

Docker Architecture and Components

Table of Contents

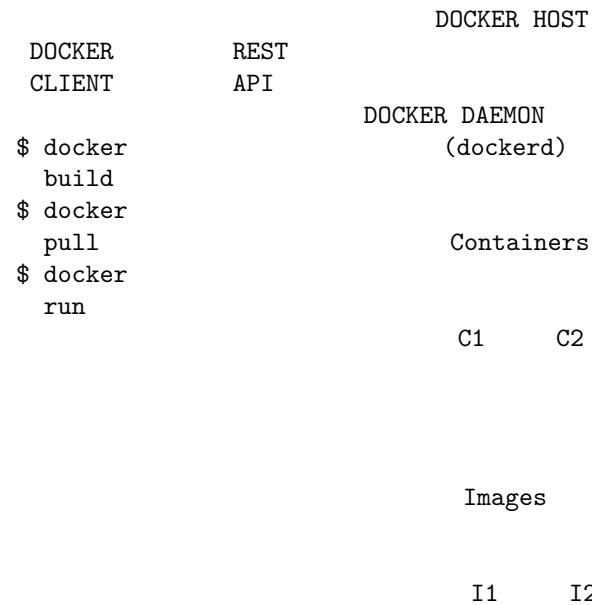
1. Architecture of Docker
 2. Architecture of Docker Engine
 3. Docker vs Virtual Machines
 4. Docker Runtime
 5. Docker Engine Components
 6. Open Container Initiative (OCI)
 7. Layers in Docker
-

Architecture of Docker

High-Level Architecture

Docker uses a **client-server architecture** with the following main components:

DOCKER ARCHITECTURE



DOCKER REGISTRY (Docker Hub, Private Registry)

nginx	node	python	redis
latest	18	3.11	7.0

Component Details

1. Docker Client The command-line interface (CLI) that users interact with.

```
# Client commands
docker build -t myapp .      # Build image
docker run myapp             # Run container
docker ps                   # List containers
docker images                # List images
docker pull nginx            # Pull image from registry
```

How it works:

```
# When you type:
docker run nginx
```

```
# The client:
```

1. Parses the command
2. Sends REST API request to Docker daemon
3. Displays the output from daemon

2. Docker Host The machine where Docker daemon runs and manages containers.

Components on Docker Host: - **Docker Daemon (dockerd):** Core service that manages containers - **Containers:** Running instances of images - **Images:** Templates for containers - **Volumes:** Persistent data storage - **Networks:** Container communication

3. Docker Daemon (dockerd) The background service that: - Listens for Docker API requests - Manages Docker objects (images, containers, networks, volumes) - Communicates with other daemons

```
# Check if daemon is running (Linux)
systemctl status docker
```

```
# View daemon logs
```

```
journalctl -u docker.service
```

```
# Daemon configuration  
cat /etc/docker/daemon.json
```

4. Docker Registry A storage and distribution system for Docker images.

Public Registries: - Docker Hub (hub.docker.com) - Default registry - GitHub Container Registry - Google Container Registry - Amazon ECR

Private Registries:

```
# Run your own registry  
docker run -d -p 5000:5000 --name registry registry:2  
  
# Push to private registry  
docker tag myapp localhost:5000/myapp  
docker push localhost:5000/myapp
```

Complete Workflow Example

```
# Step 1: Build an image (Client → Daemon)  
docker build -t mywebapp:1.0 .  
  
# What happens:  
# 1. Client sends Dockerfile to daemon  
# 2. Daemon builds image layer by layer  
# 3. Daemon stores image locally  
# 4. Client displays build output  
  
# Step 2: Run a container (Client → Daemon)  
docker run -d -p 8080:80 --name web mywebapp:1.0  
  
# What happens:  
# 1. Client sends run command to daemon  
# 2. Daemon checks if image exists locally  
# 3. Daemon creates container from image  
# 4. Daemon starts container  
# 5. Client displays container ID  
  
# Step 3: Push to registry (Client → Daemon → Registry)  
docker push myusername/mywebapp:1.0  
  
# What happens:  
# 1. Client sends push command to daemon  
# 2. Daemon authenticates with registry  
# 3. Daemon uploads image layers
```

```
# 4. Registry stores the image

# Step 4: Pull from registry (Another machine)
docker pull myusername/mywebapp:1.0

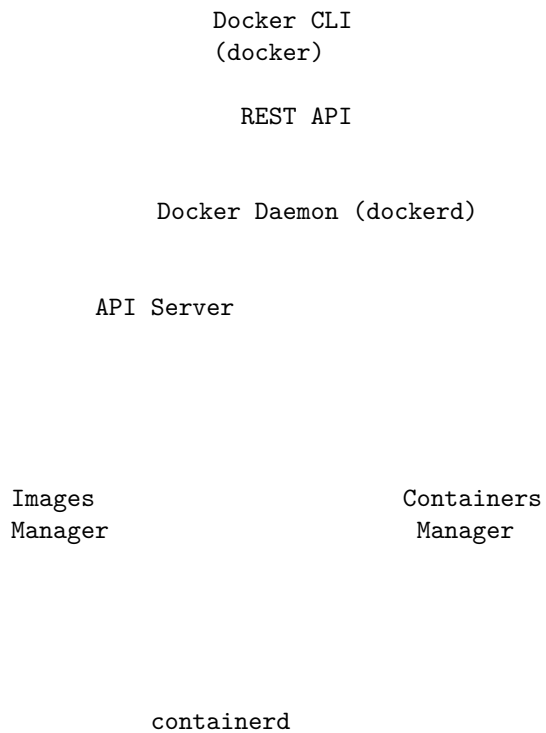
# What happens:
# 1. Client sends pull command to daemon
# 2. Daemon connects to registry
# 3. Daemon downloads image layers
# 4. Daemon stores image locally
```

Architecture of Docker Engine

Docker Engine Deep Dive

Docker Engine is the core technology that runs and manages containers.

DOCKER ENGINE ARCHITECTURE



(Container Runtime)

containerd-shim

runc
(OCI Container Runtime)

Linux Kernel
- Namespaces
- cgroups
- UnionFS

Components Explained

1. Docker CLI

```
# User interface for Docker  
docker --version  
docker info  
docker system df
```

2. Docker Daemon (dockerd) The persistent background process that manages containers.

Responsibilities: - Listen for API requests - Manage images - Manage containers - Manage networks - Manage volumes

Configuration:

```
// /etc/docker/daemon.json  
{  
  "debug": true,  
  "storage-driver": "overlay2",  
  "log-driver": "json-file",  
  "log-opts": {  
    "max-size": "10m",  
    "max-file": "3"
```

```
}  
}
```

3. containerd A high-level container runtime that manages the complete container lifecycle.

Features: - Image transfer and storage - Container execution and supervision
- Low-level storage management - Network attachments

```
# Interact with containerd directly  
ctr images list  
ctr containers list  
ctr tasks list
```

4. containerd-shim A lightweight process that acts as a parent for containers.

Purpose: - Keeps container running even if dockerd crashes - Reports container exit status - Manages STDIO streams

5. runc The low-level container runtime that actually creates and runs containers.

Based on OCI specification:

```
# runc can run containers independently  
runc run mycontainer
```

```
# List running containers  
runc list
```

6. Linux Kernel Features Namespaces - Isolation:

- PID namespace: Process isolation
- NET namespace: Network isolation
- IPC namespace: Inter-process communication isolation
- MNT namespace: Mount point isolation
- UTS namespace: Hostname isolation
- USER namespace: User ID isolation

cgroups (Control Groups) - Resource limiting:

- CPU usage
- Memory usage
- Disk I/O
- Network bandwidth

Example:

```
# Create a container with resource limits
docker run -d \
  --cpus="1.5" \
  --memory="512m" \
  --memory-swap="1g" \
  nginx

# View cgroup settings
docker inspect container_id | grep -i memory
```

Container Lifecycle Through the Stack

```
# Command
docker run -d --name myapp nginx

# Flow:
1. Docker CLI
  > Parses command
  > Sends REST API request to dockerd

2. Docker Daemon (dockerd)
  > Validates request
  > Checks if image exists
  > If not, pulls from registry
  > Calls containerd to create container

3. containerd
  > Unpacks image
  > Prepares container bundle (rootfs + config)
  > Calls runc via containerd-shim

4. containerd-shim
  > Acts as parent process
  > Monitors container

5. runc
  > Creates namespaces
  > Sets up cgroups
  > Mounts filesystem
  > Starts container process

6. Linux Kernel
  > Enforces isolation (namespaces)
  > Enforces resource limits (cgroups)
  > Container is now running!
```

Docker vs Virtual Machines

Fundamental Difference

VIRTUAL MACHINES

App A	App B	App C	App D	App E	App F
Bins/ Libs	Bins/ Libs	Bins/ Libs	Bins/ Libs	Bins/ Libs	Bins/ Libs
Guest OS (Linux) ~1GB	Guest OS (Windows) ~4GB	Guest OS (Linux) ~1GB	Guest OS (Ubuntu) ~1GB	Guest OS (Debian) ~1GB	Guest OS (CentOS) ~1GB

Hypervisor
(VMware, VirtualBox)
~1GB

Host Operating System
(Windows/Linux/Mac)

Physical Server Hardware
(CPU, RAM, Disk, Network)

Total Size: ~10-15GB+ for 6 VMs

DOCKER CONTAINERS

App A	App B	App C	App D	App E	App F
Bins/ Libs ~100MB	Bins/ Libs ~150MB	Bins/ Libs ~200MB	Bins/ Libs ~100MB	Bins/ Libs ~180MB	Bins/ Libs ~120MB

Docker Engine

~100MB

Host Operating System
(Linux Kernel)

Physical Server Hardware
(CPU, RAM, Disk, Network)

Total Size: ~850MB + shared OS for 6 containers

Detailed Comparison

Aspect	Virtual Machines	Docker Containers
OS	Each VM has full OS	Share host OS kernel
Size	GBs (1-10+ GB)	MBs (50-500 MB)
Startup Time	Minutes	Seconds (milliseconds)
Performance	Limited by hypervisor	Near-native performance
Isolation	Complete (hardware-level)	Process-level
Portability	Less portable	Highly portable
Resource Usage	Heavy	Lightweight
Density	10-20 VMs per host	100-1000s containers per host
Management	Complex	Simple

Performance Comparison Example

Scenario: Running 10 web applications on a server with 64GB RAM and 16 CPU cores

With Virtual Machines:

Server: 64GB RAM, 16 cores

VM1: 6GB RAM, 2 cores → App 1
VM2: 6GB RAM, 2 cores → App 2
VM3: 6GB RAM, 2 cores → App 3
VM4: 6GB RAM, 2 cores → App 4
VM5: 6GB RAM, 2 cores → App 5
VM6: 6GB RAM, 2 cores → App 6
VM7: 6GB RAM, 2 cores → App 7
VM8: 6GB RAM, 2 cores → App 8
VM9: 6GB RAM, 2 cores → App 9
VM10: 6GB RAM, 2 cores → App 10

Total Used: 60GB RAM (only 10 apps)

Wasted: ~40GB on OS overhead
Can't run more apps!

With Docker Containers:

Server: 64GB RAM, 16 cores

Container 1: 500MB → App 1
Container 2: 600MB → App 2
Container 3: 400MB → App 3
...
Container 50: 500MB → App 50

Total Used: ~25GB RAM (50 apps!)
Efficient: Most RAM available for apps
Can run 100+ containers easily!

Startup Time Comparison

```
# Virtual Machine
time VBoxManage startvm "Ubuntu-VM"
# Real time: 45.3 seconds

# Docker Container
time docker run -d nginx
# Real time: 0.8 seconds

# 56x FASTER!
```

Use Cases

When to Use Virtual Machines:

- Need complete OS isolation
- Running different OS kernels (Windows + Linux)
- Legacy applications
- Strong security requirements
- Desktop virtualization
- Running untrusted code

Example: Running Windows application on Linux host

When to Use Docker Containers:

- Microservices architecture
- CI/CD pipelines
- Development environments

Cloud-native applications
Scaling web applications
API services

Example: Running 100 microservices

Can You Use Both Together?

Yes! Common pattern:

Physical Server

 Virtual Machine (for isolation/security)

 Docker Containers (for efficiency)

Example:

AWS EC2 Instance (VM)

 50 Docker Containers running microservices

Docker Runtime

What is a Container Runtime?

A **container runtime** is the software responsible for running containers. It handles the low-level operations of creating and managing container processes.

Container Runtime Ecosystem

CONTAINER RUNTIME LEVELS

High-Level Runtime (CRI - Container Runtime Interface)

 containerd (Docker's default)

 CRI-O (Kubernetes-optimized)

 podman (Daemonless alternative)

 ↓ Uses ↓

Low-Level Runtime (OCI - Open Container Initiative)

 runc (Most common)

 crun (Faster, written in C)

 kata-runtime (VM-based containers)

 gVisor (Google's sandboxed runtime)

Types of Container Runtimes

1. High-Level Runtime (containerd)

```
# containerd is used by Docker
docker info | grep -i runtime
# Output: Runtimes: runc io.containerd.runc.v2

# Direct containerd usage
ctr images pull docker.io/library/nginx:latest
ctr run -d docker.io/library/nginx:latest my-nginx
```

Features: - Image management - Container lifecycle management - Snapshot management - Network namespace management

2. Low-Level Runtime (runc)

```
# Create OCI bundle
mkdir mycontainer
cd mycontainer
mkdir rootfs

# Export a container's filesystem
docker export $(docker create nginx) | tar -C rootfs -xf -

# Generate config
runc spec

# Run container
runc run mycontainer
```

Runtime Configuration

```
// /etc/containerd/config.toml
version = 2

[plugins]
[plugins."io.containerd.grpc.v1.cri"]
  [plugins."io.containerd.grpc.v1.cri".containerd]
    [plugins."io.containerd.grpc.v1.cri".containerd.runtimes]
      [plugins."io.containerd.grpc.v1.cri".containerd.runtimes.runc]
        runtime_type = "io.containerd.runc.v2"
      [plugins."io.containerd.grpc.v1.cri".containerd.runtimes.runc.options]
        SystemdCgroup = true
```

Docker Engine Components

Core Components Overview

Docker Engine = Docker Daemon + containerd + runc

1. Docker Daemon (dockerd)
Purpose: High-level management
2. containerd
Purpose: Container lifecycle management
3. runc
Purpose: Container execution

Component Interactions

Example: Starting a Container

```
# User command
docker run -d --name web nginx

# 1. Docker Daemon receives request
#   - Validates parameters
#   - Checks if image exists
#   - Prepares container config

# 2. dockerd calls containerd
#   - Passes container specification
#   - Requests container creation

# 3. containerd prepares environment
#   - Sets up container bundle
#   - Configures networking
#   - Sets up storage

# 4. containerd calls runc (via shim)
#   - Passes OCI specification
#   - Requests container start

# 5. runc creates container
#   - Creates namespaces
#   - Sets up cgroups
#   - Mounts rootfs
#   - Starts init process

# 6. Container is running!
```

```
# - dockerd monitors health
# - containerd manages lifecycle
# - shim maintains connection
```

Viewing Components

```
# Check running processes
ps aux | grep docker
# dockerd (main daemon)
# containerd (runtime)
# containerd-shim (per container)

# Check containerd
ctr version
ctr namespaces list

# Check runc
runc --version
runc list
```

Open Container Initiative (OCI)

What is OCI?

The **Open Container Initiative** is an industry standard for container formats and runtimes, established in 2015 by Docker and other industry leaders.

OCI Specifications

1. Image Specification (OCI Image) Defines how container images are structured.

```
{
  "schemaVersion": 2,
  "config": {
    "mediaType": "application/vnd.oci.image.config.v1+json",
    "size": 7023,
    "digest": "sha256:abc123..."
  },
  "layers": [
    {
      "mediaType": "application/vnd.oci.image.layer.v1.tar+gzip",
      "size": 32654,
      "digest": "sha256:def456..."
    }
  ]
}
```

```
    ]  
  }  
}
```

2. Runtime Specification (OCI Runtime) Defines how containers should be run.

```
{  
  "ociVersion": "1.0.0",  
  "process": {  
    "terminal": true,  
    "user": {  
      "uid": 0,  
      "gid": 0  
    },  
    "args": [  
      "/bin/sh"  
    ],  
    "env": [  
      "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin",  
      "TERM=xterm"  
    ],  
    "cwd": "/",  
    "capabilities": {  
      "bounding": [  
        "CAP_AUDIT_WRITE",  
        "CAP_KILL",  
        "CAP_NET_BIND_SERVICE"  
      ]  
    }  
  },  
  "root": {  
    "path": "rootfs",  
    "readonly": true  
  },  
  "hostname": "mycontainer"  
}
```

Why OCI Matters

Before OCI:

Each vendor had their own format:

- Docker containers
- rkt containers
- LXC containers
- Incompatible with each other!

After OCI:

One standard format:

- Build with Docker
- Run with containerd
- Orchestrate with Kubernetes
- Works everywhere!

OCI-Compliant Tools

```
# Build OCI images
docker build -t myapp .
buildah bud -t myapp .
kaniko --context . --destination myapp

# Run OCI containers
docker run myapp
podman run myapp
runc run myapp

# All compatible!
```

Layers in Docker

What Are Layers?

Docker images are built using a **layered filesystem**. Each instruction in a Dockerfile creates a new layer.

Docker Image = Stack of Read-Only Layers + Writable Container Layer

Container Layer (Read-Write)	← Your changes here
Layer 5: CMD ["nginx"]	← Read-only
Layer 4: COPY app /var/www	← Read-only
Layer 3: RUN npm install	← Read-only
Layer 2: WORKDIR /app	← Read-only
Layer 1: FROM node:18	← Read-only (base)

How Layers Work

Example Dockerfile:

```
# Layer 1
FROM ubuntu:22.04

# Layer 2
RUN apt-get update

# Layer 3
RUN apt-get install -y nginx

# Layer 4
COPY index.html /var/www/html/

# Layer 5
CMD ["nginx", "-g", "daemon off;"]
```

Build Output:

```
docker build -t webserver .

# Output shows layers:
Step 1/5 : FROM ubuntu:22.04
----> a1b2c3d4e5f6
Step 2/5 : RUN apt-get update
----> Running in x1y2z3a4b5c6
----> d7e8f9g0h1i2
Step 3/5 : RUN apt-get install -y nginx
----> Running in j3k4l5m6n7o8
----> p9q0r1s2t3u4
Step 4/5 : COPY index.html /var/www/html/
----> v5w6x7y8z9a0
Step 5/5 : CMD ["nginx", "-g", "daemon off;"]
----> b1c2d3e4f5g6
Successfully built b1c2d3e4f5g6
```

Layer Caching

Docker caches layers to speed up builds!

```
# First build
docker build -t app:v1 .
# Takes 5 minutes

# Change only the last line in Dockerfile
# Second build
```

```
docker build -t app:v2 .
# Takes 10 seconds! (uses cached layers)
```

Example with Cache:

```
FROM node:18
WORKDIR /app

# These layers are cached if package.json unchanged
COPY package*.json ./
RUN npm install

# Only this changes frequently
COPY . .

# Efficient builds!
```

Viewing Layers

```
# View image layers
docker history nginx

# Output:
# IMAGE          CREATED          SIZE          COMMENT
# a1b2c3d4e5f6   2 weeks ago     142MB
# <missing>      2 weeks ago     0B            CMD ["nginx"]
# <missing>      2 weeks ago     57MB          RUN apt-get install nginx
# <missing>      2 weeks ago     0B            EXPOSE 80
# <missing>      2 weeks ago     77MB          FROM ubuntu:22.04

# Detailed layer inspection
docker inspect nginx

# See layer digests
docker image inspect nginx --format='{{.RootFS.Layers}}'
```

Layer Sharing

Multiple images can share layers!

Image A: node:18 (200MB)
 Ubuntu base (77MB)
 Node.js (123MB)

Image B: My App (250MB)
 Ubuntu base (77MB) ← SHARED!
 Node.js (123MB) ← SHARED!
 App code (50MB) ← New

Total Disk Used: 250MB (not 450MB!)

Example:

```
# Pull multiple Node.js images
docker pull node:18
# Downloading: 200MB

docker pull node:18-alpine
# Already exists (shares base layers!)
# Downloading: Only 45MB new layers
```

Copy-on-Write (CoW)

When a container modifies a file, Docker uses **copy-on-write**:

1. Container tries to modify file.txt
2. Docker copies file.txt from image layer to container layer
3. Modification happens in container layer
4. Original layer remains unchanged

```
Container Layer
file.txt (modified)
```

```
Image Layer
file.txt (original)
```

Best Practices for Layers

Bad Practice:

```
FROM ubuntu:22.04
RUN apt-get update
RUN apt-get install -y package1
RUN apt-get install -y package2
RUN apt-get install -y package3
# 4 layers created!
```

Good Practice:

```
FROM ubuntu:22.04
RUN apt-get update && \
  apt-get install -y \
    package1 \
    package2 \
    package3 && \
```

```
rm -rf /var/lib/apt/lists/*  
# 1 layer created!
```

Layer Size Optimization

```
# Check image size  
docker images  
# REPOSITORY TAG SIZE  
# myapp v1 850MB ← Too big!  
  
# Optimize by:  
# 1. Using smaller base images  
FROM node:18-alpine # Instead of node:18  
  
# 2. Combining commands  
RUN apt-get update && apt-get install -y pkg && rm -rf /var/lib/apt/lists/*  
  
# 3. Using multi-stage builds (covered in next section)  
  
# Result:  
# myapp v2 150MB ← Much better!
```

Key Takeaways

1. **Docker Architecture:** Client-server model with daemon, registry, and host
 2. **Docker Engine:** Composed of dockerd, containerd, and runc
 3. **Containers vs VMs:** Containers share OS kernel, VMs have full OS
 4. **OCI Standards:** Ensure container portability across tools
 5. **Layers:** Enable efficient storage and fast builds through caching
 6. **Copy-on-Write:** Allows efficient file modifications in containers
-

What's Next?

In the next sections, we'll cover:

- Installing Docker on Windows, Mac, and Linux
- Docker CLI commands in detail
- Working with images and containers
- Building and optimizing Dockerfiles

Understanding architecture is key to mastering Docker!