Containers - introduction

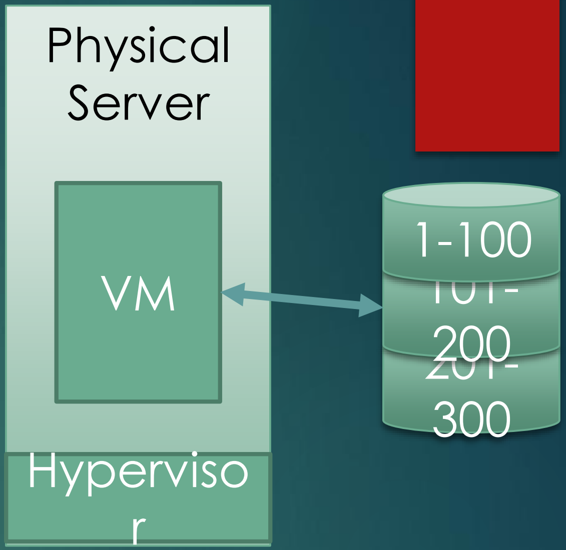
K V SUBRAMANIAM

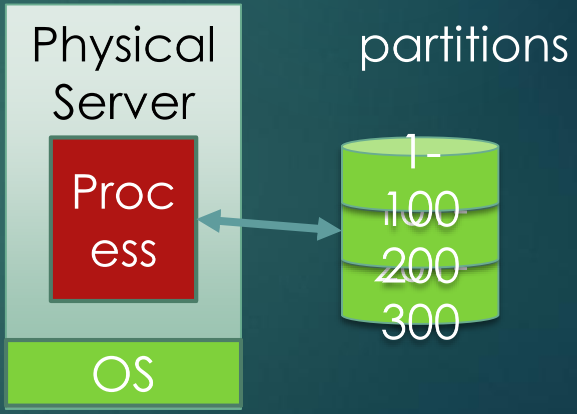
Objectives of Virtualization

- Access control
  - Prevent access outside the VM or container
- Resource sharing
  - Control resources used by each VM or container
- But programmers need additional support
  - Deploying and running applications
    - Requires dependency handling

Motivational Exercise

- Consider the following scenarios
  - Process running on VM accessing a file through a virtual disk
  - Regular process access a file on a shared disk
- Does the OS not also provide isolation?
- Why do we really need virtualization?
- What is the difference between isolation provided by OS and VM?

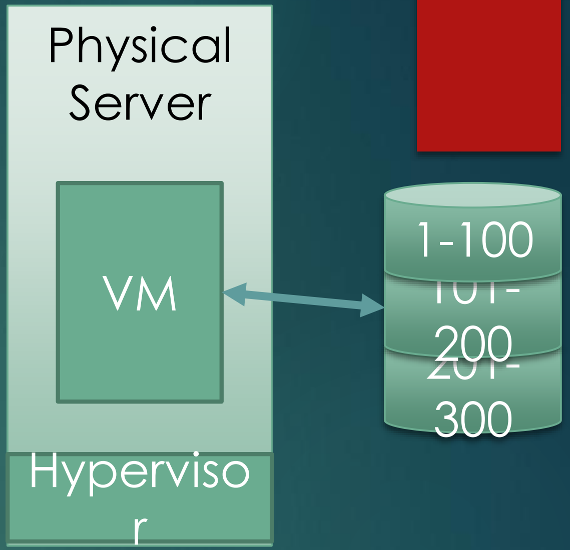


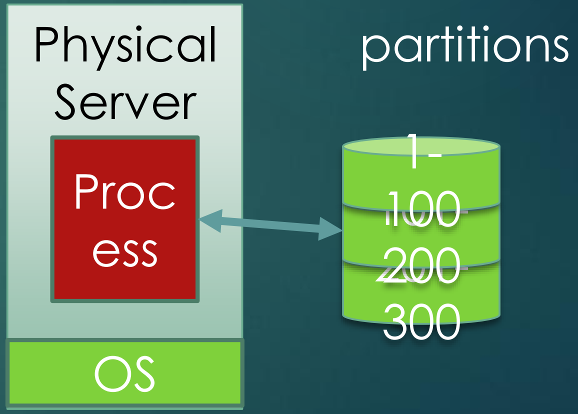


Solution

- OS already provides memory isolation
- For OS scenario
  - Disk however is shared.
  - Any process can see entire filesystem/disk
  - Access control built on top of filesystem.
  - Networking is shared
  - Know about other processes executing on the OS.
- With VMs
  - Complete isolation
    - Userid, processid, network, fs, etc.
  - Also independent OS

But it comes at a cost.



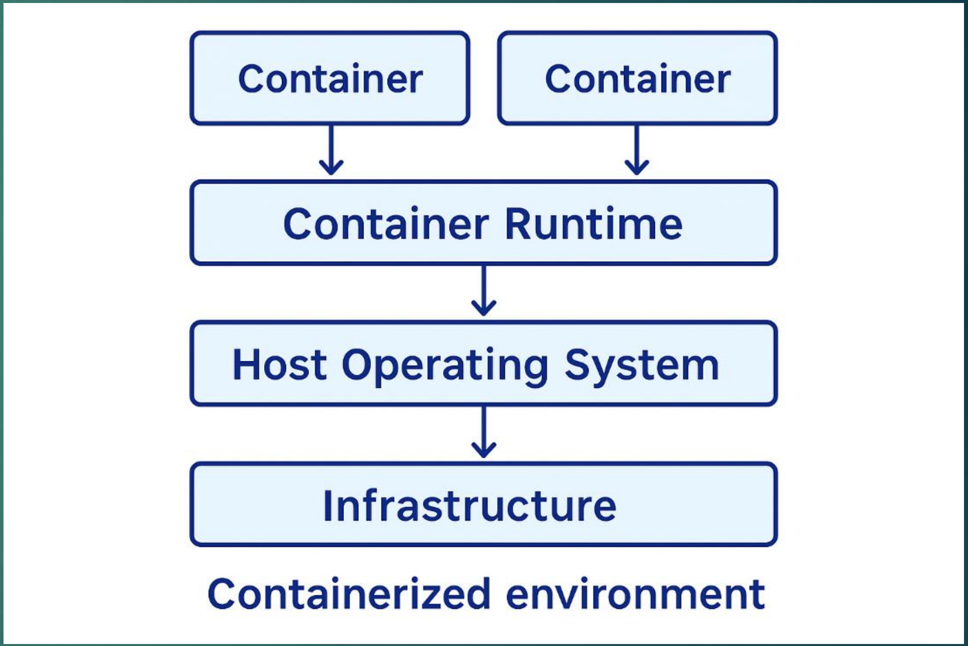


Motivation

- Since OS already provides some isolation
  - Can we build on that to provide a lightweight isolation mechanism?
  - Can we provide support for programmers needs?
- Containers – OS Virtualization

Introducing Containers

- Applications run inside containers
  - Lightweight, isolated units
  - Package software with dependencies
    - Code
    - Runtime
    - Libraries
    - Tools
- Process running in isolation from host system and other containers



Feature	Description
Isolation	Each container runs in its own namespace, isolated from others.
Portability	Containers can run consistently across environments (dev, test, prod).
Lightweight	Containers share the host OS kernel, making them faster and smaller than VMs.
Scalability	Easily scale up/down by running multiple container instances.
Reproducibility	Builds and runs are consistent across machines due to image immutability.
Orchestration Support	Tools like Kubernetes manage container deployment, scaling, and networking.

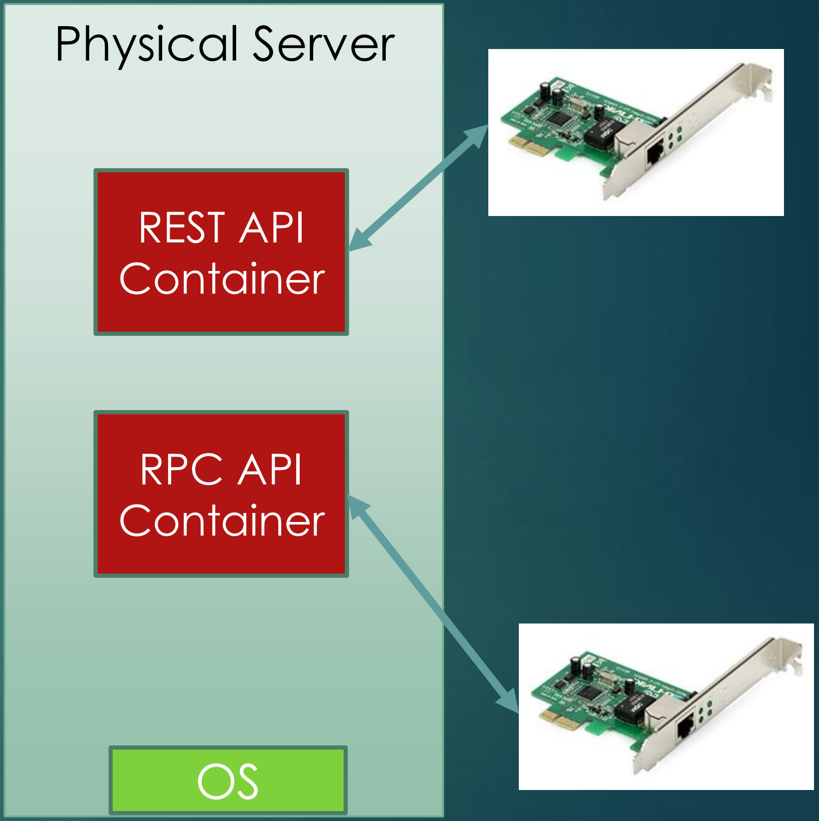
Key Characteristics

Example

In the last class you created a ToDo list management application with REST and grpc

Suppose you wished to deploy both simultaneously?

Problem: How do you start a container?



The container ecosystem: Docker as an example

Docker CLI

to issue commands

Docker Engine

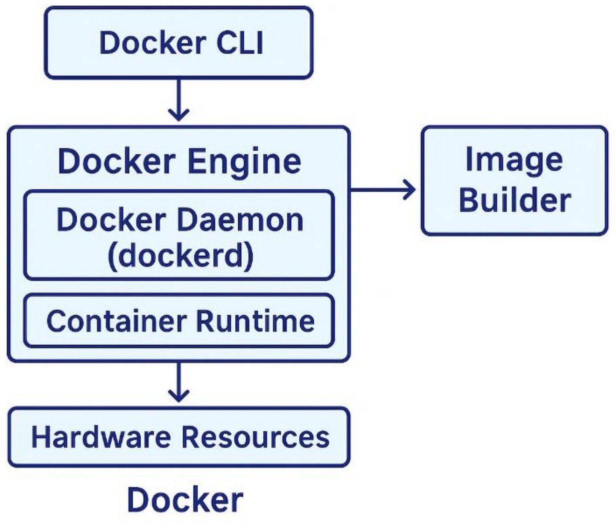
Docker Daemon (dockerd) – manages containers, images, volumes and entworks

Container Runtime – interaction with the system

Image Build System

Used to manage a runtime image of application and dependencies

Uses dockerfile and buildkit



Building a container image

Build Phase

- Specify what you want to build in a dockerfile
- Docker uses Buildkit to execute instructions

Storage Phase

- Once image is built, it is stored in a **local image cache**
- Cache managed by docker engine and stored on disk
- Image is tagged and can be listed with `docker images`

Push Phase

- Optional
- Push the image to remote registry, e.g docker hub

Component	Role
Dockerfile	Blueprint for the image
BuildKit	Executes build steps
Docker Engine	Manages image cache and container lifecycle
Local Cache	Stores built images
Registry	Optional remote storage for sharing images

Sample dockerfile

Lightweight python image required for deploying app.

Specifies your other dependenices like flask

Assumes that application code resides in current directory.

Port number and environment variables for running the application

Command line parameters

```
# Use an official Python base image
FROM python:3.10-slim

# Set working directory
WORKDIR /app
# Copy requirements file and install dependencies
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
# Copy the rest of the application code
COPY . .

# Expose the port Flask will run on
EXPOSE 5000

# Set environment variables for Flask
ENV FLASK_APP=main.py
ENV FLASK_RUN_HOST=0.0.0.0
ENV FLASK_RUN_PORT=5000

# Command to run the Flask app
CMD ["flask", "run"]
```

Exercise – create and run a docker image

- Dockerfile, main.py uploaded to the portal
- In requirements.txt add the following line
  - Flask
- Install docker and run
  - docker build –t flask-rest-api .
  - docker run –p 5000:5000 flask-rest-api

- Create a dockerfile to expose your GRPC server as a docker image and run the docker container.

The docker image - layers

```
FROM python:3.10-slim                    # Layer 1
WORKDIR /app                             # Layer 2
COPY requirements.txt .                  # Layer 3
RUN pip install r requirements.txt       # Layer 4
COPY . .                                 # Layer 5
CMD ["python", "main.py"]                # Layer 6
```

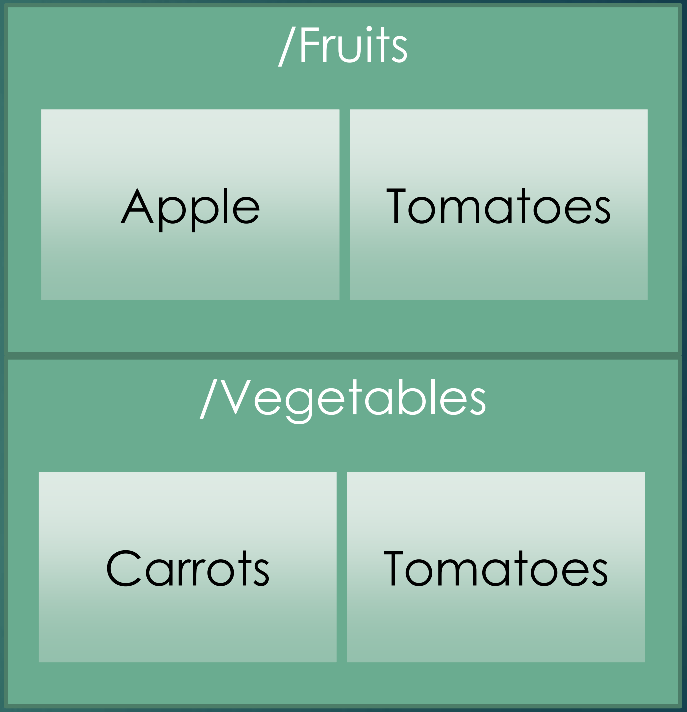
- Each line creates a layer in the image
- Layer 1: the base python image
- Layer 2-5 : Filesystem changes – directories/files
- Layer 6: Metadata (command)

What is a layer?

- A **read only filesystem change**.
- Set of changes stacked together forms an image
- Each layer represents a **delta** from the previous one
- Uses **union filesystem** to combine these layers to achieve a single view
- Layers can be cached, reused and shared across images to optimize storage and performance

Docker and Unionfs – Unionfs features - layering

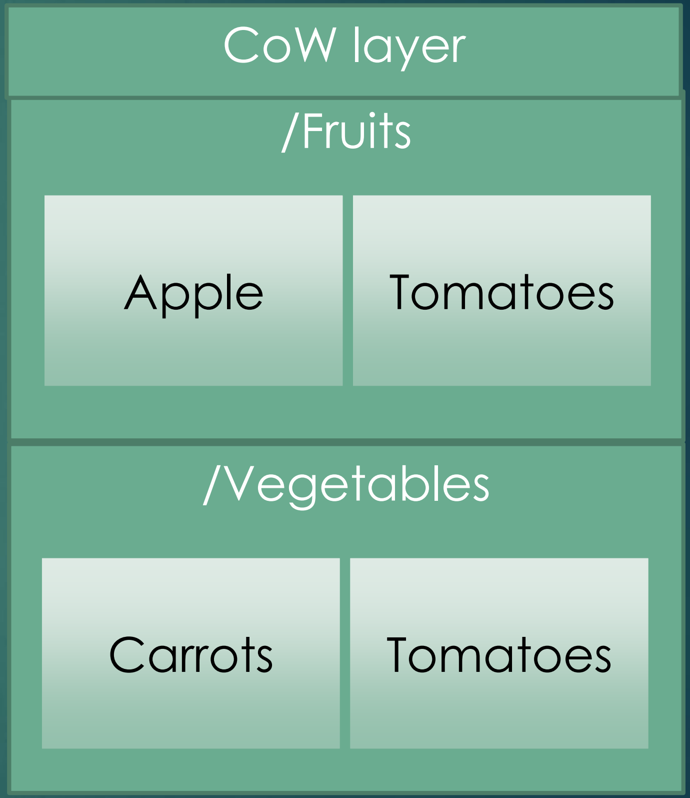
- Unionfs permits layering of file systems
  - */Fruits* contains files *Apple, Tomato*
  - */Vegetables* contains *Carrots, Tomato*
  - *mount –t unionfs –o dirs=/Fruits:/Vegetables none /mnt/healthy*
    - */mnt/healthy* has 3 files – *Apple, Tomato, Carrots*
    - *Tomato* comes from */Fruits* (1st in *dirs* option)
- As if */Fruits* is layered on top of */Vegetables*



http://www.linuxjournal.com/article/7714

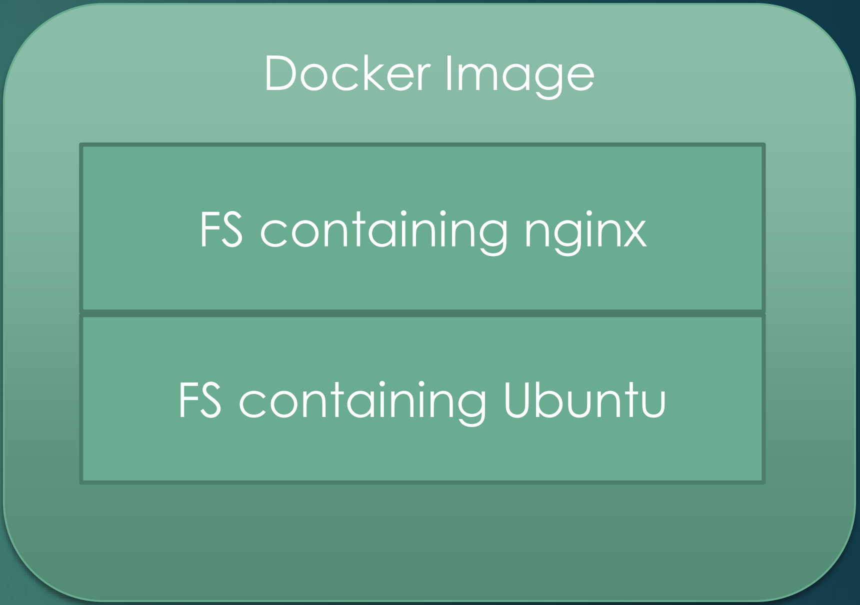
Docker and Unionfs – Unionfs features – Copy on Write

- -o *cow* option on *mount* command enables *copy on write*
  - If change is made to a file
  - Original file is not modified
  - New file is created in a hidden location
- If */Fruits* is mounted *ro*, then changes will be recorded in a temporary layer



Exercise (10 minutes)

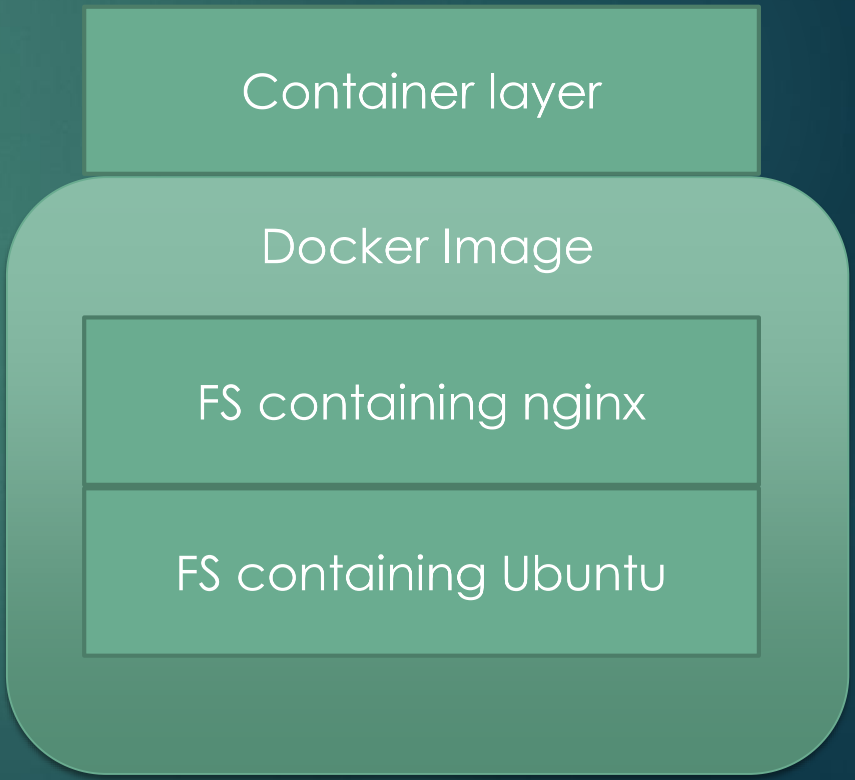
- In Docker image at right
  - Bottom layer contains Ubuntu
  - Top layer contains nginx
- Political party AAA
  - Home page says "BBB are fools"
- Political party BBB
  - Home page says "AAA are morons"
- How can this be done with Docker?



Hint
- To change home page, change index.html in nginx home directory
- Shouldn't change FS with nginx, as then it can't be shared

Exercise 1 Solution

- Political party AAA
  - Home page says "BBB are fools"
- Political party BBB
  - Home page says "AAA are morons"
- AAA, BBB
  - modify index.html in Apache home directory
  - Due to CoW, changes are made only in the container layer
  - <u>No change needs to be made to FS containing nginx</u>
  - <u>FS containing nginx can be shared by all containers</u>



Advantages of layering

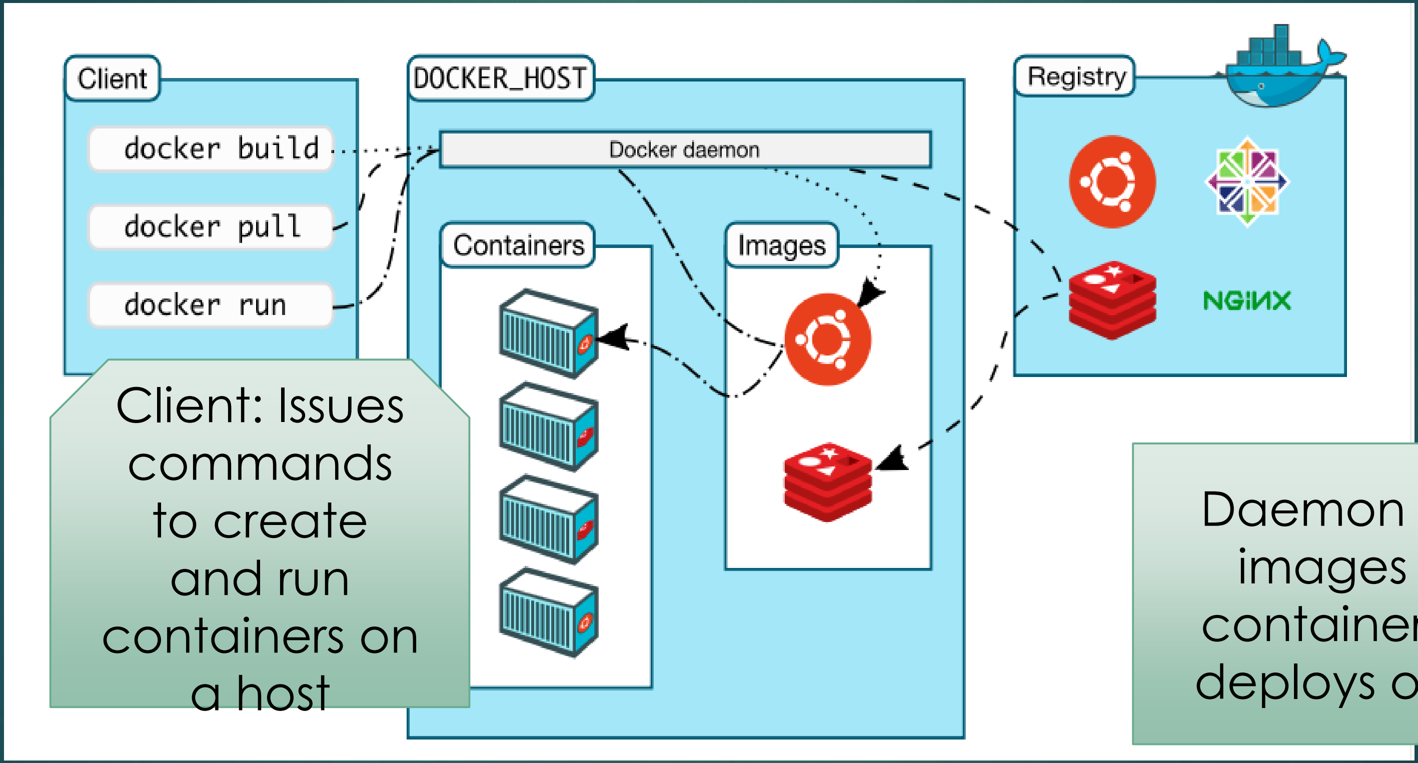
- Caching
  - Docker caches layers
  - Speeds up rebuilds
  - If layer is not changed it is reused
    - E.g, if only requirements.txt changes only that layer onwards will be rebuilt
- Efficiency
  - Shared layers across images reduce disk size
- Portability
  - Distributed and reused across system

Design Tips

  Ordering less frequently changes on top

  Use separate copy commands instead of copying everything.

Docker Architecture

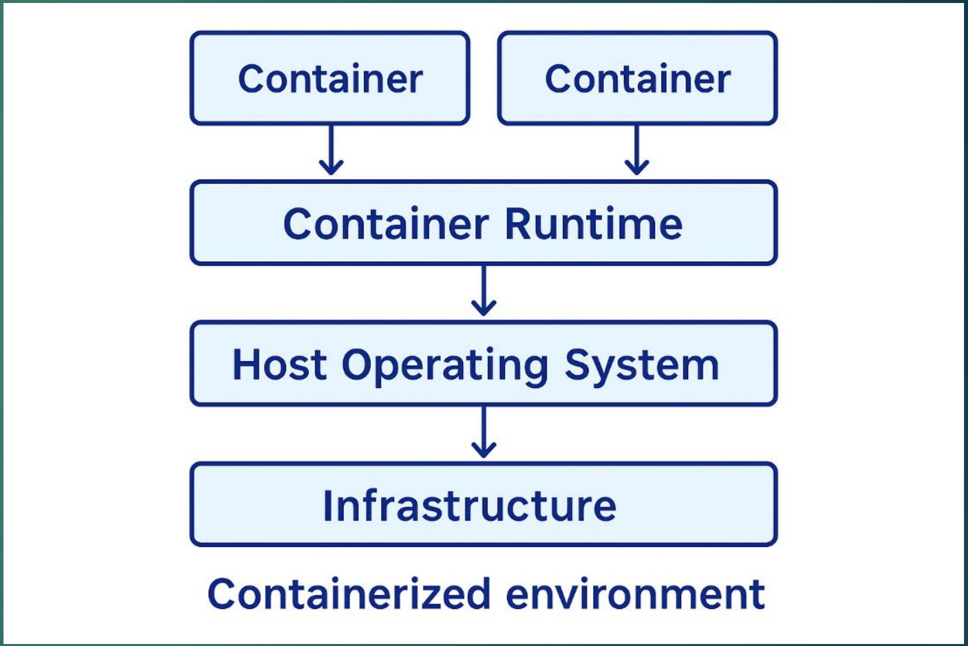


Client: Issues commands to create and run containers on a host

Daemon builds images and containers and deploys on host

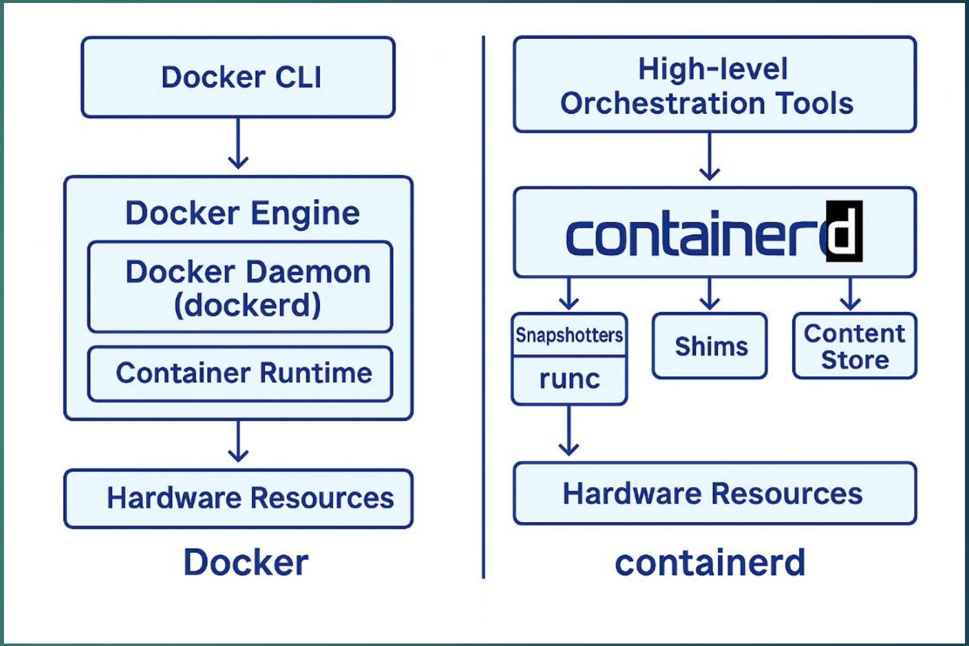
Other components

- Container runtime
  - runc
    - Helps to start and stop containers
    - In newer versions uses the containerd



Containerd

- Container runtime
  - More modular
- Shims
  - Lightweight processes that isolate container execution
- Snapshotters
  - Manage container filesystems using COW
- Content store
  - Stores image layers and metadata
- Runtime
  - Runc
  - Execute containers



Containerd - Snapshotters

- **Role**: Manage container filesystems using copy-on-write techniques.
- **Functionality**:
  - Efficiently layer images.
  - Allow multiple containers to share base layers.
  - Support various backends (OverlayFS, btrfs, ZFS).
- **Why It Matters**: Reduces disk usage and speeds up container startup.

Containerd - Shims

- **Role:** Act as a bridge between containerd and the container runtime (runc).
  - Detach containerd from the container process.
  - Enable container lifecycle management without blocking containerd.
  - Handle logging and exit status.
- **Why It Matters:** Improves reliability and modularity.

Containerd - Shims

- **Role**: Stores all image-related data.
- **Functionality**:
  - Uses content-addressable storage (based on SHA256 hashes).
  - Manages blobs for image layers, configs, and manifests.
  - Ensures deduplication and integrity.
- **Why It Matters**: Central to image management and security.

Containerd vs Docker

Feature	Docker	containerd
Scope	Full container platform	Runtime only
CLI	docker	ctr (low-level), or via Kubernetes
Image Building	Yes (Dockerfile, BuildKit)	No
Kubernetes Support	Deprecated as runtime	Native via CRI
Modularity	Monolithic	Modular (shims, snapshotters)
Use Case	Dev environments, small prod	Large-scale, Kubernetes-native
Resource Usage	Higher	Lower