Containers 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.

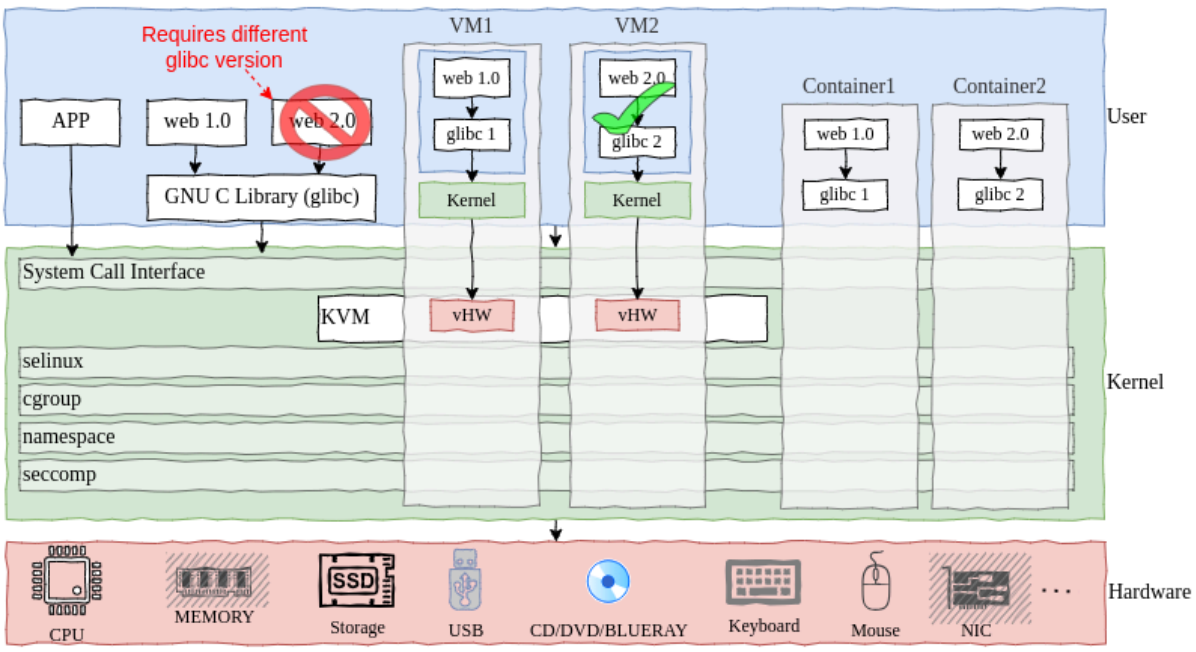

Diagram: Native vs VM vs Container

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).

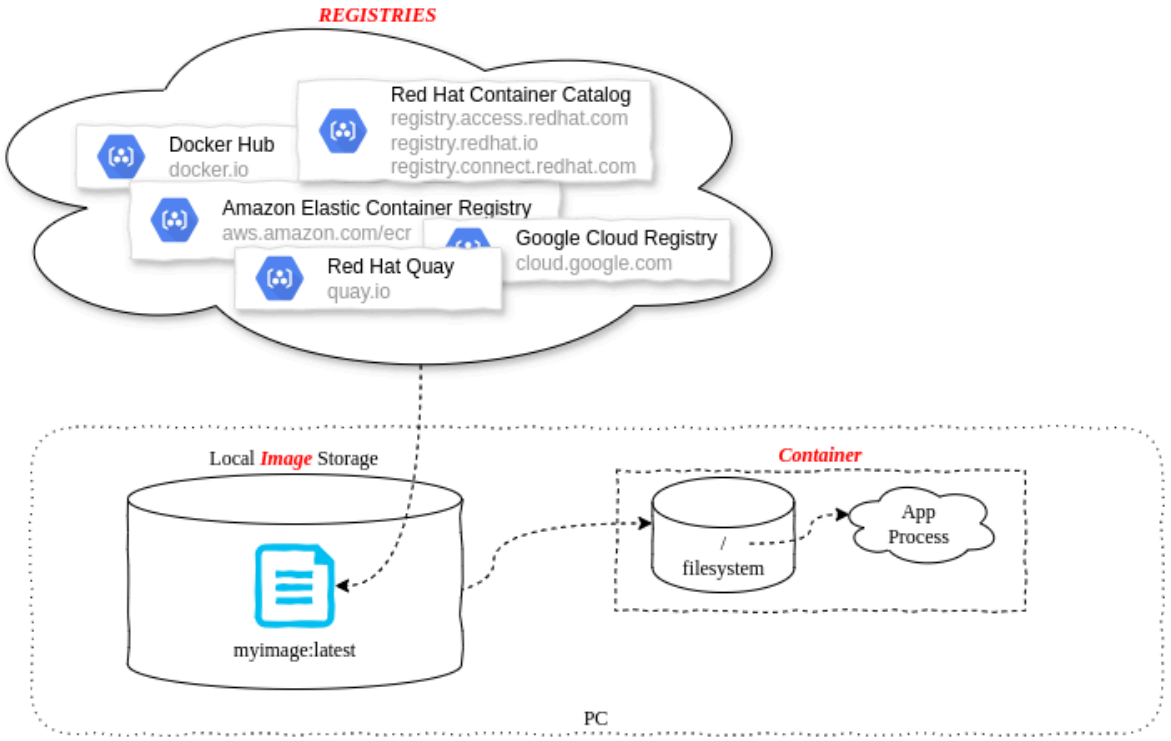

Diagram: Container Architecture

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`

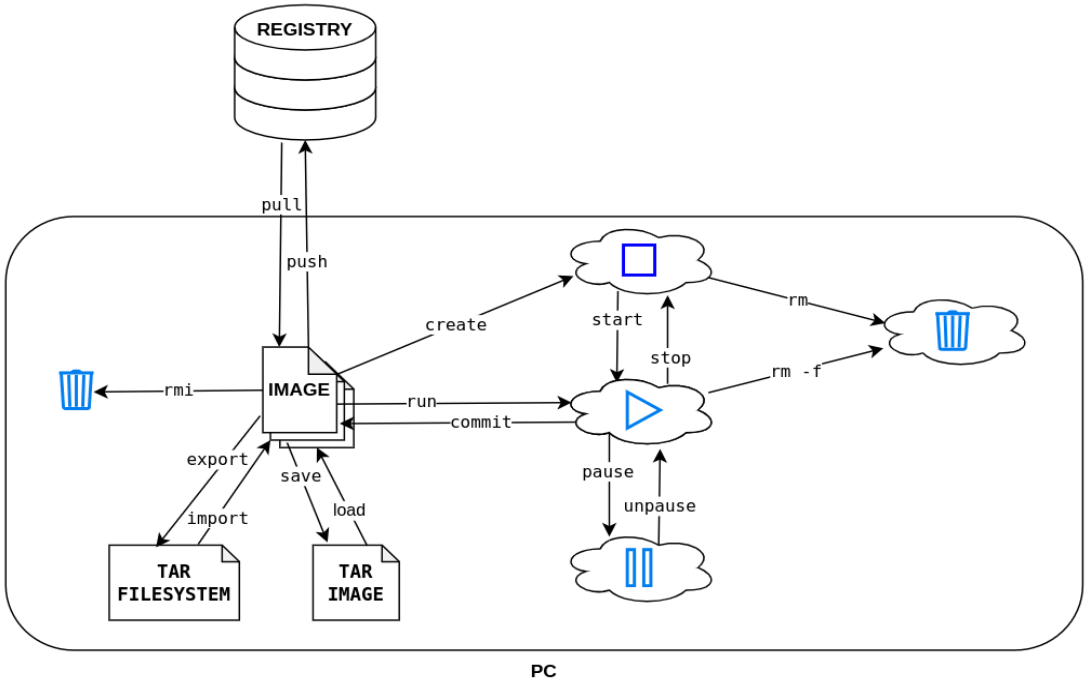

Diagram: Basic Container Lifecycle

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`

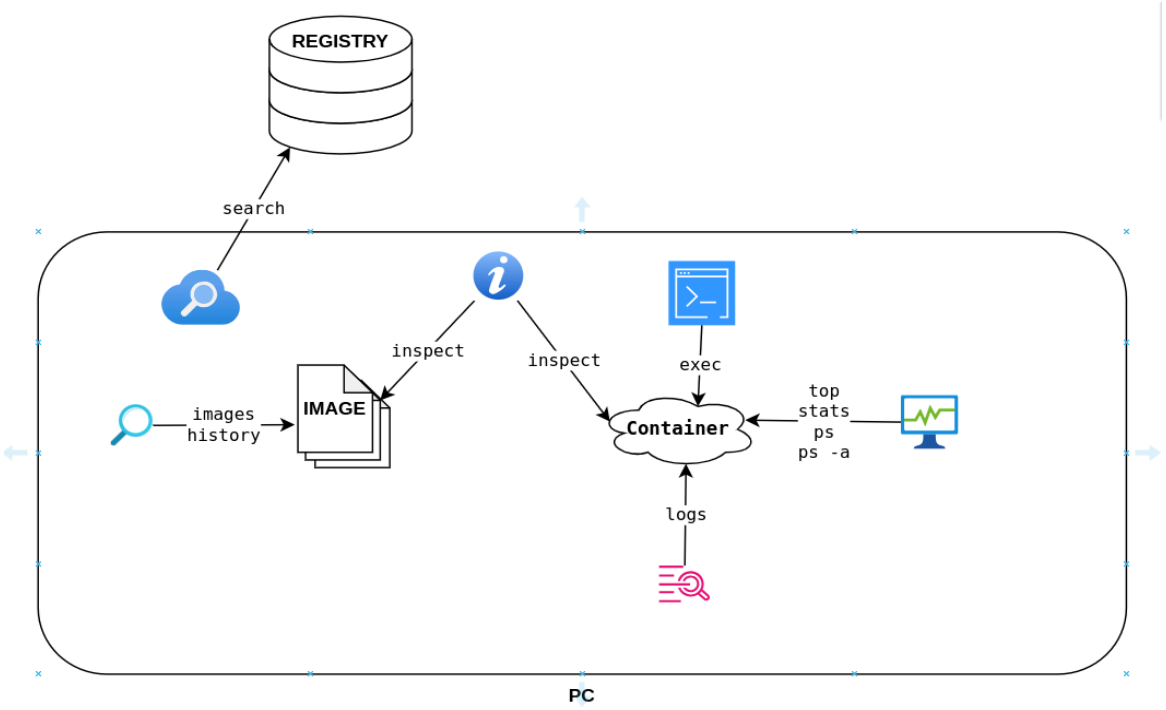

Diagram: Operations

2.2 Understanding Image and Registry

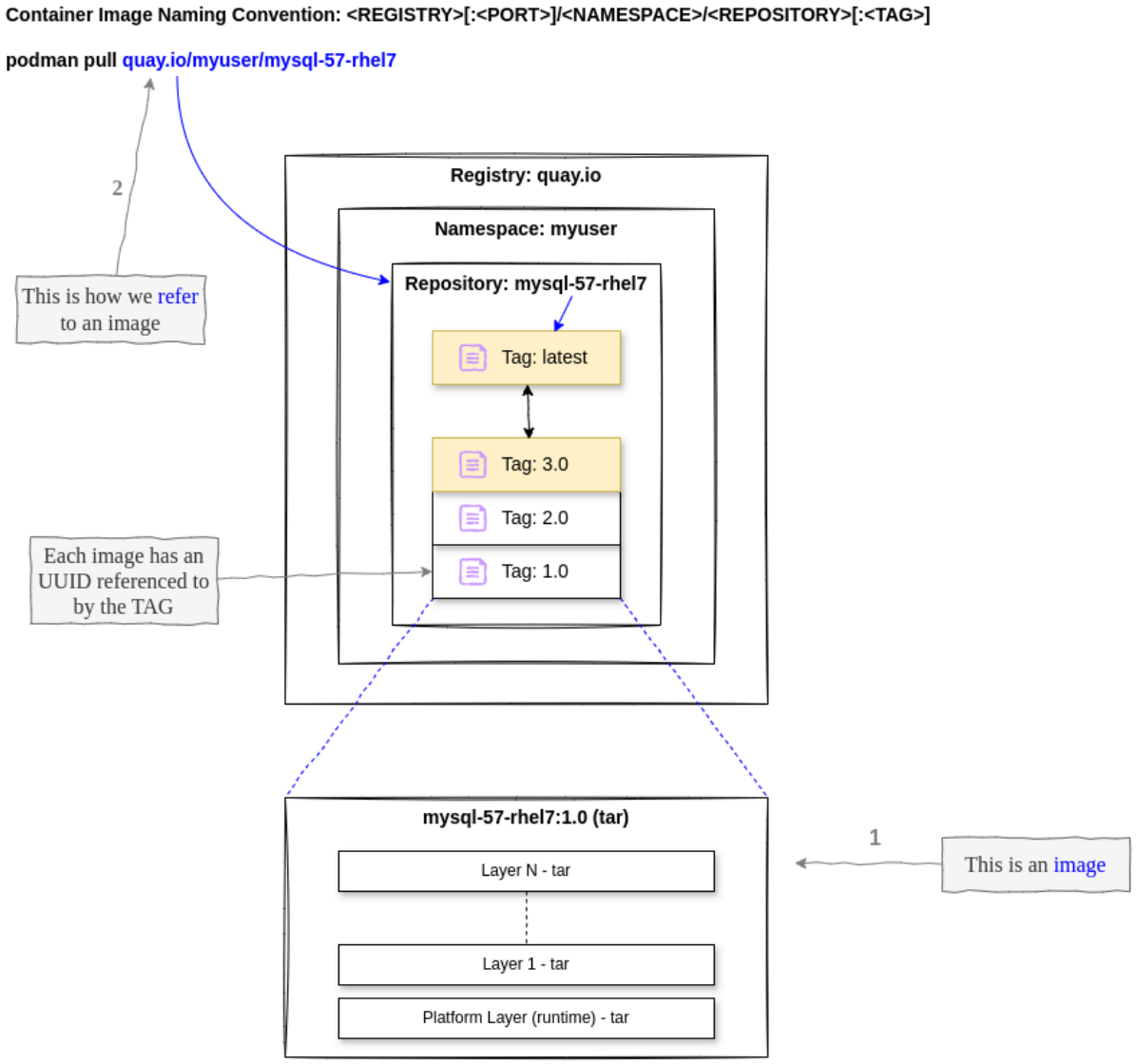

Diagram: Image and Registry

2.3 Understanding Image and Containers

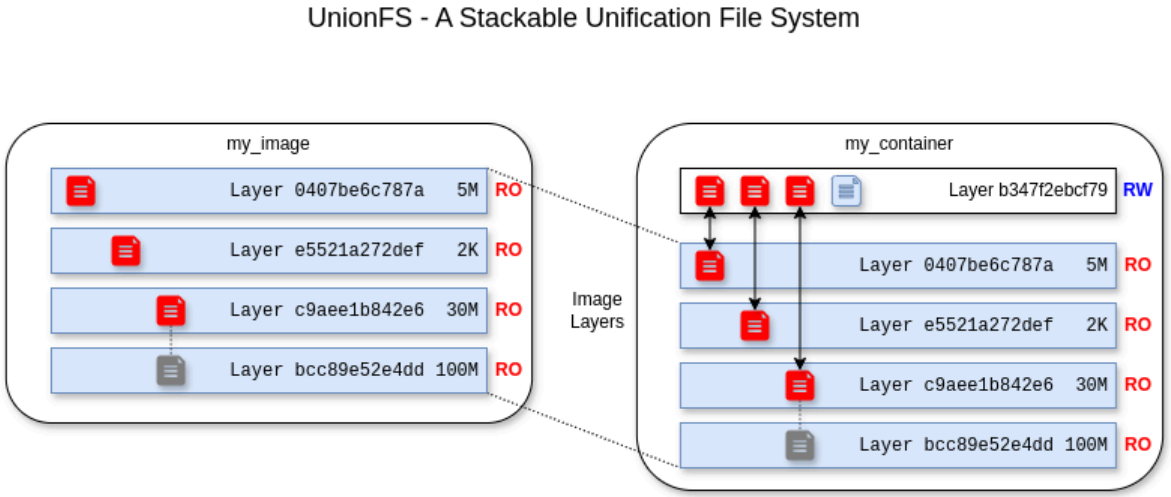

Diagram: Image and Container Relationship

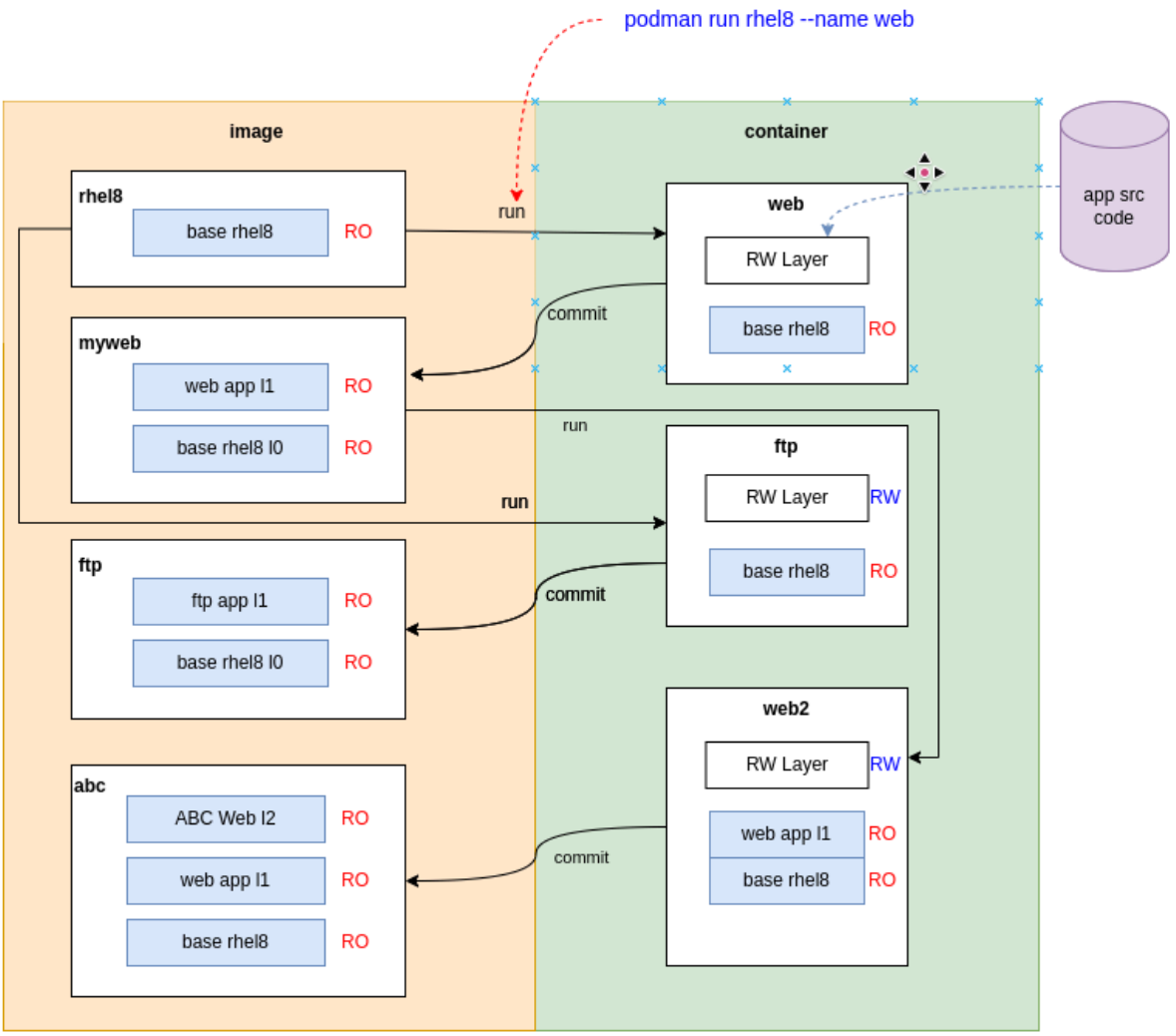

Diagram: Simple Image Creation

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

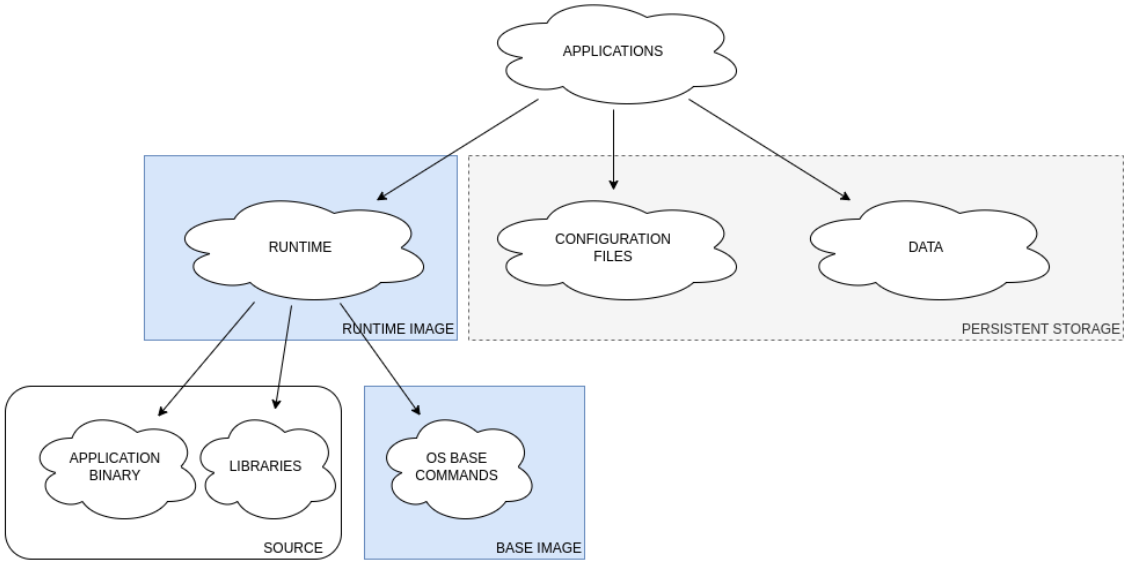

Diagram: Basic Container Design

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

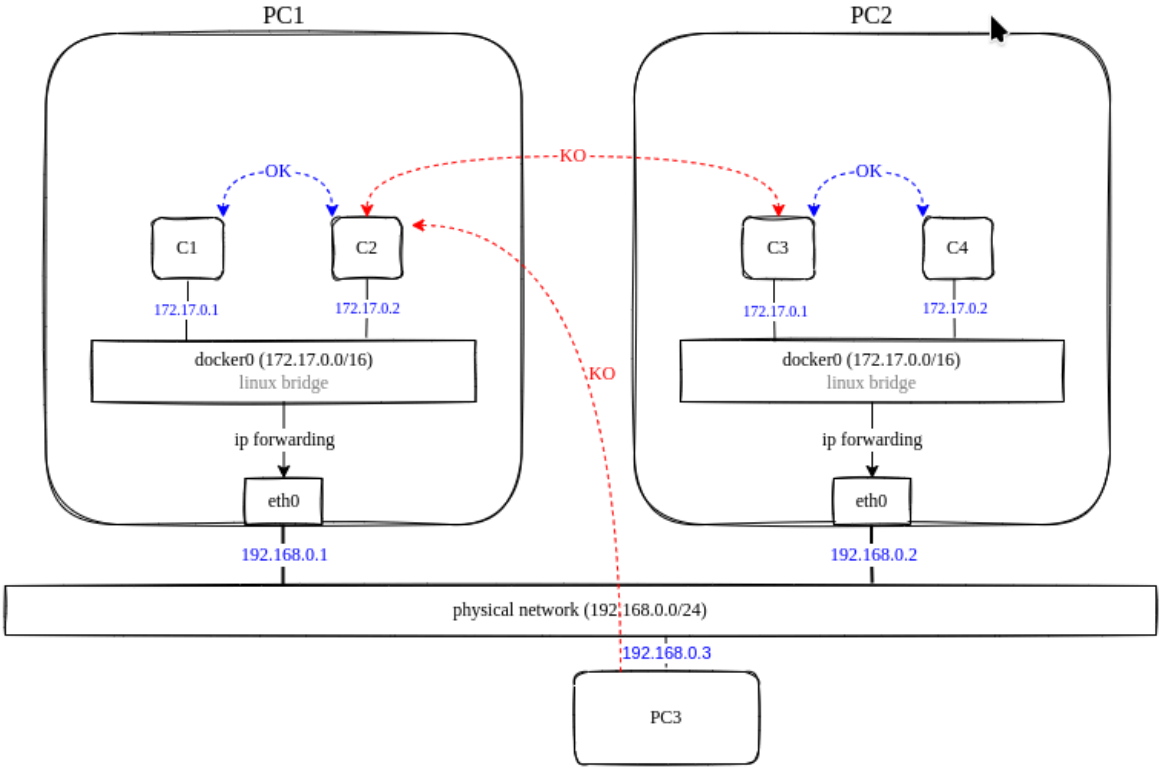

Diagram: Containers Network

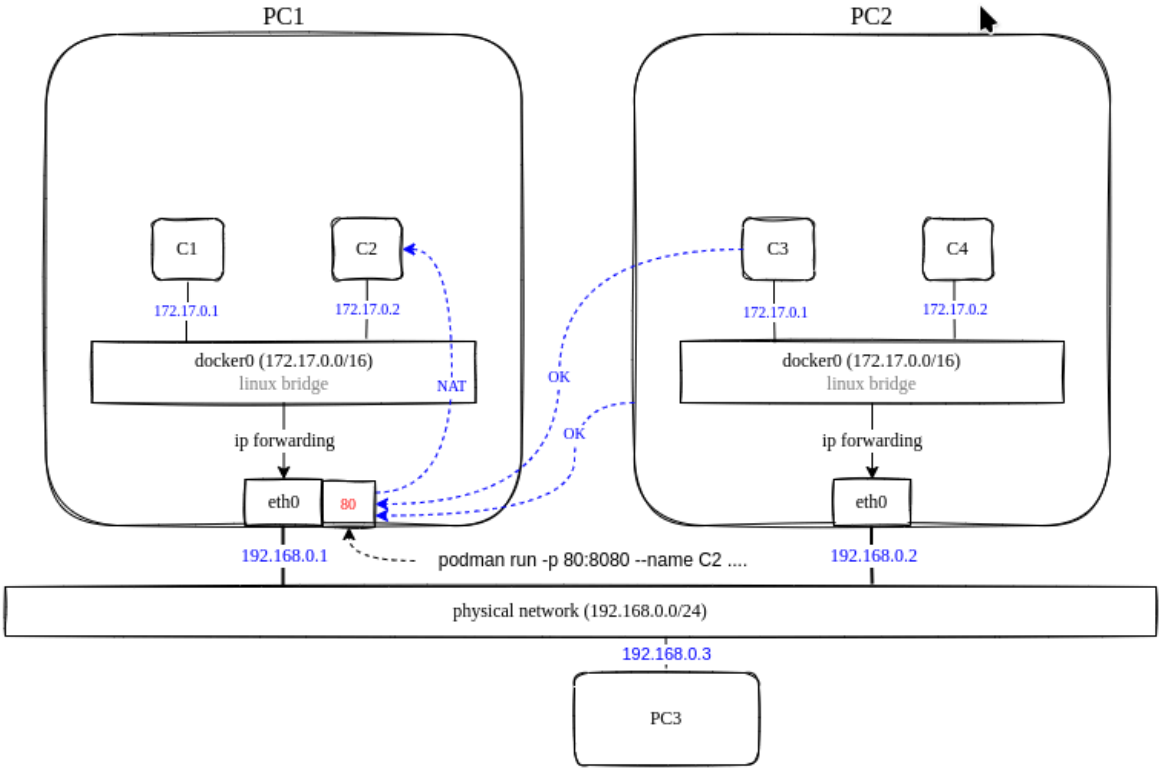

Diagram: Port Forwarding