Containers & OpenShift Workshop

By Kelvin Lai

Table of Contents

1. Introduction to Containers

1.1 Definition of containerization

Containerization is a modern approach to virtualization that enables applications to be packaged with all their dependencies and configurations into isolated units known as containers. These containers can consistently run across different environments, from development on a developer's laptop to testing and production. This ensures seamless application deployment and management.

1.2 Basic Concept of Containerization

Isolation:

Each container operates independently, with its own filesystem, networking, and process space, preventing interference with other containers or the host system.

Efficiency:

Sharing the host OS kernel allows containers to be more lightweight and resource-efficient compared to traditional VMs, resulting in faster startup times and lower resource consumption.

Portability:

Containers encapsulate all necessary components of an application, ensuring consistent performance across various environments without concern for underlying infrastructure.

Scalability:

Containers can be easily replicated, scaled, and managed across a cluster of machines, supporting horizontal scaling.

Consistency:

Packaging applications with all dependencies eliminates discrepancies across different stages of development, testing, and production.

Microservices Architecture:

Containers support microservices architecture by enabling the development, deployment, and scaling of small, independent services.

Automation:

Tools like Docker, Kubernetes, and OpenShift automate the deployment and management of containerized applications, enhancing efficiency and reliability.

1.3 Containers vs. virtual machines (VMs)

Containers and VMs are both virtualization technologies, but they have significant differences in architecture, performance, and use cases.

Architecture Comparison:

- VMs: Each VM includes a full operating system instance, a hypervisor, and virtualized hardware, providing strong isolation and the ability to run different OS types on a single physical machine.

- Containers: Containers share the host OS kernel, use fewer resources, and run as isolated processes. They are more lightweight and efficient compared to VMs.

Performance:

- VMs: Provide stronger isolation and security at the cost of higher resource consumption and longer boot times.

- Containers: Lower overhead, faster boot times, and better resource utilization due to sharing the host OS kernel.

Use Cases:

1. VMs: Suitable for legacy applications and environments requiring strong isolation and compatibility with different operating systems.

2. Containers: Ideal for microservices, CI/CD pipelines, and cloud-native applications where rapid deployment, scalability, and efficiency are crucial.

The following diagram shows the differences between a container and virtual machine.

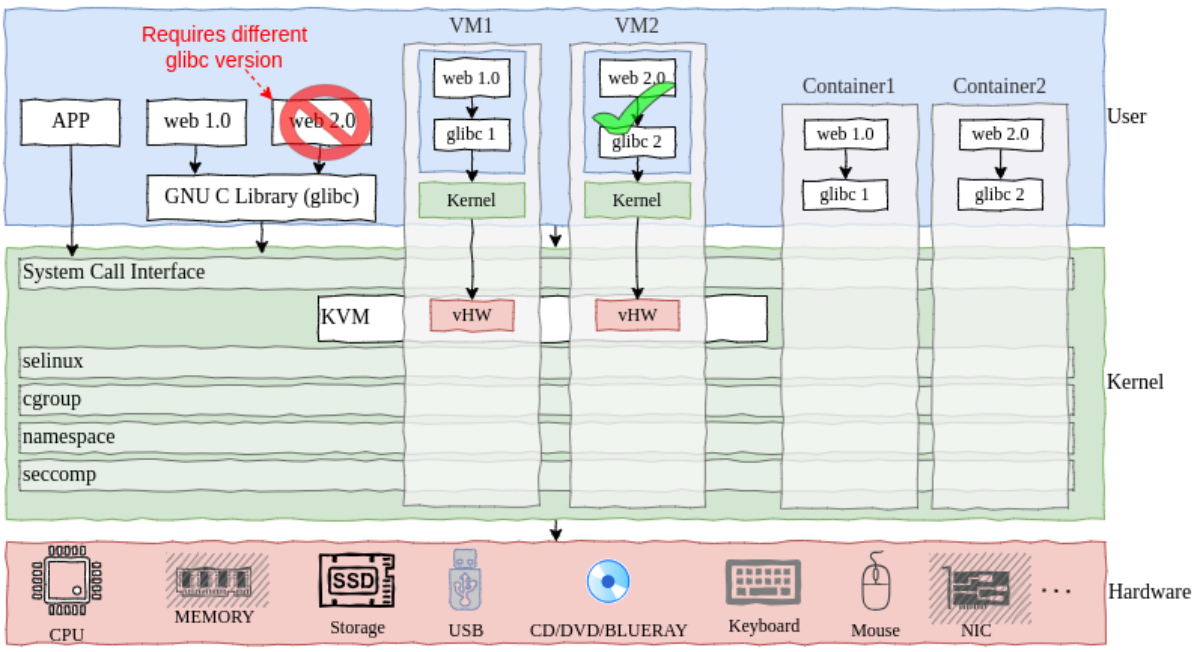

Diagram: Native vs VM vs Container

1.4 Container Architecture

The 5 components of the container architecture are:

Host OS: The operating system on which containers run.

Container Engine: Software that manages containers (e.g., Docker, Podman).

Images: Read-only templates used to create containers; contain the application and its dependencies.

Containers: Running instances of images that include application code, runtime, libraries, and dependencies.

Registry: Repository for storing and distributing container images (e.g., Docker Hub, Red Hat Quay).

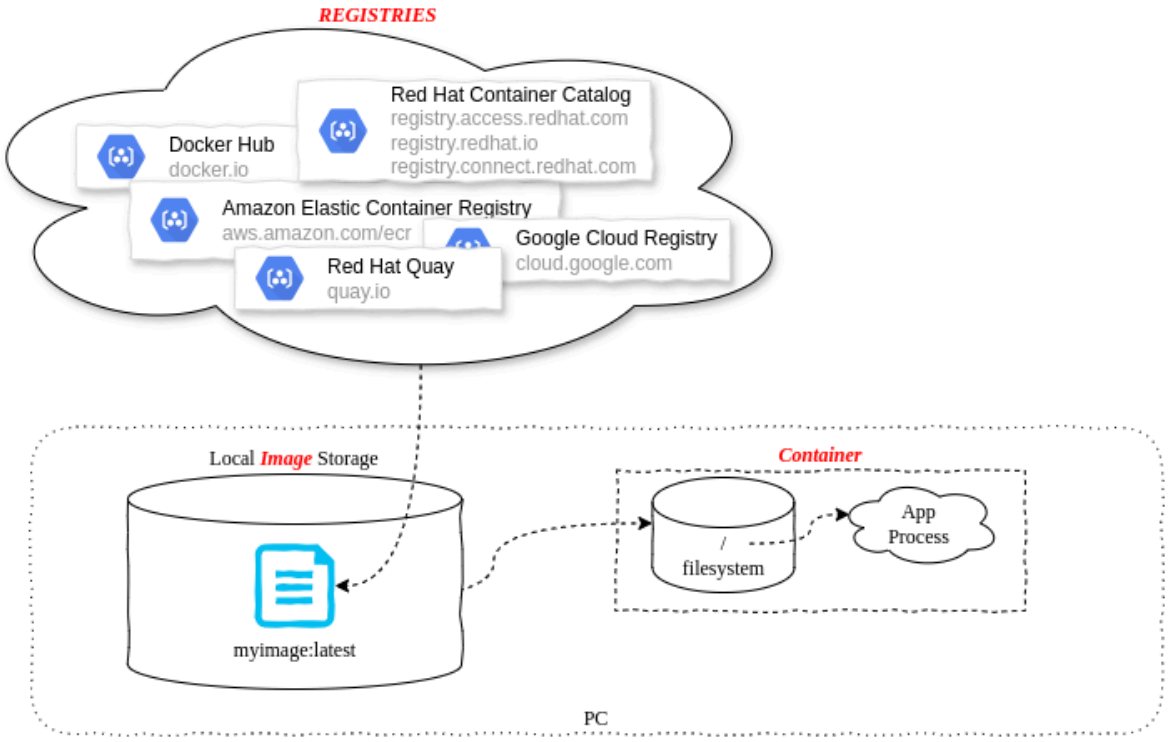

Diagram: Container Architecture

1.5 Comparison of Container Utilities

Container utilities are tools and platforms that help manage the lifecycle of containers, including creation, deployment, orchestration, and monitoring. Below is a comparison of some of the most popular container utilities:

1. *LXC/LXD*
2. *Docker*
3. *Podman*
4. *CRI-O*

Feature	Docker	Podman	CRI-O	LXC/LXD
Daemonless	No	Yes	Yes	Yes
Rootless	Limited	Yes	Yes	Yes
Primary Use Case	General-purpose containerization	General-purpose containerization	Kubernetes container runtime	System containers
Orchestration	Docker Swarm, Kubernetes	Kubernetes, OpenShift	Kubernetes	Custom orchestration, manual
Compatibility	Docker ecosystem	Docker CLI compatible	Kubernetes CRI compatible	LXC containers
Security	Moderate	High	High	High

Types of Containers:

System Containers: System containers, also known as *OS containers*, are designed to run entire operating systems or system-level services. They provide a more comprehensive and isolated environment, similar to a lightweight VM, and are capable of running multiple processes.

Application Containers: Application containers are designed to package and run a single application and its dependencies in an isolated environment. These containers focus on running individual services or applications, making them ideal for microservices architecture and modern cloud-native applications.

2. Podman Basics:

2.1 Understanding the lifecycle of a container

Understanding the lifecycle of a container is crucial for managing and deploying containerized applications efficiently.

Lifecycle Stages:

Stage	Description
Creation	A container is created from an image but not yet started. Command: `podman create`
Start	The container is started and begins executing its main process. Command: `podman start`
Running	The container is actively running and executing its main process. Command: `podman ps` (to list running containers)
Paused	The container's state is paused, halting all processes without terminating the container. Command: `podman pause`
Stopped	The container is stopped, terminating its main process but retaining its state. Command: `podman stop`
Removed	The container is removed, deleting its state and associated resources. Command: `podman rm`

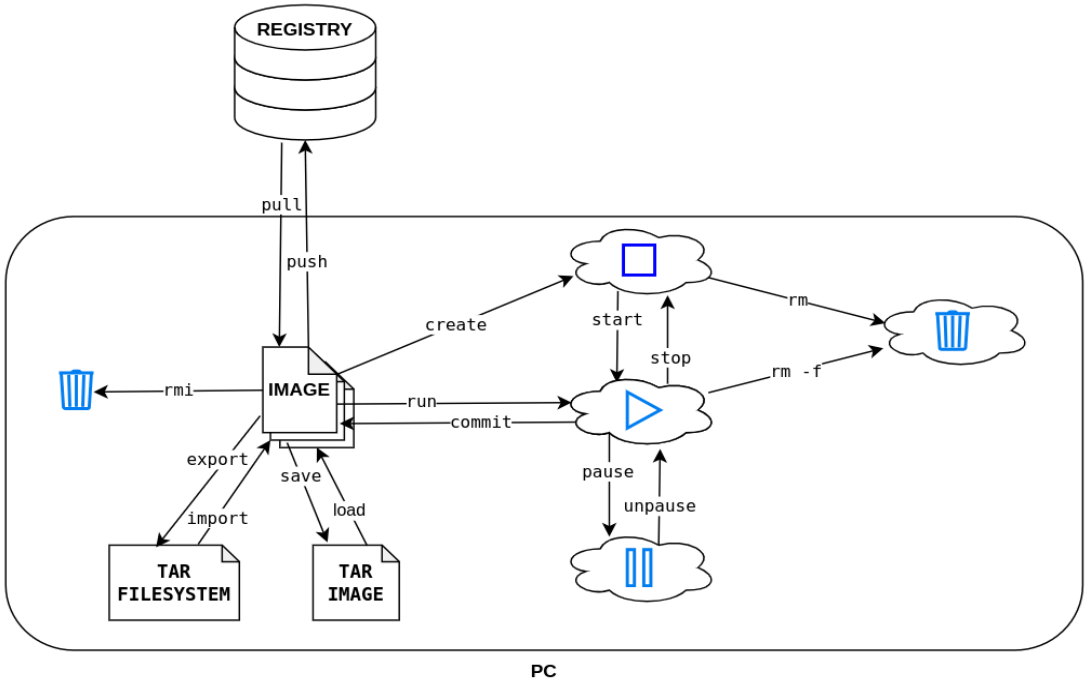

Diagram: Basic Container Lifecycle

Common Operations:

Operation	Description
Searching Registry	Searching registries for image Command: `podman search`
Listing Images	List images in the local storage Command: `podman images`
Inspecting Image/Container	Inspecting the image/container metadata Command: `podman inspect`
Checking Logs	Listing logs from container processes Command: `podman logs`
Executing Command in Container	Run command inside an existing running container Command: `podman exec`

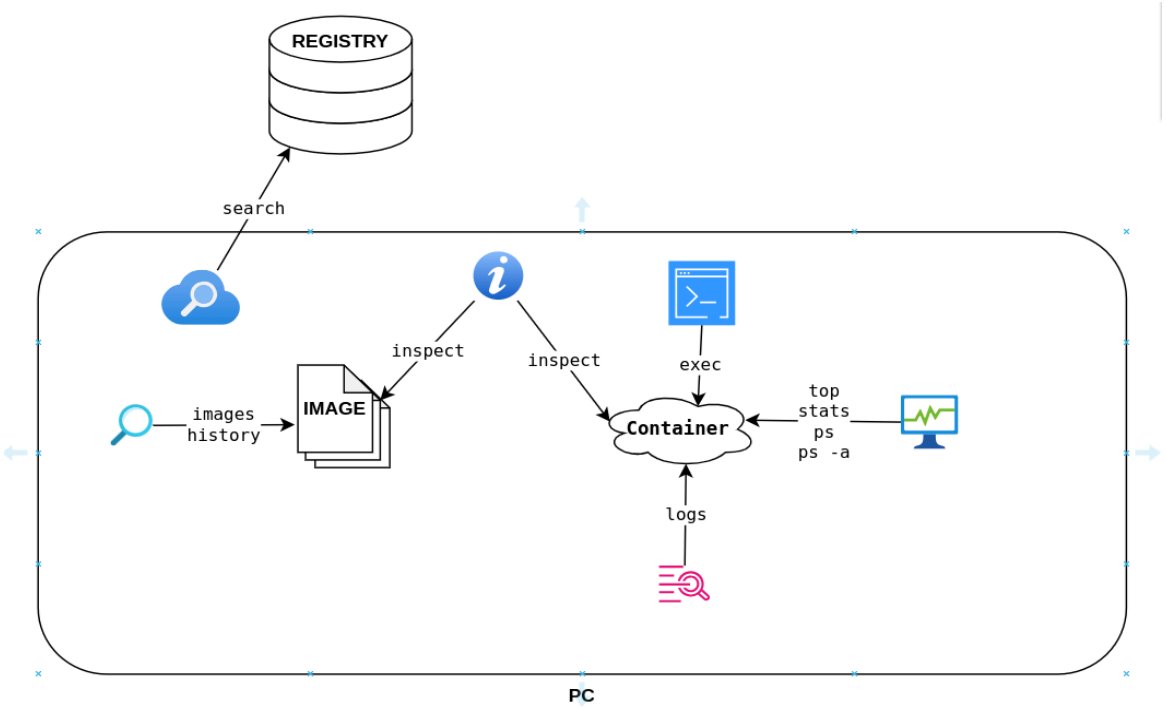

Diagram: Operations

2.2 Understanding Image and Registry

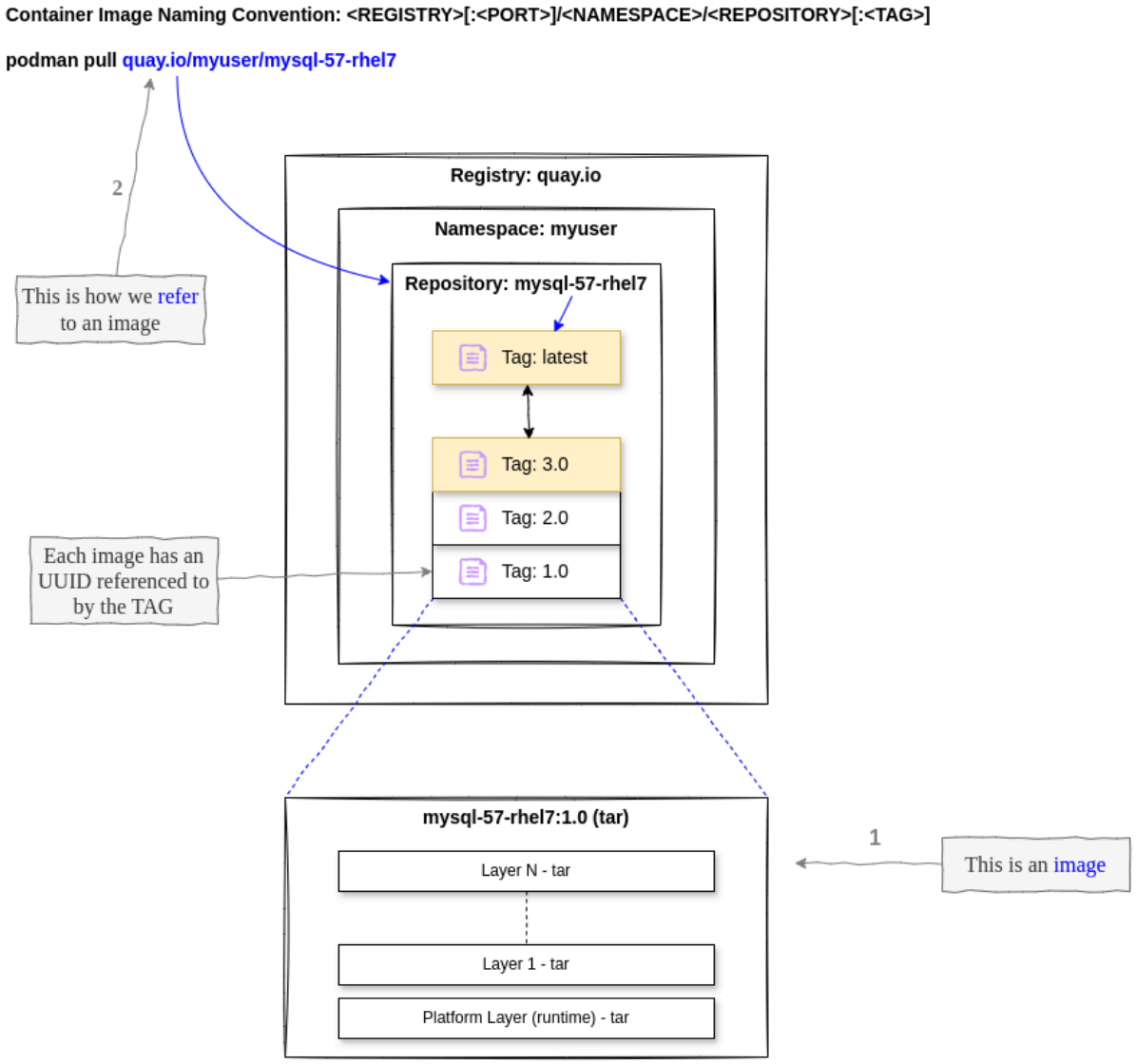

Diagram: Image and Registry

2.3 Understanding Image and Containers

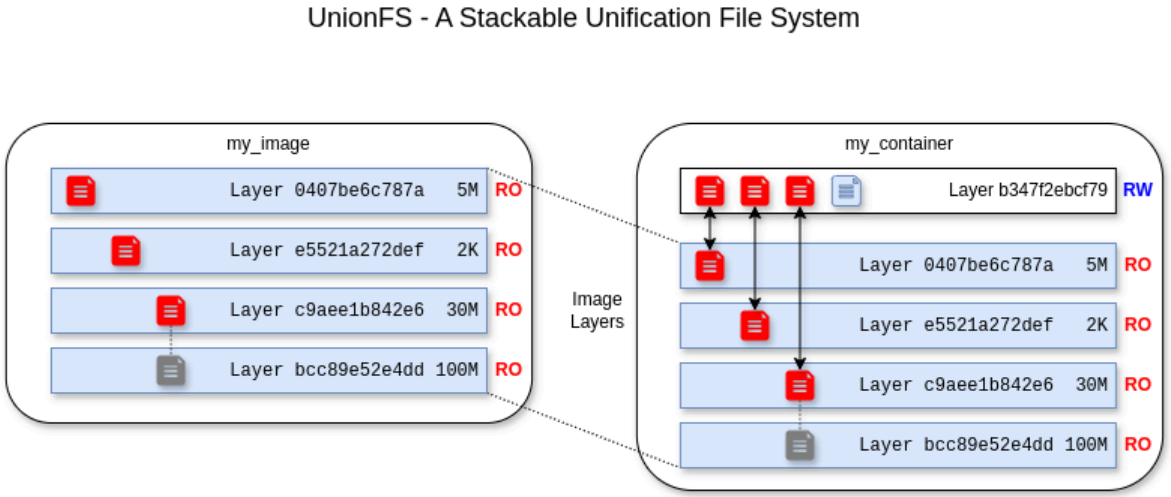

Diagram: Image and Container Relationship

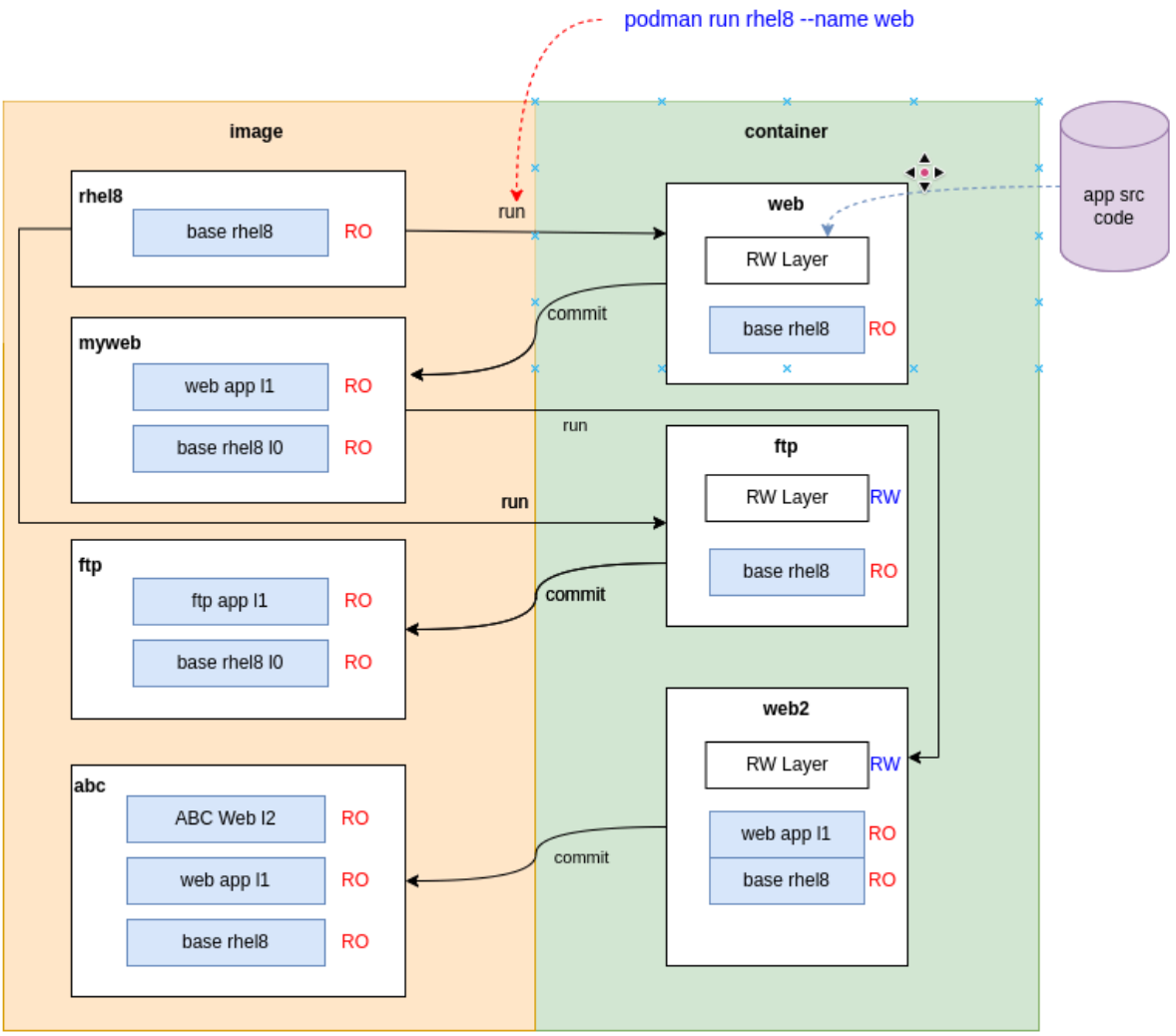

Diagram: Simple Image Creation

2.3.1 LAB: Managing Containers

In this lab session, we will be using podman command to perform the following operations:

- Searching and pulling images
- Understanding image tags and versioning
- Creating and managing containers
- Creating images from containers
- Port and volume mapping

Refer to *Containers Hands-On Examples.pdf* file
Refer to *podman.pdf* for podman syntax reference.

2.4 Building Images using Containerfile

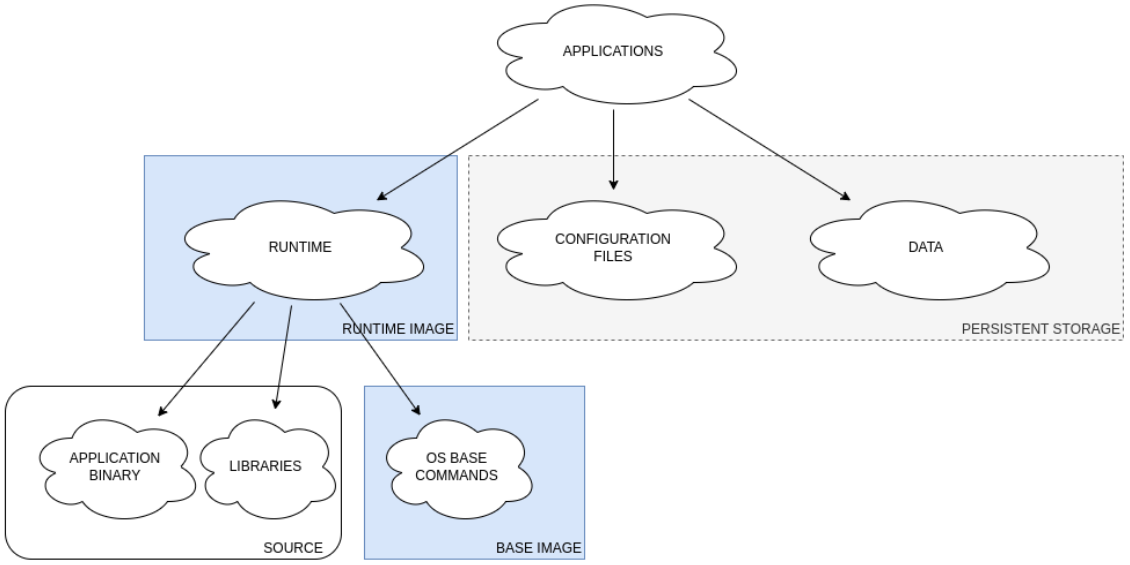

Diagram: Basic Container Design

2.4.1 Understanding Containerfile

A `Containerfile` (also known as a `Dockerfile` when used with Docker) is a script containing a series of instructions on how to build a container image. It automates the process of creating container images by specifying the base image, the software to be included, configuration files, and commands to run within the container.

Key Concepts of a Containerfile

Base Image: The starting point for your container image. It can be a minimal OS image or a pre-configured environment.

```
FROM ubuntu:20.04
```

Maintainer: Specifies the author or maintainer of the image.

```
MAINTAINER "you@example.com"
```

Run Commands: Executes commands in the shell within the container. Often used to install software packages.

```
RUN apt-get update && apt-get install -y nginx
```

Copy/Add Files: Copies files from the host machine to the container.

```
COPY index.html /usr/share/nginx/html/index.html
ADD http://www.example.com/myfile.sh /usr/local/bin/myfile.sh
```

Environment Variables: Sets environment variables which will be used inside the container.

```
ENV APP_ADMIN_USERNAME=einstein
```

Expose Ports: Indicates the ports on which the container listens for network traffic.

```
EXPOSE 80
```

Labels: Used to add metadata to an image. This metadata can include information about the image, such as the maintainer, version, description, license, and other custom information.

```
LABEL USERNAME=albert PASSWORD=einstein MSG=”Hello World”
```

Entry Point/Command: Specifies the command to run within the container when it starts.

```
ENTRYPOINT ["nginx", "-g", "daemon off;"]
CMD ["nginx", "-g", "daemon off;"]
```

Work Directory: Sets the working directory for subsequent instructions.

```
WORKDIR /app
```

2.4.2 LAB: Create an application container image.

Refer to *Containers Hands-On Examples.pdf* file

2.5 Upload Images to registry

Uploading an image to a registry typically refers to the process of transferring a container image from your local environment to a remote container registry, like Docker Hub, AWS ECR (Elastic Container Registry), Google Container Registry, quay.io, OpenShift Internal Registry or a private registry.

Steps to upload an image to the registry:

1. Tag the image
2. Login to the registry
3. Push the image to the registry
4. Verify the upload

2.5.1 LAB: Upload image to registry

Refer to *Containers Hands-On Examples.pdf* file

2.6 Summary

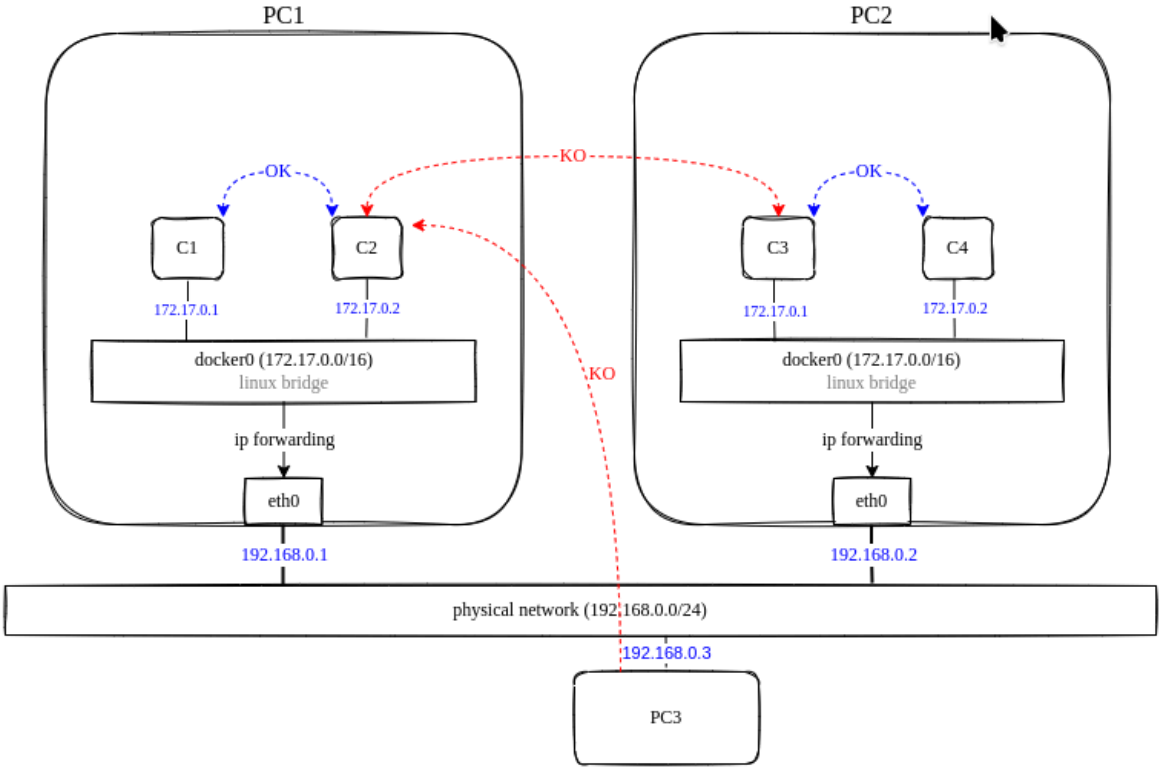

Diagram: Containers Network

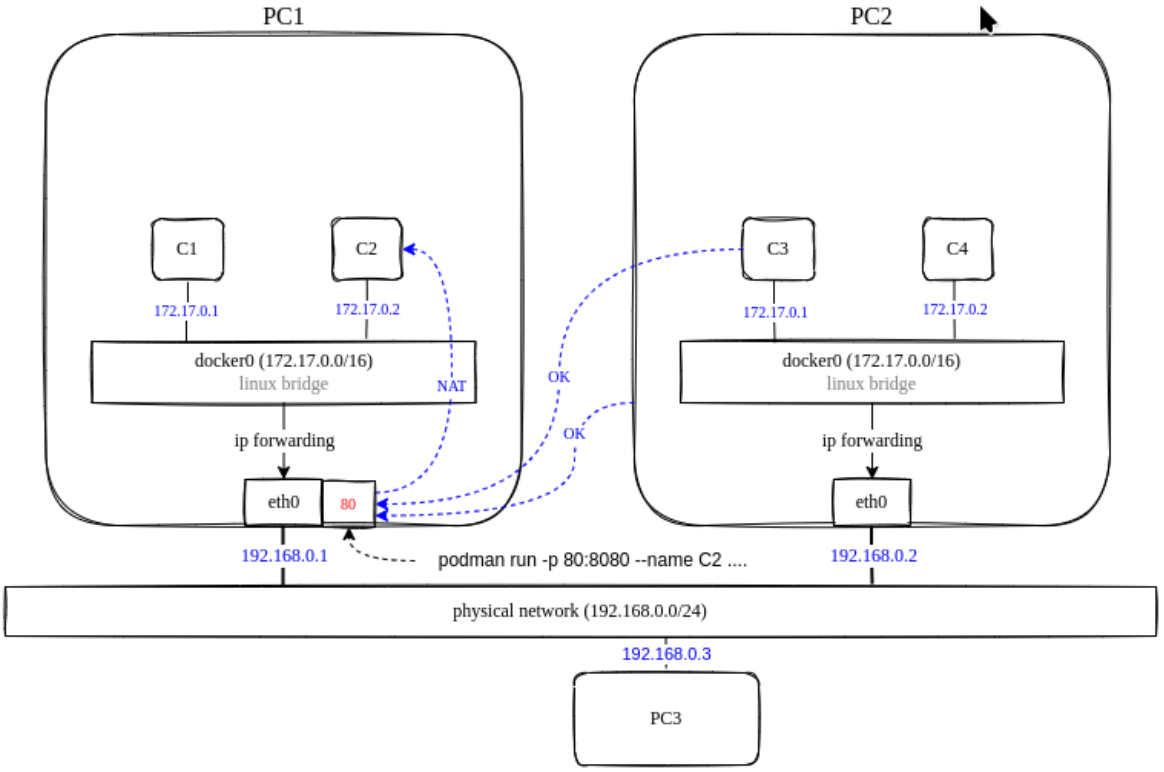

Diagram: Port Forwarding

3. Introduction to OpenShift

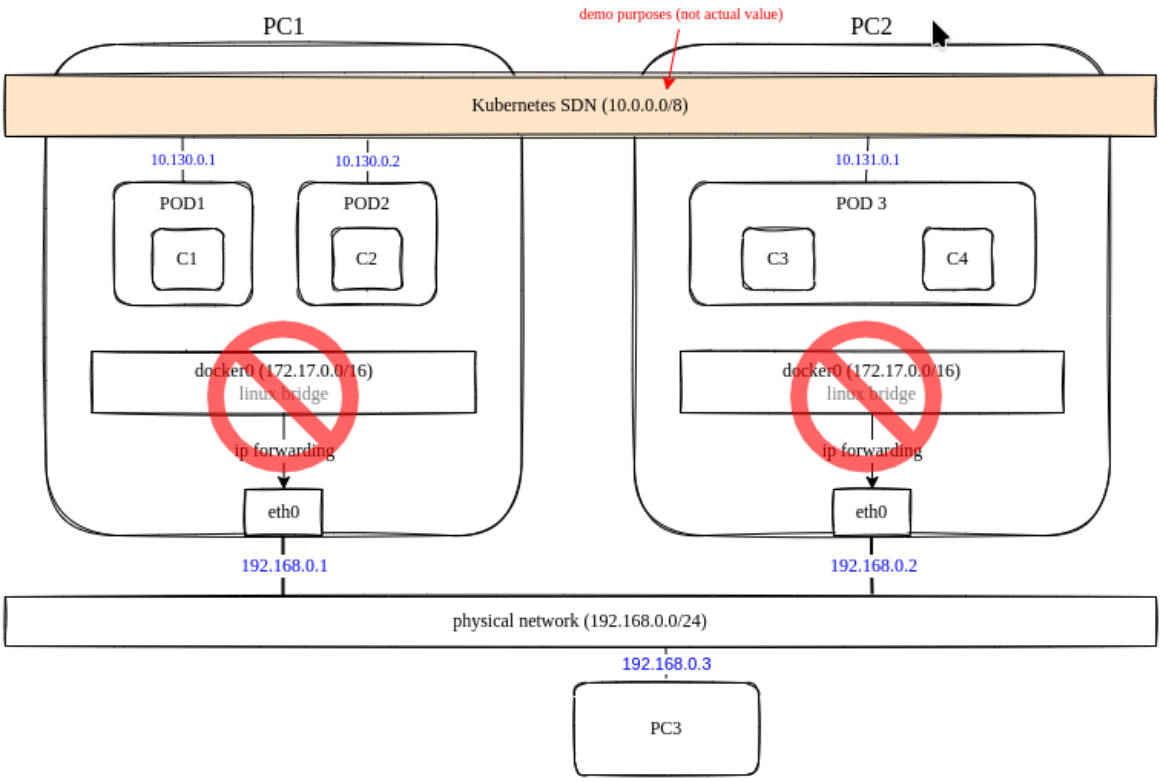

Diagram: Kubernetes Network

3.1 Components of OpenShift

OpenShift extends Kubernetes by adding developer and operational tools, which simplifies building, deploying, and managing applications.

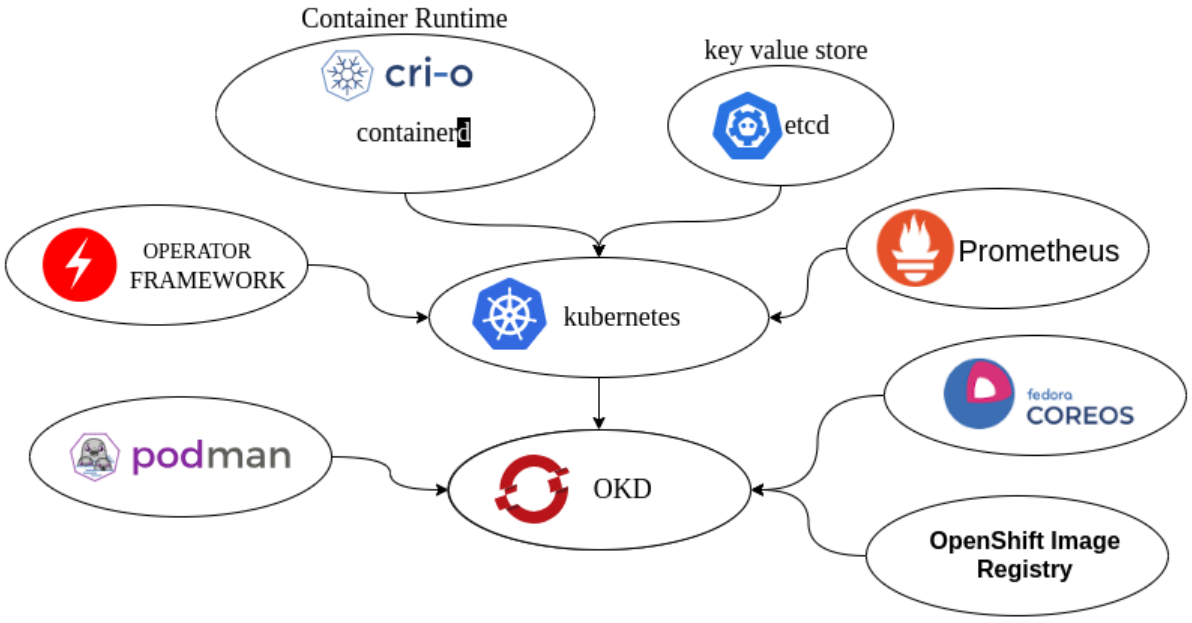

Diagram: Kubernetes and Openshift

Kubernetes Kubernetes, often abbreviated as K8s, is an open-source container orchestration platform designed to automate the deployment, scaling, and management of containerized applications. Developed by Google and now maintained by the Cloud Native Computing Foundation (CNCF), Kubernetes provides a robust framework for running distributed systems resiliently.

OpenShift OpenShift is an open-source container application platform built on top of Kubernetes, developed by Red Hat. It extends Kubernetes, adding features and tools to provide a more integrated and enhanced experience for developers and operations teams. This includes extended networking capabilities (OpenShift SDN), image building and management (Source-to-Image), routing (OpenShift Routes), and a web console for easier management.

Core Components:

Master Node Components:

- **API Server**: Central management point for the OpenShift cluster, handling RESTful API requests.
- **Controller Manager**: Manages controllers that regulate the state of the cluster, such as node and replication management.
- **Scheduler**: Allocates workloads to nodes based on resource availability and policies.
- **etcd**: Stores cluster configuration and state data in a distributed key-value store.

Worker Node Components:

- **Kubelet**: Agent that ensures containers are running in a Pod.
- **Kube Proxy**: Handles network proxying and load balancing for services.
- **Container Runtime**: Software responsible for running containers, such as containerd or CRI-O.

OpenShift Specific Components:

- **OpenShift Image Registry:** Internal registry used to store dynamically created images by S2I.
- **OpenShift Router:** Manages routing of external traffic to services within the cluster.
- **OpenShift OAuth Server:** Handles authentication.
- **OpenShift Web Console:** Provides a graphical interface for managing the cluster and applications.

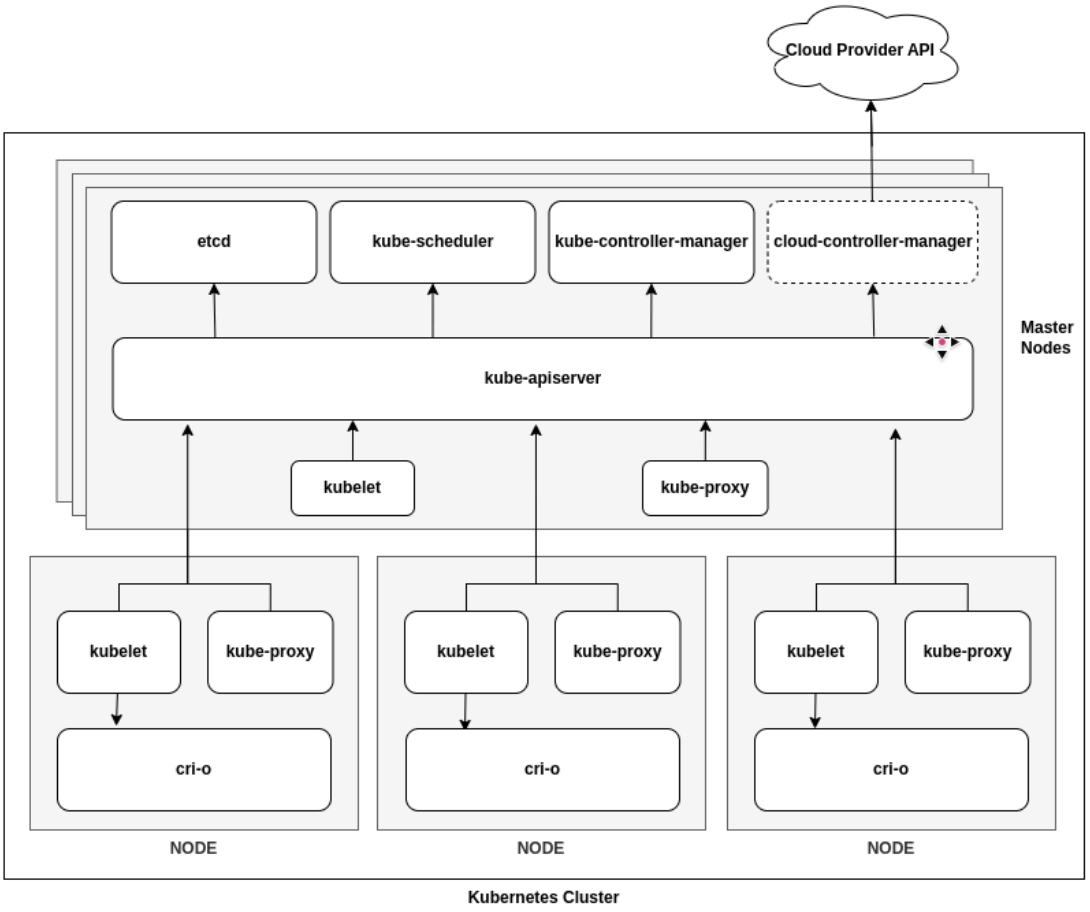

Diagram: Kubernetes Cluster

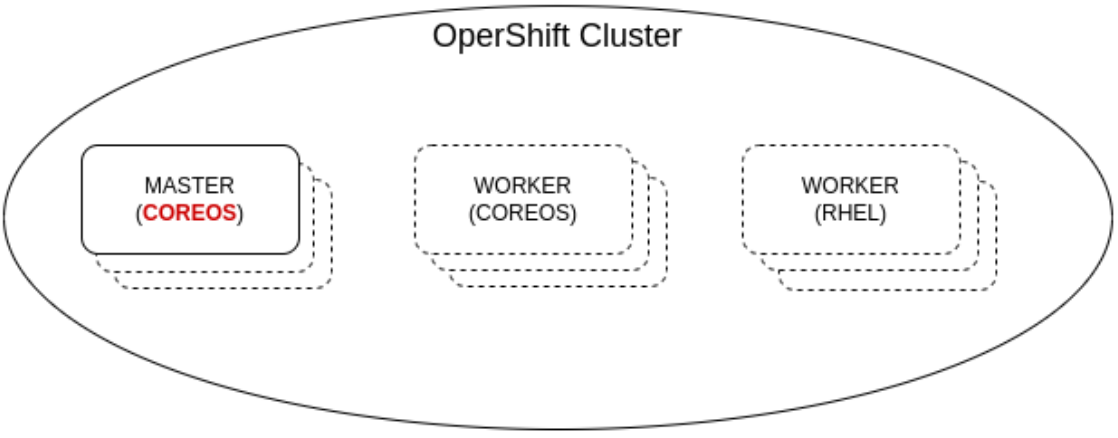

Diagram: OpenShift Setup

3.3 Resource Types

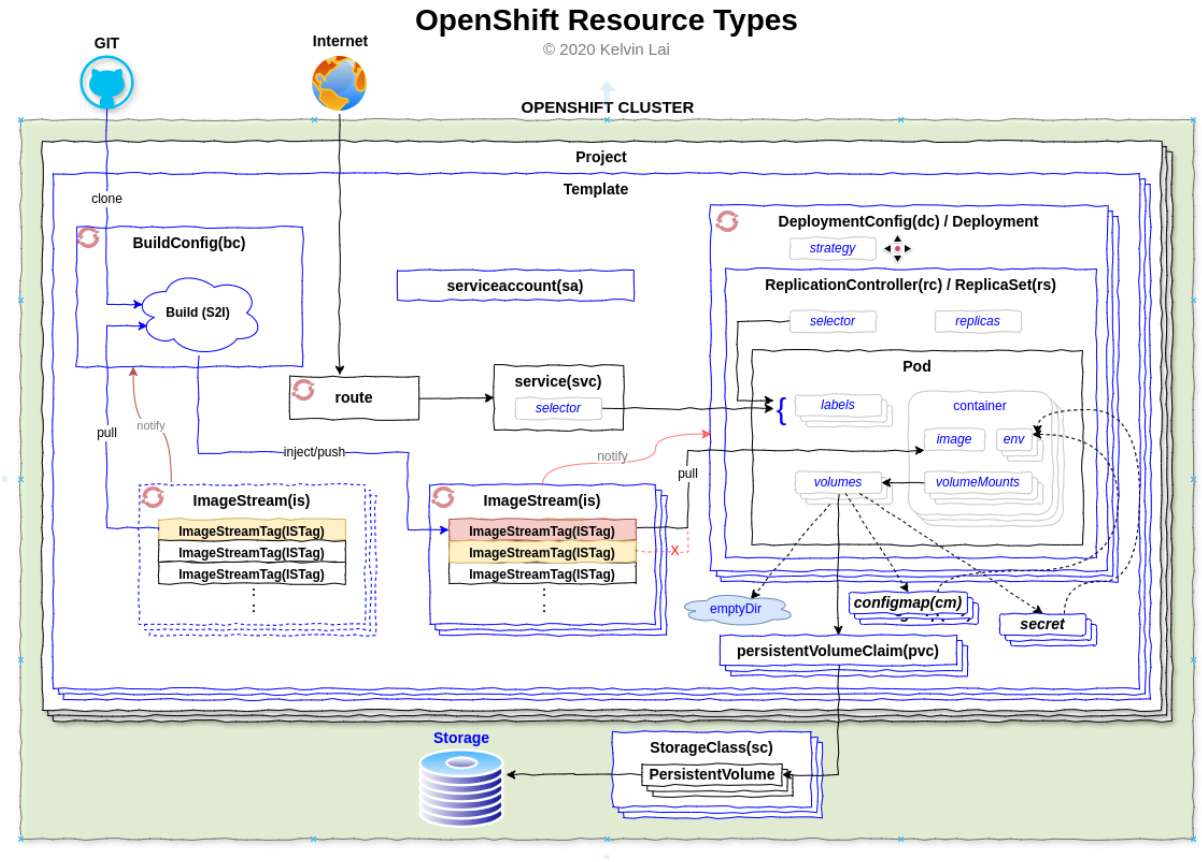

Diagram: OpenShift Resource Types

Resource	Explanation
Project	A namespace in OpenShift, used to isolate and manage a set of resources and users.
Pod	The smallest and simplest Kubernetes object. It represents a single instance of a running process in a cluster and can contain one or more containers.
Service	An abstraction that defines a logical set of Pods and a policy by which to access them, typically through a stable IP address and DNS name.
Route	An OpenShift-specific resource that exposes a Service outside the cluster, typically by mapping it to an external hostname.
ImageStream	An OpenShift resource that tracks changes to container images in a registry, allowing for easier image management and updates.
ReplicationController (rc)	Ensures a specified number of Pod replicas are running at any given time.
ReplicaSet (rs)	The next-generation ReplicationController that supports the same functionalities but with more features, used primarily by Deployments.

Resource	Explanation
DeploymentConfig (dc)	An OpenShift-specific resource that defines the desired state of an application and how it should be deployed and managed.
Deployment (deploy)	A Kubernetes resource that provides declarative updates to Pods and ReplicaSets, managing the rollout and rollback of updates.
BuildConfig (bc)	An OpenShift resource that defines how to build and produce new container images from source code.
Build	The process or instance created by a BuildConfig to produce a container image from source code.
Secret	A Kubernetes object that stores sensitive information, such as passwords, OAuth tokens, and SSH keys, securely.
ConfigMap (cm)	A Kubernetes object that stores non-sensitive configuration data, such as configuration files, environment variables, and command-line arguments.
PersistentVolumeClaim (pvc)	A request for storage by a user in Kubernetes, specifying the desired size and access mode.
PersistentVolume (pv)	A piece of storage in the cluster that has been provisioned by an administrator or dynamically provisioned using a StorageClass.
StorageClass (sc)	A Kubernetes resource that provides a way to describe the "classes" of storage available, enabling dynamic provisioning of PersistentVolumes.

3.4 Operators

Operators in Kubernetes, including OpenShift, are designed to automate the management of complex applications and resources. They leverage Kubernetes' declarative API to handle application-specific logic, such as installation, scaling, backup, and failover. There are several types of operators, each serving different purposes based on their complexity and functionality.

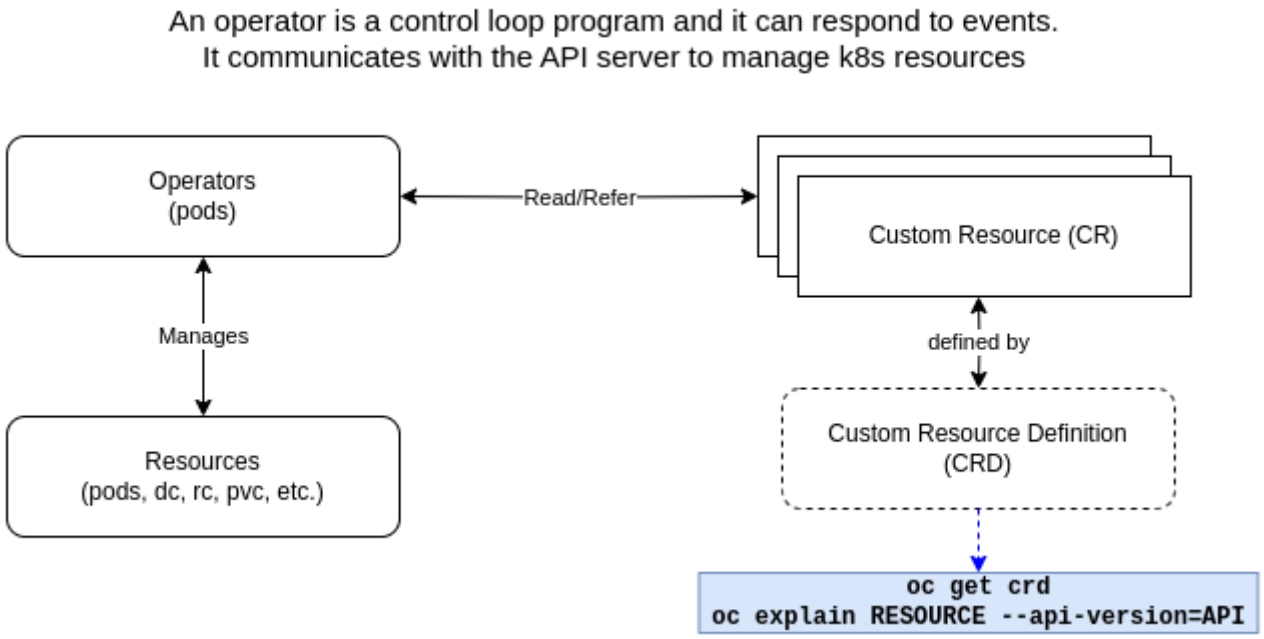

Diagram: Operator

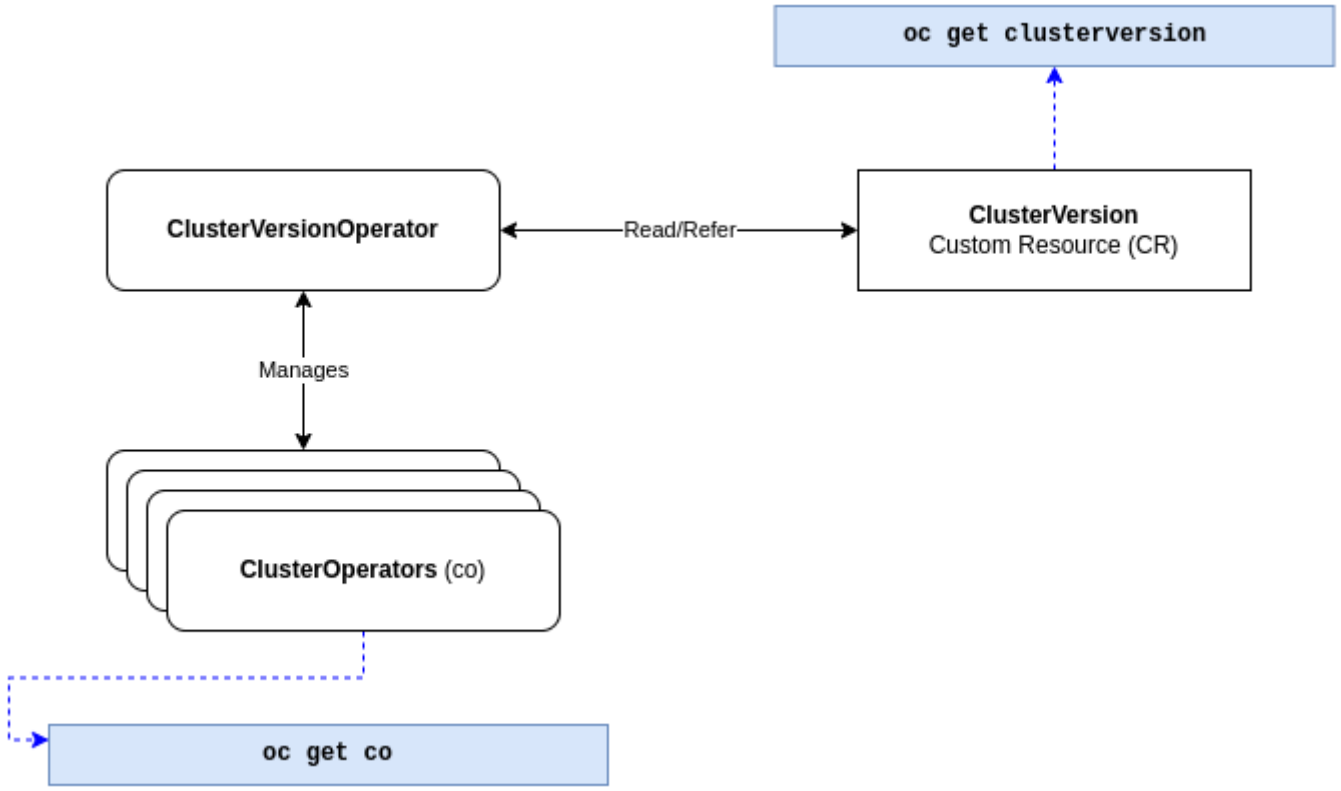

Diagram: Cluster Operator

3.4.1 Application-Specific Operators

Operator Type	Description	Use Case	Example
Basic Operators	Manage simple Kubernetes resources like Deployments and Services.	Automating deployment and configuration of stateless applications.	Basic web app operator
Lifecycle Management Operators	Handle the complete lifecycle of an application (install, configure, update, uninstall).	Managing complex applications like databases and middleware.	PostgreSQL operator
Application-Specific Operators	Tailored to manage a specific application, embedding operational knowledge and best practices.	Managing bespoke or third-party applications with specific needs.	Elasticsearch operator
Framework Operators	Designed to manage a class of applications, providing reusable patterns and scaffolding.	Simplifying the creation of new operators by providing common functionalities.	Operator SDK
Monitoring and Observability Operators	Manage monitoring and observability tools within a cluster.	Setting up and managing cluster-wide monitoring and alerting systems.	Prometheus Operator
Security Operators	Manage security-related tasks and configurations within a cluster.	Automating security policies and ensuring compliance with standards.	Cert-Manager Operator
Storage Operators	Manage storage systems and resources within a cluster.	Managing stateful applications requiring reliable and scalable storage solutions.	Rook Operator

3.4.2 Cluster Operators

Cluster Operator	Description	Key Functions
Cluster Version Operator (CVO)	Manages the lifecycle of the entire OpenShift cluster, including upgrades and configuration changes.	Ensures the cluster is running the desired version of OpenShift components.
Machine Config Operator (MCO)	Manages the configuration of the cluster's nodes, including OS updates and settings.	Ensures nodes adhere to desired configuration specifications.
Kubelet Config Operator	Manages the configuration of kubelets across the cluster.	Ensures kubelets run with desired settings and updates.
Network Operator	Manages the cluster's network components, such as CNI plugins and DNS.	Ensures consistent network configurations and handles network-related updates.
Etcd Operator	Manages the etcd cluster, ensuring high availability and data recovery.	Ensures backup, recovery, and high availability of the etcd cluster.

3.5 OpenShift CLI

The OpenShift CLI (Command Line Interface), often abbreviated as oc, is a powerful tool for interacting with OpenShift clusters. It allows administrators and developers to manage applications, services, deployments, and various resources within an OpenShift environment.

WARNING: DO NOT USE OPENSHIFT CLI ON A SHARED USER ACCOUNT!!!

Usage:

```
oc COMMAND_TYPE
```

Where,

COMMAND_TYPE	BASIC, BUILD/DEPLOY, MANAGEMENT, TROUBLESHOOTING, SETTINGS, ADVANCED, OTHERS.
BASIC	login, new-project, new-app, status, project, projects, explain
BUILD/DEPLOY	new-build, start-build, cancel-build, rollout, rollback, cancel-build, import-image, tag
MANAGEMENT	create, apply, get, describe, edit, set, label, annotate, expose, delete, scale, autoscale, secrets, serviceaccounts
TROUBLESHOOTING	logs, rsh, rsync, port-forward, debug, exec, proxy, attach, run, cp, wait
SETTINGS	logout, config, whoami, completion
ADVANCED	adm, replace, patch, process, extract, observe, policy, auth, image, registry, idle, api-versions, api-resources, cluster-info, diff, kustomize
OTHERS	ex, help, plugin, version

For a description of all the subcommand listed above, type `oc --help`

Best method for getting help: `oc COMMAND --help`

Configuration Files:

```
~/.kube/config
```

Most often used common syntax:

```
oc COMMAND RESOURCE [NAME] [-n PROJECT]
                    ↥
                space or /
```

Where,

COMMAND	get, delete, describe, edit, rsh, logs, port-forward, etc.
RESOURCE	node, project, pod, services(svc), route, persistentvolume(pv), persistentvolumeclaim(pvc), secret, configmap(cm), deploymentconfig(dc), deployment(deploy), replicationcontroller(rc), replicaset(rs), etc.

Basic Operations:

Login Operations:

```
oc login [-u USER] [-p PASSWORD] API_URL
oc whoami [--show-console] [--show-token]          # --show-token = -t
oc logout
```

```
oc login -u devuser https://api.mycluster.example.com:6443
```

Project Operations:

```
oc new-project PROJECT                    # Create a new project
oc project [PROJECT]                      # List/Change current project
oc projects OR oc get project             # list all projects
oc delete project PROJECT                 # Delete project PROJECT
```

Creating Application Resources:

```
oc create -f FILE                         # create resource from YAML/JSON file
```

Create App using existing Image/ImageStream

```
oc new-app [--as-deployment-config] \              # create app using IMAGE
       {[--docker-image] IMAGE} [--name NAME] \
       [-e KEY=VALUE]...
oc new-app [--as-deployment-config] \              # create app using IMAGE_STREAM
       IMAGE_STREAM
```

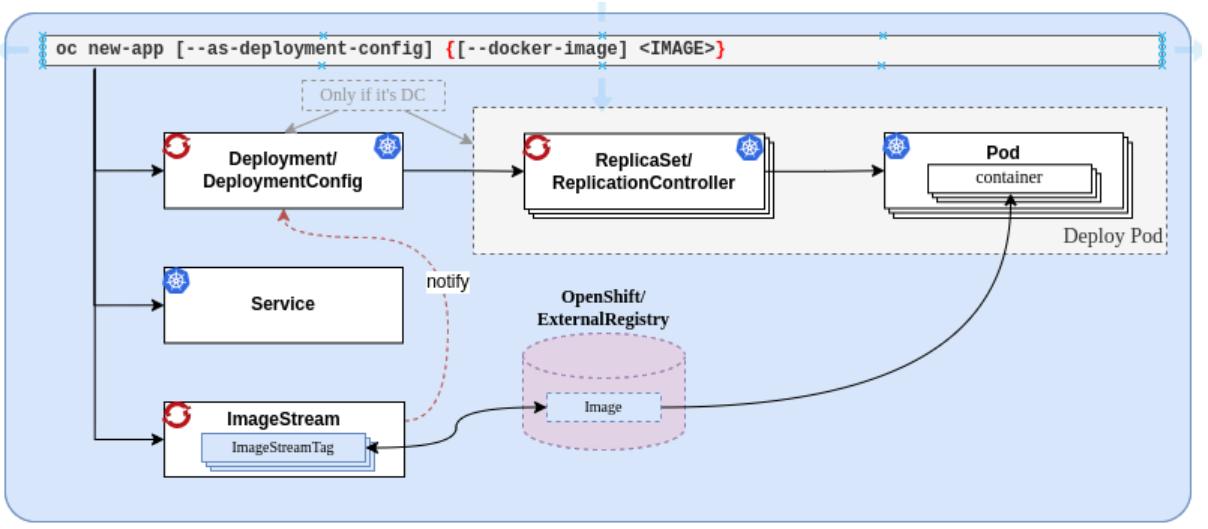

Diagram: Using existing image

Create App using Source-To-Image (S2I/STI) or Dockerfile

```
oc new-app [--as-deployment-config] \              # create app using SOURCE_CODE
       [-i BUILDER_IS] URL [--name NAME] \
       [--strategy source|docker] [-e KEY=VALUE]...
oc new-app [--as-deployment-config] \              # create app using SOURCE_CODE
       BUILDER_IS~URL [--name NAME] \              # force S2I to use BUILDER_IS
       [--strategy source|docker] [-e KEY=VALUE]...
```

```
oc new-app -i php https://github.com/user/myapp#branch --context-dir mydb
oc new-app -i php:7.1 https://github.com/user/yourapp
oc new-app php:7.1~https://github.com/user/superapp
```

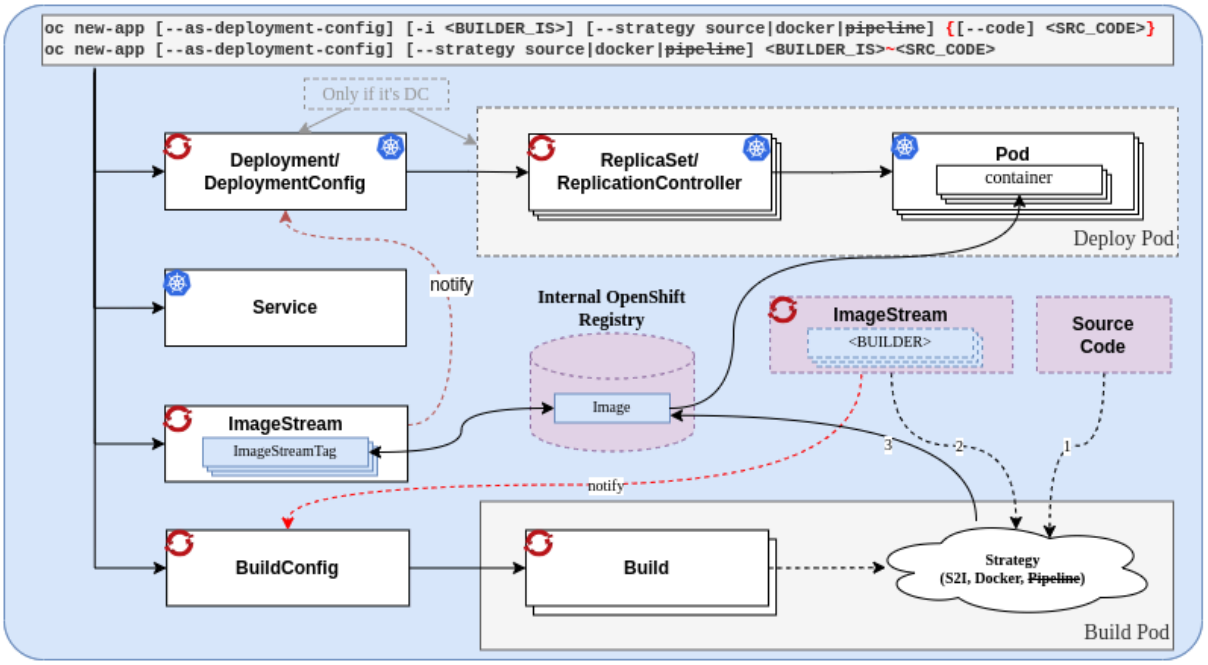

Diagram: Managed Life Cycle

Create App using Template

```
oc new-app --template TEMPLATE \                   # create app using TEMPLATE
       [-p PARAM=VALUE]... [--param-file PARAM_FILE] \
       [-e KEY=VALUE]
```

API Resources (Resource Types) Operations:

```
oc api-resources                    # List all resource types. Any namespaced resource
                                    command, accepts -n PROJECT option

oc explain RESOURCE[.FIELD]{[.FIELD]}...          # Learn resource structure
oc get RESOURCE [NAME] [-n PROJECT] [-o (json|yaml)|wide]       # Show resource info
oc describe RESOURCE NAME                         # Show more info

oc edit RESOURCE NAME [-o json]                   # Edit resource definition
oc patch ....                                     # Refer to "oc help patch"
oc set env|probe|volumes|sa|triggers|...          # refer to "oc help set RESOURCE"

oc delete RESOURCE NAME
oc delete all --all                               # delete all resource from project
oc delete all -l LABEL                            # delete all resources having LABEL
```

Scaling Pods:

```
oc scale --replicas=VALUE dc|rc NAME       # only scale rc if there's no dc !!!!
oc autoscale dc|deployment NAME \          # only works if METRICS available
       --min VALUE --max VALUE \
       --cpu-percent VALUE
oc get hpa                                 # list HorizontalPodAutoscaler
```

Image Operation:

```
oc import-image NAME [--confirm] --from IMAGE_URI [--insecure]
```

Troubleshooting:

```
oc logs bc|dc|build|pod NAME [-f]
oc get events
oc rsh POD_NAME [CMD]
oc cp FROM TO
```

where,

```
FROM = TO = [POD_NAME:]PATH
```

```
oc cp ./file1 mypod-1-ab123:/mnt/testing/fileX
oc cp mypod-1-12345:/index.html /tmp/backup.html
```

Creating Resources:

```
oc expose dc|rc|pod|svc NAME        # expose dc/rc/pod gets svc, expose svc gets a route
oc create secret generic NAME {--from-literal KEY=VALUE}...      # oc create secret -h
oc create cm NAME {--from-literal KEY=VALUE}...                  # oc create cm -h
```

```
oc expose dc/myapp
oc expose svc/myapp --name myroute --hostname myroute.apps.example.com
```

4. Deploying and maintenance of applications

4.1 Understanding relationship between pods, services and routes.

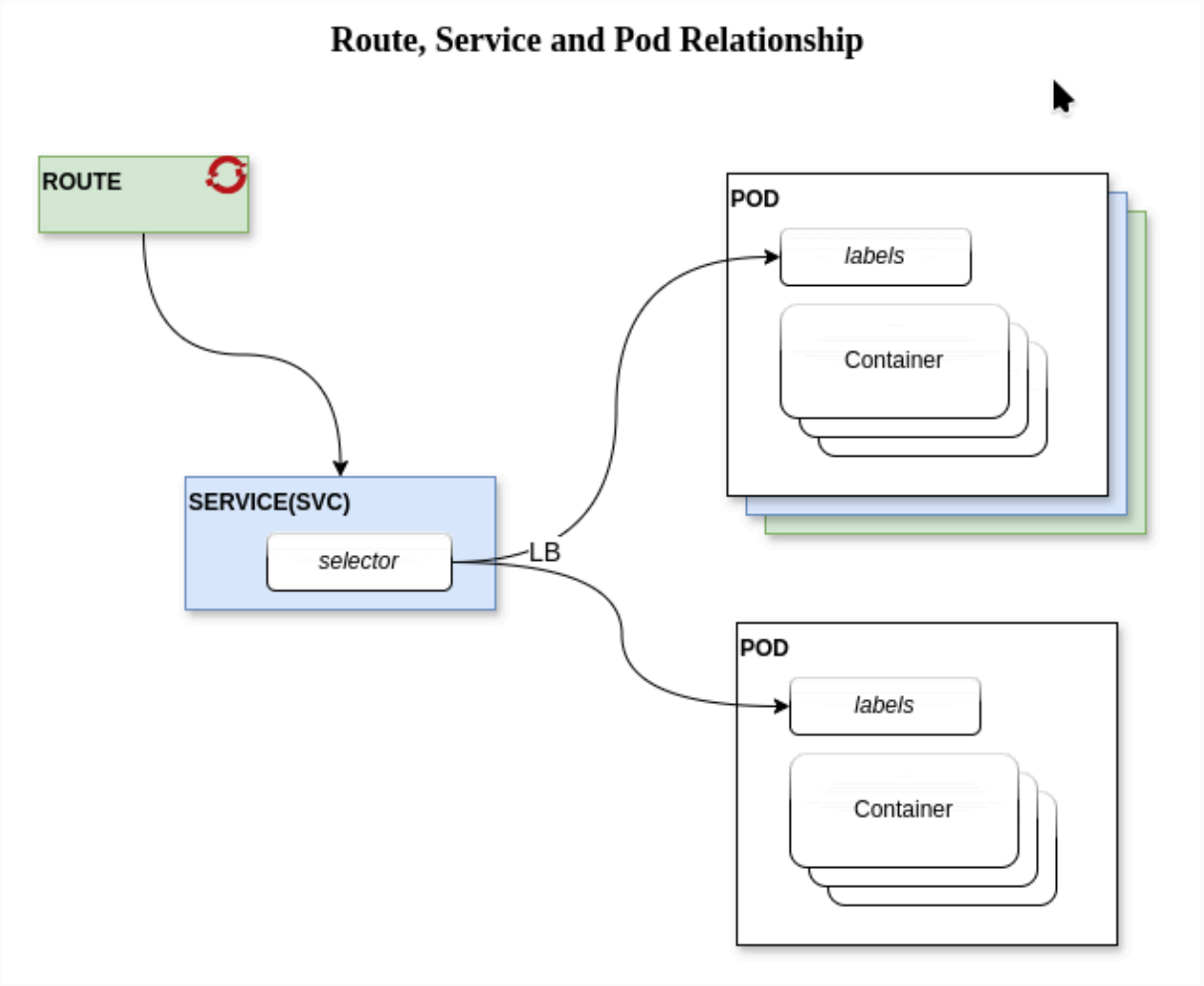

Diagram: Basic Resource

POD A pod contains one or more containers.

SERVICE A service references the pod(s) by using the label selector.
The service load balances the connections between all the pods.
To get a service, expose the dc/deploy/pod.
`oc expose deploy/myapp`

ROUTE A route exposes the service to the external world.
`oc expose svc/mysvc`

Warning: A service "can" refer to different pods, if the pods have the same label.

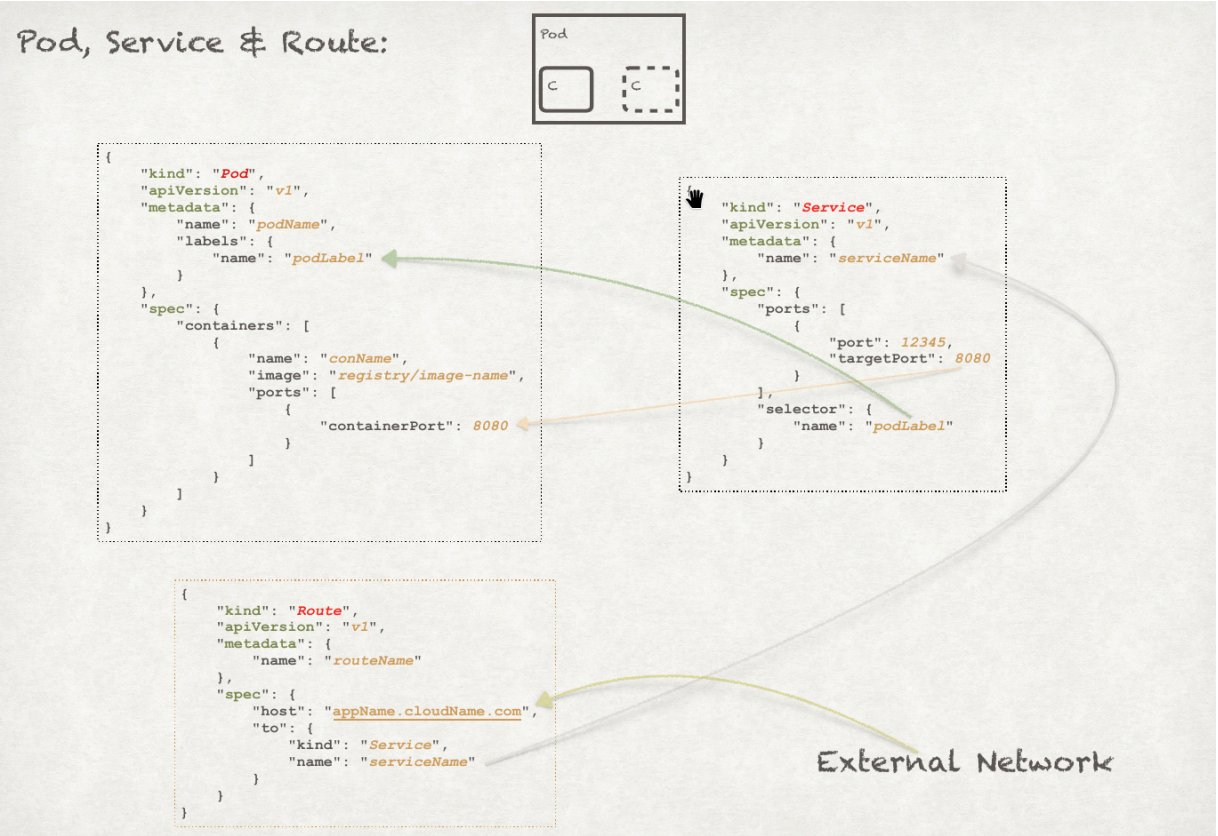

Diagram: Resource Definition

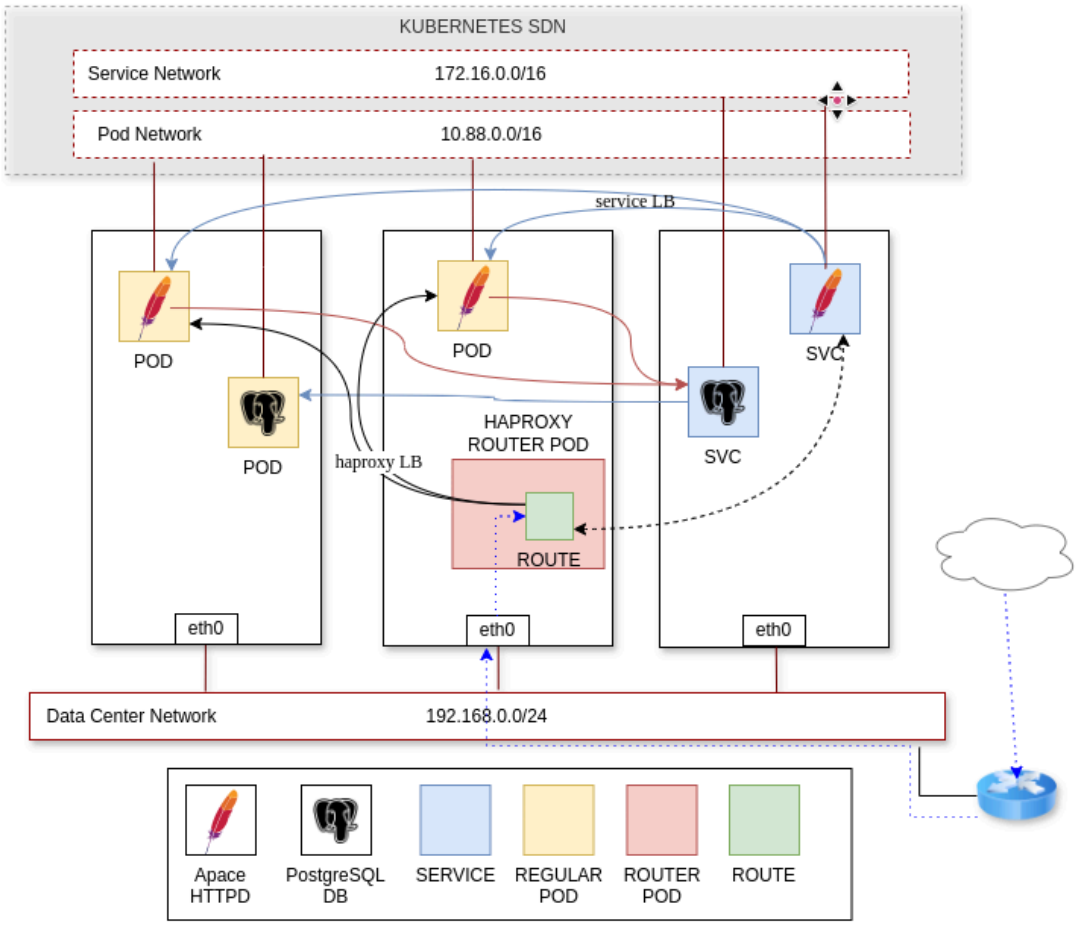

Diagram: OpenShift Network and Resources

4.2 Getting to know various deployment strategies.

OpenShift supports various deployment strategies through its deployment configurations and resource definitions. The common strategies supported by OpenShift include rolling, recreate, blue-green, canary and a/b. Below is a brief description, use cases of these strategies.

1. Rolling Update Deployment

Description:

- Gradually replaces instances of the old version with the new version without downtime.
- Ensures that at least some instances of the application are always available.

Use Cases:

- Ideal for applications requiring high availability.
- Suitable for most web applications and services that can handle traffic routed to different versions during the update.

Steps:

1. Start new pods.
2. Stop old pods incrementally.
3. Continue until all old pods are replaced.

2. Recreate Deployment

Description:

- This strategy stops the old version of the application completely before starting the new version.
- There is downtime during the transition as the old pods are terminated before the new pods are started.

Use Cases:

- Suitable for applications that cannot tolerate partial updates or where maintaining consistency is crucial.
- Simple applications where a brief downtime is acceptable.

Steps:

1. Old pods are stopped.
2. New pods are started.

3. Blue-Green Deployment

Description:

- Maintains two environments: the current production environment (blue) and the new version (green).
- Traffic is switched from the blue environment to the green environment once the new version is verified.

Use Cases:

- Critical applications where you need to minimize the risk of downtime and ensure a quick rollback.
- Scenarios requiring testing of the new version in a production-like environment before fully switching.

Steps:

1. Deploy new version in the green environment.
2. Test and verify the new version.
3. Switch traffic from blue to green.
4. Optionally, keep the blue environment running for quick rollback.

4. Canary Deployment

Description:

- Gradually shifts traffic from the old version to the new version by directing a small percentage of traffic to the new version initially.
- Monitors the new version's performance and stability before increasing traffic.

Use Cases:

- Ideal for testing new features or updates with a subset of users.
- Useful when the impact of changes needs to be evaluated in production gradually.

Steps:

1. Deploy new version.
2. Route a small percentage of traffic to the new version.
3. Monitor and evaluate performance.
4. Gradually increase traffic to the new version.

5. A/B Testing Deployment

Description:

- Deploys multiple versions of the application (A and B) simultaneously to test specific changes or features.
- Traffic is split between the versions to compare performance and user acceptance.

Use Cases:

- Useful for feature testing, UI/UX improvements, and gathering user feedback.
- Scenarios where controlled experiments are needed to evaluate the impact of changes.

Steps:

1. Deploy both versions (A and B).
2. Split traffic between A and B based on defined criteria.
3. Analyze results and choose the better-performing version.

4.3 LAB: Deploying application

4.4 LAB: Deploying multi-container applications

4.5 Understanding how scheduling works

The function of a scheduler is to ensure that Pods are placed on nodes in a way that balances the workload and meets specific scheduling constraints and policies. OpenShift scheduling builds on the core Kubernetes scheduling mechanisms and integrates additional features to enhance the management of workloads in the OpenShift environment.

Key Components and Features

1. **Kubernetes Scheduler:**
   At its core, OpenShift uses the Kubernetes scheduler for Pod assignment, inheriting all Kubernetes scheduling features like resource requests, node affinity, taints, and tolerations.
2. **Cluster Autoscaler:**
   OpenShift integrates a Cluster Autoscaler that can automatically adjust the size of the cluster based on resource usage, adding or removing nodes as needed to meet the demands of scheduled Pods.
3. **MachineSets and Machine Autoscaler:**
   OpenShift uses MachineSets to define groups of machines and a Machine Autoscaler to automatically manage the scaling of these groups, ensuring the cluster has the right number of nodes to handle workloads.
4. **Resource Quotas and LimitRanges:**
   OpenShift allows administrators to set resource quotas and LimitRanges to control the allocation of resources within namespaces, influencing scheduling decisions by preventing resource overcommitment.
5. **PodDisruptionBudgets (PDBs):**
   PDBs ensure that a minimum number of Pods remain available during disruptions, influencing scheduling and eviction decisions to maintain application availability.
6. **Node Selectors, Affinity, and Anti-Affinity:**
   OpenShift extends Kubernetes' node selectors, affinity, and anti-affinity rules, allowing more flexible and powerful placement rules for Pods.
7. **Taints and Tolerations:**
   OpenShift leverages taints and tolerations to control Pod placement on specific nodes, ensuring that workloads are scheduled on appropriate nodes based on operational policies.
8. **Scheduling Policies:**
   OpenShift allows administrators to define custom scheduling policies to influence the scheduler’s behavior, optimizing placement based on specific requirements.

Scheduling Workflow in OpenShift

1. **Pending Pods:**
   New Pods are created and remain in a Pending state until they are scheduled.
2. **Filtering:**
   The scheduler filters nodes based on resource requests, node selectors, taints, and other criteria to find suitable nodes for the Pod.
3. **Scoring:**
   The scheduler scores the remaining nodes based on resource availability, Pod distribution, custom policies, and other factors to find the optimal node for the Pod.
4. **Binding:**
   The scheduler assigns the Pod to the highest-scoring node and updates its status to Running.

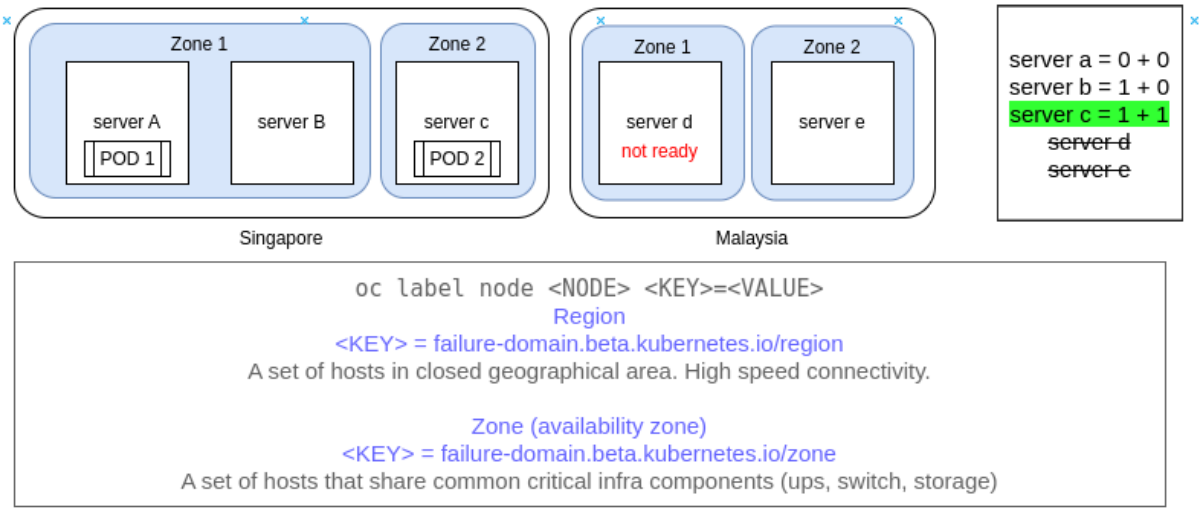

Diagram: Scheduling

5. Persistent Storage in OpenShift

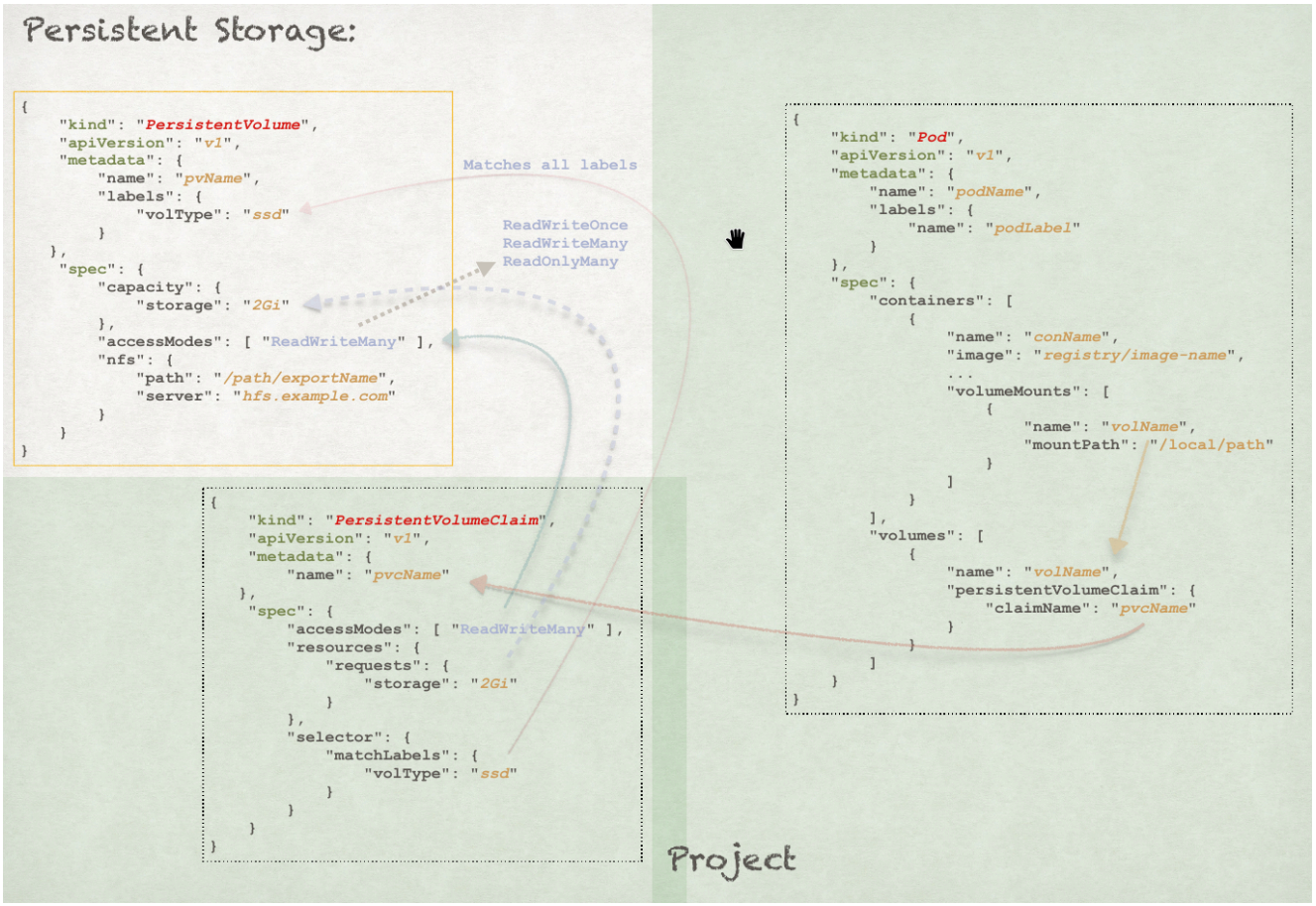

Diagram: Persistent Storage

Persistent storage in Kubernetes and OpenShift is crucial for stateful applications that need to retain data beyond the lifecycle of individual Pods. This involves mechanisms to manage storage resources that outlive Pod rescheduling and can be shared among multiple Pods if necessary. Here's an overview of the key concepts, components, and how persistent storage works in Kubernetes and OpenShift.

Key Concepts

1. **PersistentVolume (PV)**
   - A PersistentVolume is a piece of storage in the cluster that has been provisioned by an administrator or dynamically provisioned using a StorageClass.
   - PVs are cluster-wide resources that can be used by Pods.
2. **PersistentVolumeClaim (PVC)**
   - A PersistentVolumeClaim is a request for storage by a user. It is similar to a Pod in that Pods consume node resources and PVCs consume PV resources.
   - PVCs can request specific size and access modes (ReadWriteOnce, ReadOnlyMany, ReadWriteMany).

3. **StorageClass (SC)**
    - A StorageClass provides a way to describe the "classes" of storage offered by the cluster. Different classes might map to quality-of-service levels, backup policies, or arbitrary policies determined by the cluster administrators.
    - StorageClasses can be used to enable dynamic provisioning of PVs.

5.1 LAB: Persistent Storage

6. Security and Access Control

6.1 Securing OpenShift clusters: Authentication and Authorization.

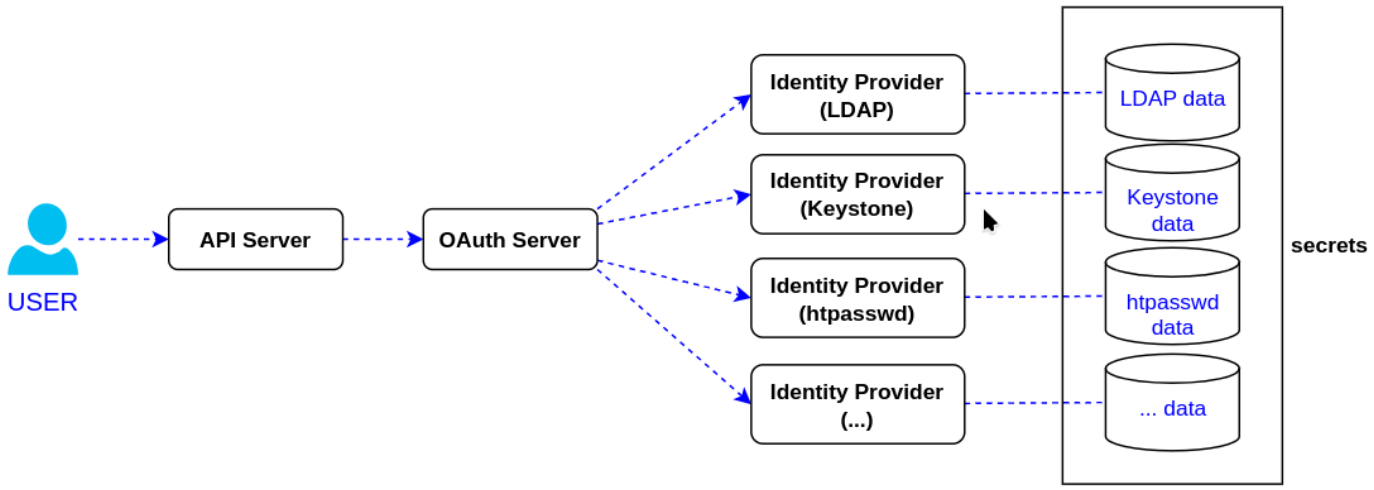

Diagram: OpenShift Request Authentication Flow

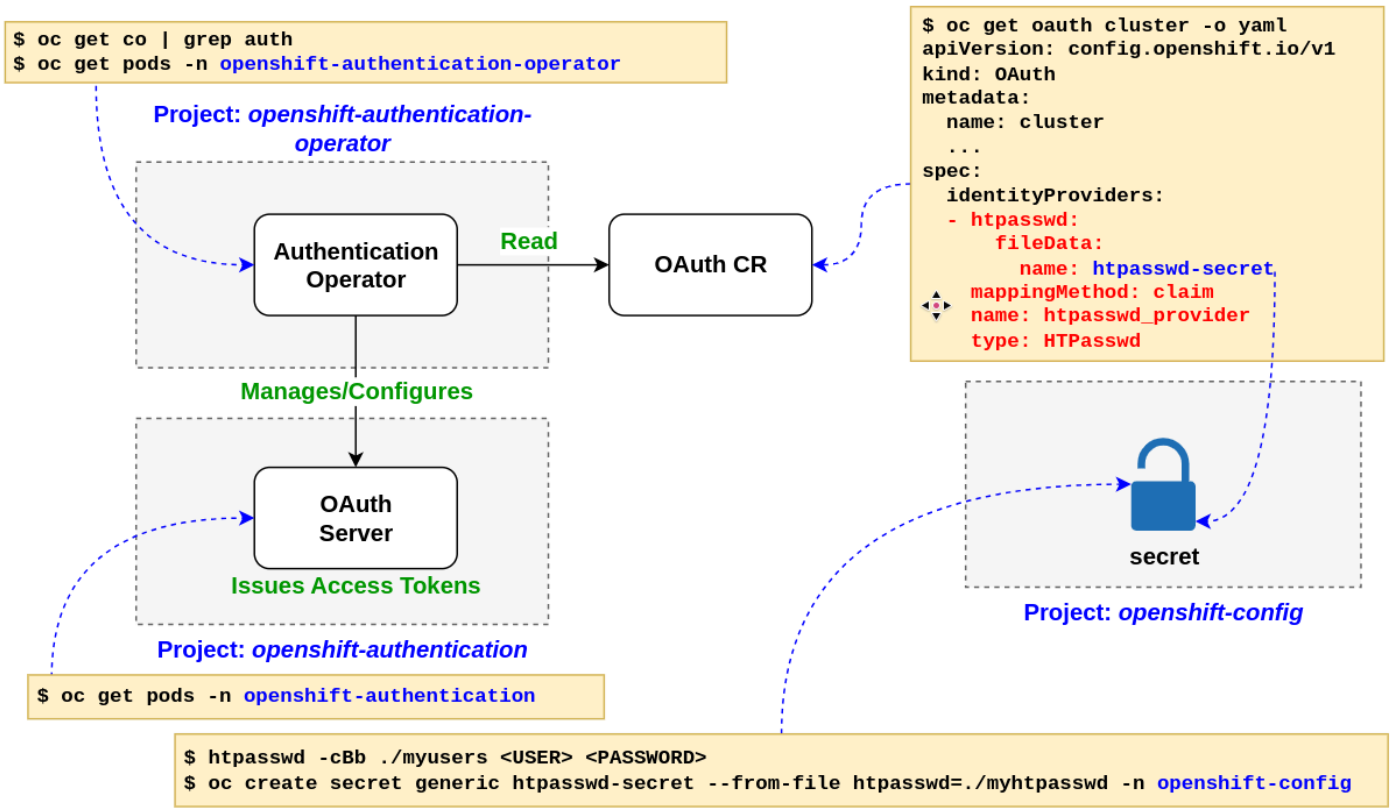

Diagram: Authentication Operator

6.2 Role-based access control (RBAC)

RBAC (Role-Based Access Control) in OpenShift is a method for regulating access to the cluster resources based on the roles assigned to users, groups, or service accounts.

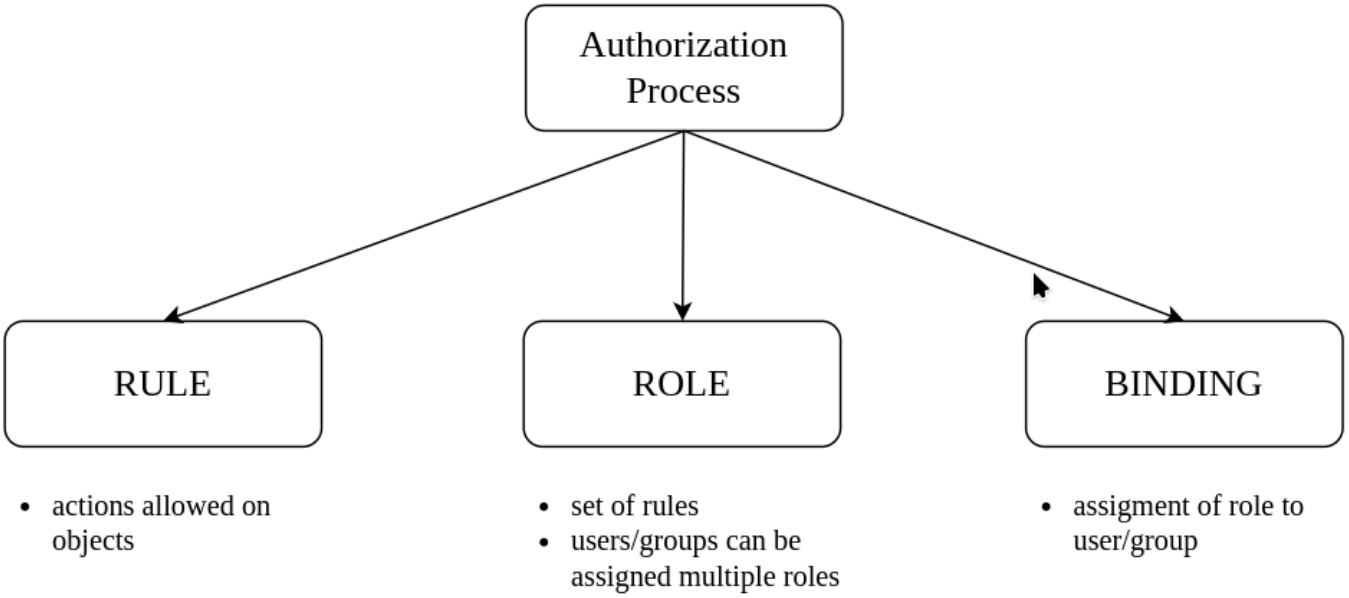

Diagram: Authorization Process

In the context of Role-Based Access Control (RBAC) in Kubernetes and OpenShift, a ***rule*** is a component that defines a set of permissions (***verbs***) that are granted by a Role or ClusterRole to a ***resource***. Verbs are actions and may be any of the following words: *get, list, create, update, delete, watch, patch,* and *exec*.

Key RBAC Components

Component	Description
Role	Defines a set of permissions within a namespace. Specifies what actions can be performed on which resources.
ClusterRole	Defines a set of permissions cluster-wide. Similar to a Role but applies to the entire cluster.
RoleBinding	Grants a Role to a user, group, or service account within a specific namespace.
ClusterRoleBinding	Grants a ClusterRole to a user, group, or service account at the cluster level.

Scope in RBAC

1. **Cluster-wide Scope**:
   - **ClusterRole**: Defines permissions that apply across the entire Kubernetes or OpenShift cluster.
     - Example: A `cluster-admin` ClusterRole that grants full access to all resources in all namespaces.
   - **ClusterRoleBinding**: Binds a ClusterRole to a user, group, or service account across the entire cluster.
     - Example: Binding the `cluster-admin` ClusterRole to the `admin-user` user, granting administrative access throughout the cluster.
2. **Namespace-specific Scope**:
   - **Role**: Defines permissions within a specific namespace.
     - Example: A `pod-reader` Role that allows reading pods (`get`, `list`, `watch`) within the `development` namespace.
   - **RoleBinding**: Binds a Role to a user, group, or service account within a specific namespace.
     - Example: Binding the `pod-reader` Role to the `developer` user within the `development` namespace.

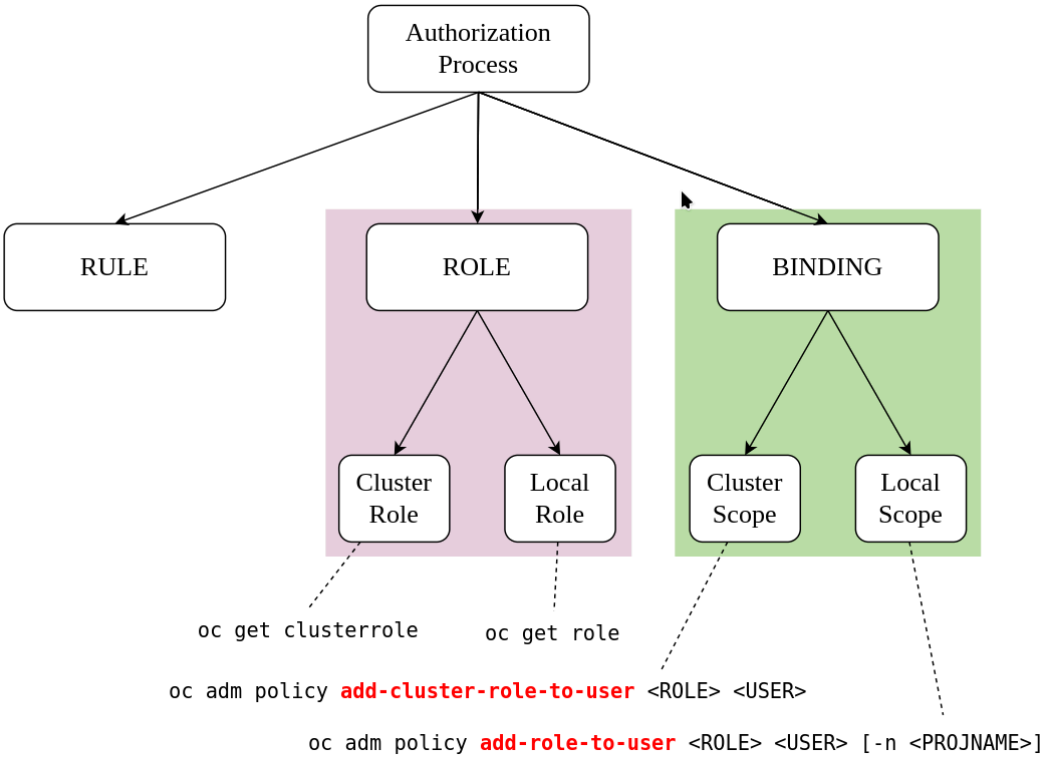

Diagram: Scope

6.2.1 LAB: RBAC

6.3 Service accounts and Security Context Constraints (SCC).

Service Accounts
Service Accounts are Kubernetes and OpenShift objects used by applications to authenticate with the API server and access cluster resources securely. They provide an identity for processes running in pods and can be used to control access to resources based on RBAC rules.

Key Points about Service Accounts:

- **Identity**: Each pod in Kubernetes/OpenShift can be associated with a Service Account, which provides an identity to interact with the API server.
- **Automounting Tokens**: By default, Service Account tokens are mounted into pods at `/var/run/secrets/kubernetes.io/serviceaccount`, allowing pods to securely access the API server.
- **RBAC Integration**: Roles and RoleBindings can grant permissions based on Service Accounts, allowing fine-grained access control for pods based on their identities.

Security Context Constraints (SCC)
Security Context Constraints (SCC) are OpenShift-specific objects that control permissions for pods and containers, including the ability to run privileged containers, access host resources, or use certain capabilities.

Key Points about SCC:

- **Granular Security Policies**: SCC allows administrators to define security policies that restrict what pods and containers can do.
- **Granular Security Policies**: SCC allows administrators to define security policies that restrict what pods and containers can do.
- **Default SCC**: Each project (namespace) in OpenShift has a default SCC assigned to its Service Accounts, ensuring that pods adhere to predefined security policies unless explicitly overridden.
- **Capabilities and Privileges**: SCC can control capabilities, SELinux context, and other security parameters for containers and pods, enforcing security best practices.

6.3.1 LAB: Service Account and SCC

6.4 Network policies for controlling traffic.

Network policies in OpenShift (and Kubernetes) are used to control the traffic flow between pods, namespaces, and external services. They help in securing the cluster by restricting the communication to only what is necessary. Here's an overview of how to use network policies to control traffic in OpenShift:

Basics of Network Policies

Namespaces:

Network policies are applied to pods within a namespace.

Each namespace can have its own set of network policies.

Selectors:

podSelector: Specifies the pods to which the policy applies.

namespaceSelector: Specifies the namespaces to which the policy applies.

Ingress and Egress Rules:

Ingress: Controls incoming traffic to the pods.

Egress: Controls outgoing traffic from the pods.

Network policies are defined using YAML files and applied using the oc command.

Example 1: Deny All Traffic

This policy denies all incoming and outgoing traffic to and from the pods in a namespace.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: deny-all
  namespace: my-namespace
spec:
  podSelector: {}
  policyTypes:
  - Ingress
  - Egress
  ingress: []
  egress: []
```

Example 2: Allow All Traffic from the Same Namespace

This policy allows all incoming traffic from pods within the same namespace.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-all-from-same-namespace
  namespace: my-namespace
spec:
  podSelector: {}
  policyTypes:
  - Ingress
  ingress:
  - from:
```

```
      - podSelector: {}
```

Example 3: Allow Traffic from Specific Pods

This policy allows traffic only from pods with a specific label.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-specific-pods
  namespace: my-namespace
spec:
  podSelector:
      matchLabels:
      role: backend
  policyTypes:
  - Ingress
  ingress:
  - from:
      - podSelector:
      matchLabels:
            role: frontend
```

Example 4: Allow Traffic from Specific Namespaces

This policy allows traffic only from pods in a specific namespace.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-from-specific-namespace
  namespace: my-namespace
spec:
  podSelector:
      matchLabels:
      role: backend
  policyTypes:
  - Ingress
  ingress:
  - from:
      - namespaceSelector:
      matchLabels:
            name: trusted-namespace
```

Example 5: Allow Egress Traffic to External Services

This policy allows egress traffic to an external service.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-external
  namespace: my-namespace
spec:
  podSelector:
      matchLabels:
      role: backend
  policyTypes:
  - Egress
  egress:
  - to:
      - ipBlock:
      cidr: 203.0.113.0/24
```

As can be seen from the above examples, we need to use namespace labels. In OpenShift, the terms "project" and "namespace" are often used interchangeably, but there are subtle differences:

1. **Namespace**: This is a Kubernetes concept used to partition resources within a single Kubernetes cluster. Namespaces are intended to be a mechanism for isolating resources and workloads.
2. **Project**: In OpenShift, a project is a higher-level concept that includes a namespace but also adds additional metadata and access controls. Projects provide a way to manage groups of resources and users, making it easier to control permissions and resource quotas.

In essence, a project in OpenShift is built on top of a Kubernetes namespace, adding extra features to support collaboration and resource management. However, when working with OpenShift, you'll often find that "project" and "namespace" can be used interchangeably, especially when referring to resource isolation. But when creating labels, we can only assign it to a namespace resource using the oc label command. We won’t be able to assign it to a project.

Use the oc describe command to describe the namespace and project resource and you will notice there are subtle differences.

```
[user@host ~]$ oc get namespace julie-dbproj
NAME          STATUS   AGE
julie-dbproj   Active   13h
[user@host ~]$ oc describe namespace julie-dbproj
Name:         julie-dbproj
Labels:       kubernetes.io/metadata.name=julie-dbproj
              pod-security.kubernetes.io/audit=restricted
              pod-security.kubernetes.io/audit-version=v1.24
              pod-security.kubernetes.io/warn=restricted
              pod-security.kubernetes.io/warn-version=v1.24
Annotations:  openshift.io/description:
              openshift.io/display-name:
              openshift.io/requester: kelvin
              openshift.io/sa.scc.mcs: s0:c30,c5
              openshift.io/sa.scc.supplemental-groups: 1000880000/10000
              openshift.io/sa.scc.uid-range: 1000880000/10000
Status:       Active

No resource quota.

No LimitRange resource.
[user@host ~]$ oc describe project julie-dbproj
Name:                   julie-dbproj
Created:                13 hours ago
Labels:                 kubernetes.io/metadata.name=julie-dbproj
                        pod-security.kubernetes.io/audit=restricted
                        pod-security.kubernetes.io/audit-version=v1.24
                        pod-security.kubernetes.io/warn=restricted
                        pod-security.kubernetes.io/warn-version=v1.24
Annotations:            openshift.io/description=
                        openshift.io/display-name=
                        openshift.io/requester=kelvin
                        openshift.io/sa.scc.mcs=s0:c30,c5
                        openshift.io/sa.scc.supplemental-groups=1000880000/10000
                        openshift.io/sa.scc.uid-range=1000880000/10000
Display Name:           <none>
Description:            <none>
Status:                 Active
Node Selector:          <none>
Quota:                  <none>
Resource limits:        <none>
[user@host ~]$
```

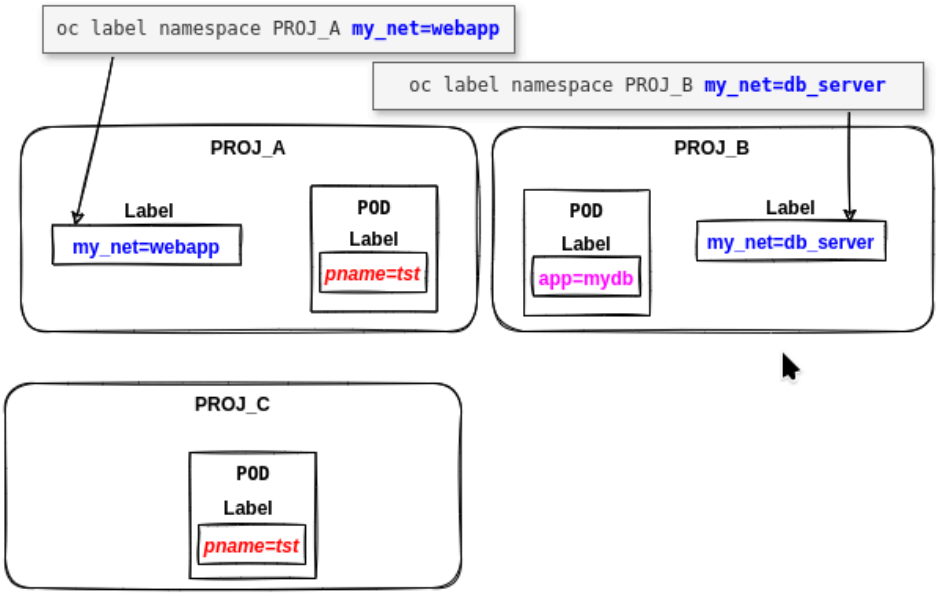

Diagram: Label Namespaces

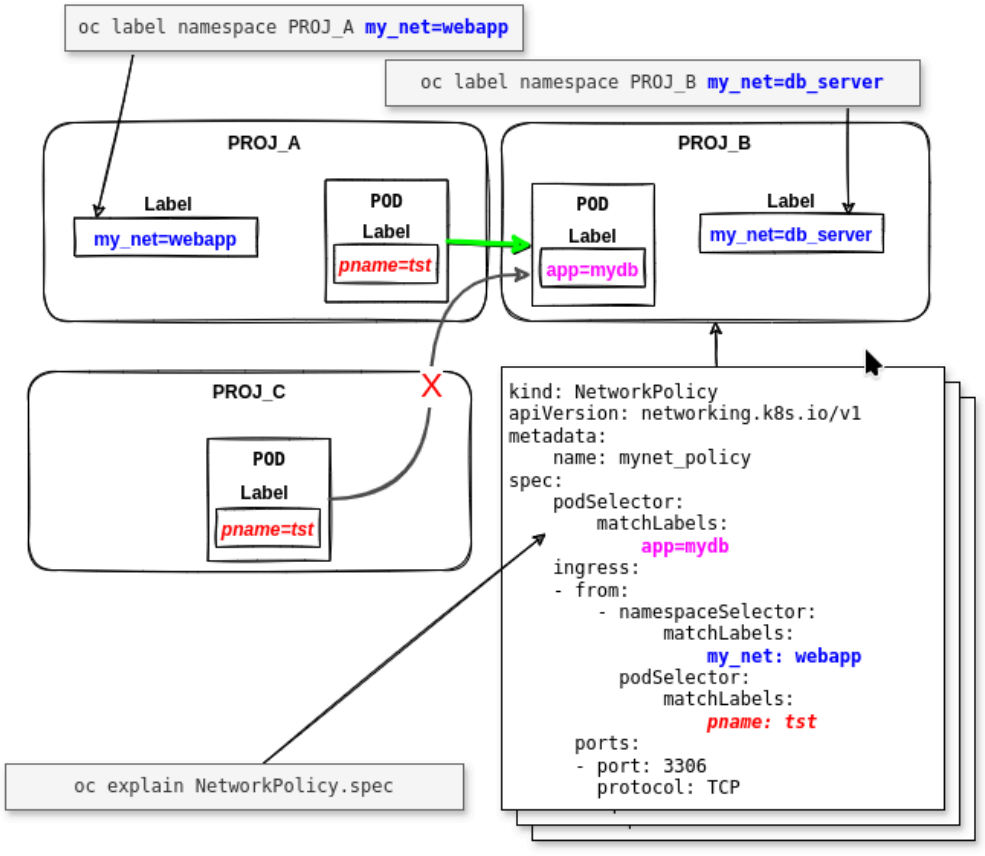

Diagram: Network Policy

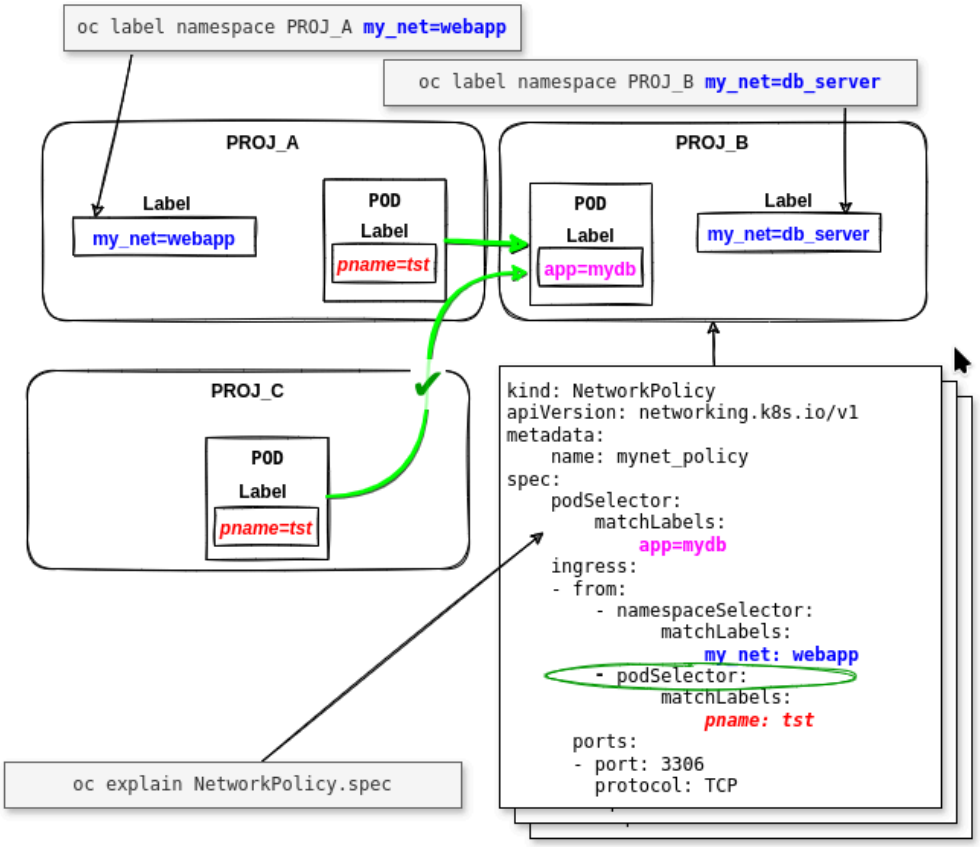

Diagram: Rules

6.4.1 LAB: Network Policy

7. Quota and capacity management.

Resource Requests

Resource Requests in Kubernetes and OpenShift specify the minimum amount of CPU and memory that a container or pod requires to run effectively. Requests ensure that Kubernetes scheduler can find a suitable node with enough capacity to accommodate the pod's resource requirements.

Key Points about Resource Requests:

- **Definition**: Requests are specified in the pod's YAML definition under `spec.containers.resources.requests`.
- **Guarantee**: Once a node accepts a pod based on its resource request, Kubernetes ensures that the requested amount of CPU and memory is reserved exclusively for that pod.
- **Implications**: Requests affect scheduling decisions and placement of pods on nodes. Nodes with insufficient available resources may not be able to schedule pods with higher resource requests.

Resource Limits

Resource Limits specify the maximum amount of CPU and memory that a container or pod is allowed to use. Limits ensure that pods do not exceed allocated resources and impact other pods or the node's stability.

Key Points about Resource Limits:

- **Definition**: Limits are specified in the pod's YAML definition under `spec.containers.resources.limits`.
- **Enforcement**: Kubernetes enforces limits by throttling CPU usage and terminating processes or containers that exceed memory limits.
- **Impact**: Exceeding limits may lead to degraded pod performance or even pod eviction if resources cannot be reclaimed.

Quota

Quota in Kubernetes and OpenShift refers to resource limits that can be set at the namespace level to control and allocate resources among different projects or teams. It helps in managing resource usage effectively and ensuring fair distribution across the cluster.

Key Points about Quota:

- **Resource Types**: Quota can be applied to various Kubernetes resources such as CPU, memory, storage, number of pods, and number of services.
- **Namespace Scope**: Quota limits apply within a specific namespace, allowing administrators to enforce resource constraints for individual projects or teams.
- **Quota Objects**: In Kubernetes, `ResourceQuota` objects are used to define these limits, specifying the maximum amount of each resource type that can be consumed.

Capacity Planning

Capacity Planning involves forecasting future resource requirements based on current usage and growth projections. It ensures that sufficient resources (CPU, memory, storage) are available to support application workloads without performance degradation or resource contention.

Considerations for Capacity Planning:

- **Monitoring and Analysis**: Regularly monitor cluster resource usage and performance metrics to identify trends and potential bottlenecks.
- **Scalability**: Plan for scalability by considering future workload growth and requirements for horizontal and vertical scaling.
- **Resource Allocation**: Allocate resources based on application requirements, considering factors like workload characteristics, resource constraints, and expected traffic patterns.
- **Resilience**: Ensure redundancy and failover capabilities to maintain service availability during peak usage or in case of node failures.
- **Efficiency**: Optimize resource utilization through techniques like resource requests and limits, pod autoscaling, and efficient workload scheduling.

Tools and Strategies:

- **Prometheus and Grafana**: Monitoring tools for collecting and visualizing cluster metrics.

- **Horizontal Pod Autoscaler (HPA)**: Automatically adjusts the number of pod replicas based on CPU utilization.
- **Cluster Autoscaler**: Scales nodes in the cluster up or down based on demand to ensure adequate capacity.
- **Capacity Planning Models**: Use historical data and workload modeling to forecast resource requirements and plan capacity upgrades or optimizations.

7.1 LAB: Quota