Docker and Kubernetes

🐳 Chapter 1: Docker Overview

🎯 Chapter Objectives

By the end of this chapter, learners will be able to:

- Understand the evolution from virtualization to containers
- Distinguish between VMs and containers
- Explain container architecture, image formats, and runtimes
- Describe Docker's architecture and role in container management
- Perform a hands-on Docker installation

1.1 Virtualization

Virtualization is a technology that enables multiple isolated computing environments (virtual machines) to run concurrently on a single physical machine. This is achieved by abstracting the hardware layer through a software component called a hypervisor.

There are two primary types of hypervisors:

- Type 1 (Bare-metal): Runs directly on hardware (e.g., VMware ESXi, Microsoft Hyper-V, KVM)
- Type 2 (Hosted): Runs on a host OS (e.g., VirtualBox, VMware Workstation)

Each virtual machine (VM) operates with:

- Its own guest operating system
- A set of virtual hardware resources (CPU, memory, storage, NIC, etc.)
- A fully isolated user space and kernel

Motivation for Virtualization

On traditional systems, multiple applications share the same operating system and system libraries. This can result in conflicts when different applications require incompatible versions of shared dependencies.

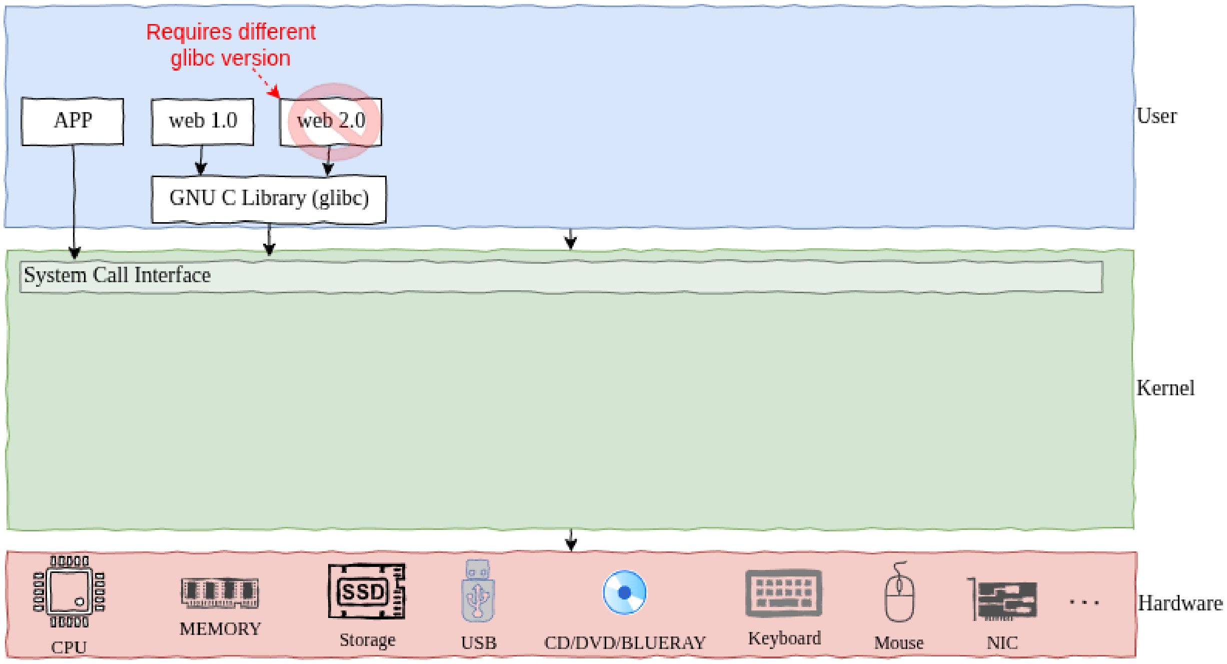


Diagram: Native Application Execution on Host

In the above example:

- The application includes `web 1.0` and `web 2.0` components.
- Both rely on different versions of the GNU C Library (glibc).
- The host OS cannot load both versions of glibc into the same user space simultaneously.
- This results in dependency conflicts and prevents the application from functioning correctly.

Use of VMs to Resolve Conflicts

Virtual machines are used to resolve such dependency issues by isolating each application stack in its own virtual environment. This enables:

- Running multiple versions of the same library or runtime without conflict
- Clear separation of application environments
- Improved compatibility across workloads

This leads directly into the next section, where we explore how VMs provide that isolation effectively — and what trade-offs they introduce.

1.2 Virtual Machines

Virtual Machines (VMs) are a form of hardware virtualization that allow multiple operating systems to run independently on a single physical host. This is achieved through a software layer known as a hypervisor, which abstracts and emulates physical hardware.

Each VM includes:

- A complete guest operating system
- A set of virtualized hardware components (vCPU, vRAM, vNIC, etc.)
- An application stack isolated from the host and other VMs

Purpose of VMs in Application Isolation

VMs are used to address application-level conflicts, particularly when different applications require incompatible versions of system libraries or dependencies.

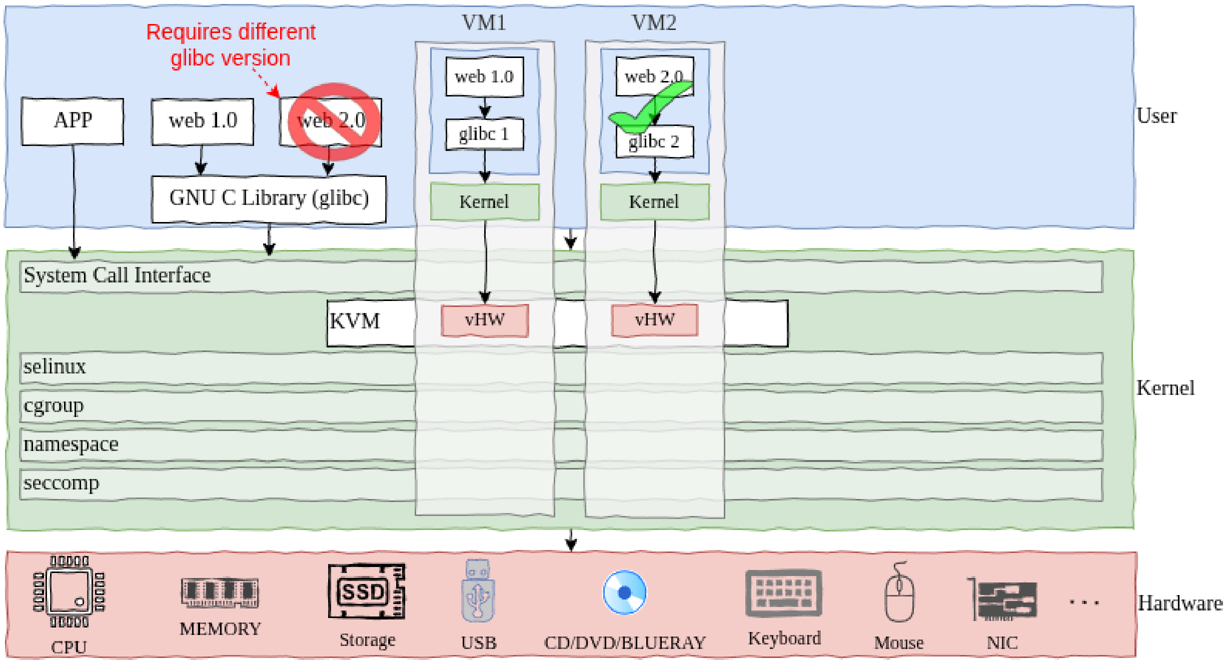


Diagram: Virtual Machines Isolating Conflicting Dependencies

In the example:

- Two components (`web 1.0` and `web 2.0`) require different versions of the GNU C Library (glibc).
- These conflicts cannot be resolved when both components run natively on the same host OS.
- By deploying each component in a separate VM:
    - VM1 includes `web 1.0` with `glibc 1`
    - VM2 includes `web 2.0` with `glibc 2`

Each VM provides an isolated environment with its own kernel, resolving dependency issues effectively.

Characteristics of Virtual Machines

Feature	Description
Isolation	Full process and kernel-level isolation
Guest OS	Each VM runs its own independent OS instance
Resource Overhead	High — includes full OS and virtual hardware
Startup Time	Typically slower (in the range of seconds to minutes)
Use Cases	Legacy systems, full-stack testing, security-isolated workloads

Limitations

While VMs provide strong isolation and compatibility, they introduce considerable overhead:

- Increased memory and storage consumption
- Slower provisioning time
- Inefficiency when scaling microservices or running small, distributed workloads

In the next section, we will examine how containers achieve similar application isolation using a more lightweight, kernel-shared model — enabling faster and more efficient deployment at scale.

1.3 Containers

Containers are a lightweight alternative to virtual machines that provide application isolation at the **operating system level** rather than the hardware level. Instead of emulating physical hardware and running a full guest OS, containers share the **host operating system kernel** while maintaining isolated user spaces for individual applications.

Containers are well-suited for modern software delivery models such as **microservices**, **DevOps pipelines**, and **cloud-native applications** due to their efficiency and scalability.

Key Characteristics of Containers

Feature	Description
OS-level Isolation	Applications run in separate user spaces, but share the host kernel
Lightweight	No need for full OS per application; significantly reduces resource usage
Fast Startup	Containers typically start in under a second
Portability	Run consistently across environments (development, test, production)
Ephemeral by Design	Designed to be disposable and reproducible

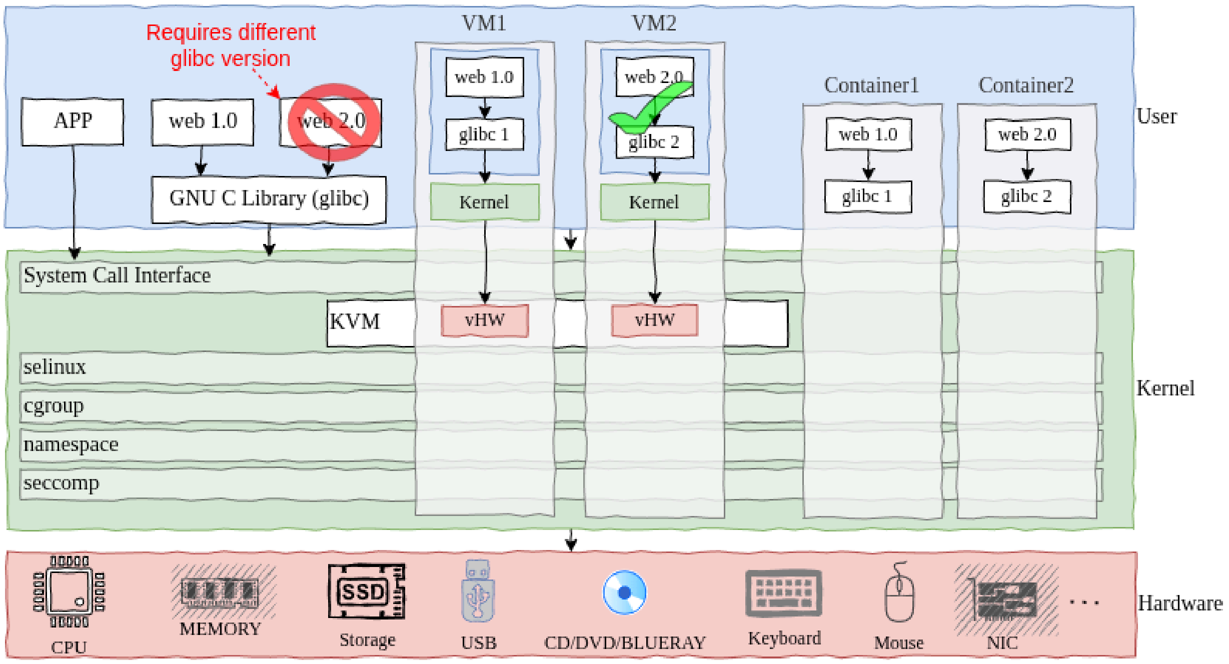


Diagram: Containers vs Virtual Machines

In the diagram above:

- The top half shows two virtual machines:
  - Each VM has its own **OS kernel**, system libraries, and application stack
  - This results in significant duplication of resources

- The bottom half shows two containers:
  - Both containers share the **host OS kernel**
  - Each container maintains its own **user space**, including specific libraries (e.g., different `glibc` versions)
  - **Isolation** is achieved using kernel features like namespaces, cgroups, and security modules (e.g., SELinux, seccomp)

Comparison: Containers vs Virtual Machines

Attribute	Virtual Machine	Container
OS Overhead	Full guest OS per instance	Shared host OS kernel
Startup Time	Minutes	Seconds
Resource Efficiency	High resource usage	Low resource footprint
Isolation Level	Hardware-level (via hypervisor)	OS-level (via kernel namespaces)
Portability	Limited (depends on hypervisor, OS)	High (runs anywhere with container runtime)
Use Cases	Monolithic apps, full OS testing	Microservices, CI/CD, cloud workloads

Advantages of Containers in Practice

- Run multiple isolated applications on a single host with minimal overhead
- Ideal for **stateless services**, **automated testing**, and **scalable deployments**
- Containers integrate seamlessly with orchestration platforms like **Kubernetes** for enterprise-scale management

The next section (1.4) will outline the specific benefits of container-based architectures in more detail, before moving on to container internals and Docker itself.

1.4 Benefits of Containers

Containers have become a foundational technology in modern software development and deployment workflows due to their efficiency, consistency, and scalability. Their adoption is driven by both operational and architectural benefits.

1. Portability

Containers encapsulate an application along with its runtime environment, configuration files, and dependencies. This makes them highly portable across different environments:

- Development laptops
- Testing servers
- Production clusters
- On-premise infrastructure or cloud platforms

Key Point: If a container runs on one system, it will run the same way on another system with a compatible container runtime.

2. Lightweight and Efficient

Containers do not require a full guest OS. Instead, they share the host kernel, which results in:

- Lower memory and CPU usage
- Smaller disk footprints
- Higher density of workloads per host

Example: A typical container image may be under 100MB, whereas a VM image may exceed several GB.

3. Fast Startup and Teardown

Containers can start and stop in seconds, enabling:

- Rapid deployment of applications
- Faster scaling in response to load
- Efficient automation in CI/CD pipelines

4. Consistency Across Environments

Containers ensure that the application behaves the same regardless of the underlying infrastructure. This eliminates the common issue of:

- “It works on my machine, but not in production.”

Use Case: Developers can build and test locally using containers, and deploy the same container image to staging or production.

5. Scalability and Automation

Containers are stateless and easily replicable, making them ideal for:

- Horizontal scaling (adding more container instances)
- Automated orchestration (e.g., Kubernetes, Docker Swarm)
- Dynamic load balancing and self-healing systems

6. Isolation

Each container runs in an isolated user space, which helps:

- Reduce risk of conflict between applications
- Improve security boundaries between services
- Simplify dependency management

Note: While not as isolated as VMs, containers provide sufficient process-level isolation for most use cases, enhanced further with tools like AppArmor, seccomp, and SELinux.

7. Developer and DevOps Productivity

Containers integrate well with modern development workflows and automation tools:

- Easy to define using **Dockerfiles**
- Build pipelines can produce container images as artifacts
- Seamless integration with CI/CD systems (e.g., GitLab CI, Jenkins, GitHub Actions)

Summary Table

Benefit	Description
Portability	Runs the same across dev, staging, and production
Lightweight	Minimal overhead compared to VMs
Speed	Fast startup and teardown
Consistency	Eliminates environment-specific issues
Scalability	Ideal for container orchestration and cloud-native patterns
Isolation	Applications are sandboxed to avoid conflicts
Automation	Supports CI/CD, infrastructure as code, and versioned deployments

In the following sections, we will explore the **container architecture** (1.5) and the **image format** (1.6) that make these benefits possible in practice.

1.5 Architecture of Containers

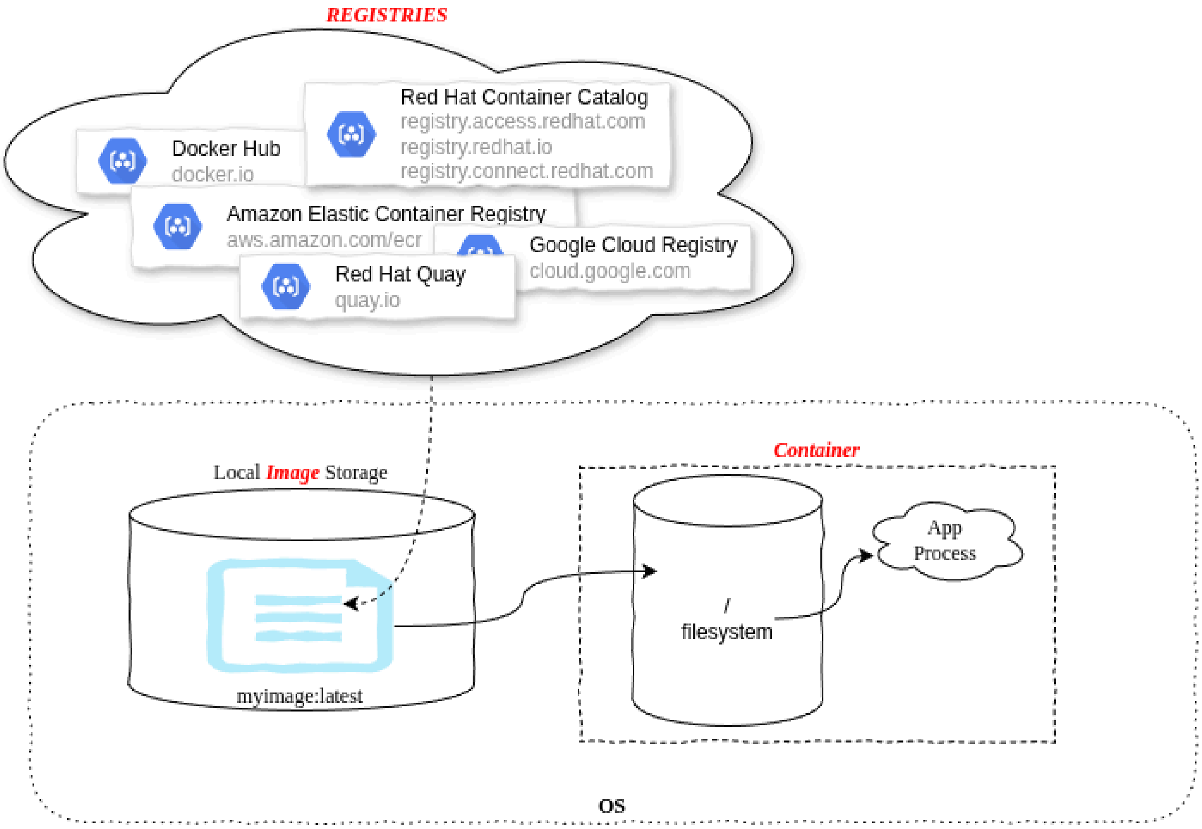


Diagram: Container Architecture

Containers provide a lightweight and portable environment for running applications by isolating processes at the operating system level. This is achieved through a combination of Linux kernel features and a structured packaging format.

The core components of container architecture include:

- Container images
- The container runtime
- The host kernel
- Optional storage and networking layers
- Registries for image distribution

1. Container Image

A **container image** is a read-only filesystem snapshot that includes everything required to run an application:

- Application binaries and scripts
- Configuration files
- Required libraries and runtimes
- Base OS layer (e.g., Alpine, Ubuntu, Debian)

Images are **built in layers**, with each layer corresponding to a step in the image creation process (typically defined in a Dockerfile). These layers are reusable and cached for performance and efficiency.

2. Writable Container Layer

When a container runs from an image, a new **writable layer** is created on top of the image stack. This layer stores runtime changes such as logs, temporary files, or modified data.

- It is unique to the container
- It is discarded when the container is deleted (unless volumes are used)

3. Container Runtime

The container runtime is responsible for:

- Creating and managing containers
- Starting and stopping processes inside containers
- Applying kernel-level isolation (namespaces, cgroups)
- Managing storage and networking

Examples of container runtimes:

- `runc` (default low-level runtime used by Docker and containerd)
- `containerd` (higher-level runtime)
- `CRI-O` (used in Kubernetes environments)

Docker uses `containerd` and `runc` internally to manage container lifecycles.

4. Host Operating System Kernel

Containers do not virtualize hardware or run their own kernel. Instead, they **share the kernel of the host operating system**.

Key Linux kernel features used:

Feature	Description
Namespaces	Isolate process trees, users, mount points, network stacks
cgroups	Control resource usage (CPU, memory, I/O)
seccomp	Restrict syscalls that a container can invoke
AppArmor/SELinux	Enforce security policies for container processes

Because containers depend on the host kernel, container images must be compatible with the host OS type (e.g., Linux containers run on Linux hosts).

5. Image Registry

A **container registry** is a centralized service for storing and distributing container images.

Common registries:

- **Docker Hub** (default public registry)
- **GitLab Container Registry**
- **Amazon ECR**, **Google Container Registry**
- **Harbor**, **Quay** (for private registries)

Images can be **pushed to** and **pulled from** a registry using CLI tools like `docker`, `podman`, or `ctr`.

6. Optional Components

- **Volumes**: Used for persistent data storage outside the container's writable layer.
- **Container Networks**: Enable communication between containers and with external systems using virtual bridges, NAT, and overlays.

This architecture enables fast, repeatable, and resource-efficient application deployment. In the next section, we will explore how **container images** are structured — with a focus on layers and image formats.

1.6 Image Format

A container image is a structured, immutable filesystem snapshot that includes the application code, required libraries, dependencies, and configuration metadata. Docker uses a layered filesystem format that supports reuse, caching, and efficient distribution.

1. Structure of a Container Image

Docker images conform to the **OCI (Open Container Initiative) Image Specification** and consist of:

Component	Description
Layers	Filesystem changes (usually one per Dockerfile instruction)
Config	JSON metadata about environment variables, commands, entrypoint, etc.
Manifest	Describes the image structure and lists its layers
Tags	Human-readable references (e.g., `nginx:1.25`, `app:latest`)
Digest	SHA256 hash uniquely identifying the image's contents

Each image is composed of **read-only layers** stacked in order, and a **container** adds a final writable layer on top when it runs.

2. Building Images from Dockerfiles

Each instruction in a Dockerfile results in a new image layer. Example:

Dockerfile

```
FROM ubuntu:24.04
RUN apt update
RUN apt install -y nginx
COPY ./web /var/www/html
```

This creates a layered image structure:

```
Layer 0: Base image (Ubuntu)
Layer 1: apt update
Layer 2: apt install nginx
Layer 3: copy application files
```

These layers are cached and reused to speed up future builds and minimize storage duplication.

3. Image → Container Runtime Model

When a container is started from an image (using `docker run`), Docker adds a **writable container layer** on top of the read-only image layers. Any file changes (writes, deletions, logs) happen in this layer.

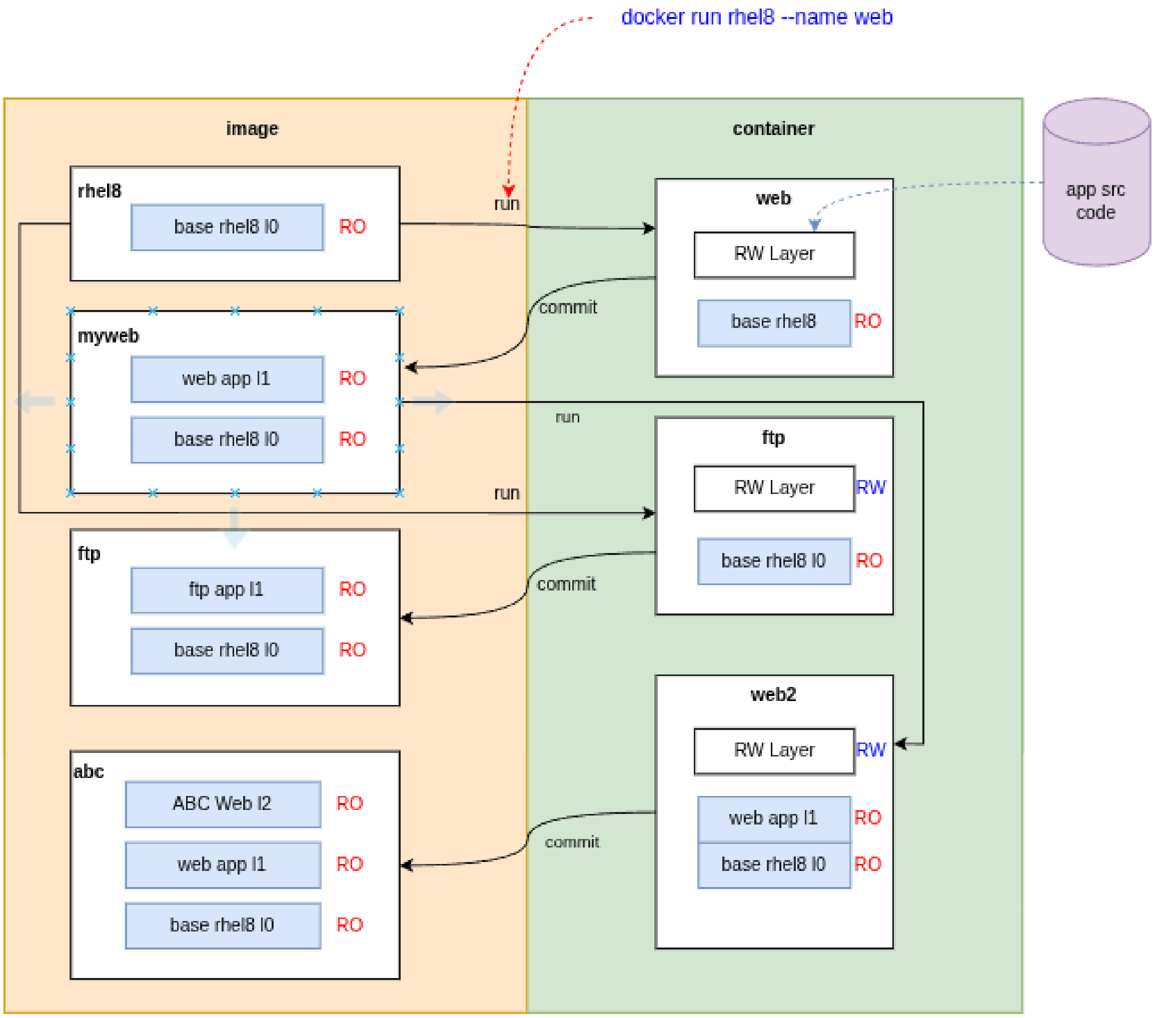


Diagram: Image and Container Lifecycle

The diagram shows multiple base and derived images on the left, and containers created from them on the right. Each container has a writable layer on top of read-only layers inherited from the image. The `docker commit` command can be used to convert the running container state into a new image, which itself may be reused.

4. Image Layer Internals and UnionFS

Docker uses a **union filesystem (UnionFS)** to stack image layers. The filesystem exposed to a running container is the result of **merging read-only image layers with the writable container layer**.

This provides:

- Efficient storage
- Fast startup time
- Shared caching across containers

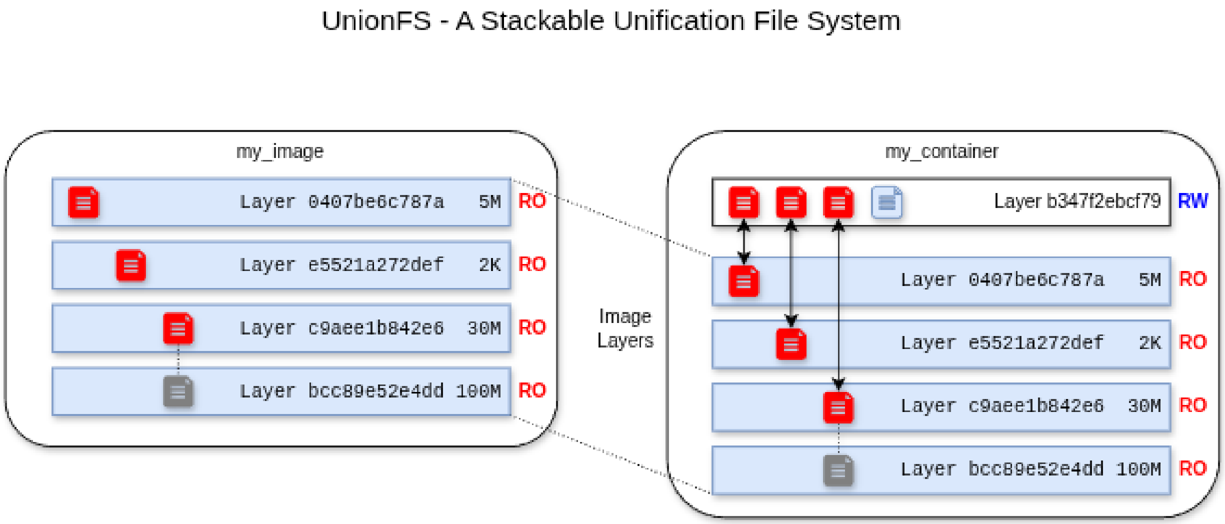


Diagram: UnionFS File Structure

Each image layer is stored separately and marked as `RO` (read-only). When a container runs, a writable `RW` layer is added at the top. UnionFS provides a unified view where all layers appear as a single coherent filesystem. When the container modifies a file, a copy-on-write mechanism ensures it is handled in the writable layer only.

5. Pulling and Distributing Images

Images can be pushed to or pulled from **container registries**, which are remote storage systems compliant with the OCI distribution spec.

Common registries include:

- Docker Hub: `docker.io`
- Red Hat Quay: `quay.io`
- Amazon ECR: `aws.ecr`
- Google Container Registry: `gcr.io`
- Red Hat Container Catalog: `registry.redhat.io`, `registry.access.redhat.com`

Images can be referenced by **tag** or **digest**. Example:

```
docker pull nginx:1.25
docker pull nginx@sha256:<digest>
```

6. Summary

- Images are composed of **stacked read-only layers** with metadata and configuration.
- A container adds a **writable layer** during runtime.
- Layers are cached, reusable, and form the basis of Docker’s storage efficiency.
- **UnionFS** merges these layers to expose a complete filesystem to the container.
- Images are stored and distributed using registries and versioned via tags and digests.

1.7 Docker

Docker is an open-source platform for building, shipping, and running containers. It simplifies the creation of container images and provides tools to manage container lifecycles.

Why Docker?

- Industry standard in container tooling
- Rich CLI and API support
- Docker Compose for multi-container apps
- Broad adoption and ecosystem (Docker Hub, extensions)

1.8 Docker Architecture

Docker uses a client-server model:

- Docker Client (CLI): Used by users to issue commands
- Docker Daemon (`dockerd`): Runs in the background, builds/runs containers
- Docker Images: Stored locally or in registries
- Docker Registries: Used to distribute and pull/push images
- Docker Engine: The complete platform including CLI, daemon, and tooling

Diagram (Text-based):

```
User --> Docker CLI --> Docker Daemon --> Container Runtime

                                 |

                                 +--> Images

                                 +--> Containers

                                 +--> Registries
```

1.9 Lab: Installing Docker on Ubuntu 24.04

This lab guides you through installing Docker on your lab VM (master1) running Ubuntu 24.04.

Step 1: Login to student on master1, with the password student.

```
Ubuntu 24.04.2 LTS master1 ttyS0
master1 login: labuser
Password: password-that-you-received
```

Step 2: Update package index and install the docker.io package

```
sudo apt update

sudo apt install -y docker.io
```

Step 3: Start and Enable Docker

```
sudo systemctl enable docker --now

systemctl status docker
```

Step 4: Run Docker Without sudo

To avoid typing sudo before every Docker command

```
sudo usermod -aG docker $USER
```

Then log out and back in

```
docker version
```

Knowledge Check

1. How do containers differ from virtual machines?
2. What are the key components of a container architecture?
3. What is the purpose of a container image layer?

Summary

In this chapter, we covered:

- The evolution from virtualization to containers
- Benefits and architecture of containers
- Image layering and the container runtime
- Docker as a powerful container tool
- Hands-on installation of Docker on Ubuntu

Chapter 2: Docker Architecture

This chapter provides a technical overview of Docker's architecture and its key components. It includes how Docker interacts with the Linux kernel, the role of container runtimes, and the underlying technologies that make containers possible.

2.1 Docker and the Linux Kernel

Docker relies on features provided by the Linux kernel to implement process and resource isolation without the need for full virtualization.

Key kernel features include:

- Namespaces — for isolation
- Control Groups (cgroups) — for resource limits
- Union File Systems — for layer-based storage

Docker does not reinvent container technology but provides a consistent interface and toolchain for leveraging these capabilities.

2.2 Docker and Linux Kernel Namespaces

Namespaces isolate system resources between containers. Each container sees only its own isolated view of the system.

Namespace	Isolates
`pid`	Process IDs (each container has its own PID tree)
`net`	Network interfaces, IP addresses, routing tables
`mnt`	Mount points and filesystems
`ipc`	Inter-process communication (e.g., shared memory)
`uts`	Hostname and domain name

`user`	User and group IDs
`cgroup`	Access to control groups

Each container has its own instance of these namespaces, providing process-level isolation from the host and other containers.

2.3 Docker and Linux Kernel Control Groups (cgroups)

Control Groups (cgroups) allow Docker to limit and monitor container resource usage:

Resource	Example Configuration
CPU	Limit CPU shares or cores
Memory	Restrict RAM usage
Disk I/O	Limit block device throughput
Network	Can be shaped using tc/iptables
PIDs	Limit the number of processes

Docker automatically applies cgroup policies using flags such as `--memory`, `--cpus`, etc.

Example:

```
docker run --memory=512m --cpus=1 nginx
```

2.4 Docker and Union File Systems

Docker uses a union filesystem (UnionFS) to manage image layers. Each container image is built from a stack of read-only layers, combined at runtime with a writable container layer.

Popular UnionFS backends include:

- OverlayFS (default in modern Linux)

- AUFS (deprecated)
- Btrfs
- ZFS

UnionFS enables fast builds, image reuse, and reduced storage requirements.

Example Workflow:

- `docker build` creates layers
- `docker run` adds a writable container layer
- Changes are lost when the container is removed (unless persisted via volumes)

2.5 Docker Architecture

Docker is composed of several components organized in a client-server model:

```
+------------------------+
|     Docker CLI (User)   |
+------------------------+
            ↓
+------------------------+
|     Docker Daemon       |
|  (docker-engine/dockerd)|
+------------------------+
            ↓
+------------------------+
| Container Runtime (runc)|
+------------------------+
            ↓
+------------------------+
| Linux Kernel (cgroups,  |
| namespaces, OverlayFS)  |
+------------------------+
```

2.6 Docker Components

Component	Description
Docker CLI	User interface for interacting with the Docker daemon

Docker Daemon	Core service that manages containers, images, and volumes
containerd	Container runtime manager used by Docker
runc	Low-level runtime that actually creates container processes
Docker Engine	Entire platform including CLI, daemon, runtime stack
Docker Registry	Stores and distributes container images (e.g., Docker Hub)

Summary

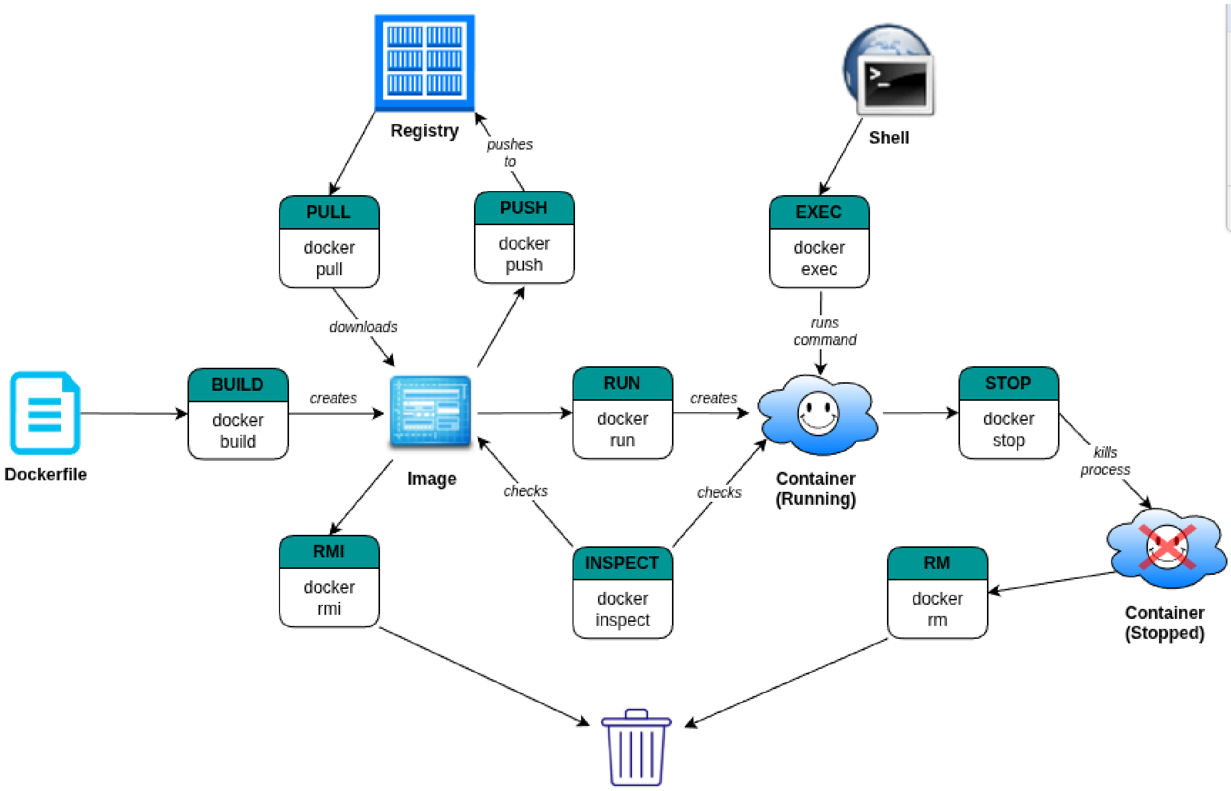
In this chapter, you’ve learned:

- Docker relies on core Linux kernel features to isolate and manage containers
- Namespaces, cgroups, and UnionFS are foundational to container behavior
- Docker’s architecture follows a modular client-server model
- containerd and runc are essential runtime components used internally

Chapter 3: Docker Container Lifecycle and CLI Usage

3.1 Understanding the Docker Container Lifecycle

To effectively manage containers, it’s important to understand the **lifecycle of a containerized application** — from building the image, running and managing the container, to cleaning up when it's no longer needed. The diagram below provides a high-level view of the key stages and Docker commands involved throughout this lifecycle. Each command corresponds to a specific phase in working with containers, whether you're preparing an image, deploying a container, or managing its runtime behavior and storage.



Each component in this lifecycle plays a role in managing containers:

Sub-command	Syntax Example	Description
`build`	`docker build -t <image-name>:<tag> <build-context>`	Builds a Docker image from a Dockerfile.
`pull`	`docker pull <registry>/<image>:<tag>`	Retrieves an image from a remote registry.
`push`	`docker push <registry>/<image>:<tag>`	Uploads an image to a registry for sharing or reuse.
`run`	`docker run [-d] [-p SRC:DST] [-v SRC:DST] [-i] [-t] [--name NAME] <image> [CMD]`	Creates and starts a new container from a specified image.
`exec`	`docker exec -it <container-name> <command>`	Executes a command inside a running container.
`stop`	`docker stop <container-name>`	Gracefully stops a running container.
`rm`	`docker rm <container-name>`	Removes a stopped container.
`rmi`	`docker rmi <image-name>`	Deletes an image from local storage.
`inspect`	`docker inspect <container-name or image-name>`	Displays low-level details about a container or image (e.g., config, IP).

3.2 Lab: Basic Docker Workflow

In this lab, we will walk through the basic commands used in the Docker container lifecycle. The primary objectives are:

- To create an image that we will upload to our own registry and reuse later.
- To understand the relationship between containers, images, and registries.
- To become familiar with Docker CLI commands used throughout the container lifecycle.

1. Search for an image

Let’s begin by locating a base image from Red Hat’s registry. The UBI (Universal Base Image) is a lightweight and redistributable starting point for container development.

```
docker search registry.access.redhat.com/ubi8
```

2. Pull the image

After identifying the correct image, pull it locally:

```
docker pull registry.access.redhat.com/ubi8/ubi
```

3. List local images

Verify the image has been successfully downloaded:

```
docker images
```

4. Inspect the image

Examine metadata and configuration of the pulled image:

```
docker inspect registry.access.redhat.com/ubi8/ubi:latest
```

5. Run a container

Start a container from the image without assigning a name:

```
docker run registry.access.redhat.com/ubi8/ubi:latest
```

6. Check for running containers

Check the list of currently active containers:

```
docker ps
```

❓ **Quiz**: Why do you think the container is not listed?

7. List all containers

This will include stopped containers:

```
docker ps -a
```

8. Create and configure a container with Apache web server

We’ll now create a container interactively, install the Apache HTTP server inside it, and set up a basic web page.

```
docker run --name source -it registry.access.redhat.com/ubi8/ubi:latest
/bin/bash
```

Inside the container:

```
yum install -y httpd --nodocs
yum clean all
echo Hello World > /var/www/html/index.html
exit
```

9. Check and verify that the container is there

The name or ID of the container will be used in Step 10.

```
docker ps -a
```

10. Commit the container as a new image

Now save this container state as a new image that we can reuse or share with others. Use the CONTAINER_NAME that was displayed in Step 9 above.

```
docker commit -a "Lab User" -c 'ENTRYPOINT httpd -DFOREGROUND' CONTAINER_NAME
REG_ID/myweb:1.0
```

11. Verify image creation

Use the following command to confirm that the new image has been created and tagged.

```
docker images
```

12. Inspect your custom image

Check how many layers and what metadata your image has:

```
docker inspect REG_ID/myweb:1.0
```

13. Attempt to push to Docker Hub

Try pushing the image to your Docker Hub registry. This might fail if you're not logged in yet.

```
docker push REG_ID/myweb:1.0
```

❓**Quiz**: Why do you think this push might fail?

14. Log in to Docker Hub

If the previous step failed due to authentication, log in using your Docker Hub credentials:

```
docker login -u REG_ID
```

15. Push the image to your registry

Once authenticated, push the image again:

```
docker push REG_ID/myweb:1.0
```

16. Open your browser and login to hub.docker.com to check if the image was uploaded.

16. Remove the image from your local system

Now that you have your image backed up in your registry, remove it from the local system:

```
docker rmi REG_ID/myweb:1.0
docker images
```

3.3 Networking and Port Mapping

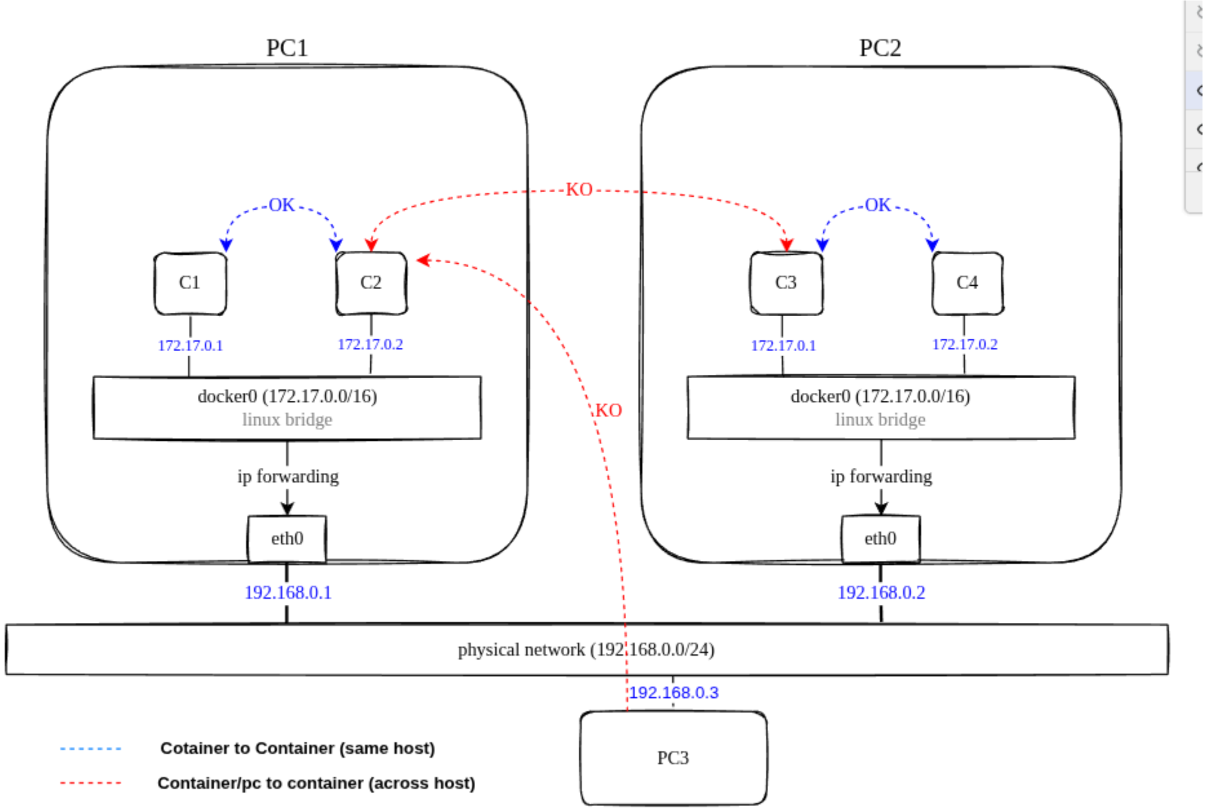


Diagram: Docker Bridge

In Docker, the default network mode creates a **software bridge** called `docker0`, which is automatically set up during installation. This bridge is **local to each host** and allows containers on the same machine to communicate with each other using internal IP addresses from a default subnet (usually `172.17.0.0/16`). Since each Docker host independently creates its own bridge with the same default subnet, containers on different hosts may appear to be on the same network — but they are **not actually connected**.

Because this default bridge network is **private and isolated per host**, containers on one machine cannot directly communicate with containers on another machine using their internal IPs. There's **no built-in routing** between these local bridges across different hosts. However, containers can still access external networks, including the internet, as long as the host itself has connectivity. This is because Docker sets up IP forwarding and NAT on the host, allowing outbound traffic.

To enable cross-host container communication, you would need to expose ports (`-p`), use a custom bridge with appropriate routing, or adopt a higher-level networking solution like Docker Swarm or Kubernetes overlay networks.

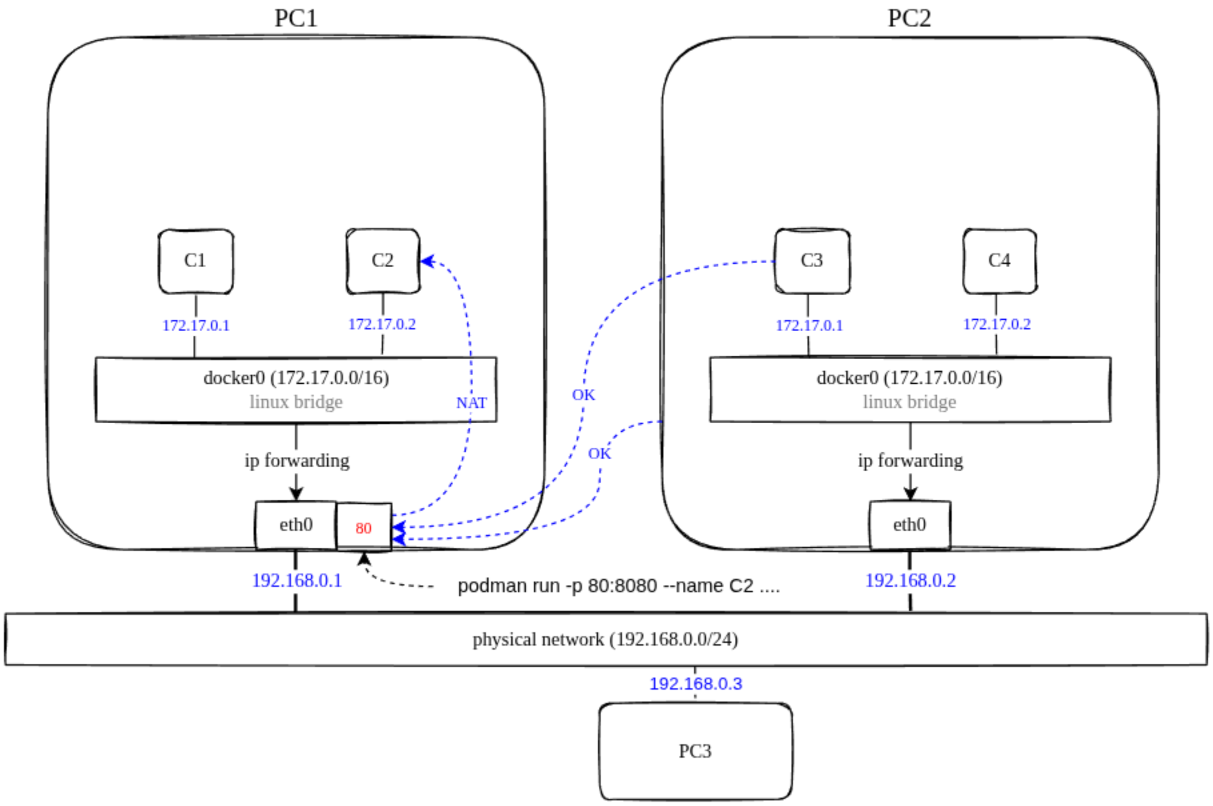


Diagram: Port Mapping

Port Mapping is the mechanism Docker provides to expose container services to the outside world, including access from the host machine or other computers. This is done by mapping a host port to a container port using the `-p` option in the `docker run` command:

```
docker run -d -p <host-port>:<container-port> <image>
```

Examples:

- Exposes port 80 inside the container to port 8080 on the host.

  ```
  docker run -d -p 8080:80 REG_ID/myweb:1.0
  ```

- Exposes the container port only on localhost.

  ```
  docker run -d -p 127.0.0.1:8080:80 REG_ID/myweb:1.0
  ```

Common Use Cases:

- Hosting web services on custom ports.
- Running multiple containers on the same host with different external ports.
- Debugging applications locally while testing their internal networking.

3.4 Lab: Test Port Mapping with Pulled Image

Let’s now test the image we pushed earlier to Docker Hub by pulling it back and exposing it via port mapping.

1. **Pull your own image from the registry**

```
docker images
docker pull REG_ID/myweb:1.0
docker images
```

2. **Run the container with port mapping**
   Map container port 80 (where Apache serves the website) to port 12345 on the host.

```
docker run -d -p 12345:80 --name test_web REG_ID/myweb:1.0
```

3. **Inspect and test**
   First, verify the IP and port configuration:

```
docker inspect test_web
```

   Then, access the website using curl from your host:

```
curl localhost:12345
```

   If you see the "Hello World" response, the container is running and the port mapping works.

3.5 Volume Mappings

When working with containers, data stored inside the container is ephemeral — meaning it is lost once the container stops or is removed. To persist data beyond the lifecycle of a container, **Docker provides volume mapping** mechanisms that allow the container to store or access data on the host system or managed volumes.

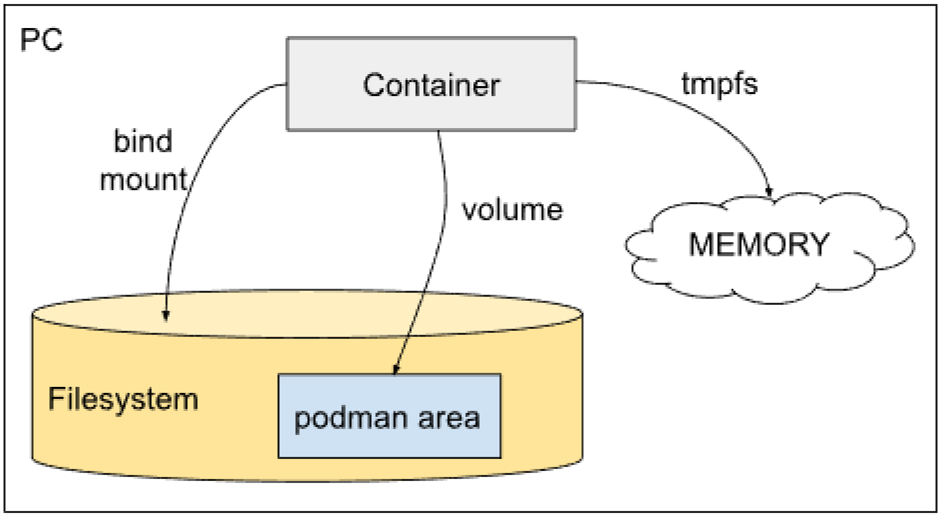


Diagram 1: bind mount vs volume vs tmpfs mount

There are **three** primary types of **volume mappings** in Docker:

Bind Mounts

Bind mounts allow you to map a specific file or directory on the **host filesystem** into the container. These are useful for development, testing, or sharing configuration/data between the host and container.

```
docker run -v /host/path:/container/path <image>
```

Example:

```
docker run -v /home/user/app:/usr/share/nginx/html nginx
```

This maps the host's `/home/user/app` directory to the NGINX container's web root.

Named Volumes (Managed by Docker)

Docker volumes are managed storage locations created under `/var/lib/docker/volumes/`. These are decoupled from the host path and more portable and consistent across environments. They are especially useful for production environments.

To create and use a named volume:

```
docker volume create myvolume
docker run -v myvolume:/container/path <image>
```

This ensures that the data is persistent and managed independently of the host's file structure.

tmpfs Mounts

`tmpfs` mounts are used when you need **temporary in-memory storage** that does not persist on disk. Data written to a tmpfs volume disappears when the container stops. This is ideal for sensitive data or runtime caches.

```
docker run --tmpfs /container/path <image>
```

Each type has its purpose:

- Use **bind mounts** for fast iteration and testing.
- Use **named volumes** for durability and portability.
- Use **tmpfs** for volatile, secure, in-memory operations.

Understanding these mapping options is critical for managing stateful applications and ensuring data durability in containerized environments.

3.6 Lab: Volume Mapping with Custom Web Content

Let’s put this knowledge into practice by using a **bind mount** to serve custom content through an existing image.

1. **Create a directory and content on the host**

   This creates a basic HTML file that we’ll serve from within the container.

   ```
   mkdir ~/abc
   echo 'Welcome to abc website' > ~/abc/index.html
   ```

2. **Run the container with volume mapping**

   ```
   docker run -d -p 12346:80 --name abc_web -v ~/abc:/var/www/html
   REG_ID/myweb:1.0
   ```

   Here we:

   - Map `~/abc` (host) → `/var/www/html` (container)
   - So the container serves our custom `index.html` page instead of the default one.

3. **Test using curl**

   ```
   curl localhost:12346
   ```

   You should see the message:

   ```
   Welcome to abc website
   ```

   This confirms that the bind mount works and the container is serving host-managed content.

Chapter 4: Building Container Images with Dockerfile

4.1 Understanding Dockerfile and Image Construction

In Docker, while it's possible to create an image using `docker commit`, the preferred, reproducible, and professional method for creating container images is through a **Dockerfile**.

A **Dockerfile** is a plain-text script that contains instructions to assemble a Docker image. Each instruction adds a layer on top of the previous one, forming a stack that defines everything needed for the container to run — including the base image, installed packages, copied files, exposed ports, default commands, and more.

🔹 Why use a Dockerfile?

- **Repeatability**: You can recreate the exact same image every time.
- **Version control**: Dockerfiles can be stored in Git and audited like code.
- **Portability**: Shared with others, allowing consistent build across teams.
- **Automation**: Easily integrated into CI/CD pipelines.

📦 Structure of a Dockerfile

A Dockerfile consists of a series of **instructions**, each specifying an action in the image build process. Below are the most common Dockerfile commands:

Instruction	Description
`FROM`	Sets the base image for the container. This must be the first instruction.
`LABEL`	Adds metadata to the image (e.g., author, description).
`RUN`	Executes commands in a new layer during build (commonly used for installing packages).
`COPY`	Copies files/directories from host to the container image.
`ADD`	Like `COPY` but with additional features (e.g., extracting archives).
`EXPOSE`	Documents which ports the container will listen on at runtime.
`CMD`	Sets the default command to run when the container starts (can be overridden).

`ENTRYPOINT`	Sets the command that will always run (often used for foreground daemons).
`VOLUME`	Declares a mount point for externally stored data.
`ENV`	Sets environment variables inside the container.
`WORKDIR`	Sets the working directory inside the container for subsequent instructions.

🔍 Key Differences between CMD and ENTRYPOINT

- **CMD** is overridden if the user provides a command in `docker run`.
- **ENTRYPOINT** is not overridden and is often used with `CMD` to pass default arguments.

4.2 Lab: Building a PHP Runtime Image Using Dockerfile

In this lab, we will build a custom Docker image to serve PHP-based websites using **Apache** and **PHP-FPM**. This complements the earlier Apache-only container we manually created in Chapter 3.

A key difference in this lab is the use of a **Dockerfile**, which allows us to automate and document the image creation process — a preferred practice in modern infrastructure automation.

> 🔧 **Note**: Your lab environment has already been prepared. The Dockerfile and associated web content are available in the `~/myphp` directory on your worker node.

📝 Step 1: Review the Dockerfile

Navigate to the working directory and review the provided Dockerfile:

```
cd ~/myphp
cat Dockerfile
```

Dockerfile contents:

```
FROM registry.access.redhat.com/ubi8/ubi:latest

LABEL maintainer="labuser@example.com"

EXPOSE 8080

ENV MSG="Containerfile"

RUN yum install -y httpd php-fpm; \
    yum clean all; \
    sed -i 's/^Listen 80 *$/Listen 8080/' /etc/httpd/conf/httpd.conf; \
    sed -i 's/^;clear_env/clear_env/' /etc/php-fpm.d/www.conf; \
    mkdir /run/php-fpm; \
    chgrp -R 0 /var/log/httpd /var/run/httpd /run/php-fpm; \
    chmod -R g=u /var/log/httpd /var/run/httpd /run/php-fpm

ADD src/* /var/www/html/

USER root

ENTRYPOINT php-fpm & httpd -DFOREGROUND
```

✅ **Explanation Highlights**:

- Uses UBI 8 as a secure base.
- Installs both web server and PHP runtime.
- Exposes port 8080 and runs services in the foreground.
- Prepares configuration and file permissions to work well in containers.

🏗️ Step 2: Build the Image

```
docker build -t yourid/myphp:1.0 .
```

This will create your new image with the tag `1.0`.

🔍 Step 3: Inspect the Image

```
docker inspect yourid/myphp:1.0
```

Use this to explore image layers, exposed ports, environment variables, and entrypoints.

❓ Quiz: Is it good to have many layers?

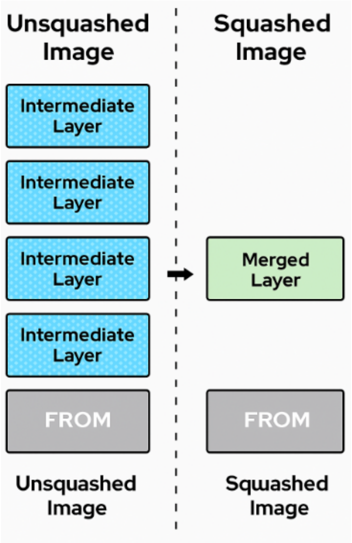
Discuss:

- ✅ **Pros**: Efficient caching during rebuilds, better traceability.

- ❌ **Cons**: Potential for larger image sizes, redundant file system overhead.

💡 Extra Tip: Squashing Layers for Optimization

In some cases, you may want to **flatten** your image by combining all layers into one to reduce image size or simplify deployments.



🔧 Extra 1: Enable Docker Experimental Feature

Edit the Docker daemon config to enable experimental features:

```
sudo vi /etc/docker/daemon.json
```

Add this content:

```
{
    "experimental": true
}
```

Restart the Docker daemon:

```
sudo systemctl restart docker
```

🔄 Extra 2: Build with Squash Enabled

Now rebuild the image with the `--squash` flag:

```
docker build -t yourid/myphp:2.0 --squash .
```

Inspect to verify reduced layers:

```
docker inspect yourid/myphp:2.0 | tail
```

Run and test the new squashed image:

```
docker run -d -p 12347:8080 --name php_test yourid/myphp:2.0
curl localhost:12347
```

🧩 Challenge Activity

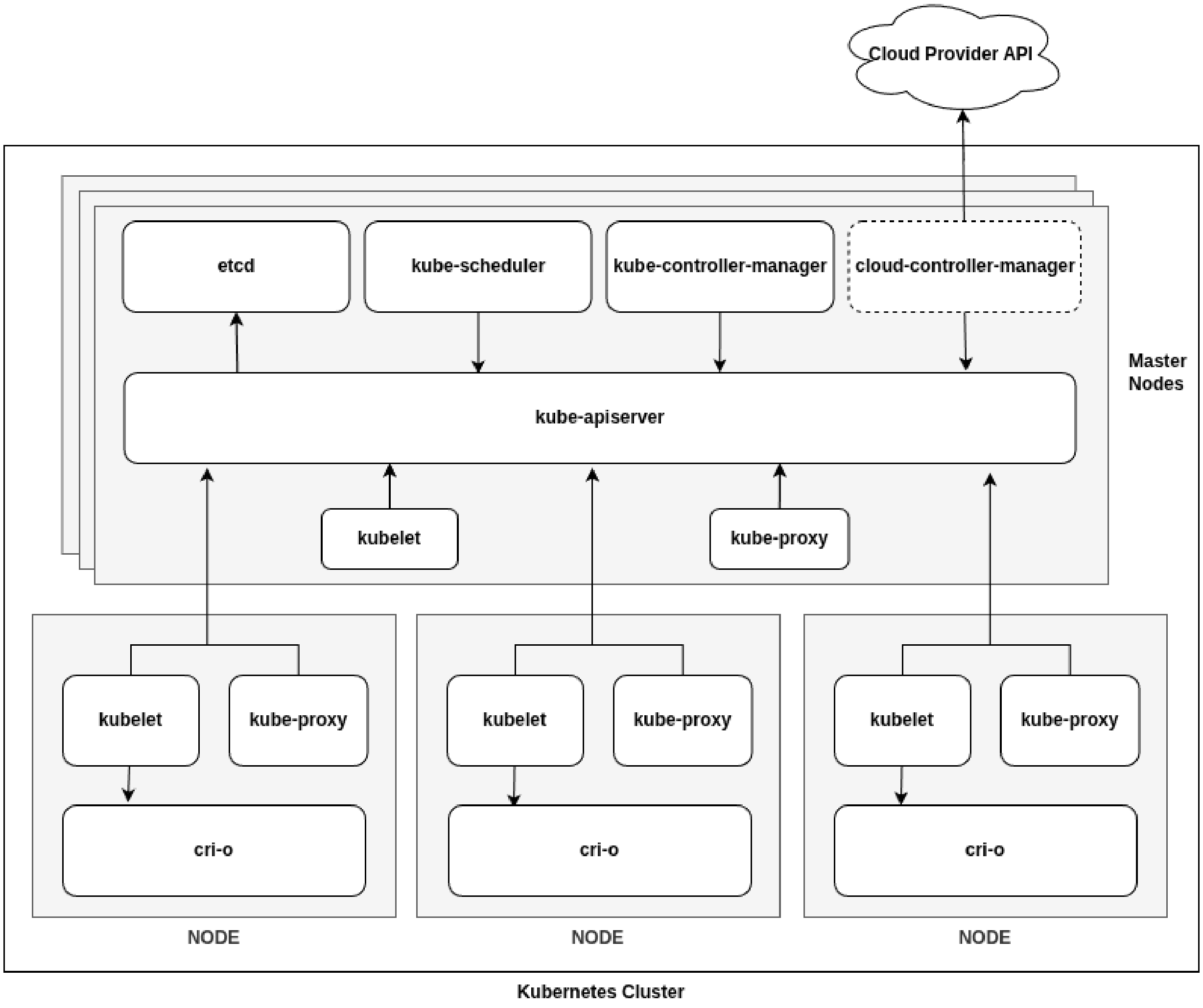
Try deploying the contents of https://github.com/kelvinlnx/myphp using the newly built `myphp:2.0` image.

Question:
What approach will you use to serve those files without rebuilding the image?
(Hint: Consider volume mounting.)

Chapter 5: Kubernetes Architecture and Installation Basics

5.1 Kubernetes Architecture Overview

Kubernetes is a powerful, open-source container orchestration system designed to automate the deployment, scaling, and management of containerized applications. It provides a consistent framework to run distributed systems resiliently across clusters of machines, whether on-premises or in the cloud.



At its core, Kubernetes operates as a **cluster** consisting of two main types of components:

5.1.1. Control Plane (Master Nodes)

The control plane is responsible for the **global decisions** about the cluster (e.g., scheduling, managing the desired state, responding to events).

- **kube-apiserver**
  Acts as the **front door** to the cluster. All internal and external communications with Kubernetes go through the API server. It validates requests and updates the cluster state.

- **etcd**
  A distributed **key-value store** used as the cluster's **source of truth** for configuration and state.

- **kube-scheduler**
  Assigns newly created Pods to available Nodes based on resource requirements, taints/tolerations, affinity rules, and policies.

- **kube-controller-manager**
  Runs various background **controllers** (e.g., replication controller, endpoint controller, namespace controller) to ensure the desired state is constantly maintained.

- **cloud-controller-manager** *(optional)*
  Integrates the cluster with cloud-specific APIs to manage resources like load balancers, storage volumes, or routes.

5.1.2. Worker Nodes (Minions)

Worker nodes run the **actual workloads** (containers inside Pods). Each node hosts the following components:

- **kubelet**
  The **node agent** that communicates with the API server and ensures the containers are running as instructed.

- **kube-proxy**
  Manages **networking** and implements the rules for forwarding traffic to the correct Pods.

- **Container runtime (e.g., cri-o, containerd)**
  The engine responsible for running the containers. Kubernetes supports pluggable runtimes via the Container Runtime Interface (CRI).

5.2 Installing Kubernetes

In this section, we summarize the main steps required to set up a Kubernetes cluster manually, focusing on understanding the key configuration and orchestration pieces.

> ⚙️ For our lab setup, the cluster installation is driven by an Ansible-based playbook (`class.yaml`) that automates these steps. However, the following highlights the conceptual flow behind what's being configured.

Installation Workflow Overview

1. **Prepare the Hosts**
   - Set hostnames and `/etc/hosts` entries.
   - Enable IP forwarding and required kernel modules.
   - Disable swap as Kubernetes requires it to be off.

2. **Install CRI (Container Runtime Interface)**
   - Install and configure **CRI-O** as the container runtime.
   - Configure `crio.conf` and ensure the CNI (Container Network Interface) plugins are present.

3. **Install Kubernetes Binaries**
   - Install `kubeadm`, `kubelet`, and `kubectl` on all nodes.
   - Enable and start `kubelet`.

4. **Initialize the Control Plane (Master Node)**
   - Use `kubeadm init` with a customized configuration file (from `class.yaml`) to:
     - Set pod subnet range.
     - Choose CRI socket.
     - Enable necessary APIs.

5. **Configure kubectl**
   - Set up `~/.kube/config` for the admin user to interact with the API server via `kubectl`.

6. **Deploy Pod Network**
   - Deploy the SDN (e.g., Flannel, Calico, or Cilium) to enable inter-pod networking.

7. **Join Worker Nodes**

    - Use the `kubeadm join` token command generated during `init` to add nodes to the cluster.

8. **Post-Setup Tasks**
    - Install useful add-ons like:
        - Metrics server
        - Ingress controller
        - Kubernetes dashboard (optional)
        - Role-Based Access Control (RBAC) for lab users

> ✅ **Tip**: While `kubeadm` makes installation easier, it still requires configuration files and proper understanding of components. Tools like **RKE**, **k3s**, or managed Kubernetes services (e.g., **GKE**, **EKS**, **OpenShift**) abstract even more of the setup process and are commonly used in production environments.

📚 **Reference**:
For a deeper dive into `kubeadm` installation, visit the official documentation:
👉 https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/install-kubeadm/

5.3 Post-install Operations

Once your Kubernetes cluster is installed, there are several essential steps you should take to ensure it is functioning properly and ready for workloads.

5.3.1. Check Node Status

Run the following command to ensure that all nodes are registered and in the "Ready" state:

```
kubectl get nodes
```

This confirms that both control plane and worker nodes are communicating correctly with the API server.

5.3.2. Check System Pods

Verify that the default system components are running:

```
kubectl get pods -A
```

This lists pods across all namespaces. You should see components such as:

- `kube-apiserver`
- `kube-controller-manager`
- `kube-scheduler`
- `etcd`
- `kube-proxy`
- `coredns`

All should show a status of `Running`.

5.3.3. Test Network and DNS

Create a simple pod and verify connectivity and DNS resolution within the cluster:

```
kubectl run testpod --image=busybox --command -- sleep 3600
kubectl exec -it testpod -- nslookup kubernetes
```

You should see a DNS response for the `kubernetes` service — this indicates that CoreDNS is functioning properly.

5.3.4. (Optional) Save Your `kubeconfig`

Ensure you save the `admin.conf` file, which contains cluster credentials and connection details. This step is optional for our lab because it has already been done. You can copy it to the default location for kubectl:

```
mkdir -p ~/.kube
cp /etc/kubernetes/admin.conf ~/.kube/config
```

This lets you run `kubectl` without specifying the `--kubeconfig` flag.

5.3.5. (Optional) Install a Pod Network Add-on

If your installation didn't already include a CNI plugin, install one (e.g., Calico, Flannel, Cilium):

```
# Example: install Calico
kubectl apply -f
https://raw.githubusercontent.com/projectcalico/calico/v3.26.1/manifests/calic
o.yaml
```

✅ **Tip**: A Kubernetes cluster is technically running once kubeadm finishes, but the network and DNS plugins must be properly installed and healthy for real workloads to function correctly.

5.4 Kubernetes Resource Types

Kubernetes uses **declarative objects** to define the desired state of the system. These objects, or *resources*, are submitted to the cluster using manifests (usually YAML or JSON) and managed by the Kubernetes control plane.

Below is a high-level categorization of key Kubernetes resource types:

5.4.1. Workload Resources

These define applications running in containers and how they are deployed and scaled.

Resource	Description
Pod	The smallest deployable unit. Wraps one or more containers.
ReplicaSet	Ensures a specified number of pod replicas are running at any time.
Deployment	Manages updates and rollbacks for ReplicaSets.
StatefulSet	Like Deployment, but maintains persistent identity for each pod.
DaemonSet	Ensures one pod runs on each (or selected) node, often for monitoring.
Job	Creates one or more pods to run a task and then terminates.
CronJob	Runs Jobs on a scheduled time (like a cron job).

5.4.2. Service Resources

These expose workloads to other services or the outside world and manage communication.

Resource	Description

Service	A stable network endpoint for a set of pods.
Ingress	Manages external access (HTTP/HTTPS) to services using rules.
Endpoint	Internal representation of the IPs that back a service.
NetworkPolicy	Controls the network flow between pods (like firewall rules).

5.4.3. Configuration Resources

Used to decouple configuration from application code.

Resource	Description
ConfigMap	Stores configuration data as key-value pairs. Injected into containers.
Secret	Stores sensitive data (e.g., passwords, tokens) in base64 format.
PersistentVolumeClaim (PVC)	Requests storage from the cluster.
PersistentVolume (PV)	Represents the actual storage provisioned.
StorageClass	Defines how storage is provisioned dynamically.

5.4.4. Cluster Resources

Control authentication, authorization, namespaces, and policies.

Resource	Description
Namespace	Provides logical separation between different projects/environments.
Role/RoleBinding	Defines permissions within a namespace.
ClusterRole/ ClusterRoleBinding	Grants permissions across the entire cluster.

ResourceQuota	Limits resource consumption per namespace.
LimitRange	Sets default or max values for compute resources in pods.

📌 Summary

Kubernetes provides a rich set of resource types to:

- Deploy and scale containerized workloads
- Secure and expose applications
- Manage configuration and storage
- Control user access and cluster policies

Understanding these core resource types is essential for working effectively with Kubernetes.

Chapter 6. Working with Kubernetes Objects

6.1 Declarative vs Imperative Approach

When working with Kubernetes, you have two primary ways of interacting with the cluster to manage resources:

6.1.1 Imperative Approach

This is where you tell Kubernetes *what to do*, step by step, using direct commands. It's useful for quick testing, debugging, and one-off tasks.

✅ **Characteristics:**

- Fast and ad hoc
- Not easily repeatable or version-controlled
- Executes immediately without storing a record of the configuration

🛠️ **Example Commands:**

```
kubectl run nginx --image=nginx
kubectl expose pod nginx --port=80 --target-port=80 --type=ClusterIP
kubectl delete pod nginx
```

6.1.2 Declarative Approach

In this method, you define the *desired state* of your application in a configuration file (typically YAML) and submit it to the Kubernetes API.

✅ **Characteristics:**

- Version-controllable (can be stored in Git)
- Repeatable and reproducible
- Encourages collaboration and CI/CD automation
- Suitable for complex, production-grade environments

🛠️ **Example Commands:**

```
kubectl apply -f nginx-deployment.yaml
kubectl delete -f nginx-deployment.yaml
```

6.1.3 Summary Table

Aspect	Imperative	Declarative
How	Direct commands	Config file describes desired state
Use Case	Quick fixes, debugging	Production, repeatable deployments
Repeatable	❌ Not easily repeatable	✅ Easily repeatable via YAML
Version Control	❌ No	✅ Yes
Automation Friendly	❌ No	✅ Yes

6.2 Using `kubectl` to Manage Kubernetes Objects

The `kubectl` command-line tool is your primary interface for interacting with a Kubernetes cluster. Whether you're inspecting the state of the system, deploying new resources, or troubleshooting issues — `kubectl` is the go-to tool.

6.2.1 Viewing Resources

📦 Get a List of Resources

Use `kubectl get` to retrieve basic information about resources in your cluster.

```
kubectl get pods
kubectl get services
kubectl get deployments
```

📖 View Detailed Description

`kubectl describe` provides extended information such as events, configuration, and status.

```
kubectl describe pod <pod-name>
kubectl describe service <service-name>
```

🧾 View Application Logs

Access logs from a container inside a pod using:

```
kubectl logs <pod-name>
kubectl logs <pod-name> -c <container-name>  # if multiple containers exist
```

6.2.2 Creating Resources from YAML Files

The declarative approach uses `kubectl apply -f` to create or update resources based on YAML files.

```
kubectl apply -f deployment.yaml
kubectl apply -f service.yaml
```

This command will:

- Create the object if it does not exist
- Update the object if it already exists and the configuration has changed

6.2.3 Editing and Deleting Resources

✏️ Edit Resources Inline

Make live changes to objects by launching the default editor:

```
kubectl edit deployment <deployment-name>
```

🗑️ Delete Resources

Remove an object from the cluster:

```
kubectl delete pod <pod-name>
kubectl delete -f deployment.yaml
```

You can delete entire types of resources using selectors:

```
kubectl delete pods -l app=nginx
```

6.2.4 Summary Table: Common `kubectl` Commands

Command	Syntax Example	Purpose
View resources	`kubectl get pods`	List pods in the current namespace
	`kubectl get services`	List services
	`kubectl get deployments`	List deployments
Detailed resource info	`kubectl describe pod nginx`	Show detailed info and events for a pod
View logs	`kubectl logs nginx-pod`	Show logs for a container in a pod
	`kubectl logs nginx-pod -c app-container`	Show logs from a specific container
Apply YAML (create/update)	`kubectl apply -f deployment.yaml`	Create or update resources from a YAML file
Edit live resource	`kubectl edit deployment nginx-deployment`	Open resource in editor for on-the-fly changes
Delete resource	`kubectl delete pod nginx-pod`	Delete a specific pod
Delete from YAML	`kubectl delete -f deployment.yaml`	Delete object defined in file
Delete by label	`kubectl delete pods -l app=nginx`	Delete all pods with a specific label

6.3 Object Metadata and Labels

Kubernetes resources contain **metadata** that helps identify and organize them. This metadata is especially important when managing large numbers of objects across namespaces and deployments.

6.3.1 Common Metadata Fields

Field	Description

`name`	Unique name of the object within a namespace
`namespace`	Logical grouping of objects (optional if using default namespace)
`labels`	Key-value pairs used for grouping and selecting objects
`annotations`	Key-value metadata not used for selection, often used for tooling/documentation
`uid`	Unique identifier assigned by Kubernetes (read-only)
`creationTimestamp`	Timestamp of when the object was created

6.3.2 Labels and Selectors

Labels are key-value pairs you can attach to any Kubernetes object. Unlike annotations, labels are **used for selection and filtering**.

```
metadata:
  name: nginx-pod
  labels:
    app: nginx
    env: dev
```

You can query objects by their labels using `kubectl`:

```
kubectl get pods -l app=nginx
kubectl get pods -l env=dev
kubectl get pods -l "env!=prod"
```

You can combine multiple label queries:

```
kubectl get pods -l app=nginx,env=dev
```

This helps when targeting specific sets of objects, for example during scaling, monitoring, or deleting resources.

6.3.3 Annotations

Annotations are also key-value pairs, but they are meant to **store non-identifying information**:

```
metadata:
  annotations:
    description: "This pod runs a development version of nginx"
    owner: "devops-team@example.com"
```

These are typically used by tools and automation, and are **not used for filtering**.

6.4 Namespaces and Resource Isolation

Kubernetes clusters are often shared by multiple users or applications. To prevent conflicts and allow logical separation, Kubernetes introduces the concept of **Namespaces**.

6.4.1 What is a Namespace?

A **Namespace** is a virtual cluster within a Kubernetes cluster. It allows grouping of resources under a unique context, providing isolation between teams, environments (like dev, test, prod), or applications.

Think of namespaces as folders that help organize and isolate objects like pods, services, and deployments.

6.4.2 Default Namespaces

Kubernetes includes several namespaces by default:

Namespace	Purpose
`default`	The default namespace for objects with no other namespace specified
`kube-system`	Reserved for Kubernetes system components like kube-dns, kube-proxy
`kube-public`	Readable by all users; used for public info like cluster configuration
`kube-node-lease`	Used for node heartbeats to determine node availability

6.4.3 Custom Namespaces

You can create your own namespaces to isolate environments or workloads.

```
kubectl create namespace myteam
```

Then, when you deploy an object, specify the namespace:

```
kubectl apply -f app.yaml -n myteam
```

Or include the namespace in your YAML definition:

```
metadata:
  name: myapp
  namespace: myteam
```

6.4.4 Switching Namespace Context

You can set a default namespace for your `kubectl` context to avoid repeating `-n <namespace>`:

```
kubectl config set-context --current --namespace=myteam
```

To check the current context and namespace:

```
kubectl config view --minify | grep namespace:
```

6.4.5 Viewing Resources in Namespaces

```
kubectl get pods -n kube-system       # View pods in kube-system namespace
kubectl get all --all-namespaces      # View all resources in all namespaces
```

6.4.6 When to Use Namespaces

Namespaces are especially useful when:

- Different teams share the same cluster.
- You want to isolate dev/test/prod environments.
- You want to apply quotas or policies to specific workloads.

6.5 Lab: Creating and Managing Kubernetes Objects with YAML

In this lab, you'll practice writing and deploying basic Kubernetes resource definitions using YAML. These exercises will help reinforce your understanding of Pods, Services, Deployments, Namespaces, ConfigMaps, and the usage of labels/selectors.

Lab Objectives

- Create and isolate resources in a custom namespace.
- Deploy a simple web server (nginx) using a Deployment and expose it with a Service.
- Inject configuration via a ConfigMap.
- Use labels to group and select objects.
- Practice deleting objects based on labels.

Step-by-Step Tasks

1. Create a Namespace

Namespace must be created first.

```
kubectl create namespace web-lab
```

2. Define and Deploy a Pod (nginx)

Save the following as `nginx-pod.yaml`:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
  namespace: web-lab
  labels:
    app: web
spec:
  containers:
    - name: nginx
      image: nginx
```

Apply it:

```
kubectl apply -f nginx-pod.yaml
```

3. Create a Deployment

Save as `nginx-deployment.yaml`:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  namespace: web-lab
spec:
  replicas: 2
  selector:
    matchLabels:
      app: web
  template:
    metadata:
      labels:
        app: web
    spec:
      containers:
        - name: nginx
          image: nginx
```

Apply:

```
kubectl apply -f nginx-deployment.yaml
```

4. Expose the Deployment with a Service

Save as `nginx-service.yaml`:

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
  namespace: web-lab
spec:
  selector:
    app: web
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
  type: NodePort
```

Apply:

```
kubectl apply -f nginx-service.yaml
```

Check assigned port:

```
kubectl get svc -n web-lab
```

Access it from the host machine:

```
curl <NodeIP>:<NodePort>
```

5. Create a ConfigMap

Save as `web-config.yaml`:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: web-config
  namespace: web-lab
data:
  welcome_msg: "Welcome to the Web Lab!"
```

Apply it:

```
kubectl apply -f web-config.yaml
```

You can then reference this ConfigMap in another pod or deployment as needed (e.g., via environment variables or mounted volumes).

6. Practice Label-Based Selection and Deletion

List resources by label:

```
kubectl get pods -n web-lab -l app=web
```

Delete all resources with label `app=web`:

```
kubectl delete all -l app=web -n web-lab
```

Chapter 7: Deploying and Exposing Applications in Kubernetes

🎯 Learning Objectives:

- Understand how Kubernetes Deployments manage pods and application replicas
- Learn how Services expose applications and manage access
- Apply labels and selectors for targeted management
- Practice deploying a basic application using YAML
- Perform updates with minimal downtime using rolling updates

7.1 Introduction to Deployments

A **Deployment** is a high-level Kubernetes object used to:

- Define a desired state for your application (e.g., 3 replicas of a web app)
- Automatically manage pod creation, scaling, and rolling updates
- Enable recovery from failures (e.g., rescheduling pods when nodes fail)

Key fields in a Deployment YAML:

- `replicas`: Number of desired pod instances
- `selector`: Label query to match pods
- `template`: Defines the pod spec (containers, volumes, etc.)

Example:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myweb-deploy
spec:
  replicas: 2
  selector:
    matchLabels:
      app: myweb
  template:
    metadata:
      labels:
        app: myweb
    spec:
      containers:
        - name: myweb
          image: yourid/myweb:1.0
          ports:
            - containerPort: 80
```

7.2 Exposing Applications with Services

A **Service** provides a stable IP and DNS name to access a group of pods, even if the actual pods are changing.

Types of Services:

- **ClusterIP** (default): Accessible only within the cluster
- **NodePort**: Exposes on a static port on each worker node
- **LoadBalancer**: Provisions an external IP via cloud provider (e.g., GCP, AWS)
- **ExternalName**: Maps to external DNS names

Example:

```
apiVersion: v1
kind: Service
metadata:
  name: myweb-service
spec:
  selector:
    app: myweb
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
  type: NodePort
```

7.3 Labels and Selectors in Practice

Labels are key-value pairs attached to Kubernetes objects (e.g., pods, services).

Selectors are used to:

- Connect Deployments to Pods
- Match Services with Pods
- Filter resources with `kubectl`

Examples:

```
kubectl get pods --selector app=myweb
kubectl delete pods -l app=myweb
```

7.4 Lab: Deploy and Expose a Web App

Objective: Create a Deployment and expose it using a NodePort Service.

Create a namespace

```
kubectl create ns webapp
kubectl config set-context --current --namespace=webapp
```

1. Deploy the application
Create `myweb-deploy.yaml`:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myweb-deploy
spec:
  replicas: 2
  selector:
    matchLabels:
      app: myweb
  template:
    metadata:
      labels:
        app: myweb
    spec:
      containers:
        - name: myweb
          image: yourid/myweb:1.0
          ports:
            - containerPort: 80
```

Apply it:

```
kubectl apply -f myweb-deploy.yaml
```

2. Create and apply a Service
Create `myweb-svc.yaml`:

```
apiVersion: v1
kind: Service
metadata:
  name: myweb-service
spec:
  selector:
    app: myweb
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
  type: NodePort
```

Apply it:

```
kubectl apply -f myweb-svc.yaml
```

3. Test the application

```
kubectl get svc
curl <NODE_IP>:<NODE_PORT>
```

4. Optional: List pods by label

```
kubectl get pods -l app=myweb
```

7.5 Bonus: Rolling Updates

Rolling updates allow seamless version updates without downtime.

Update the image version:

```
kubectl set image deployment/myweb-deploy myweb=yourid/myweb:2.0
```

Check rollout status:

```
kubectl rollout status deployment/myweb-deploy
```

Rollback if needed:

```
kubectl rollout undo deployment/myweb-deploy
```

Chapter 8: Kubernetes Addons

🎯 Learning Objectives

- Understand the concept and purpose of Kubernetes addons
- Learn how to use Helm as a package manager for Kubernetes
- Deploy common addons such as ingress controllers, dashboards, and metrics tools
- Improve cluster usability, observability, and extensibility

8.1 What are Kubernetes Addons?

Kubernetes provides a lean, modular core. Its capabilities can be extended with **addons**—optional components that enhance cluster functionality. These are often deployed as pods using YAML manifests or Helm charts.

Examples include:

- **Helm** – A package manager for Kubernetes
- **Metrics collectors** – For resource monitoring
- **Network plugins** (e.g., Calico, Flannel)
- **Storage solutions** (e.g., NFS, CSI plugins)
- **Ingress controllers** (e.g., NGINX, HAProxy, Traefik)
- **Monitoring tools** (e.g., Prometheus, Grafana)
- **Logging stacks** (e.g., Fluentd, ELK stack)
- **Dashboards** and **UI tools**
- **Security tools**, **cert managers**, and more

8.2 Helm – The Kubernetes Package Manager

Helm simplifies application deployment and management in Kubernetes. It uses versioned packages called **charts**, making it easier to install, upgrade, rollback, and share apps.

Why use Helm?

- Manage complex apps with a single command
- Customize deployments with configuration values
- Track and rollback changes
- Discover and reuse community-maintained charts

Helm Key Concepts

Term	Description
Chart	A package of YAML templates and metadata
Release	An instance of a deployed chart
Repository	A collection of charts, typically online
Values	Custom settings used to override default chart configs

8.3 Installing and Using Helm

Install Helm (Linux)

```
curl https://raw.githubusercontent.com/helm/helm/master/scripts/get-helm-3 |
bash
```

Verify Helm

```
helm version
```

Using Helm

Add a repository:

```
helm repo add bitnami https://charts.bitnami.com/bitnami
helm repo update
```

Install a chart:

```
helm install my-nginx bitnami/nginx
```

List Helm releases:

```
helm list
```

Uninstall a release:

```
helm uninstall my-nginx
```

8.4 Ingress Controller and Ingress Resource

In Kubernetes, **Services** (like NodePort or LoadBalancer) are used to expose applications. However, these methods can become inefficient when:

- You have many applications.
- You want to consolidate traffic through a single entry point.
- You need advanced traffic routing (e.g., path-based or host-based).

This is where **Ingress** and the **Ingress Controller** come in.

An **Ingress** is an API object that defines access rules (e.g., host/path) for external HTTP/S traffic to reach internal services.

An **Ingress Controller** is a special type of Kubernetes controller that listens for ingress resource definitions and handles the actual routing.

🧠 **Tip:** Ingress is not enabled by default. You need to install an ingress controller that suits your environment (NGINX, HAProxy, Traefik, etc.).

Ingress Architecture Overview

A typical ingress setup looks like this:

- External client sends a request to the Kubernetes cluster's ingress endpoint (e.g., cloud load balancer).
- The **Ingress Controller** receives the request.
- It matches the request against the defined **Ingress rules** (based on host or path).
- It routes the request to the correct **Kubernetes Service**, which forwards to the appropriate Pod.

📌 **Key Components:**

- **Ingress Resource**: Defines rules (host, path) for routing traffic.
- **Ingress Controller**: Watches ingress resources and configures a reverse proxy.
- **Service**: Internal routing target.
- **Pod**: Application instance.

To expose internal services externally, Kubernetes uses an **Ingress Controller**. It routes incoming HTTP/HTTPS traffic to the correct internal service based on rules defined in an **Ingress** resource.

8.5 Lab: Deploying NGINX Ingress Controller

You can deploy nginx controller by applying the resource definition located at https://raw.githubusercontent.com/kubernetes/ingress-nginx/controller-v1.9.4/deploy/static/provider/cloud/deploy.yaml, but in this Lab we will be deploying the controller using helm.

Step 1: Add and update helm repo

```
helm repo add ingress-nginx https://kubernetes.github.io/ingress-nginx
helm repo update
```

Step 2: Configure the nginx controller by creating the following file

```
cat <<EOF > nginx-hostnet.yaml
controller:
  hostNetwork: true       # Bind directly to host ports
  hostPort:
    enabled: true         # Open ports on the host
  kind: DaemonSet        # Ensures one pod per node
  service:
    enabled: false        # Disable NodePort/LoadBalancer services
  ingressClassResource:
    enabled: true
    name: nginx
    default: true
  ingressClass: nginx
  watchIngressWithoutClass: false  # Only watch Ingresses with class: nginx

  publishService:
    enabled: false        # Disable publishing a Service
  extraArgs:
    publish-status-address: "0.0.0.0"  # Allow external status checks
EOF
```

Step 3: Install nginx controller

```
helm upgrade --install ingress-nginx ingress-nginx/ingress-nginx \
  --namespace ingress-nginx \
  --create-namespace \
  -f nginx-hostnet.yaml
```

Step 4: Verify Deployment

```
kubectl get pods -n ingress-nginx -o wide
kubectl get svc -n ingress-nginx
```

8.6 Lab: Deploying Kubernetes Dashboard

The **Kubernetes Dashboard** is a web-based UI to manage cluster resources.

Step 1: Add and update helm repo

```
helm repo add kubernetes-dashboard https://kubernetes.github.io/dashboard/
helm repo list
```

Step 2: Install the dashboard

```
helm upgrade --install kubernetes-dashboard \
     kubernetes-dashboard/kubernetes-dashboard \
     --create-namespace --namespace kubernetes-dashboard
```

Step 3: Verify

```
kubectl get all -n kubernetes-dashboard
```

Step 3: Create a service account to manage cluster:

```
kubectl create serviceaccount dashboard-admin -n kubernetes-dashboard

kubectl create clusterrolebinding dashboard-admin \
  --clusterrole=cluster-admin \
  --serviceaccount=kubernetes-dashboard:dashboard-admin
```

Step 4: Create an ingress for the kubernetes-dashboard-kong-proxy

Make sure you replace the domain hostname with a resolvable name.

```
echo <<EOF > dashboard-ingress.yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: kubernetes-dashboard
  namespace: kubernetes-dashboard
  annotations:
    nginx.ingress.kubernetes.io/backend-protocol: "HTTPS"  # Dashboard uses HTTPS
    nginx.ingress.kubernetes.io/ssl-passthrough: "true"    # Bypass Nginx SSL termination
    nginx.ingress.kubernetes.io/rewrite-target: /          # Rewrite path if needed
spec:
  ingressClassName: nginx   # Matches the IngressClass you configured
  rules:
  - host: dashboard.example.com  # Replace with your domain (or use IP if no DNS)
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: kubernetes-dashboard-kong-proxy  # Dashboard Service name
            port:
              number: 443               # Dashboard Service port (HTTPS)
EOF

kubectl apply -f dashboard-ingress.yaml
kubectl get ingress -n kubernetes-dashboard
```

Step 5: Open your browser and browse to the hostname that you set in the dashboard-ingress.yaml file.

Get the login token:

```
kubectl -n kubernetes-dashboard create token dashboard-admin
```

8.6 Metrics Server

Metrics Server collects resource metrics (CPU, memory) used by components like the Horizontal Pod Autoscaler.

Tip: Verify that there are no metrics displayed in the dashboard or command line (kubectl top nodes/pods) before deploying metrics.

Step 1: Deploy Metrics Server via Helm

```
helm repo add metrics-server https://kubernetes-sigs.github.io/metrics-server/
helm repo update

helm upgrade --install metrics-server metrics-server/metrics-server \
  --create-namespace --namespace metrics-server
```

Step 2: Verify installation:

```
kubectl get deployment metrics-server -n kube-system
```

Step 3: Test.

Refresh your dashboard to see metrics being displayed now. You can also try the following commands

```
kubectl top nodes
kubectl top pods
```

8.7 Prometheus and Grafana (Monitoring Stack)

Prometheus is a powerful metrics collection system. **Grafana** provides dashboards for visualizing those metrics.

Install via Helm

```
helm repo add prometheus-community
https://prometheus-community.github.io/helm-charts
helm repo update

helm install kube-prometheus-stack prometheus-community/kube-prometheus-stack
\
  --namespace monitoring --create-namespace
```

Verify and Access Grafana

```
kubectl port-forward svc/kube-prometheus-stack-grafana -n monitoring 3000:80
```

Browse to: http://localhost:3000

Get the login password:

```
kubectl get secret -n monitoring kube-prometheus-stack-grafana -o
jsonpath="{.data.admin-password}" | base64 -d
```

✅ Summary

- Addons enhance Kubernetes with monitoring, networking, UI, security, and automation features

- Helm simplifies managing these components through reusable charts

- You deployed:

    - Helm
    - NGINX ingress controller
    - Kubernetes Dashboard
    - Metrics Server
    - Prometheus and Grafana monitoring stack