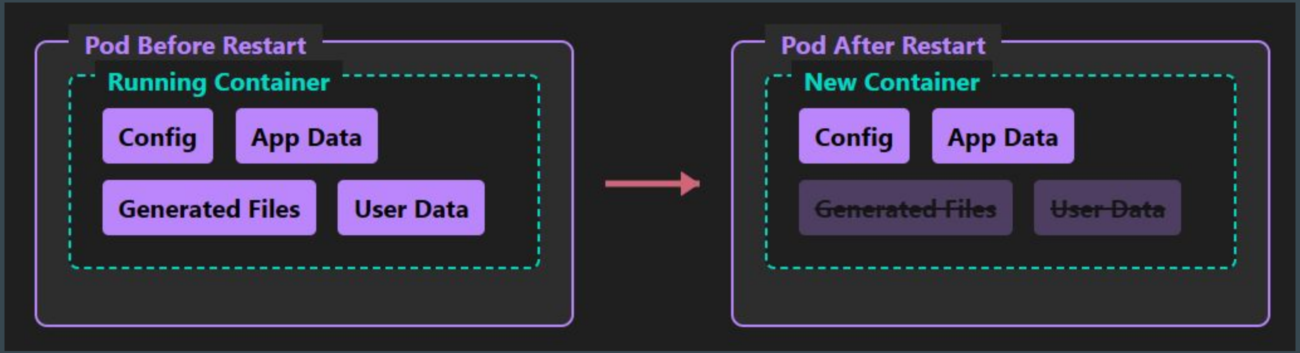
Volumes in Kubernetes

Understanding Challenge - State Persistence

When a container crashes or restarted, the container state is not saved so all of the files that were created during the lifetime of the container are lost.

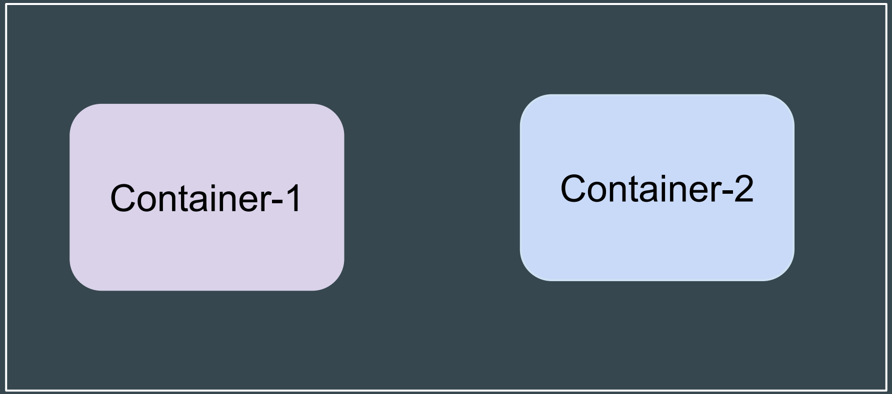
After a crash, kubelet restarts the container with a clean state.



Understanding Challenge - Shared Storage

Another problem occurs when multiple containers are running in a Pod and need to share files.

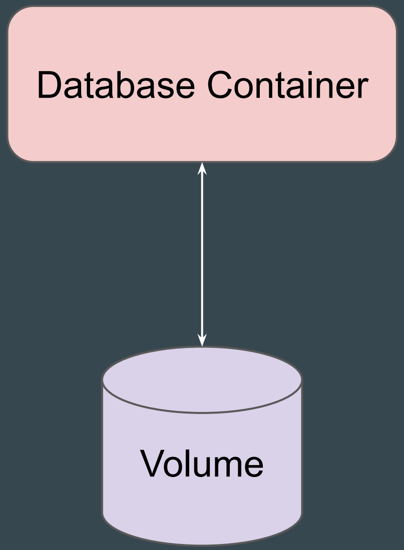
It can be challenging to set up and access a shared filesystem across all of the containers.



Multi-Container Pod

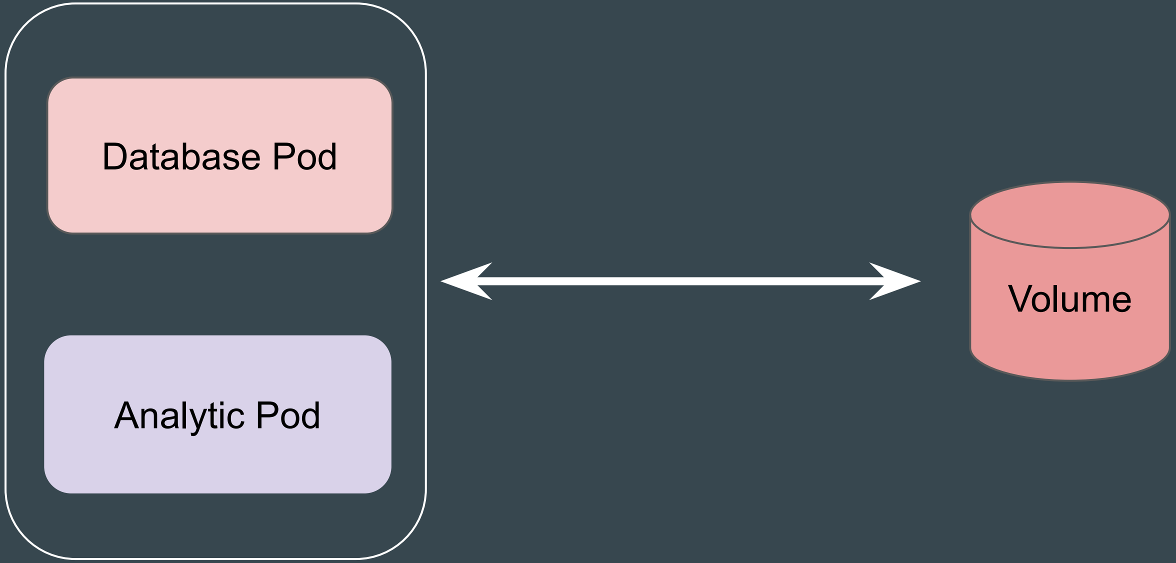
Introducing Volumes

Volumes can provide a way to store data that persists beyond the lifecycle of a container.



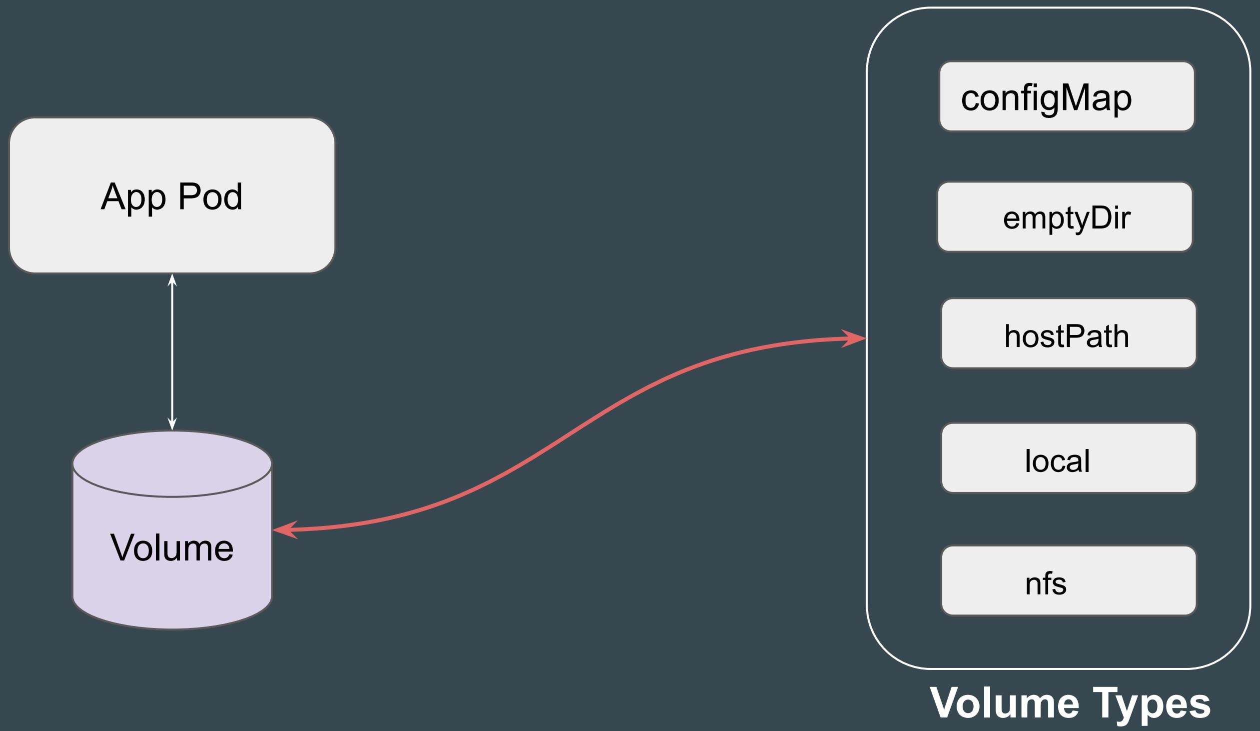
Benefit - Shared Storage

A single volume can be attached to multiple Pods.



Kubernetes Volumes

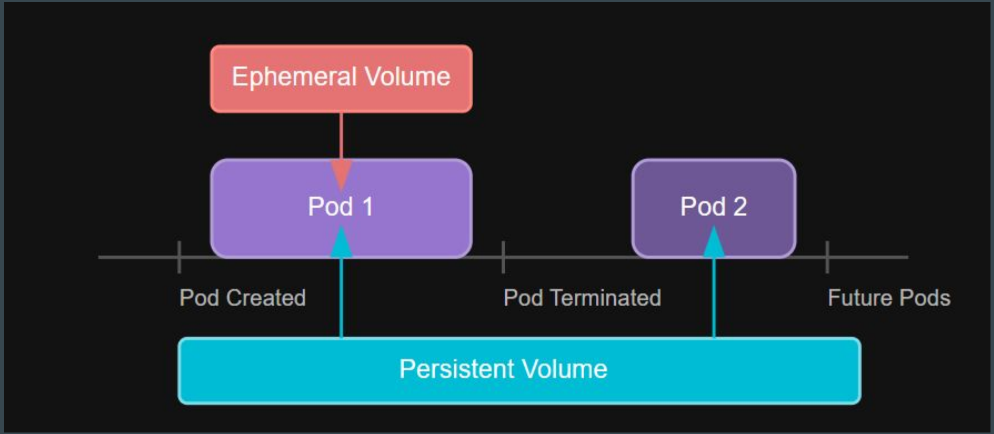
Kubernetes offers many volume types depending on the use-case.

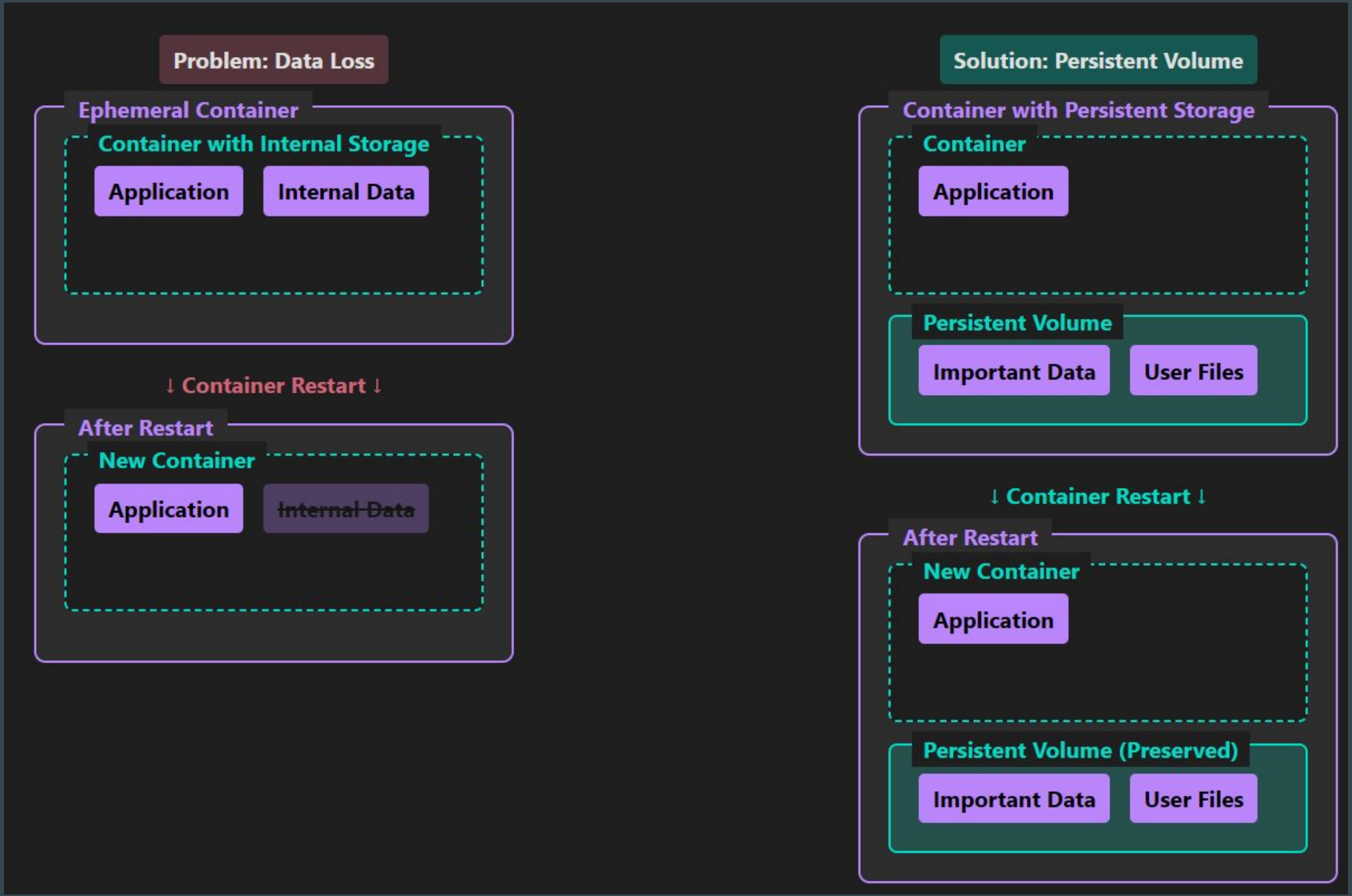


Ephemeral and Persistent Volumes

Ephemeral volume types have a lifetime linked to a specific Pod, BUT persistent volumes exist beyond the lifetime of any individual pod.

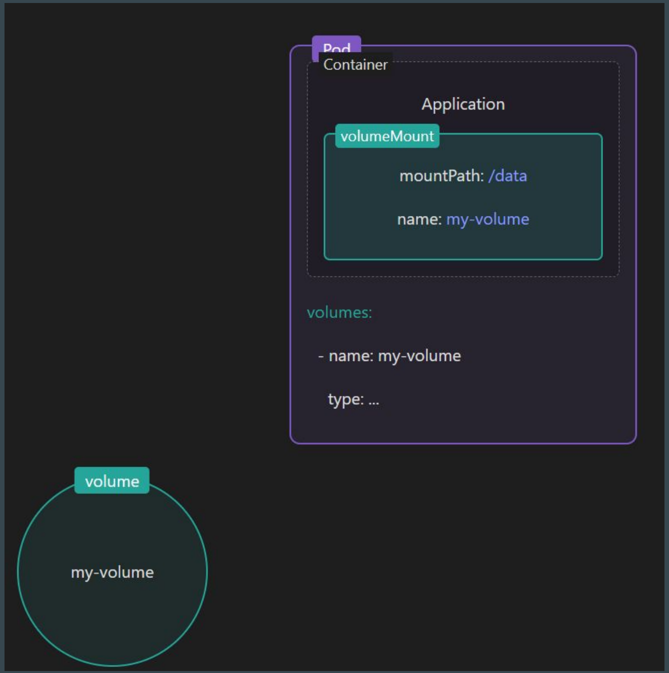
When a pod ceases to exist, Kubernetes destroys ephemeral volumes; however, Kubernetes does not destroy persistent volumes.

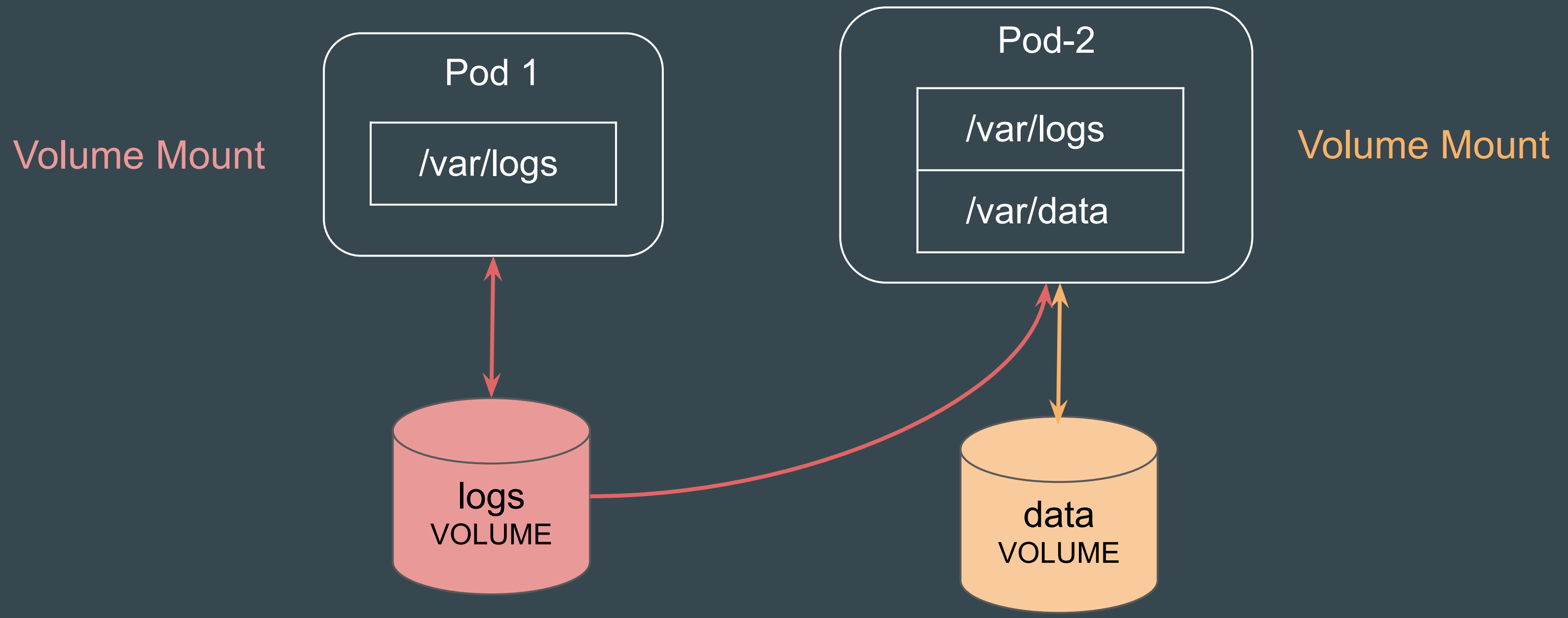



Problem: Data Loss
Ephemeral Container
Container with Internal Storage
Application
Internal Data
↓ Container Restart ↓
After Restart
New Container
Application
~~Internal Data~~
Solution: Persistent Volume
Container with Persistent Storage
Container
Application
Persistent Volume
Important Data
User Files
↓ Container Restart ↓
After Restart
New Container
Application
Persistent Volume (Preserved)
Important Data
User Files

Volume and Volume Mounts

To use a volume, specify the volumes to provide for the Pod in .spec.volumes and declare where to mount those volumes into containers in .spec.containers[*].volumeMounts.




Volume Mount
Pod 1
/var/logs
Pod-2
/var/logs
/var/data
Volume Mount
logs
VOLUME
data
VOLUME

Sample Reference Code

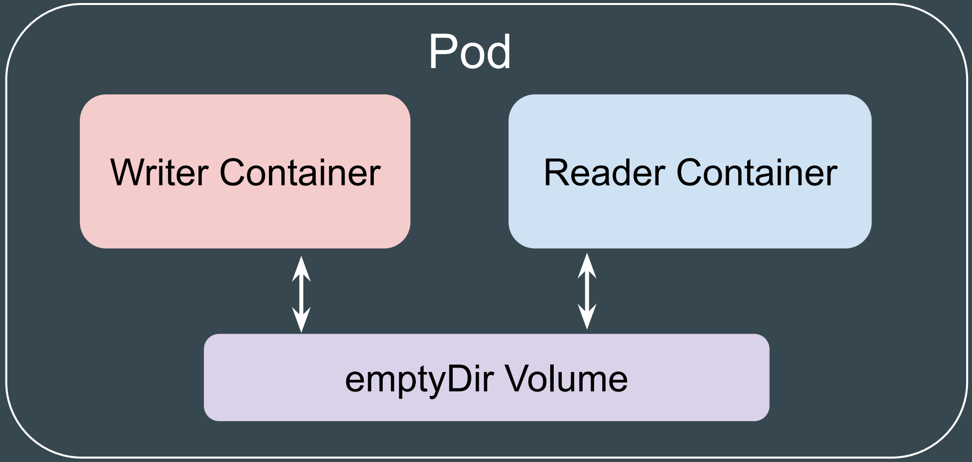
```yaml
apiVersion: v1
kind: Pod
metadata:
  name: test-pd
spec:
  containers:
  - image: nginx:latest
    name: test-container
    volumeMounts:
    - mountPath: /my-data
      name: data-volume
  volumes:
  - name: data-volume
    emptyDir:
      sizeLimit: 500Mi
```

Volume Type - emptyDir

Setting the Base

An emptyDir volume is a temporary storage directory.

All containers in the Pod can read and write the same files in the emptyDir volume.



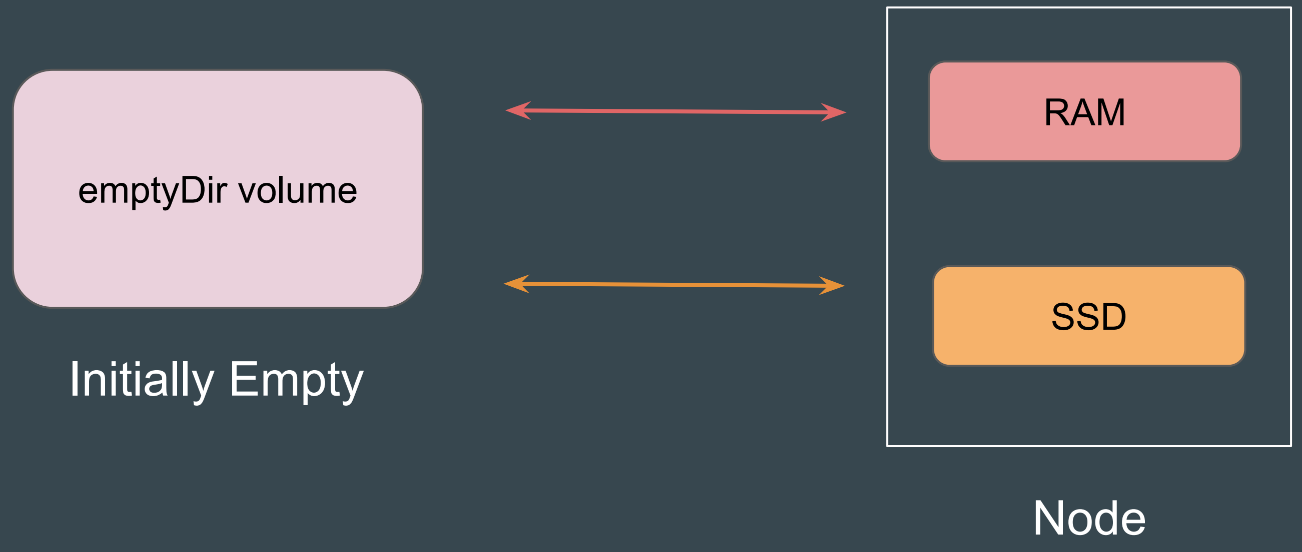
Key Features of emptyDir

It is deleted when the pod is removed.

It can use memory instead of disk for performance.

A container crashing does not remove a Pod from a node. The data in an emptyDir volume is safe across container crashes.

Workflow - EmptyDir



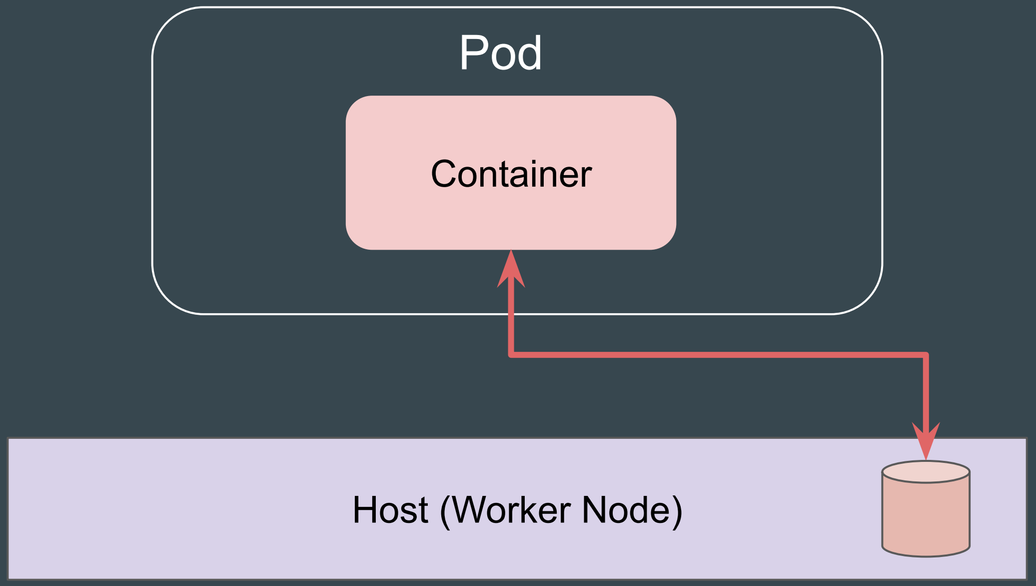
```yaml
apiVersion: v1
kind: Pod
metadata:
  name: emptydir-demo
spec:
  volumes:
    - name: shared-storage
      emptyDir: {}
  containers:
    - name: busybox-container-1
      image: busybox
      command: ["sleep", "36000"]
      volumeMounts:
        - name: shared-storage
          mountPath: /data

    - name: busybox-container-2
      image: busybox
      command: ["sleep", "36000"]
      volumeMounts:
        - name: shared-storage
          mountPath: /data
```

Volume Type - hostPath

Setting the Base

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



Use-Cases of hostPath

Container needing access to worker node-specific logs like /var/logs for analysis.

Container that needs access to worker node-specific configuration files

Container that wants to write persistent data to a specific path in the node.

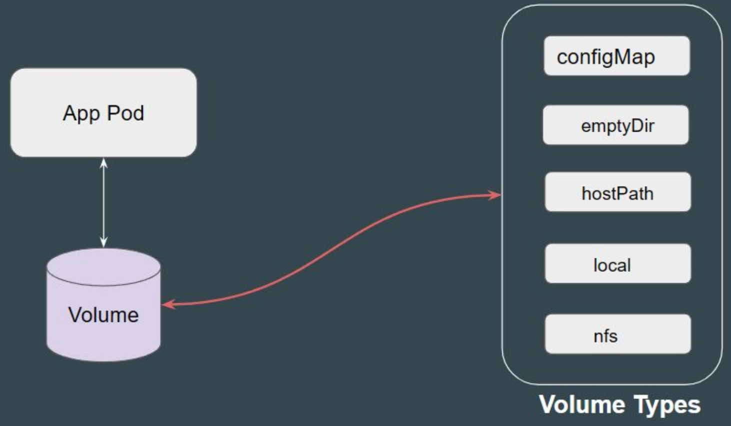
PersistentVolume and PersistentVolumeClaim

Setting the Base

Developers deploy application pods based on their their requirements.

Supplying storage specific configurations can introduce complexity due to the many different storage types and their settings

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nfs-client-pod
spec:
  containers:
    - name: app-container
      image: nginx
      volumeMounts:
        - name: nfs-volume
          mountPath: /mnt/nfs
  volumes:
    - name: nfs-volume
      nfs:
        server: 192.168.1.100
        path: /exported/path
        readOnly: false
```



Developer

Being a developer, my goal is to create a straightforward Pod manifest that includes volume information. I'd prefer not to handle storage provisioning and its associated configurations.

Storage Administrator

As a Storage Administrator, I am responsible for provisioning storage and managing its configurations. Developers can then reference this storage within their Pod specifications.

Overview of Persistent Volume

Persistent Volume is a piece of storage in the cluster that has been provisioned by an administrator or dynamically provisioned using Storage Classes.

Every volume is created can be of different type and specifications.

Volume 1
Size: Small
Speed: Fast

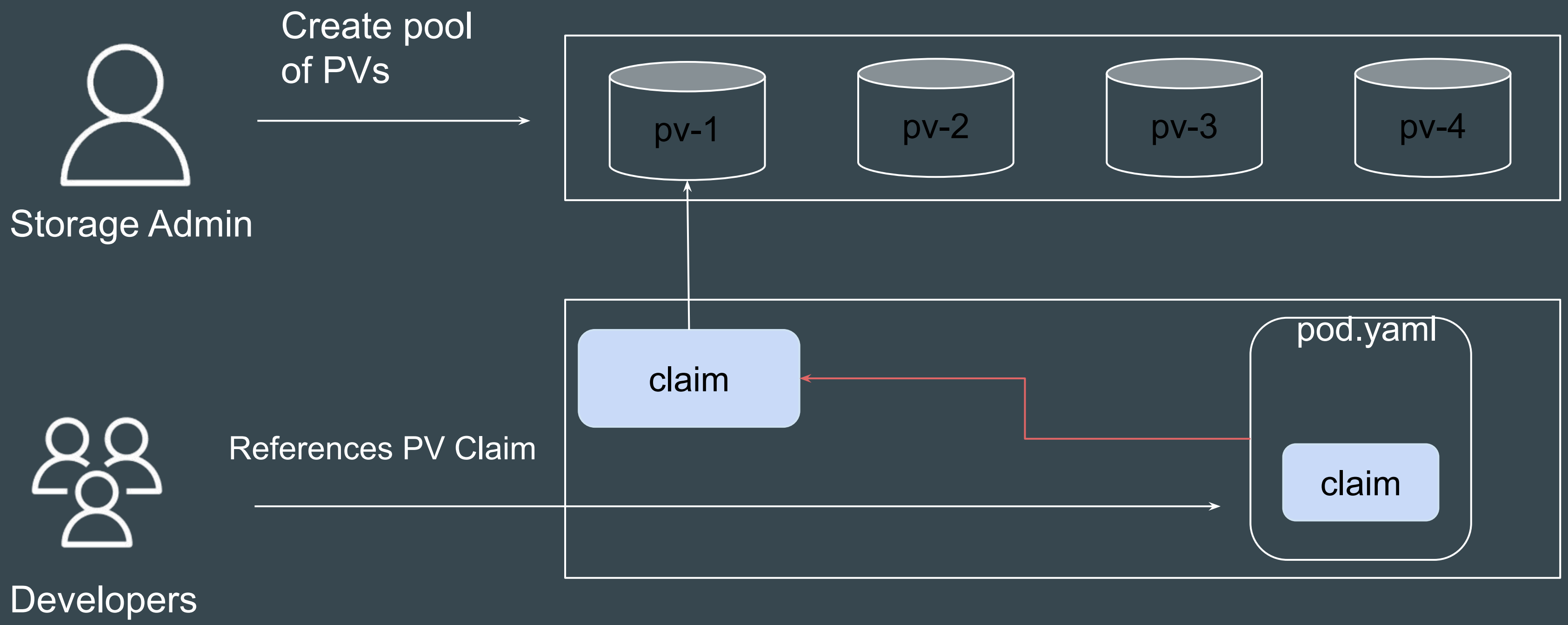
Volume 2
Size: Medium
Speed: Fast

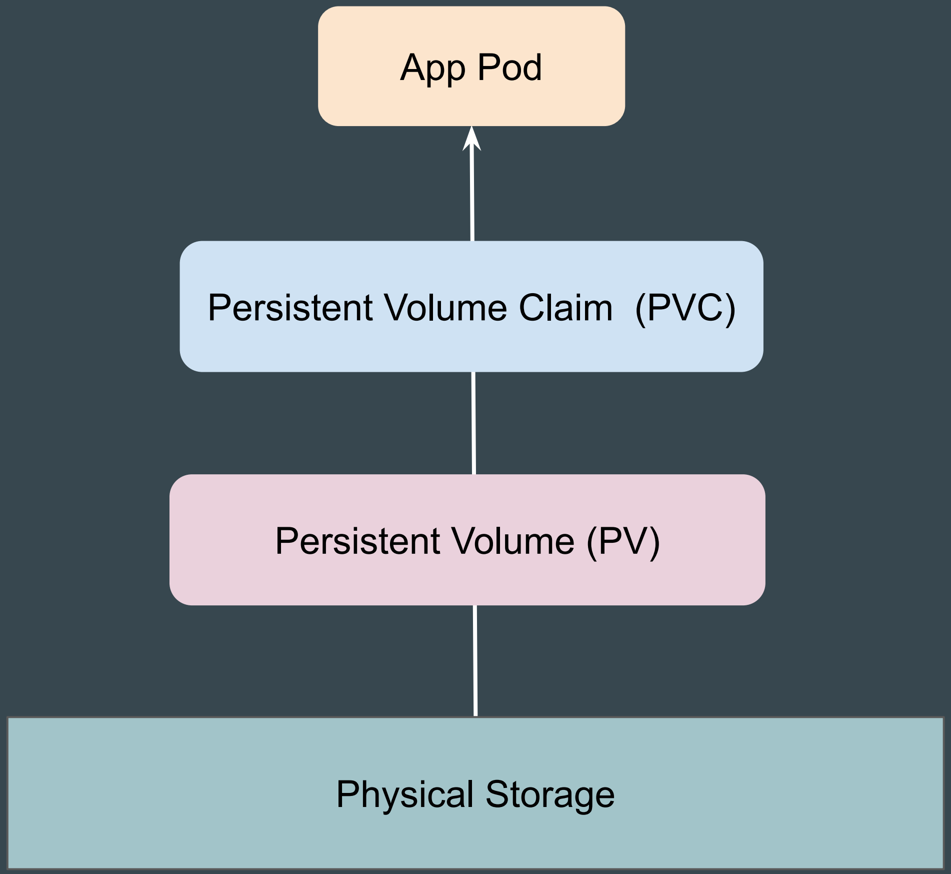
Volume 3
Size: Big
Speed: Slow

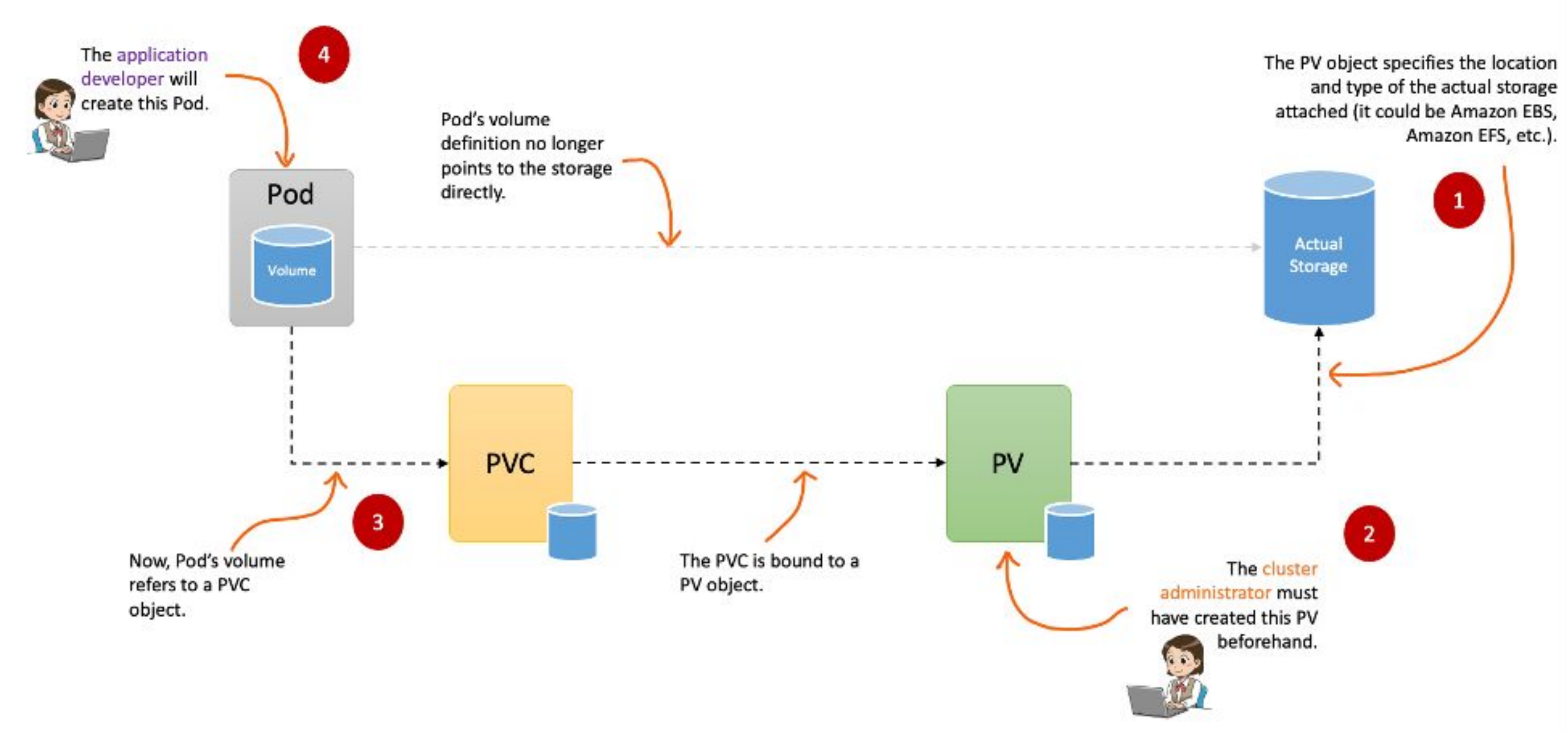
Volume 4
Size: VSmall
Speed: Ultra Fast

PersistentVolumeClaim

PersistentVolumeClaim (PVC) is a request for storage by a user (e.g., a developer). It specifies size, access modes, and other requirements.



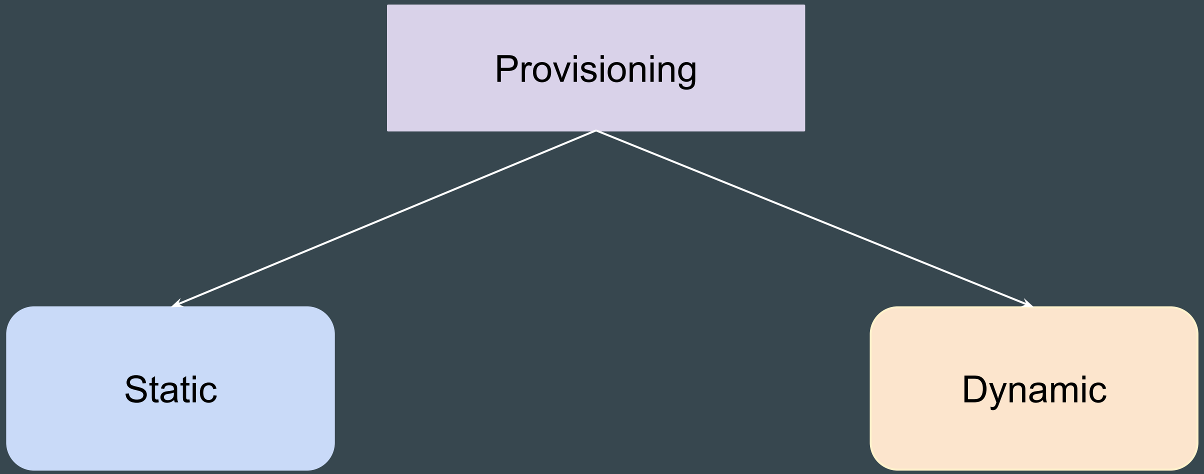

App Pod
Persistent Volume Claim (PVC)
Persistent Volume (PV)
Physical Storage



The application developer will create this Pod.
4
Pod's volume definition no longer points to the storage directly.
The PV object specifies the location and type of the actual storage attached (it could be Amazon EBS, Amazon EFS, etc.).
Pod
Volume
1
Actual Storage
PVC
PV
3
2
Now, Pod's volume refers to a PVC object.
The PVC is bound to a PV object.
The cluster administrator must have created this PV beforehand.

Persistent Volumes - Static vs Dynamic Provisioning

Provisioning of Persistent Volumes



PV must be created before PVC

PV is created dynamically at same type of PVC.

Basic of Static Provisioning

Admin manually creates PVs ahead of time.

Users create PVCs that match the available PVs.

If no matching PV exists, the PVC remains unbound.

```
C:\kplabs-k8s>kubectl get pv
NAME             CAPACITY    ACCESS MODES    RECLAIM POLICY    STATUS       CLAIM    STORAGECLASS
manual-pv-1gb    1Gi         RWO             Retain            Available             manual
manual-pv-3gb    3Gi         RWO             Retain            Available             manual
```

Basic of Dynamic Provisioning

With dynamic provisioning, you do not have to create a PV object.

Instead, it will be automatically created under the hood when you create the PVC. Kubernetes does so using another object called Storage Class

```
C:\>kubectl get pvc
NAME          STATUS    VOLUME                                     CAPACITY   ACCESS MODES   STORAGECLASS
SCLASS    AGE
do-pvc-1gb    Bound     pvc-86690381-df9a-4966-be5b-db45c61971c8   1Gi        RWO            do-block-storage
```

Storage Class

StorageClass specifies the plugin or driver that determines how persistent volumes (PVs) are dynamically provisioned.

They contain appropriate provisioner used to provision volumes.

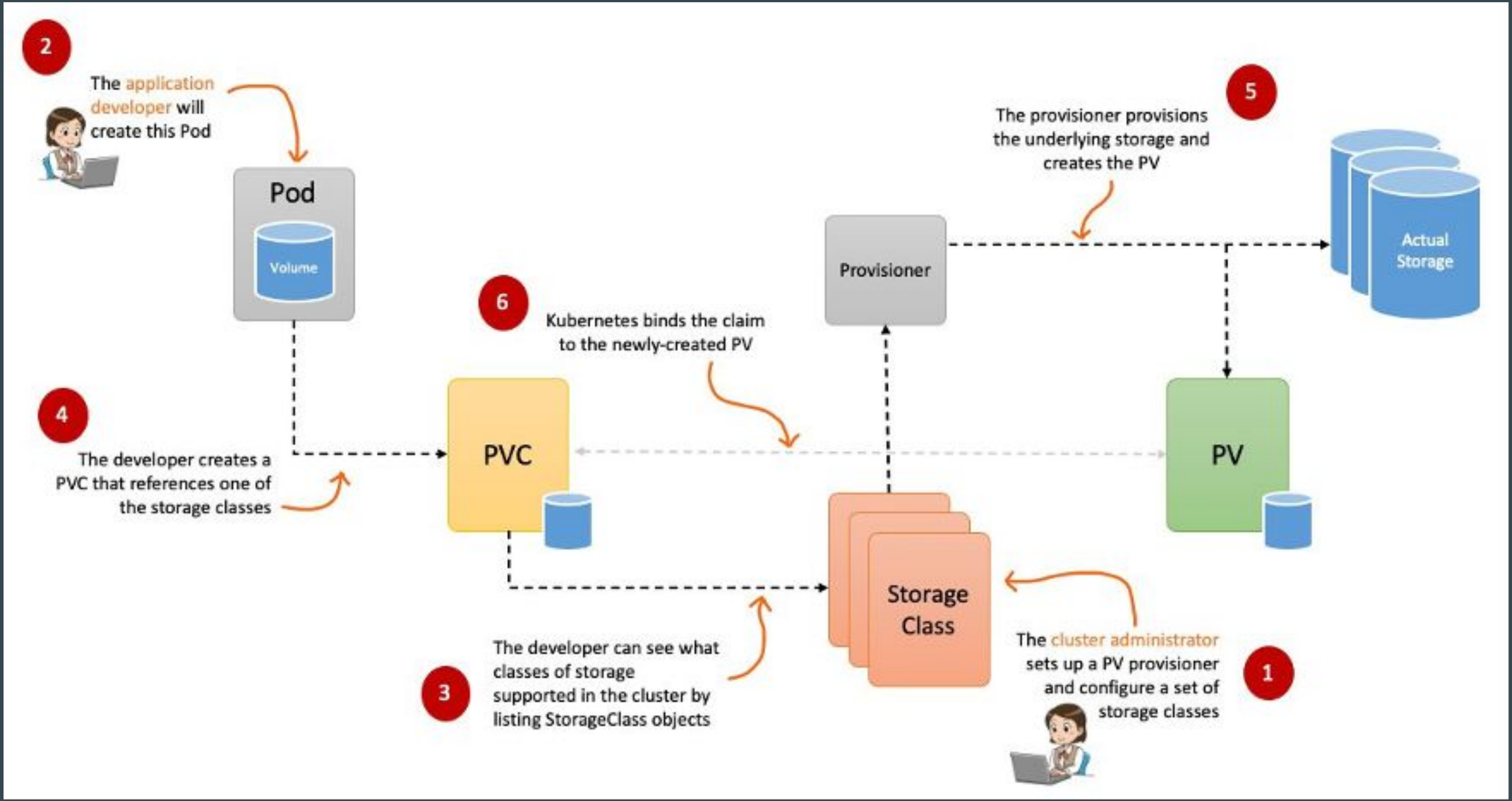
```
C:\>kubectl get storageclass
NAME                          PROVISIONER                 RECLAIMPOLICY   VOLUMEBINDINGMODE   ALLOWVOLUMEEXPANSION
do-block-storage (default)    dobs.csi.digitalocean.com   Delete          Immediate           true
do-block-storage-retain       dobs.csi.digitalocean.com   Retain          Immediate           true
do-block-storage-xfs          dobs.csi.digitalocean.com   Delete          Immediate           true
do-block-storage-xfs-retain   dobs.csi.digitalocean.com   Retain          Immediate           true
```

Dynamic Provisioning using Local Path Provisioner

Setting the Base

The Local Path Provisioner offered by Rancher is a lightweight and simple dynamic storage provisioner for Kubernetes that uses local storage on the host node to provide PersistentVolumes (PVs).

```
root@kubeadm:~# kubectl get storageclass
NAME         PROVISIONER             RECLAIMPOLICY    VOLUMEBINDINGMODE      ALLOWVOLUMEEXPANSION   AGE
local-path   rancher.io/local-path   Delete           WaitForFirstConsumer   false                  15h
```


2
The application developer will create this Pod
Pod
Volume
5
The provisioner provisions the underlying storage and creates the PV
Actual Storage
Provisioner
6
Kubernetes binds the claim to the newly-created PV
4
The developer creates a PVC that references one of the storage classes
PVC
PV
Storage Class
The developer can see what classes of storage supported in the cluster by listing StorageClass objects
3
The cluster administrator sets up a PV provisioner and configure a set of storage classes
1

Storage Class

Setting the Base

StorageClass specifies the plugin or driver that determines how persistent volumes (PVs) are dynamically provisioned.

They contain appropriate provisioner used to provision volumes.

```
C:\>kubectl get storageclass
NAME                          PROVISIONER                 RECLAIMPOLICY   VOLUMEBINDINGMODE   ALLOWVOLUMEEXPANSION
do-block-storage (default)    dobs.csi.digitalocean.com   Delete          Immediate           true
do-block-storage-retain       dobs.csi.digitalocean.com   Retain          Immediate           true
do-block-storage-xfs          dobs.csi.digitalocean.com   Delete          Immediate           true
do-block-storage-xfs-retain   dobs.csi.digitalocean.com   Retain          Immediate           true
```

Understanding the Structure

Each StorageClass contains the fields provisioner, parameters, and reclaimPolicy, which are used when a PersistentVolume belonging to the class needs to be dynamically provisioned to satisfy a PersistentVolumeClaim (PVC).

```yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: low-latency
provisioner: csi-driver.example-vendor.example
allowVolumeExpansion: true
volumeBindingMode: WaitForFirstConsumer
```

Default Storage Class

You can mark a StorageClass as the default for your cluster.

When a PVC does not specify a storageClassName, the default StorageClass is used.

```yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: low-latency
  annotations:
    storageclass.kubernetes.io/is-default-class: "true"
provisioner: csi-driver.example-vendor.example
allowVolumeExpansion: true
volumeBindingMode: WaitForFirstConsumer
```

Volume binding mode

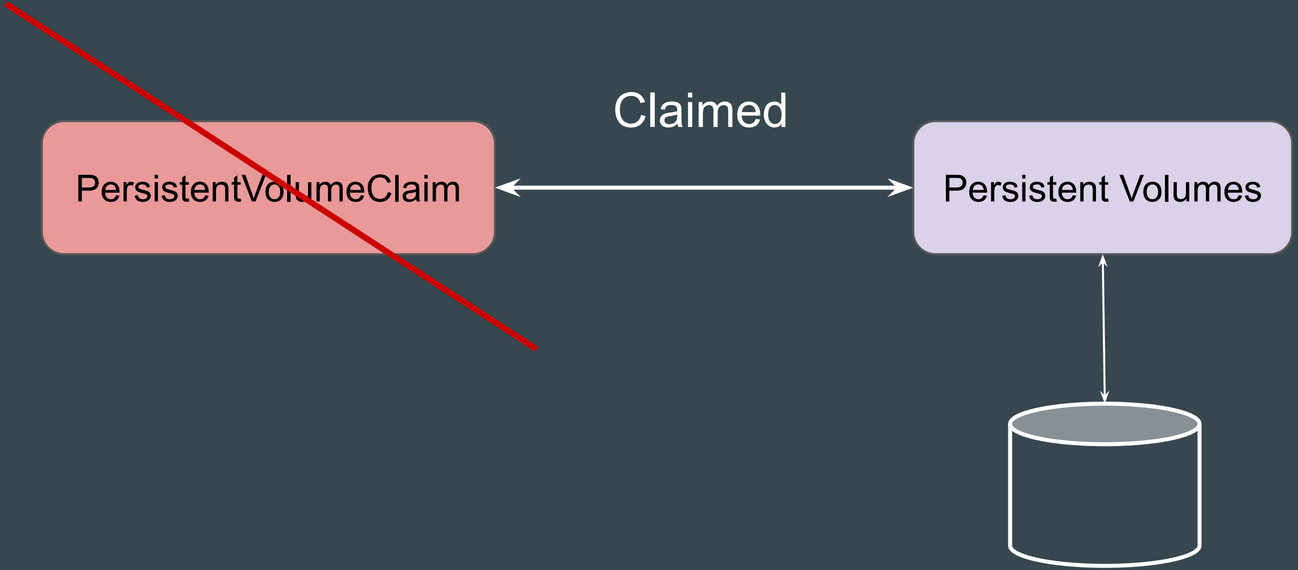
VolumeBindingMode determines when volume binding and dynamic provisioning occurs.

Binding Mode	Bind Time	Description
Immediate	At PVC creation	Volumes are provisioned and bound as soon as the PVC is created.
WaitForFirstConsumer	Volume binding and provisioning is delayed until a Pod using the PVC is scheduled.	At Pod scheduling

Persistent Volumes - Reclaim Policy

Setting the Base

The ReclaimPolicy specifies what happens to a Persistent Volume (PV) and its underlying storage resource after the Persistent Volume Claim (PVC) that was bound to it is deleted.



Types of Reclaim Policy

Access Mode	Description
Delete	When the PVC is deleted, both the PV object in Kubernetes and the associated underlying storage volume in the infrastructure are deleted.
Retain	When the PVC is deleted, the PV object remains in Kubernetes with its status marked as Released. Crucially, the underlying storage volume and its data are preserved.

Setting Reclaim Policy

Reclaim policy can be set both at a persistent volume level as well as Storage class level.

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv0003
spec:
  capacity:
    storage: 5Gi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Recycle
  nfs:
    path: /tmp
    server: 172.17.0.2
```

```yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: standard
provisioner: kubernetes.io/aws-ebs
reclaimPolicy: Retain
```

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv0003
spec:
  capacity:
    storage: 5Gi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Recycle
  nfs:
    path: /tmp
    server: 172.17.0.2
```

Use this when you're statically provisioning a volume (i.e., creating the PV manually).

```yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: standard
provisioner: kubernetes.io/aws-ebs
reclaimPolicy: Retain
```

Use this when you're using dynamic provisioning, where PVCs automatically create PVs through the StorageClass.

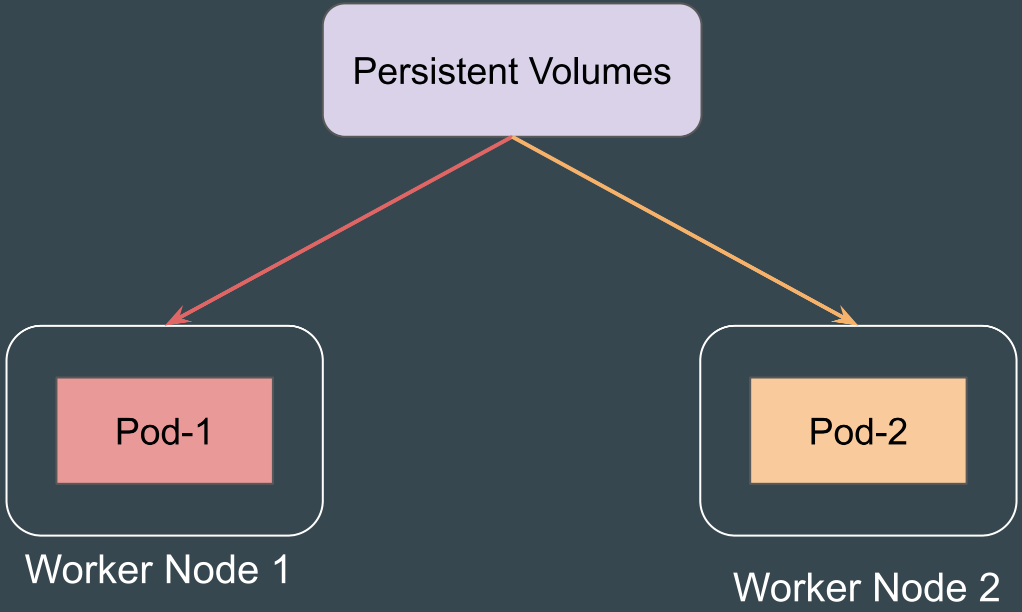
Summary Table

Feature	Retain	Delete
Data Retention After PVC Deletion	✅ Yes — data is preserved	❌ No — underlying storage is deleted
Underlying Storage Reused Automatically	❌ No — requires manual intervention	✅ Yes — storage is deleted and cleaned up
Manual Admin Cleanup Needed	✅ Yes — admin must delete or reuse volume	❌ No — handled automatically
Use Case	Critical data, backups, manual recovery	Temporary or dynamically provisioned data
Common Storage Backend Support	All backends	Dynamic provisioners (e.g., AWS EBS, GCE PD)
Recommended for	Long-term storage, compliance, audits	Short-lived workloads, automation

Persistent Volumes - Access Modes

Setting the Base

Access Modes define how a Persistent Volume can be mounted and accessed by Nodes and Pods in the cluster.



Access Mode	Abbreviation	Description
ReadWriteOnce	RWO	Volume can be mounted as read-write by Pods in a single node.
ReadOnlyMany	ROX	Volume can be mounted as read-only by many nodes.
ReadWriteMany	RWX	Volume can be mounted as read-write by many nodes.
ReadWriteOncePod	RWOP	Volume can be mounted as read-write by a single Pod.

1 - ReadWriteOnce (RWO)

The volume can be mounted as read-write by a single Node at a time.

Even if multiple Pods on the same Node use the same PVC requesting RWO, they can all potentially read from and write to the volume.

However, Pods on different Nodes cannot simultaneously mount this volume as read-write.

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: my-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 5Gi
  storageClassName: standard
```

2 - ReadOnlyMany (ROX)

The volume can be mounted as read-only by many Nodes simultaneously.

Multiple Pods across multiple Nodes can mount and read from this volume concurrently. No Pod can write to it.

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: config-data
spec:
  accessModes:
    - ReadOnlyMany
  resources:
    requests:
      storage: 5Gi
  storageClassName: nfs-storage
```

3 - ReadWriteMany (RWX)

The volume can be mounted as read-write by many Nodes simultaneously.

Multiple Pods running on different Nodes can all read from and write to the same volume concurrently. This requires a storage backend that supports shared access.

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: nfs-pvc
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 5Gi
  storageClassName: nfs-storage
```

4 - ReadWriteOncePod (RWOP)

The volume can be mounted as read-write by a single Pod only.

This is the most restrictive mode. Only one specific Pod in the entire cluster can mount this volume as read-write.

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: secure-pvc
spec:
  accessModes:
    - ReadWriteOncePod
  resources:
    requests:
      storage: 5Gi
  storageClassName: csi-storage
```

Important Considerations

Not all the volume types support all the access modes.

Always check the cloud provider or volume plugin documentation to see which access modes are supported.

Volume Plugin	ReadWriteOnce	ReadOnlyMany	ReadWriteMany	ReadWriteOncePod
AzureFile	✓	✓	✓	-
CephFS	✓	✓	✓	-
CSI	depends on the driver	depends on the driver	depends on the driver	depends on the driver
FC	✓	✓	-	-
FlexVolume	✓	✓	depends on the driver	-
HostPath	✓	-	-	-
iSCSI	✓	✓	-	-
NFS	✓	✓	✓	-

Summary Table

Access Mode	Abbreviation	Mounted By	Read	Write	Use Case Example
ReadWriteOnce	RWO	Single Node	✅	✅	AWS EBS, GCP PD, Azure Disk — standard volume for stateful apps
ReadWriteOncePod	RWOP	Single Pod (1 Node)	✅	✅	Security-sensitive apps needing strict single-pod access
ReadOnlyMany	ROX	Multiple Nodes	✅	❌	Shared read-only config or datasets
ReadWriteMany	RWX	Multiple Nodes	✅	✅	NFS, CephFS — shared writable volume across pods/nodes

ConfigMaps

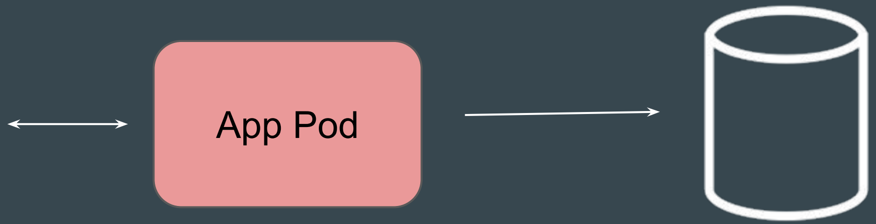
Understanding the Challenge - Part 1

Whenever an App pod gets deployed, it might need to connect to an external database to store data.

Issue: In many cases, these details are hard coded as part of the container image.

Key	Value
db_user	dbadmin
db_pass	A123#
db_url	172.31.10.30:3306

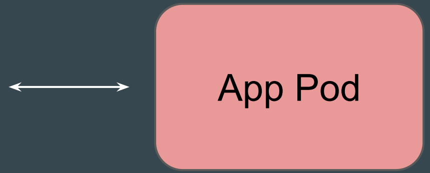
Hardcoded Data



Understanding the Challenge - Part 2

If a container has hardcoded data, any change to the data requires you to create a new set of Docker images and recreate the Pod

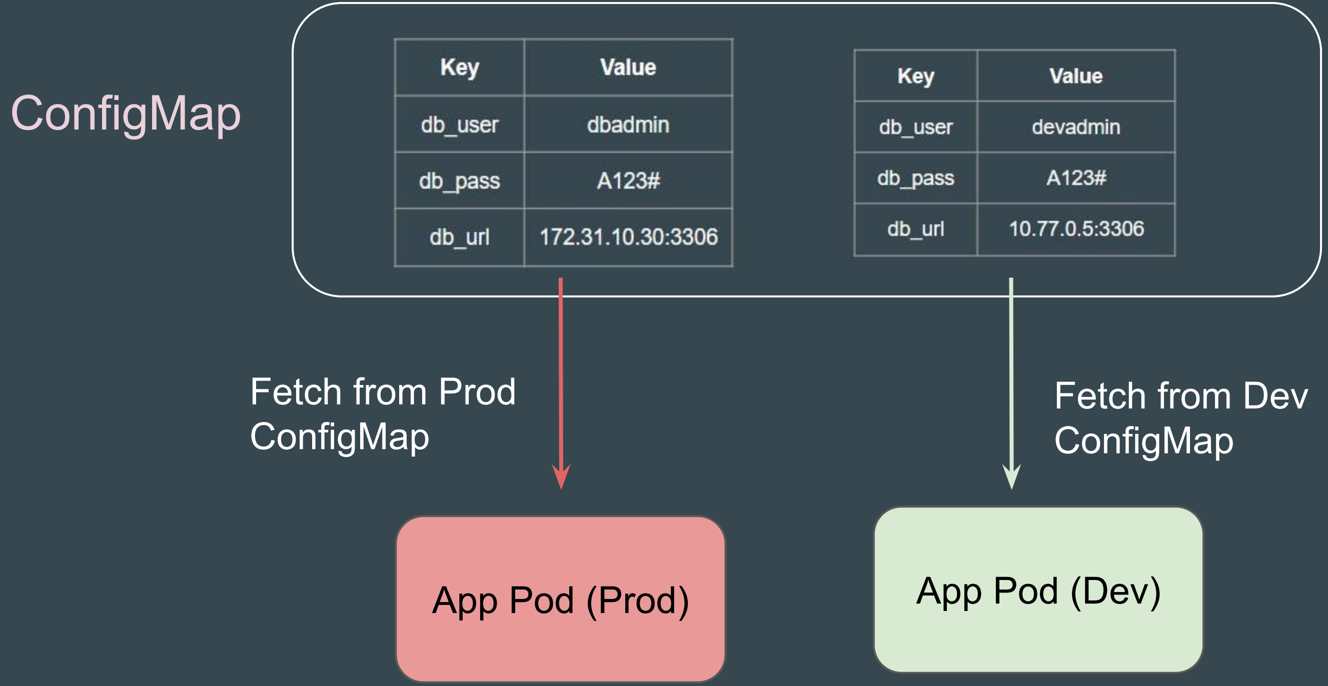
Key	Value
db_user	dbadmin
db_pass	A123#
db_url	172.31.10.30:3306



Hardcoded Data

Introduction to the ConfigMaps

A ConfigMap in Kubernetes is used to store key-value pairs of configuration data.



Key	Value
db_user	dbadmin
db_pass	A123#
db_url	172.31.10.30:3306

Key	Value
db_user	devadmin
db_pass	A123#
db_url	10.77.0.5:3306

Point to Note

The primary purpose of ConfigMap is to store non-sensitive configuration data, such as configuration files, environment variables, etc.

It stores data in plain-text and is not intended for sensitive information.

Practical Approach for Entire Workflow

Part 1: Create ConfigMap

Part 2: Configure Pod to use that appropriate ConfigMap

ConfigMap Practical - Part 1 (Create ConfigMap)

Command to Create ConfigMap

The kubectl create configmap command allows us to create ConfigMap from a file, directory, or specified literal value

```
Examples:
  # Create a new config map named my-config based on folder bar
  kubectl create configmap my-config --from-file=path/to/bar

  # Create a new config map named my-config with specified keys instead of file basenames on disk
  kubectl create configmap my-config --from-file=key1=/path/to/bar/file1.txt --from-file=key2=/path/to/bar/file2.txt

  # Create a new config map named my-config with key1=config1 and key2=config2
  kubectl create configmap my-config --from-literal=key1=config1 --from-literal=key2=config2

  # Create a new config map named my-config from the key=value pairs in the file
  kubectl create configmap my-config --from-file=path/to/bar

  # Create a new config map named my-config from an env file
  kubectl create configmap my-config --from-env-file=path/to/foo.env --from-env-file=path/to/bar.env
```

Approach 1 - From a Literal

The --from-literal option allows you to directly specify key-value pairs as command-line arguments.

```
C:\>kubectl create configmap dev-config --from-literal=db_user=dbadmin --from-literal=db_host=172.31.0.5
configmap/dev-config created
```

Approach 2 - From a File

The --from-file option allows you to create a ConfigMap from the contents of one or more files.

large-file.txt

```
kubectl create configmap --from-file=large-file.txt
```

Approach 3 - From a Directory

You can also use the --from-file option with a directory instead of a single file.

big-folder

kubectl create configmap demo --from-file=<path/to/directory>

Comparison of All the Methods

Method	Use-Case	Advantage	Disadvantage
--from-literal	Simple, one-off key-value pairs	Quick and easy for simple configurations	Not suitable for complex configurations or large amounts of data
--from-file (file)	Individual configuration files	Good for managing separate configuration files	Can become cumbersome for many files
--from-file (dir)	Multiple configuration files organized in a directory	Convenient for grouping related configuration files	Less control over individual key names (filenames are used as keys)

Type of Data In ConfigMap

In the ConfigMap manifest file, you can represent data in multiple distinct formats.

Simple Key Value pair

Multiline Block literal

! configmap.yaml

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: demo-configmap
data:
  key1: "value1"
  key2: "value2"

  essay: |
    This is the first line of the essay1.
    It spans multiple lines and contains various information.
    This is Line 3

  json_data: |
     {
       "name": "Alice",
       "age": 26,
       "skills": ["Kubernetes", "Docker", "DevOps"]
     }
```

Point to Note - Directory Approach

If you are referencing to an entire directory, Kubernetes will create a ConfigMap with each file in that directory becoming a key-value pair.

The filename (without extension) becomes the key, and the file content becomes the value

ConfigMap Practical - Part 2 (Mounting to Pods)

Setting the Base

Once ConfigMaps is created, we also need to reference it to appropriate Pod.

ConfigMap

Key	Value
db_user	dbadmin
db_pass	A123#
db_url	172.31.10.30:3306

Key	Value
db_user	devadmin
db_pass	A123#
db_url	10.77.0.5:3306

Mount from Prod ConfigMap

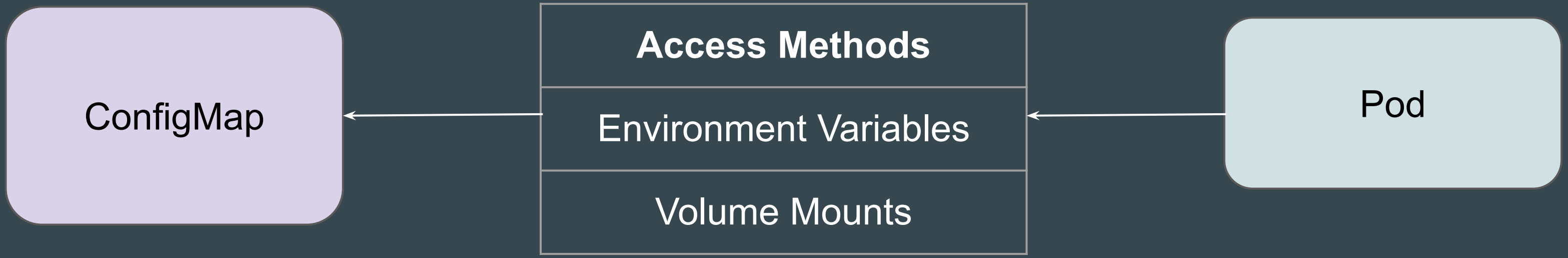
Mount from Dev ConfigMap

App Pod (Prod)

App Pod (Dev)

Different Approaches for Reference

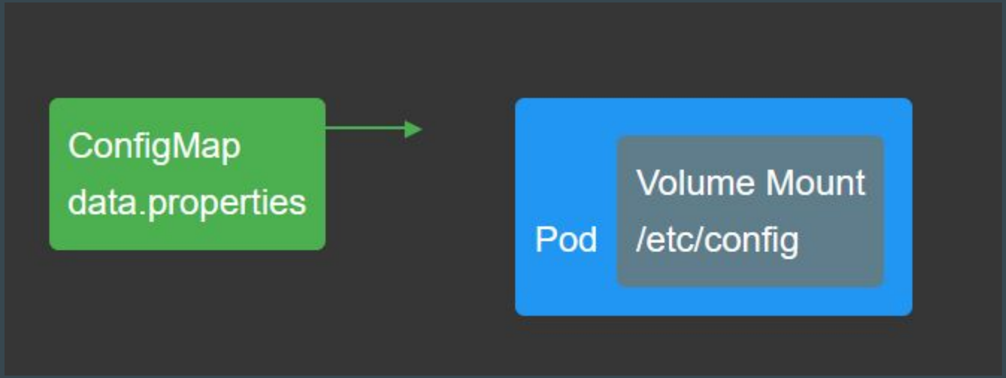
There are multiple ways through which a Pod can fetch the data of a ConfigMap.



Approach 1 - Volume Mount

In this method, the ConfigMap is mounted directly as a volume in the Pod.

Each key-value pair in the ConfigMap appears as a file in the volume.



Reference Manifest File

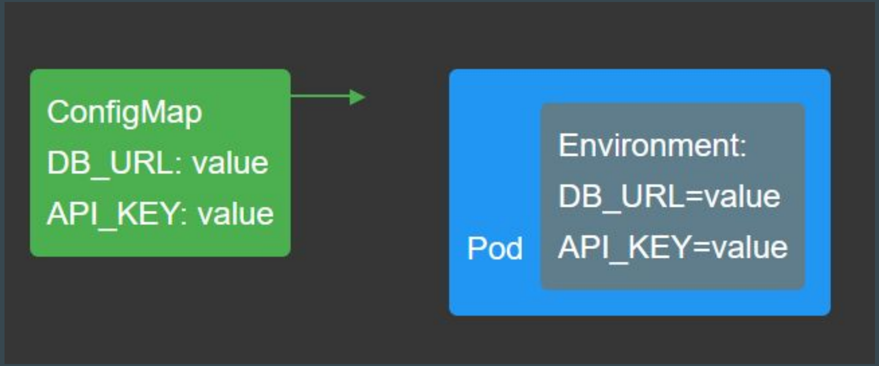
! pod-volume.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: app-pod
spec:
  containers:
    - name: app-container
      image: nginx
      volumeMounts:
        - name: config-volume
          mountPath: /etc/config
  volumes:
    - name: config-volume
      configMap:
        name: app-config
```

Approach 2 - Environment Variables

In this method, the values in the ConfigMap are exposed as environment variables to the container.

The application can then access these values using standard environment variable lookup.



Reference Screenshot

! pod-env.yaml

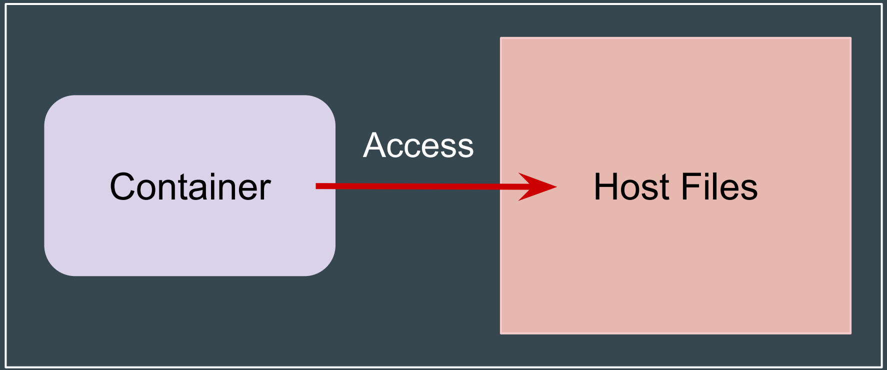
```yaml
apiVersion: v1
kind: Pod
metadata:
  name: app-pod
spec:
  containers:
    - name: app-container
      image: nginx
      env:
        - name: APP_MODE
          valueFrom:
            configMapKeyRef:
              name: app-config
              key: APP_MODE
        - name: APP_ENV
          valueFrom:
            configMapKeyRef:
              name: app-config
              key: APP_ENV
```

Security Context

Understanding the Challenge

Many times, the containers run with root user privileges.

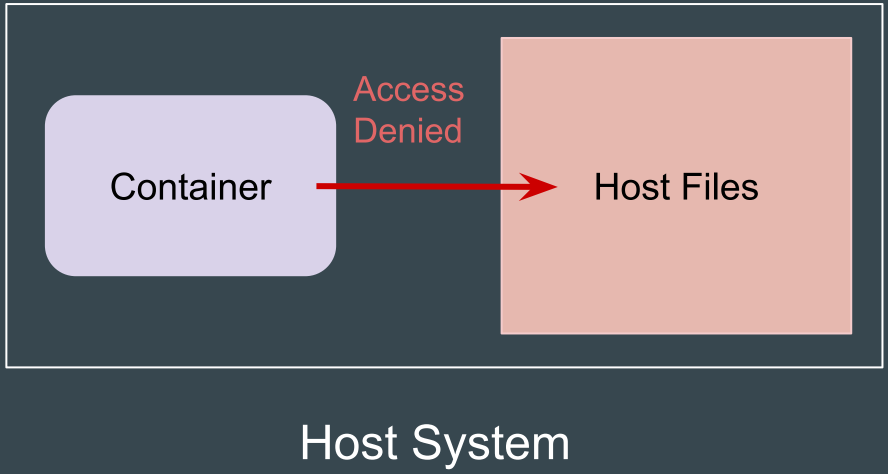
In case of container breakouts, an attacker can get full access to the host system.



Host System

Running Container with Non Root User

If the container runs with non-root privileges, it will be unable to modify the critical host files and will have limited access to the host system.



Introduction to Security Context

A security context defines privilege and access control settings for a Pod or Container.

Run as non-privileged user

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: better-pod
spec:
  securityContext:
    runAsUser: 1000
    runAsGroup: 1000
  containers:
  - name: better-container
    image: busybox
    command: ["sleep", "36000"]
```

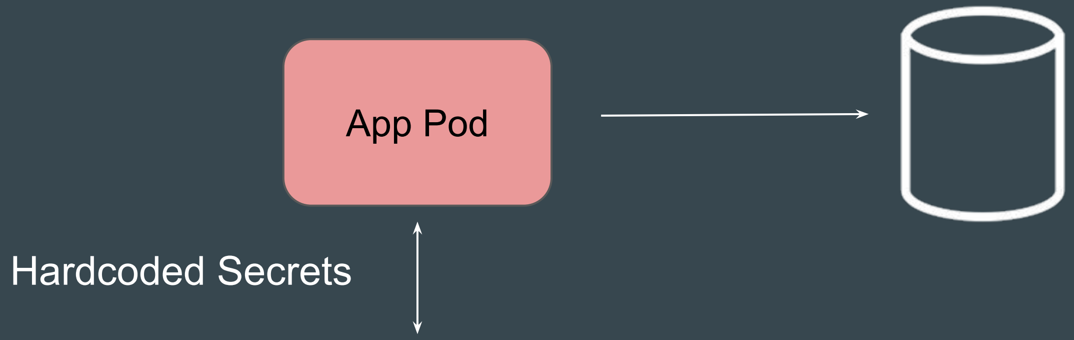
Comparison Table

Field	Description	Use-Case
runAsUser	Specifies the user ID (UID) a container's process runs as.	Use when you want the container to run as a specific user rather than the default (commonly root).
runAsGroup	Specifies the primary group ID (GID) a container's process runs as.	Use when you want the container's primary group to be a specific GID.
fsGroup	Specifies a group ID (GID) for volume-mounted files. Files created in mounted volumes will be owned by this GID.	Use when you need to control file permissions for a shared volume (e.g., for multiple containers in a Pod).

Kubernetes Secrets

HardCoding Secrets Should be Avoided

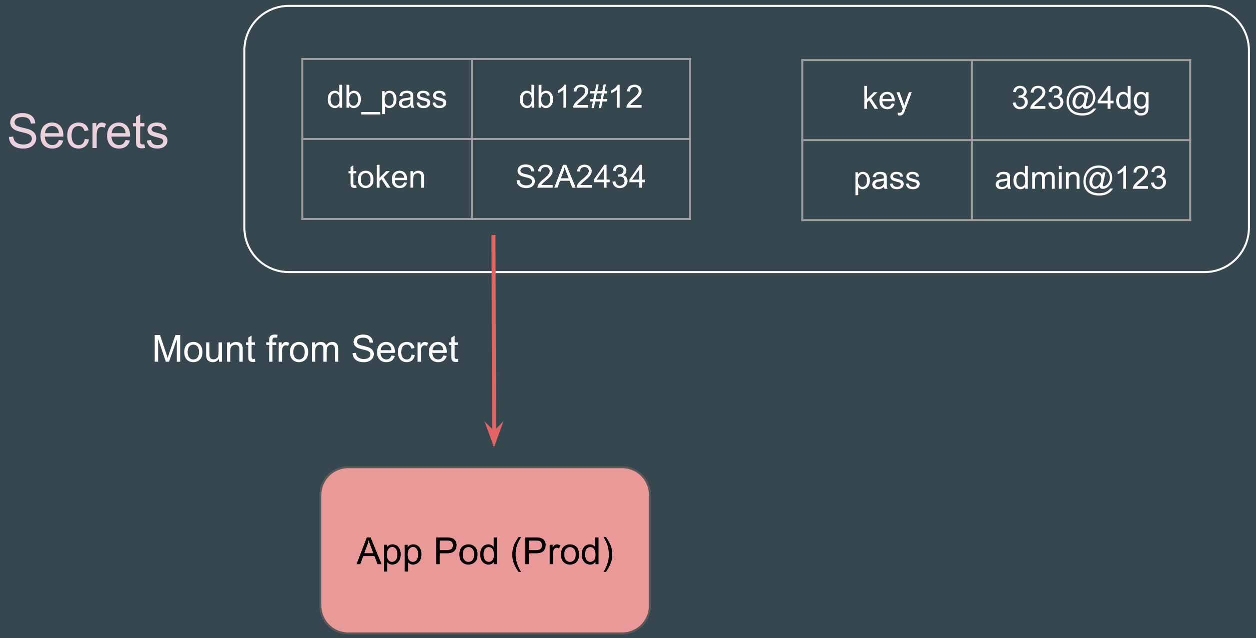
It is frequently observed that sensitive data like passwords, tokens, etc., are hard-coded as part of the container image.



Key	Value
db_user	dbadmin
db_pass	A1S323#

Introducing Secret

Kubernetes Secrets is a feature that allows us to store these sensitive data.



Reference Screenshot

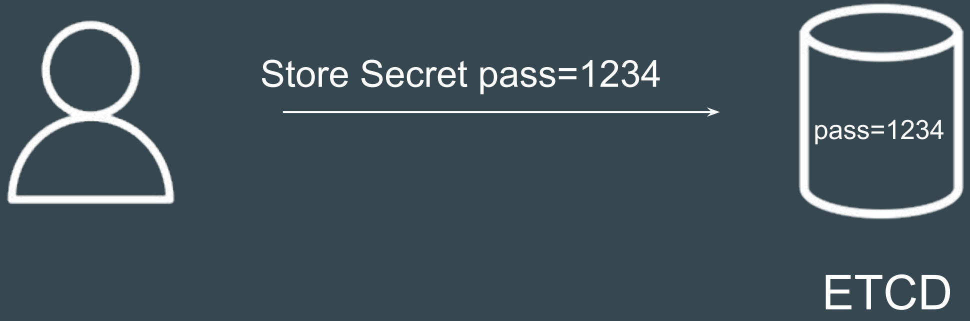
```
C:\>kubectl get secret
NAME        TYPE     DATA   AGE
my-secret   Opaque   1      22m
```

```
C:\>kubectl get secret my-secret -o yaml
apiVersion: v1
data:
  db_pass: QTIjMTI1U0A=
kind: Secret
metadata:
  creationTimestamp: "2025-01-16T02:34:21Z"
  name: my-secret
  namespace: default
  resourceVersion: "5877928"
  uid: d3c383cb-c2e3-4f9b-95ef-d1f0ec6a04be
type: Opaque
```

Point to Note - Part 1

By default, Secrets are not very secure as they are not stored in encrypted format in the data store (ETCD). You can setup this configuration manually.

You can also additionally protect access to secrets using RBAC for access control.



Point to Note - Part 2

When you view a secret, Kubectl will print the Secret in base64 encoded format.

You'll have to use an external base64 decoder to decode the Secret fully

```
C:\>kubectl get secret my-secret -o yaml
apiVersion: v1
data:
  db_pass: QTIjMTI1U0A=
kind: Secret
metadata:
  creationTimestamp: "2025-01-16T02:34:21Z"
  name: my-secret
  namespace: default
  resourceVersion: "5877928"
  uid: d3c383cb-c2e3-4f9b-95ef-d1f0ec6a04be
type: Opaque
```

base64 encoded