**DOCKER**

1. What is containerization?

**Containerization** is a lightweight form of virtualization that involves packaging an application **and all its dependencies** (libraries, configuration files, binaries, etc.) into a single unit called a **container**.

This ensures that the application runs **consistently** across different computing environments — from a developer’s laptop to test, staging, and production servers.

**🧱 Key Concept**

A **container** is like a **mini virtual machine**, but instead of running an entire operating system, it **shares the host OS kernel** while keeping processes **isolated** from one another.

Each container has:

* The **application code**
* Required **libraries and dependencies**
* A lightweight **runtime environment**

**⚙️ How It Works**

* Containers run on top of a **Container Engine** like **Docker**, **containerd**, or **Podman**.
* The engine uses **Linux kernel features** such as:
  + **Namespaces** → isolate processes, network, and file systems.
  + **cgroups (control groups)** → limit CPU, memory, and I/O usage.

**📦 Example**

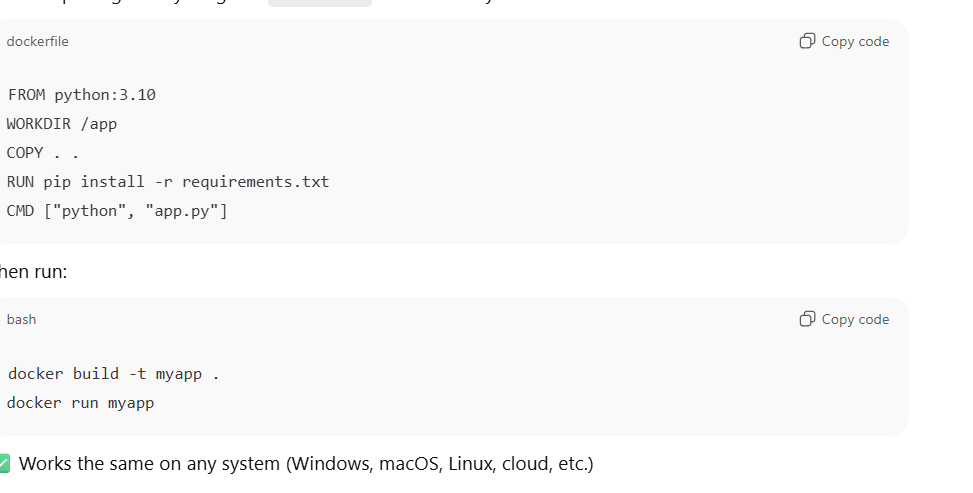
Let’s say you have a Python app.

Without containers:

* You need to install Python, dependencies, and correct versions on every system.

With containers:

* You package everything in a Dockerfile and run it anywhere with Docker installed.



1. **What is the difference between an image and a container?**

**🖼️ Docker Image**

* A **Docker image** is a **blueprint or template** for creating containers.
* It’s a **read-only**, **static** file that contains:
  + Application code
  + Dependencies
  + Libraries
  + Environment variables
  + Configuration files

You can think of it as a **snapshot** of everything your app needs to run.

**📦 Container**

* A **container** is a **runtime instance** of an image.
* When you **run** an image, Docker creates a container from it.
* The container is:
  + **Live** (it can be started, stopped, deleted)
  + **Writable** — changes made inside a running container don’t affect the original image
  + **Isolated** — runs in its own environment using the image as a base

1. What happens when you run docker run?

**⚙️ Step-by-Step Breakdown**

**🧩 1. Docker CLI sends the command to the Docker Daemon**

* The Docker client (docker run ...) talks to the **Docker daemon** (background service) using the Docker API.
* The **daemon** actually performs all heavy lifting — pulling images, creating containers, networking, etc.

**📥 2. Docker checks if the image is available locally**

* Docker looks for the image (e.g., nginx) in your **local image cache**.
* If it’s not found, Docker automatically **pulls** it from a **registry** (like Docker Hub):
* docker pull nginx:latest

**🧱 3. A writable container layer is created on top of the image**

* Docker creates a **thin writable layer** on top of the **read-only image layers**.
* This writable layer is where all container-specific changes happen:
  + Files written by the app
  + Logs generated
  + Temporary data, etc.

This is implemented using **Union File System / OverlayFS**.

**🪄 4. A filesystem and container ID are assigned**

* Docker sets up a unique **container ID** (a long hash).
* It mounts the combined filesystem (read-only image + writable layer).
* The container now has its **own isolated filesystem view**.

**🔒 5. Network and namespace isolation are set up**

Docker uses Linux kernel features to isolate the container:

* **Namespaces** → isolate processes, users, network, and mount points.
* **cgroups** → control resource limits (CPU, memory, I/O).
* A **virtual network interface** (bridge network by default) is attached.

**🏃 6. The container process is started**

* Docker executes the **default command** defined in the image’s CMD or ENTRYPOINT instruction.

Example:  
If the image’s Dockerfile has:

CMD ["nginx", "-g", "daemon off;"]

Then Docker runs:

nginx -g "daemon off;"

This command becomes **PID 1** inside the container.

1. What is a docker Layer?

A **Docker layer** is one of the **building blocks** that make up a **Docker image**.  
Each instruction in a **Dockerfile** (like FROM, RUN, COPY, etc.) creates a **new layer**.

These layers are:

* **Stacked on top of each other**
* **Read-only** (for images)
* **Reused (cached)** across builds to save time and space

Together, they form the **Union File System** that Docker uses to present a **single unified view** of your image or container.

A screenshot of a computer

AI-generated content may be incorrect.

Each layer is **immutable (read-only)**.  
When you change something in the Dockerfile, Docker **rebuilds only that layer and those above it**, using the cache for the rest.

**⚡ Why Layers Matter**

**1. Caching for Faster Builds**

* Docker reuses unchanged layers.
* For example, if you only modify app.py, Docker reuses all previous layers — making the build **much faster**.

A white background with black text

AI-generated content may be incorrect.

**2. Efficiency and Reuse**

* Many images share the same base layers (e.g., ubuntu:20.04).
* Docker stores identical layers **only once** on disk — saving space.

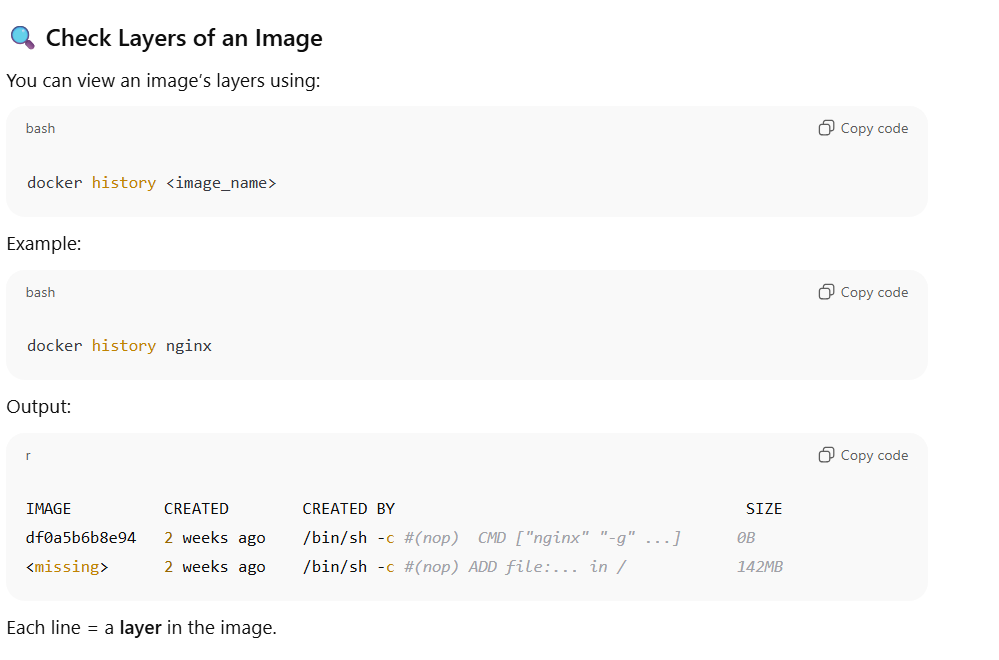
**3. Writable Container Layer**

When you run a container from an image:

* Docker adds **one writable layer** on top of all read-only layers.
* Any file changes, logs, or new data go here.
* Once the container is removed, this layer is deleted.

**🧱 Image layers (read-only)  
➕  
🧾 Container layer (read-write)**

🏃 **Running container**

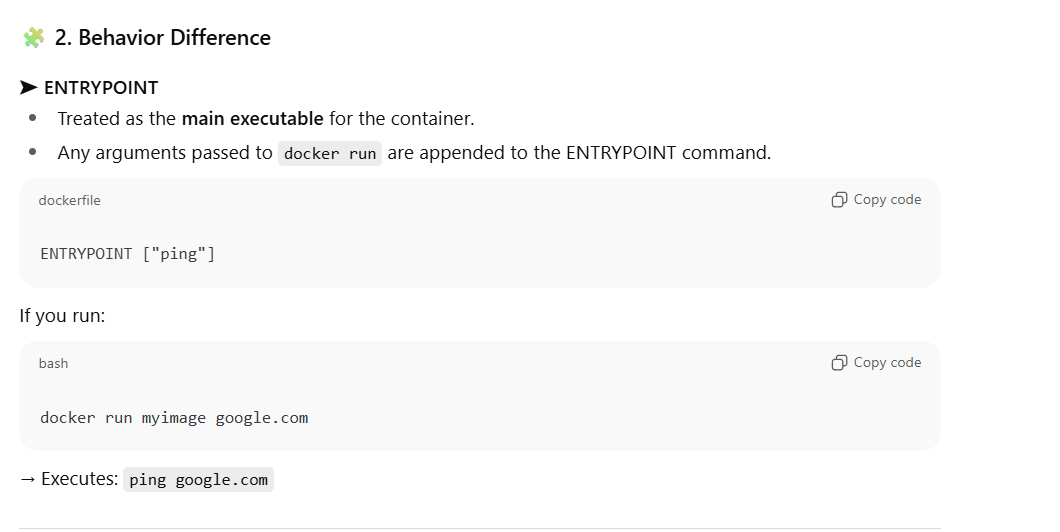


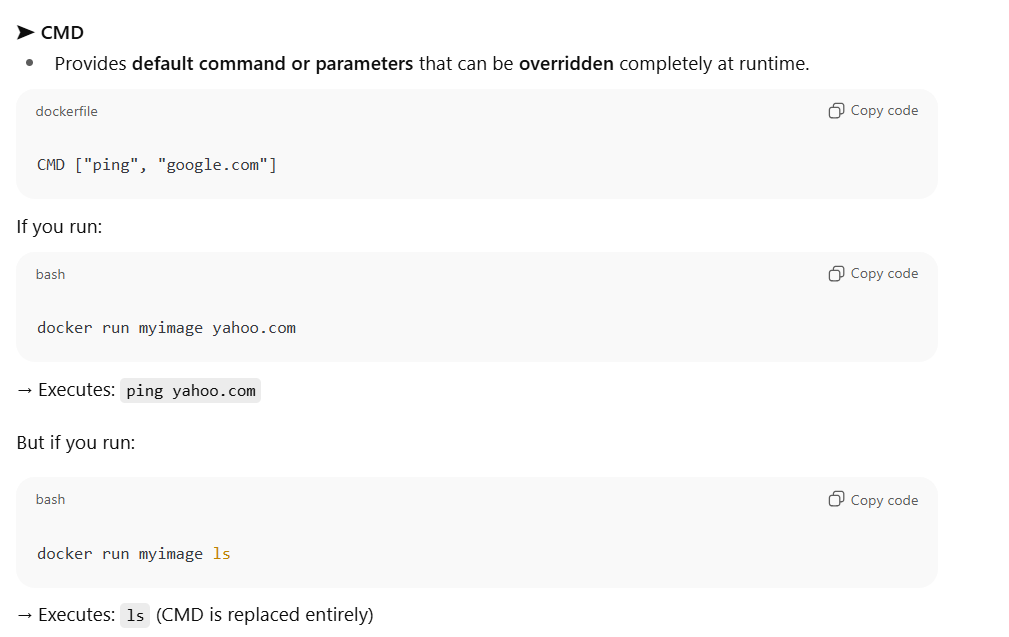
1. What is the difference between ENTRYPOINT and CMD in Dockerfile?

Both **ENTRYPOINT** and **CMD** in a Dockerfile define **what command runs** when a container starts — but they serve slightly different purposes.

**⚙️ 1. Purpose**

| **Instruction** | **Purpose** |
| --- | --- |
| **ENTRYPOINT** | Defines the **main command** that *always* runs when the container starts. |
| **CMD** | Defines the **default arguments** (or command) that can be **overridden** at runtime. |





A screenshot of a computer

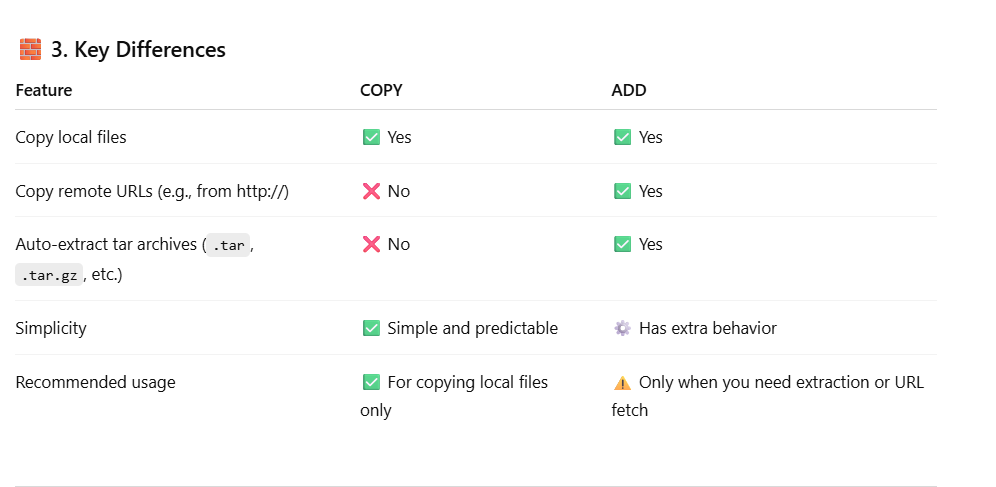
AI-generated content may be incorrect.

1. What is the difference between COPY and ADD in Dockerfile?

Both **COPY** and **ADD** in a Dockerfile are used to **add files** into a Docker image — but they are **not identical**.  
COPY is simpler and preferred in most cases, while ADD has **extra features**.

**⚙️ 1. Purpose**

| **Instruction** | **Purpose** |
| --- | --- |
| **COPY** | Copies files or directories from the **host machine** into the **image**. |
| **ADD** | Does everything COPY does **plus** supports **remote URLs** and **auto-extracts compressed archives**. |



A screenshot of a computer

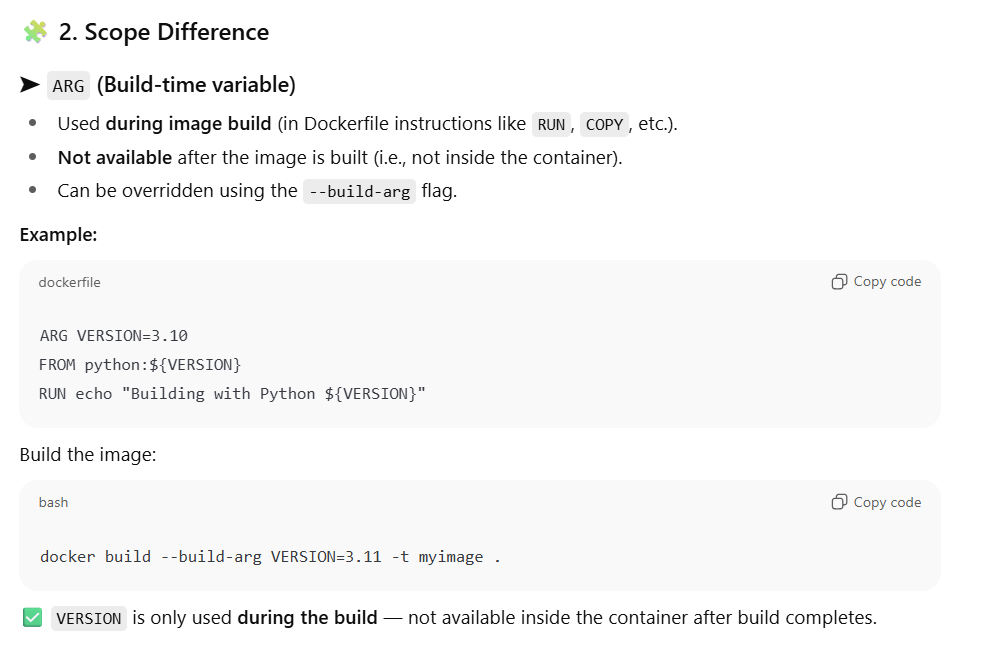
AI-generated content may be incorrect.

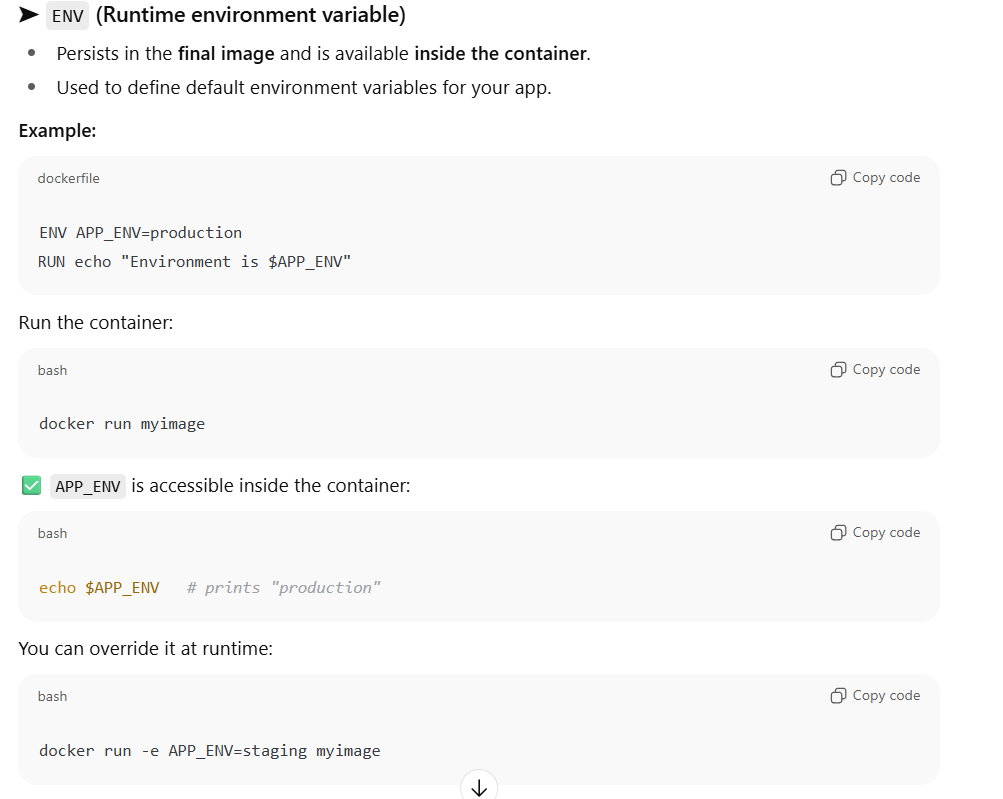
1. What is the difference between ENV and ARG command in dockerfile?

Both **ARG** and **ENV** in a Dockerfile are used to **define variables**, but they differ in **when** and **where** they are available.

**⚙️ 1. Purpose**

| **Instruction** | **Purpose** |
| --- | --- |
| **ARG** | Defines **build-time** variables — available **only while building** the image. |
| **ENV** | Defines **runtime** environment variables — available **inside the running container**. |





A screenshot of a computer

AI-generated content may be incorrect.

1. What is the usage of EXPOSE command in dockerfile?

The **EXPOSE** command in a Dockerfile is used to **document** which **network ports** the container will listen on at runtime.

It **does not actually publish or open the port** — it simply tells Docker (and others reading your Dockerfile) that the application inside the container uses that port.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

1. Explain Multi stage build and its usage.

**Multi-stage builds** in Docker are a powerful feature that helps you create **smaller, more secure, and efficient Docker images** by separating the **build environment** from the **runtime environment** — all within a single Dockerfile.

Traditionally, when you build applications (especially compiled ones like Go, Java, .NET, etc.), you need extra tools — compilers, SDKs, build dependencies, etc.  
If you install all those tools in one Docker image, the final image becomes **large** and includes **unnecessary build tools** that are not needed at runtime.

Multi-stage builds solve this by:

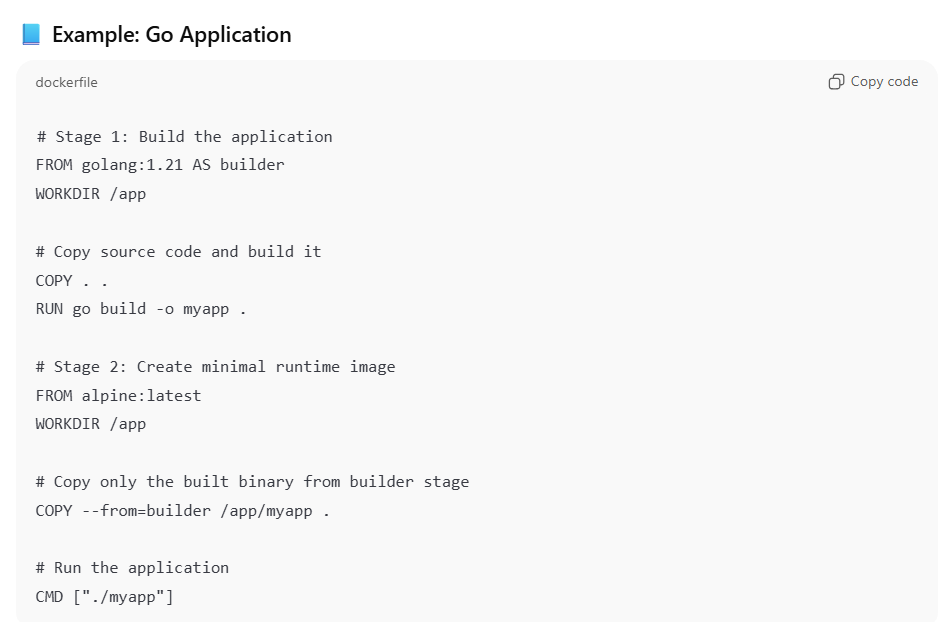
* Using **one stage** for building the application.
* Using **another stage** for running it — copying only the needed artifacts (like binaries or compiled files).

**⚙️ How It Works**

You define multiple FROM statements in a single Dockerfile.  
Each FROM starts a **new build stage**.

You can:

* Name stages using AS <name>
* Copy artifacts between stages using COPY --from=<name>



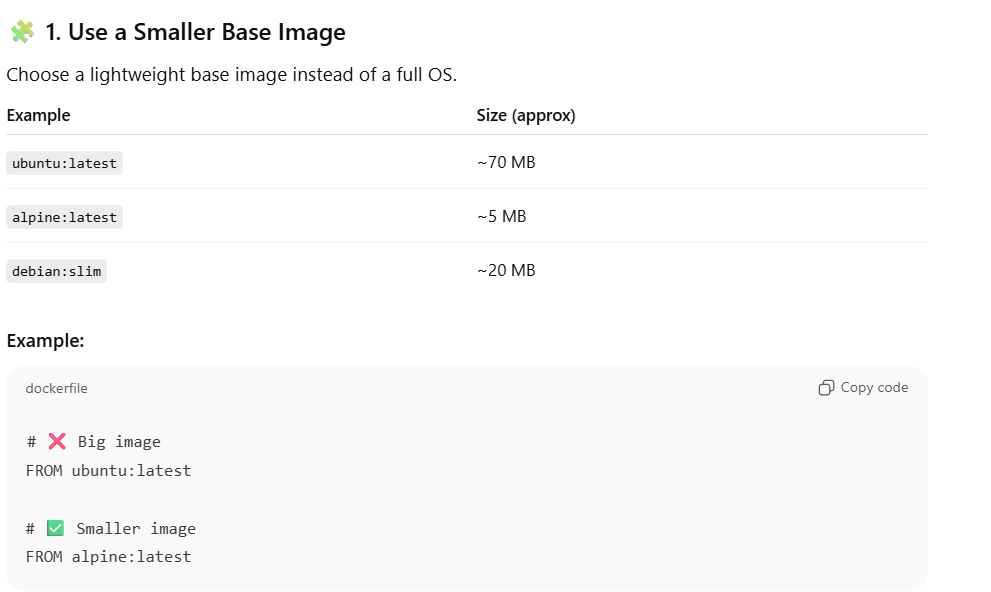
**🔍 Explanation:**

1. **golang:1.21** – used for compiling the code (large image with compiler).
2. **alpine:latest** – used for running the compiled binary (small and lightweight).
3. Only the compiled binary (myapp) is copied into the final image.
4. The final image is clean and small — no build tools, no source code.

**📦 Benefits of Multi-Stage Builds**

✅ **Smaller Image Size** – Only essential files go into the final image.  
✅ **Better Security** – No compilers or build tools left behind.  
✅ **Simplified Workflow** – Single Dockerfile instead of multiple ones for build and runtime.  
✅ **Easier Maintenance** – You can easily add or remove stages for testing or debugging.

1. How do you reduce Docker image size?



A screenshot of a computer program

AI-generated content may be incorrect.



A screenshot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

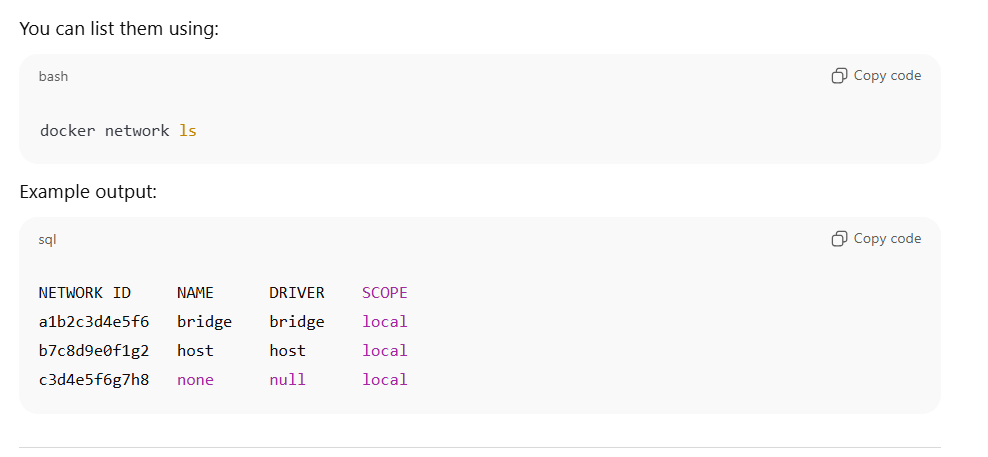
1. How does Docker handle networking by default?

By default, **Docker automatically sets up networking** so that containers can communicate with each other and the outside world — securely and in isolation.

**🌐 1. Docker’s Default Network Setup**

When you install Docker, it automatically creates **three default networks**:

| **Network Name** | **Type** | **Description** |
| --- | --- | --- |
| **bridge** | Bridge network | Default network for standalone containers |
| **host** | Host network | Shares the host’s network stack |
| **none** | Null network | No network connectivity |

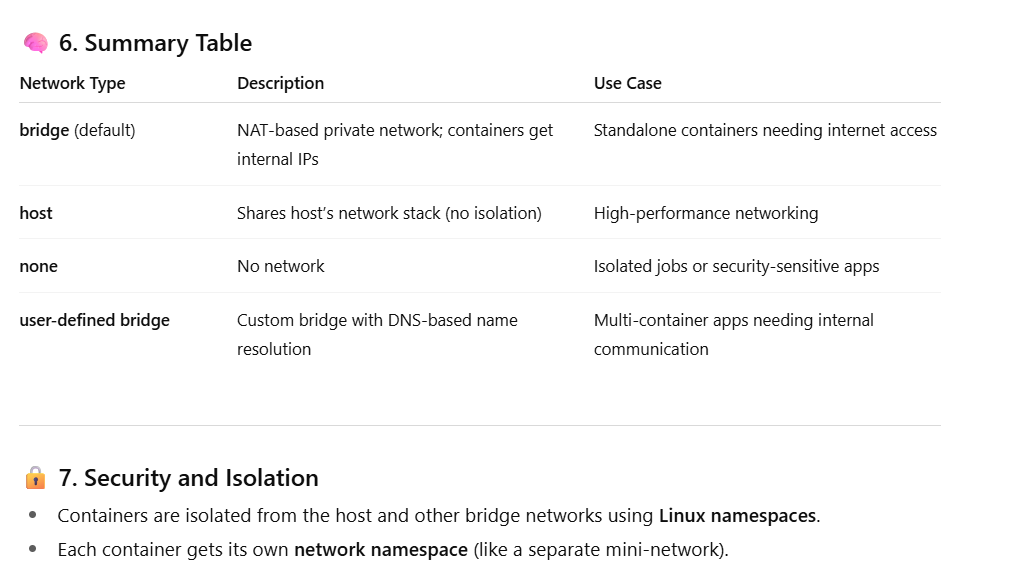


A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.



1. How does Docker DNS work?

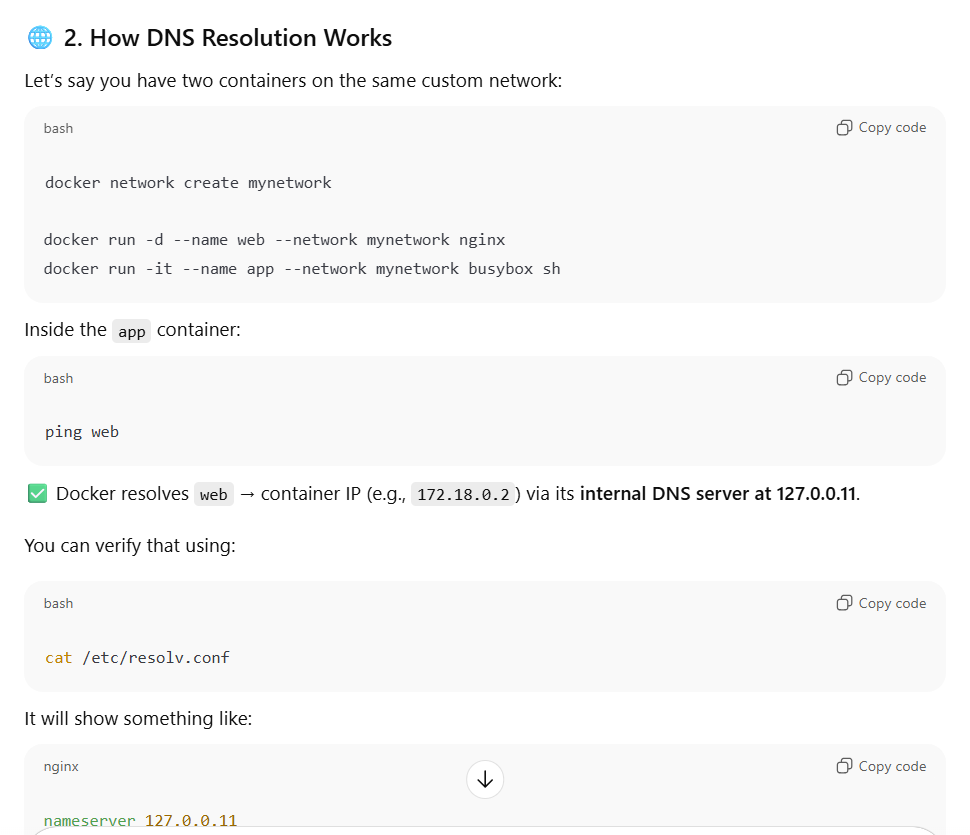
Docker has a **built-in DNS (Domain Name System)** service that lets containers **discover and communicate** with each other by **name instead of IP address** — even as container IPs change dynamically.

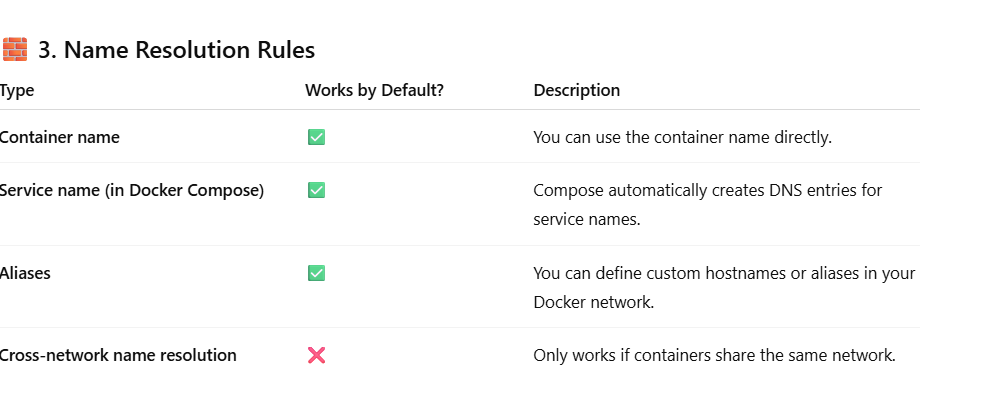
**🧩 1. Default Behavior**

When Docker starts, it automatically runs an **embedded DNS server** (usually at 127.0.0.11) inside every container.

This internal DNS:

* Resolves container names to their IP addresses.
* Works only **within the same Docker network**.
* Updates automatically when containers start, stop, or get recreated.

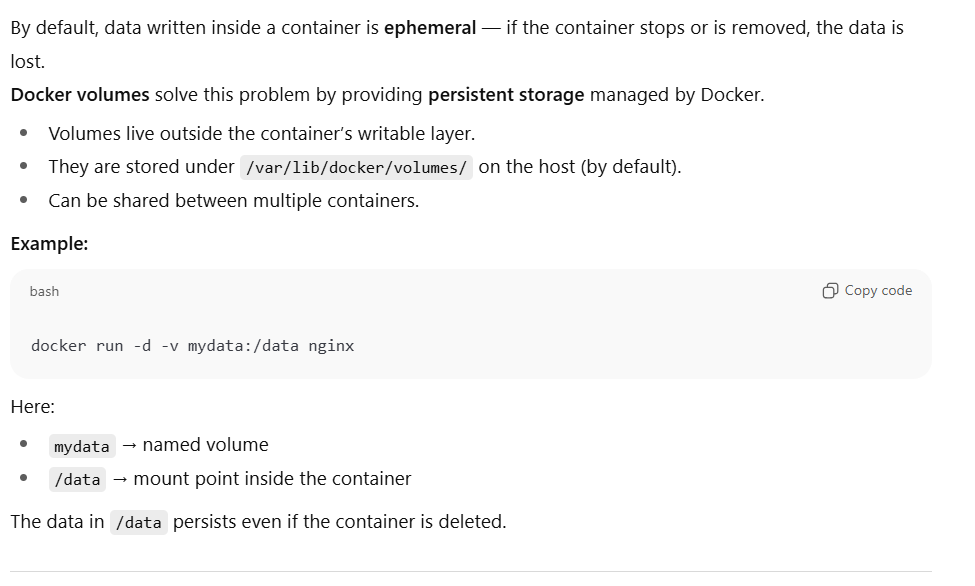




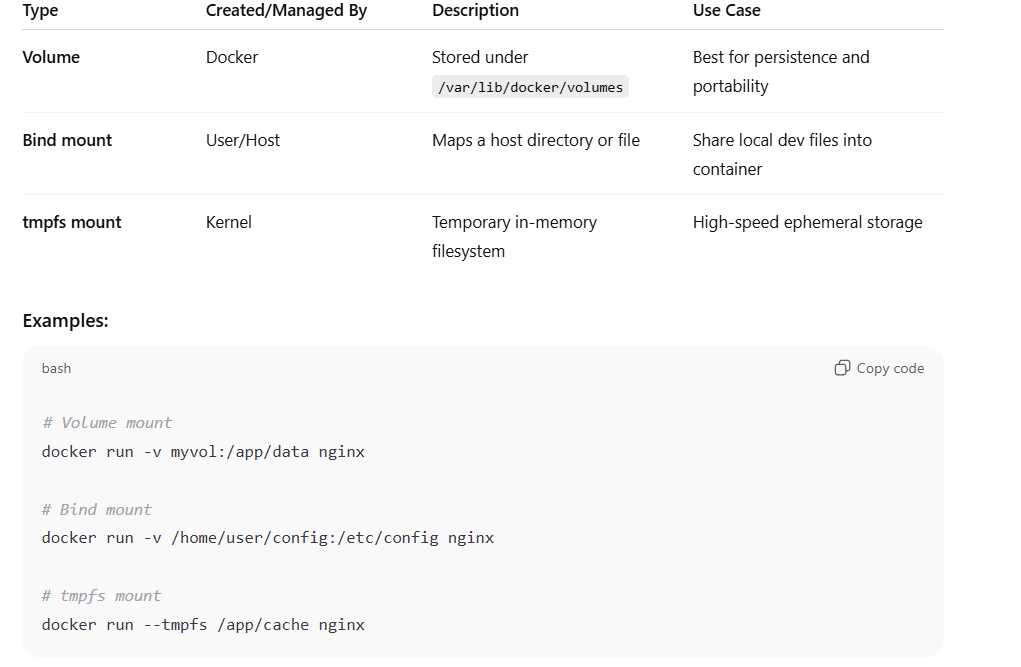
A screenshot of a computer

AI-generated content may be incorrect.

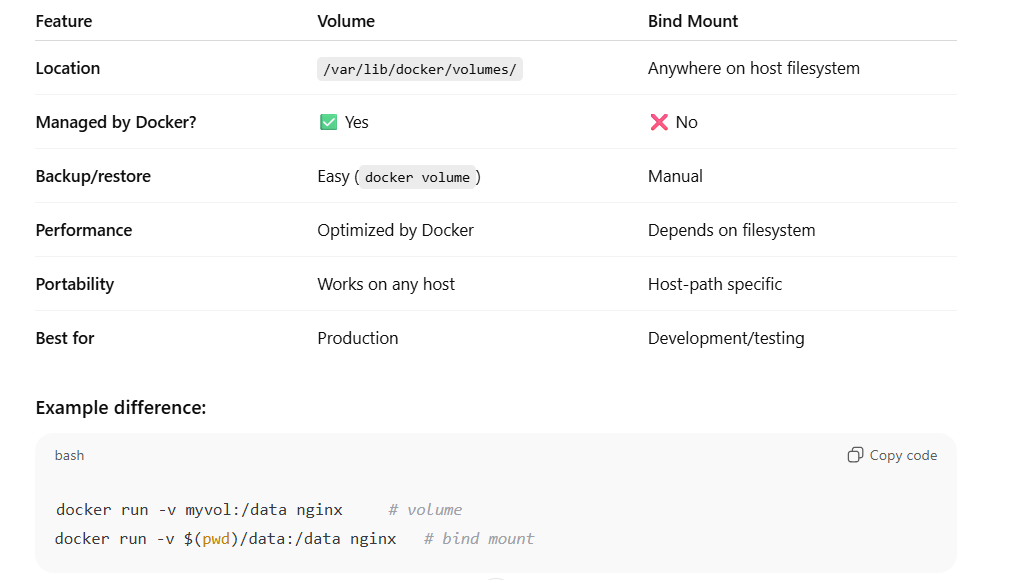
1. What are Docker volumes and why are they needed?



1. What are the different types of Docker mount types?



1. What is the difference between bind mounts and volumes?



1. How does Docker’s filesystem layering work?

When you run a container, Docker adds a **thin, writable layer** on top of these read-only layers.

So conceptually:

+-----------------------+

| Writable Container FS | ← you can modify this

+-----------------------+

| Image Layer N |

| Image Layer N-1 |

| ... |

| Base Image Layer |

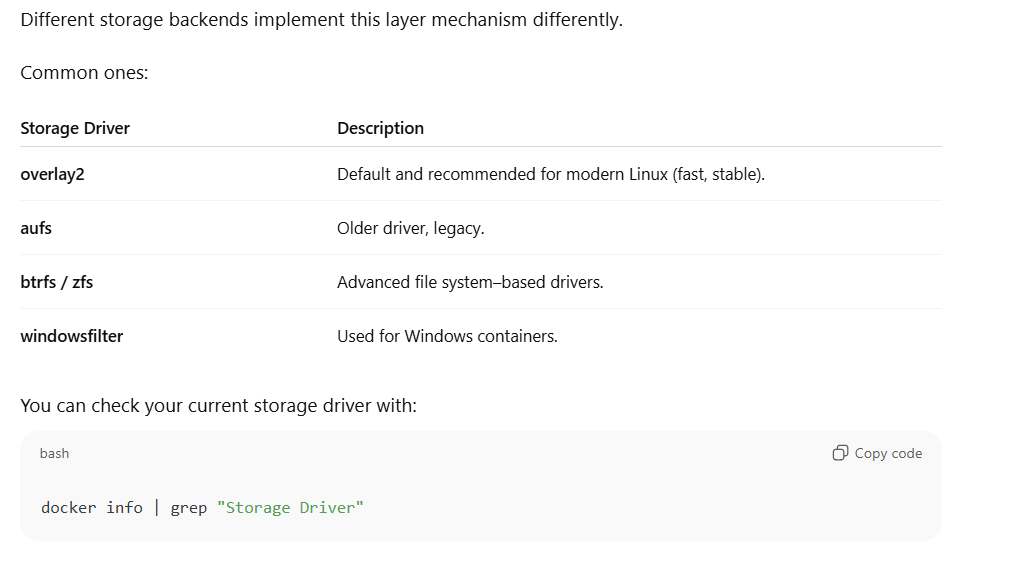
+-----------------------+

Each layer stores only **the differences (deltas)** from the previous one.

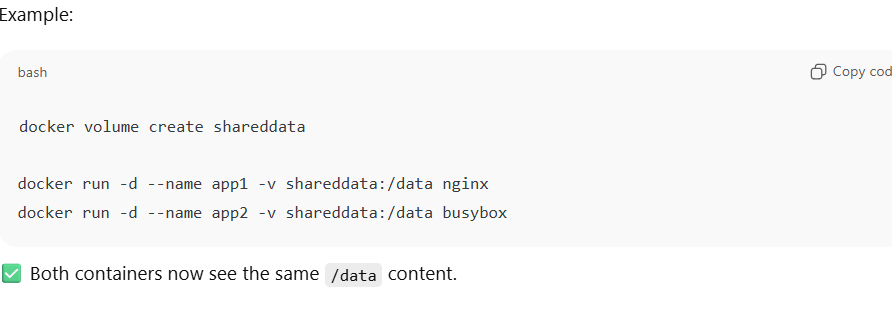
Docker uses a **Union File System** (like OverlayFS, AUFS, or Btrfs) to **merge** these layers together into one unified view.

So even though there are many layers under the hood, inside the container you see **a single coherent filesystem**.

When you cat, ls, or cd, the filesystem appears whole.



1. How can you share data between two running containers?



1. How do you clean up unused volumes?

A screenshot of a computer

AI-generated content may be incorrect.

1. **How would you debug performance issues in a running container?**

* Use docker stats for CPU/memory usage.
* Use docker top to see running processes.
* Enter container shell: docker exec -it <container> sh
* Use host tools like cgroups or sysdig.
* Check logs and events: docker inspect --format='{{.State}}'

**🚦 Step 1: Check Container Resource Usage**

Use Docker’s built-in tools:

docker stats

Shows CPU, memory, network I/O, and block I/O usage per container.

If one container is using abnormally high CPU/memory → focus there.

**🧰 Step 2: Inspect Container Configuration**

docker inspect <container\_name> | grep -E 'Cpu|Memory'

Verify if CPU/memory limits (--cpus, --memory) were set correctly.  
Sometimes no limits → a single container hogs host resources.

**🔍 Step 3: Check Processes Inside the Container**

docker exec -it <container> top

# or

docker exec -it <container> ps aux

Identify CPU/memory-hungry processes.

**📊 Step 4: Check Disk I/O Performance**

docker stats --no-stream

Check **BLOCK I/O** column.  
If it’s high → check volume or filesystem bottlenecks.

You can also use:

iotop

df -h

on the host to monitor disk usage and I/O wait.

**🌐 Step 5: Check Network Latency or Bandwidth**

docker exec -it <container> ping <other\_service>

docker exec -it <container> curl -w "%{time\_total}\n" -o /dev/null <url>

If networking is slow → investigate Docker’s bridge network or overlay network latency.

**⚙️ Step 6: Check Logs for Errors**

docker logs <container>

Slow performance can be due to:

* Application-level timeouts
* Resource starvation
* Disk write failures

**🧩 Step 7: Inspect Storage Driver and OverlayFS Performance**

docker info | grep "Storage Driver"

If using overlay2, check if many small layers or copy-on-write operations are causing slow I/O.  
For heavy write workloads, prefer volumes or tmpfs.

**🧠 Step 8: Advanced — Use Host Monitoring Tools**

Monitor the underlying host:

top

htop

vmstat 1

iostat -x 1

netstat -tupan

Identify whether the issue is at container or host level.