Volume (PV) and associates it with this Persistent Volume Claim (PVC).

```
kubectl get pvc
NAME          STATUS    VOLUME                                     CAPACITY
ACCESS MODES  STORAGECLASS  AGE
san-pvc      Bound       pvc-8a814d62-bd58-4253-b0d1-82f2885db671  1Gi
RWX          ontap-san    8s

kubectl get pv
NAME          CAPACITY  ACCESS MODES
```

RECLAIM POLICY	STATUS	CLAIM	STORAGECLASS	REASON	AGE
pvc-8a814d62-bd58-4253-b0d1-82f2885db671	1Gi	RWO			
Delete	Bound	default/san-pvc	ontap-san		10s

Step 3: Define a pod that attaches the PVC

Attach the PV to a pod for it to be resized. There are two scenarios when resizing an FC PV:

- If the PV is attached to a pod, Trident expands the volume on the storage backend, rescans the device, and resizes the filesystem.
- When attempting to resize an unattached PV, Trident expands the volume on the storage backend. After the PVC is bound to a pod, Trident rescans the device and resizes the filesystem. Kubernetes then updates the PVC size after the expand operation has successfully completed.

In this example, a pod is created that uses the `san-pvc`.

```
kubectl get pod
NAME          READY   STATUS    RESTARTS   AGE
ubuntu-pod    1/1     Running   0           65s

kubectl describe pvc san-pvc
Name:          san-pvc
Namespace:     default
StorageClass:  ontap-san
Status:        Bound
Volume:        pvc-8a814d62-bd58-4253-b0d1-82f2885db671
Labels:        <none>
Annotations:   pv.kubernetes.io/bind-completed: yes
               pv.kubernetes.io/bound-by-controller: yes
               volume.beta.kubernetes.io/storage-provisioner:
               csi.trident.netapp.io
Finalizers:    [kubernetes.io/pvc-protection]
Capacity:      1Gi
Access Modes:  RWO
VolumeMode:    Filesystem
Mounted By:    ubuntu-pod
```

Step 4: Expand the PV

To resize the PV that has been created from 1Gi to 2Gi, edit the PVC definition and update the `spec.resources.requests.storage` to 2Gi.

```
kubectl edit pvc san-pvc
```

```
# Please edit the object below. Lines beginning with a '#' will be
ignored,
# and an empty file will abort the edit. If an error occurs while saving
this file will be
# reopened with the relevant failures.
#
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  annotations:
    pv.kubernetes.io/bind-completed: "yes"
    pv.kubernetes.io/bound-by-controller: "yes"
    volume.beta.kubernetes.io/storage-provisioner: csi.trident.netapp.io
  creationTimestamp: "2019-10-10T17:32:29Z"
  finalizers:
    - kubernetes.io/pvc-protection
  name: san-pvc
  namespace: default
  resourceVersion: "16609"
  selfLink: /api/v1/namespaces/default/persistentvolumeclaims/san-pvc
  uid: 8a814d62-bd58-4253-b0d1-82f2885db671
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 2Gi
# ...
```

Step 5: Validate the expansion

You can validate the expansion worked correctly by checking the size of the PVC, PV, and the Trident volume:

```
kubectl get pvc san-pvc
```

NAME	STATUS	VOLUME	CAPACITY
ACCESS MODES STORAGECLASS AGE			
san-pvc	Bound	pvc-8a814d62-bd58-4253-b0d1-82f2885db671	2Gi
RWO		ontap-san	11m

```
kubectl get pv
```

NAME	CAPACITY	ACCESS MODES			
RECLAIM POLICY STATUS CLAIM STORAGECLASS REASON AGE					
pvc-8a814d62-bd58-4253-b0d1-82f2885db671	2Gi	RWO			
Delete		Bound	default/san-pvc	ontap-san	12m

```
tridentctl get volumes -n trident
```

```
+-----+-----+
```

PROTOCOL	NAME	SIZE	STORAGE CLASS
BACKEND UUID	STATE	MANAGED	
pvc-8a814d62-bd58-4253-b0d1-82f2885db671	2.0 GiB	ontap-san	
block a9b7bfff-0505-4e31-b6c5-59f492e02d33	online	true	

Expand an NFS volume

Trident supports volume expansion for NFS PVs provisioned on `ontap-nas`, `ontap-nas-economy`, `ontap-nas-flexgroup`, and `azure-netapp-files` backends.

Step 1: Configure the StorageClass to support volume expansion

To resize an NFS PV, the admin first needs to configure the storage class to allow volume expansion by setting the `allowVolumeExpansion` field to `true`:

```
cat storageclass-ontapnas.yaml
```

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: ontapnas
provisioner: csi.trident.netapp.io
parameters:
  backendType: ontap-nas
allowVolumeExpansion: true
```

If you have already created a storage class without this option, you can simply edit the existing storage class by using `kubectl edit storageclass` to allow volume expansion.

Step 2: Create a PVC with the StorageClass you created

```
cat pvc-ontapnas.yaml
```

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: ontapnas20mb
spec:
```

```

accessModes:
- ReadWriteOnce
resources:
  requests:
    storage: 20Mi
storageClassName: ontapnas

```

Trident should create a 20 MiB NFS PV for this PVC:

```

kubectl get pvc
NAME          STATUS    VOLUME
CAPACITY      ACCESS MODES  STORAGECLASS  AGE
ontapnas20mb  Bound       pvc-08f3d561-b199-11e9-8d9f-5254004dfdb7  20Mi
RWO           ontapnas      9s

kubectl get pv pvc-08f3d561-b199-11e9-8d9f-5254004dfdb7
NAME          CAPACITY  ACCESS MODES
RECLAIM POLICY STATUS    CLAIM          STORAGECLASS  REASON
AGE
pvc-08f3d561-b199-11e9-8d9f-5254004dfdb7  20Mi      RWO
Delete          Bound     default/ontapnas20mb  ontapnas
2m42s

```

Step 3: Expand the PV

To resize the newly created 20 MiB PV to 1 GiB, edit the PVC and set `spec.resources.requests.storage` to 1 GiB:

```

kubectl edit pvc ontapnas20mb

```

```

# Please edit the object below. Lines beginning with a '#' will be
# ignored,
# and an empty file will abort the edit. If an error occurs while saving
# this file will be
# reopened with the relevant failures.
#
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  annotations:
    pv.kubernetes.io/bind-completed: "yes"
    pv.kubernetes.io/bound-by-controller: "yes"
    volume.beta.kubernetes.io/storage-provisioner: csi.trident.netapp.io

```

```

creationTimestamp: 2018-08-21T18:26:44Z
finalizers:
- kubernetes.io/pvc-protection
name: ontapnas20mb
namespace: default
resourceVersion: "1958015"
selfLink: /api/v1/namespaces/default/persistentvolumeclaims/ontapnas20mb
uid: c1bd7fa5-a56f-11e8-b8d7-fa163e59eaab
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
# ...

```

Step 4: Validate the expansion

You can validate the resize worked correctly by checking the size of the PVC, PV, and the Trident volume:

```

kubectl get pvc ontapnas20mb
NAME          STATUS    VOLUME
CAPACITY     ACCESS MODES  STORAGECLASS  AGE
ontapnas20mb  Bound       pvc-08f3d561-b199-11e9-8d9f-5254004dfdb7  1Gi
RWO          ontapnas      4m44s

kubectl get pv pvc-08f3d561-b199-11e9-8d9f-5254004dfdb7
NAME          CAPACITY  ACCESS MODES
RECLAIM POLICY STATUS    CLAIM          STORAGECLASS  REASON
AGE
pvc-08f3d561-b199-11e9-8d9f-5254004dfdb7  1Gi      RWO
Delete      Bound     default/ontapnas20mb  ontapnas
5m35s

tridentctl get volume pvc-08f3d561-b199-11e9-8d9f-5254004dfdb7 -n trident
+-----+-----+-----+-----+
+-----+-----+-----+-----+
|          NAME          | SIZE  | STORAGE CLASS |
+-----+-----+-----+-----+
|          BACKEND UUID  | STATE | MANAGED  |
+-----+-----+-----+-----+
| pvc-08f3d561-b199-11e9-8d9f-5254004dfdb7 | 1.0 GiB | ontapnas      |
| file          | c5a6f6a4-b052-423b-80d4-8fb491a14a22 | online | true      |
+-----+-----+-----+-----+
+-----+-----+-----+-----+

```

Understand RWX NVMe subsystem limits

ReadWriteMany (RWX) volumes that use the NVMe protocol have a scalability limit of 64 nodes per volume. The following includes the limitations, explains the NVMe subsystem architecture involved, and outlines the required resolution steps.

Understand the 64-node limit

If you plan to use ReadWriteMany (RWX) volumes with the NVMe protocol, a single RWX NVMe volume cannot be mounted by more than 64 nodes in a Kubernetes cluster.

Do not schedule workloads that mount the same RWX NVMe PersistentVolumeClaim across more than 64 nodes.

This limitation applies only to RWX volumes that use the NVMe protocol.

Understand NVMe subsystem models

Per-volume subsystem model (Trident releases earlier than 26.02)

In Trident releases earlier than 26.02, RWX NVMe volumes are provisioned using a per-volume subsystem model.

Each RWX NVMe volume is mapped to its own dedicated NVMe subsystem on ONTAP.

This model is simple, but it has a lower scalability limit.

In large Kubernetes clusters, subsystem controller limits are reached quickly because each RWX volume consumes a dedicated subsystem.

Super-subsystem model (introduced in Trident 26.02)

Starting with Trident 26.02, RWX NVMe volumes use a shared super-subsystem model.

Multiple RWX NVMe volumes share the same NVMe subsystem.

Each super-subsystem supports up to 1024 namespaces (volumes).

This model significantly improves scalability for RWX workloads and reduces the likelihood of reaching ONTAP subsystem limits.

Each RWX NVMe volume supports up to 64 nodes.

Identify error symptoms

If you create or attach RWX NVMe volumes at scale, you might observe errors similar to the following:

```
Maximum number of controllers reached. No more controllers can be created.
```

This error indicates that the ONTAP NVMe subsystem controller limit has been reached.

Resolve subsystem limit errors

To move beyond per-volume subsystem limitations and take advantage of the super-subsystem model, upgrade to Trident 26.02 or later.

Upgrade Trident to apply the super-subsystem model

To apply the super-subsystem model for RWX NVMe volumes:

1. Upgrade Trident to version 26.02 or later.
2. Scale down all pods that use RWX NVMe volumes to zero replicas.
3. Verify that no workloads are actively using RWX NVMe volumes.
4. Scale the pods back up.

This restart sequence ensures that RWX NVMe volumes are attached using the super-subsystem model.

- This limitation applies only to RWX volumes that use the NVMe protocol.
- The 64-node limit applies per RWX NVMe volume.
- Other access modes and other protocols are not affected.

Controller scalability

Trident improves controller scalability through increased concurrency across multiple storage drivers. When you enable controller scalability, the Trident controller processes storage operations in parallel instead of serializing them, which increases throughput in Kubernetes environments with many concurrent operations.

Before you deploy, determine which Trident drivers support controller scalability at general availability and which drivers are available as a technical preview in Trident 26.06. This helps you make informed deployment decisions and manage risk. Controller scalability is disabled by default.

Key concepts and definitions

Controller scalability

Controller scalability refers to the Trident controller's ability to process multiple storage operations in parallel rather than serializing them behind a single lock. These operations include volume creation, deletion, and resizing; snapshot creation and deletion; volume publish and unpublish; and backend management.

When you enable controller scalability, operations on different volumes and backends proceed concurrently. This increases throughput and reduces end-to-end operation time in environments with high numbers of concurrent `PersistentVolumeClaim` and `VolumeSnapshot` operations.

Default behavior (serial mode)

By default, the Trident controller processes operations one at a time. Each create, delete, resize, or snapshot request completes before Trident starts the next one. Serial mode is the supported default for all installations and requires no configuration.

Serial mode is sufficient for most workloads. Enable controller scalability only when operation volume creates a backlog under serial processing.

Controller scalability support

Trident supports controller scalability at different maturity levels for different storage drivers.

General availability

The following drivers support controller scalability at general availability in Trident 26.06:

- `ontap-san`
- `ontap-nas`
- `ontap-nas-economy`
- `ontap-san-economy`
- `google-cloud-netapp-volumes`
- `azure-netapp-files`
- `solidfire-san`



The `google-cloud-netapp-volumes` and `google-cloud-netapp-volumes-san` drivers are different. Trident supports only `google-cloud-netapp-volumes`. Do not use `google-cloud-netapp-volumes-san` in backend configurations or examples.

Technical preview

The following driver supports controller scalability as a technical preview feature in Trident 26.06:

- `asa-r2` (SAN and NVMe)

This driver has the following limitations:

- Controller concurrency is available for evaluation and testing only.
- Behavior can change from one release to the next.
- NetApp does not recommend use in production environments.

How enableConcurrency works

When you set `enableConcurrency` to `true`, Trident applies concurrent processing across all backends that the controller manages. The setting applies to every backend at once. You cannot enable it for individual backends or individual drivers.

Every configured backend must use a driver in the general availability or technical preview list. If any backend uses an unsupported driver, Trident does not start, and Trident does not add a backend that uses an unsupported driver.

To restore startup, remove or reconfigure any backend that uses an unsupported driver, or disable controller scalability.

Before you enable



Before you enable controller scalability, confirm that every configured backend uses a driver in the general availability or technical preview list. If any backend uses an unsupported driver, Trident does not start after you enable the feature.

Use the following table to decide whether to enable controller scalability.

If your environment	Then
Uses only supported drivers and experiences a backlog of controller operations	Enable controller scalability.
Uses any unsupported driver	Keep the default serial mode. Do not enable controller scalability.
Handles low operation volume with no backlog	Keep the default serial mode.

Enable controller scalability

The `enableConcurrency` configuration option controls controller scalability. You must explicitly enable this option during Trident installation, or when you update an existing deployment.

Trident operator deployment

To enable controller scalability with the Trident operator, set `enableConcurrency` to `true` in the `TridentOrchestrator` custom resource (CR).

New installation

Create or edit the `TridentOrchestrator` CR and set `enableConcurrency` to `true`:

```
apiVersion: trident.netapp.io/v1
kind: TridentOrchestrator
metadata:
  name: trident
spec:
  namespace: trident
  enableConcurrency: true
```

Apply the CR:

```
kubectl apply -f tridentorchestrator_cr.yaml
```

Existing installation

Patch the existing `TridentOrchestrator` CR to enable controller scalability:

```
kubectl patch torc trident --type=merge -p
'{"spec":{"enableConcurrency":true}}'
```

Verify that Trident applied the setting:

```
kubectl get torc trident -o
```

```
jsonpath='{.status.currentInstallationParams.enableConcurrency}'
```

Helm deployment

To enable controller scalability with Helm, set the `enableConcurrency` value to `true`.

New installation

```
helm install trident netapp-trident/trident-operator --namespace trident  
--create-namespace --set enableConcurrency=true
```

Existing installation

```
helm upgrade trident netapp-trident/trident-operator --namespace trident  
--set enableConcurrency=true
```

Alternatively, set `enableConcurrency` to `true` in a custom `values.yaml` file:

```
# values.yaml  
enableConcurrency: true
```

Then install or upgrade with the values file:

```
helm install trident netapp-trident/trident-operator --namespace trident  
--create-namespace -f values.yaml
```

tridentctl deployment

To enable controller scalability with `tridentctl`, pass the `--enable-concurrency` flag during installation.

New installation

```
tridentctl install -n trident --enable-concurrency
```

Existing installation

To enable controller scalability on an existing `tridentctl` deployment, uninstall Trident and reinstall it with the flag:

```
tridentctl uninstall -n trident  
tridentctl install -n trident --enable-concurrency
```

Verify that controller scalability is enabled

After you enable controller scalability, verify that the Trident controller runs with concurrency enabled. Check the controller pod logs:

```
kubectl logs -n trident deploy/trident-controller | grep -i concurrency
```

The output includes a log entry that confirms concurrency is enabled.

Concurrency behavior

When controller scalability is enabled, the Trident controller applies the following behavior:

- Trident replaces the single global lock with fine-grained, per-resource locking.
- Trident serializes operations that modify the same resource to maintain data consistency.
- Operations that only read from a resource proceed concurrently with other read operations on that resource.
- Trident limits concurrent ONTAP API requests to 20 per management LIF to prevent overload of backend storage systems.
- If multiple backends share the same management LIF, they share this 20-request limit.

Caveats and limitations

The following considerations apply to controller scalability in Trident 26.06:

- Controller scalability supports only the drivers in the general availability and technical preview lists. For details, see [Before you enable](#).
- The Trident controller manages concurrency internally. This release provides no user-configurable concurrency limits.
- Overall throughput depends on the storage driver in use, backend responsiveness, and Kubernetes API server performance.
- High concurrency can increase load on backend storage systems.
- Controller scalability behavior is not identical across all drivers.
- The technical preview driver can exhibit inconsistent performance under high load and can change behavior between releases.
- Debugging concurrent operations can be more complex because of parallel execution. Metrics and logs can show interleaved operation output.

Recommendations

Before you enable controller scalability, complete the following steps:

1. Confirm that every configured backend uses a driver in the general availability or technical preview list.
2. Test the change in a non-production cluster before you apply it in production.
3. Verify that controller scalability is enabled after you apply the change.

Apply the following general recommendations when you operate with controller scalability:

- Use general availability drivers for production environments that require high scalability.
- Evaluate the technical preview driver only in non-production environments.
- Monitor backend and controller performance when you operate at scale.
- Do not assume operation ordering in automation scripts.

Automatic volume expansion

Automatic volume expansion enables Persistent Volumes provisioned by Trident to grow automatically when used capacity reaches a defined threshold. This capability reduces operational overhead and helps prevent application disruption caused by capacity exhaustion. Automatic volume expansion is implemented using Autogrow Policies. An Autogrow Policy defines:

- The utilization threshold that triggers expansion
- The amount by which the volume grows
- The maximum size the volume can reach



Volumes increase in size automatically when the defined utilization threshold is exceeded. Volumes are never automatically reduced.

Requirements

Before configuring automatic volume expansion, ensure that the following requirements are met:

- Trident 26.02 or later
- Role-based access control permissions to create `TridentAutogrowPolicy` custom resources
- StorageClasses configured with `allowVolumeExpansion: true`
- Supported ONTAP protocols:
 - Network File System (NFS)
 - Internet Small Computer Systems Interface (iSCSI)
 - Non-Volatile Memory Express (NVMe)
 - Fibre Channel Protocol (FCP)

Limitations

- ONTAP Non-Volatile Memory Express raw block volumes earlier than ONTAP 9.16.1 do not support automatic expansion.
- For storage area network volumes, if `growthAmount` is less than or equal to 50 mebibytes, Trident automatically increases the value to 51 mebibytes before resizing, provided the resulting size does not exceed `maxSize`.
- In brownfield environments, automatic expansion might not function for certain existing volumes due to volume publication migration behavior.
- When a volume reaches `maxSize`, no further expansion occurs.

- Supported protocols for automatic volume expansion:
 - Network File System (NFS)
 - Internet Small Computer Systems Interface (iSCSI)
 - Non-Volatile Memory Express (NVMe)
 - Fibre Channel Protocol (FCP)

Provision Volumes with Autogrow Policy

Autogrow Policy can be configured at two levels:

- Storage class level: Sets default for all volumes (using annotation)
- PVC level: Overrides storage class default (using annotation)

Create an Autogrow Policy

Autogrow Policies enable automatic volume expansion when volumes reach a defined capacity threshold.

Ensure you have:

- Trident 26.02 or later installed
- Role-based access control permissions to create `TridentAutogrowPolicy` resources
- Understanding of workload growth requirements

An Autogrow Policy defines how volumes expand automatically when they reach a defined capacity threshold.

You can create Autogrow Policies at any point in your workflow:

- Before `StorageClasses` and volumes are created
- After `StorageClasses` exist
- After volumes are provisioned

This flexibility allows you to introduce automatic expansion without recreating existing resources.

Autogrow Policy specifications

Autogrow Policies are Kubernetes custom resources defined as follows:

Field	Description	Format	Required	Example	Default
name	Unique policy identifier	String	Yes	production-db-policy	None
usedThreshhold	Capacity percentage that triggers expansion	Percentage string	Yes	"80%"	None
growthAmount	Amount to grow when threshold is reached	Percentage or size	No	"10%" or "5Gi"	"10%"
maxSize	Maximum volume size limit	Kubernetes quantity	No	"500Gi"	Unlimited

Create an Autogrow Policy

Steps

1. Create a YAML file that defines your Autogrow Policy:

```
apiVersion: trident.netapp.io/v1
kind: TridentAutogrowPolicy
metadata:
  name: standard-autogrow
spec:
  usedThreshold: "80%"
  growthAmount: "10%"
  maxSize: "500Gi"
```

2. Apply the policy to your cluster:

```
kubectl apply -f autogrow-policy.yaml
```

3. Verify that the policy was created:

```
kubectl get tridentautogrowpolicy standard-autogrow
```

Expected output

NAME	USED THRESHOLD	GROWTH AMOUNT	STATE
standard-autogrow	80%	10%	Success

Policy states

After you create a policy, Trident validates the specification and assigns one of the following states:

State	Description	Action required
Success	Policy is validated and ready to use.	None.
Failed	Validation errors detected.	Review and fix the specification.
Deleting	Deletion is in progress.	Wait for completion.

Associate a policy with a StorageClass

You can associate an Autogrow Policy with a StorageClass by using the `trident.netapp.io/autogrowPolicy` annotation. All volumes provisioned from that StorageClass inherit the policy.



The StorageClass must have `allowVolumeExpansion: true`.

Steps

1. Create or modify a StorageClass with the Autogrow Policy annotation:

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: ontap-gold
  annotations:
    trident.netapp.io/autogrowPolicy: "production-db-policy"
provisioner: csi.trident.netapp.io
parameters:
  backendType: "ontap-san"
  fsType: "ext4"
allowVolumeExpansion: true
```

2. Apply the StorageClass:

```
kubectl apply -f storageclass.yaml
```

3. Verify the annotation:

```
kubectl get storageclass ontap-gold -o
jsonpath='{.metadata.annotations.trident\.netapp\.io/autogrowPolicy}'
```

Expected output

```
production-db-policy
```

Policy precedence

When Autogrow Policy annotations are set on both a StorageClass and a PVC, Trident applies the following precedence rules:

1. **PVC annotation takes priority.** If a PVC sets `trident.netapp.io/autogrowPolicy`, that value is always used.
2. **StorageClass annotation applies only when the PVC has no annotation.**
3. **If neither has the annotation,** no Autogrow Policy is applied.

Table 1. Precedence examples

StorageClass annotation	PVC annotation	Effective behavior
trident.netapp.io/autogrowPolicy: standard-agp	Not set	Uses standard-agp.

StorageClass annotation	PVC annotation	Effective behavior
trident.netapp.io/autogrowPolicy: standard-agp	trident.netapp.io/autogrowPolicy: logs-policy	Uses logs-policy (PVC overrides StorageClass).
trident.netapp.io/autogrowPolicy: standard-agp	trident.netapp.io/autogrowPolicy: "none"	No Autogrow Policy (PVC disables autogrow).
Not set	trident.netapp.io/autogrowPolicy: dev-policy	Uses dev-policy.
Not set	Not set	No Autogrow Policy.

Configuration examples

The following examples show common Autogrow Policy configurations for different use cases.

Conservative policy for production databases

```
apiVersion: trident.netapp.io/v1
kind: TridentAutogrowPolicy
metadata:
  name: production-db-policy
spec:
  usedThreshold: "75%"
  growthAmount: "20%"
  maxSize: "5Ti"
```

Log storage with fixed growth increments

```
apiVersion: trident.netapp.io/v1
kind: TridentAutogrowPolicy
metadata:
  name: log-storage-policy
spec:
  usedThreshold: "90%"
  growthAmount: "10Gi"
  maxSize: "100Gi"
```

Minimal policy with defaults

When you omit growthAmount and maxSize, Trident uses the defaults (10% growth, unlimited size):

```
apiVersion: trident.netapp.io/v1
```

```
kind: TridentAutogrowPolicy
metadata:
  name: logs-policy
spec:
  usedThreshold: "85%"
```

Policy with a custom maxSize and default growthAmount

```
apiVersion: trident.netapp.io/v1
kind: TridentAutogrowPolicy
metadata:
  name: default-ga-policy
spec:
  usedThreshold: "70%"
  maxSize: "100Gi"
```

Aggressive growth with unlimited maxSize

```
apiVersion: trident.netapp.io/v1
kind: TridentAutogrowPolicy
metadata:
  name: aggressive-growth-policy
spec:
  usedThreshold: "80%"
  growthAmount: "150%"
```

Policy with fractional percentages

```
apiVersion: trident.netapp.io/v1
kind: TridentAutogrowPolicy
metadata:
  name: precise-policy
spec:
  usedThreshold: "80.28%"
  growthAmount: "10.65%"
  maxSize: "100Gi"
```



Fractional percentages are supported. If you specify more than three decimal places, Trident rounds the value to three decimal places.

NAS StorageClass with Autogrow

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: ontap-nas-autogrow
  annotations:
    trident.netapp.io/autogrowPolicy: "standard-autogrow"
provisioner: csi.trident.netapp.io
parameters:
  backendType: "ontap-nas"
  fsType: "ext4"
allowVolumeExpansion: true
```

SAN StorageClass with Autogrow

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: database-storage
  annotations:
    trident.netapp.io/autogrowPolicy: "production-db-policy"
provisioner: csi.trident.netapp.io
parameters:
  backendType: "ontap-san"
  fsType: "ext4"
allowVolumeExpansion: true
```

Manage Autogrow Policies

After you create Autogrow Policies, you can view, update, and delete them. You can also monitor which volumes use a given policy.

View Autogrow Policies

List all policies

Use `kubectl` to list all Autogrow Policies in your cluster:

```
kubectl get tridentautogrowpolicy
```

Alternatively, use `tridentctl`:

```
tridentctl get autogrowpolicy
```

View policy details

To view the full specification and status of a policy:

```
kubectl describe tridentautogrowpolicy production-db-policy
```

To view a policy with its associated volumes in YAML format:

```
tridentctl get autogrowpolicy production-db-policy -o yaml
```

Update an Autogrow Policy

You can modify an existing policy to change its threshold, growth amount, or maximum size. Changes take effect immediately for all volumes that use the policy.



Changes affect all volumes that use the policy. When possible, test changes in a non-production environment first.

Steps

1. Edit the policy:

```
kubectl edit tridentautogrowpolicy production-db-policy
```

2. Modify the `spec` fields as needed:

```
spec:
  usedThreshold: "75%"      # Changed from 80%
  growthAmount:  "20%"      # Changed from 10%
  maxSize:       "1Ti"      # Changed from 500Gi
```

3. Save and exit. The changes take effect immediately.

Update considerations

- **Immediate effect:** All volumes that use the policy adopt the new parameters at the next growth evaluation.
- **No volume restart required:** Changes apply to the next growth operation.
- **Test first:** Validate changes in a non-production environment when possible.
- **Communicate changes:** Notify teams when you modify shared policies.

Delete an Autogrow Policy

Autogrow Policies use finalizer protection to prevent accidental deletion while volumes actively use them.

Steps

1. Delete the policy:

```
kubectl delete tridentautogrowpolicy production-db-policy
```

2. If volumes still use the policy, the deletion enters a `Deleting` state. Check which volumes are affected:

```
tridentctl get autogrowpolicy production-db-policy -o yaml
```

3. Remove the policy from each affected volume. Choose one of the following options:

- **Option A: Explicitly disable autogrow** by setting the annotation to "none":

```
kubectl annotate pvc <pvc-name> \
  trident.netapp.io/autogrowPolicy="none" \
  --overwrite
```

- **Option B: Remove the annotation entirely:**

```
kubectl annotate pvc <pvc-name> \
  trident.netapp.io/autogrowPolicy-
```

Deletion behavior

Scenario	Behavior
No volumes use the policy	Policy is deleted immediately.
Volumes use the policy	Policy enters <code>Deleting</code> state. A finalizer blocks completion until all volumes are removed.
All volumes are removed from the policy	Finalizers are removed and the policy is deleted.

Monitor Autogrow Policy usage

Check volumes using a policy

```
tridentctl get autogrowpolicy production-db-policy -o json | jq '.volumes'
```

Find which policy a volume uses

```
kubectl get pvc database-pvc -o
jsonpath='{.metadata.annotations.trident\.netapp\.io/autogrowPolicy}'
```

Monitor policy events

```
kubectl get events --field-selector
involvedObject.kind=TridentAutogrowPolicy
```

Supported protocols

Autogrow supports the following storage protocols:

- NFS
- iSCSI
- FCP
- NVMe



For SAN volumes, if the configured `growthAmount` is 50 MiB or less, Trident automatically increases the growth amount to 51 MB for the resize operation, as long as the resulting size does not exceed `maxSize`.

Supported backends

Autogrow supports ONTAP backends. Autogrow also supports Google Cloud NetApp Volumes (GCNV) for both NAS and SAN volumes.

Known limitations

- **ONTAP NVMe raw block volumes:** Volumes created with ONTAP versions earlier than 9.16.1 do not support autogrow.
- **Existing volumes (brownfield deployments):** Autogrow might not work for existing volumes even if a valid Autogrow Policy is applied. This limitation is due to an ongoing migration of volume publications. To confirm that migration is complete, check the Trident controller logs for "Migration completed" messages.

Frequently asked questions

When does Trident evaluate the threshold?

Trident continuously monitors volume usage. When the used capacity crosses the `usedThreshold`, Trident creates an internal resize request and expands the volume by the configured `growthAmount`.

For example, this policy triggers expansion at 80% capacity and grows the volume by 10% each time, up to a maximum of 500 GiB:

```
apiVersion: trident.netapp.io/v1
kind: TridentAutogrowPolicy
metadata:
  name: standard-autogrow
spec:
  usedThreshold: "80%"
  growthAmount: "10%"
  maxSize: "500Gi"
```

Can I apply a policy after volumes are already provisioned?

Yes. You can create an Autogrow Policy at any time and apply it to existing PVCs by adding or updating the `trident.netapp.io/autogrowPolicy` annotation. You do not need to recreate the PVC or the StorageClass.

To apply a policy to an existing PVC:

```
kubectl annotate pvc <pvc-name> \
  trident.netapp.io/autogrowPolicy="production-db-policy" \
  --overwrite
```

To apply a policy to an existing StorageClass:

```
kubectl annotate storageclass ontap-gold \
  trident.netapp.io/autogrowPolicy="production-db-policy" \
  --overwrite
```

What happens if I set an Autogrow Policy on both the StorageClass and the PVC?

The PVC annotation always takes precedence. If a PVC has the `trident.netapp.io/autogrowPolicy` annotation, Trident uses that value regardless of what the StorageClass specifies. Refer to [Policy precedence](#) for details.

For example, given this StorageClass:

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: ontap-gold
  annotations:
    trident.netapp.io/autogrowPolicy: "standard-agp"
provisioner: csi.trident.netapp.io
allowVolumeExpansion: true
```

And this PVC that overrides the StorageClass policy:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: database-pvc
  annotations:
    trident.netapp.io/autogrowPolicy: "logs-policy"
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 50Gi
  storageClassName: ontap-gold
```

Trident uses logs-policy for database-pvc, not standard-agp.

How do I disable autogrow for a specific volume?

Set the PVC annotation to "none". This setting overrides any StorageClass-level policy for that volume:

```
kubectl annotate pvc <pvc-name> \
  trident.netapp.io/autogrowPolicy="none" \
  --overwrite
```

You can verify that autogrow is disabled:

```
kubectl get pvc <pvc-name> -o jsonpath
='{.metadata.annotations.trident\.netapp\.io/autogrowPolicy}'
```

Expected output

```
none
```

What happens when a volume reaches maxSize?

Trident stops expanding the volume. It creates no further resize requests for that volume, even if usage continues to increase beyond the `usedThreshold`.

For example, with this policy, Trident stops growing the volume after it reaches 100 GiB:

```
apiVersion: trident.netapp.io/v1
kind: TridentAutogrowPolicy
```

```

metadata:
  name: capped-policy
spec:
  usedThreshold: "90%"
  growthAmount: "10Gi"
  maxSize: "100Gi"

```

To allow unlimited growth, omit `maxSize` or set it to 0:

```

apiVersion: trident.netapp.io/v1
kind: TridentAutogrowPolicy
metadata:
  name: unlimited-policy
spec:
  usedThreshold: "85%"
  growthAmount: "10%"

```

Can I change a policy without restarting volumes?

Yes. When you update a policy, all volumes that use that policy adopt the new parameters at the next growth evaluation. No volume restarts are required.

To update a policy in place:

```
kubectl edit tridentautogrowpolicy production-db-policy
```

Modify the fields as needed:

```

spec:
  usedThreshold: "75%"      # Changed from 80%
  growthAmount: "20%"      # Changed from 10%
  maxSize: "1Ti"           # Changed from 500Gi

```

Save and exit. Verify the updated policy:

```
kubectl get tridentautogrowpolicy production-db-policy
```

Expected output

NAME	USED THRESHOLD	GROWTH AMOUNT	STATE
production-db-policy	75%	20%	Success

Why is my policy in a Failed state?

A `Failed` state indicates that the policy specification contains validation errors. Run the following command to view the error details:

```
kubectl describe tridentautogrowpolicy <policy-name>
```

Common causes include an invalid `usedThreshold` (must be 1–99%), a `growthAmount` that exceeds `maxSize`, or an invalid Kubernetes quantity format. Correct the specification and reapply:

```
kubectl apply -f autogrow-policy.yaml
```

Why can't I delete a policy?

Policies use finalizer protection. If volumes still use the policy, deletion enters a `Deleting` state and waits until you remove all volumes from the policy.

Identify the affected volumes:

```
tridentctl get autogrowpolicy production-db-policy -o yaml
```

Then remove the annotation from each PVC:

```
# Option A: Explicitly disable autogrow
kubectl annotate pvc <pvc-name> \
  trident.netapp.io/autogrowPolicy="none" \
  --overwrite

# Option B: Remove the annotation entirely
kubectl annotate pvc <pvc-name> \
  trident.netapp.io/autogrowPolicy-
```

After you remove all volumes, Trident releases the finalizer and deletes the policy.

Which backends does autogrow support?

Autogrow supports ONTAP backends. It also supports Google Cloud NetApp Volumes (GCNV) for both NAS and SAN volumes.

Autogrow supports the NFS, iSCSI, FCP, and NVMe protocols. NVMe raw block volumes require ONTAP 9.16.1 or later.

Existing volumes in brownfield deployments might require volume publication migration to complete before autogrow takes effect. To verify migration status, check the Trident controller logs:

```
kubectl logs -l app=trident-controller -n trident | grep "Migration completed"
```

The following StorageClass examples show autogrow configured for NAS and SAN backends:

NAS backend

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: ontap-nas-autogrow
  annotations:
    trident.netapp.io/autogrowPolicy: "standard-autogrow"
provisioner: csi.trident.netapp.io
parameters:
  backendType: "ontap-nas"
  fsType: "ext4"
allowVolumeExpansion: true
```

SAN backend

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: database-storage
  annotations:
    trident.netapp.io/autogrowPolicy: "production-db-policy"
provisioner: csi.trident.netapp.io
parameters:
  backendType: "ontap-san"
  fsType: "ext4"
allowVolumeExpansion: true
```

What is the minimum growth amount for SAN volumes?

For SAN volumes, the effective minimum growth amount is 51 MB. If you configure a `growthAmount` of 50 MiB or less, Trident automatically increases the growth to 51 MB for the resize operation.

For example, this policy sets a `growthAmount` of "40Mi", but Trident applies a 51 MB growth for any SAN volume that uses it:

```
apiVersion: trident.netapp.io/v1
kind: TridentAutogrowPolicy
metadata:
  name: san-minimal-policy
```

```
spec:
  usedThreshold: "85%"
  growthAmount: "40Mi"
  maxSize: "100Gi"
```

To avoid this automatic adjustment, set `growthAmount` to a value greater than 50 MiB:

```
apiVersion: trident.netapp.io/v1
kind: TridentAutogrowPolicy
metadata:
  name: san-policy
spec:
  usedThreshold: "85%"
  growthAmount: "100Mi"
  maxSize: "500Gi"
```

Import Trident volumes

You can import existing storage volumes as a Kubernetes PV using `tridentctl import` or by creating a Persistent Volume Claim (PVC) with Trident import annotations.

Overview and considerations

You might import a volume into Trident to:

- Containerize an application and reuse its existing data set
- Use a clone of a data set for an ephemeral application
- Rebuild a failed Kubernetes cluster
- Migrate application data during disaster recovery

Considerations

Before importing a volume, review the following considerations.

- Trident can import RW (read-write) type ONTAP volumes only. DP (data protection) type volumes are SnapMirror destination volumes. You should break the mirror relationship before importing the volume into Trident.
- We suggest importing volumes without active connections. To import an actively-used volume, clone the volume and then perform the import.



This is especially important for block volumes as Kubernetes would be unaware of the previous connection and could easily attach an active volume to a pod. This can result in data corruption.

- Though `StorageClass` must be specified on a PVC, Trident does not use this parameter during import. Storage classes are used during volume creation to select from available pools based on storage characteristics. Because the volume already exists, no pool selection is required during import. Therefore,

the import will not fail even if the volume exists on a backend or pool that does not match the storage class specified in the PVC.

- The existing volume size is determined and set in the PVC. After the volume is imported by the storage driver, the PV is created with a ClaimRef to the PVC.
 - The reclaim policy is initially set to `retain` in the PV. After Kubernetes successfully binds the PVC and PV, the reclaim policy is updated to match the reclaim policy of the Storage Class.
 - If the reclaim policy of the Storage Class is `delete`, Trident deletes the storage volume when the PV is deleted.
- By default, Trident manages the PVC and renames the FlexVol volume and the LUN or qtree on the backend. You can pass the `--no-manage` flag to import an unmanaged volume and the `--no-rename` flag to retain the volume name.
 - **--no-manage** - If you use the `--no-manage` flag, Trident does not perform any additional operations on the PVC or PV for the lifecycle of the objects. The storage volume is not deleted when the PV is deleted and other operations such as volume clone and volume resize are also ignored.
 - **--no-rename** - If you use the `--no-rename` flag, Trident retains the existing volume name while importing volumes, and manages the lifecycle of the volumes. This option is supported only for the `ontap-nas`, `ontap-san` (including ASA r2 systems), `ontap-san-economy`, and `ontap-nas-economy` (qtree import) drivers.



These options are useful if you want to use Kubernetes for containerized workloads but otherwise want to manage the lifecycle of the storage volume outside of Kubernetes.

- An annotation is added to the PVC and PV that serves a dual purpose of indicating that the volume was imported and if the PVC and PV are managed. This annotation should not be modified or removed.

Import a volume

You can import a volume using either `tridentctl import` or by creating a PVC with Trident import annotations.



If you use PVC annotations, you don't need to download or use `tridentctl` to import the volume.

Using tridentctl

Steps

1. Create a PVC file (for example, `pvc.yaml`) that will be used to create the PVC. The PVC file should include `name`, `namespace`, `accessModes`, and `storageClassName`. Optionally, you can specify `unixPermissions` in your PVC definition.

The following is an example of a minimum specification:

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: my_claim
  namespace: my_namespace
```

```
spec:
  accessModes:
    - ReadWriteOnce
  storageClassName: my_storage_class
```



Include only the required parameters. Additional parameters such as PV name or volume size can cause the import command to fail.

2. Use the `tridentctl import volume` command to specify the name of the Trident backend containing the volume and the name that uniquely identifies the volume on the storage (for example: ONTAP FlexVol, Element Volume). The `-f` argument is required to specify the path to the PVC file.

```
tridentctl import volume <backendName> <volumeName> -f <path-to-pvc-file>
```

Using PVC annotations

Steps

1. Create a PVC YAML file (for example, `pvc.yaml`) with the required Trident import annotations. The PVC file should include:
 - name and namespace in metadata
 - accessModes, resources.requests.storage, and storageClassName in spec
 - Annotations:
 - `trident.netapp.io/importOriginalName`: Volume name on the backend
 - `trident.netapp.io/importBackendUUID`: Backend UUID where volume exists
 - `trident.netapp.io/notManaged` (*Optional*): Set to "true" for unmanaged volumes. Default is "false".

The following is an example specification for importing a managed volume:

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: <pvc-name>
  namespace: <namespace>
  annotations:
    trident.netapp.io/importOriginalName: "<volume-name>"
    trident.netapp.io/importBackendUUID: "<backend-uuid>"
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: <size>
```

```
storageClassName: <storage-class-name>
```

2. Apply the PVC YAML file to your Kubernetes cluster:

```
kubectl apply -f <pvc-file>.yaml
```

Trident will automatically import the volume and bind it to the PVC.

Examples

Review the following volume import examples for supported drivers.

ONTAP NAS and ONTAP NAS FlexGroup

Trident supports volume import using the `ontap-nas` and `ontap-nas-flexgroup` drivers.



- The `ontap-nas-economy` driver manages storage at the qtree level, so it does not support importing a FlexVol volume on its own. To import qtrees with the `ontap-nas-economy` driver, refer to [Import qtrees](#) (Tech Preview). When Trident imports a qtree, it also imports the parent FlexVol as a bucket that contains the qtree.
- The `ontap-nas` and `ontap-nas-flexgroup` drivers do not allow duplicate volume names.

Each volume created with the `ontap-nas` driver is a FlexVol volume on the ONTAP cluster. Importing FlexVol volumes with the `ontap-nas` driver works the same. A FlexVol volumes that already exists on an ONTAP cluster can be imported as a `ontap-nas` PVC. Similarly, FlexGroup vols can be imported as `ontap-nas-flexgroup` PVCs.

ONTAP NAS examples using tridentctl

The following examples show how to import managed and unmanaged volumes using `tridentctl`.

Managed volume

The following example imports a volume named `managed_volume` on a backend named `ontap_nas`:

```
tridentctl import volume ontap_nas managed_volume -f <path-to-pvc-file>
```

```
+-----+-----+-----+-----+
+-----+-----+-----+-----+
|          NAME          | SIZE  | STORAGE CLASS |
+-----+-----+-----+-----+
|          |          |          |          |
|          |          |          |          |
|          |          |          |          |
+-----+-----+-----+-----+
+-----+-----+-----+-----+
| pvc-bf5ad463-afbb-11e9-8d9f-5254004dfdb7 | 1.0 GiB | standard      |
| file          | c5a6f6a4-b052-423b-80d4-8fb491a14a22 | online | true          |
+-----+-----+-----+-----+
```

+-----+-----+-----+-----+-----+

Unmanaged volume

When using the `--no-manage` argument, Trident does not rename the volume.

The following example imports `unmanaged_volume` on the `ontap_nas` backend:

```
tridentctl import volume nas_blog unmanaged_volume -f <path-to-pvc-
file> --no-manage
```

PROTOCOL	NAME	BACKEND UUID	SIZE	STATE	STORAGE CLASS	MANAGED
file	pvc-df07d542-afbc-11e9-8d9f-5254004dfdb7	c5a6f6a4-b052-423b-80d4-8fb491a14a22	1.0 GiB	online	standard	false

ONTAP NAS examples using PVC annotations

The following examples show how to import managed and unmanaged volumes using PVC annotations.

Managed volume

The following example imports a 1GiB `ontap-nas` volume named `ontap_volume1` from backend `81abcb27-ea63-49bb-b606-0a5315ac5f21` with `RWO` access mode set using PVC annotations:

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: <managed-imported-volume>
  namespace: <namespace>
  annotations:
    trident.netapp.io/importOriginalName: "ontap_volume1"
    trident.netapp.io/importBackendUUID: "81abcb27-ea63-49bb-b606-
0a5315ac5f21"
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

```
storageClassName: <storage-class-name>
```

Unmanaged volume

The following example imports 1Gi ontap-nas volume named ontap-volume2 from backend 34abcb27-ea63-49bb-b606-0a5315ac5f34 with RWO access mode set using PVC annotations:

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: <unmanaged-imported-volume>
  namespace: <namespace>
  annotations:
    trident.netapp.io/importOriginalName: "ontap-volume2"
    trident.netapp.io/importBackendUUID: "34abcb27-ea63-49bb-b606-0a5315ac5f34"
    trident.netapp.io/notManaged: "true"
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
  storageClassName: <storage-class-name>
```

Import qtrees



Qtree import is a Tech Preview feature in Trident 26.06. Tech Preview features are not supported for production use.

Beginning with Trident 26.06, you can import existing ONTAP qtrees into Trident using the `ontap-nas-economy` driver. Identify the qtree by its FlexVol volume and qtree names in `flexvol/mtree` format, along with the UUID of the backend that contains it. The general import considerations in [Overview and considerations](#) also apply to qtree import.

Import a qtree using `tridentctl`

Specify the qtree in `flexvol/mtree` format as the volume name. The following example imports the qtree `mtree1` from the FlexVol volume `flexvol1` on the backend `ontap_nas_economy`:

```
tridentctl import volume ontap_nas_economy flexvol1/mtree1 -f <path-to-pvc-file>
```

To import the qtree without renaming it, add the `--no-rename` flag. To import the qtree without Trident managing its lifecycle, add the `--no-manage` flag.

Import a qtree using PVC annotations

Create a PVC with the following annotations:

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: <pvc-name>
  namespace: <namespace>
  annotations:
    trident.netapp.io/importOriginalName: "flexvol1/qtree1"
    trident.netapp.io/importBackendUUID: "<backend-uuid>"
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: <size>
  storageClassName: <storage-class-name>
```

To keep the existing FlexVol volume and qtree names, add the `trident.netapp.io/importNoRename: "true"` annotation. To import the qtree as unmanaged, add the `trident.netapp.io/notManaged: "true"` annotation.

Naming and management behavior

Managed import (default)

Trident renames the qtree to match the PVC name and renames the FlexVol volume to the `ontap-nas-economy` naming convention. The junction path does not change.

`--no-rename`

Trident keeps the existing FlexVol volume and qtree names. The names must already follow the Trident naming conventions, or Trident cannot manage delete or mount operations for the qtree.

`--no-manage`

Trident references the qtree but does not manage its lifecycle. The names must follow the Trident naming conventions, and Trident does not use or manage auto export policies for the qtree.

Export policy behavior

If the backend uses auto export policies, the imported qtree uses the per-qtree export policy model, in which each qtree has its own export policy. Trident assigns the qtree an empty export policy at import. At publish time, Trident updates the FlexVol volume to the per-backend export policy if auto export is enabled and the existing FlexVol export policy is `default` or `empty`.



Updating the FlexVol volume export policy can disrupt other qtrees in the same FlexVol volume that are currently mounted. Import qtrees only when the qtrees in the FlexVol volume are unmounted.



Do not point two Kubernetes clusters at the same active SVM, and do not import a qtree that


```
+-----+-----+-----+-----+
| pvc-1fc999c9-ce8c-459c-82e4-ed4380a4b228 | 1.0 GiB | san-blog |
block | e3275890-7d80-4af6-90cc-c7a0759f555a | online | false |
+-----+-----+-----+-----+
+-----+-----+-----+-----+
```

If you have LUNS mapped to igroups that share an IQN with a Kubernetes node IQN, as shown in the following example, you will receive the error: LUN already mapped to initiator(s) in this group. You will need to remove the initiator or unmap the LUN to import the volume.



Vserver	Igroup	Protocol	OS Type	Initiators
svm0	k8s-nodename.example.com-fe5d36f2-cded-4f38-9eb0-c7719fc2f9f3	iscsi	linux	iqn.1994-05.com.redhat:4c2e1cf35e0
svm0	unmanaged-example-igroup	mixed	linux	iqn.1994-05.com.redhat:4c2e1cf35e0

ONTAP SAN-economy examples

The following examples show how to import managed and unmanaged volumes for the `ontap-san-economy` backend.

Managed volume

When you import a managed volume, Trident takes ownership of the FlexVol and renames it. You must account for this renaming when you import multiple LUNs from the same FlexVol.

The following example imports `lun1` from the FlexVol `toimport` as a managed volume named `vol-managed-saneco`:

```
tridentctl import volume vol-managed-saneco toimport/lun1 -f
import1.yaml
```

After importing `lun1`, Trident renames the FlexVol (for example, to `trident_lun_pool_xyz`). To import additional LUNs from the same FlexVol, use the new FlexVol name:

```
tridentctl import volume vol-managed-saneco trident_lun_pool_xyz/lun2
-f import2.yaml
```



The `ontap-san-economy` backend imports one LUN at a time. You can automate multiple imports using a script.

When you import an unmanaged volume, Trident does not take ownership of the FlexVol. However, the FlexVol and LUN must follow Trident naming conventions.

trident_lun_pool_STORAGEPREFIX_RANDOMSTRING

- ```
trident lun pool xyz randomstring/lun0
```

```
+-----+-----+-----+-----+
+-----+-----+-----+-----+
| NAME | SIZE | STORAGE CLASS |
+-----+-----+-----+-----+
| PROTOCOL | BACKEND UUID | STATE | MANAGED |
+-----+-----+-----+-----+
+-----+-----+-----+-----+
| pvc-970ce1ca-2096-4ecd-8545-ac7edc24a8fe | 10 GiB | basic-element |
+-----+-----+-----+-----+
| block | d3ba047a-ea0b-43f9-9c42-e38e58301c49 | online | true |
+-----+-----+-----+-----+
+-----+-----+-----+-----+
```

## Azure NetApp Files

Trident supports volume import using the `azure-netapp-files` driver.



To import an Azure NetApp Files volume, identify the volume by its volume path. The volume path is the portion of the volume's export path after the `:/`. For example, if the mount path is `10.0.0.2:/importvol1`, the volume path is `importvol1`.

The following example imports an `azure-netapp-files` volume on backend `azurenetaappfiles_40517` with the volume path `importvol1`.

```
tridentctl import volume azurenetaappfiles_40517 importvol1 -f <path-to-pvc-file> -n trident
```

|          |                                          |  |         |  |               |
|----------|------------------------------------------|--|---------|--|---------------|
|          |                                          |  |         |  |               |
|          |                                          |  |         |  |               |
|          | NAME                                     |  | SIZE    |  | STORAGE CLASS |
| PROTOCOL | BACKEND UUID                             |  | STATE   |  | MANAGED       |
|          |                                          |  |         |  |               |
|          |                                          |  |         |  |               |
|          | pvc-0ee95d60-fd5c-448d-b505-b72901b3a4ab |  | 100 GiB |  | anf-storage   |
| file     | 1c01274f-d94b-44a3-98a3-04c953c9a51e     |  | online  |  | true          |
|          |                                          |  |         |  |               |
|          |                                          |  |         |  |               |

## Google Cloud NetApp Volumes

Trident supports volume import using the `google-cloud-netapp-volumes` driver.

The following example imports a volume on backend `backend-tbc-gcnv1` with the volume `testvoleasiaeast1`.

```
tridentctl import volume backend-tbc-gcnv1 "testvoleasiaeast1" -f < path-to-pvc> -n trident
```

|          |                                          |                                      |        |        |               |
|----------|------------------------------------------|--------------------------------------|--------|--------|---------------|
|          |                                          |                                      |        |        |               |
|          |                                          |                                      |        |        |               |
|          |                                          |                                      |        |        |               |
|          | NAME                                     |                                      | SIZE   |        | STORAGE CLASS |
| PROTOCOL | BACKEND UUID                             |                                      | STATE  |        | MANAGED       |
|          |                                          |                                      |        |        |               |
|          |                                          |                                      |        |        |               |
|          |                                          |                                      |        |        |               |
|          | pvc-a69cda19-218c-4ca9-a941-aea05dd13dc0 |                                      | 10 GiB |        | gcnv-nfs-sc-  |
| identity | file                                     | 8c18cdf1-0770-4bc0-bcc5-c6295fe6d837 |        | online | true          |
|          |                                          |                                      |        |        |               |

```
+-----+-----+
+-----+-----+-----+-----+
+-----+-----+
```

The following example imports a `google-cloud-netapp-volumes` volume when two volumes are present in the same region:

```
tridentctl import volume backend-tbc-gcnv1
"projects/123456789100/locations/asia-east1-a/volumes/testvoleasiaeast1"
-f <path-to-pvc> -n trident
```

|                                          | NAME                                 | SIZE         | STORAGE CLASS |
|------------------------------------------|--------------------------------------|--------------|---------------|
| PROTOCOL                                 | BACKEND UUID                         | STATE        | MANAGED       |
| pvc-a69cda19-218c-4ca9-a941-aea05dd13dc0 | 10 GiB                               | gcnv-nfs-sc- |               |
| identity   file                          | 8c18cdf1-0770-4bc0-bcc5-c6295fe6d837 | online       | true          |

## Customize Trident volume names and labels

With Trident, you can assign meaningful names and labels to volumes you create. This helps you identify and easily map volumes to their respective Kubernetes resources (PVCs). You can also define templates at the backend level for creating custom volume names and custom labels; any volumes that you create, import, or clone will adhere to the templates.

### Before you begin

Customizable volume names and labels support:

- Volume create, import, and clone operations.
- In the case of the `ontap-nas-economy` driver, only the name of the Qtree volume complies with the name template.
- In the case of the `ontap-san-economy` driver, only the LUN name complies with the name template.

## Limitations

- Custom volume names are compatible with ONTAP on-premises drivers only.
- Custom labels are supported only for the `ontap-san`, `ontap-nas`, and `ontap-nas-flexgroup` drivers.
- Custom volume names do not apply to existing volumes.

## Key behaviors of customizable volume names

- If a failure occurs due to invalid syntax in a name template, the backend creation fails. However, if the template application fails, the volume will be named according to existing naming convention.
- Storage prefix is not applicable when a volume is named using a name template from the backend configuration. Any desired prefix value may be directly added to the template.

## Backend configuration examples with name template and labels

Custom name templates can be defined at the root and/or pool level.

### Root level example

```
{
 "version": 1,
 "storageDriverName": "ontap-nas",
 "backendName": "ontap-nfs-backend",
 "managementLIF": "<ip address>",
 "svm": "svm0",
 "username": "<admin>",
 "password": "<password>",
 "defaults": {
 "nameTemplate":
 "{{.volume.Name}}_{{.labels.cluster}}_{{.volume.Namespace}}_{{.volume.RequestName}}"
 },
 "labels": {
 "cluster": "ClusterA",
 "PVC": "{{.volume.Namespace}}_{{.volume.RequestName}}"
 }
}
```

### Pool level example

```
{
 "version": 1,
 "storageDriverName": "ontap-nas",
 "backendName": "ontap-nfs-backend",
 "managementLIF": "<ip address>",
 "svm": "svm0",
 "username": "<admin>",
```

```

"password": "<password>",
"useREST": true,
"storage": [
 {
 "labels": {
 "labelname": "label1",
 "name": "{{ .volume.Name }}"
 },
 "defaults": {
 "nameTemplate": "pool01_{{ .volume.Name }}_{{ .labels.cluster }}_{{ .volume.Namespace }}_{{ .volume.RequestName }}"
 }
 },
 {
 "labels": {
 "cluster": "label2",
 "name": "{{ .volume.Name }}"
 },
 "defaults": {
 "nameTemplate": "pool02_{{ .volume.Name }}_{{ .labels.cluster }}_{{ .volume.Namespace }}_{{ .volume.RequestName }}"
 }
 }
]
}

```

## Name template examples

### Example 1:

```

"nameTemplate": "{{ .config.StoragePrefix }}_{{ .volume.Name }}_{{ .config.BackendName }}"

```

### Example 2:

```

"nameTemplate": "pool_{{ .config.StoragePrefix }}_{{ .volume.Name }}_{{ slice .volume.RequestName 1 5 }}"

```

## Points to consider

1. In the case of volume imports, the labels are updated only if the existing volume has labels in a specific format. For example: {"provisioning":{"Cluster":"ClusterA", "PVC": "pvcname"}}.
2. In the case of managed volume imports, the volume name follows the name template defined at the root level in the backend definition.

3. Trident does not support the use of a slice operator with the storage prefix.
4. If the templates do not result in unique volume names, Trident will append a few random characters to create unique volume names.
5. If the custom name for a NAS economy volume exceeds 64 characters in length, Trident will name the volumes according to the existing naming convention. For all other ONTAP drivers, if the volume name exceeds the name limit, the volume creation process fails.

## Share a Trident NFS volume across namespaces

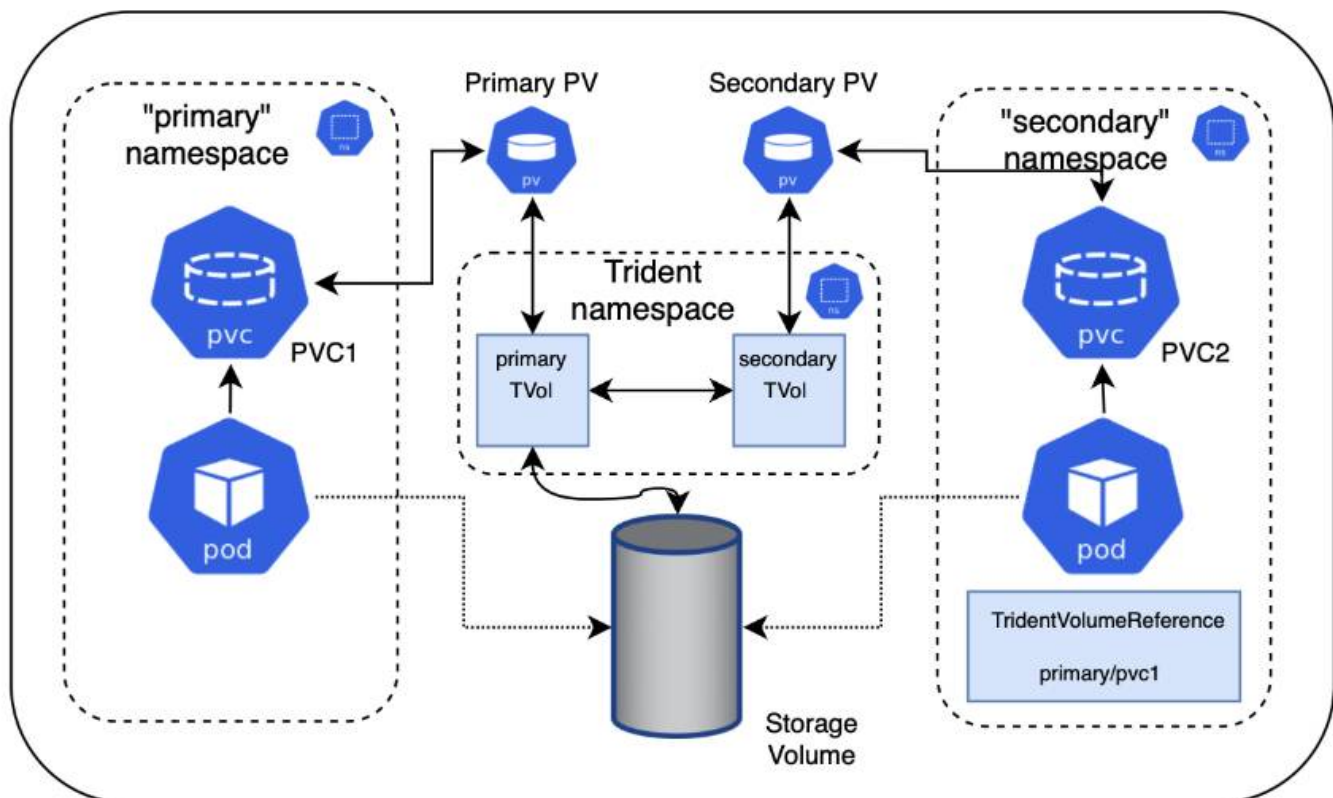
Using Trident, you can create a volume in a primary namespace and share it in one or more secondary namespaces.

### Features

The TridentVolumeReference CR allows you to securely share ReadWriteMany (RWX) NFS volumes across one or more Kubernetes namespaces. This Kubernetes-native solution has the following benefits:

- Multiple levels of access control to ensure security
- Works with all Trident NFS volume drivers
- No reliance on tridentctl or any other non-native Kubernetes feature

This diagram illustrates NFS volume sharing across two Kubernetes namespaces.



## Quick start

You can set up NFS volume sharing in just a few steps.

1

### Configure source PVC to share the volume

The source namespace owner grants permission to access the data in the source PVC.

2

### Grant permission to create a CR in the destination namespace

The cluster administrator grants permission to the owner of the destination namespace to create the `TridentVolumeReference` CR.

3

### Create `TridentVolumeReference` in the destination namespace

The owner of the destination namespace creates the `TridentVolumeReference` CR to refer to the source PVC.

4

### Create the subordinate PVC in the destination namespace

The owner of the destination namespace creates the subordinate PVC to use the data source from the source PVC.

## Configure the source and destination namespaces

To ensure security, cross namespace sharing requires collaboration and action by the source namespace owner, cluster administrator, and destination namespace owner. The user role is designated in each step.

### Steps

1. **Source namespace owner:** Create the PVC (`pvc1`) in the source namespace that grants permission to share with the destination namespace (`namespace2`) using the `shareToNamespace` annotation.

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
 name: pvc1
 namespace: namespace1
 annotations:
 trident.netapp.io/shareToNamespace: namespace2
spec:
 accessModes:
 - ReadWriteMany
 storageClassName: trident-csi
 resources:
 requests:
 storage: 100Gi
```

Trident creates the PV and its backend NFS storage volume.



- You can share the PVC to multiple namespaces using a comma-delimited list. For example, `trident.netapp.io/shareToNamespace: namespace2, namespace3, namespace4`.
- You can share to all namespaces using `*`. For example, `trident.netapp.io/shareToNamespace: *`
- You can update the PVC to include the `shareToNamespace` annotation at any time.

2. **Cluster admin:** Ensure that proper RBAC is in place to grant permission to the destination namespace owner to create the `TridentVolumeReference` CR in the destination namespace.
3. **Destination namespace owner:** Create a `TridentVolumeReference` CR in the destination namespace that refers to the source namespace `pvc1`.

```
apiVersion: trident.netapp.io/v1
kind: TridentVolumeReference
metadata:
 name: my-first-tvr
 namespace: namespace2
spec:
 pvcName: pvc1
 pvcNamespace: namespace1
```

4. **Destination namespace owner:** Create a PVC (`pvc2`) in destination namespace (`namespace2`) using the `shareFromPVC` annotation to designate the source PVC.

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
 annotations:
 trident.netapp.io/shareFromPVC: namespace1/pvc1
 name: pvc2
 namespace: namespace2
spec:
 accessModes:
 - ReadWriteMany
 storageClassName: trident-csi
 resources:
 requests:
 storage: 100Gi
```



The size of the destination PVC must be less than or equal than the source PVC.

## Results

Trident reads the `shareFromPVC` annotation on the destination PVC and creates the destination PV as a subordinate volume with no storage resource of its own that points to the source PV and shares the source PV storage resource. The destination PVC and PV appear bound as normal.

## Delete a shared volume

You can delete a volume that is shared across multiple namespaces. Trident will remove access to the volume on the source namespace and maintain access for other namespaces that share the volume. When all namespaces that reference the volume are removed, Trident deletes the volume.

## Use `tridentctl get` to query subordinate volumes

Using the `tridentctl` utility, you can run the `get` command to get subordinate volumes. For more information, refer to `tridentctl` [commands and options](#).

```
Usage:
 tridentctl get [option]
```

Flags:

- `-h, --help`: Help for volumes.
- `--parentOfSubordinate string`: Limit query to subordinate source volume.
- `--subordinateOf string`: Limit query to subordinates of volume.

## Limitations

- Trident cannot prevent destination namespaces from writing to the shared volume. You should use file locking or other processes to prevent overwriting shared volume data.
- You cannot revoke access to the source PVC by removing the `shareToNamespace` or `shareFromNamespace` annotations or deleting the `TridentVolumeReference` CR. To revoke access, you must delete the subordinate PVC.
- Snapshots, clones, and mirroring are not possible on subordinate volumes.

## For more information

To learn more about cross-namespace volume access:

- Watch the demo on [NetAppTV](#).

## Clone Trident volumes across namespaces

Using Trident, you can create new volumes using existing volumes or `volumeSnapshots` from a different namespace inside the same Kubernetes cluster.

## Prerequisites

Before cloning volumes, ensure that the source and destination backends are of the same type and have the

same storage class.



Cloning across namespaces is supported only for the `ontap-san` and `ontap-nas` storage drivers. Read-only clones are not supported.

## Quick start

You can set up volume cloning in just a few steps.

1

### Configure source PVC to clone the volume

The source namespace owner grants permission to access the data in the source PVC.

2

### Grant permission to create a CR in the destination namespace

The cluster administrator grants permission to the owner of the destination namespace to create the `TridentVolumeReference` CR.

3

### Create `TridentVolumeReference` in the destination namespace

The owner of the destination namespace creates the `TridentVolumeReference` CR to refer to the source PVC.

4

### Create the clone PVC in the destination namespace

The owner of the destination namespace creates PVC to clone the PVC from the source namespace.

## Configure the source and destination namespaces

To ensure security, cloning volumes across namespaces requires collaboration and action by the source namespace owner, cluster administrator, and destination namespace owner. The user role is designated in each step.

### Steps

1. **Source namespace owner:** Create the PVC (`pvc1`) in the source namespace (`namespace1`) that grants permission to share with the destination namespace (`namespace2`) using the `cloneToNamespace` annotation.

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
 name: pvc1
 namespace: namespace1
 annotations:
 trident.netapp.io/cloneToNamespace: namespace2
spec:
 accessModes:
 - ReadWriteMany
```

```
storageClassName: trident-csi
resources:
 requests:
 storage: 100Gi
```

Trident creates the PV and its backend storage volume.



- You can share the PVC to multiple namespaces using a comma-delimited list. For example, `trident.netapp.io/cloneToNamespace: namespace2, namespace3, namespace4`.
- You can share to all namespaces using `*`. For example, `trident.netapp.io/cloneToNamespace: *`
- You can update the PVC to include the `cloneToNamespace` annotation at any time.

2. **Cluster admin:** Ensure that proper RBAC is in place to grant permission to the destination namespace owner to create the `TridentVolumeReference` CR in the destination namespace (`namespace2`).
3. **Destination namespace owner:** Create a `TridentVolumeReference` CR in the destination namespace that refers to the source PVC `pvc1`.

```
apiVersion: trident.netapp.io/v1
kind: TridentVolumeReference
metadata:
 name: my-first-tvr
 namespace: namespace2
spec:
 pvcName: pvc1
 pvcNamespace: namespace1
```

4. **Destination namespace owner:** Create a PVC (`pvc2`) in destination namespace (`namespace2`) using the `cloneFromPVC` or `cloneFromSnapshot`, and `cloneFromNamespace` annotations to designate the source PVC.

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
 annotations:
 trident.netapp.io/cloneFromPVC: pvc1
 trident.netapp.io/cloneFromNamespace: namespace1
 name: pvc2
 namespace: namespace2
spec:
 accessModes:
 - ReadWriteMany
 storageClassName: trident-csi
```

```
resources:
 requests:
 storage: 100Gi
```

## Limitations

- For PVCs provisioned using ontap-nas-economy drivers, read-only clones are not supported.

## Replicate volumes using SnapMirror

Trident supports mirror relationships between a source volume on one cluster and the destination volume on the peered cluster for replicating data for disaster recovery. You can use a namespaced Custom Resource Definition (CRD), called Trident Mirror Relationship (TMR) to perform the following operations:

- Create mirror relationships between volumes (PVCs)
- Remove mirror relationships between volumes
- Break the mirror relationships
- Promote the secondary volume during disaster conditions (failovers)
- Perform lossless transition of applications from cluster to cluster (during planned failovers or migrations)

## Replication prerequisites

Ensure that the following prerequisites are met before you begin:

### ONTAP clusters

- **Trident:** Trident version 22.10 or later must exist on both the source and destination Kubernetes clusters that utilize ONTAP as a backend.
- **Licenses:** ONTAP SnapMirror asynchronous licenses using the Data Protection bundle must be enabled on both the source and destination ONTAP clusters. Refer to [SnapMirror licensing overview in ONTAP](#) for more information.

Beginning with ONTAP 9.10.1, all licenses are delivered as a NetApp license file (NLF), which is a single file that enables multiple features. Refer to [Licenses included with ONTAP One](#) for more information.



Only SnapMirror asynchronous protection is supported.

### Peering

- **Cluster and SVM:** The ONTAP storage backends must be peered. Refer to [Cluster and SVM peering overview](#) for more information.



Ensure that the SVM names used in the replication relationship between two ONTAP clusters are unique.

- **Trident and SVM:** The peered remote SVMs must be available to Trident on the destination cluster.

### Supported drivers

NetApp Trident supports volume replication with NetApp SnapMirror technology using storage classes backed by the following drivers:

**ontap-nas: NFS**

ontap-san: iSCSI

**ontap-san: FC**

ontap-san: NVMe/TCP (requires minimum ONTAP version 9.15.1)



Volume replication using SnapMirror is not supported for ASA r2 systems. For information about ASA r2 systems, see [Learn about ASA r2 storage systems](#).

## Create a mirrored PVC

Follow these steps and use the CRD examples to create mirror relationship between primary and secondary volumes.

### Steps

1. Perform the following steps on the primary Kubernetes cluster:
  - a. Create a StorageClass object with the `trident.netapp.io/replication: true` parameter.

#### Example

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
 name: csi-nas
provisioner: csi.trident.netapp.io
parameters:
 backendType: "ontap-nas"
 fsType: "nfs"
 trident.netapp.io/replication: "true"
```

- b. Create a PVC with previously created StorageClass.

#### Example

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
 name: csi-nas
spec:
 accessModes:
 - ReadWriteMany
 resources:
 requests:
 storage: 1Gi
 storageClassName: csi-nas
```

- c. Create a MirrorRelationship CR with local information.

#### Example

```
kind: TridentMirrorRelationship
apiVersion: trident.netapp.io/v1
metadata:
 name: csi-nas
spec:
 state: promoted
 volumeMappings:
 - localPVCName: csi-nas
```

Trident fetches the internal information for the volume and the volume's current data protection (DP) state, then populates the status field of the MirrorRelationship.

- d. Get the TridentMirrorRelationship CR to obtain the internal name and SVM of the PVC.

```
kubectl get tmr csi-nas
```

```
kind: TridentMirrorRelationship
apiVersion: trident.netapp.io/v1
metadata:
 name: csi-nas
 generation: 1
spec:
 state: promoted
 volumeMappings:
 - localPVCName: csi-nas
status:
 conditions:
 - state: promoted
 localVolumeHandle:
 "datavserver:trident_pvc_3bedd23c_46a8_4384_b12b_3c38b313c1e1"
 localPVCName: csi-nas
 observedGeneration: 1
```

2. Perform the following steps on the secondary Kubernetes cluster:

- a. Create a StorageClass with the trident.netapp.io/replication: true parameter.

#### Example

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
```

```

name: csi-nas
provisioner: csi.trident.netapp.io
parameters:
 trident.netapp.io/replication: true

```

- b. Create a MirrorRelationship CR with destination and source information.

#### Example

```

kind: TridentMirrorRelationship
apiVersion: trident.netapp.io/v1
metadata:
 name: csi-nas
spec:
 state: established
 replicationPolicy: MirrorAllSnapshots
 replicationSchedule: 5min
 volumeMappings:
 - localPVCName: csi-nas
 remoteVolumeHandle:
 "datavserver:trident_pvc_3bedd23c_46a8_4384_b12b_3c38b313cle1"

```

The TMR spec has four top-level fields: `state`, which sets the desired relationship state (see [Volume Replication States](#)); `volumeMappings`, which associates local PVCs with their remote volume handles; and the optional `replicationPolicy` and `replicationSchedule` fields described below.

The `replicationPolicy` and `replicationSchedule` fields are optional. Set `replicationPolicy` to the name of an asynchronous SnapMirror policy on ONTAP and `replicationSchedule` to the name of an ONTAP schedule. If either field is omitted, Trident uses the corresponding replication policy or schedule configured on the backend. Set these fields when you create the relationship; you cannot change the policy or schedule after the relationship is established. To use different settings, delete and recreate the TMR.

Trident creates and initializes a SnapMirror relationship using the configured replication policy and schedule.

- a. Create a PVC with previously created StorageClass to act as the secondary (SnapMirror destination).

#### Example

```

kind: PersistentVolumeClaim
apiVersion: v1
metadata:
 name: csi-nas
 annotations:
 trident.netapp.io/mirrorRelationship: csi-nas
spec:
 accessModes:
 - ReadWriteMany
resources:

```

```
requests:
 storage: 1Gi
storageClassName: csi-nas
```

Trident will check for the `TridentMirrorRelationship` CRD and fail to create the volume if the relationship does not exist. If the relationship exists, Trident will ensure the new FlexVol volume is placed onto an SVM that is peered with the remote SVM defined in the `MirrorRelationship`.

## Volume Replication States

A Trident Mirror Relationship (TMR) is a CRD that represents one end of a replication relationship between PVCs. The destination TMR has a state, which tells Trident what the desired state is. The destination TMR has the following states:

- **Established:** the local PVC is the destination volume of a mirror relationship, and this is a new relationship.
- **Promoted:** the local PVC is ReadWrite and mountable, with no mirror relationship currently in effect.
- **Reestablished:** the local PVC is the destination volume of a mirror relationship and was also previously in that mirror relationship.
  - The reestablished state must be used if the destination volume was ever in a relationship with the source volume because it overwrites the destination volume contents.
  - The reestablished state will fail if the volume was not previously in a relationship with the source.

## Promote secondary PVC during an unplanned failover

Perform the following step on the secondary Kubernetes cluster:

- Update the `spec.state` field of `TridentMirrorRelationship` to `promoted`.

## Promote secondary PVC during a planned failover

During a planned failover (migration), perform the following steps to promote the secondary PVC:

### Steps

1. On the primary Kubernetes cluster, create a snapshot of the PVC and wait until the snapshot is created.
2. On the primary Kubernetes cluster, create the `TridentTridentSnapshotInfoCR` to obtain internal details.

### Example

```
kind: TridentSnapshotInfo
apiVersion: trident.netapp.io/v1
metadata:
 name: csi-nas
spec:
 snapshotName csi-nas-snapshot
```

- a. On secondary Kubernetes cluster, update the `spec.state` field of the `TridentMirrorRelationship` CR to `promoted` and `spec.volumeMappings[].promotedSnapshotHandle` to be the `internalName` of the snapshot.

3. On secondary Kubernetes cluster, confirm the status (`status.state` field) of `TridentMirrorRelationship` to *promoted*.

## Restore a mirror relationship after a failover

Before restoring a mirror relationship, choose the side that you want to make as the new primary.

### Steps

1. On the secondary Kubernetes cluster, ensure that the values for the `spec.remoteVolumeHandle` field on the `TridentMirrorRelationship` is updated.
2. On secondary Kubernetes cluster, update the `spec.state` field of `TridentMirrorRelationship` to *reestablished*.

## Additional operations

Trident supports the following operations on the primary and secondary volumes:

### Replicate primary PVC to a new secondary PVC

Ensure that you already have a primary PVC and a secondary PVC.

### Steps

1. Delete the `PersistentVolumeClaim` and `TridentMirrorRelationship` CRDs from the established secondary (destination) cluster.
2. Delete the `TridentMirrorRelationship` CRD from the primary (source) cluster.
3. Create a new `TridentMirrorRelationship` CRD on the primary (source) cluster for the new secondary (destination) PVC you want to establish.

### Resize a mirrored, primary or secondary PVC

The PVC can be resized as normal, ONTAP will automatically expand any destination flexvols if the amount of data exceeds the current size.

### Remove replication from a PVC

To remove replication, perform one of the following operations on the current secondary volume:

- Delete the `MirrorRelationship` on the secondary PVC. This breaks the replication relationship.
- Or, update the `spec.state` field to *promoted*.

### Delete a PVC (that was previously mirrored)

Trident checks for replicated PVCs, and releases the replication relationship before attempting to delete the volume.

### Delete a TMR

Deleting a TMR on one side of a mirrored relationship causes the remaining TMR to transition to *promoted* state before Trident completes the deletion. If the TMR selected for deletion is already in *promoted* state, there is no existing mirror relationship and the TMR will be removed and Trident will promote the local PVC to *ReadWrite*. This deletion releases `SnapMirror` metadata for the local volume in ONTAP. If this volume is used in a mirror relationship in the future, it must use a new TMR with an *established* volume replication state when

creating the new mirror relationship.

## Update mirror relationships when ONTAP is online

Mirror relationships can be updated any time after they are established. You can use the `state: promoted` or `state: reestablished` fields to update the relationships. When promoting a destination volume to a regular ReadWrite volume, you can use *promotedSnapshotHandle* to specify a specific snapshot to restore the current volume to.

## Update mirror relationships when ONTAP is offline

You can use a CRD to perform a SnapMirror update without Trident having direct connectivity to the ONTAP cluster. Refer to the following example format of the `TridentActionMirrorUpdate`:

### Example

```
apiVersion: trident.netapp.io/v1
kind: TridentActionMirrorUpdate
metadata:
 name: update-mirror-b
spec:
 snapshotHandle: "pvc-1234/snapshot-1234"
 tridentMirrorRelationshipName: mirror-b
```

`status.state` reflects the state of the `TridentActionMirrorUpdate` CRD. It can take a value from *Succeeded*, *In Progress*, or *Failed*.

## Use CSI Topology

Trident can selectively create and attach volumes to nodes present in a Kubernetes cluster by making use of the [CSI Topology feature](#).

### Overview

Using the CSI Topology feature, access to volumes can be limited to a subset of nodes, based on regions and availability zones. Cloud providers today enable Kubernetes administrators to spawn nodes that are zone based. Nodes can be located in different availability zones within a region, or across various regions. To facilitate the provisioning of volumes for workloads in a multi-zone architecture, Trident uses CSI Topology.



Learn more about the CSI Topology feature [here](#).

Kubernetes provides two unique volume binding modes:

- With `VolumeBindingMode` set to `Immediate`, Trident creates the volume without any topology awareness. Volume binding and dynamic provisioning are handled when the PVC is created. This is the default `VolumeBindingMode` and is suited for clusters that do not enforce topology constraints. Persistent Volumes are created without having any dependency on the requesting pod's scheduling requirements.
- With `VolumeBindingMode` set to `WaitForFirstConsumer`, the creation and binding of a Persistent Volume for a PVC is delayed until a pod that uses the PVC is scheduled and created. This way, volumes

are created to meet the scheduling constraints that are enforced by topology requirements.



The `WaitForFirstConsumer` binding mode does not require topology labels. This can be used independent of the CSI Topology feature.

### What you'll need

To make use of CSI Topology, you need the following:

- A Kubernetes cluster running a [supported Kubernetes version](#)

```
kubectl version
Client Version: version.Info{Major:"1", Minor:"19",
GitVersion:"v1.19.3",
GitCommit:"1e11e4a2108024935ecfcb2912226cedeaafd99df",
GitTreeState:"clean", BuildDate:"2020-10-14T12:50:19Z",
GoVersion:"go1.15.2", Compiler:"gc", Platform:"linux/amd64"}
Server Version: version.Info{Major:"1", Minor:"19",
GitVersion:"v1.19.3",
GitCommit:"1e11e4a2108024935ecfcb2912226cedeaafd99df",
GitTreeState:"clean", BuildDate:"2020-10-14T12:41:49Z",
GoVersion:"go1.15.2", Compiler:"gc", Platform:"linux/amd64"}
```

- Nodes in the cluster should have labels that introduce topology awareness (topology.kubernetes.io/region and topology.kubernetes.io/zone). These labels **should be present on nodes in the cluster** before Trident is installed for Trident to be topology aware.

```
kubectl get nodes -o=jsonpath='{range .items[*]}[{.metadata.name},
{.metadata.labels}]{ "\n"}{end}' | grep --color "topology.kubernetes.io"
[node1,
{"beta.kubernetes.io/arch":"amd64","beta.kubernetes.io/os":"linux","kubernetes.io/arch":"amd64","kubernetes.io/hostname":"node1","kubernetes.io/os":"linux","node-role.kubernetes.io/master":"","topology.kubernetes.io/region":"us-east1","topology.kubernetes.io/zone":"us-east1-a"}]
[node2,
{"beta.kubernetes.io/arch":"amd64","beta.kubernetes.io/os":"linux","kubernetes.io/arch":"amd64","kubernetes.io/hostname":"node2","kubernetes.io/os":"linux","node-role.kubernetes.io/worker":"","topology.kubernetes.io/region":"us-east1","topology.kubernetes.io/zone":"us-east1-b"}]
[node3,
{"beta.kubernetes.io/arch":"amd64","beta.kubernetes.io/os":"linux","kubernetes.io/arch":"amd64","kubernetes.io/hostname":"node3","kubernetes.io/os":"linux","node-role.kubernetes.io/worker":"","topology.kubernetes.io/region":"us-
```

```
east1", "topology.kubernetes.io/zone": "us-east1-c" } }
```

## Step 1: Create a topology-aware backend

Trident storage backends can be designed to selectively provision volumes based on availability zones. Each backend can carry an optional `supportedTopologies` block that represents a list of zones and regions that are supported. For `StorageClasses` that make use of such a backend, a volume would only be created if requested by an application that is scheduled in a supported region/zone.

Here is an example backend definition:

### YAML

```

version: 1
storageDriverName: ontap-san
backendName: san-backend-us-east1
managementLIF: 192.168.27.5
svm: iscsi_svm
username: admin
password: password
supportedTopologies:
- topology.kubernetes.io/region: us-east1
 topology.kubernetes.io/zone: us-east1-a
- topology.kubernetes.io/region: us-east1
 topology.kubernetes.io/zone: us-east1-b
```

### JSON

```
{
 "version": 1,
 "storageDriverName": "ontap-san",
 "backendName": "san-backend-us-east1",
 "managementLIF": "192.168.27.5",
 "svm": "iscsi_svm",
 "username": "admin",
 "password": "password",
 "supportedTopologies": [
 {
 "topology.kubernetes.io/region": "us-east1",
 "topology.kubernetes.io/zone": "us-east1-a"
 },
 {
 "topology.kubernetes.io/region": "us-east1",
 "topology.kubernetes.io/zone": "us-east1-b"
 }
]
}
```

```
]
}
```



`supportedTopologies` is used to provide a list of regions and zones per backend. These regions and zones represent the list of permissible values that can be provided in a StorageClass. For StorageClasses that contain a subset of the regions and zones provided in a backend, Trident creates a volume on the backend.

You can define `supportedTopologies` per storage pool as well. See the following example:

```

version: 1
storageDriverName: ontap-nas
backendName: nas-backend-us-centrall
managementLIF: 172.16.238.5
svm: nfs_svm
username: admin
password: password
supportedTopologies:
 - topology.kubernetes.io/region: us-centrall
 topology.kubernetes.io/zone: us-centrall-a
 - topology.kubernetes.io/region: us-centrall
 topology.kubernetes.io/zone: us-centrall-b
storage:
 - labels:
 workload: production
 supportedTopologies:
 - topology.kubernetes.io/region: us-centrall
 topology.kubernetes.io/zone: us-centrall-a
 - labels:
 workload: dev
 supportedTopologies:
 - topology.kubernetes.io/region: us-centrall
 topology.kubernetes.io/zone: us-centrall-b
```

In this example, the region and zone labels stand for the location of the storage pool.

`topology.kubernetes.io/region` and `topology.kubernetes.io/zone` dictate where the storage pools can be consumed from.

## Step 2: Define StorageClasses that are topology aware

Based on the topology labels that are provided to the nodes in the cluster, StorageClasses can be defined to contain topology information. This will determine the storage pools that serve as candidates for PVC requests made, and the subset of nodes that can make use of the volumes provisioned by Trident.

See the following example:

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
 name: netapp-san-us-east1
provisioner: csi.trident.netapp.io
volumeBindingMode: WaitForFirstConsumer
allowedTopologies:
 - matchLabelExpressions:
 - key: topology.kubernetes.io/zone
 values:
 - us-east1-a
 - us-east1-b
 - key: topology.kubernetes.io/region
 values:
 - us-east1
parameters:
 fsType: ext4
```

In the StorageClass definition provided above, `volumeBindingMode` is set to `WaitForFirstConsumer`. PVCs that are requested with this StorageClass will not be acted upon until they are referenced in a pod. And, `allowedTopologies` provides the zones and region to be used. The `netapp-san-us-east1` StorageClass creates PVCs on the `san-backend-us-east1` backend defined above.

### Step 3: Create and use a PVC

With the StorageClass created and mapped to a backend, you can now create PVCs.

See the example spec below:

```

kind: PersistentVolumeClaim
apiVersion: v1
metadata: null
name: pvc-san
spec: null
accessModes:
 - ReadWriteOnce
resources:
 requests:
 storage: 300Mi
storageClassName: netapp-san-us-east1
```

Creating a PVC using this manifest would result in the following:

```
kubectl create -f pvc.yaml
persistentvolumeclaim/pvc-san created
kubectl get pvc
NAME STATUS VOLUME CAPACITY ACCESS MODES STORAGECLASS
AGE
pvc-san Pending netapp-san-us-east1
2s
kubectl describe pvc
Name: pvc-san
Namespace: default
StorageClass: netapp-san-us-east1
Status: Pending
Volume:
Labels: <none>
Annotations: <none>
Finalizers: [kubernetes.io/pvc-protection]
Capacity:
Access Modes:
VolumeMode: Filesystem
Mounted By: <none>
Events:
 Type Reason Age From Message
 ---- -
Normal WaitForFirstConsumer 6s persistentvolume-controller waiting
for first consumer to be created before binding
```

For Trident to create a volume and bind it to the PVC, use the PVC in a pod. See the following example:

```
apiVersion: v1
kind: Pod
metadata:
 name: app-pod-1
spec:
 affinity:
 nodeAffinity:
 requiredDuringSchedulingIgnoredDuringExecution:
 nodeSelectorTerms:
 - matchExpressions:
 - key: topology.kubernetes.io/region
 operator: In
 values:
 - us-east1
 preferredDuringSchedulingIgnoredDuringExecution:
 - weight: 1
 preference:
```

```

 matchExpressions:
 - key: topology.kubernetes.io/zone
 operator: In
 values:
 - us-east1-a
 - us-east1-b
 securityContext:
 runAsUser: 1000
 runAsGroup: 3000
 fsGroup: 2000
 volumes:
 - name: voll
 persistentVolumeClaim:
 claimName: pvc-san
 containers:
 - name: sec-ctx-demo
 image: busybox
 command: ["sh", "-c", "sleep 1h"]
 volumeMounts:
 - name: voll
 mountPath: /data/demo
 securityContext:
 allowPrivilegeEscalation: false

```

This podSpec instructs Kubernetes to schedule the pod on nodes that are present in the us-east1 region, and choose from any node that is present in the us-east1-a or us-east1-b zones.

See the following output:

```

kubectl get pods -o wide
NAME READY STATUS RESTARTS AGE IP NODE
NOMINATED NODE READINESS GATES
app-pod-1 1/1 Running 0 19s 192.168.25.131 node2
<none> <none>
kubectl get pvc -o wide
NAME STATUS VOLUME CAPACITY
ACCESS MODES STORAGECLASS AGE VOLUMEMODE
pvc-san Bound pvc-ecb1e1a0-840c-463b-8b65-b3d033e2e62b 300Mi
RWO netapp-san-us-east1 48s Filesystem

```

## Update backends to include supportedTopologies

Pre-existing backends can be updated to include a list of supportedTopologies using tridentctl backend update. This will not affect volumes that have already been provisioned, and will only be used for subsequent PVCs.

## Find more information

- [Manage resources for containers](#)
- [nodeSelector](#)
- [Affinity and anti-affinity](#)
- [Taints and Tolerations](#)

## Work with Trident snapshots

Kubernetes volume snapshots of Persistent Volumes (PVs) enable point-in-time copies of volumes. You can create a snapshot of a volume created using Trident, import a snapshot created outside of Trident, create a new volume from an existing snapshot, and recover volume data from snapshots.

### Overview

Volume snapshot is supported by `ontap-nas`, `ontap-nas-flexgroup`, `ontap-san`, `ontap-san-economy`, `solidfire-san`, `azure-netapp-files`, and `google-cloud-netapp-volumes` drivers.

#### Before you begin

You must have an external snapshot controller and Custom Resource Definitions (CRDs) to work with snapshots. This is the responsibility of the Kubernetes orchestrator (for example: Kubeadm, GKE, OpenShift).

If your Kubernetes distribution does not include the snapshot controller and CRDs, refer to [Deploy a volume snapshot controller](#).



Don't create a snapshot controller if creating on-demand volume snapshots in a GKE environment. GKE uses a built-in, hidden snapshot controller.

## Create a volume snapshot

### Steps

1. Create a `VolumeSnapshotClass`. For more information, refer to [VolumeSnapshotClass](#).
  - The `driver` points to the Trident CSI driver.
  - `deletionPolicy` can be `Delete` or `Retain`. When set to `Retain`, the underlying physical snapshot on the storage cluster is retained even when the `VolumeSnapshot` object is deleted.

### Example

```
cat snap-sc.yaml
```

```
apiVersion: snapshot.storage.k8s.io/v1
kind: VolumeSnapshotClass
metadata:
 name: csi-snapclass
driver: csi.trident.netapp.io
```

```
deletionPolicy: Delete
```

## 2. Create a snapshot of an existing PVC.

### Examples

- This example creates a snapshot of an existing PVC.

```
cat snap.yaml
```

```
apiVersion: snapshot.storage.k8s.io/v1
kind: VolumeSnapshot
metadata:
 name: pvc1-snap
spec:
 volumeSnapshotClassName: csi-snapclass
 source:
 persistentVolumeClaimName: pvc1
```

- This example creates a volume snapshot object for a PVC named `pvc1` and the name of the snapshot is set to `pvc1-snap`. A `VolumeSnapshot` is analogous to a PVC and is associated with a `VolumeSnapshotContent` object that represents the actual snapshot.

```
kubectl create -f snap.yaml
volumesnapshot.snapshot.storage.k8s.io/pvc1-snap created

kubectl get volumesnapshots
NAME AGE
pvc1-snap 50s
```

- You can identify the `VolumeSnapshotContent` object for the `pvc1-snap` `VolumeSnapshot` by describing it. The `Snapshot Content Name` identifies the `VolumeSnapshotContent` object which serves this snapshot. The `Ready To Use` parameter indicates that the snapshot can be used to create a new PVC.

```
kubectl describe volumesnapshots pvc1-snap
Name: pvc1-snap
Namespace: default
...
Spec:
 Snapshot Class Name: pvc1-snap
 Snapshot Content Name: snapcontent-e8d8a0ca-9826-11e9-9807-
525400f3f660
 Source:
```

```
API Group:
Kind: PersistentVolumeClaim
Name: pvc1
Status:
 Creation Time: 2019-06-26T15:27:29Z
 Ready To Use: true
 Restore Size: 3Gi
 ...
```

## Create a PVC from a volume snapshot

You can use `dataSource` to create a PVC using a `VolumeSnapshot` named `<pvc-name>` as the source of the data. After the PVC is created, it can be attached to a pod and used just like any other PVC.



The PVC will be created in the same backend as the source volume. Refer to [KB: Creating a PVC from a Trident PVC Snapshot cannot be created in an alternate backend](#).

The following example creates the PVC using `pvc1-snap` as the data source.

```
cat pvc-from-snap.yaml
```

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
 name: pvc-from-snap
spec:
 accessModes:
 - ReadWriteOnce
 storageClassName: golden
 resources:
 requests:
 storage: 3Gi
 dataSource:
 name: pvc1-snap
 kind: VolumeSnapshot
 apiGroup: snapshot.storage.k8s.io
```

## Import a volume snapshot

Trident supports the [Kubernetes pre-provisioned snapshot process](#) to enable the cluster administrator to create a `VolumeSnapshotContent` object and import snapshots created outside of Trident.

### Before you begin

Trident must have created or imported the snapshot's parent volume.

## Steps

1. **Cluster admin:** Create a `VolumeSnapshotContent` object that references the backend snapshot. This initiates the snapshot workflow in Trident.
  - Specify the name of the backend snapshot in annotations as `trident.netapp.io/internalSnapshotName: <"backend-snapshot-name">`.
  - Specify `<name-of-parent-volume-in-trident>/<volume-snapshot-content-name>` in `snapshotHandle`. This is the only information provided to Trident by the external snapshotter in the `ListSnapshots` call.



The `<volumeSnapshotContentName>` cannot always match the backend snapshot name due to CR naming constraints.

## Example

The following example creates a `VolumeSnapshotContent` object that references backend snapshot `snap-01`.

```
apiVersion: snapshot.storage.k8s.io/v1
kind: VolumeSnapshotContent
metadata:
 name: import-snap-content
 annotations:
 trident.netapp.io/internalSnapshotName: "snap-01" # This is the
name of the snapshot on the backend
spec:
 deletionPolicy: Retain
 driver: csi.trident.netapp.io
 source:
 snapshotHandle: pvc-f71223b5-23b9-4235-bbfe-e269ac7b84b0/import-
snap-content # <import PV name or source PV name>/<volume-snapshot-
content-name>
 volumeSnapshotRef:
 name: import-snap
 namespace: default
```

2. **Cluster admin:** Create the `VolumeSnapshot` CR that references the `VolumeSnapshotContent` object. This requests access to use the `VolumeSnapshot` in a given namespace.

## Example

The following example creates a `VolumeSnapshot` CR named `import-snap` that references the `VolumeSnapshotContent` named `import-snap-content`.

```
apiVersion: snapshot.storage.k8s.io/v1
kind: VolumeSnapshot
metadata:
 name: import-snap
```

```
spec:
 # volumeSnapshotClassName: csi-snapclass (not required for pre-
 # provisioned or imported snapshots)
 source:
 volumeSnapshotContentName: import-snap-content
```

3. **Internal processing (no action required):** The external snapshotter recognizes the newly created `VolumeSnapshotContent` and runs the `ListSnapshots` call. Trident creates the `TridentSnapshot`.
  - The external snapshotter sets the `VolumeSnapshotContent` to `readyToUse` and the `VolumeSnapshot` to `true`.
  - Trident returns `readyToUse=true`.
4. **Any user:** Create a `PersistentVolumeClaim` to reference the new `VolumeSnapshot`, where the `spec.dataSource` (or `spec.dataSourceRef`) name is the `VolumeSnapshot` name.

### Example

The following example creates a PVC referencing the `VolumeSnapshot` named `import-snap`.

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
 name: pvc-from-snap
spec:
 accessModes:
 - ReadWriteOnce
 storageClassName: simple-sc
 resources:
 requests:
 storage: 1Gi
 dataSource:
 name: import-snap
 kind: VolumeSnapshot
 apiGroup: snapshot.storage.k8s.io
```

## Recover volume data using snapshots

The snapshot directory is hidden by default to facilitate maximum compatibility of volumes provisioned using the `ontap-nas` and `ontap-nas-economy` drivers. Enable the `.snapshot` directory to recover data from snapshots directly.

Use the volume snapshot restore ONTAP CLI to restore a volume to a state recorded in a prior snapshot.

```
cluster1::*> volume snapshot restore -vserver vs0 -volume vol3 -snapshot
vol3_snap_archive
```



When you restore a snapshot copy, the existing volume configuration is overwritten. Changes made to volume data after the snapshot copy was created are lost.

## In-place volume restoration from a snapshot

Trident provides rapid, in-place volume restoration from a snapshot using the `TridentActionSnapshotRestore` (TASR) CR. This CR functions as an imperative Kubernetes action and does not persist after the operation completes.

Trident supports snapshot restore on the `ontap-san`, `ontap-san-economy`, `ontap-nas`, `ontap-nas-flexgroup`, `azure-netapp-files`, `google-cloud-netapp-volumes`, and `solidfire-san` drivers.

### Before you begin

You must have a bound PVC and available volume snapshot.

- Verify the PVC status is bound.

```
kubectl get pvc
```

- Verify the volume snapshot is ready to use.

```
kubectl get vs
```

### Steps

1. Create the TASR CR. This example creates a CR for PVC `pvc1` and volume snapshot `pvc1-snapshot`.



The TASR CR must be in a namespace where the PVC & VS exist.

```
cat tasr-pvc1-snapshot.yaml
```

```
apiVersion: trident.netapp.io/v1
kind: TridentActionSnapshotRestore
metadata:
 name: trident-snap
 namespace: trident
spec:
 pvcName: pvc1
 volumeSnapshotName: pvc1-snapshot
```

2. Apply the CR to restore from the snapshot. This example restores from snapshot `pvc1`.

```
kubectl create -f tasr-pvc1-snapshot.yaml
```

```
tridentactionsnapshotrestore.trident.netapp.io/trident-snap created
```

## Results

Trident restores the data from the snapshot. You can verify the snapshot restore status:

```
kubectl get tasr -o yaml
```

```
apiVersion: trident.netapp.io/v1
items:
- apiVersion: trident.netapp.io/v1
 kind: TridentActionSnapshotRestore
 metadata:
 creationTimestamp: "2023-04-14T00:20:33Z"
 generation: 3
 name: trident-snap
 namespace: trident
 resourceVersion: "3453847"
 uid: <uid>
 spec:
 pvcName: pvc1
 volumeSnapshotName: pvc1-snapshot
 status:
 startTime: "2023-04-14T00:20:34Z"
 completionTime: "2023-04-14T00:20:37Z"
 state: Succeeded
kind: List
metadata:
 resourceVersion: ""
```



- In most cases, Trident will not automatically retry the operation in case of failure. You will need to perform the operation again.
- Kubernetes users without admin access might have to be granted permission by the admin to create a TASR CR in their application namespace.

## Delete a PV with associated snapshots

When deleting a Persistent Volume with associated snapshots, the corresponding Trident volume is updated to a "Deleting state". Remove the volume snapshots to delete the Trident volume.

## Deploy a volume snapshot controller

If your Kubernetes distribution does not include the snapshot controller and CRDs, you can deploy them as follows.

## Steps

### 1. Create volume snapshot CRDs.

```
cat snapshot-setup.sh
```

```
#!/bin/bash
Create volume snapshot CRDs
kubectl apply -f https://raw.githubusercontent.com/kubernetes-csi/external-snapshotter/release-6.1/client/config/crd/snapshot.storage.k8s.io_volumesnapshotclasses.yaml
kubectl apply -f https://raw.githubusercontent.com/kubernetes-csi/external-snapshotter/release-6.1/client/config/crd/snapshot.storage.k8s.io_volumesnapshotcontents.yaml
kubectl apply -f https://raw.githubusercontent.com/kubernetes-csi/external-snapshotter/release-6.1/client/config/crd/snapshot.storage.k8s.io_volumesnapshots.yaml
```

### 2. Create the snapshot controller.

```
kubectl apply -f https://raw.githubusercontent.com/kubernetes-csi/external-snapshotter/release-6.1/deploy/kubernetes/snapshot-controller/rbac-snapshot-controller.yaml
```

```
kubectl apply -f https://raw.githubusercontent.com/kubernetes-csi/external-snapshotter/release-6.1/deploy/kubernetes/snapshot-controller/setup-snapshot-controller.yaml
```



If necessary, open `deploy/kubernetes/snapshot-controller/rbac-snapshot-controller.yaml` and update namespace to your namespace.

## Related links

- [Volume snapshots](#)
- [VolumeSnapshotClass](#)

## Work with Trident volume group snapshots

Kubernetes volume group snapshots of Persistent Volumes (PVs) NetApp Trident provides the ability to create snapshots of multiple volumes ( a group of volume snapshots). This volume group snapshot represents copies from multiple volumes that

are taken at the same point-in-time.



VolumeGroupSnapshot is a beta feature in Kubernetes with beta APIs. Kubernetes 1.32 is the minimum version required for VolumeGroupSnapshot.

## Create volume group snapshots

Volume group snapshot is supported with the following storage drivers:

- `ontap-san` driver - only for the iSCSI and FC protocols, not for the NVMe/TCP protocol.
- `ontap-san-economy` - only for the iSCSI protocol.
- `ontap-nas`



Volume group snapshot is not supported for NetApp ASA r2 or AFX storage systems.

### Before you begin

- Ensure that your Kubernetes version is K8s 1.32 or higher.
- You must have an external snapshot controller and Custom Resource Definitions (CRDs) to work with snapshots. This is the responsibility of the Kubernetes orchestrator (for example: Kubeadm, GKE, OpenShift).

If your Kubernetes distribution does not include the external snapshot controller and CRDs, refer to [Deploy a volume snapshot controller](#).



Don't create a snapshot controller if creating on-demand volume group snapshots in a GKE environment. GKE uses a built-in, hidden snapshot controller.

- In the snapshot controller YAML, set the `CSIVolumeGroupSnapshot` feature gate to 'true' to ensure that volume group snapshot is enabled.
- Create the required volume group snapshot classes before creating a volume group snapshot.
- Ensure that all PVCs/volumes are on the same SVM to be able to create VolumeGroupSnapshot.

### Steps

- Create a VolumeGroupSnapshotClass prior to creating a VolumeGroupSnapshot. For more information, refer to [VolumeGroupSnapshotClass](#).

```
apiVersion: groupsnapshot.storage.k8s.io/v1beta1
kind: VolumeGroupSnapshotClass
metadata:
 name: csi-group-snap-class
 annotations:
 kubernetes.io/description: "Trident group snapshot class"
driver: csi.trident.netapp.io
deletionPolicy: Delete
```

- Create PVCs with required labels using existing storage classes, or add these labels to existing PVCs.

The following example creates the PVC using `pvc1-group-snap` as the data source and label `consistentGroupSnapshot: groupA`. Define the label key and value based on your requirements.

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
 name: pvc1-group-snap
 labels:
 consistentGroupSnapshot: groupA
spec:
 accessModes:
 - ReadWriteOnce
 resources:
 requests:
 storage: 100Mi
 storageClassName: sc1-1
```

- Create a VolumeGroupSnapshot with the same label (`consistentGroupSnapshot: groupA`) specified in the PVC.

This example creates a volume group snapshot:

```
apiVersion: groupsnapshot.storage.k8s.io/v1beta1
kind: VolumeGroupSnapshot
metadata:
 name: "vgs1"
 namespace: trident
spec:
 volumeGroupSnapshotClassName: csi-group-snap-class
 source:
 selector:
 matchLabels:
 consistentGroupSnapshot: groupA
```

## Recover volume data using a group snapshot

You can restore individual Persistent Volumes using the individual snapshots which have been created as part of the Volume Group Snapshot. You cannot recover the Volume Group Snapshot as a unit.

Use the volume snapshot restore ONTAP CLI to restore a volume to a state recorded in a prior snapshot.

```
cluster1::*> volume snapshot restore -vserver vs0 -volume vol3 -snapshot
vol3_snap_archive
```



When you restore a snapshot copy, the existing volume configuration is overwritten. Changes made to volume data after the snapshot copy was created are lost.

## In-place volume restoration from a snapshot

Trident provides rapid, in-place volume restoration from a snapshot using the `TridentActionSnapshotRestore` (TASR) CR. This CR functions as an imperative Kubernetes action and does not persist after the operation completes.

For more information, see [In-place volume restoration from a snapshot](#).

## Delete a PV with associated group snapshots

When deleting a group volume snapshot:

- You can delete `VolumeGroupSnapshots` as a whole, not individual snapshots in the group.
- If `PersistentVolumes` are deleted while a snapshot exists for that `PersistentVolume`, Trident will move that volume to a "deleting" state because the snapshot must be removed before the volume can be safely removed.
- If a clone has been created using a grouped snapshot and then the group is to be deleted, a split-on-clone operation will begin and the group cannot be deleted until the split is complete.

## Deploy a volume snapshot controller

If your Kubernetes distribution does not include the snapshot controller and CRDs, you can deploy them as follows.

### Steps

1. Create volume snapshot CRDs.

```
cat snapshot-setup.sh
```

```
#!/bin/bash
Create volume snapshot CRDs
kubectl apply -f https://raw.githubusercontent.com/kubernetes-csi/external-snapshotter/release-8.2/client/config/crd/groupsnapshot.storage.k8s.io_volumegroupsnapshotclasses.yaml
kubectl apply -f https://raw.githubusercontent.com/kubernetes-csi/external-snapshotter/release-8.2/client/config/crd/groupsnapshot.storage.k8s.io_volumegroupsnapshotcontents.yaml
kubectl apply -f https://raw.githubusercontent.com/kubernetes-csi/external-snapshotter/release-8.2/client/config/crd/groupsnapshot.storage.k8s.io_volumegroupsnapshots.yaml
```

## 2. Create the snapshot controller.

```
kubectl apply -f https://raw.githubusercontent.com/kubernetes-csi/external-snapshotter/release-8.2/deploy/kubernetes/snapshot-controller/rbac-snapshot-controller.yaml
```

```
kubectl apply -f https://raw.githubusercontent.com/kubernetes-csi/external-snapshotter/release-8.2/deploy/kubernetes/snapshot-controller/setup-snapshot-controller.yaml
```



If necessary, open `deploy/kubernetes/snapshot-controller/rbac-snapshot-controller.yaml` and update namespace to your namespace.

## Related links

- [VolumeGroupSnapshotClass](#)
- [Volume snapshots](#)

## Copyright information

Copyright © 2026 NetApp, Inc. All Rights Reserved. Printed in the U.S. No part of this document covered by copyright may be reproduced in any form or by any means—graphic, electronic, or mechanical, including photocopying, recording, taping, or storage in an electronic retrieval system—without prior written permission of the copyright owner.

Software derived from copyrighted NetApp material is subject to the following license and disclaimer:

THIS SOFTWARE IS PROVIDED BY NETAPP “AS IS” AND WITHOUT ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, WHICH ARE HEREBY DISCLAIMED. IN NO EVENT SHALL NETAPP BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

NetApp reserves the right to change any products described herein at any time, and without notice. NetApp assumes no responsibility or liability arising from the use of products described herein, except as expressly agreed to in writing by NetApp. The use or purchase of this product does not convey a license under any patent rights, trademark rights, or any other intellectual property rights of NetApp.

The product described in this manual may be protected by one or more U.S. patents, foreign patents, or pending applications.

LIMITED RIGHTS LEGEND: Use, duplication, or disclosure by the government is subject to restrictions as set forth in subparagraph (b)(3) of the Rights in Technical Data -Noncommercial Items at DFARS 252.227-7013 (FEB 2014) and FAR 52.227-19 (DEC 2007).

Data contained herein pertains to a commercial product and/or commercial service (as defined in FAR 2.101) and is proprietary to NetApp, Inc. All NetApp technical data and computer software provided under this Agreement is commercial in nature and developed solely at private expense. The U.S. Government has a non-exclusive, non-transferrable, nonsublicensable, worldwide, limited irrevocable license to use the Data only in connection with and in support of the U.S. Government contract under which the Data was delivered. Except as provided herein, the Data may not be used, disclosed, reproduced, modified, performed, or displayed without the prior written approval of NetApp, Inc. United States Government license rights for the Department of Defense are limited to those rights identified in DFARS clause 252.227-7015(b) (FEB 2014).

## Trademark information

NETAPP, the NETAPP logo, and the marks listed at <http://www.netapp.com/TM> are trademarks of NetApp, Inc. Other company and product names may be trademarks of their respective owners.