

# Container Orchestration Using Kubernetes



**Storage**



# Learning Objectives

By the end of this lesson, you will be able to:

- Analyze the role of storage in Kubernetes for optimizing application performance and data persistence
- Identify and distinguish the various types of volumes in Kubernetes, highlighting their specific use cases
- Interpret the function and significance of storage classes within Kubernetes storage management
- Demonstrate an understanding of dynamic volume provisioning and its role in efficient storage allocation
- Classify the different types of ephemeral volumes and elucidate their benefits and applications in temporary data storage





# Overview of Storage in Kubernetes

# Persistent Storage

Kubernetes has become crucial for hosting microservice-based processes and storing data due to its persistent storage capabilities.

Users can create and access databases, as well as store data for various applications.

Administrators can map service quality levels to different storage classes using StorageClass.



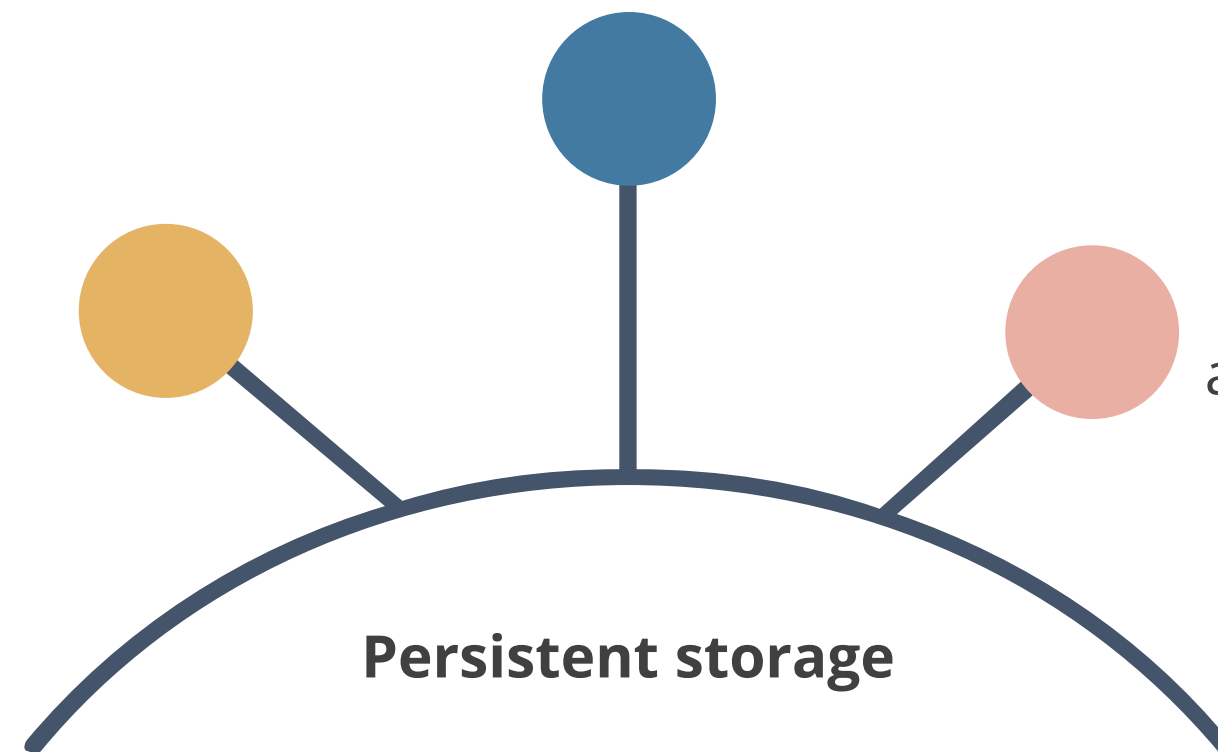
Users can implement arbitrary and backup policies that cluster administrators have assigned.

# Kubernetes and Persistent Storage

Applications must remain functional and retain their original state.

Without Kubernetes, deleting localized data disrupts information sharing among applications within the same category.

Applications must remain functional and retain their original state.



Kubernetes enables data storage outside of containers, allowing uninterrupted access.

# Requirements for Persistent Storage

These elements are essential for persistent storage:



# Backend

Kubernetes persistent storage conceals the data from both applications and users, utilizing protocols such as NFS, iSCSI, and SMB.

01

Storage services and systems on cloud platforms broaden access to user information.

02

Third-party cloud storage creates environments that grant users complete access to their data, facilitating integration.

03

Storage providers, including Amazon S3 and LINBIT, offer an array of tools and applications that support persistent storage.



## Quick Check

Your team is developing a containerized application that handles customer transactions and stores critical data. The application must remain functional and preserve its state even if a container is terminated or rescheduled. What solution should your team implement to ensure uninterrupted data access and preservation of the application state?

- A. Store all data within the container
- B. Use local storage on the host machine where the container runs
- C. Utilize Kubernetes persistent storage to store data outside of containers
- D. Implement a file-based cache within each container






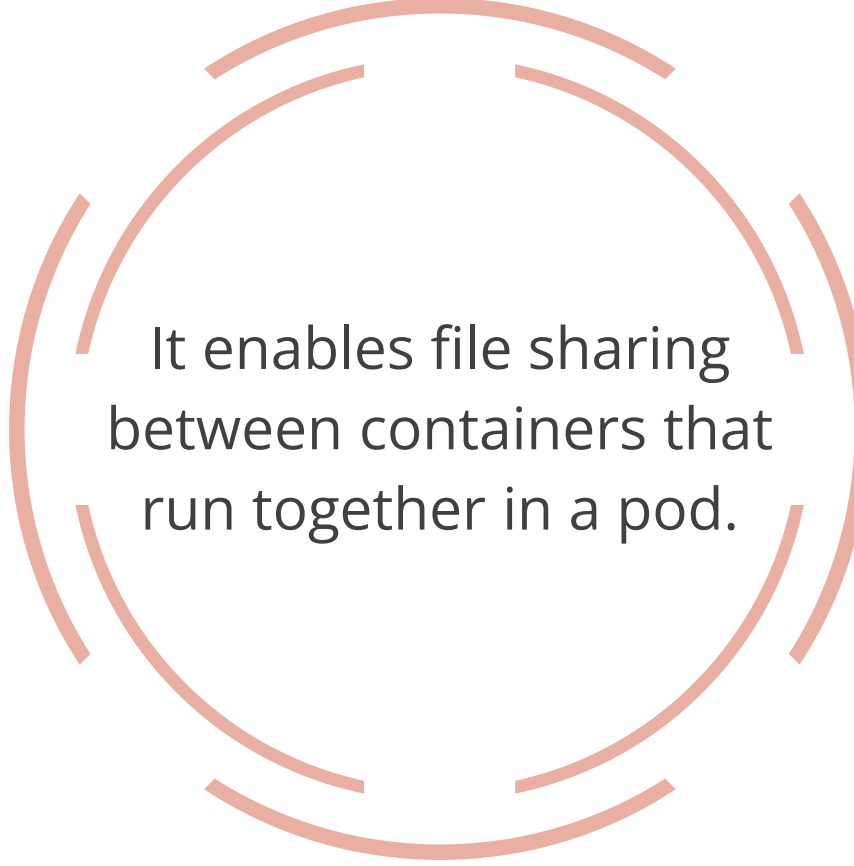
# Volumes

# Overview of Volumes

The volume abstraction in Kubernetes helps business-critical applications running in containers to address challenges.



It prevents data and file loss when a container crashes.



It enables file sharing between containers that run together in a pod.

# Background of Volumes

A pod in Kubernetes can utilize numerous volumes and volume types concurrently.

A volume's lifespan exceeds that of any container running within the pod, ensuring data preservation across container restarts or terminations.

A user should specify the volume in **.spec.volumes** for the pod and declare the necessary volume mounts in **.spec.containers[\*].volumeMounts** for the containers.

# Types of Volumes

Kubernetes supports a variety of volume types, such as:

awsElasticBlockStore

azureDisk

azureFile

cephfs

cinder

ConfigMap

downwardAPI

emptyDir

fibre channel

gcePersistentDisk

glusterfs

hostPath

# Types of Volumes

Kubernetes supports a variety of volume types, such as:

iscsi

local

nfs

PV

PVC

portworxVolume

projected

quobyte

rbd

scaleIO  
(deprecated)

storageOS

vsphereVolume

## Quick Check

You are running an application in Kubernetes where you need to ensure that certain data persists across container restarts or terminations. What step should you take to achieve this in your pod specification?

- A. Store data directly within each container
- B. Define volumes in the **.spec.volumes** field and mount them in **.spec.containers[\*].volumeMounts**
- C. Use environment variables to store data
- D. Enable auto-scaling for persistent storage



The background features abstract geometric shapes in blue and light blue. A large blue shape is in the top-left corner, and several light blue shapes are scattered across the top and left edges. The text "Ephemeral Volumes" is centered in the middle of the page.

# Ephemeral Volumes



# Overview of Ephemeral Volumes

Caching services, which often operate with limited memory, can transfer infrequently used data to storage.

Ephemeral volumes, designed for specific use cases, allow for specification directly within the pod spec.

This simplicity streamlines application deployment and management.

# Ephemeral Volumes

They cater to applications that require additional temporary storage without needing data persistence across restarts.

Types of ephemeral volumes are:

1

emptyDir

2

hostPath

5

Generic ephemeral volumes

3

ConfigMap

4

Secret

6

CSI ephemeral volumes

# emptyDir

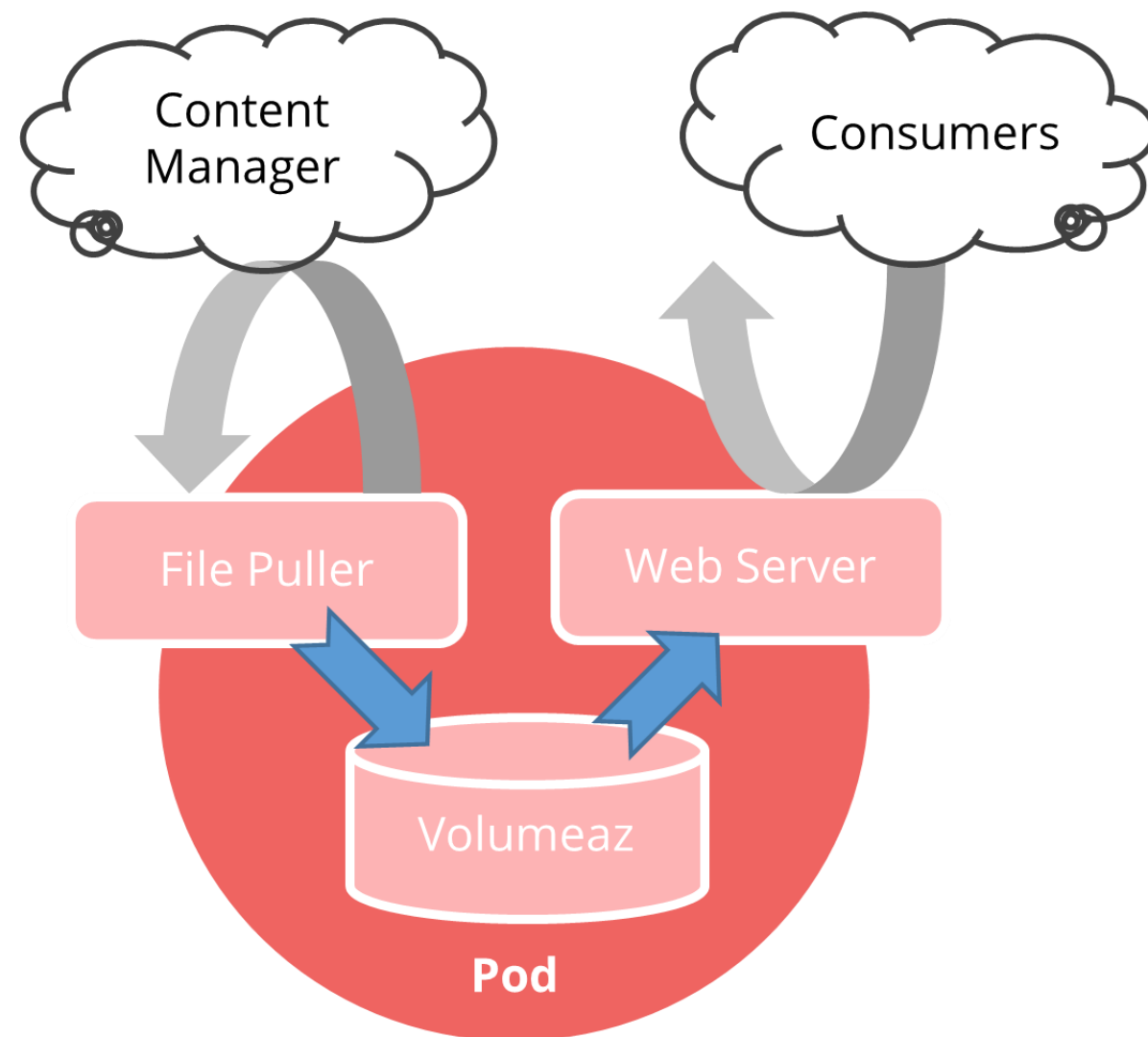
Kubernetes creates an **emptyDir** volume whenever it assigns a pod to a node. The **emptyDir** volume exists if the pod runs on that node.

## Uses of emptyDir:

- It provides a scratch space for operations like a disk-based merge sort.
- It stores checkpoints for long computations to aid recovery from crashes.
- It holds files, serving as a bridge to share content between containers within a pod.

# emptyDir

Pods are designed to support multiple cooperating processes, encapsulated as containers, that together provide a cohesive unit of service.



One container might function as a web server for files in a shared volume, such as **emptyDir**, while a separate sidecar container could update those files from a remote source.

# emptyDir: Example

The following example demonstrates how to create a pod definition using an emptyDir volume configuration:

## Example

```
apiVersion: v1
kind: Pod
metadata:
  name: test-pd
spec:
  containers:
  - image: registry.k8s.io/test-webserver
    name: test-container
    volumeMounts:
    - mountPath: /cache
      name: cache-volume
  volumes:
  - name: cache-volume
    emptyDir:
      sizeLimit: 500Mi
```

# hostPath

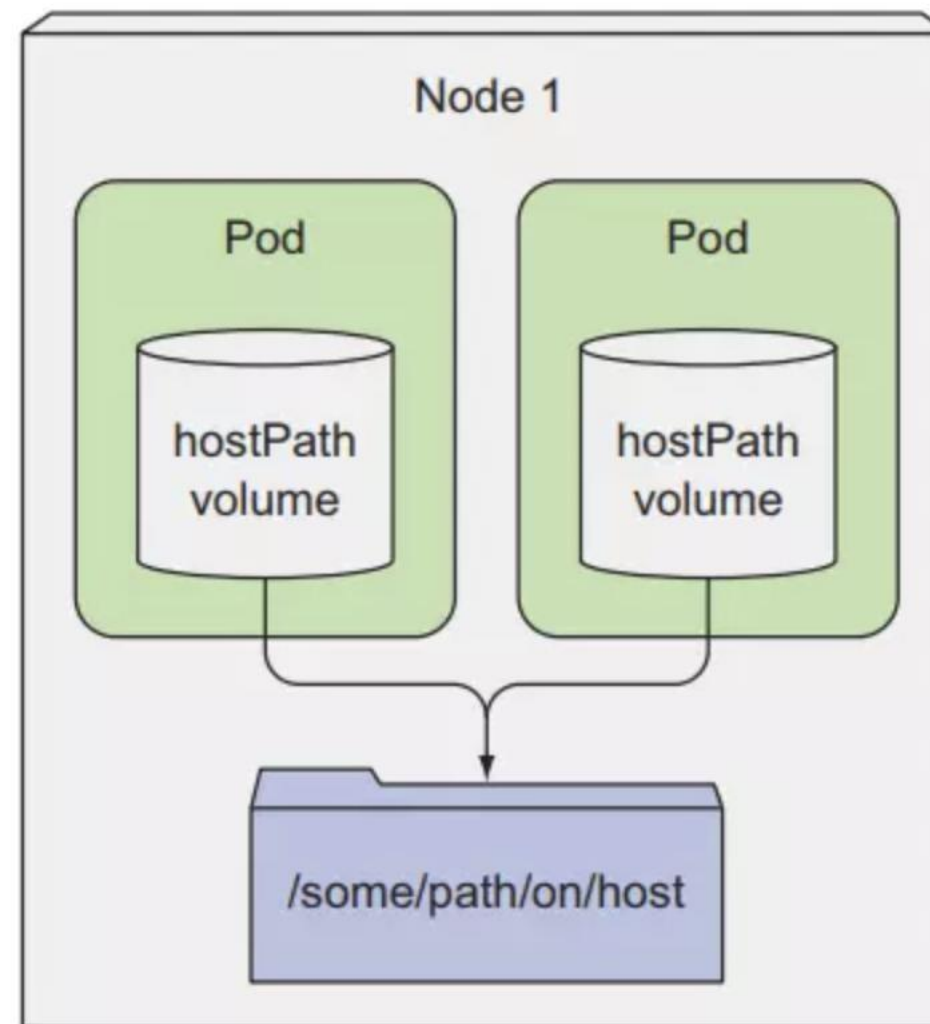
A hostPath volume mounts a specific file or directory from the host node's filesystem into the pod.

## Uses of hostPath:

- It provides a container access to the internals of the container runtime.
- A user can use a hostPath of **/sys** for running cAdvisor in a container.
- It allows a pod to specify the prerequisites of a given hostPath, determining whether it should exist before the pod runs and dictating its required attributes if it needs creation.

# hostPath

Multiple pods on the same host can share the same hostPath.



## hostPath: Example

The following example demonstrates configuring a Kubernetes pod to use a hostPath volume, enabling access to the host's **/var/log** directory directly from the container:

```
apiVersion: v1
kind: Pod
metadata:
  name: test-hostpath
spec:
  containers:
  - image: registry.k8s.io/test-webserver
    name: test-container
    volumeMounts:
    - mountPath: /log
      name: log-volume
  volumes:
  - name: log-volume
    hostPath:
      path: /var/log
```



# ConfigMap

**ConfigMap** injects configuration data into pods.

It is possible to configure the path that is to be used for a specific entry in the **ConfigMap**.

Data that is stored in a **ConfigMap** can be referenced in a Volume of type **ConfigMap**. It can then be consumed by a pod's containerized applications.

# ConfigMap: Example

The following example demonstrates how to configure a Kubernetes pod to utilize a **ConfigMap** for injecting configuration data, specifically demonstrating the mounting of log level settings into the pod:

```
apiVersion: v1
kind: Pod
metadata:
  name: ConfigMap-pod
spec:
  containers:
  - name: test
    image: busybox:1.28
    command: ['sh', '-c', 'echo "The app is running!"
&& tail -f /dev/null']
    volumeMounts:
    - name: config-vol
      mountPath: /etc/config
  volumes:
  - name: config-vol
    ConfigMap:
      name: log-config
      items:
      - key: log_level
        path: log_level
```

# Secret

A secret volume securely conveys sensitive information, such as passwords, to pods.

One can store secrets in the Kubernetes API, and pods can then mount them as files for use.

Kubernetes mounts secrets as read-only by default.

# Secret: Example

The following example demonstrates configuring a Kubernetes pod to use a **secret** for securely injecting sensitive data, showing how to mount secret configurations into the pod:

```
apiVersion: v1
kind: Pod
metadata:
  name: test-hostpath
spec:
  containers:
  - image: registry.k8s.io/test-webserver
    name: test-container
    volumeMounts:
    - mountPath: /secrets
      name: secret-volume
  volumes:
  - name: secret-volume
    secret:
      name: my-secret
```

# Generic Ephemeral Volumes

Generic ephemeral volumes serve a similar purpose to **emptyDir** volumes, providing temporary storage that is created and deleted automatically with the pod life cycle.

Below is an example of a generic ephemeral volume:

## Example

```
kind: Pod
apiVersion: v1
metadata:
  name: my-app
spec:
  containers:
  - name: my-frontend
    image: busybox
    volumeMounts:
    - mountPath: "/scratch"
      name: scratch-volume
    command: [ "sleep", "1000000" ]
  volumes:
  - name: scratch-volume
    ephemeral:
```

## Example

```
volumeClaimTemplate:
  metadata:
    labels:
      type: my-frontend-volume
  spec: inline.storage.kubernetes.io
  accessModes: [ "ReadWriteOnce" ]
  storageClassName: "scratch-storage-
class"
  resources:
    requests:
      storage: 1Gib
```

# Security Considerations for Ephemeral Volumes

The **GenericEphemeralVolume** feature in Kubernetes permits users to indirectly create Persistent Volume Claims (PVCs) during pod deployment, bypassing the need for direct PVC creation permissions.

For those concerned about the security implications of this capability, two remediation options are available:

Disable the feature explicitly through the feature gate

Implement a pod security policy

# CSI Ephemeral Volumes

CSI ephemeral volumes are locally managed on each node. They are created together with other local resources once a node is scheduled to run a pod.

Here is an example of a manifest for a pod utilizing CSI ephemeral storage:

## Example:

```
kind: Pod
apiVersion: v1
metadata:
  name: my-csi-app
spec:
  containers:
    - name: my-frontend
      image: busybox
      volumeMounts:
        - mountPath: "/data"
          name: my-csi-inline-vol
      command: [ "sleep", "1000000" ]
  volumes:
    - name: my-csi-inline-vol
      csi:
        driver: inline.storage.kubernetes.io
        volumeAttributes:
          foo: bar
```

# CSI Ephemeral Volumes: Third-Party CSI Storage Drivers

Third-party CSI storage drivers and additional storage drivers that support dynamic provisioning may facilitate generic ephemeral volumes.

Third-party drivers can offer functionalities not available in Kubernetes.

Some CSI drivers, designed specifically for CSI ephemeral volumes, do not offer support for dynamic provisioning.



However, these drivers are not compatible with generic ephemeral volumes.



## Assisted Practice



### Sharing Data between Containers in the Same Pod

Duration: 10 Min.

#### Problem statement:

You have been asked to demonstrate data-sharing between containers in the same pod through hostPath volumes for mounting pod files onto the file system of the host node.

#### Outcome:

By the end of this demo, you will be able to share data between containers in the same pod using hostPath volumes for mounting files onto the host node's file system.

**Note:** Refer to the demo document for detailed steps

# Assisted Practice: Guidelines



Steps to be followed:

1. Configure and launch the pod with the shared volume
2. Interact with the shared volume from both containers
3. Test data persistence and sharing capability

## Assisted Practice



### Mounting Pod Files to Host with hostPath

Duration: 10 Min.

#### Problem statement:

You have been asked to create a hostPath volume to mount files from a pod onto the file system of the host node.

#### Outcome:

By the end of this demo, you will be able to create a hostPath volume to mount pod files onto the file system of the host node.

**Note:** Refer to the demo document for detailed steps

# Assisted Practice: Guidelines



Steps to be followed:

1. Create a pod using hostPath
2. Create files within the pod
3. Access files on other nodes

## Assisted Practice



### Creating a Deployment with ConfigMap as Volume

Duration: 10 Min.

#### Problem statement:

You have been asked to create a deployment with ConfigMap as volume to enhance the flexibility, manageability, and scalability of your application.

#### Outcome:

By the end of this demo, you will be able to create a deployment with a ConfigMap as a volume to improve the flexibility, manageability, and scalability of your application.

**Note:** Refer to the demo document for detailed steps

# Assisted Practice: Guidelines



Steps to be followed:

1. Create a ConfigMap
2. Create a deployment to attach a ConfigMap as volume

## Assisted Practice



### Creating and Using Secrets in a Volume

Duration: 10 Min.

#### Problem statement:

You have been asked to create a Kubernetes secret and mount it as a volume inside a pod for enhancing security in the Kubernetes environment.

#### Outcome:

By the end of this demo, you will be able to create a Kubernetes secret and mount it as a volume inside a pod to enhance security in a Kubernetes environment.

**Note:** Refer to the demo document for detailed steps

# Assisted Practice: Guidelines



Steps to be followed:

1. Create a Kubernetes secret
2. Create a pod that uses the secret as a volume



## Quick Check

You need to create a temporary storage space that is created when a pod is started and deleted when the pod is removed in Kubernetes. What type of volume would you configure in your pod specification?

- A. emptyDir volume
- B. NFS volume
- C. ConfigMap volume
- D. hostPath volume





## Persistent Volumes

# Persistent Volumes

The PersistentVolume subsystem in Kubernetes provides an API for users and administrators to manage storage resources.

Kubernetes introduces two key API resources:

## PersistentVolume

Represents a provisioned storage resource in the cluster that operates independently of the pod lifecycle, similar to how a node functions independently of a pod

## PersistentVolumeClaim

Represents a user's request for storage, specifying the desired storage size and access modes for the **PersistentVolume**

# PersistentVolume

Each PersistentVolume (PV) defines the volume's specifications and status within the cluster.

## Example:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv0003
spec:
  capacity:
    storage: 5Gi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Recycle
  storageClassName: slow
  mountOptions:
    - hard
    - nfsvers=4.1
  nfs:
    path: /tmp
    server: 172.17.0.2
```

# PersistentVolumeClaim

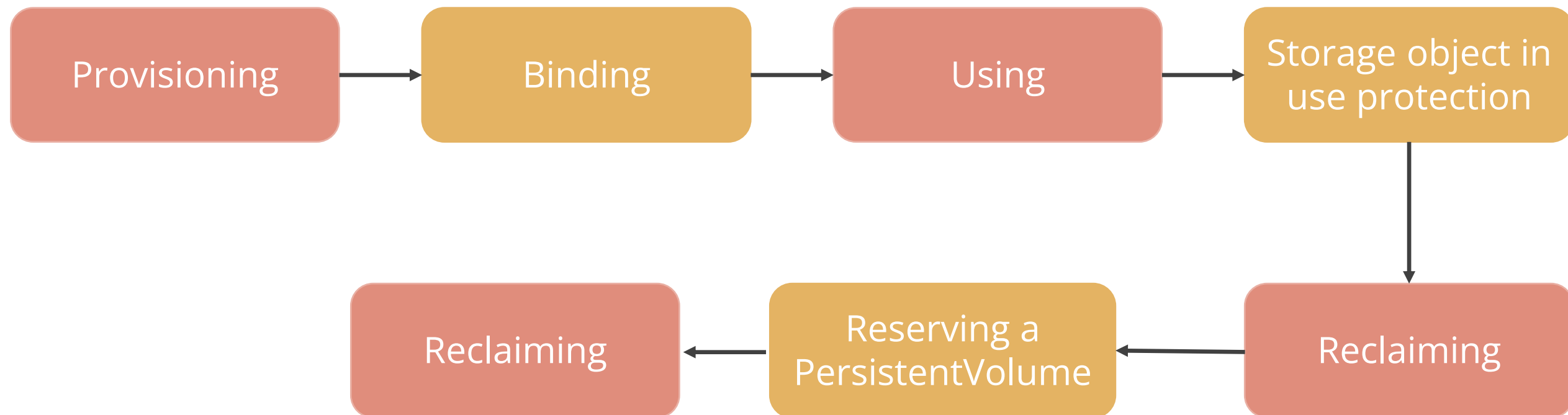
Each PersistentVolumeClaim (PVC) defines the claim's specifications and status within the cluster.

## Example:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: myclaim
spec:
  accessModes:
    - ReadWriteOnce
  volumeMode: Filesystem
  resources:
    requests:
      storage: 1Gi
  storageClassName: slow
  selector:
    matchLabels:
      release: "stable"
    matchExpressions:
      - {key: environment, operator: In,
values: [dev]}
```

# Lifecycle of Volume and Claim

The life cycle of interaction between PVs and PVCs is as shown below:



# Types of Persistent Volumes

Kubernetes supports various non-deprecated volume plugins that allow integration with different storage systems. Some of the currently supported plugins include:

1

awsElasticBlockStore - AWS Elastic Block Store (EBS)

2

azureDisk and azureFile - Azure Disk and Azure File from Microsoft

3

cephfs - CephFS volume

4

csi - Container Storage Interface (CSI)

5

fc - Fibre Channel (FC) storage

6

flexVolume - FlexVolume

7

flocker - Flocker storage

8

gcePersistentDisk - GCE Persistent Disk

# Types of Persistent Volumes

Some of the currently supported plugins include:

9

glusterfs - Glusterfs volume

10

scsi - iSCSI (SCSI over IP) storage

11

local - local storage devices  
mounted on nodes

12

nfs - Network File System (NFS)  
storage

13

portworxVolume - Portworx volume

14

quobyte - Quobyte volume

15

rbd - Rados Block Device (RBD)  
volume

16

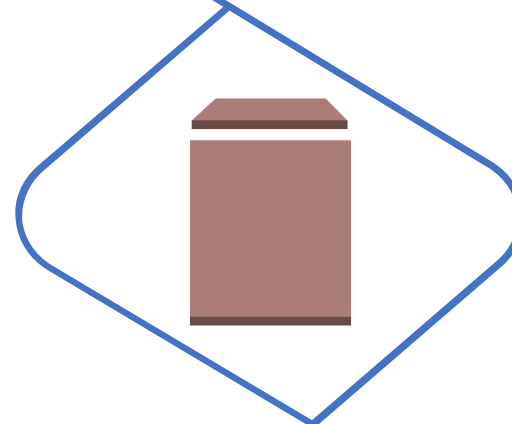
storageos - StorageOS volume



# AwsElasticBlockStore

The **AwsElasticBlockStore** volume is part of Amazon Web Services (AWS) Elastic Block Store (EBS), which provides persistent block-level storage for EC2 instances.

The EBS volume can remain unmounted while keeping its data intact, ensuring durability even when not in use.



Users can pre-load the EBS volume with data, allowing it to be shared between pods. This makes it useful for distributed applications.

# Azure Disk

The **Azure Disk** from Microsoft attaches to a Kubernetes pod using the **azureDisk** volume type, providing persistent storage for containerized applications.



# Cephfs

A **CephFS** volume allows users to mount an existing CephFS volume to their pod, providing a distributed file system for shared data access.



A CephFS volume remains unmounted while preserving its contents.



CephFS volumes can be prepopulated with data for immediate use.



Multiple writers can simultaneously access a CephFS volume.

# gcePersistentDisk and iscsi

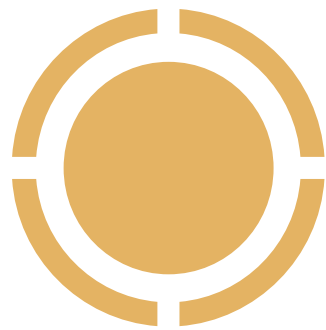
A **gcePersistentDisk** volume mounts a Google Compute Engine Persistent Disk (GCE PD) to the pod, providing persistent block storage for Kubernetes workloads.

These disks can be prepopulated with data, enabling sharing across multiple pods.

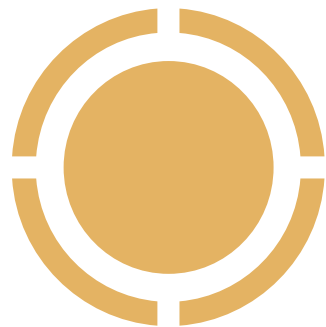
An **iscsi** volume allows an existing iSCSI (SCSI over IP) volume to be mounted to the pod.

iSCSI volumes can be mounted as read-only by multiple consumers simultaneously, making them suitable for shared access scenarios.

# NFS



An existing NFS (Network File System) share can be mounted onto a pod using an NFS volume.



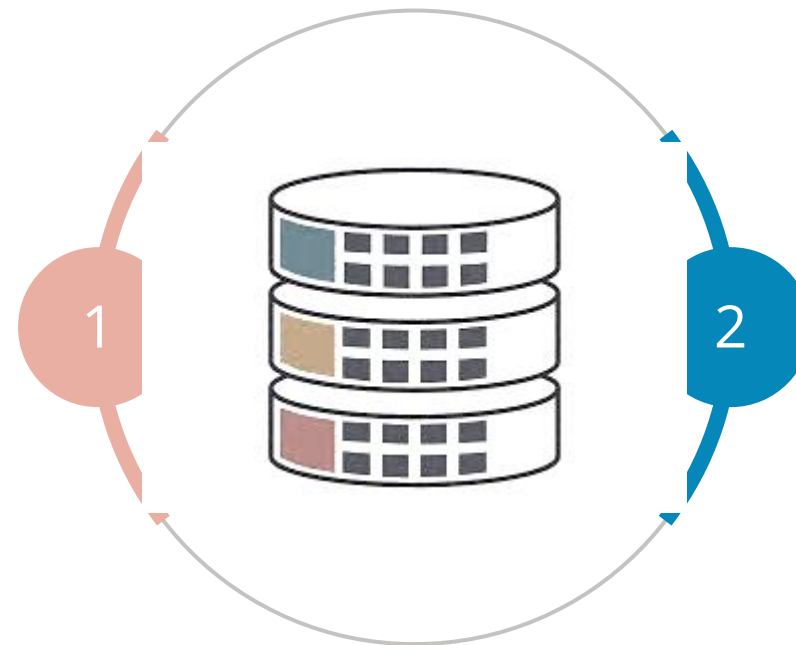
NFS volumes can be prepopulated with data and shared between multiple pods. Additionally, NFS allows simultaneous access by multiple writers, making it ideal for shared workloads.

# StorageOS

An existing **StorageOS** volume can be mounted onto a pod using a **StorageOS** volume type.

**StorageOS** operates as a container within the Kubernetes environment, making local or attached storage accessible from any node in the Kubernetes cluster.

It replicates data, providing redundancy and protection against node failure.

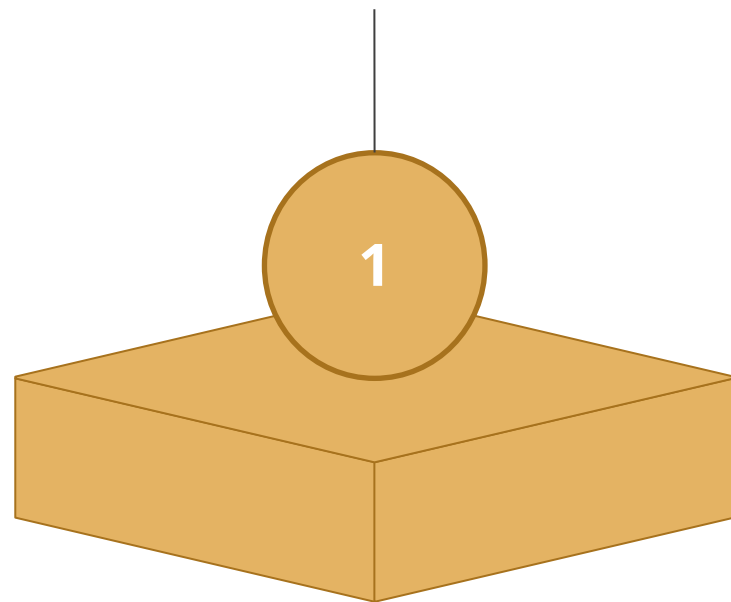


It provides block storage to containers, accessible through a file system, ensuring efficient storage management and high availability.

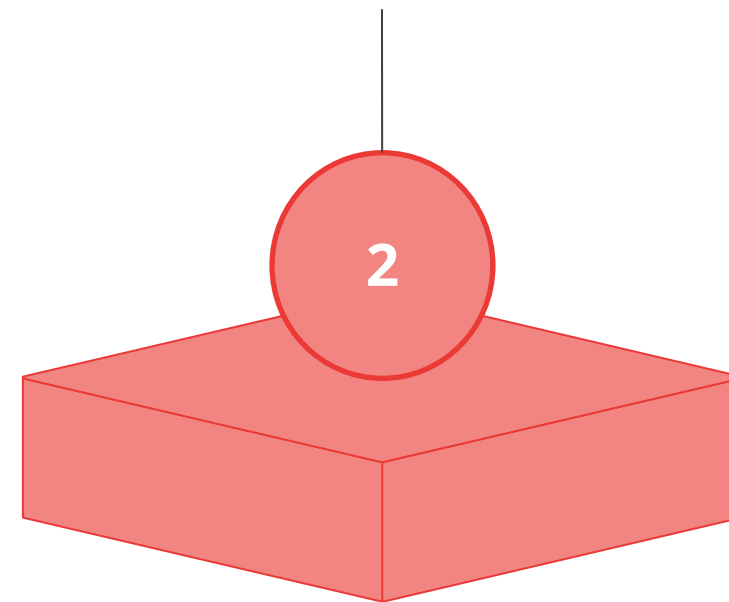
# Claim as Volumes

Pods access storage by using PersistentVolumeClaims (PVCs) as volumes.

Claims must be in the same namespace as the pod that is using the claim.



The cluster uses the claim within the pod's namespace to bind it to the corresponding PersistentVolume (PV) that provides the actual storage.



# Raw Block Volume Support

The following volume plugins in Kubernetes support raw block storage:

AWSElasticBlockStore

AzureDisk

CSI

FC (Fibre Channel)

GCEPersistentDisk

iSCSI

Local volume

OpenStack Cinder

RBD (Ceph Block  
Device)

VsphereVolume



# PersistentVolume: Using a Raw Block Volume

The following example represents how to configure a **PersistentVolume** using a raw block volume with the **ReadWriteOnce** access mode:

## Example

```
apiVersion: v1
Kind: PersistentVolume
Metadata:
  name: block-pv
spec:
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteOnce
  volumeMode: Block
  persistentVolumeclaimPolicy: Retain
  fc:
    targettwWWNs: { "50060e801049cfd1" }
    lun: 0
    readOnly: false
```

# PersistentVolumeClaim: Requesting a Raw Block Volume

The following example represents how a **PersistentVolumeClaim** can request a raw block volume with the **ReadWriteOnce** access mode:

## Example

```
apiVersion: v1
Kind: PersistentVolumeClaim
Metadata:
  name: block-pvc
Spec:
  accessModes:
    - ReadWriteOnce
  volumeMode: Block
  resources:
    requests:
      storage: 10Gi
```

# Pod Specification: Adding Raw Block Device Path

The following example represents how to add a raw block device path to a container:

## Example

```
apiVersion: v1
Kind: Pod
Metadata:
  name :   pod-with-block-volume
Spec:
  Containers:
    - name:   fc-container
      image:   fedora:26
      Command: ["/bin/sh", "-c"]
      args:    [ "tail -f /dev/null" ]
      volumeDevices:
        - name:   data
          devicePath: /dev/xvda
  volume :
    - name:   data
      persistentVolumeClaim:
        claimName: block-pvc
```

# Binding Block Volumes

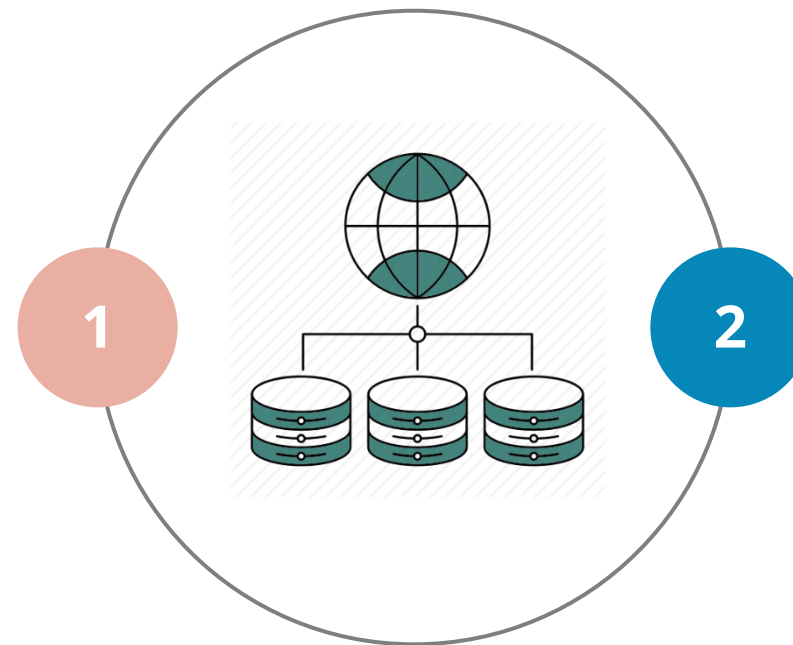
The following table indicates whether a volume will be bound based on the combination of PersistentVolume (PV) and PersistentVolumeClaim (PVC) volume modes:

PV volumeMode	PVC volumeMode	Result
Unspecified	Unspecified	BIND
Unspecified	Block	NO BIND
Unspecified	Filesystem	BIND
Block	Unspecified	NO BIND
Block	Block	BIND
Block	Filesystem	NO BIND
Filesystem	Filesystem	BIND
Filesystem	Block	NO BIND
Filesystem	Unspecified	BIND

# Lifecycle and PersistentVolumeClaim

The fundamental design principle is allowing parameters for a volume claim within a pod's volume source.

In terms of resource ownership, a pod with generic ephemeral storage owns the PersistentVolumeClaim(s) that provide the ephemeral storage.



When a user deletes the pod, the Kubernetes garbage collector automatically deletes the PersistentVolumeClaim(s).

## Assisted Practice



### Configuring Pod Storage Using hostPath-Based PV and PVC

Duration: 10 Min.

#### Problem statement:

You have been asked to configure pod storage using hostPath-based PersistentVolume (PV) and PersistentVolumeClaim (PVC) in Kubernetes for efficient data storage and retrieval.

#### Outcome:

By the end of this demo, you will be able to configure pod storage using HostPath-based PersistentVolume (PV) and PersistentVolumeClaim (PVC) in Kubernetes for efficient data storage and retrieval.

**Note:** Refer to the demo document for detailed steps

# Assisted Practice: Guidelines



Steps to be followed:

1. Create PersistentVolume
2. Create PersistentVolumeClaim
3. Deploy a pod in a new namespace
4. Validate the pod
5. Verify data persistence

## Assisted Practice



### Configuring Pod Using NFS Based PV and PVC

Duration: 10 Min.

#### Problem statement:

You have been asked to configure a pod using NFS based PersistentVolume (PV) and PersistentVolumeClaim (PVC) for more efficient storage management.

#### Outcome:

By the end of this demo, you will be able to configure a pod using NFS-based PersistentVolume (PV) and PersistentVolumeClaim (PVC) for more efficient storage management.

**Note:** Refer to the demo document for detailed steps



# Assisted Practice: Guidelines



Steps to be followed:

1. Configure the NFS kernel server
2. Set the permissions
3. Configure the NFS common on client machines
4. Create PersistentVolume
5. Create PersistentVolumeClaim
6. Create the deployment for MySQL

## Quick Check

You are tasked with defining a persistent storage volume in a Kubernetes cluster that should support NFS-based storage with a ReadWriteOnce access mode. The volume should have a total capacity of 5Gi, and when no longer needed, it should be reclaimed by the cluster for reuse. Which configuration would meet these requirements?

- A. A hostPath storage with ReadWriteOnce and 5Gi storage capacity
- B. An NFS-backed PersistentVolume with ReadWriteOnce, 5Gi capacity, and Recycle reclaim policy
- C. An emptyDir volume with a size limit of 5Gi
- D. A PersistentVolume with NFS storage and a Delete reclaim policy





## Volume Snapshots

# Introduction to Volume Snapshots

Volume snapshots provide Kubernetes users with a standardized way to copy a volume's contents at a particular point in time without creating an entirely new volume.

A **VolumeSnapshot** is a user's request for a snapshot of a volume. This snapshot is similar to a **PersistentVolumeClaim**.

# VolumeSnapshotContent and VolumeSnapshotClass

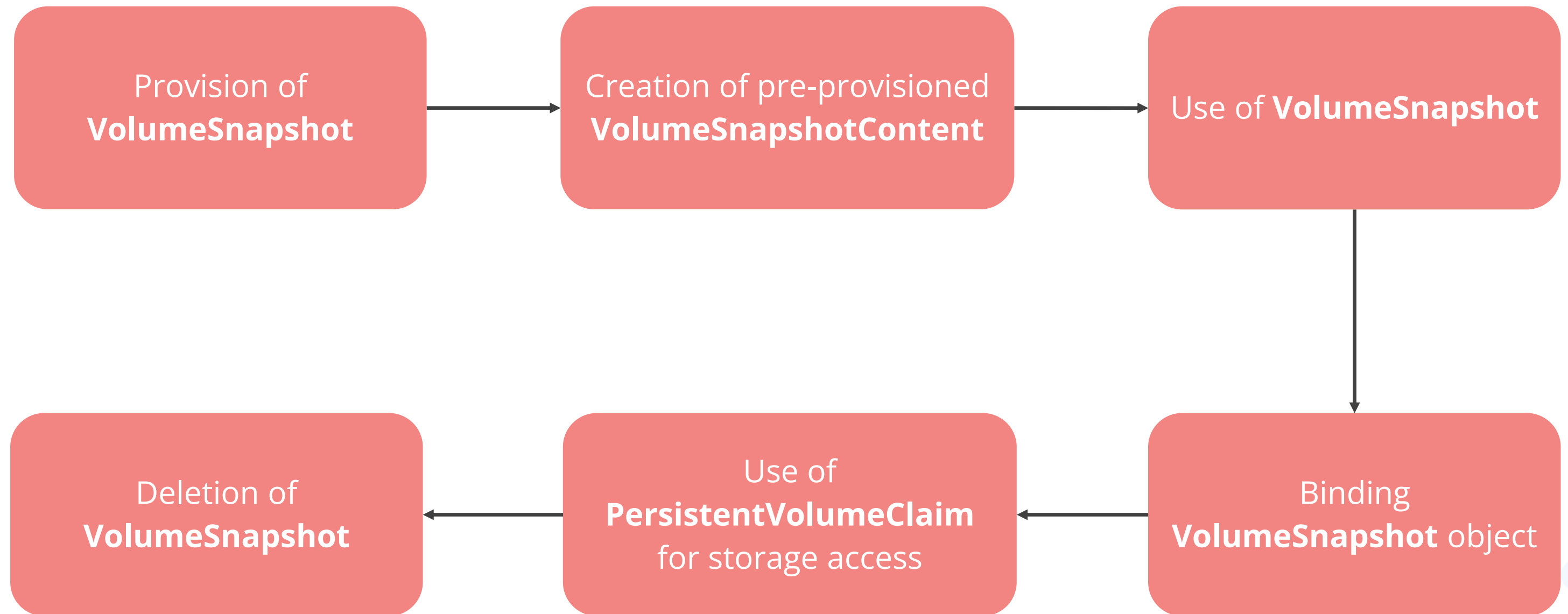
**VolumeSnapshotContent** and **VolumeSnapshot** API resources enable the creation of volume snapshots for users and administrators.

A **VolumeSnapshotContent** is a snapshot originating from a volume in the cluster that has been provisioned by an administrator.

The **VolumeSnapshotClass** allows the specification of different attributes for a **VolumeSnapshot**.

# Lifecycle of VolumeSnapshot and VolumeSnapshotContent

The life cycle of interaction between **VolumeSnapshot** and **VolumeSnapshotContent**:



# VolumeSnapshot

A spec and a status are parts of a **VolumeSnapshot**.

## Example

```
apiVersion: snapshot.storage.k8s.io/v1
kind:      VolumeSnapshot
metadata:
  name:     new-snapshot-test
spec:
  volumeSnapshotClassName: csi-hostpath-snapclass
  source:
    persistentVolumeClaimName: pvc-test
```

# VolumeSnapshot

The **volumeSnapshotContentName** source field is required for pre-provisioned snapshots.

## Example

```
apiVersion:  snapshot.storage.k8s.io/v1
kind:      VolumeSnapshot
metadata:
  name:     test-snapshot
spec:
  source:
    volumeSnapshotContentName: test-contest
```



# Provisioning Volumes from Snapshots

The **dataSource** field in the PersistentVolumeClaim can be used to provision a new volume that has been pre-populated with data from a snapshot.



# VolumeSnapshotContent

In dynamic provisioning, the snapshot common controller creates **VolumeSnapshotContent** objects.

## Example

```
apiVersion:  snapshot.storage.k8s.io/v1
kind:      VolumeSnapshot
metadata:
  name: snapcontent- 72d9a349-aacd-42d2-a248-d77560d2455
spec:
  deletionPolicy: Delete
  driver: hostpath.csi.k8s.io
  source:
    volumeHandle: ee0cfb94-f8d4-11e9-b2d8-0242ac1110002
  volumeSnapshotClassName: csi-hostpath-snapclass
  volumeSnapshotRef:
    name: new-snapshot-test
    namespace: default
  vid: 72d9a349-aacd-42d2-a240-d775650d2455
```

# VolumeSnapshotContent

In pre-provisioned snapshots, the cluster administrator creates the **VolumeSnapshotContent** object.

## Example

```
apiVersion:  snapshot.storage.k8s.io/v1
kind:      VolumeSnapshot
metadata:
  name:     new-snapshot-content-test
spec:
  deletionPolicy: Delete
  driver:        hostpath.csi.k8s.io
  source:
    snapshotHandle: 7bdd0de3-aaeb-9aae-0242ac110002
  volumeSnapshotRef:
    name: new-snapshot-test
    namespace: default
```

## Quick Check

You are tasked with provisioning a new Persistent Volume in Kubernetes using data from an existing snapshot. Which field in the PersistentVolumeClaim (PVC) allows you to specify the source of this data?

- A. `.storageClassName`
- B. `.accessModes`
- C. `.dataSource`
- D. `.volumeName`





## Storage Classes

# Storage Classes

Administrators use a **StorageClass** to define the different types of storage they offer.



Kubernetes is neutral about the meaning or purpose of storage classes.

# StorageClass Resource

A **StorageClass** allows administrators to define different storage classes. Administrators can also set a default **StorageClass** for PersistentVolumeClaims (PVCs) that do not specify a particular class.

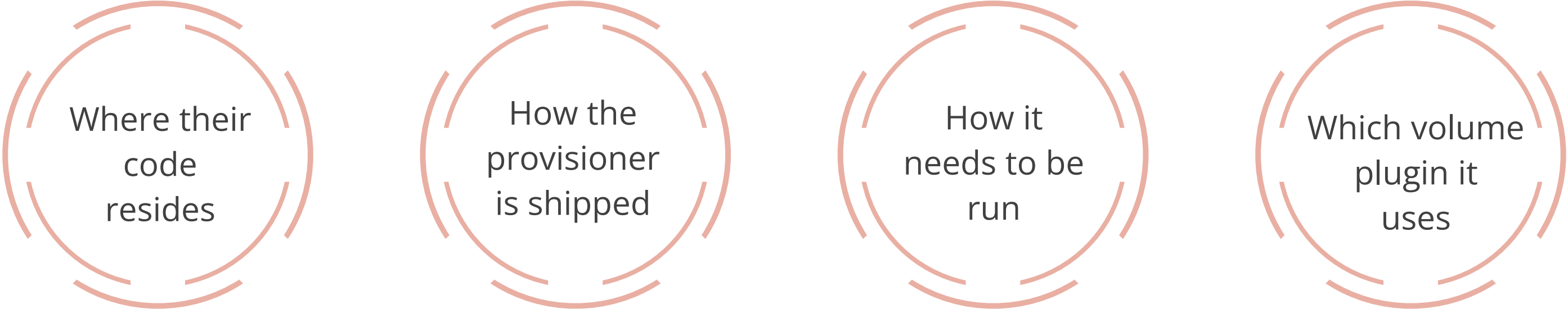
## Example

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: standard
provisioner: kubernetes.io/aws-ebc
parameters:
  type: gp3
reclaimPolicy: Retain
allowVolumeExpansion: true
mountOptions:
  - debug
volumeBindingMode: Immediate
```

# Provisioner

Each **StorageClass** defines a provisioner, which determines the volume plugin to be used for provisioning PersistentVolumes (PVs).

External provisioners have full control over:



Where their  
code  
resides

How the  
provisioner  
is shipped

How it  
needs to be  
run

Which volume  
plugin it  
uses



# StorageClass Resource: Fields

## Reclaim policy

PersistentVolumes created dynamically by a StorageClass will follow the reclaim policy specified in the **reclaimPolicy** field. The policy can either be **Delete** or **Retain**.

## Volume expansion

PVs can be configured to allow expansion, making them flexible to changes in storage needs.

## Mount options

PVs will apply the mount options specified in the **mountOptions** field of the StorageClass.

## Volume binding mode

PVs will follow the binding process as defined in the **volumeBindingMode** field, determining when volume binding and dynamic provisioning occur.

# Allowed Topologies

When a cluster operator uses the **WaitForFirstConsumer** volume binding mode, provisioning is not restricted to specific topologies.

The following code snippet demonstrates how to restrict the provisioning of volumes to specific zones based on topology settings:

## Example

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: standard
provisioner: kubernetes.io/gce-pd
parameters:
  type: pd-standard
volumeBindingMode: WaitForFirstCustomer
allowedTopologies:
- matchLabelExpressions:
  - key: failure-domain.beta.kubernetes.io/zone
    values
  - vs-central-a
  - vs-central-b
```

# Supported Storage Provisioners

**StorageClasses** include parameters that define the characteristics of volumes assigned to the storage class; a maximum of 512 parameters can be specified for a StorageClass.

The provisioners include the following providers:

- 1 AWS EBS
- 2 GCE PD
- 3 Glusterfs
- 4 OpenStack Cinder

- 5 vSphere
- 6 CSI Provisioner
- 7 vCP Provisioner
- 8 Ceph RBD

# Supported Storage Provisioners

The provisioners include the following providers:

9 QuoByte

10 Azure Disk

11 Azure File

12 Portworx Volume

13 ScaleIO

14 StorageOS

15 Local

## Quick Check

You need to configure a PersistentVolumeClaim (PVC) without specifying a storage class in your Kubernetes cluster. Which resource can be configured by the administrator to define the default behavior for such claims?

- A. Pod
- B. StorageClass
- C. PersistentVolume
- D. VolumeSnapshot



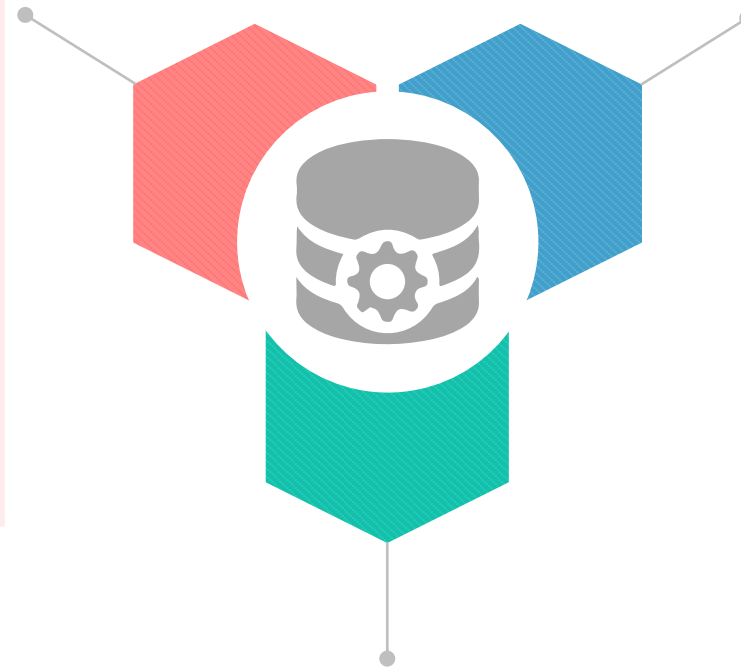


# **Dynamic Volume Provisioning**

# Overview of Dynamic Volume Provisioning

**Dynamic volume provisioning** enables storage volumes to be created automatically when needed, without manual intervention.

**Cluster administrators** can define **StorageClass** objects based on specific requirements.



When a user makes a request, **automatic storage provisioning** occurs, creating the required volumes.

**Dynamic provisioning** removes the necessity for cluster administrators to allocate storage resources ahead of time.

# Enabling Dynamic Provisioning

A cluster administrator defines one or more **StorageClass** objects in advance to enable dynamic provisioning.

## Example

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: custom-managed-premium
provisioner: disk.csi.azure.com
parameters:
  cachingmode: ReadOnly
  kind: Managed
  storageaccounttype: Premium_LRS
reclaimPolicy: Delete
volumeBindingMode: Immediate
```

These objects define storage specifications that are used when dynamically provisioning volumes.



# Using Dynamic Provisioning

A user can create a **PersistentVolumeClaim (PVC)** to dynamically provision storage by specifying a **StorageClass** and access modes, as shown in the example:

## Example:

```
apiVersion: v1
Kind: PersistentVolumeClaim
Metadata:
  name: claim1
Spec:
  accessModes:
    - ReadWriteOnce
  storageClassName: custom-managed-premium
  resources:
    requests:
      storage: 30Gi
```

# Defaulting Behavior

A cluster admin should perform the following steps to enable defaulting behavior:

Mark one **StorageClass** object as the default

Ensure that the **Default StorageClass Admission Controller** is enabled on the API server

## Assisted Practice



### Configuring Multi-Container Pods with RWX Access Using PV and PVC

Duration: 10 Min.

#### Problem statement:

You have been asked to configure multi-container pods with ReadWriteMany (RWX) access in Kubernetes using PersistentVolume (PV) and PersistentVolumeClaim (PVC) for shared storage and data operations.

#### Outcome:

By the end of this demo, you will be able to configure multi-container pods with ReadWriteMany (RWX) access in Kubernetes using PersistentVolume (PV) and PersistentVolumeClaim (PVC) for shared storage and data operations.

**Note:** Refer to the demo document for detailed steps

# Assisted Practice: Guidelines



Steps to be followed:

1. Create PersistentVolume
2. Create PersistentVolumeClaim
3. Deploy a pod in a new namespace
4. Demonstrate shared storage and data operations
5. Continue data operations

## Quick Check

You are tasked with dynamically provisioning storage for a Kubernetes pod by creating a PersistentVolumeClaim (PVC). What must be specified in the PVC to ensure that the appropriate storage class and access modes are used?

- A. Pod name
- B. PersistentVolume
- C. StorageClass and access modes
- D. Node selector





## Storage Capacity

# Storage Capacity

It is limited and can vary depending on the node where a pod is running.



Storage capacity tracking is supported for Container Storage Interface (CSI) drivers and must be enabled when a CSI driver is installed.

# API Extensions

Kubernetes provides two API extensions for tracking storage capacity with CSI drivers:

**CSI storage capacity object**

Tracks available storage space for CSI drivers

**CSIDriverSpec storage capacity  
field**

Indicates if a CSI driver supports reporting storage capacity



# Criteria Scheduling

Scenarios in which the Kubernetes scheduler utilizes storage capacity information are:



The CSIStorageCapacity feature gate is set to true.



A pod references a volume that has not yet been created.



A volume uses a StorageClass with a CSI driver, and the WaitForFirstConsumer binding mode is applied.



The CSIDriver object for the CSI driver has StorageCapacity enabled.

# Scheduling

The system compares the volume's size with the capacity listed in CSI storage capacity objects, considering the topology that includes the node.

Volumes set to immediate volume binding mode rely on the storage driver to determine the volume's creation location.

Scheduling processes do not take storage capacity into account for CSI ephemeral volumes.

# Rescheduling

After selecting a node for a pod that uses **WaitForFirstConsumer** volumes, the system asks the CSI storage driver to create the volume, ensuring its availability on the designated node.

## Storage capacity tracking

Tracking storage capacity enhances the likelihood of successful scheduling on the first attempt.

## Potential for permanent scheduling failure

When a pod utilizes multiple volumes, scheduling might permanently fail.

# Enable Storage Capacity Tracking

Storage capacity tracking is an alpha feature. It operates only when the CSIStorageCapacity feature gate, and the storage.k8s.io/v1alpha1 API groups are enabled.

The following code can be used to check and list CSIStorageCapacity objects:

## Example

```
#A quick check whether a Kubernetes cluster supports the feature is to  
list CSIStorageCapacity objects with:
```

```
kubectl get csistoragecapacities --all-namespaces
```

```
#If the cluster supports CSIStorageCapacity, the response is either a  
list of CSIStorageCapacity objects or:
```

```
No resources found
```

```
#If not supported, this error is printed instead:
```

```
error: the server doesn't have a resource type "csistoragecapacities"
```

## Quick Check

While setting up scheduling for Kubernetes volumes, you notice that the system must determine the volume's creation location based on certain conditions. Which of the following accurately describes how the system behaves during this process?

- A. Volumes with delayed volume binding modes select a random node.
- B. Immediate volume binding modes rely on the storage driver to determine the volume's creation location.
- C. Scheduling takes storage capacity into account for CSI ephemeral volumes.
- D. The Kubernetes scheduler always defines the volume's location, regardless of the volume binding mode.

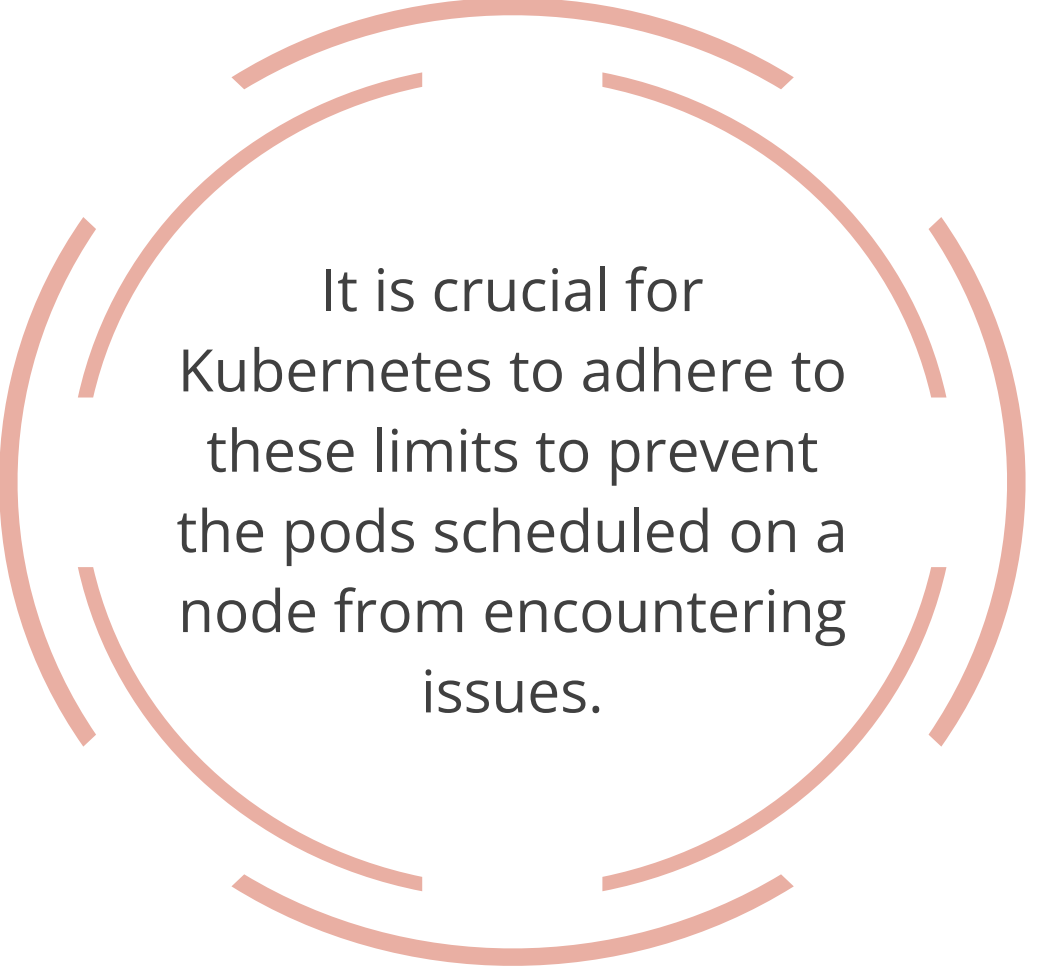




## **Node-Specific Volume Limits**

# Introduction to Node-Specific Volume Limits

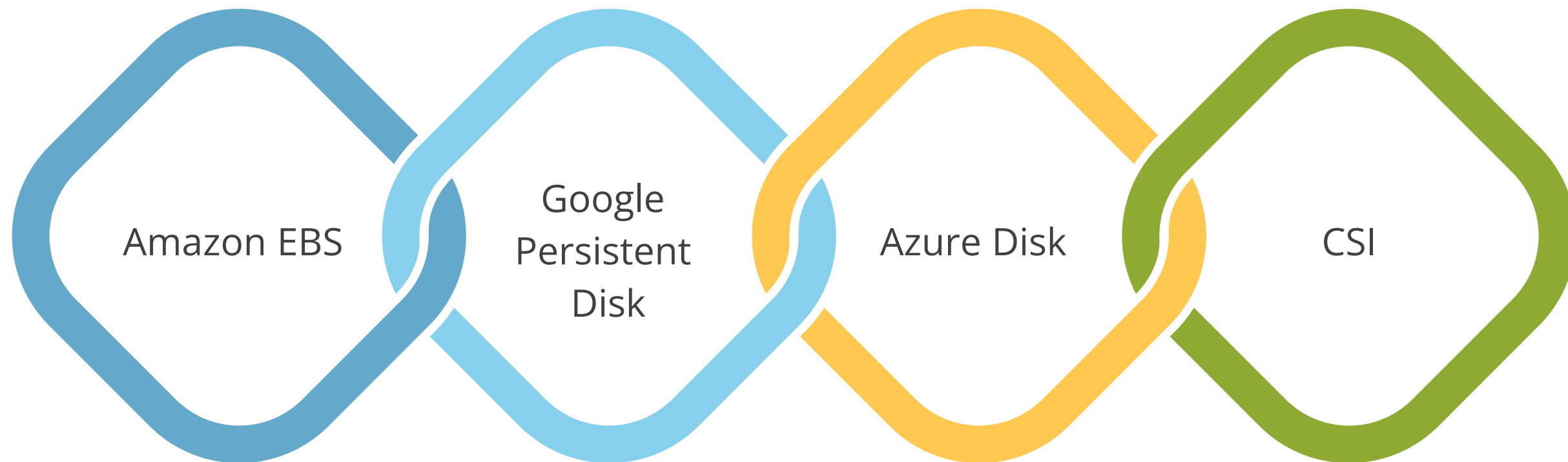
Cloud providers like Google, Amazon, and Microsoft impose limits on the number of volumes that can be attached to a single node.



It is crucial for Kubernetes to adhere to these limits to prevent the pods scheduled on a node from encountering issues.

# Dynamic Volume Limits

Dynamic volume limits are supported for the following storage types:





# Kubernetes Default Limits

The Kubernetes scheduler enforces default volume limits based on the cloud service being used.

Cloud service	Maximum volumes per node
Amazon Elastic Block Store (EBS)	39
Google Persistent Disk	16
Microsoft Azure Disk Storage	16

## Custom Limits

The volume limits can be adjusted by setting the **KUBE\_MAX\_PD\_VOLS** environment variable and restarting the scheduler.



### Note

Use caution if you set a limit that is higher than the default limit

## Quick Check

You are a Kubernetes administrator responsible for your organization's cluster. You are configuring the volume limits in a Kubernetes cluster and need to increase the maximum number of volumes that can be attached to a node. Which of the following steps should you take to achieve this?

- A. Change the maximum volume limit directly in the Kubernetes manifest
- B. Increase the PersistentVolumeClaim (PVC) size to handle more volumes
- C. Adjust the node's capacity manually in the node configuration file
- D. Modify the **KUBE\_MAX\_PD\_VOLS** environment variable and restart the scheduler



# Key Takeaways

- A pod can use any number of volumes and volume types simultaneously while working with Kubernetes.
- A StorageClass provides a way for administrators to describe the classes of storage that they offer.
- The CSI volume cloning feature adds support for specifying existing PVCs in the dataSource field to indicate that a user would like to clone a volume.
- Tracking storage capacity is supported for container storage interface (CSI) drivers and needs to be enabled when installing a CSI driver.



# Deploying WordPress and MySQL Using PersistentVolume

Duration: 25 minutes

**Project agenda:** To deploy WordPress and MySQL on Kubernetes with PersistentVolume using NFS and hostPath for web-based access

**Description:** This project involves the deployment of WordPress and MySQL on a Kubernetes cluster, leveraging PersistentVolume with NFS and hostPath configurations to host a web-based WordPress application with data persistence and accessibility.



# Deploying WordPress and MySQL Using PersistentVolume

Duration: 25 minutes

## Steps to perform:

1. Configure the NFS kernel server
2. Set the permissions
3. Configure NFS common on client machines
4. Create a MySQL manifest file and deploy it using NFS-based PersistentVolume
5. Create a WordPress manifest file and deploy it using hostPath-based PersistentVolume

**Expected deliverables:** A fully deployed Kubernetes cluster and MySQL and WordPress applications, with PersistentVolume, that allow web-based access to the WordPress site





**Thank You**