kubernetes

Foundation

One-on-One Video Training

Downloadable Resources

Certificate of Comlpletion

Prerequisites

- Familiarity with Docker
- Comfortable with Command line interface (Windows/Linux/Mac)
- Familiarity with Git/GitHub
- Visual Studio Code
- Concepts - YAML
- Tools: Minikube/Kind, kubectl, Helm

What is Kubernetes

Kubernetes, often abbreviated as K8s, is an open-source container orchestration platform that automates the deployment, scaling, and management of containerized applications.

Originally developed by Google and now maintained by the Cloud Native Computing Foundation (CNCF), Kubernetes is designed to handle the complexities of running applications in production environments across distributed systems.

Key Features

- Container Orchestration
- Automatic Scaling
- Self-Healing
- Service Discovery and Load Balancing
- Declarative Configuration
- Rolling Updates and Rollbacks
- Storage Orchestration

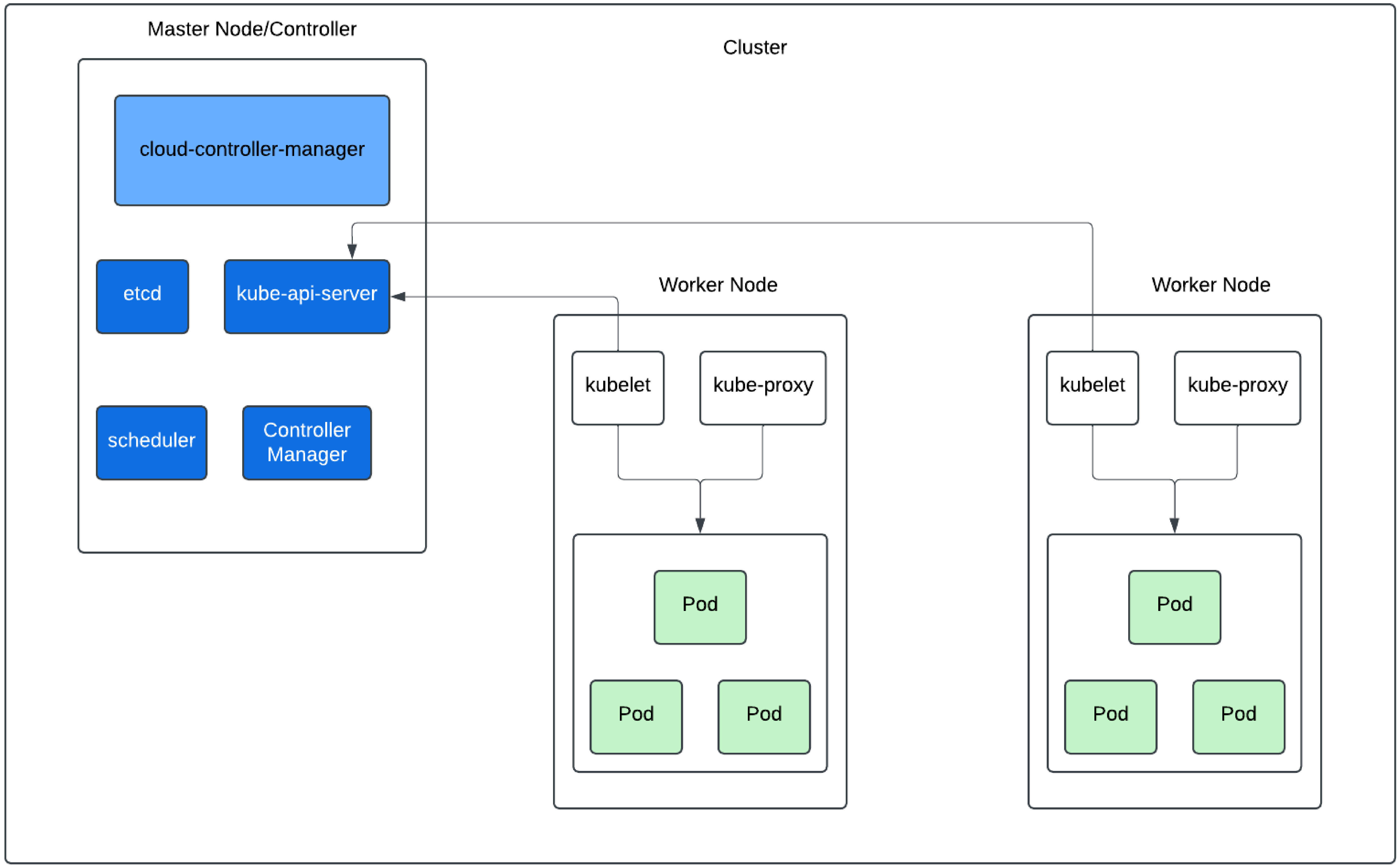
Why Kubernetes

- **Efficient Resource Utilization:**
  - Kubernetes optimizes resource usage across nodes, ensuring applications run efficiently.
- **Resilience and High Availability:**
  - It ensures applications remain operational even in case of node failures or disruptions.
- **Portability**:
  - Kubernetes works on-premises, in public clouds, or in hybrid environments, making it platform-agnostic.
- **Ecosystem Integration**:
  - Kubernetes integrates with a vast ecosystem of tools for CI/CD, monitoring, logging, and more.
- **Community and Support**:
  - As a widely adopted open-source platform, Kubernetes has a vibrant community and extensive documentation.

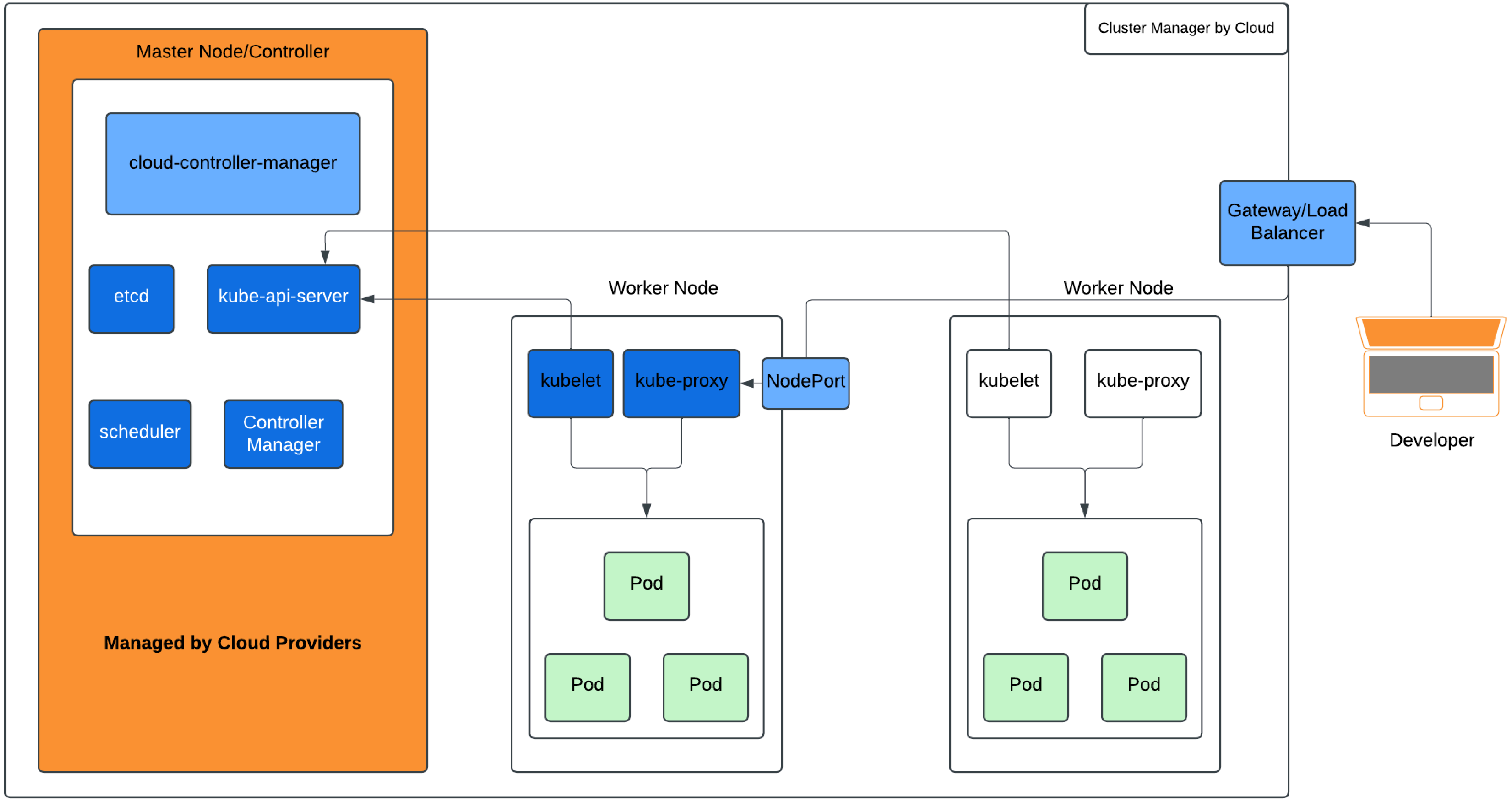
Concepts

- **Cluster** - Group of connected Computing devices
- **Control Plane** - Master nodes responsible for managing the cluster
- **Worker Nodes** - Computing nodes responsible for running application
- **Pod** - Group of container which could be scheduled on a node
- **Containers** - Single instance of running container (eg docker)
- **YAML** - Yet Another Markup Language/YAML Ain't Markup Language

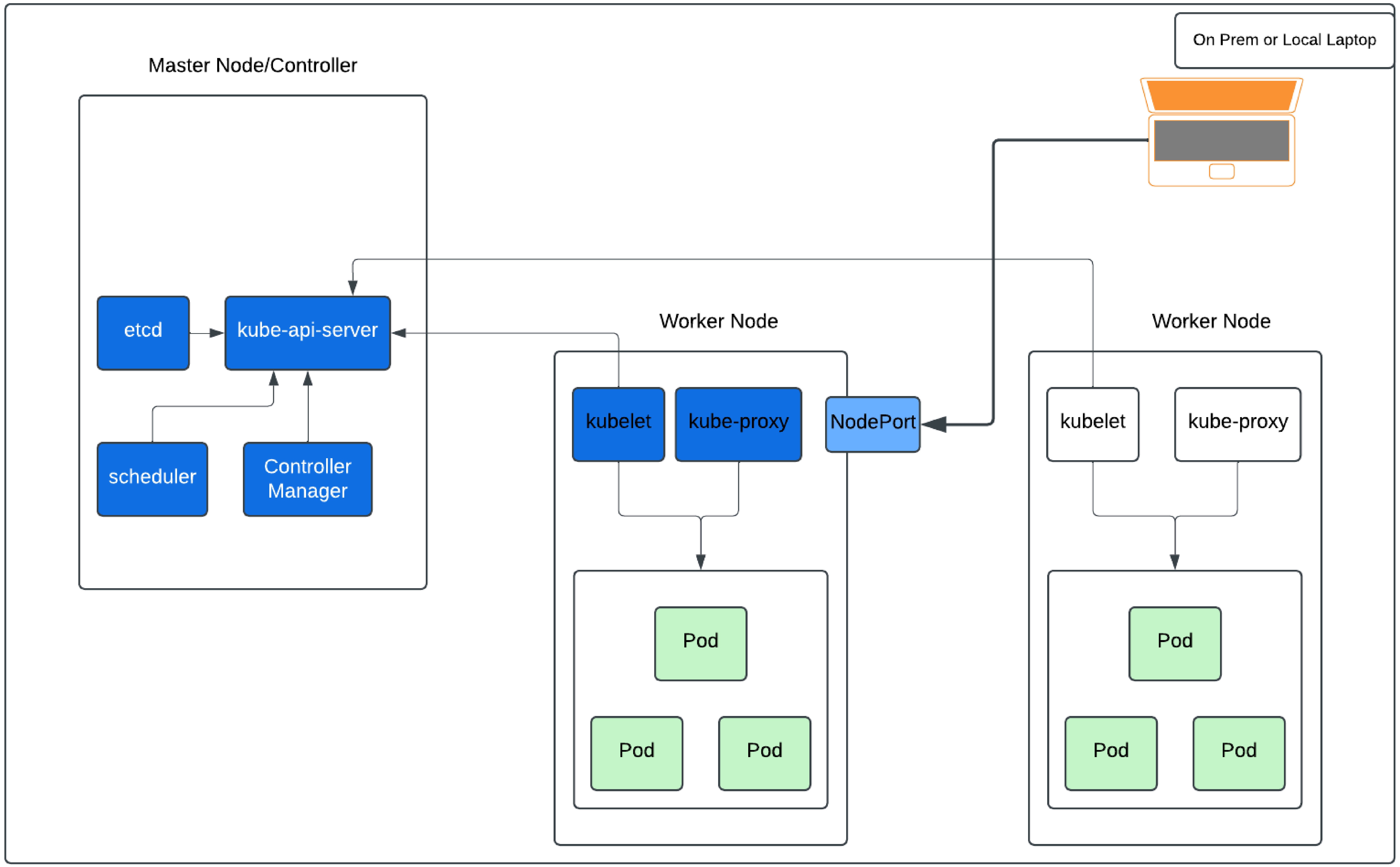
Kubernetes Architecture



Kubernetes Architecture - Public Cloud Implementation



Kubernetes Architecture - On-Prem Implementation



Control Plane Components

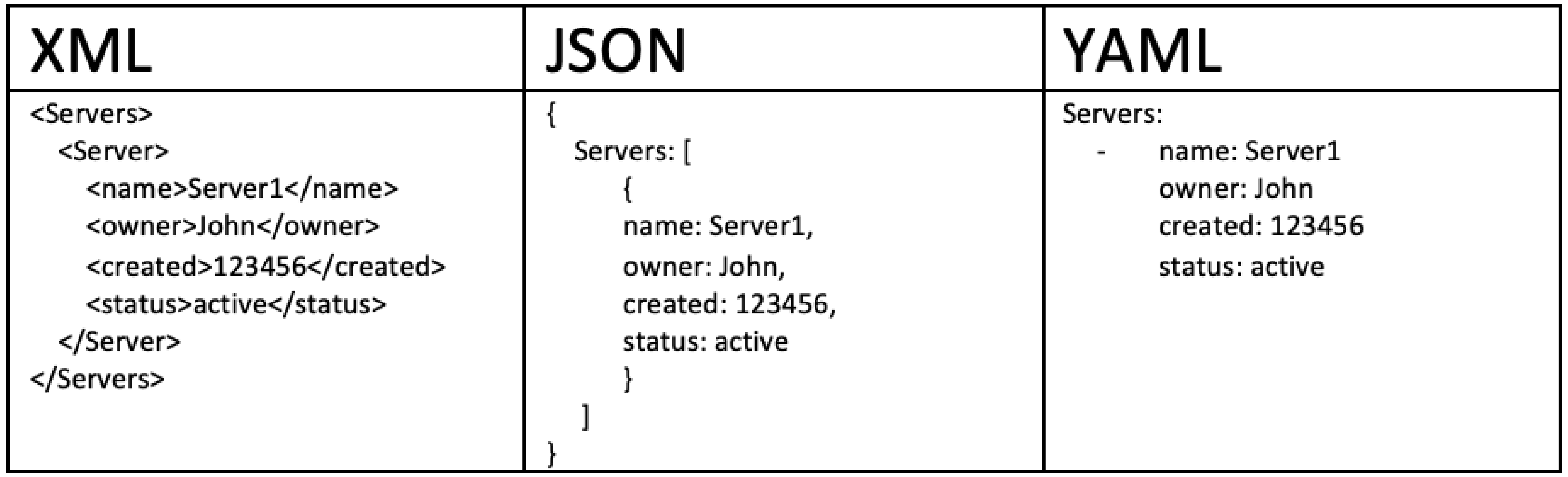
- **kube-apiserver**
  - The core component server that exposes the Kubernetes HTTP API
- **etcd**
  - Consistent and highly-available key value store for all API server data
- **kube-scheduler**
  - Looks for Pods not yet bound to a node, and assigns each Pod to a suitable node.
- **kube-controller-manager**
  - Runs controllers to implement Kubernetes API behavior.
- **cloud-controller-manager (optional)**
  - Integrates with underlying cloud provider(s)

Node Components

- **kubelet**
  - Ensures that Pods are running, including their containers

- **kube-proxy**
  - Maintains network rules on node to implement Services
- **Container runtime**
  - Software responsible for running containers (docker,containerd etc)
- **cloud-controller-manager (optional)**
  - Integrates with underlying cloud provider(s)

YAML



XML

```
<Servers>
  <Server>
    <name>Server1</name>
    <owner>John</owner>
    <created>123456</created>
    <status>active</status>
  </Server>
</Servers>
```

JSON

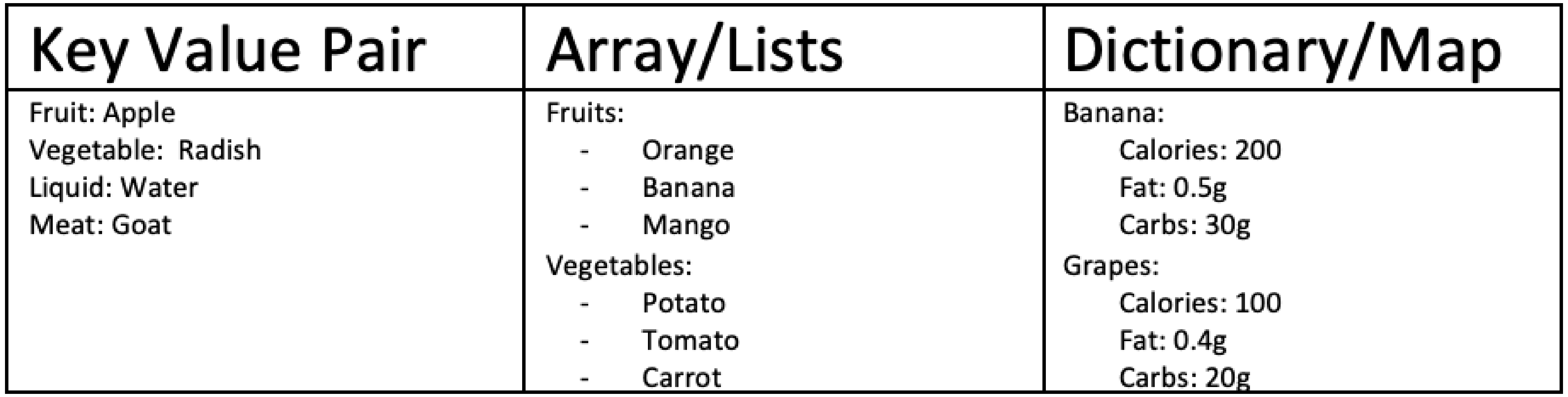
```
{
  Servers: [
      {
      name: Server1,
      owner: John,
      created: 123456,
      status: active
      }
    ]
}
```

YAML

```
Servers:
  -     name: Server1
        owner: John
        created: 123456
        status: active
```

- **Declarative**: You describe what you want (e.g., "2 replicas of my app") rather than how to achieve it.
- **Key-Value Pairs**: YAML uses indentation to represent hierarchical relationships.
- **Reusability**: YAML files can be reused and shared for consistent deployments.

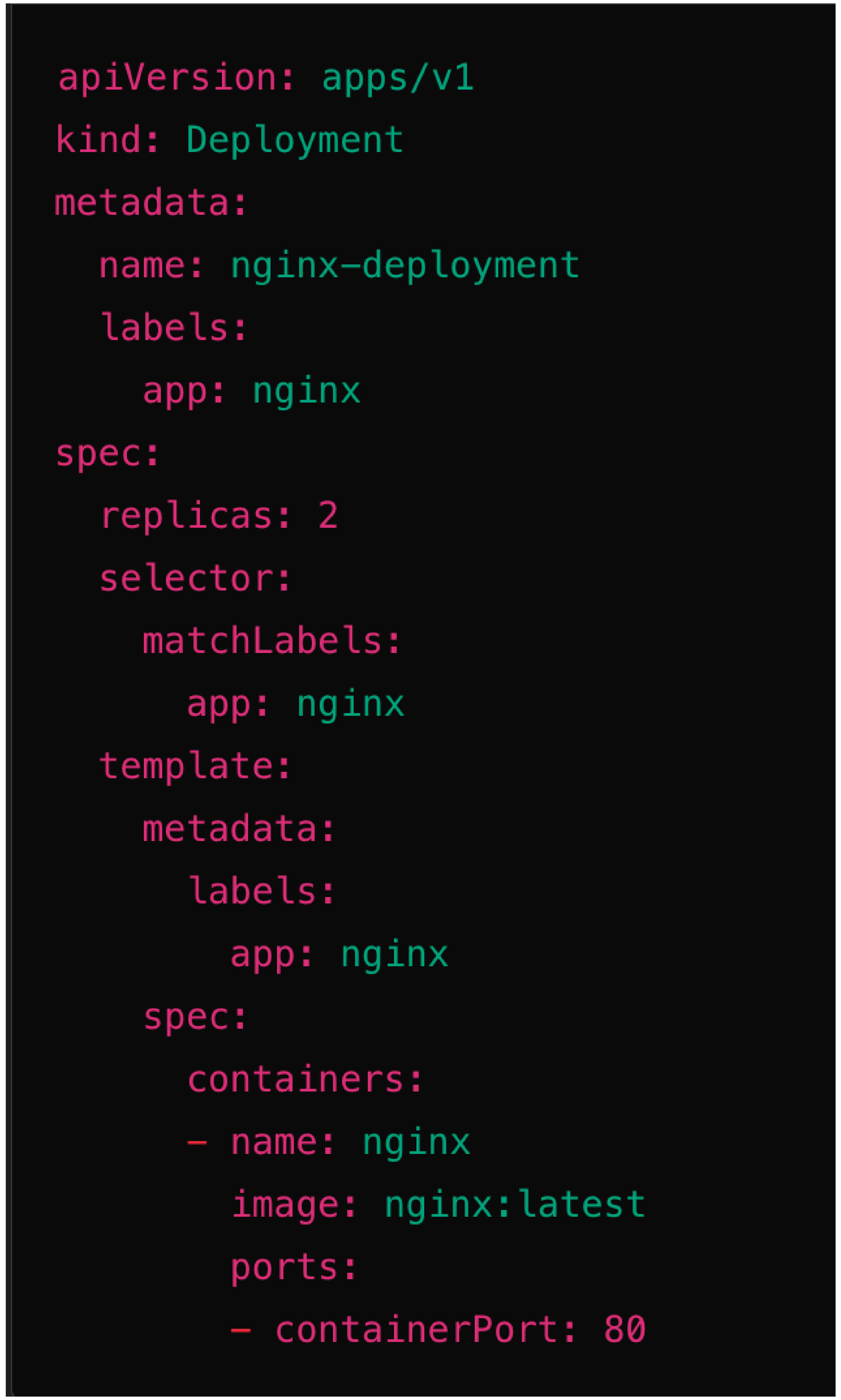
YAML



Key Value Pair	Array/Lists	Dictionary/Map
Fruit: Apple	Fruits:	Banana:
Vegetable: Radish	- Orange	Calories: 200
Liquid: Water	- Banana	Fat: 0.5g
Meat: Goat	- Mango	Carbs: 30g
	Vegetables:	Grapes:
	- Potato	Calories: 100
	- Tomato	Fat: 0.4g
	- Carrot	Carbs: 20g

- Human Readable: Easy for developers and operators to understand.
- Version Control: YAML files can be tracked in Git for collaboration and history.
- Reusable: Use the same file for different environments by parameterizing values (e.g., using Helm).

YAML in k8s



```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:latest
        ports:
        - containerPort: 80
```

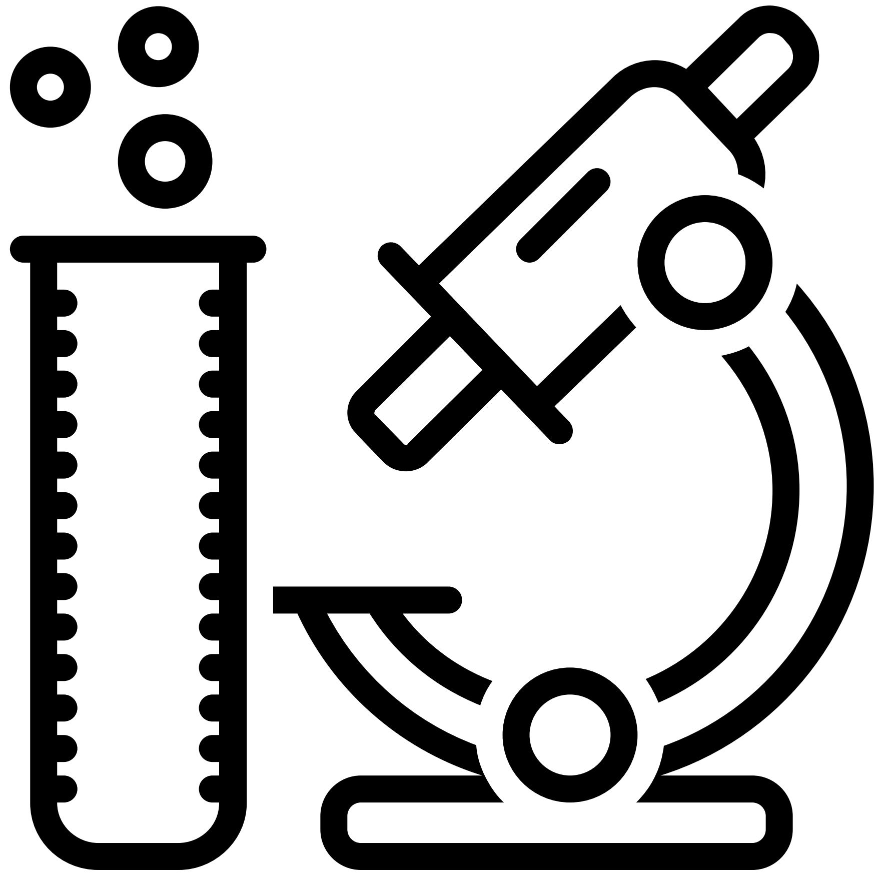
Pod Basics

- **Smallest Deployable Unit**: A pod is the smallest unit in Kubernetes that you can create or manage.

- **Single or Multiple Containers**:
  - Typically contains a single container (e.g., Docker container).
  - Can host multiple containers that share resources and are tightly coupled.

- **Shared Resources**:
  - Networking: All containers in a pod share the same network namespace, meaning they share a single IP address and port space.
  - Storage: Containers in a pod can share mounted volumes.
- **Ephemeral Nature**: Pods are designed to be temporary. If a pod dies, Kubernetes replaces it with a new one
- **Communication**:
  - Containers within a pod can communicate directly via localhost.
  - External communication is managed via services or ingress.
- **Lifecycle**:
  - Pods have a well-defined lifecycle (Pending, Running, Succeeded, Failed, etc.).

Lab 1 - Pod Basics



- Setup kubectl
- Deploy your first Application
- Explore kubectl command options

Key Concepts

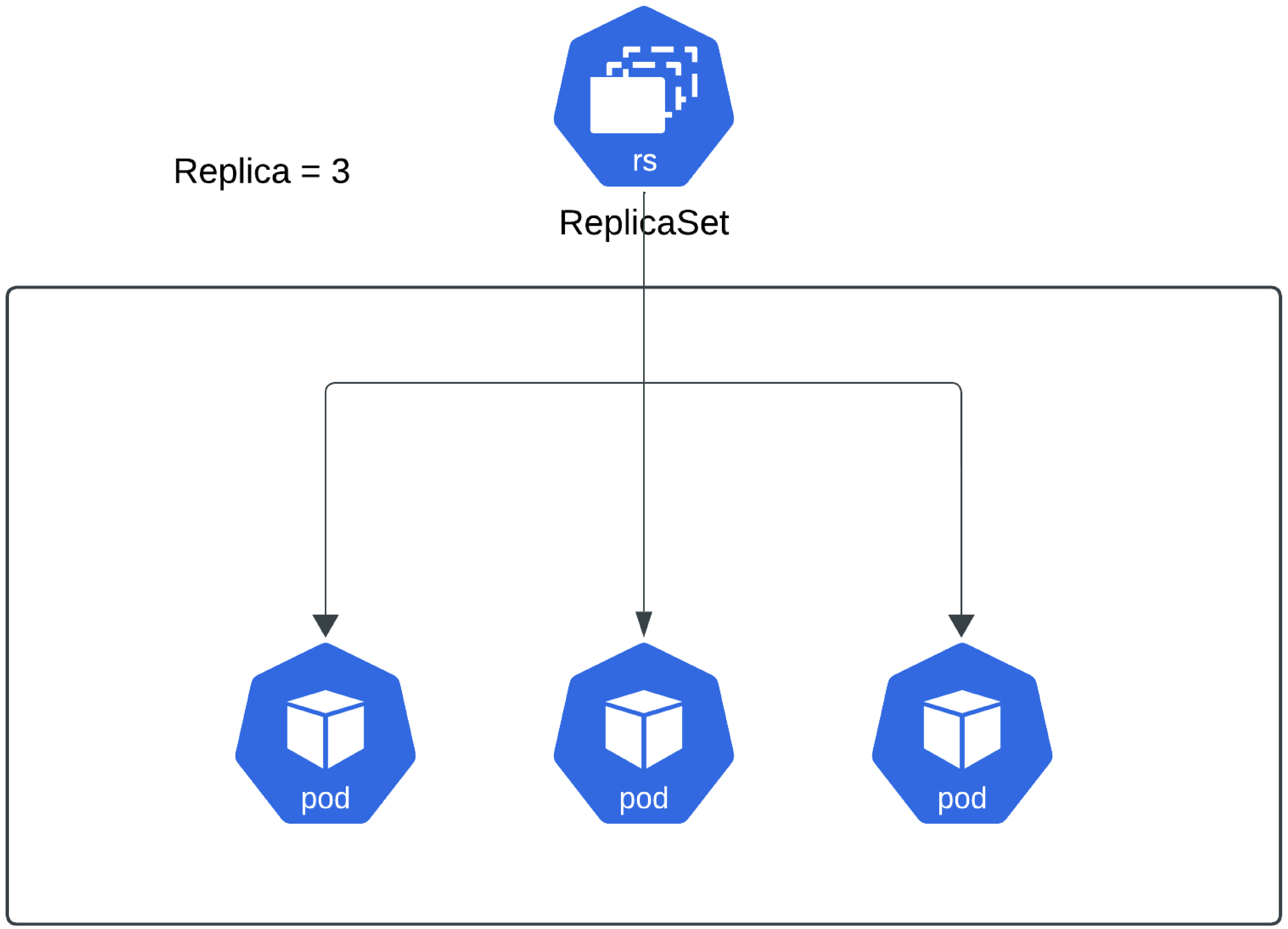
- **ReplicaSet**
- **Deployment**
- **StatefulSets**
- **DaemonSet**

ReplicaSet

- **Ensures Desired Pod Replication**: A ReplicaSet guarantees that a specified number of identical pods are running at any given time, maintaining availability and scalability.

- **Automatic Pod Replacement**: If a pod fails or is deleted, the ReplicaSet automatically creates a new one to replace it, ensuring the desired number of replicas is met.

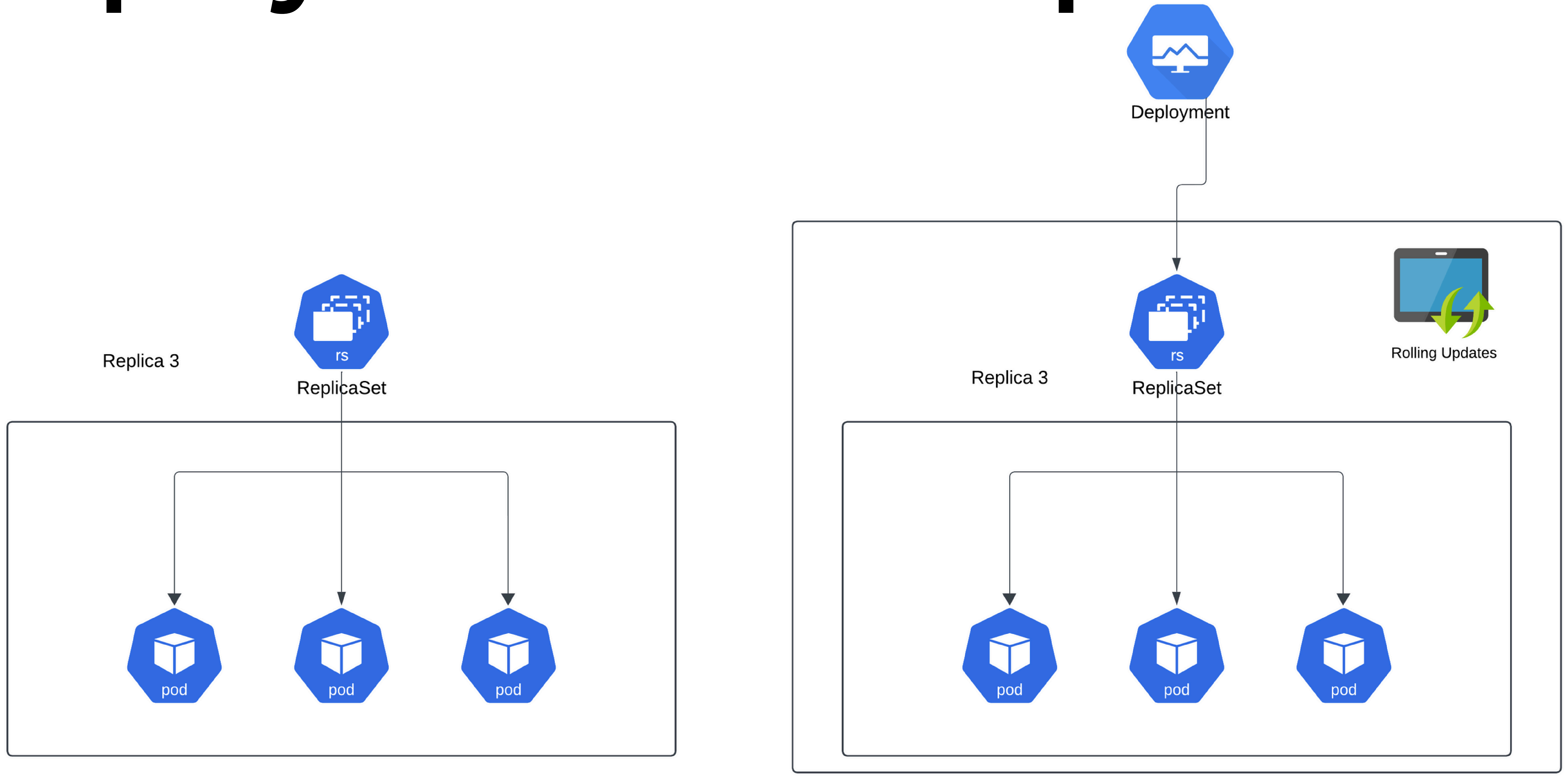
ReplicaSet



Deployments

- **Manages ReplicaSets:** A Deployment automates the creation, scaling, and management of ReplicaSets, ensuring desired pod availability.
- **Supports Rolling Updates**: Deployments handle rolling updates, allowing you to update your application with zero downtime by gradually replacing old pods with new ones.
- **Automatic Rollback:** If a deployment update causes issues, Kubernetes can automatically roll back to a previous, stable version of the application.
- **Declarative Management:** Deployments provide a declarative way to manage application states, allowing you to specify the desired state and letting Kubernetes handle the rest.

Deployments and Replicasets

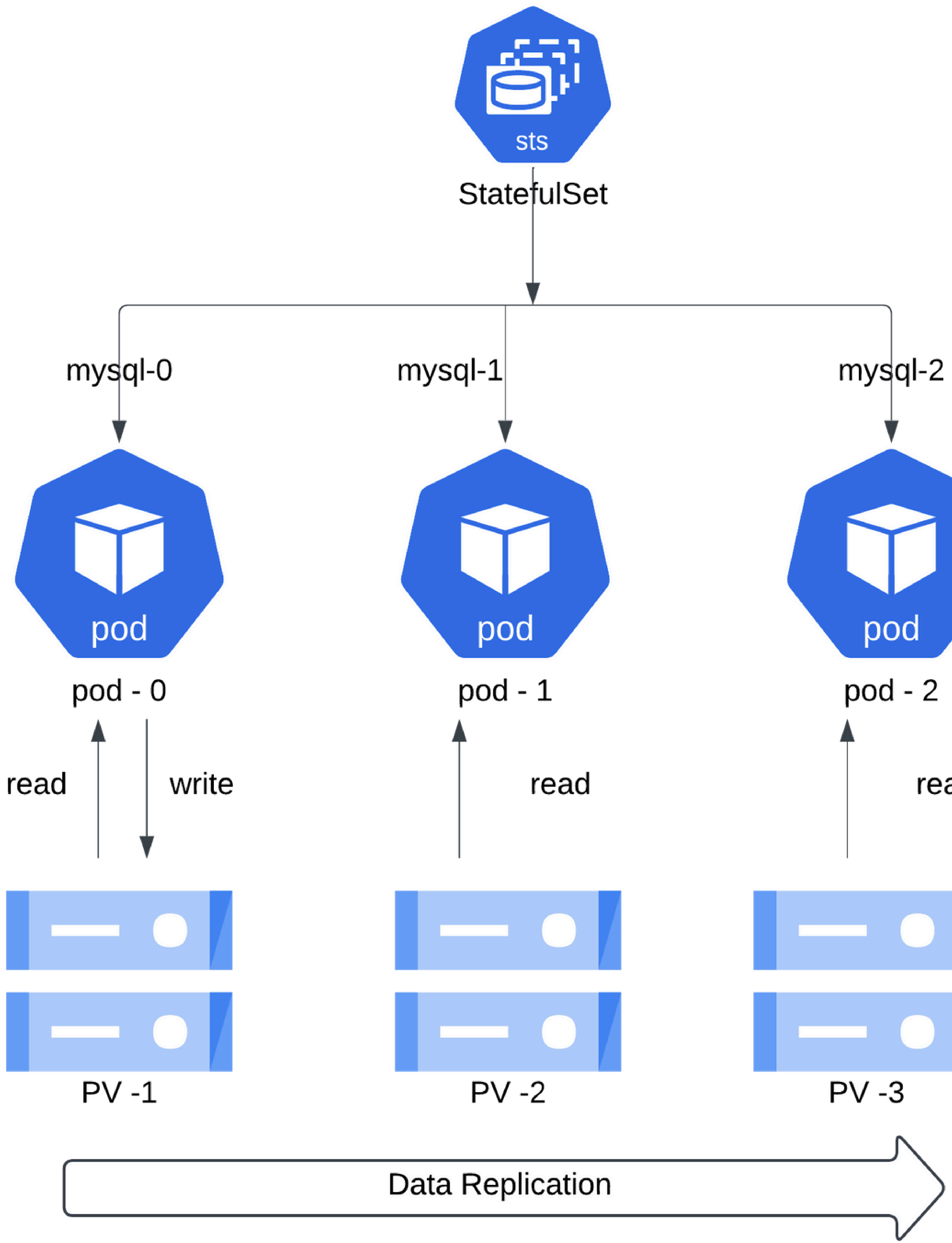


Feature	Deployment	ReplicaSet
Primary Purpose	Manages ReplicaSets and controls application lifecycle (updates, rollbacks)	Ensures a specified number of identical pods are running
Rolling Updates	Supports rolling updates to ensure zero downtime during application updates	Does not handle updates, just maintains the number of replicas
Automatic Rollback	Can automatically roll back to a previous version if an update fails	No support for rollbacks
Abstraction Level	Higher-level abstraction that manages ReplicaSets	Lower-level, more granular control of pod replication
Use Case	Preferred for application deployment, scaling, and version control	Used for more fine-grained control or specific use cases
Pod Management	Automatically creates and manages ReplicaSets to handle pod scaling	Directly manages pods, ensuring the desired number are always running
Lifecycle Management	Manages the entire lifecycle of applications, including upgrades and rollbacks	Only manages the replication of pods, with no built-in update mechanism

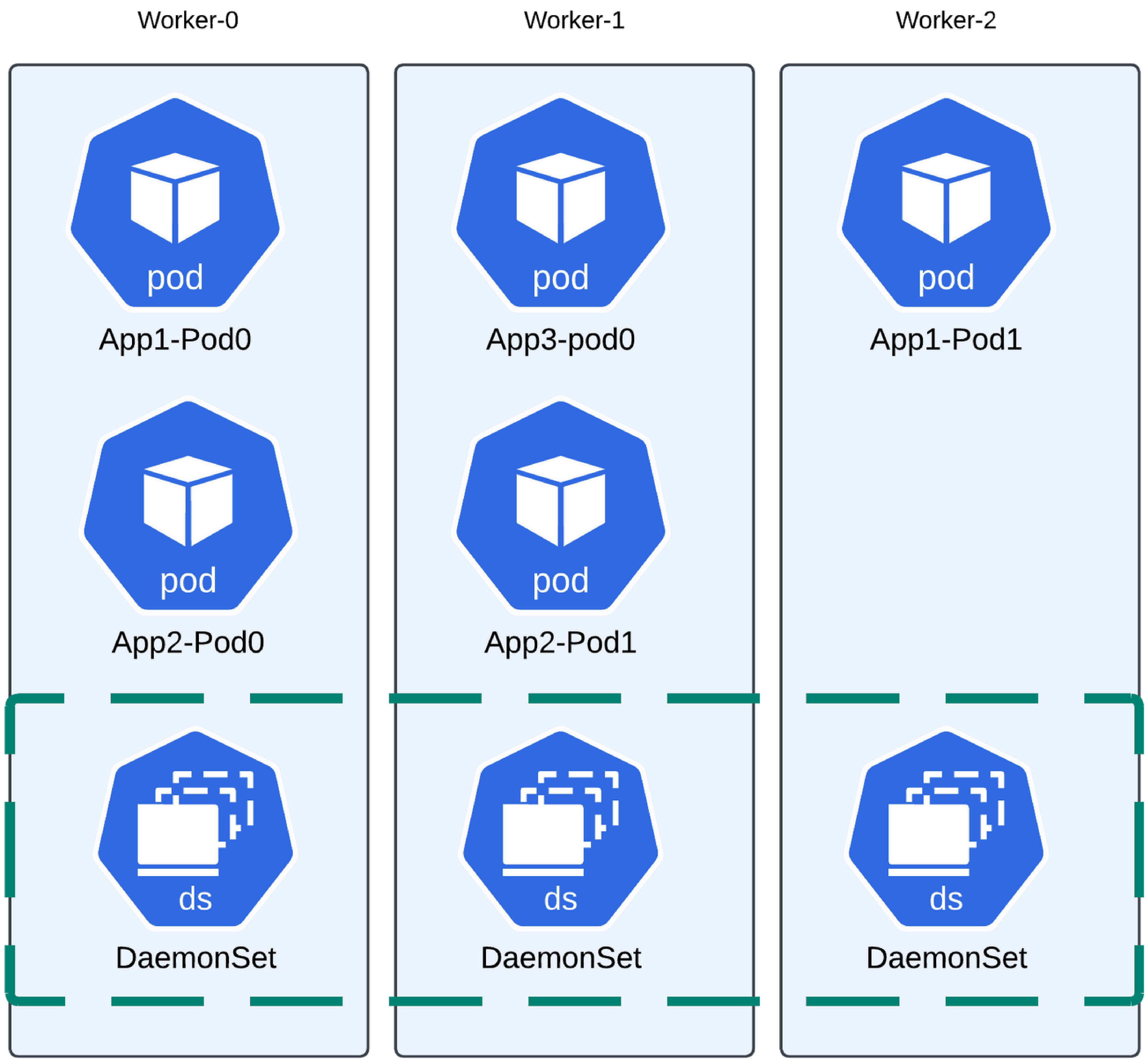
StatefulSets

- **Manages Stateful Applications**: StatefulSets are designed for applications that require stable, unique network identifiers, persistent storage, and ordered deployment/termination (e.g., databases, distributed systems).
- **Stable Network Identity:** Each pod in a StatefulSet gets a unique, stable network identity (e.g., pod-0, pod-1), which is essential for applications that rely on stable endpoints.
- **Persistent Storage:** StatefulSets ensure that each pod has its own persistent storage volume that is not shared with other pods, even if the pod is rescheduled or recreated.
- **Ordered Deployment and Scaling:** Pods are created and deleted in a strict order (one at a time), ensuring predictable scaling and deployment, which is important for stateful applications like databases.
- **Support for Rolling Updates:** StatefulSets provide rolling updates for stateful applications while preserving the identity and data of each pod during the update process.

StatefulSets



DaemonSets

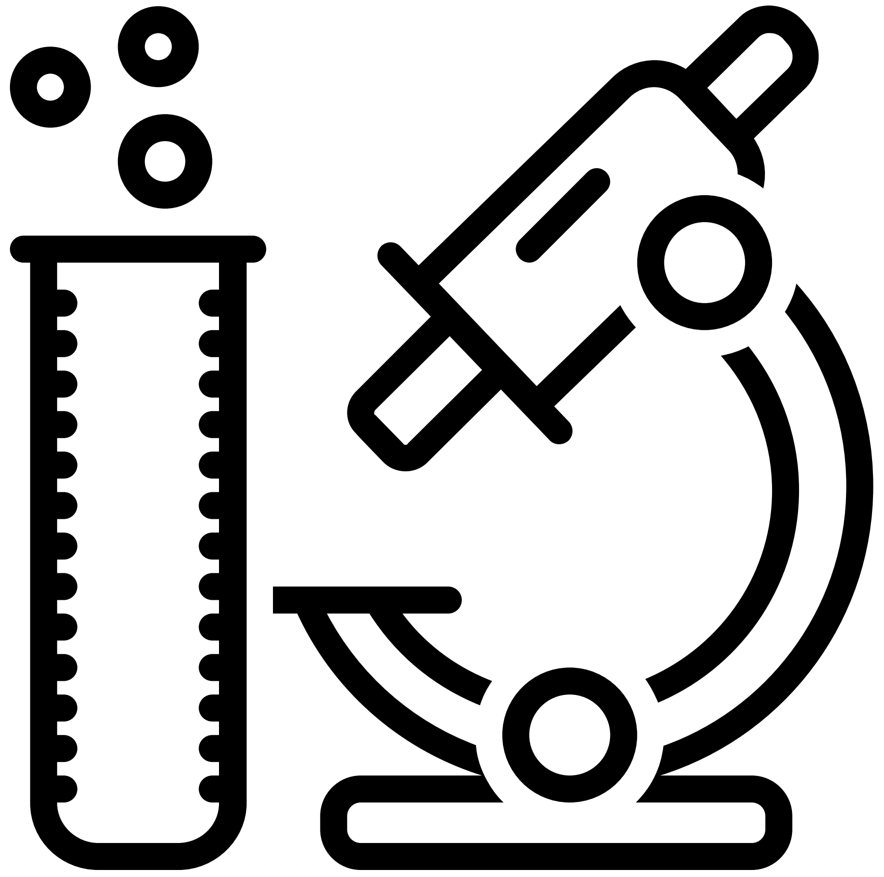


Worker-0
Worker-1
Worker-2
pod
App1-Pod0
pod
App3-pod0
pod
App1-Pod1
pod
App2-Pod0
pod
App2-Pod1
ds
DaemonSet
ds
DaemonSet
ds
DaemonSet

DaemonSets

- **Ensures Pod Deployment on All Nodes**: A DaemonSet ensures that a specific pod runs on all (or selected) nodes in a Kubernetes cluster, making it ideal for cluster-wide services (e.g., logging, monitoring).
- **Automatic Pod Management:** When new nodes are added to the cluster, DaemonSets automatically deploy the associated pod to the new nodes.
- **Pod Deletion on Node Removal:** When nodes are removed from the cluster, the DaemonSet automatically deletes the associated pods on those nodes.
- **Selective Node Deployment:** DaemonSets can be configured to deploy pods only on specific nodes using node selectors, taints, and tolerations.
- **Common Use Cases:** DaemonSets are typically used for running cluster-wide services like log collection (Fluentd), monitoring (Prometheus node exporter), or network management

Lab - Workload Concepts



- Deployments
- ReplicaSet
- StatefulSet
- DaemonSet

Kubernetes Network Model

- Pod Networking
  - Each pod receives a unique, cluster-wide IP address.
  - Containers within a pod share the same network namespace and communicate over localhost.

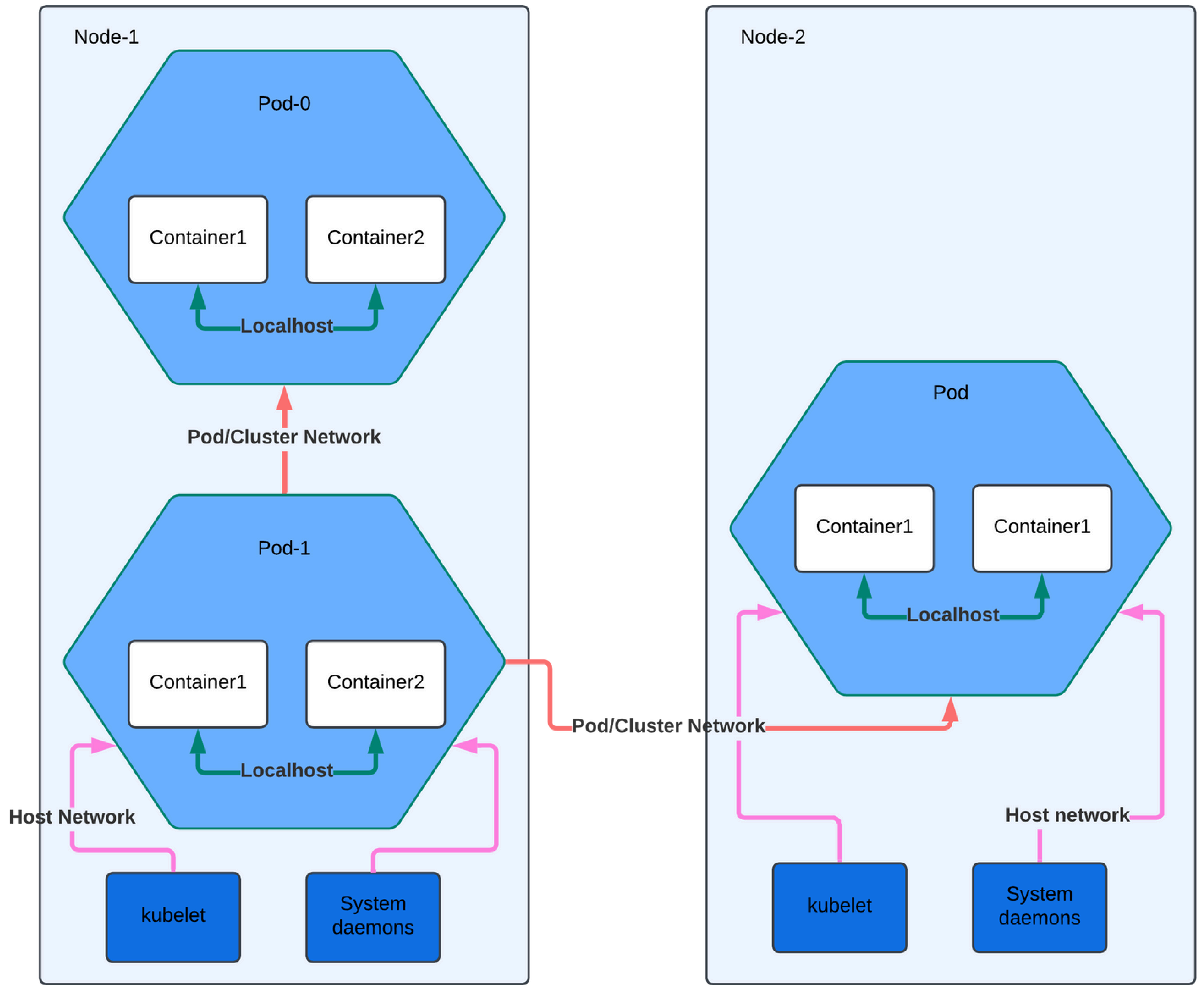
- Cluster Networking
  - All pods can communicate directly across nodes without NAT or proxies.
  - Node-level agents (e.g., kubelet) can access all pods on their node.

- Services & Endpoint Management
  - The Service API offers a stable IP/hostname for a set of backend pods.
  - Kubernetes manages EndpointSlice objects to track the pods backing each Service.
  - A service proxy dynamically programs the data plane to route traffic to the correct pods.

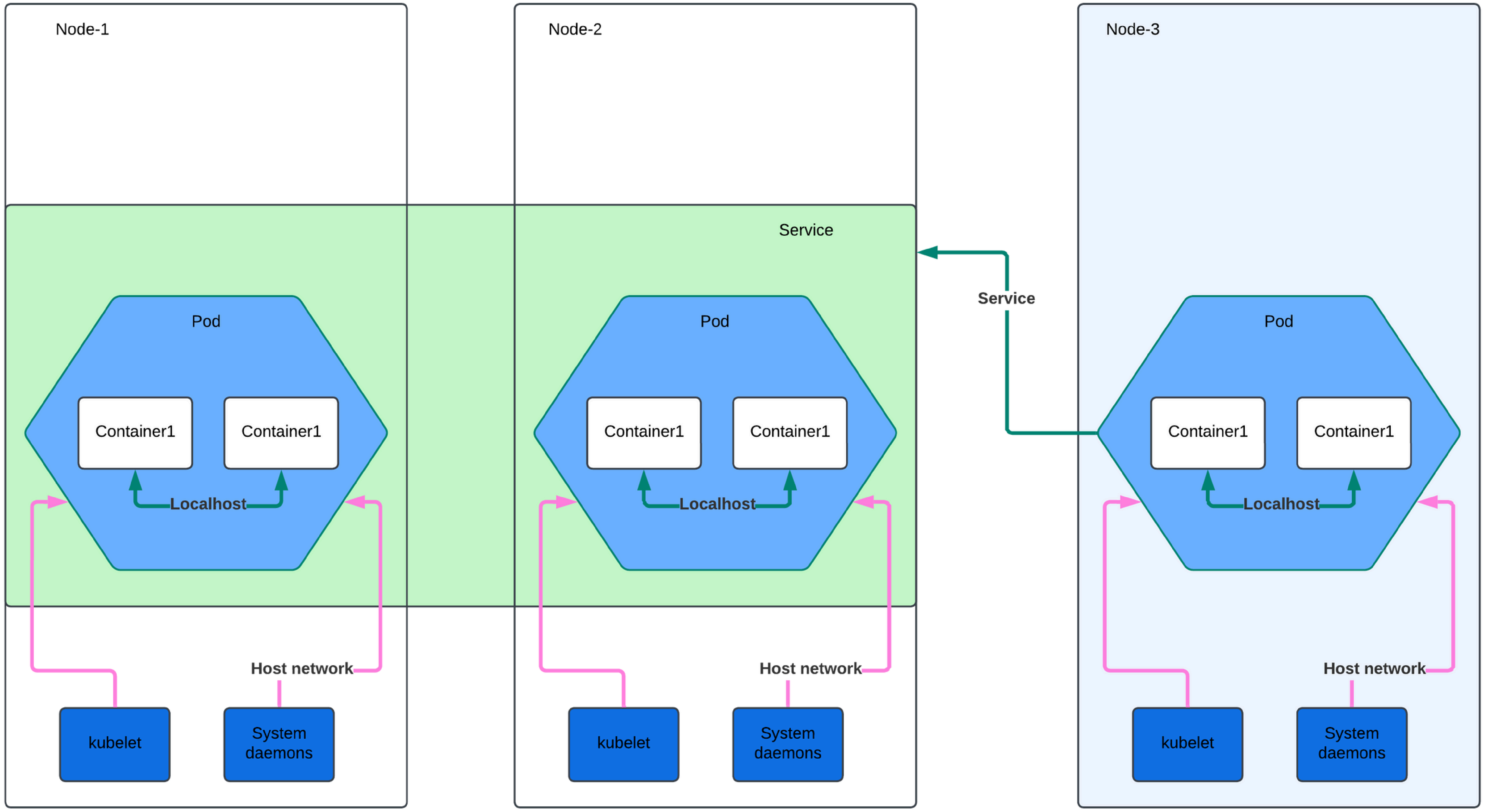
- External Access
  - The Gateway API exposes services to external clients.
  - The LoadBalancer service type provides a straightforward ingress option on supported cloud providers.

- Security
  - NetworkPolicy allows fine-grained control over traffic between pods and external endpoints.

Network Model



Service

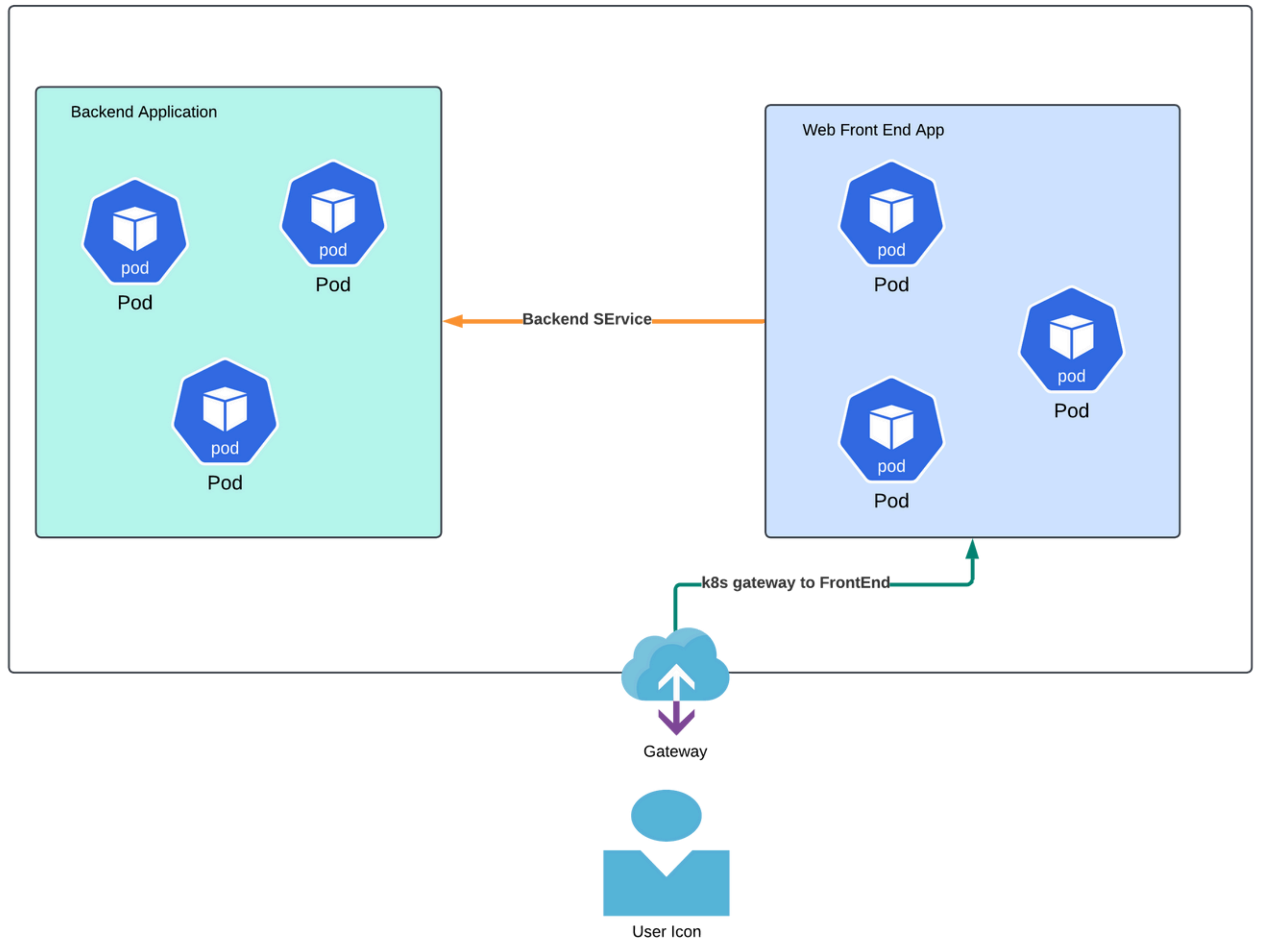

Node-1
Node-2
Node-3
Service
Service
Pod
Pod
Pod
Container1
Container1
Container1
Container1
Container1
Container1
Localhost
Localhost
Localhost
Host network
Host network
Host network
kubelet
System daemons
kubelet
System daemons
kubelet
System daemons

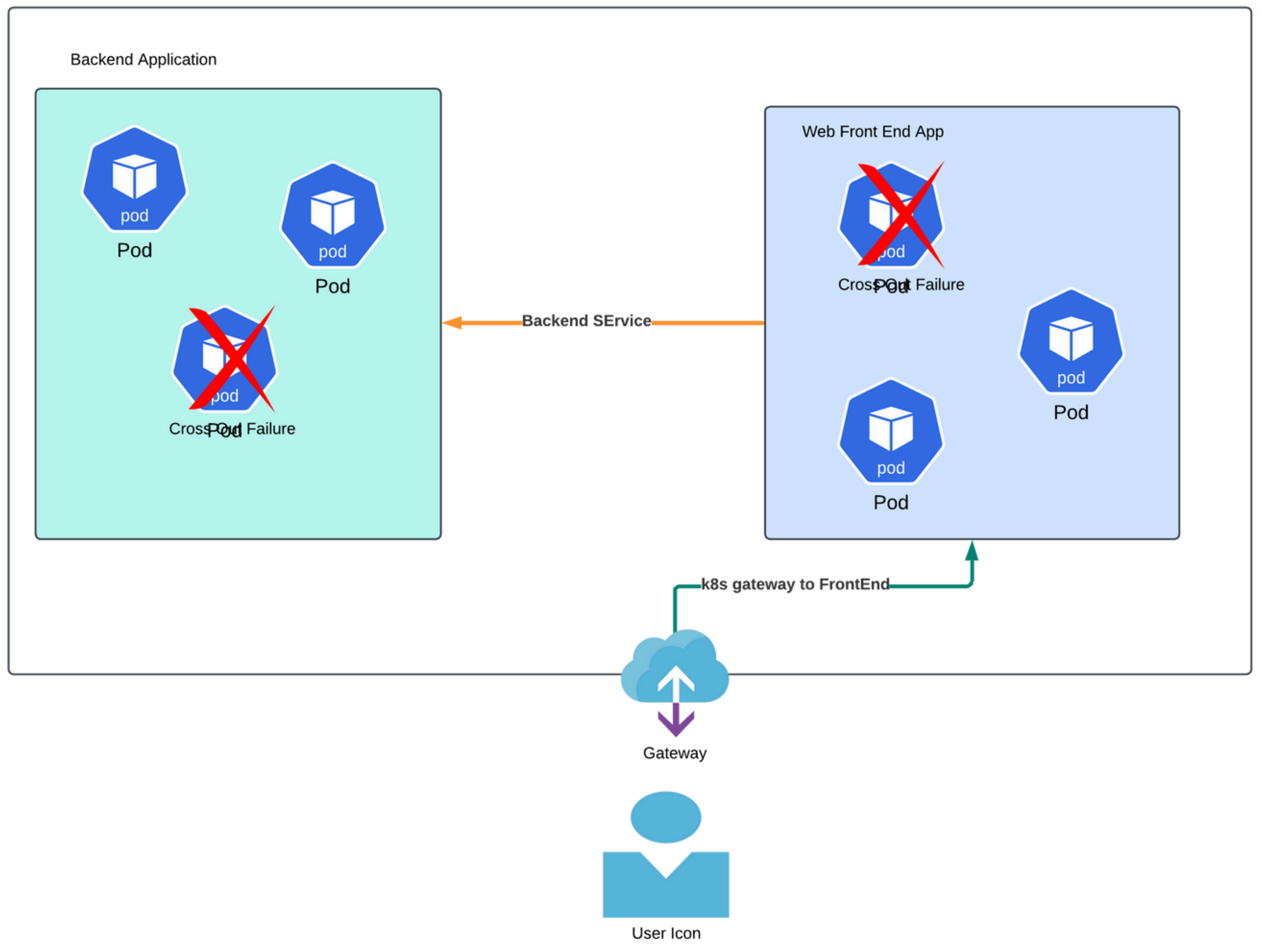
Service

- Expose an application running in your cluster behind a single outward-facing endpoint, even when the workload is split across multiple backends.

- The Service API abstracts network access to groups of pods. It defines logical endpoints (typically pods) and enforces policies for making them accessible.

Services in Kubernetes





Service Types

- **ClusterIP**
    - Exposes the Service on a cluster-internal IP. Choosing this value makes the Service only reachable from within the cluster. This is the default that is used if you don't explicitly specify a type for a Service. You can expose the Service to the public internet using an Ingress or a Gateway.

- **NodePort**
    - Exposes the Service on each Node's IP at a static port (the NodePort). To make the node port available, Kubernetes sets up a cluster IP address, the same as if you had requested a Service of type: ClusterIP.

- **LoadBalancer**
    - Exposes the Service externally using an external load balancer. Kubernetes does not directly offer a load balancing component; you must provide one, or you can integrate your Kubernetes cluster with a cloud provider.

- **ExternalName**
    - Maps the Service to the contents of the externalName field (for example, to the hostname api.foo.bar.example). The mapping configures your cluster's DNS server to return a CNAME record with that external hostname value. No proxying of any kind is set up.

Service Types

- **ClusterIP**
    - Exposes the Service on a cluster-internal IP. Choosing this value makes the Service only reachable from within the cluster. This is the default that is used if you don't explicitly specify a type for a Service. You can expose the Service to the public internet using an Ingress or a Gateway.
- **NodePort**
    - Exposes the Service on each Node's IP at a static port (the NodePort). To make the node port available, Kubernetes sets up a cluster IP address, the same as if you had requested a Service of type: ClusterIP.
- **LoadBalancer**
    - Exposes the Service externally using an external load balancer. Kubernetes does not directly offer a load balancing component; you must provide one, or you can integrate your Kubernetes cluster with a cloud provider.
- **ExternalName**
    - Maps the Service to the contents of the externalName field (for example, to the hostname api.foo.bar.example). The mapping configures your cluster's DNS server to return a CNAME record with that external hostname value. No proxying of any kind is set up.

Lab - Network Concepts

Volumes

- Data Persistence:
  - Container files are ephemeral and lost when a container crashes or restarts.
  - Kubelet restarts the container with a clean state, losing all modifications.
- Shared Storage:
  - Multiple containers in a pod may need to share files.
  - Setting up a shared filesystem across containers can be complex.

Volumes

Kubernetes volumes provide a way for containers in a pods to access and share data via the filesystem.

- Populating a configuration file based on a ConfigMap or a Secret
- Providing some temporary scratch space for a pod
- Sharing a filesystem between two different containers in the same pod
- Sharing a filesystem between two different pods (even if those Pods run on different nodes)
- Durably storing data so that it stays available even if the Pod restarts or is replaced

Persistent Volumes

Kubernetes Persistent Storage

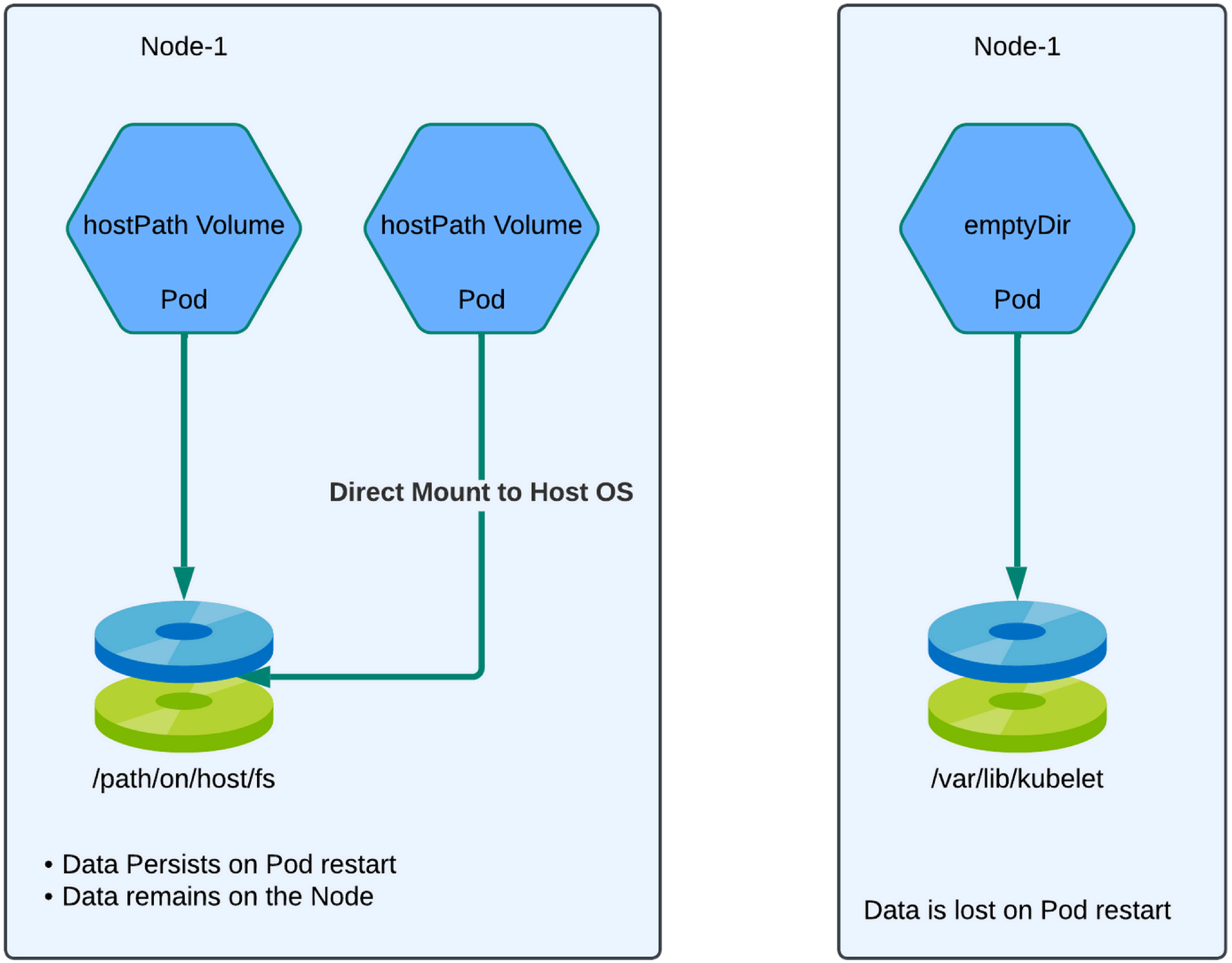
- Storage management is separate from compute management.

- The PersistentVolume (PV) subsystem abstracts storage provisioning from consumption.
    - Key API resources:PersistentVolume (PV) – Represents a provisioned storage unit.
    - PersistentVolumeClaim (PVC) – A request by a pod for storage.

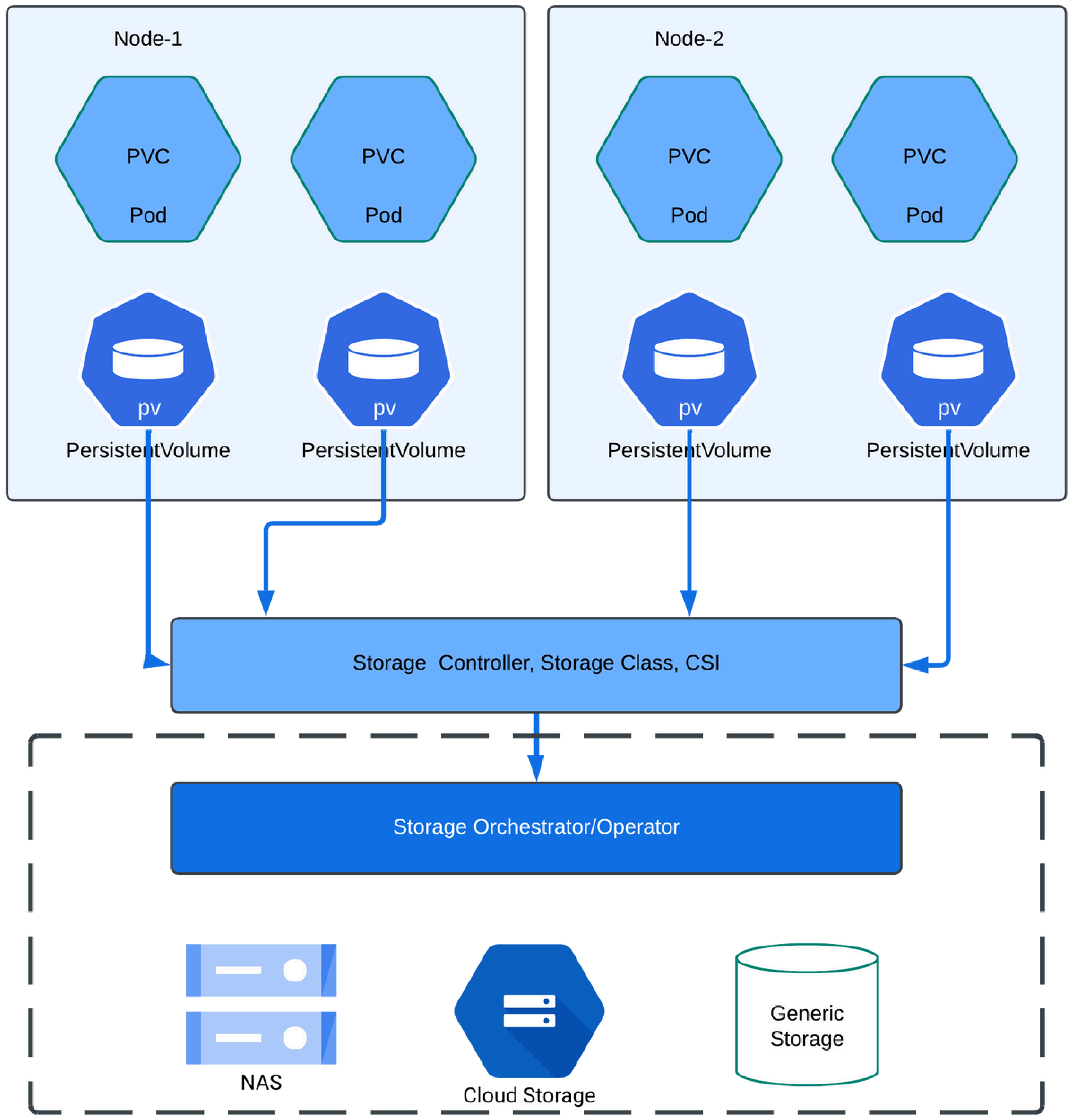
Persistent Volumes Types

- emptyDir – Temporary storage that exists as long as the pod runs.
- hostPath – Mounts a host machine directory into the pod.
- PersistentVolume (PV) & PersistentVolumeClaim (PVC) – Abstracts storage for long-term persistence.
- configMap & secret – Used for injecting configuration data and credentials.
- CSI (Container Storage Interface) – Enables external storage providers to integrate with Kubernetes.

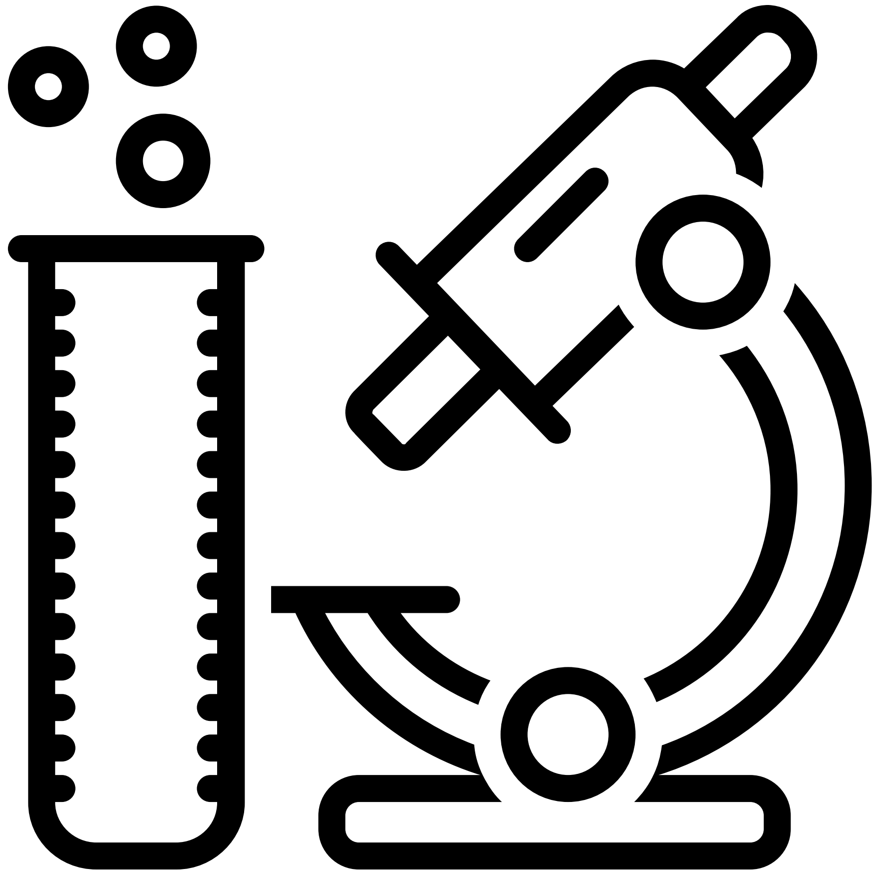
Hostpath emptyDir



PVC


Node-1
Node-2
PVC
Pod
PVC
Pod
PVC
Pod
PVC
Pod
pv
PersistentVolume
pv
PersistentVolume
pv
PersistentVolume
pv
PersistentVolume
Storage Controller, Storage Class, CSI
Storage Orchestrator/Operator
NAS
Cloud Storage
Generic Storage

Lab - Storage Concepts



- HostPath
- emptyDir
- PVC

Labels & Selectors

- What are Labels?
  - Key/value pairs attached to Kubernetes objects
  - Used to organize, group, and select resources
- Label Selectors:
  - Identify subsets of objects
  - Example: app: frontend, env: production
- Usage:
  - Deployments, Services, and other controllers use selectors for targeting

Annotations

- What are Annotations?
    - Key/value pairs for storing non-identifying metadata
    - Ideal for debugging, build information, or external tool integrations
- Difference from Labels:
    - Not used for grouping/selecting objects
    - Can contain larger, unstructured data (e.g., JSON, URLs)
- Usage Example:
    - buildInfo: "v1.2.3, built on 2025-02-18"

Taints and Tolerations

- Taints:
  - Applied to nodes to repel specific pods
  - Format: key=value:effect (e.g., dedicated=database:NoSchedule)
- Tolerations:
  - Applied to pods to allow them to be scheduled on tainted nodes
- Use Case:
  - Reserve nodes for particular workloads (e.g., databases)

Affinity & Anti-Affinity

- Node Affinity:
  - Rules for scheduling pods on specific nodes based on node labels
  - Example: Schedule pods only on nodes with region: us-east
- Pod Affinity/Anti-Affinity:
  - Affinity: Co-locate pods with similar characteristics
  - Anti-Affinity: Spread pods across nodes for high availability
- Use Case:
  - Optimize performance and fault tolerance

Assignment

45 Minutes

Pod Eviction

- Kubernetes evicts (removes) pods when:
  - Nodes experience resource pressure (CPU, memory, storage).
  - Nodes are cordoned or drained for maintenance.
  - Scheduling policies prioritize higher-priority workloads.
-
- Evictions help maintain cluster stability by freeing up resources.

Key Concept:

- Evictions ≠ Pod Failures → Evicted pods can restart on other nodes if allowed by scheduling rules.

Pod Eviction

- Voluntary Evictions (Planned by Kubernetes/Admins)
  - When nodes are drained for maintenance.
  - When PodDisruptionBudget (PDB) allows controlled removal.
  - When a Deployment or StatefulSet scales down.

- Involuntary Evictions (Triggered by Kubernetes)
  - When a node experiences resource pressure (out of memory, disk space, etc.).
  - When Quality of Service (QoS) priorities determine which pods are removed.
  - When a node becomes unavailable (crash, network failure).

Node Cordon & Drain

- Cordon:
  - Prevents new pods from scheduling on a node

```
kubectl cordon <node-name>
```

- Drain
  - Evicts all non-DaemonSet pods from node before maintenance

```
kubectl drain <node-name> --ignore-daemonsets --delete-local-data
```

Quality of Service

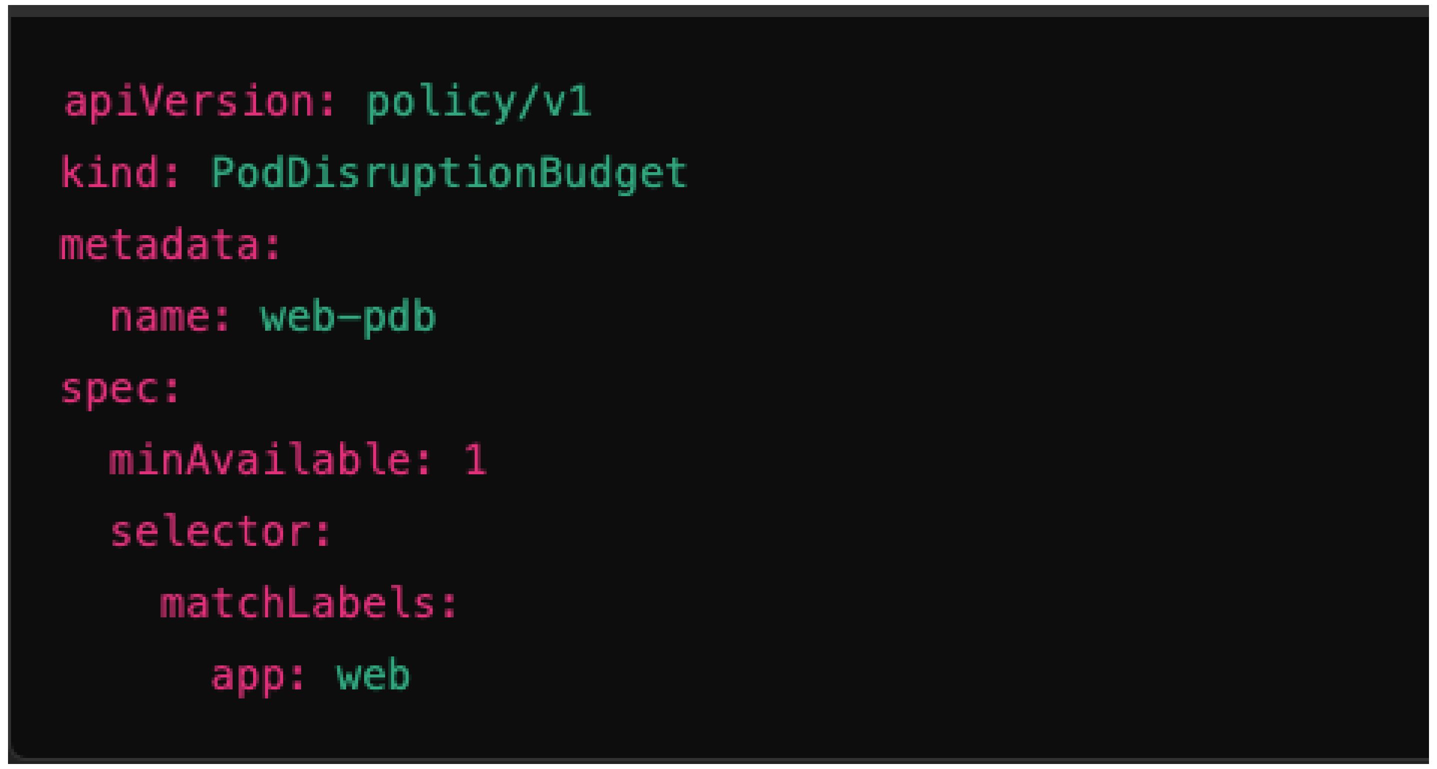
QoS Class	Description	When is it Evicted?
Guaranteed	Requests = Limits (CPU & Memory set exactly)	Last to be evicted
Burstable	Requests < Limits (Some guaranteed resources)	Evicted after BestEffort pods
BestEffort	No CPU/Memory requests or limits	First to be evicted

Resource Pressure & Node Conditions

Condition	Cause	Effect
MemoryPressure	High memory usage	Evicts lowest-priority pods
DiskPressure	Low disk space	Evicts high ephemeral storage pods
PIDPressure	Too many processes	Kills new low-priority processes
OutOfDisk	Node cannot create new pods	Stops scheduling

Pod Eviction - Best Practices

- Define Resource Requests & Limits
- Prevents one pod from consuming all resources.
- Use Pod Priority & Preemption
- Protect critical workloads from eviction.
- Use Persistent Storage
- Reduces risk of DiskPressure evictions.
- Monitor Nodes for Resource Usage
- Use Prometheus, Kubernetes Metrics Server, or kubectl top nodes.
- Use PodDisruptionBudgets (PDBs) for Stateful Apps



```yaml
apiVersion: policy/v1
kind: PodDisruptionBudget
metadata:
  name: web-pdb
spec:
  minAvailable: 1
  selector:
    matchLabels:
      app: web
```

Helm

- Helm is the package manager for Kubernetes.
- It simplifies the deployment, management, and upgrading of complex Kubernetes applications.
- Helm uses “charts” which are packages of pre-configured Kubernetes resources.

Helm Chart Structure

A typical Helm chart includes:

- Chart.yaml: Metadata about the chart.
- values.yaml: Default configuration values.
- templates/ directory: Contains Kubernetes manifest templates using Go templating.
- charts/ directory (optional): For chart dependencies.
- README.md: Documentation for the chart.