ChatGPT

# Docker Knowledge Base

This document collates factual information about Docker from the sources consulted during research. Each statement is accompanied by citations linking back to the original text. Use this as a reference for creating course materials, generating assessments or verifying statements in the curriculum.

## Docker architecture

- **Client-server model:** Docker follows a client–server architecture where the **Docker daemon** (`dockerd`) runs on the host machine and manages container lifecycle (building, running and distributing images). The **Docker client** (`docker` CLI) communicates with the daemon using a REST API [1]. Developers issue commands through the client, which forwards them to the daemon.

- **Registry:** A **Docker registry** (e.g. Docker Hub) acts as a central store for images. The daemon interacts with the registry to push and pull images. Registries allow organisations to share images internally or publicly [2].

- **Underlying isolation mechanisms:** Docker relies on Linux **namespaces** to isolate process trees, network interfaces and file systems, and **control groups (cgroups)** to limit CPU and memory usage [3]. These kernel features make containers lightweight compared to virtual machines.

## Images and containers

- **Image definition:** A **Docker image** is a read-only template consisting of a stack of layers. It includes the application code, libraries and runtime environment and can be versioned by tags [4] [5]. Images are immutable and reproducible; building an image always yields the same result when the source and instructions are unchanged [6].

- **Container definition:** A **container** is a running instance of an image. Containers share the host operating system kernel, which makes them more resource-efficient and faster to start than virtual machines [7] [8]. Each container runs in isolated namespaces with limited resources via cgroups.

- **Containers vs virtual machines:** Traditional virtual machines require a full guest operating system on a hypervisor, consuming more resources. Docker containers use namespaces and cgroups to isolate processes while sharing the host kernel [9] [10]. This approach offers faster startup times and lower overhead. Deploying software with Docker improves consistency, versioning and portability across environments [11].

## Persistent storage: volumes

- **Why volumes are preferred:** Docker volumes are storage locations managed by Docker that live outside the container's writable layer. They are the recommended way to persist data because they

are easier to back up and migrate, safe to share between containers, and do not increase the size of the container image [12]. Volumes offer better performance than writing directly to the container file system [13].

- **Volume lifecycle:** Volumes exist independently of any container. Deleting a container does not remove its associated volume, and multiple containers can mount the same volume to share data [14].

## Networking

- **Built-in network drivers:** Docker ships with several network drivers. Key drivers include:
- **bridge** – default driver; creates a private internal network on a single host [15].
- **host** – removes network isolation by sharing the host's network stack; used for cases where performance is critical [15].
- **overlay** – connects containers across multiple Docker daemons; commonly used for Swarm clusters or multi-host communication [15].
- **ipvlan** – provides fine-grained control over IPv4 and IPv6 assignment; useful when MAC address constraints exist [15].
- **macvlan** – assigns a unique MAC address to each container so that it appears as a physical device on the network; helpful for migrating virtual machines or legacy applications [15].
- **none** – attaches a container to no network interface, isolating it completely [15].

- **Network selection guidance:** When choosing a driver, Docker documentation recommends using the default bridge network for simple standalone containers, user-defined bridge networks to isolate application components, host mode for near-native networking, overlay for multi-host connectivity and Swarm mode, macvlan when containers must appear as full machines on the network, and ipvlan when MAC addresses are limited [16]. Third-party plugins are available for specialised requirements.

## Docker Compose

- **Purpose: Docker Compose** allows developers to define multi-container application stacks in a single YAML file. It provides commands to start, stop, rebuild and inspect the entire stack with one invocation, making it suitable for development, testing and CI environments [17].

- **Key benefits:** Compose files are portable across environments and can define services, networks and volumes declaratively. With a single command (`docker compose up`), all specified services are brought up consistently. Compose commands also support streaming logs, scaling services, running one-off tasks and gracefully tearing down the stack [17].

## Writing Dockerfiles and best practices

- **RUN instruction:** Executes commands during the image build. Each `RUN` creates a new layer; combining commands using `&&` reduces the number of layers and helps produce smaller images [18].

- **CMD instruction:** Sets the default command to execute when a container starts. It can be overridden by arguments to `docker run` [19]. Use `CMD` to provide sensible defaults without locking in the executable.

- **ENTRYPOINT instruction:** Specifies the main executable for the container; arguments passed to `docker run` are appended to `ENTRYPOINT` [20]. Use `ENTRYPOINT` when the container should always run the same executable (e.g. a server process) and combine it with `CMD` to set default arguments.

- **Multi-stage builds:** Splitting a Dockerfile into multiple stages allows you to compile or build dependencies in one stage and copy only the final artefacts into a smaller runtime image. This reduces the final image size and attack surface [21].

- **Reusable stages and base images:** You can create reusable stages to share build logic across images [22]. It is important to choose a minimal and official base image, keeping build and runtime environments separate to reduce security risks [23].

- **Rebuilding and maintenance:** Rebuild images regularly to obtain patched dependencies [24]. Use a `.dockerignore` file to exclude unnecessary files and secrets from the build context and strive for one primary process per container [25].

## Docker Swarm and orchestration

- **What is a Swarm?** Docker Swarm is a native container orchestrator built into Docker. A **Swarm** is a collection of Docker nodes (physical or virtual machines) acting as a single cluster [26].

- **Node roles:** Nodes in a Swarm are either **manager nodes**, which run the Swarm Manager, reconcile the desired state and participate in Raft consensus, or **worker nodes**, which run tasks assigned by managers [27]. For high availability, at least three manager nodes are recommended [28].

- **Services and tasks:** A **service** defines the desired state of containers in the Swarm. A **task** is an individual container instance that runs on a node. Services can be of type *replicated* (a defined number of identical tasks) or *global* (one task per node) [29]. Docker Swarm automatically balances incoming connections across tasks.

- **Benefits:** Swarm provides high availability and load balancing through Raft consensus and integrated routing mesh. The Docker CLI can manage both Swarm clusters and single-node setups, making it easy to get started [28] [29].

## Comparison with virtual machines

Virtual machines (VMs) and containers both provide isolation, but they achieve it differently. VMs run a complete guest operating system on a hypervisor, while containers share the host OS kernel [9]. Docker uses namespaces and cgroups to isolate processes and limit resource usage, resulting in faster startup times and reduced overhead [10]. Containers are therefore more portable, easier to deploy and often more scalable than VMs [11].

[1] [2] [3] Docker Architecture Explained: Client, Daemon & Registry
https://kodekloud.com/blog/docker-architecture/

[4] [5] [6] [7] [8] Docker image vs container: What are the differences? - CircleCI
https://circleci.com/blog/docker-image-vs-container/

[9] [10] [11] How is Docker different from a virtual machine? - Microsoft Q&A
https://learn.microsoft.com/en-nz/answers/questions/2109219/how-is-docker-different-from-a-virtual-machine

[12] [13] [14] Volumes | Docker Docs
https://docs.docker.com/engine/storage/volumes/

[15] [16] Network drivers | Docker Docs
https://docs.docker.com/engine/network/drivers/

[17] Docker Compose | Docker Docs
https://docs.docker.com/compose/

[18] [19] [20] Docker Best Practices: Choosing Between RUN, CMD, and ENTRYPOINT | Docker
https://www.docker.com/blog/docker-best-practices-choosing-between-run-cmd-and-entrypoint/

[21] [22] [23] [24] [25] Best practices | Docker Docs
https://docs.docker.com/build/building/best-practices/

[26] [27] [28] [29] What is Docker Swarm? | Sysdig
https://www.sysdig.com/learn-cloud-native/what-is-docker-swarm