Kubernetes Storage

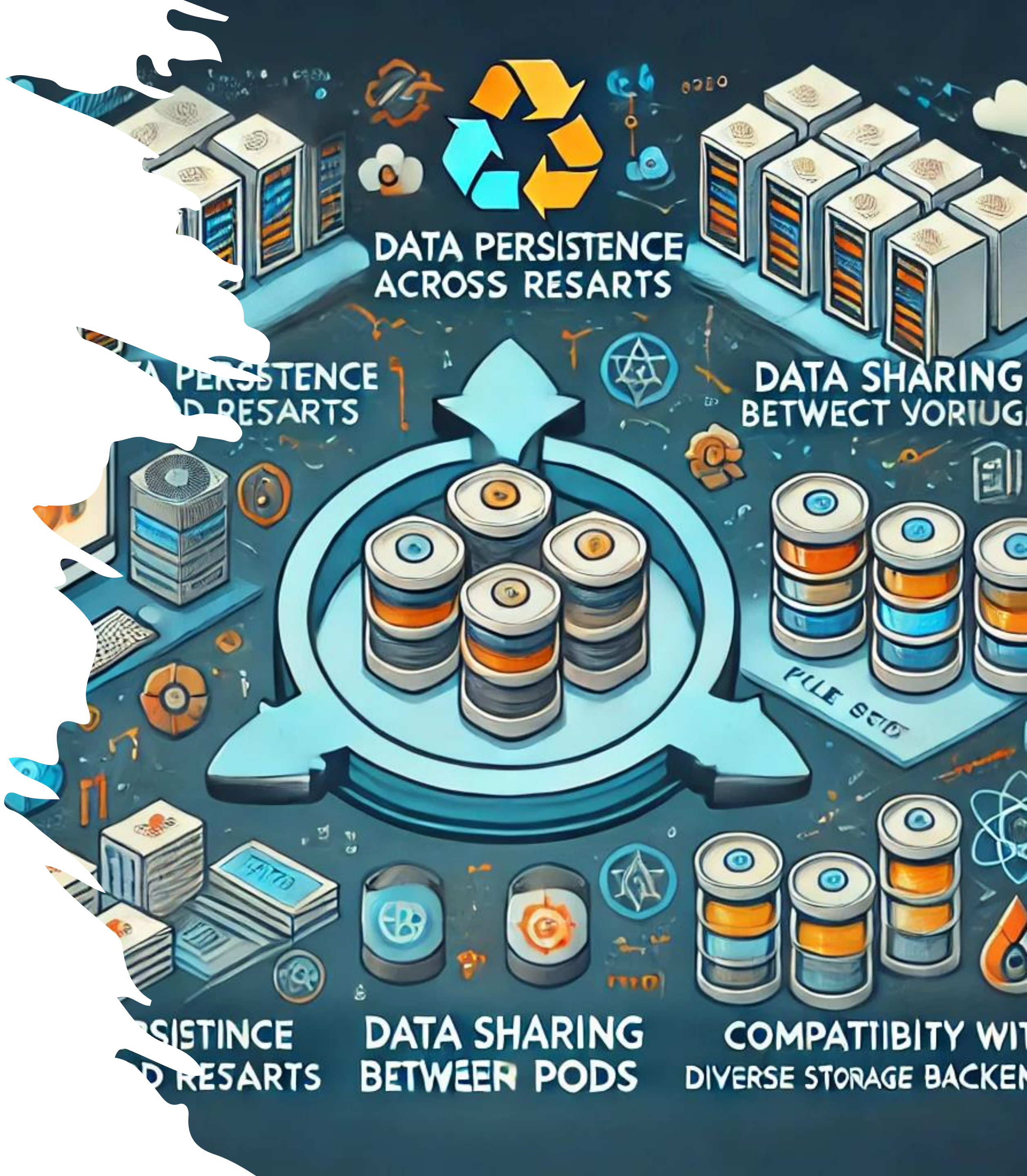
Kubernetes Installation and Administration Course

Lisa Zangrando (lisa.zangrando@pd.infn.it)

Kubernetes and Storage: beyond Cloud-Native applications

- **Kubernetes for Cloud-Native applications**
  - designed to manage distributed architectures, dynamic scalability, and resilience.
  - ideal for stateless workloads.

- **The challenge of scientific applications**
  - scientific applications often require **persistent and reliable data access**, especially for use cases like: Data Analysis applications, logging systems, databases

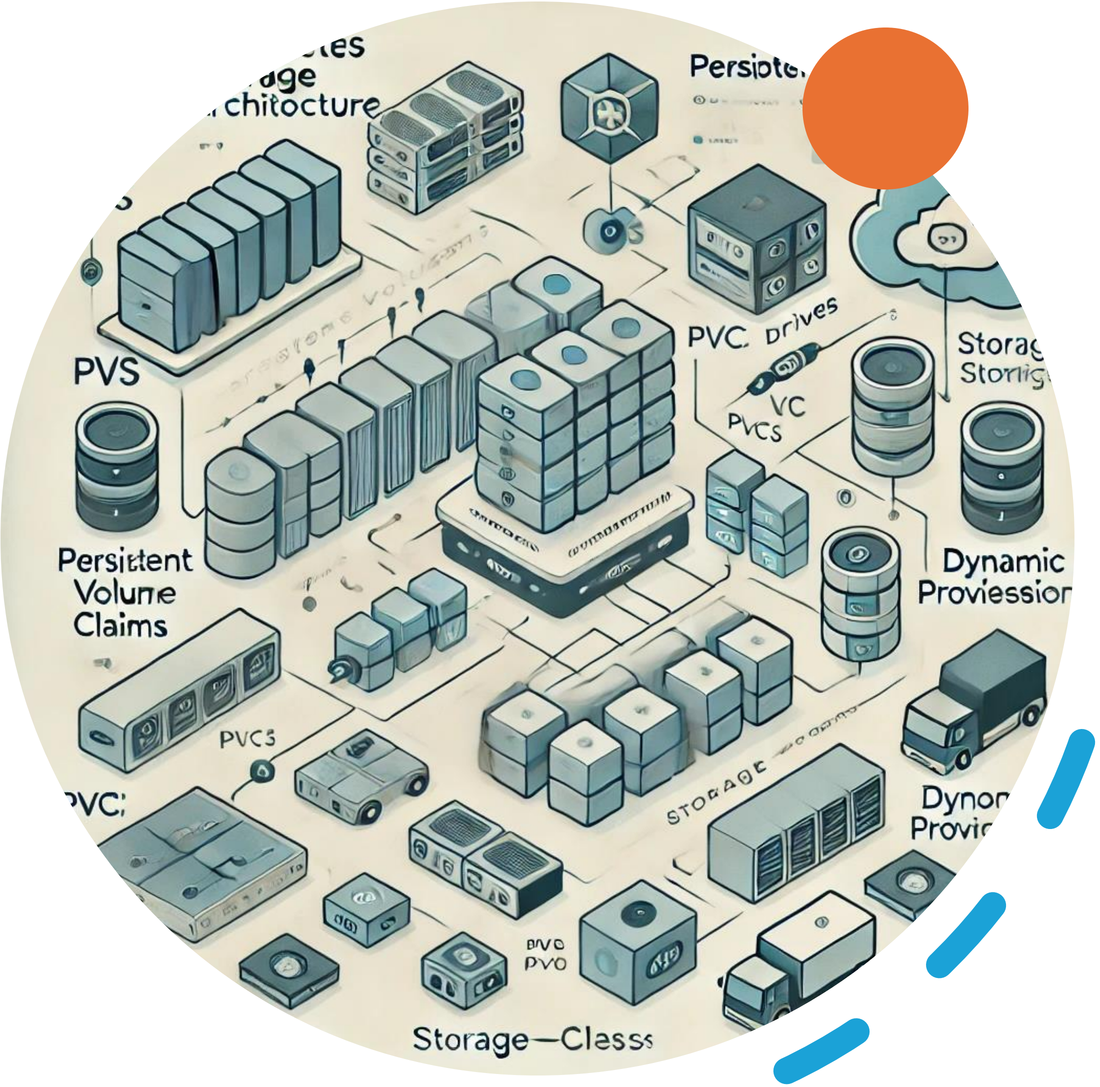


Why storage is essential in Kubernetes?

1. **Data persistence across Pod restarts**
    - Pods are **temporary** by design.
    - Without persistent storage, data is **lost** when Pods are deleted or restarted.
    - Persistent storage decouples the **lifecycle of data** from the lifecycle of Pods, ensuring data survives infrastructure changes.

2. **Data sharing between Pods**
    - Distributed applications often require multiple Pods to **simultaneously access** the same data.
    - Kubernetes supports **shared volumes**, enabling efficient and straightforward data sharing.

3. **Compatibility with Diverse Storage Backends**
    - Kubernetes supports a wide range of storage solutions to meet application needs:
        - **Block Storage**: high-performance storage for transactional databases.
        - **Shared File Systems**: collaborative or distributed applications.
        - **Object Storage**: scalable, long-term storage for big data systems.



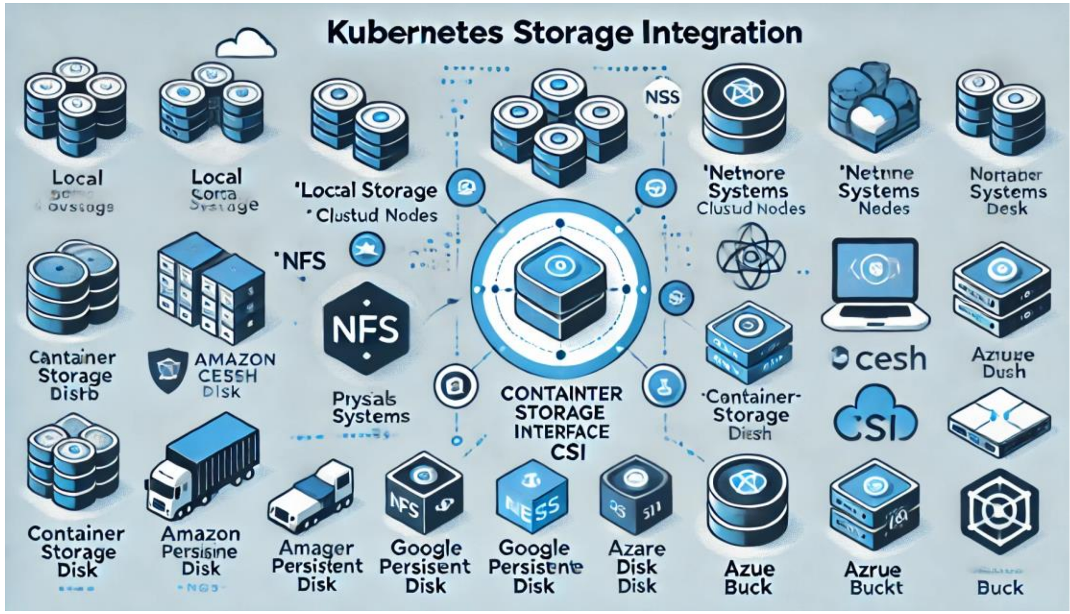
Storage integration in Kubernetes

- Kubernetes simplifies storage management by providing native integrations with various types of infrastructures:
  - **Local storage: d**irectly using physical disks attached to cluster nodes.
  - **Network systems:** solutions like **NFS** or **Ceph**, offering shared access and scalability.
  - **Cloud-Native services:** options such as **Amazon EBS**, **Google Persistent Disk**, **Azure Disk**, and S3-like object storage services.

- Kubernetes supports custom storage plugins through the **Container Storage Interface (CSI)**.

- CSI allows developers to integrate **any storage system**:
  - **Commercial** solutions
  - **Custom** setups

- This makes Kubernetes a **universal solution** for:
  - **On-Premises**
  - **Hybrid**
  - **Cloud-Native Environments**



A flexible architecture based on Volumes

- **The core of Kubernetes storage: Volumes**
  - Volumes are abstractions that allow containers to access storage resources without being dependent on the underlying infrastructure.
  - A Kubernetes Volume is essentially a directory mounted into containers within a Pod.

- **Simplified data handling for applications**
  - Volumes provide Pods with a mechanism to **read and write files**, abstracting the complexities of storage backend connections.
  - Containers remain unaware of the complexity of the underlying storage.

- **Volume usage in Pods**
  - Once created, a Volume is **mounted into containers** as a directory, becoming an integral part of the container's filesystem.
  - Enables **data sharing between containers** in the same Pod:
    - Ideal for multi-container applications where one container generates data for another.

How Volumes work

- Volumes are defined in the **pod spec (YAML).**
- They are **mounted** into containers, making them shared and accessible at specified paths.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
  - name: container-1
    image: my-image
    volumeMounts:
    - name: my-volume
      mountPath: /mnt/data
  volumes:
  - name: my-volume
    <VOLUME DEFINITION>
```

Use cases

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
  - name: my-container
    image: busybox
    volumeMounts:
    - mountPath: /data
      name: my-volume-1
    - mountPath: /app
      name: my-volume-2
  volumes:
  - name: my-volume-1
    <VOLUME-DEFINITON>
  - name: my-volume-2
    <VOLUME-DEFINITON>
```

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
  - name: my-container-1
    image: busybox
    volumeMounts:
    - mountPath: /data
      name: my-volume
  - name: my-container-2
    image: busybox
    volumeMounts:
    - mountPath: /storage
      name: my-volume
  volumes:
  - name: my-volume
    <VOLUME-DEFINITON>
```

Types of Volumes

- **Ephemeral Volumes:**
  - Temporary and tied to the **lifecycle of the Pod**.
  - Commonly used for caching, temporary data, or inter-container communication.

- **Persistent Volumes (PVs):**
  - Designed for long-term storage, independent of a Pod's lifecycle.
  - Ideal for applications requiring **durable storage**, such as databases.
  - Each type of volume serves distinct use cases, addressing different levels of **data persistence** and lifecycle requirements.

Ephemeral Volumes

- Ephemeral volumes in Kubernetes are temporary storage resources that are created and destroyed with the lifecycle of a pod. They are ideal for storing data that is transient, such as application logs, temporary files, or caches. Once the pod is deleted, the data in ephemeral volumes is also removed.
- **Types of Ephemeral Volumes**
  - **EmptyDir:** A temporary directory for a pod that can be shared among containers within the same pod.
  - **ConfigMap:** Allows injecting configuration data into containers, where the data is mounted as files.
  - **Secrets:** A Kubernetes object used to store sensitive data, such as passwords, tokens, or SSH keys, which can be mounted as files or environment variables.
- **Use cases**
  - **Logs storage:** Temporary storage for logs or runtime data.
  - **Scratch space:** For computation or intermediate data storage.
  - **Configuration storage:** Injecting configuration or sensitive information into containers.
  - **Caching**: speed up operations with local, short-term storage.

EmptyDir

- `EmptyDir` is an ephemeral volume that is created when a pod is assigned to a node and is deleted when the pod is terminated. It provides a temporary directory shared by all containers in the pod.
- **Use cases:**
  - Sharing temporary data between containers in the same pod, such as caches or intermediate computation results.
  - Used for storing temporary files that do not need to persist beyond the lifecycle of the pod.

```yaml
apiVersion: v1
kind: Pod
metadata:
   name: ephemeral-example
spec:
  containers:
  - name: app-container
    image: nginx
    volumeMounts:
    - mountPath: "/usr/share/nginx/html"
      name: scratch-volume
  volumes:
  - name: scratch-volume
    EmptyDir:
     sizeLimit: 500Mi
     medium: Memory
```

Explanation:
- The `emptyDir` volume is created in memory when the pod starts.
- The size limit is 500Mb
- It mounts at `/usr/share/nginx/html` in the `app-container`.
- The volume is deleted when the pod is terminated.

ConfigMap (1/3)

- A ConfigMap is a type of ephemeral volume used to store non-sensitive configuration data in the form of key-value pairs. It can be mounted as a volume or exposed as environment variables in a container.
- Each data item in the ConfigMap is represented by an individual file in the volume.
- **Key Features**:
  - Allows decoupling configuration from container images.
  - Automatically updates when the ConfigMap changes (depending on pod settings).
- **Example**:
  - The ConfigMap example-config contains a simple HTML file.
  - The pod mounts the ConfigMap as a volume (config-volume) at /usr/share/nginx/html.
  - When the pod runs, the NGINX server serves the content of the index.html file from the ConfigMap.

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: example-config
data:
  index.html: |
    <html>
      <head><title>Welcome</title></head>
      <body><h1>Hello, Kubernetes!</h1></body>
    </html>

---

apiVersion: v1
kind: Pod
metadata:
  name: configmap-example
spec:
  containers:
  - name: app-container
    image: nginx
    volumeMounts:
    - mountPath: "/usr/share/nginx/html"
      name: config-volume
  volumes:
  - name: config-volume
    configMap:
      name: example-config
```

ConfigMap (2/3)

- In this example a ConfigMap is exposed as environment variables in a container.
- It defines a container environment variable with data from a single or multiple ConfigMaps

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
data:
  setting1: "true"
  setting2: "value"

---

apiVersion: v1
kind: Pod
metadata:
  name: configmap-env-example
spec:
  containers:
    - name: app-container
      image: nginx
      env:
        - name: SETTING1
          valueFrom:
            configMapKeyRef:
              name: app-config
              key: setting1
        - name: SETTING2
          valueFrom:
            configMapKeyRef:
              name: app-config
              key: setting2
```

ConfigMap (3/3)

- We can configure all key-value pairs in a ConfigMap as container environment variables

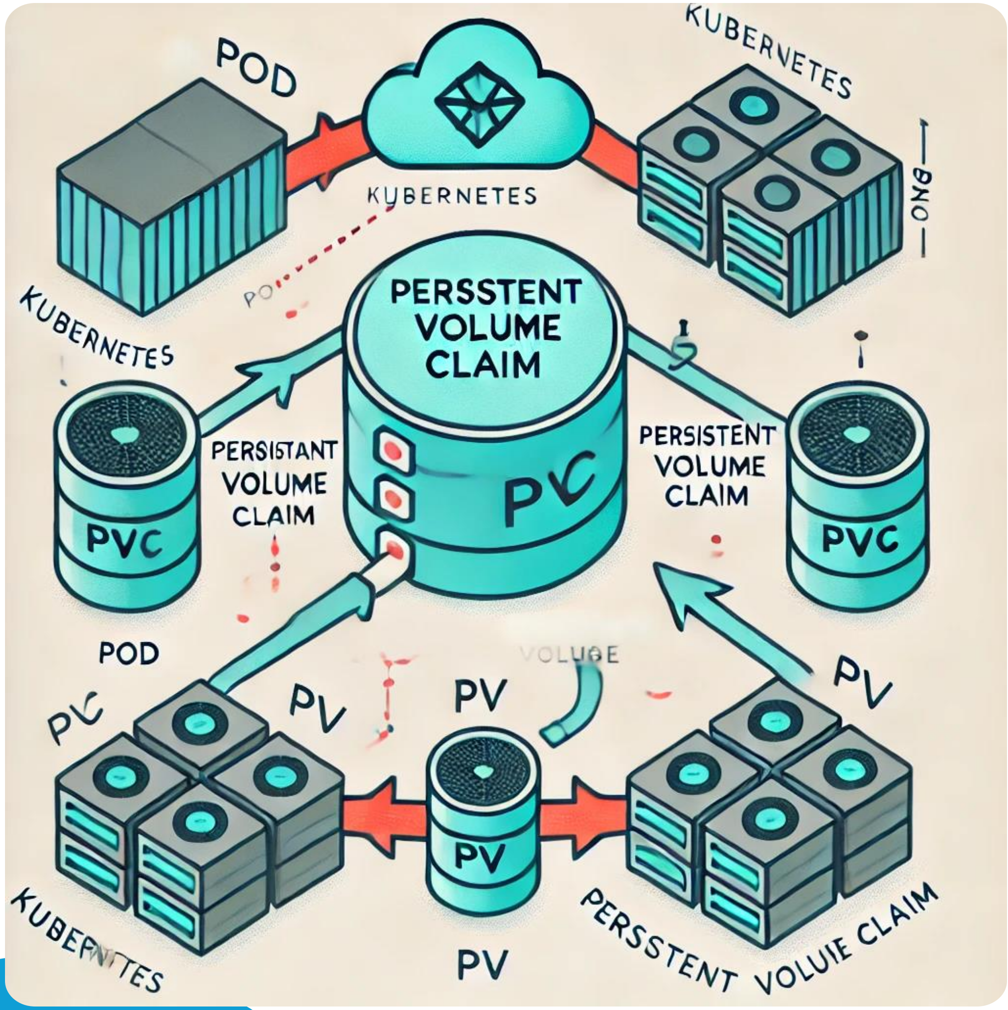
- **Use cases (recap):**
  - Storing configuration files and injecting them into containers via volumes.
  - Injecting configuration data as environment variables for easy access by applications running inside containers.
  - Useful for managing environment-specific settings without rebuilding images.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
data:
  setting1: "true"
  setting2: "value"

---

apiVersion: v1
kind: Pod
metadata:
  name: configmap-env-example
spec:
  containers:
    - name: app-container
      image: nginx
      envFrom:
       - configMapRef:
           name: app-config
```

Secrets

- `Secrets` are used to store sensitive data, such as passwords, OAuth tokens, or SSH keys.
- Secrets are not so secrets:
  - Secrets are **Base64 encoded:** this encoding is used to store binary data in a textual format, not for security purposes.
  - Secret values are stored **unencrypted in etcd** by default but can be configured to be encrypted.
- Best practices
  - Enable Secret encryption (**encryption at rest**)
  - Restrict access to Secrets using RBAC
  - Avoid exposing Secrets in environment variables
  - Use external Secret management solutions (e.g., HashiCorp Vault, Sealed Secrets)
- A Pod can reference the Secret in a variety of ways, such as in a volume mount or as an environment variable.

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: db-password
data:
  password: cGFzc3dvcmQ=  # base64 encoded password
---

apiVersion: v1
kind: Pod
metadata:
  name: secret-example
spec:
  containers:
    - name: app-container
      image: nginx
      volumeMounts:
        - mountPath: /etc/secret-volume
          name: secret-volume
  volumes:
    - name: secret-volume
      secret:
        secretName: db-password
```



Persistent storage

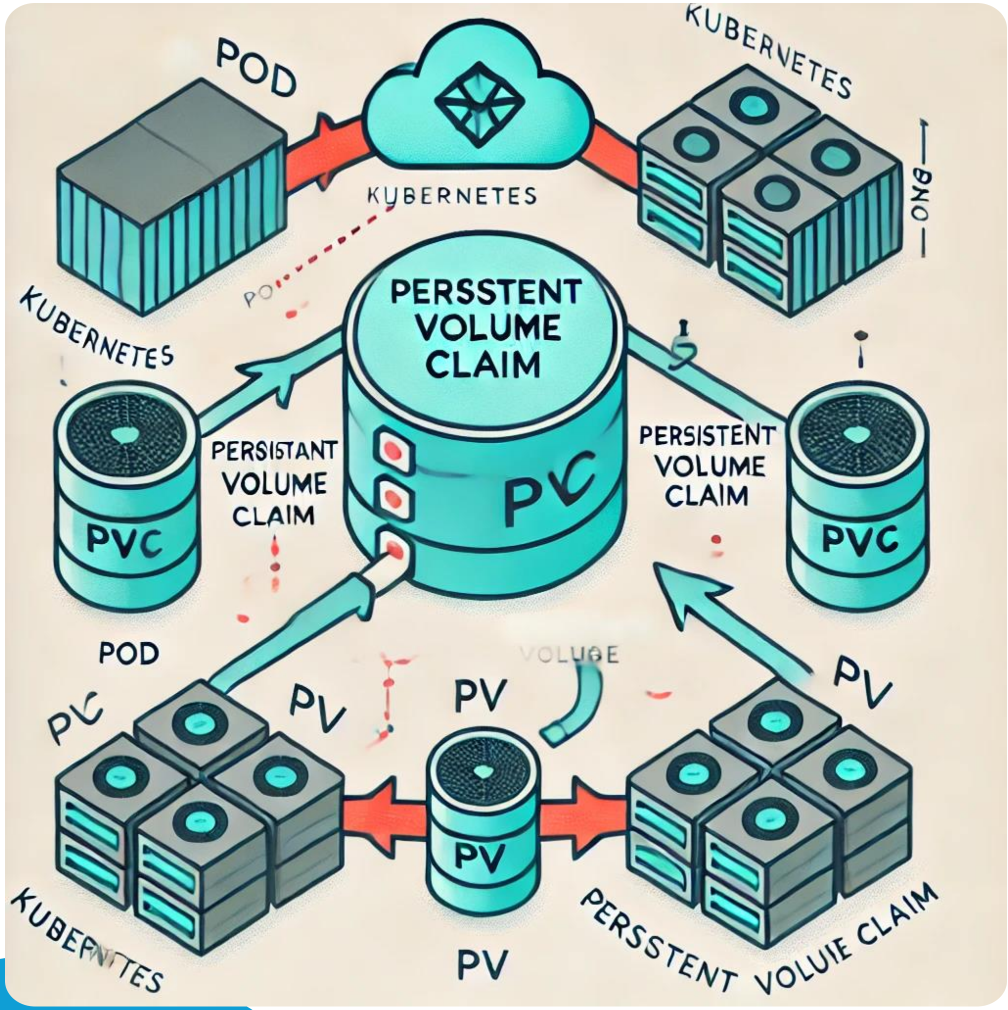
- Kubernetes offers two primary methods for accessing persistent storage, depending on how the storage is managed: at user or cluster level.
- **User-Level Access (via Volumes)**:
  Users can directly mount storage from external sources, such as an NFS server, by specifying the volume in the pod configuration. This approach allows users to connect to shared storage resources independently, without requiring intervention from the cluster administrator.
- **Cluster-Level Access (via PV and PVC)**:
  When storage is managed at the cluster level, Kubernetes uses the PersistentVolumes (PVs) and PersistentVolumeClaims (PVCs) mechanism to separate storage provisioning from pod lifecycle management. In this setup, the administrator defines PVs to represent the storage resources, while users request storage through PVCs. Kubernetes then binds the appropriate PVC to a suitable PV.

Feature	**Volumes**	**PersistentVolumes (PV)**
Management	Defined at the pod level by the user	Provisioned at the cluster level by the administrator
Persistence	Guaranteed only while the pod is running	Data remains available even after pod restarts or rescheduling
Scalability	Requires manual creation	Can be dynamically provisioned using StorageClasses
Storage Backend	Supports multiple storage backends (e.g., NFS, Ceph, AWS EBS)	Supports multiple storage backends (e.g., NFS, Ceph, AWS EBS)
Flexibility	Simple to use but tied to the pod	Decouples storage from pod lifecycle, allowing long-term data retention

User level persistent storage

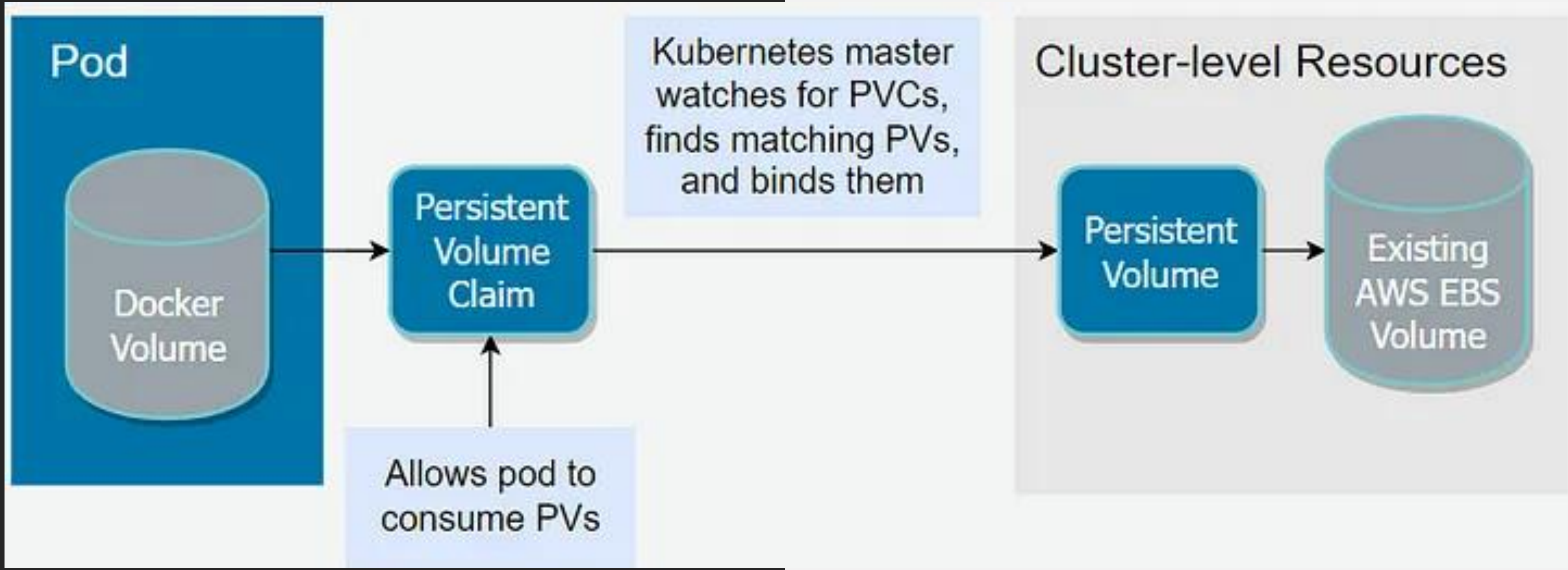
- Kubernetes allows users to directly access persistent storage at the pod level. We provide three examples of how users can access storage via different protocols: nfs, fibrechannel and cephfs
- **NFS (Network File System)**:
  Users can mount an NFS share directly into a pod, allowing multiple pods to access the same shared storage across nodes.
- **Fibre Channel**:
  Users can access high-performance storage via Fibre Channel, a low-latency, block-level network technology used to connect storage devices to servers.
- **Ceph**:
  Ceph provides highly scalable and reliable distributed storage. Users can mount Ceph volumes in a pod for block or object storage access.

```
<POD DEFINITION>

volumes:
  - name: nfs-data
    nfs:
      server: 10.64.56.102
      path: /data
  - name: fc-data
    fibreChannel:
      targetWWN: 20:00:00:00:00:00:00:00
      lun: 0
      fsType: ext4
  - name: ceph-data
    cephfs:
      monitors:
        - 10.64.56.101:6789
      path: /mnt/ceph
      user: admin
      secretRef:
        name: ceph-secret
```



Cluster level persistent storage

- When storage is managed at the cluster level, Kubernetes leverages the PersistentVolumes (PVs) and PersistentVolumeClaims (PVCs) mechanism to decouple storage provisioning from pod lifecycle management.
- **Persistent Volume (PV):** A storage resource provisioned at the cluster level by an administrator. A PV represents a physical storage resource that exists independently of any specific pod.
  - **Decouples storage from pod lifecycle**, ensuring data durability.
  - **Supports multiple storage backends**, including NFS, Ceph, AWS EBS, and GCE Persistent Disk.
  - **Enables storage sharing across pods**, depending on the access mode.
- **PersistentVolumeClaim (PVC):** A request for storage made by users or applications. When a PVC is created, Kubernetes automatically searches for an available PV that matches the specified requirements and binds them together.

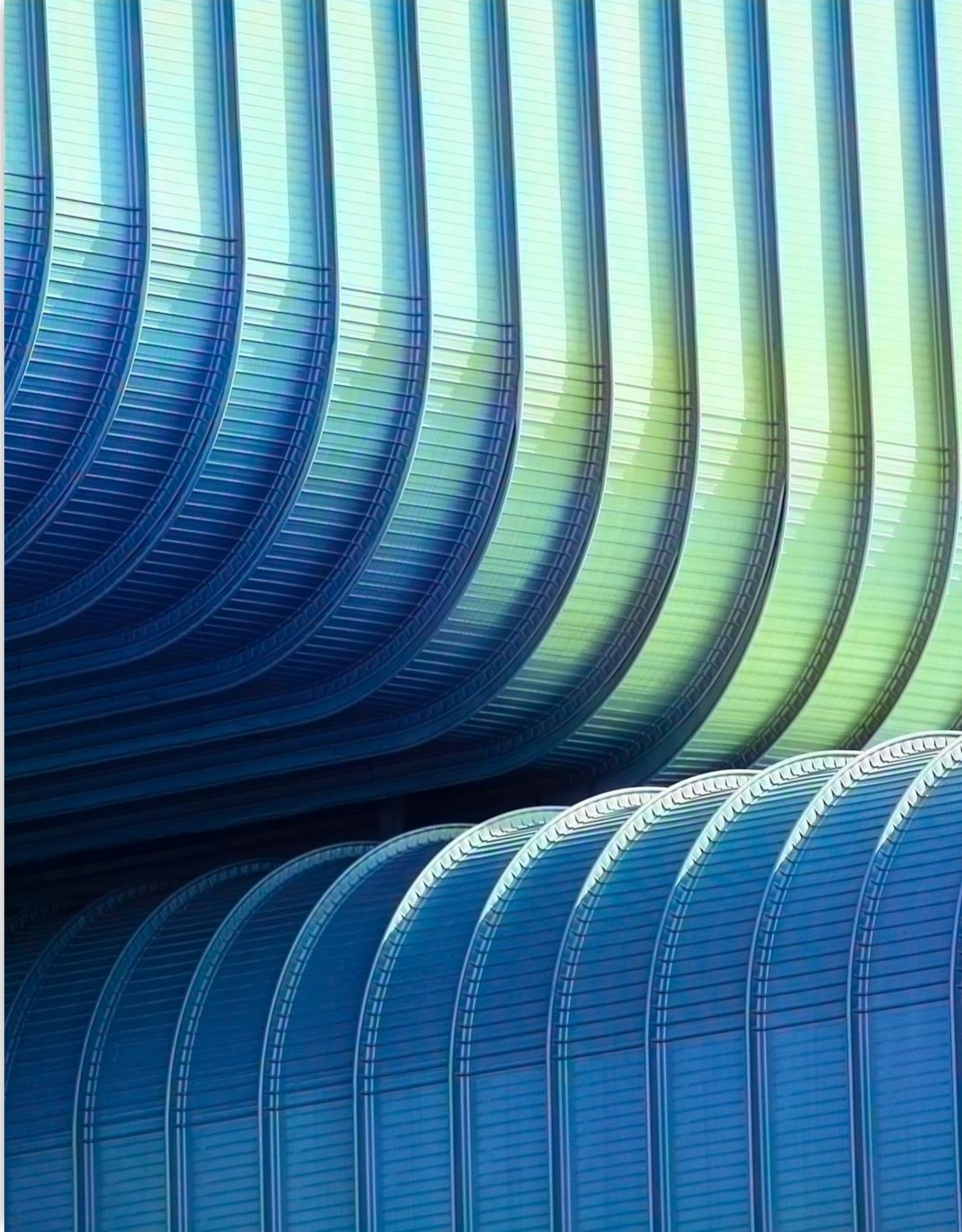


How PV and PVC work together

- **Request and Provision**: users create a PVC specifying their storage requirements.
- **Binding process**: Kubernetes finds a suitable PV and binds it to the PVC.
- **Access**: the PVC can then be used by pods to mount the storage defined by the PV.

PV provisioning methods

- Kubernetes supports two methods for provisioning Persistent Volumes (PVs), depending on the storage management approach:
- **Static Provisioning**
  - The administrator manually creates PVs in advance.
  - Each PV is configured with specific capacity, access modes, and storage backend.
  - Users create PersistentVolumeClaims (PVCs) that Kubernetes binds to an existing PV matching the request.
- **Dynamic Provisioning**
  - Kubernetes automatically provisions storage on demand using a **StorageClass**.
  - Users request storage through a PVC, specifying the required **StorageClass**.
  - Kubernetes creates a PV dynamically, ensuring flexibility and efficient resource allocation.
- This approach allows **predefined** storage (static) when control is needed and **automated** provisioning (dynamic) for scalability and ease of management.



Understanding PV

- Let's look at an example of a **statically provisioned PV**.
- Each PV contains a spec and status, which is the specification and status of the volume.
- **Key elements:**
  - capacity: Defines the storage size (e.g., 10Gi)
  - volumeMode (optional): Filesystem (default) and Block
  - accessModes: Access modes for the volume (ReadWriteOnce, ReadWriteOncePod, ReadWriteMany, ReadOnlyMany).
  - persistentVolumeReclaimPolicy: Policy when PVC is deleted (Retain/Delete/Reclaim).
  - StorageClassName (optional): Associates the PV with a storage class (optional).
  - hostPath: Path on the physical node (development and testing only). In production use NFS, AWS EBS, Ceph, etc.

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: static-pv
spec:
  capacity:
    storage: 10Gi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteOncePod
  persistentVolumeReclaimPolicy: Retain
  storageClassName: manual
  hostPath:
    path: /data/static-pv
```

- A static `hostPath` volume is created, with 10Gi of storage and access mode **`ReadWriteOncePod`**.
- The volume is created on a node at the path `/data/static-pv`.

```
# kubectl get pv static-pv
NAME        CAPACITY   ACCESS MODES    RECLAIM POLICY    STATUS CLAIM   SC
static-pv   10Gi       RWOP            Retain            Available      manual
```

Example NFS PV

- This **PV** configuration defines a **network storage resource**, using an **NFS (Network File System)** backend. The PV allows multiple nodes to read and write data simultaneously.
- **Capacity:** the PV provides **20Gi** of storage.
- **Access Mode:** set to `ReadWriteMany (RWX)`, meaning multiple pods across different nodes can access and write to the volume.
- **Reclaim Policy:** configured as `Delete`, ensuring that the data is deletedwhen the PV is released.
- **Storage Backend:** uses **NFS** with the server at `nfs-server.example.com`, and the storage path is `/mnt/data`.
- This configuration is ideal for applications that require **shared storage**, such as distributed workloads or shared data repositories.

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: nfs-pv
spec:
  capacity:
    storage: 20Gi
  accessModes:
    - ReadWriteMany
  persistentVolumeReclaimPolicy: Delete
  nfs:
    path: /mnt/data
    server: nfs-server.example.com
```

Understanding PVC

- A **PVC** is a request for storage made by a user or application.
- **Key elements:**
  - accessModes: Should match the access modes defined in the PV.
  - resources.requests.storage: The amount of storage requested.
  - storageClassName: Must match the PV's storage class for proper binding (if defined!).

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: static-pvc
spec:
  accessModes:
    - ReadWriteOncePod
  resources:
    requests:
      storage: 10Gi
  storageClassName: manual
```

```
# kubectl get pv static-pv
NAME        CAPACITY   ACCESS MODES     RECLAIM POLICY   STATUS CLAIM       SC
static-pv   10Gi       RWOP             Retain           Bound static-pvc   manual

# kubectl get pvc static-pvc
NAME          STATUS    VOLUME      CAPACITY    ACCESS MODES   SC
static-pvc    Bound     static-pv   10Gi        RWOP           manual
```

How to mount PVC as Volumes

- Pods access storage by using the claim as a volume.
- Claims must exist in the same namespace as the Pod using the claim.
- The cluster finds the claim in the Pod's namespace and uses it to get the PersistentVolume backing the claim.
- The volume is then mounted to the host and into the Pod.

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
    - name: myfrontend
      image: nginx
      volumeMounts:
      - mountPath: "/var/www/html"
        name: my-vol
  volumes:
    - name: my-vol
      persistentVolumeClaim:
        claimName: static-pvc
```

StorageClass
(dynamic provisioning)

- A **StorageClass** defines the **storage provisioning strategy** in Kubernetes.
- It is a Kubernetes abstraction that defines the characteristics of **dynamic storage**.
- It allows dynamic provisioning of **PVs** based on demand.
- Each StorageClass uses a **provisioner** (e.g., AWS EBS, GCE PD, NFS, Ceph, etc.) and its options.

StorageClass example

- The provided **StorageClass** example defines the use of **AWS EBS** (via the provisioner kubernetes.io/aws-ebs), specifically using the gp2 storage type.
- This **StorageClass** is set as the **default**, meaning it will be automatically selected for PVCs that do not explicitly specify a storageClassName.
- The **reclaimPolicy** is set to Retain, ensuring that the volume will remain even if the associated PVC is deleted.
- The **volumeBindingMode** is set to Immediate, meaning the volume is bound to the PVC as soon as the PVC is created, without any delay.
- Additionally, **volume expansion** is enabled, allowing the volume size to be adjusted as needed, offering flexibility for storage requirements over time.

```yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: aws-ebs
  annotations:
    storageclass.kubernetes.io/is-default-class: "true"
provisioner: kubernetes.io/aws-ebs  #deprecated
parameters:
  type: gp2
  fsType: ext4
reclaimPolicy: Retain # or Delete, Recycle
volumeBindingMode: Immediate # or WaitForFirstConsumer
allowVolumeExpansion: true

---

apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: my-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
  storageClassName: aws-ebs
```

Driver	Description	Access Modes	Status
HostPath	Uses a local directory on the node. Suitable only for testing.	RWO	Active, but not recommended for production
NFS	Shares storage across multiple nodes using NFS.	RWO, RWX, ROX	Active, but CSI-NFS is recommended
iSCSI	Network block storage.	RWO (RWX requires FS-level management)	Active, but CSI-iSCSI is recommended
Ceph RBD	Block storage on Ceph.	RWO	Active, but CSI-RBD is recommended
CephFS	File-based storage on Ceph, supports multiple access.	RWO, RWX, ROX	Active, but CSI-CephFS is recommended
AWS EBS	Block storage on AWS.	RWO	Deprecated in favor of CSI-EBS
Google Persistent Disk (PD)	Block storage on GCP.	RWO	Deprecated in favor of CSI-PD
Azure Disk	Block storage on Azure.	RWO	Deprecated in favor of CSI-AzureDisk
Azure File	SMB file system on Azure.	RWO, RWX, ROX	Deprecated in favor of CSI-AzureFile
GlusterFS	Distributed file system based on Gluster.	RWO, RWX, ROX	**Deprecated** (removed in v1.28)
Flocker	Block storage for Kubernetes.	RWO	**Removed in v1.22**
OpenStack Cinder	Block storage for OpenStack.	RWO	Deprecated in favor of CSI-Cinder

In-tree drivers

- Kubernetes supports various types of PV, each with different characteristics and use cases.
- Kubernetes manages the two kind of storage plugins (drivers) that handle these PVs: **in-tree** and **out-of-tree**
- **In-tree drivers**
  - Integrated directly into Kubernetes' source code.
  - Each driver is part of the core system, meaning updates and modifications depend on Kubernetes release cycles.
  - **Examples:** awsElasticBlockStore, gcePersistentDisk, azureDisk.
- **Almost all deprecated or in the process of removal**
- **Migration to out-of-tree (CSI) drivers is strongly recommended!**

Driver	Description	Access Modes	Status
AWS EBS (csi-ebs)	CSI driver for AWS Elastic Block Store.	RWO	Actively maintained
AWS EFS (csi-efs)	CSI driver for AWS Elastic File System.	RWX	Actively maintained
Google PD (csi-gcepd)	CSI driver for Google Persistent Disk.	RWO	Actively maintained
Google Filestore (csi-gcs)	Managed file storage for GCP.	RWX	Actively maintained
Azure Disk (csi-azuredisk)	CSI driver for Azure Managed Disks.	RWO	Actively maintained
Azure File (csi-azurefile)	CSI driver for Azure Files (SMB/NFS).	RWX, ROX	Actively maintained
Ceph RBD (csi-rbd)	CSI driver for Ceph RADOS Block Device.	RWO	Actively maintained
CephFS (csi-cephfs)	CSI driver for CephFS (file-based storage).	RWX	Actively maintained
NFS (csi-nfs)	CSI driver for NFS-based storage.	RWX	Actively maintained
iSCSI (csi-iscsi)	CSI driver for iSCSI-based block storage.	RWO	Actively maintained
GlusterFS (csi-gluster)	CSI driver for GlusterFS.	RWX	Actively maintained
OpenStack Cinder (csi-cinder)	CSI driver for OpenStack block storage.	RWO	Actively maintained
VMware vSphere (csi-vsphere)	CSI driver for VMware vSphere storage.	RWO, RWX	Actively maintained
Rook (rook-ceph, rook-edgefs, rook-nfs)	CSI drivers for Ceph, EdgeFS, and NFS via Rook.	RWO, RWX	Actively maintained
Portworx (csi-portworx)	CSI driver for Portworx software-defined storage.	RWO, RWX	Actively maintained
Longhorn (csi-longhorn)	Lightweight, cloud-native distributed block storage.	RWO	Actively maintained
StorageOS (csi-storageos)	CSI driver for StorageOS, a Kubernetes-native storage solution.	RWO, RWX	Actively maintained
NetApp Trident (csi-trident)	CSI driver for NetApp [illegible]rage systems.	RWO, RWX	Actively maintained

Out-of-tree drivers

- Out-of-tree drivers are implemented using **CSI (Container Storage Interface)**, which has become the standard for storage integrations in Kubernetes.
- These drivers are maintained separately from Kubernetes and provide better flexibility, security, and compatibility with modern storage solutions.
- Updates independent of Kubernetes release cycles.
- Each CSI driver has specific installation steps. Check the official documentation for the correct YAML manifests, Helm charts, or operator-based deployments.

Thanks!

References

- https://kubernetes.io/docs/concepts/storage/volumes/
- https://kubernetes.io/docs/concepts/storage/persistent-volumes/
- https://kubernetes.io/docs/concepts/storage/ephemeral-volumes/
- https://kubernetes.io/docs/concepts/storage/storage-classes/
- https://kubernetes.io/docs/concepts/storage/dynamic-provisioning/
- https://medium.com/geekculture/storage-kubernetes-92eb3d027282
- https://kubernetes.io/docs/concepts/security/secrets-good-practices/
- https://kubernetes.io/docs/tasks/administer-cluster/encrypt-data/#ensure-all-secrets-are-encrypted
- https://medium.com/@martin.hodges/adding-persistent-storage-to-your-kubernetes-cluster-5e12adb81592