Problem Without Containers	Solution with Containers
"It works on my machine!" issue	Containers bundle everything
Slow deployment C heavy VMs	Containers are fast and lightweight
App crashes due to different OS	Containers work the same everywhere
Complex dependency setup	Everything is predefined in the image

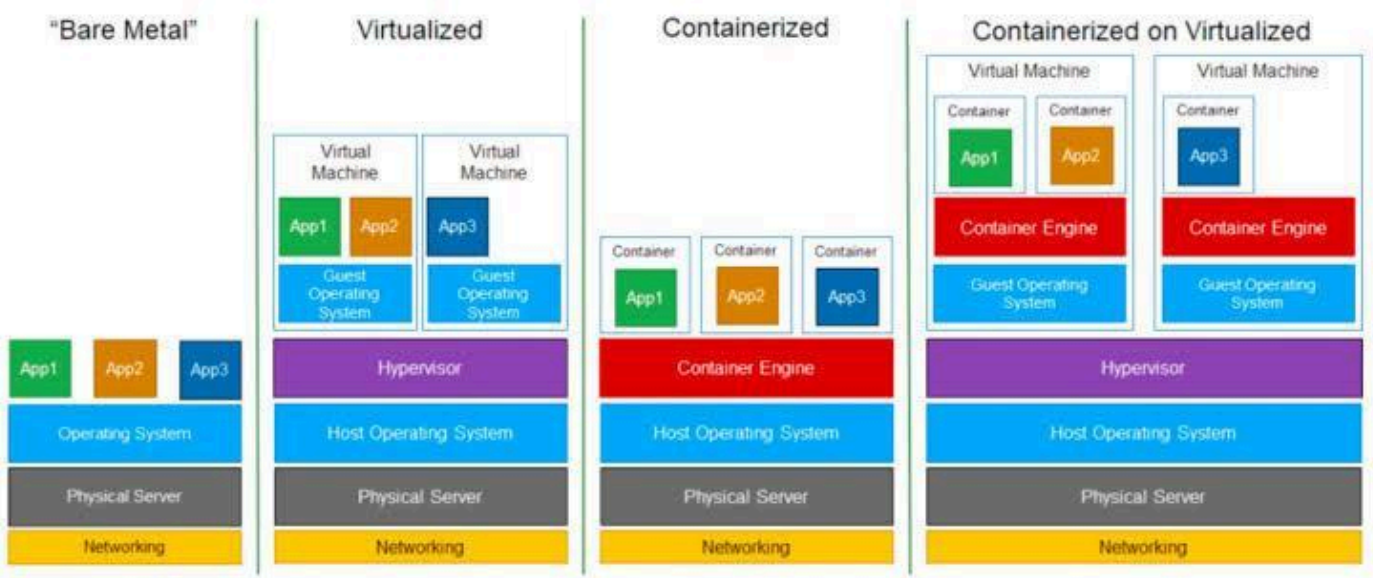
---

🐳 What is Docker?

Docker is an open source containerization platform used to create, run, and manage containers.

Docker helps **package your application and its environment** into a single container image, and run it reliably anywhere.

Virtualization vs Containerization



🔄 How Docker Works (Internally)?

1. **Docker file**
   You write instructions (like install Python, copy code, etc.)
2. **Docker Image**
   Docker reads the Docker file and builds an image.

3. **Docker Container**

Docker runs the image as an isolated app (container).

Ą Key Docker Components

Component	What It Does
Docker file	Recipe to build an image
Image	Blueprint of a container
Container	Running instance of an image
Docker Hub	Online registry to store images (like GitHub)
Volume	Store data outside the container
Network	Let containers communicate

‘\’z Basic Docker Commands

Command	Description
docker --version	Check Docker version
docker pull nginx	Download an image from Docker Hub
docker images	List all images
docker run nginx	Run container from nginx image
docker ps	List running containers
docker stop <container_id>	Stop a container
docker rm <container_id>	Remove a container
docker rmi <image_id>	Remove an image

Day to Day use Docker commands

Command	Description
docker ps -a	List all containers (running C stopped)
docker start <container_id>	Start a stopped container
docker restart <container_id>	Restart a running/stopped container
docker exec -it <container_id> sh/bash	Open a shell inside a running container
docker logs <container_id>	View logs of a container
docker inspect <container_id or image_id>	View detailed info about container/image
docker run -d <image>	Run container in detached (background) mode
docker run --name <name> <image>	Run container with a custom name
docker run -p <host_port>:<container_port> <image>	Run with port mapping
docker build -t <image_name> .	Build custom image from Dockerfile
docker image ls	List all images (alias for docker images)
docker system prune -af	Remove unused images, containers, volumes
docker volume create <volume_name>	Create a Docker volume
docker volume ls	List all volumes
docker cp <container_id>:<src> <dst>	Copy files from container to host

docker cp <src> <container_id>:<dst>	Copy files from host to container
docker stats	Display container resource usage
docker network ls	List all Docker networks
docker network create <network_name>	Create a custom Docker network

A **Sample Docker file**

```
# Start from base image
FROM node:16

# Set working directory
WORKDIR /app

# Copy files and
install COPY . .
RUN npm install


# Run the app
```

Then build and run

```
docker build -t my-node-app .
docker run -p 3000:3000 my-node-app
```

🔁 Real-World Usage in DevOps

- Deploy microservice

Isolate testing environments

- Run CI/CD pipelines (Jenkins + Docker)
- Package apps for portability (same config on dev/test/prod)

🧠 Tips to Remember

- **Image = Blueprint**
- Container = Live app
- Containers are **stateless** — use **Volumes** for saving data
- Docker removes the "Works on my machine" problem forever

Multi-Stage Docker Builds

¸*C What is a Multi-Stage Build?

- A way to use **multiple FROM instructions** in a Docker file to create **clean, small, production-ready images**.
- Helps **separate build and runtime** environments.

´'' Why Use Multi-Stage Builds?

- Avoid shipping build tools and source code into production.
- Reduce image size drastically.
- Improve security and performance.

\ ‘ Example of a Multi-Stage Docker file (Node.js)

```
# Stage 1: Build

FROM node:18 AS builder
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
RUN npm run build

# Stage 2: Serve
FROM nginx:alpine
COPY --from=builder /app/build
```

·Q Explanation

- **First stage**: Installs dependencies, builds the app.
- **Second stage**: Uses a lightweight NGINX image to serve built static files.
- Only the final built output is included in the final image.

✅ Benefits

- Smaller final image.
- Separation of concerns (build vs runtime).
- Supports any language or framework.

Distroless Container Images

''c.· ⬛·– What is a Distroless Image?

- A **distroless image** is a minimal Docker image that contains only the **application and its runtime dependencies**.

- It **does not include a package manager, shell, or other common Linux tools** (like apt, bash, curl).
- Maintained by Google.

🔍 Why Use Distroless?

- **Smaller size**: Less overhead, faster to pull and deploy.
- **More secure**: Smaller attack surface; no shell or tools means fewer vulnerabilities.
- **Immutable**: Cannot SSH or change the container at runtime.
- Ideal for **production** containers.

🧰 Common Base Images vs Distroless

Base Image	Approx Size	Contains Shell?	Securit y Risk
ubuntu	~60MB	Yes	High
alpine	~5MB	Yes	Medium
distroless/base	~2MB	No	Very Low

Ą How to Use Distroless

Here's an example of using a distroless image in your Dockerfile:

```
# Stage 1: Build the app
FROM golang:1.21 AS builder
WORKDIR /app
COPY . .
```

```
RUN go build -o main .

# Stage 2: Create minimal final image
FROM gcr.io/distroless/static
COPY --from=builder
```

.ı Limitations

- No debugging inside the container.
- Harder to troubleshoot.
- Needs multi-stage builds for compiling code.

🐳 Docker Networking

Docker containers are isolated, but they need to communicate with each other and the outside world. Docker networking allows this communication in various ways.

✅ Why Networking in Docker?

- Connect containers with each other.
- Allow containers to access external services (internet).
- Expose container services to the host or external users.

¸Ç˙·y˙ 1.Bridge Networking (Default Network Type)

◆ **What is it?**

- The **bridge network** is the default network created by Docker on a single host.
- It uses a **virtual Ethernet bridge** (docker0) on the host to allow containers to communicate **internally** and **with the internet**, but not **with containers on other hosts**.

🔹 **How it works:**

- Docker creates a **virtual bridge** on the host (like a virtual switch).
- Each container gets a **virtual Ethernet interface (veth)** that connects to this bridge.
- These interfaces are assigned **private IPs**, and containers can talk to each other using these IPs or container names.

🔹 **Use case:**

- When running multiple containers on a **single host** that need to talk to each other.

🔹 **Commands:**

```
docker network create --driver bridge my_bridge
docker run --name app1 --network my_bridge
```

🔹 Diagram:

🐳 2. Host Networking

- **What is it?**
  - Host networking **shares the host's network namespace** with the container.
  - The container **does not get its own IP address**; it uses the host's IP.
- **How it works:**
  - No virtual bridge or isolation is used.
  - Useful for **performance** and **low-latency** requirements.
  - Can be **insecure** because the container has access to all host network interfaces.
- **Use case:**
  - You want the container to behave like a native process on the host, e.g., when running Prometheus or monitoring agents.
- Command:

```
docker run --network host nginx
```

˙y¸Ç·ˇ 3. Overlay Networking

- **What is it?**

- Overlay networks enable containers running on **different Docker hosts** to communicate securely.
- Works by creating a **virtual network on top of the physical network**
  using encapsulation (VXLAN).
- Requires **Docker Swarm mode** to be active.

🔹 **How it works:**

- Uses **libnetwork** and **VXLAN tunnels**.
- Each container on any node gets an **IP address on the same overlay subnet**.

🔹 **Use case:**

- Required when you're running containers on a **multi-host cluster** using Docker Swarm or Kubernetes.

🔹 **Commands:**

```
docker swarm init
docker network create -d overlay my_overlay
```

🧠 Summary Comparison

Feature	Bridge	Host	Overlay
Host isolation	Yes	No	Yes
Cross-host	No	No	Yes
IP per container	Yes	No (uses host IP)	Yes
Use case	Single-host apps	High-performance	Multi-host apps
Needs Swarm	No	No	Yes

⬛ Containers vs Virtual Machines

Feature	Containers	Virtual Machines (VMs)
Architecture	Shares the host OS kernel	Has its own OS, runs over a hypervisor
Boot Time	Seconds	Minutes
Size	Lightweight (MBs)	Heavy (GBs)
Resource Usage	Minimal (shares kernel, low overhead)	High (each VM needs CPU, RAM, storage)
Isolation	Process-level isolation (less secure)	Full machine-level isolation (more secure)
Performance	Near-native performance	Slightly slower due to virtualization layer
Portability	Highly portable (across environments)	Less portable (depends on OS/hypervisor)
Use Case	Microservices, CI/CD pipelines, DevOps	Monolithic apps, legacy workloads
Dependency Handling	Each container has its own dependencies	VM includes dependencies in its OS image
Management Tools	Docker, Podman, Kubernetes	VMware, VirtualBox, Hyper-V, KVM

Docker components

🧱 1. Docker Engine

- **Core of Docker—responsible for building, running, and managing containers.**

- Includes:
    - Docker Daemon (dockerd): Runs in the background and manages containers/images.
    - Docker CLI (docker): Command-line tool to interact with Docker Daemon.

🧰 2. Docker Images

- Read-only templates used to create containers.
- Contains application code, runtime, libraries, dependencies, etc.
- Created using a Dockerfile.

📦 3. Docker Containers

- Lightweight, runnable instances created from Docker images.
- They are isolated and portable, behaving like mini virtual machines.

🗁 4. Docker file

- A text file with step-by-step instructions to build a Docker image.

🏗️ 5. Docker

- A tool to run multi-container Docker applications using a docker- compose.yml file.
- Helps define and manage multiple services, networks, and volumes.

🌐 6. Docker Network

- Manages communication between containers and the outside world.
- Types:
  - Bridge (default)
  - Host
  - Overlay
  - None
  - Custom networks

💾 7. Docker Volumes

- Used to persist data generated by and used by Docker containers.
- Survive container restarts/removals.

🏪 8. Docker Hub / Docker Registry

- Docker Hub is the public registry to host/share images.
- You can also use private registries like AWS ECR, GitHub Container Registry.