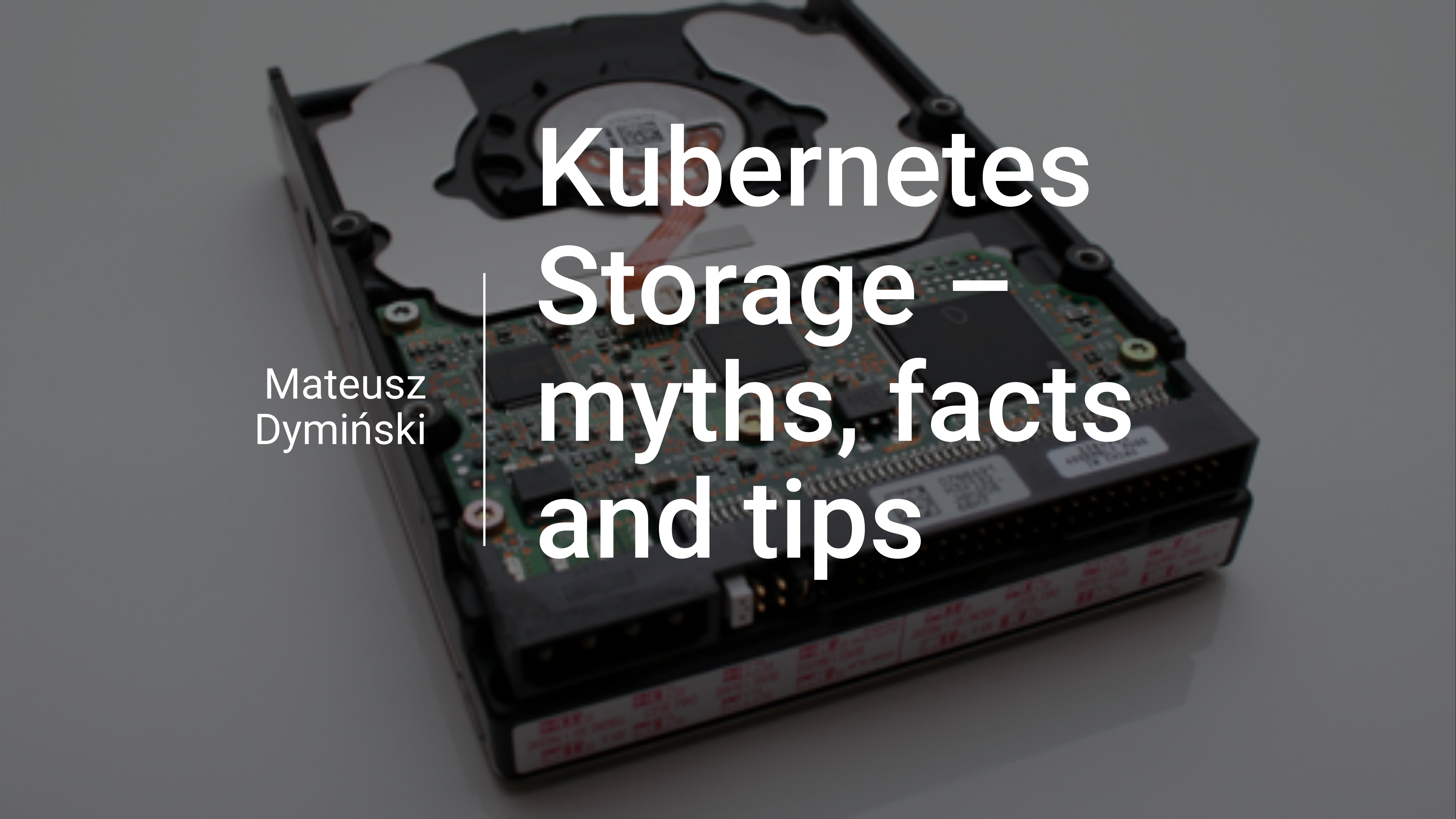


Kubernetes Storage – myths, facts and tips

Mateusz Dymiński

whoami

Mateusz Dymiński

- Software Developer at Nokia
- 5+ exp with Java
- 5+ exp with Go
- One of the organizer GoWroc - Golang Wroclaw Meetup
- Page: https://mateuszdyminski.com
- Github: github.com/mateuszdyminski
- Twitter: @m_dyminski
- LinkedIn: linkedin.com/in/mdyminski

Agenda

- Challenges in Stateful Containers
- Volume Plugins
- Stateful Applications
- Storage Features
- MythBusters
- Summary

github.com/mateuszdyminski/storage

Database in Pod

```yaml
apiVersion: apps/v1
kind: Pod
metadata:
  name: mysql
spec:
  containers:
  - image: mysql:5.6
    name: mysql
    env:
    - name: MYSQL_ROOT_PASSWORD
      value: password
    ports:
    - containerPort: 3306
      name: mysql
```

What happen with data when container ends?

All the data is lost

Problems with stateful containers

- Containers are ephemeral by design
- Container termination/crashes result in loss of data
- Containers can’t share data between each other.
- Can’t run stateful applications
- Unless the storage is provisioned and attached to container by Volume Plugin

Volume Plugin

- A way to reference block device or mounted filesystem
- Ordered Volume is accessible by all containers in pod
- Volume plugin specifies
  - How volume is setup in pod
  - Where the data resides
- Lifetime of volume could be longer than lifetime of pod

Kubernetes Storage - options

- File Storage
  - NFS, SMB, etc.
- Block Storage
  - GCE PD, AWS EBS, iSCSI, Fibre Channel, etc.
- File on Block Storage

Data Path Standarized – Posix, SCSI

Kubernetes Volumes Plugins

Remote Storage

- GCEPersistentDisk
- AWSElasticBlockStore
- AzureFile
- AzureDisk
- CSI
- FC (Fibre Channel)
- FlexVolume
- Flocker
- NFS
- iSCSI
- RBD (Ceph Block Device)
- CephFS
- Cinder (OpenStack block storage)
- Glusterfs
- VsphereVolume
- Quobyte Volumes

Ephemeral Storage

- EmptyDir
- Expose Kubernetes API
  - Secret
  - ConfigMap
  - DownwardAPI

Local Persistent Volume

Out-of-Tree

- FlexVolume (exec a binary)
- CSI

Host path

Ephemeral Storage

Ephemeral Storage

- File space from host
- Temporary!
- Data exists only for lifecycle of pod.
- Can only be referenced “in-line” in pod definition not via PV/PVC.
- Volume Plugin: EmptyDir

Ephemeral Storage - EmptyDir

```yaml
apiVersion: apps/v1
kind: Pod
metadata:
  name: mysql
spec:
  containers:
  - image: mysql:5.6
    name: mysql
    env:
    - name: MYSQL_ROOT_PASSWORD
      value: password
    ports:
    - containerPort: 3306
      name: mysql
    volumeMounts:
    - name: data
      mountPath: /var/lib/mysql
  volumes:
  - name: data
    emptyDir: {}
```

Ephemeral Storage

- Built on top of EmptyDir:
  - Secret Volume
  - ConfigMap Volume
  - DownwardAPI Volume
- Populate Kubernetes API as files in to an EmptyDir

Ephemeral Storage – ConfigMap

```yaml
apiVersion: apps/v1
kind: Pod
metadata:
  name: mysql
spec:
  containers:
  - image: mysql:5.6
    name: mysql
    env:
    - name: MYSQL_ROOT_PASSWORD
      value: password
    ports:
    - containerPort: 3306
      name: mysql
    volumeMounts:
    - name: config-map
      mountPath: /etc/mysql.conf
  volumes:
  - name: config-map
    configMap:
      name: mysql
```

Remote Storage

Remote Storage – EBS example

```yaml
apiVersion: apps/v1
kind: Pod
metadata:
  name: mysql
spec:
  containers:
  - image: mysql:5.6
    name: mysql
    env:
    - name: MYSQL_ROOT_PASSWORD
      value: password
    volumeMounts:
    - mountPath: /var/lib/mysql
      name: ebs-volume
  volumes:
  - name: ebs-volume
    awsElasticBlockStore:
      volumeID: <volume-id>
      fsType: ext4
```

There are some restrictions when using an awsElasticBlockStore volume:

- the nodes on which Pods are running must be AWS EC2 instances
- those instances need to be in the same region and availability-zone as the EBS volume
- EBS only supports a single EC2 instance mounting a volume

Pod yaml is no longer portable across clusters!!

DON'T DIRECTLY
REFERENCE A VOLUME
imgflip.com

Persistent Volume
Persistent Volume Claim

Persistent Volume
Persistent Volume Claim

Abstracts and Decouples storage implementation from storage consumption

Pod mounts PersistentVolumeClaim into container(s)

```
apiVersion: apps/v1
kind: Pod
metadata:
  name: mysql
spec:
  containers:
  - image: mysql:5.6
    name: mysql
    env:
    - name: MYSQL_ROOT_PASSWORD
      value: password
    volumeMounts:
    - name: mysql-persistent-storage
      mountPath: /var/lib/mysql
  volumes:
  - name: mysql-persistent-storage
    persistentVolumeClaim:
      claimName: mysql-pv-claim
```

Pod yaml is portable again!!

PersistentVolumeClaim = request for storage

```yaml
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: mysql-pv-claim
spec:
  resources:
    requests:
      storage: 1Gi
    accessModes:
    - ReadWriteOnce
```

- "Give me 1 GiB of storage."
- "That is mountable to single pod as read/write."
- "And I don't really care about the rest."

PersistentVolumeClaim (PVC)

- Application request for storage.
- Created by user / devops.
- Binds to single PV.
- Usable in Pods.

PersistentVolume

```yaml
kind: PersistentVolume
apiVersion: v1
metadata:
  name: task-pv-volume
labels:
  type: local
spec:
  capacity:
    storage: 10Gi
  accessModes:
  - ReadWriteMany
  - ReadWriteOnce
  - ReadOnlyMany
  persistentVolumeReclaimPolicy: Retain
  awsElasticBlockStore:
    fsType: "ext4"
    volumeID: "vol-f37a03aa"
```

PersistentVolume

```yaml
kind: PersistentVolume
apiVersion: v1
metadata:
  name: task-pv-volume
labels:
  type: local
spec:
  capacity:
    storage: 10Gi
  accessModes:
  - ReadWriteMany
  - ReadWriteOnce
  - ReadOnlyMany
  persistentVolumeReclaimPolicy: Retain
  awsElasticBlockStore:
    fsType: "ext4"
    volumeID: "vol-f37a03aa"
```

Size of the Volume

PersistentVolume

```
kind: PersistentVolume
apiVersion: v1
metadata:
  name: task-pv-volume
labels:
  type: local
spec:
  capacity:
    storage: 10Gi
  accessModes:
  - ReadWriteMany
  - ReadWriteOnce
  - ReadOnlyMany
  persistentVolumeReclaimPolicy: Retain
  awsElasticBlockStore:
    fsType: "ext4"
    volumeID: "vol-f37a03aa"
```

Access modes that the volume supports

PersistentVolume

```
kind: PersistentVolume
apiVersion: v1
metadata:
  name: task-pv-volume
labels:
  type: local
spec:
  capacity:
    storage: 10Gi
  accessModes:
  - ReadWriteMany
  - ReadWriteOnce
  - ReadOnlyMany
  persistentVolumeReclaimPolicy: Retain
  awsElasticBlockStore:
    fsType: "ext4"
    volumeID: "vol-f37a03aa"
```

What to do when the volume is not needed any longer. Options:

- Recycle (deprecated),
- Retain,
- Delete

PersistentVolume

```
kind: PersistentVolume
apiVersion: v1
metadata:
  name: task-pv-volume
labels:
  type: local
spec:
  storageClassName: cheap
  capacity:
    storage: 10Gi
  accessModes:
  - ReadWriteMany
  - ReadWriteOnce
  - ReadOnlyMany
  persistentVolumeReclaimPolicy: Retain
  awsElasticBlockStore:
    fsType: "ext4"
    volumeID: "vol-f37a03aa"
```

Pointer to Storage

AWS EBS, Azure DD, Ceph FS & RBD, CSI, FC, Flex, GCE PD, Gluster, iSCSI, NFS, OpenStack Cinder, Photon, Quobyte, StorageOS, vSphere

PV and PVC step by step

```
$ kubectl create -f pv.yaml
persistentvolum/task-pv-volume created
```

PV and PVC step by step

```
$ kubectl create -f pv.yaml
persistentvolum/task-pv-volume created
```

```
$ kubectl get pv
NAME             CAPACITY    ACCESSMODES    STATUS       CLAIM            REASON      AGE
task-pv-volume   10Gi        RWO            Available                                 1m
```

PV and PVC step by step

```
$ kubectl create -f pv.yaml
persistentvolum/task-pv-volume created

$ kubectl get pv
NAME              CAPACITY     ACCESSMODES     STATUS      CLAIM           REASON      AGE
task-pv-volume    10Gi         RWO             Available                               1m
```

```
$ kubectl create -f pvc.yaml
persistentvolumeclaim/mysql-pv-claim created
```

PV and PVC step by step

```
$ kubectl create -f pv.yaml
persistentvolum/task-pv-volume created

$ kubectl get pv
NAME            CAPACITY   ACCESSMODES   STATUS      CLAIM              REASON      AGE
task-pv-volume  10Gi       RWO           Available                                  1m

$ kubectl create -f pvc.yaml
persistentvolumeclaim/mysql-pv-claim created

$ kubectl get pvc
NAME            STATUS     VOLUME     CAPACITY   ACCESSMODES        STORAGECLASS   AGE
my-mysql-claim  Bound      pvc-6428   10Gi       RWO                standard       1m
```

PV and PVC step by step

```
$ kubectl create -f pv.yaml
persistentvolum/task-pv-volume created

$ kubectl get pv
NAME             CAPACITY    ACCESSMODES    STATUS       CLAIM                 REASON     AGE
task-pv-volume   10Gi        RWO            Available                                     1m

$ kubectl create -f pvc.yaml
persistentvolumeclaim/mysql-pv-claim created

$ kubectl get pvc
NAME             STATUS       VOLUME      CAPACITY    ACCESSMODES          STORAGECLASS    AGE
my-mysql-claim   Bound        pvc-6428    10Gi        RWO                  standard        1m

$ kubectl get pv
NAME             CAPACITY    ACCESSMODES    STATUS       CLAIM                 REASON     AGE
task-pv-volume   10Gi        RWO            Bound        mysql-pv-claim                   1m
```

Dynamic provisioning

- Cluster admin pre-provisioning PVs is painful and wasteful.
- Dynamic provisioning creates new volumes on-demand (when requested by user).
- Eliminates need for cluster administrators to pre-provision storage.

Dynamic provisioning

- Dynamic provisioning “enabled” by creating StorageClass.
- StorageClass defines the parameters used during creation.
- StorageClass parameters opaque to Kubernetes so storage providers can expose any number of custom parameters for the cluster admin to use.

StorageClass

```yaml
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: slow
provisioner: kubernetes.io/gce-pd
annotations:
  storageclass.kubernetes.io/is-default-class: "true"
parameters:
  type: pd-standard
  zone: us-east1-d
```

- Collection of PersistentVolumes with the same characteristics.
- Usually admin territory.
- Global, not namespaced.

StorageClass

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: slow
provisioner: kubernetes.io/gce-pd
annotations:
  storageclass.kubernetes.io/is-default-class: "true"
parameters:
  type: pd-standard
  zone: us-east1-d
```

- Who dynamically provisions volumes.
  - Name of hardcoded volume plugin.
  - Name of external provisioner.
  - Name of CSI driver.

StorageClass

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: slow
provisioner: kubernetes.io/gce-pd
annotations:
  storageclass.kubernetes.io/is-default-class: "true"
parameters:
  type: pd-standard
  zone: us-east1-d
```

- Parameters for dynamic provisioning.
  - Depend on the provisioner.

StorageClass

```yaml
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: slow
provisioner: kubernetes.io/gce-pd
annotations:
  storageclass.kubernetes.io/is-default-class: "true"
parameters:
  type: pd-standard
  zone: us-east1-d
```

- One StorageClass in the cluster can be default.
  - PVC without any StorageClass gets the default one.

Dynamic Provisioning Step by Step

Dynamic Provisioning – mysql yaml

```
apiVersion: apps/v1
kind: Pod
metadata:
  name: mysql
spec:
  containers:
  - image: mysql:5.6
    name: mysql
    env:
    - name: MYSQL_ROOT_PASSWORD
      value: password
    volumeMounts:
    - name: mysql-persistent-storage
      mountPath: /var/lib/mysql
  volumes:
  - name: mysql-persistent-storage
    persistentVolumeClaim:
      claimName: mysql-pv-claim
```

We don’t need to touch Pod Yaml

Dynamic Provisioning – PVC yaml

```yaml
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: mysql-pv-claim
spec:
  resources:
    requests:
      storage: 1Gi
  accessModes:
  - ReadWriteOnce
  storageClassName: slow
```

We need to add storageClassName

Dynamic Provisioning - step by step

```
$ kubectl create -f storage_class.yaml
storageclass „slow” created
```

Dynamic Provisioning - step by step

```
$ kubectl create -f storage_class.yaml
storageclass „slow” created
```

```
$ kubectl create -f pvc.yaml
persistentvolumeclaim/mysql-pv-claim created
```

Dynamic Provisioning - step by step

```
$ kubectl create -f storage_class.yaml
storageclass „slow” created

$ kubectl create -f pvc.yaml
persistentvolumeclaim/mysql-pv-claim created

$ kubectl get pvc
NAME            STATUS      VOLUME      CAPACITY    ACCESSMODES         STORAGECLASS     AGE
my-mysql-claim  Bound       pvc-6428    10Gi        RWO                 standard         1m
```

Dynamic Provisioning - step by step

```
$ kubectl create -f storage_class.yaml
storageclass „slow” created

$ kubectl create -f pvc.yaml
persistentvolumeclaim/mysql-pv-claim created

$ kubectl get pvc
NAME             STATUS     VOLUME     CAPACITY   ACCESSMODES     STORAGECLASS   AGE
my-mysql-claim   Bound      pvc-6428   10Gi       RWO             standard       1m
```

```
$ kubectl get pv
NAME             CAPACITY   ACCESSMODES   STATUS    CLAIM             REASON     AGE
task-pv-volume   10Gi       RWO           Bound     mysql-pv-claim               1m
```

PV and PVC release

```
$ kubectl delete pvc mysql-pv-claim
persistentvolumeclaim „mysql-pv-claim” deleted
```

PersistentVolume – Release

PVC is deleted: persistentVolumeReclaimPolicy is executed:

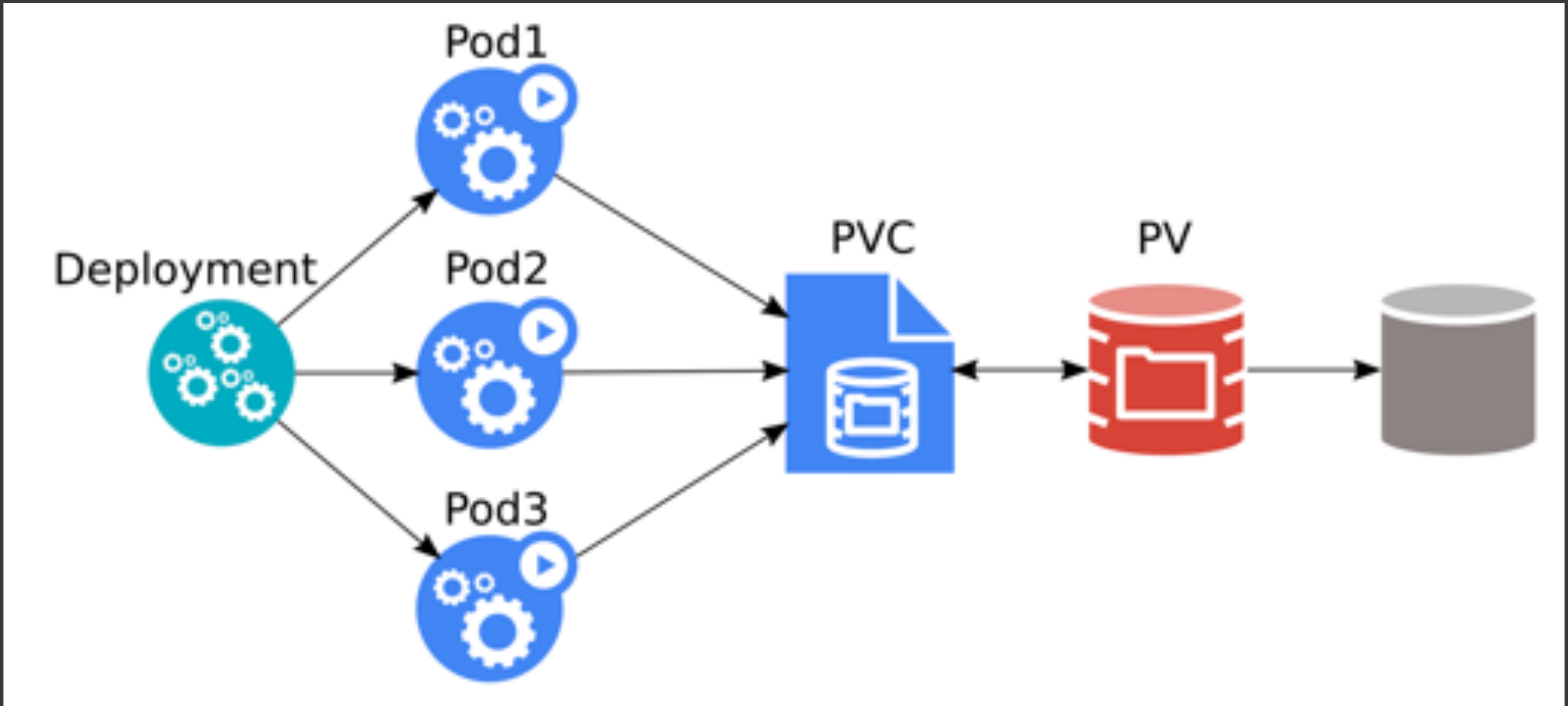
- Recycle (deprecated):
  - All data from the volume are removed ("rm -rf *").
  - PV is Available for new PVCs.
- Delete:
  - Volume is deleted in the storage backend.
  - PV is deleted.
  - Usually for dynamically-provisioned volumes
- Retain:
  - PV is kept Released.
  - No PVC can bind to it.
  - Admin should manually prune Released volumes. In all cases, user can't access the data!

Stateful applications

Deployment

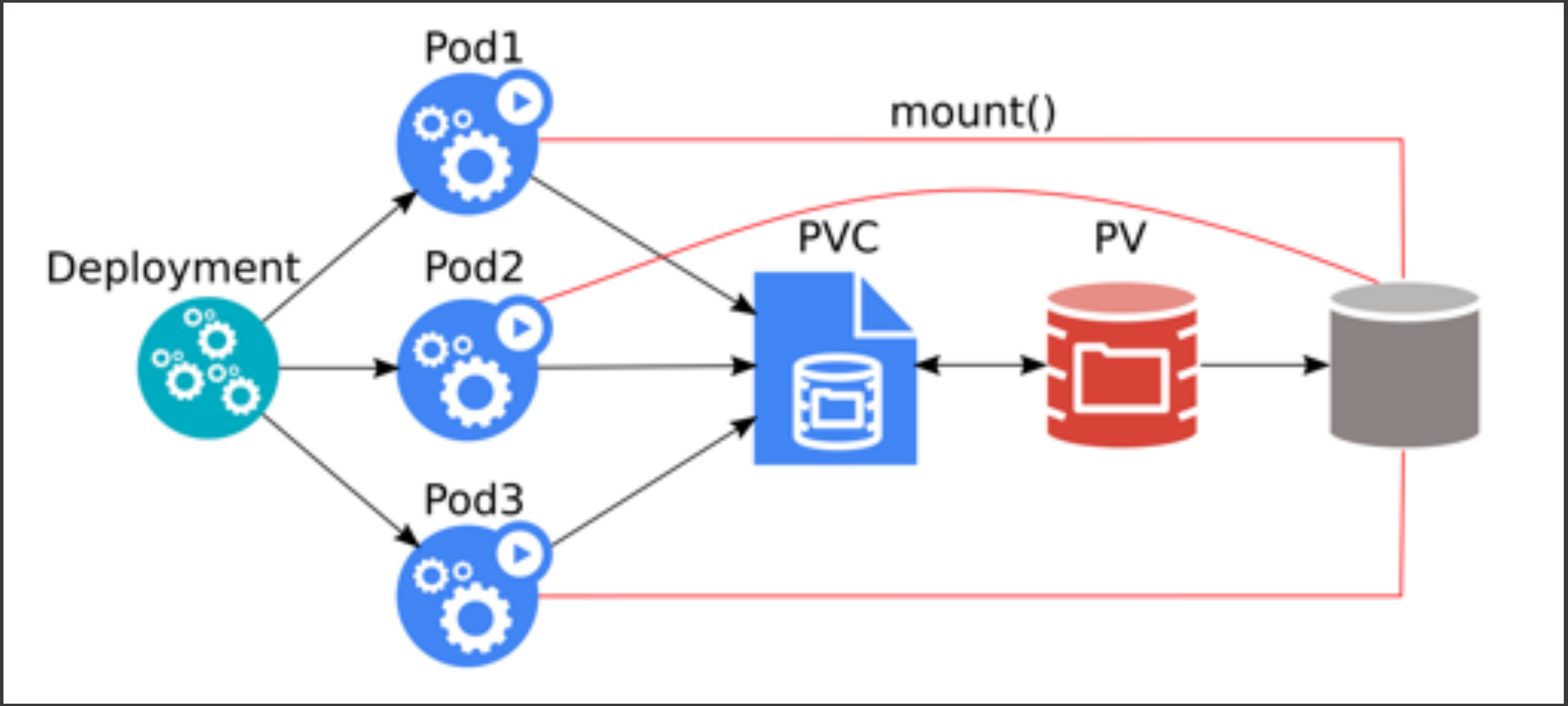
- Runs X replicas of a single Pod template.
- When a pod is deleted, Deployment automatically creates a new one.
- Scalable up & down.
- All pods share the same PVC!

Deployment



Source: Kubernetes Storage 101 - Jan Šafránek, Red Hat & David Zhu, Google

Deployment



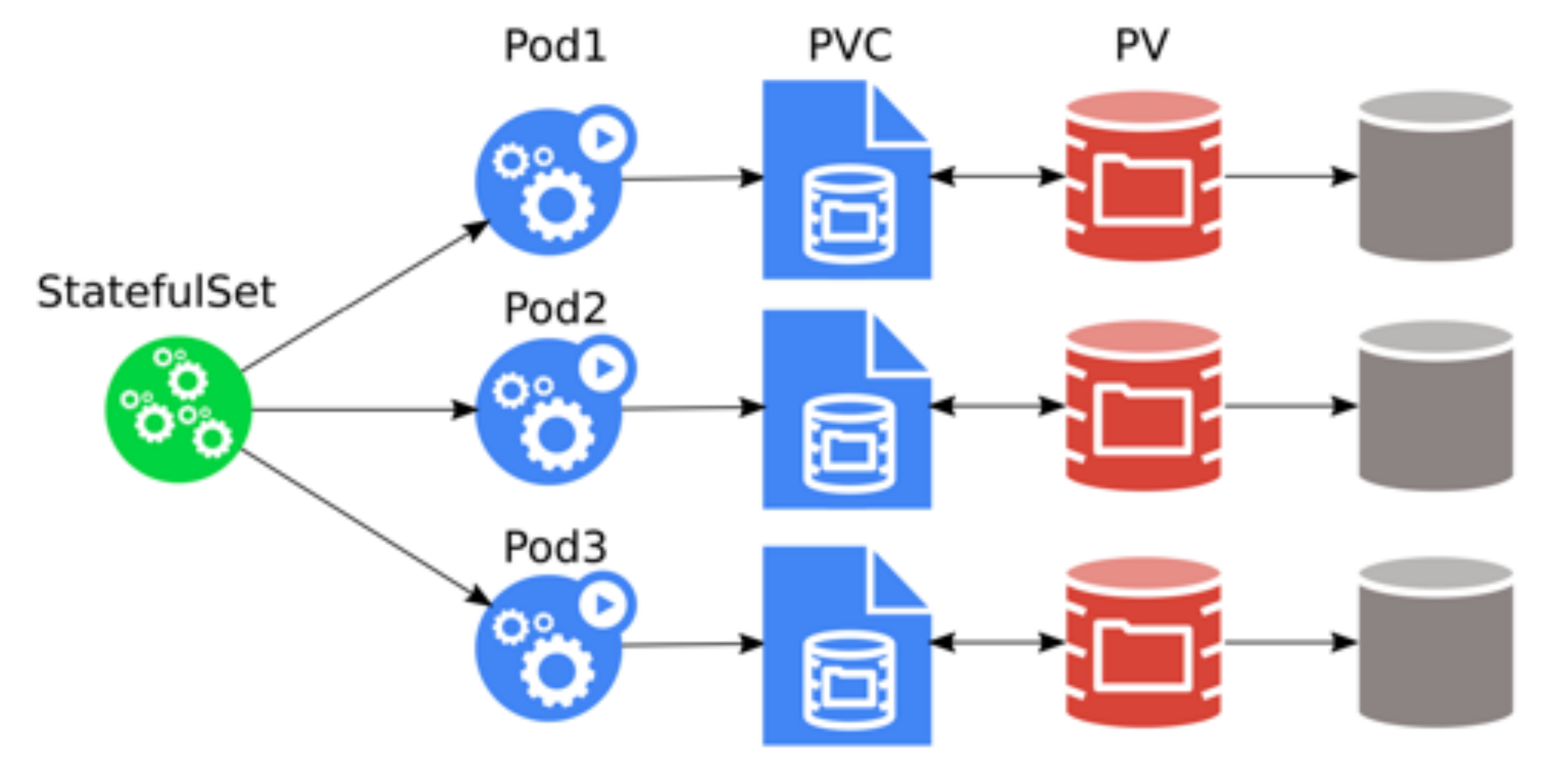
- All three pods can overwrite data of each other!
- Most applications crash / refuse to work.

Source: Kubernetes Storage 101 - Jan Šafránek, Red Hat & David Zhu, Google

StatefulSet

- Runs X replicas of a single Pod template.
  - Each pod gets its own PVC(s) from a PVC template.
- When a pod is deleted, StatefulSet automatically creates a new one.
- Each pod has a stable identity.
- Scalable up & down.

StatefulSet



The pods must be aware of the other StatefulSet members!

Source: Kubernetes Storage 101 - Jan Šafránek, Red Hat & David Zhu, Google

StatefulSet vs Deployment

```yaml
apiVersion: apps/v1
kind: Pod
metadata:
  name: mysql
spec:
  containers:
  - image: mysql:5.6
    name: mysql
    env:
    - name: MYSQL_ROOT_PASSWORD
      value: password
    volumeMounts:
    - name: mysql-persistent-storage
      mountPath: /var/lib/mysql
  volumes:
  - name: mysql-persistent-storage
    persistentVolumeClaim:
      claimName: mysql-pv-claim
```

```yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: mysql
spec:
  selector:
    matchLabels:
      app: mysql
  serviceName: "mysql"
  replicas: 3
  template:
    metadata:
      labels:
        app: mysql
    spec:
      containers:
      - name: mysql
        image: mysql:5.6
        volumeMounts:
        - name: mysql-persistent-storage
          mountPath: /var/lib/mysql
  volumeClaimTemplates:
  - metadata:
      name: mysql-persistent-storage
  spec:
    accessModes: [ "ReadWriteOnce" ]
    storageClassName: "slow"
    resources:
      requests:
        storage: 1Gi
```

Kubernetes Volumes Plugins

Remote Storage

- GCE Persistent Disk
- AWS Elastic Block Store
- Azure File Storage
- Azure Data Disk
- Dell EMC ScaleIO
- iSCSI
- Flocker
- NFS
- vSphere
- GlusterFS
- Ceph File and RBD
- Cinder
- Quobyte Volume
- FibreChannel
- VMware Photon PD

Ephemeral Storage

- EmptyDir
- Expose Kubernetes API
  - Secret
  - ConfigMap
  - DownwardAPI

Local Persistent Volume

Out-of-Tree

- Flex (exec a binary)
- CSI

Host path

Local Volumes

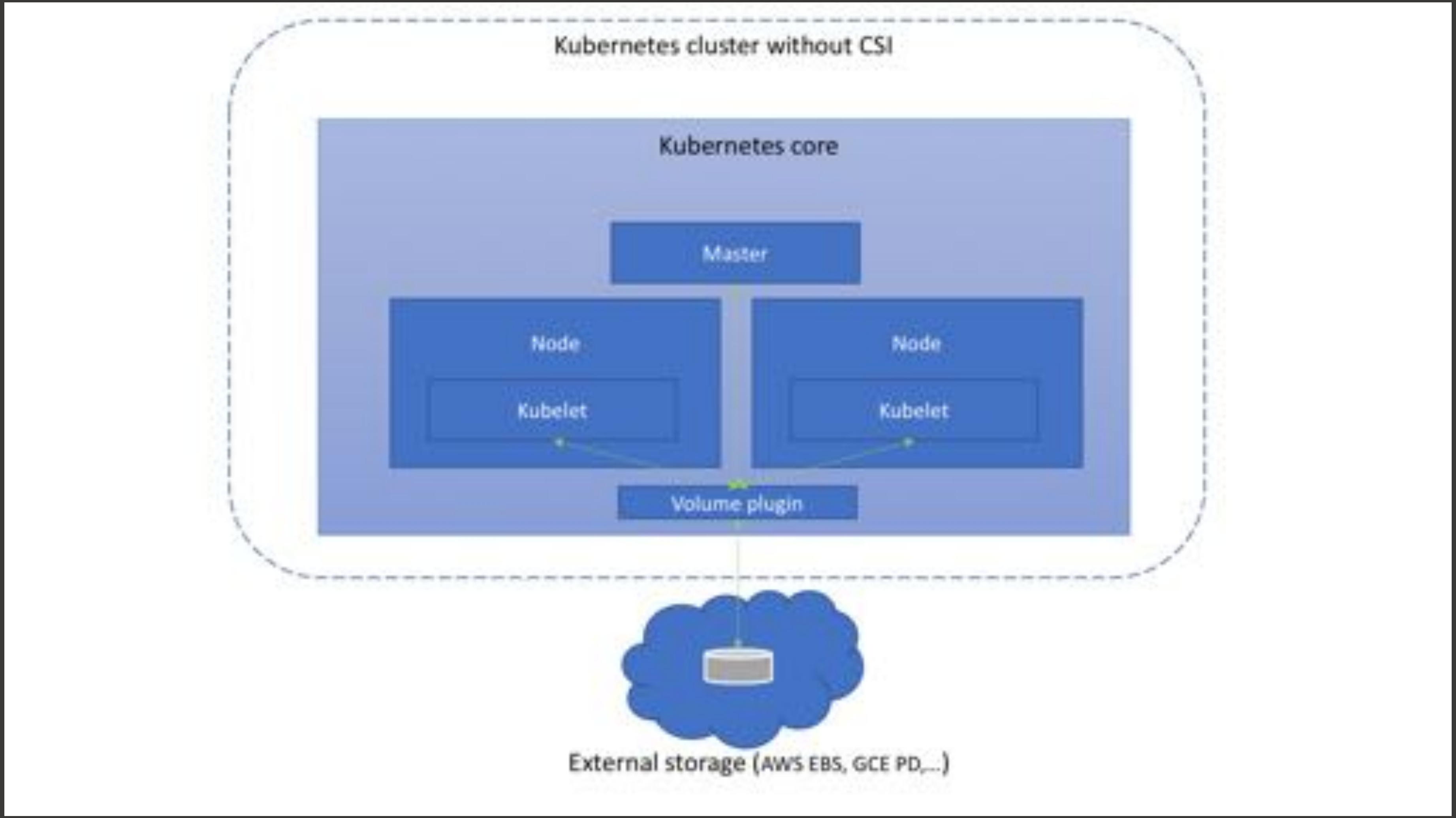
- Local disks can be used as PVs.
- Expose a local block or file as a PersistentVolume
- Reduced durability
- Extra speed
- Useful for building distributed storage systems
- Useful for high performance caching
- Kubernetes takes care of data gravity
- Referenced via PV/PVC so workload portability is maintained

CSI

CSI = Container Storage Interface

- Container Storage Interface (CSI) is an initiative to unify the storage interface of Container Orchestrator Systems (COs) like Kubernetes, Mesos, Docker swarm, cloud foundry, etc.
- Implementing a single CSI for a storage vendor is guaranteed to work with all COs.

Before CSI

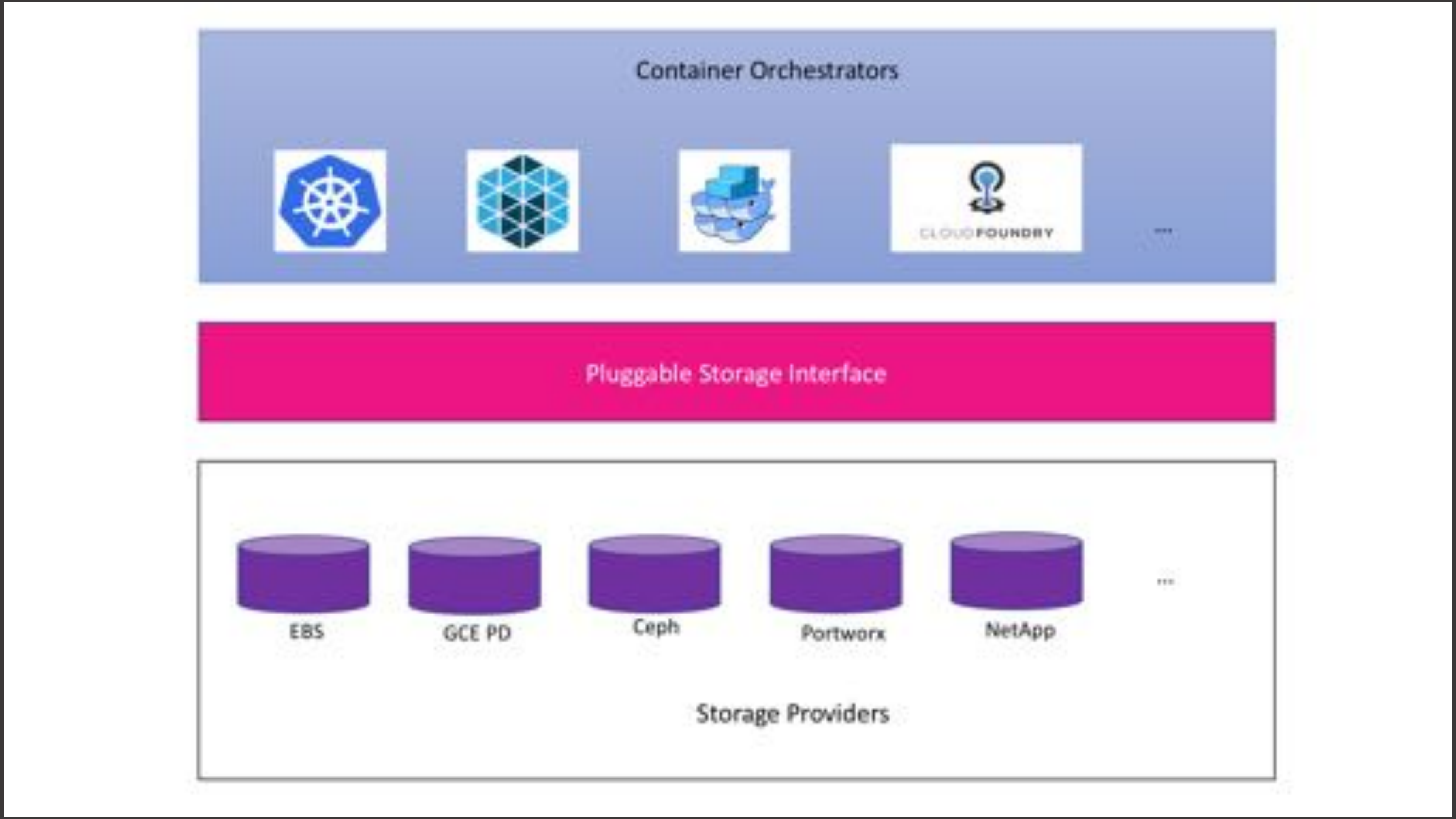

Kubernetes cluster without CSI
Kubernetes core
Master
Node
Kubelet
Node
Kubelet
Volume plugin
External storage (AWS EBS, GCE PD,...)

Source: https://medium.com/google-cloud/understanding-the-container-storage-interface-csi-ddbeb966a3b

Before CSI - problems

- Volume plugin development coupled and dependent on Kubernetes releases.
- Kubernetes developers/community are responsible for testing and maintaining all volume plugins.
- Bugs in volume plugins can crash Kubernetes core components
- Volume plugins get full privileges of kubernetes components (kubelet and kube-controller-manager).
- Plugin developers are forced to make plugin source code available – open-source

CSI concept



Source: CloudNativeCon EU 2018 CSI Jie Yu

CSI exmaple

```yaml
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: fast-storage
provisioner: csi-driver.example.com
parameters:
  type: pd-ssd
  csi.storage.k8s.io/provisioner-secret-name: mysecret
  csi.storage.k8s.io/provisioner-secret-namespace: mynamespace
```

CSI – materials

- Webpage - https://kubernetes-csi.github.io/
- Specification - https://github.com/container-storage-interface/spec
- Great talks:
  - Container Storage Interface: Present and Future - Jie Yu
  - Container Storage Interface for Kubernetes

Storage features

Resize of PV

- Only expansion is supported.
- Offline.
- Online (beta).

Snapshots

- Part of CSI.
- Can take a snapshot of PV.
- PV can be provisioned from a snapshot.
- VolumeSnapshotContent, VolumeSnapshot, VolumeSnapshotClass CRDs are intruduced
- More: https://kubernetes.io/blog/2018/10/09/introducing-volume-snapshot-alpha-for-kubernetes/

MYTHBUSTERS

Myth 1: Applications in containers must be stateless

- Unless persistent volume is used
- Statistic shows that 40% of workloads are stateful
- It’s myth

Myth 2: Writes done by container apps are slow

- hostPath and Local Volumes are almost as fast as bare metal
- Perf depends on remote storage
- Perf depends on cloud provider
- You need to test it by your own
- Tools:
  - Sysbench
  - Pg_bench
- Recommended to watch
  - Benchmarking Cloud Native Storage - Josh Berkus, Red Hat – KubeCon Europe 2019 - https://www.youtube.com/watch?v=4V-4yPSfN3U
- It’s not myth

Myth 3: Storing data on k8s requires remote distributed storage

- It’s myth
- We have an options:
  - Persistent Local Volume
  - Host Path – don’t use it unless you know what you are doing

Myth 4: Never run DB on k8s

- It’s myth
- But:

DB as a service >>>> DB in K8s

Summary

Takeaways

- Storage on K8s is not that complicated as many think
- Persistent Local Volume is ‘*almost*’ as fast as bare metal storage
- Don’t use direct references of volumes in your Pod
- Use dynamic provisioning
- Still DB as a service is better (not cheaper) than DB on K8s

Q&A