CONTAINERS

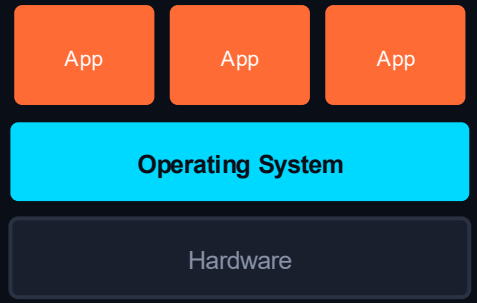
FROM SCRATCH TO PRODUCTION

Deep dive into kernel primitives and container runtimes

Technical Workshop | 2026

WHY DO WE NEED CONTAINERS?

Traditional Deployment Era



Traditional Deployment

> Applications ran directly on physical servers with no way to define resource boundaries.

Problems:

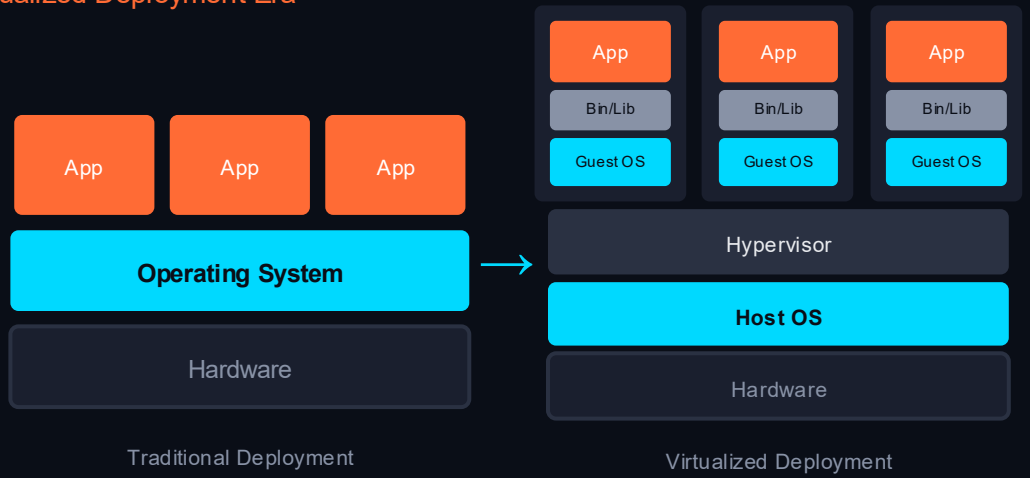
→ Resource allocation conflicts

→ Underutilized servers

→ Expensive to maintain multiple servers

WHY DO WE NEED CONTAINERS?

Virtualized Deployment Era



Traditional Deployment

Virtualized Deployment

VMs allow multiple virtual machines on a single physical server's CPU, providing isolation and better resource utilization.

Benefits:

→ Better resource utilization

→ Isolation between applications

→ Reduced hardware costs

WHY DO WE NEED CONTAINERS?

Container Deployment Era

> Containers share the OS kernel, are lightweight, and provide isolation at the process level.

Benefits:

→ Lightweight (no Guest OS overhead)

→ Fast startup times

→ Consistent across environments

→ High density deployments

THE HISTORY OF CONTAINERS

From Unix origins to cloud-native orchestration

1979 - 2000
Early Foundations

1979 Unix V7 chroot
1982 chroot in BSD
2000 FreeBSD Jails

2002 - 2008
Linux Innovations

2002 Linux namespaces
2006 cgroups (Google)
2008 LXC (Linux Containers)

2013 - 2015
Docker Revolution

2013 Docker released (dotCloud)
2014 Docker 1.0, Docker Hub
2015 OCI founded, runc released

2014 - Present
Cloud Native Era

2014 Kubernetes (Google)
2015 CNCF founded
2017 containerd donated

NAMESPACES
Isolate what processes can see: PID, network, mount, user, UTS, IPC

CGROUPS
Limit what processes can use: CPU, memory, I/O, network bandwidth

DOCKER'S CONTRIBUTION
Developer UX, image format, registry, build tooling, ecosystem

WHAT IS A CONTAINER?

> **An isolated process with controlled access to system resources**

NOT A VM
Process on host kernel

NOT MAGIC
Kernel features only

NOT SECURE
Depends on config

KEY COMPONENTS:

→ Namespaces: Isolate what a process can SEE

→ Cgroups: Limit what a process can USE

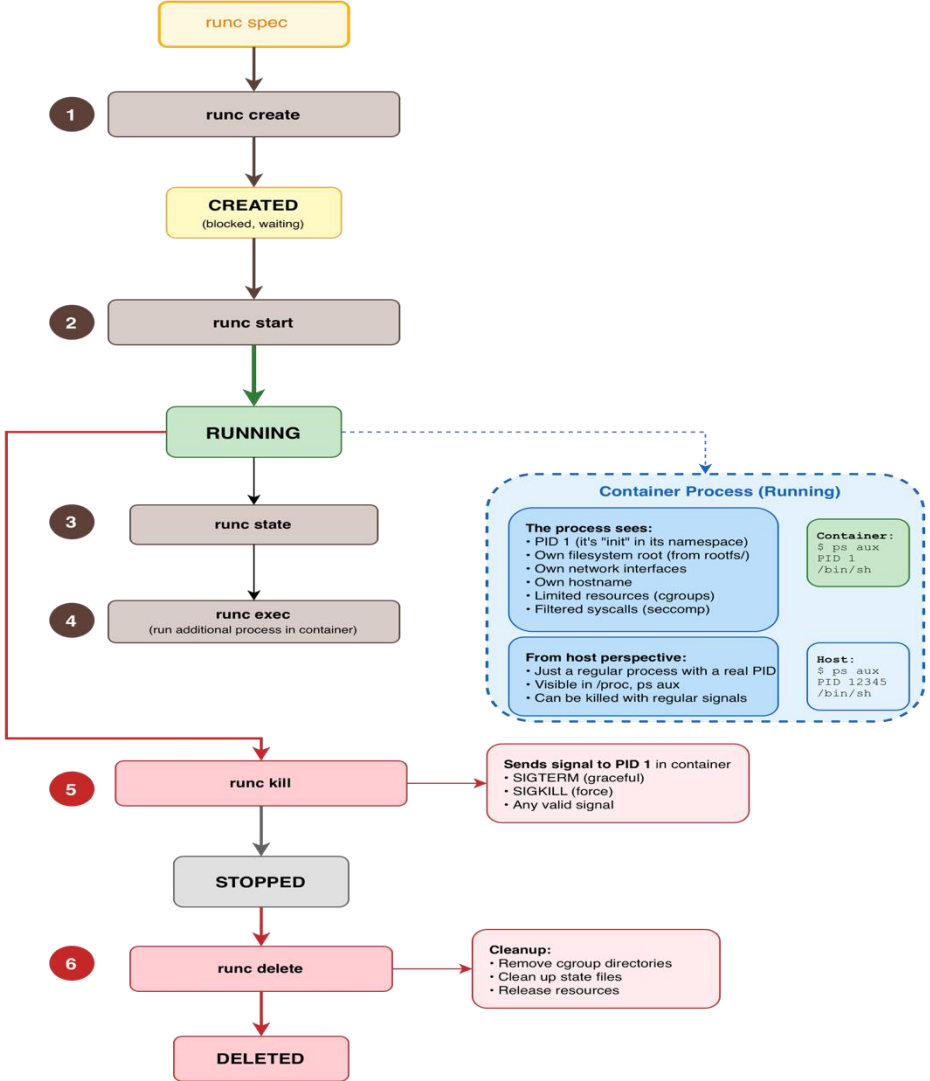
→ Union filesystems: Layer the filesystem

→ Capabilities: Fine-grained privilege control

Controlled execution environment with explicit boundaries

Container Lifecycle

runc + Linux Kernel



HOW CONTAINERS WORK

When you run a container, the following happens:

1. **CLONE syscall with namespace flags**
   Creates new namespaces (PID, mount, network, etc.). Child process lives in isolated namespace view.
2. **Cgroup setup**
   Process added to cgroup hierarchy. Resource limits enforced by kernel.
3. **Filesystem preparation**
   Mount namespace established. Root filesystem pivoted (pivot_root). Union filesystem mounted.
4. **Capabilities dropped**
   Process starts with reduced privileges. Even UID 0 inside has restricted capabilities.
5. **Process exec**
   Container runtime execs your application. Application thinks it's alone on the system.

KEY INSIGHT: Your kernel is doing ALL the work

The container runtime just orchestrates syscalls. No hypervisor, no separate kernel.

Pure kernel feature orchestration

DEMO 1

BUILD A CONTAINER FROM SCRATCH!

Building a minimal container runtime in Go

Linux

Go

DEMO 2

BUILD A CONTAINER FROM SCRATCH!

Building a minimal container with runc

Linux

Go

CONTAINER RUNTIME LANDSCAPE

Docker is NOT containers. Docker is ONE implementation.

containerd

Docker's underlying runtime. CNCF graduated. What K8s uses.

gVisor (runsc)

User-space kernel. Intercepts syscalls. Security over performance.

CRI-O

Built for Kubernetes. Implements K8s CRI. Minimal, focused.

Kata Containers

Lightweight VMs that look like containers. Own kernel per container.

runc

Low-level OCI runtime. What containerd/CRI-O use underneath.

Podman

Daemonless. Docker CLI compatible. Rootless by default.

Most production systems use containerd or CRI-O, not Docker

OPEN CONTAINER INITIATIVE (OCI)

Why It Exists and Why You Should Care

THE PROBLEM OCI SOLVES

Before OCI, Docker was the de facto standard. This created vendor lock-in risks and fragmentation as alternatives emerged. The industry needed open standards for interoperability.

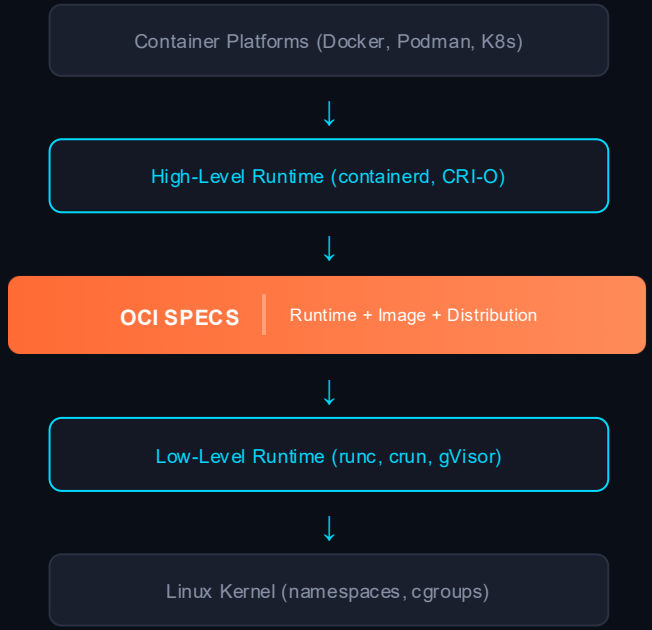
WHAT IS OCI?

Founded in 2015 by Docker, CoreOS, and others under the Linux Foundation. OCI creates open industry standards for container formats and runtimes.

THE THREE OCI SPECIFICATIONS

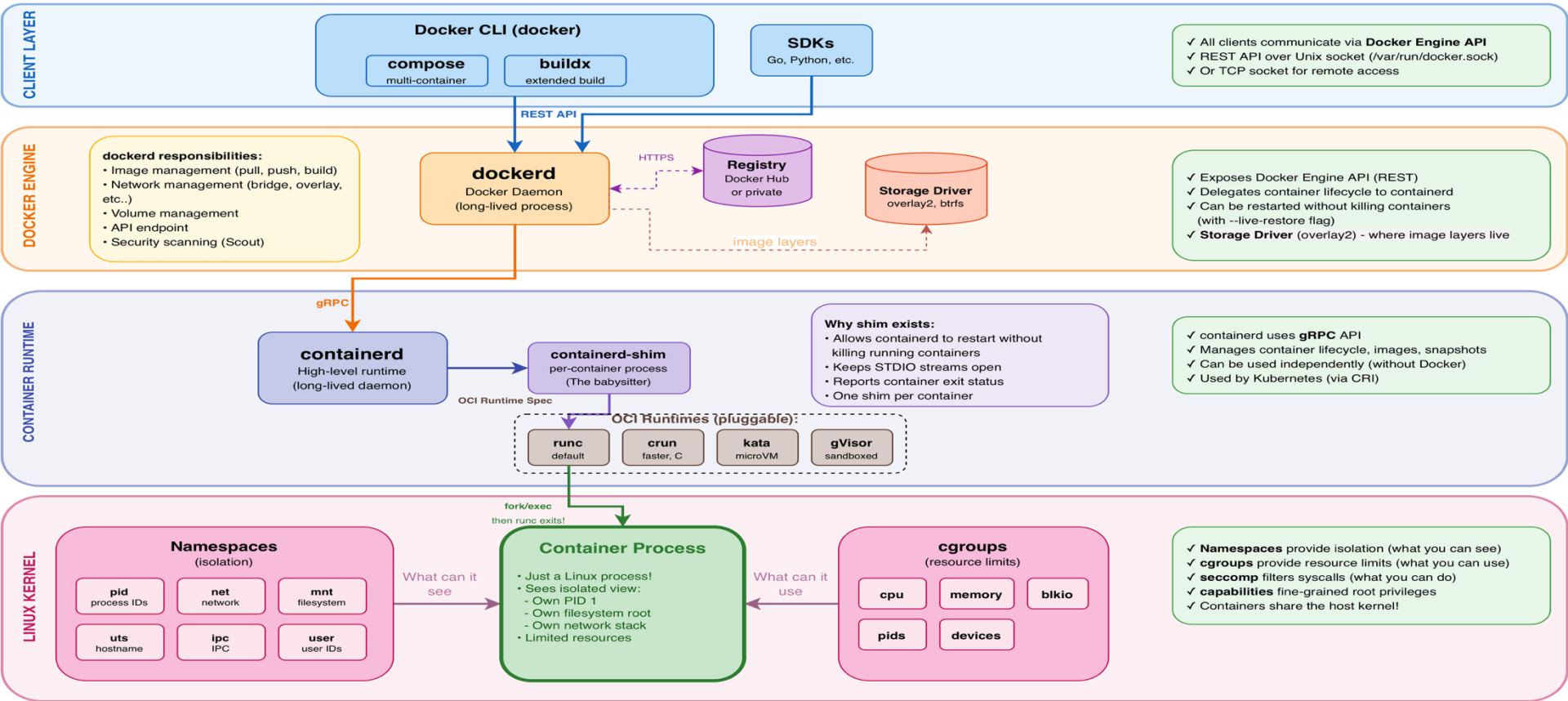
- **Runtime Spec** How to run containers
- **Image Spec** How to package containers
- **Distribution Spec** How to distribute images

WHERE OCI SITS IN THE STACK



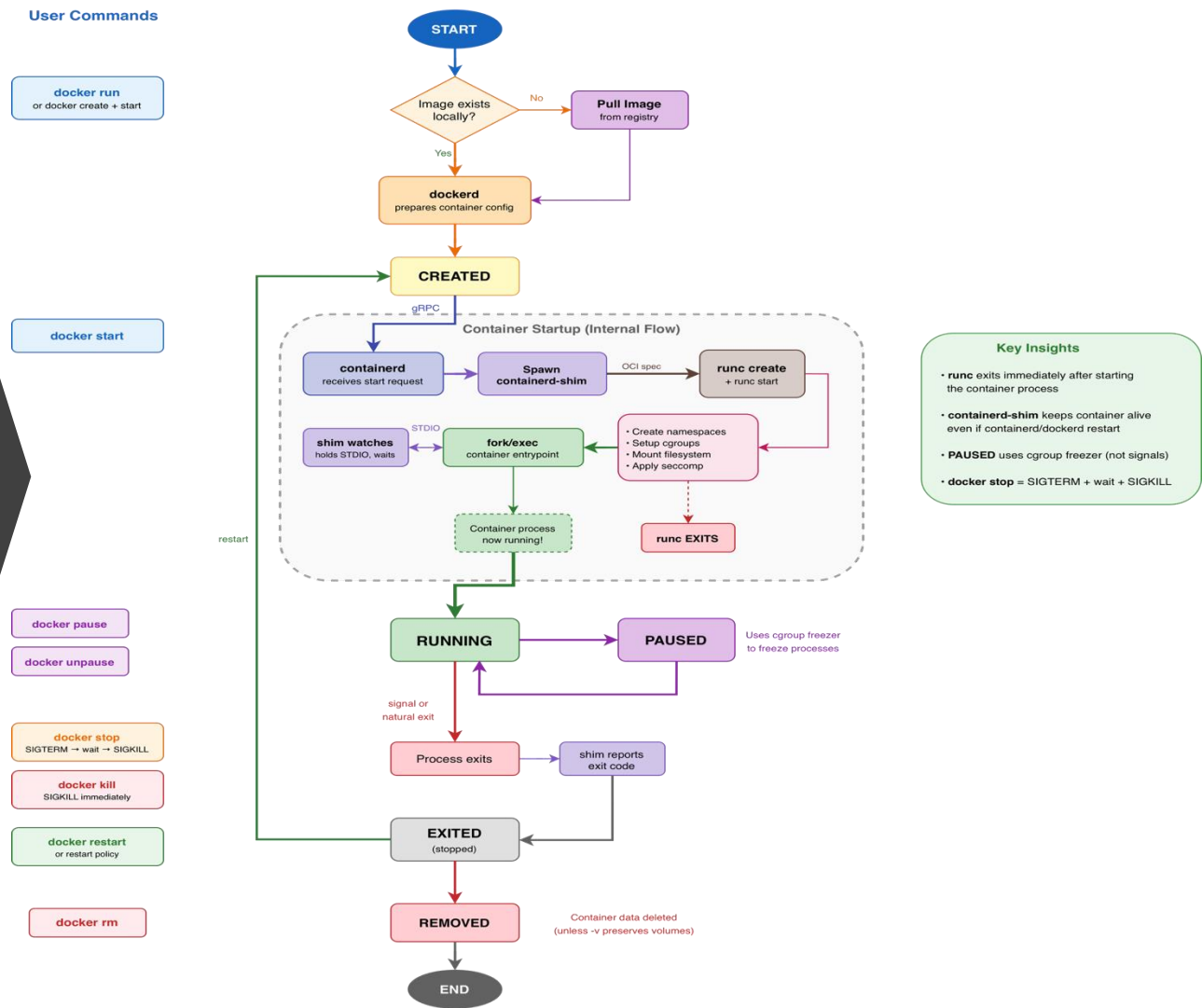
OCI ensures your containers work everywhere, regardless of tooling

DOCKER ARCHITECTURE



Container Lifecycle

Docker



ESSENTIAL DOCKER COMMANDS

LIFECYCLE

```
docker run -d --name web nginx
```

```
docker stop web
```

```
docker rm web
```

DEBUGGING

```
docker logs -f web
```

```
docker exec -it web /bin/bash
```

```
docker inspect web
```

IMAGES

```
docker build -t app:v1 .
```

```
docker pull nginx:latest
```

NETWORKING

```
docker network create mynet
```

```
docker run --network mynet app
```

Critical flags for docker run:

-d detached • -p 8080:80 port mapping • -v /host:/container volume • --memory 512m memory limit

Grouped by function for real-world workflows

DEMO 3

LET'S DOCKERIZE AN APPLICATION!

Dockerize and Containerize mini-note app

Java

Docker

DOCKER COMPOSE

Define and run multi-container applications

Single YAML file to define services, networks, and volumes

KEY FEATURES

→ Automatic service networking
→ Volume and secret management
→ Environment variable support
→ Service dependency ordering

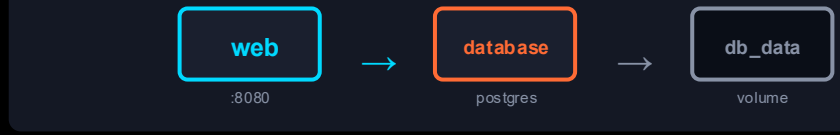
COMMANDS

`up` `down` `logs` `ps` `build`

Perfect for: Local dev, CI/CD, single-host

docker-compose.yml

```yaml
services:
  my_app:
    build:
    ports: ["8080:8080"]
    depends_on: [database]
  database_name:
    image: postgres:15
    volumes: [db_data:/var/lib/postgresql]
volumes:
  db_data:
```



DOCKERFILE DIRECTIVES

FROM
Base image. Use specific versions, prefer alpine/distroless.

RUN
Execute commands during build. Combine with && to reduce layers.

COPY vs ADD
COPY: just copy (preferred). ADD: auto-extract tar + fetch URLs.

WORKDIR
Set working directory. Don't use RUN cd /app.

ENV vs ARG
ENV: runtime + build. ARG: build-time only.

ENTRYPOINT vs CMD
ENTRYPOINT: main executable. CMD: default args (easy override).

USER
Switch user. Critical for security. Never run as root.

EXPOSE
Documents port. Doesn't actually publish.

VOLUME
Mount point. Documents stateful data location.

HEALTHCHECK
Container health verification command.

Pro tip: Use DOCKER_BUILDKIT=1 for better caching and parallelization

Each instruction creates a layer—minimize and order wisely

DOCKERFILE ANTI-PATTERNS

Common mistakes that bloat images and reduce security

```
FROM ubuntu

RUN apt-get update
RUN apt-get install -y python3
RUN apt-get install -y pip
COPY . /app
RUN pip install -r requirements.txt
CMD python3 /app/main.py
```

PROBLEMS:

- ✗ Large base image (ubuntu)
- ✗ Multiple RUN layers (wastes space)
- ✗ No layer caching optimization
- ✗ Runs as root
- ✗ No .dockerignore
- ✗ Build deps in final image
- ✗ No version pinning

Result: 500MB+ image with security vulnerabilities

Don't ship your build toolchain to production

DOCKERFILE BEST PRACTICES

Multi-stage build: Separate build from runtime

BUILD STAGE

```
FROM golang:1.21-alpine AS builder
WORKDIR /build
COPY go.mod go.sum ./
RUN go mod download
COPY . .
RUN CGO_ENABLED=0 go build -o app .
```

RUNTIME STAGE

```
FROM scratch
COPY --from=builder /build/app /app
USER 65534:65534
ENTRYPOINT ["/app"]
```

BENEFITS:

- ✓ Final image < 10MB
- ✓ No build tools in prod
- ✓ Layer caching optimized
- ✓ Non-root user
- ✓ Static binary
- ✓ Minimal attack surface

KEY TECHNIQUES:

Copy deps before source (caching)

FROM scratch or distroless base

Always specify USER directive

Ship only what you need to run, not what you needed to build

DOCKER NETWORKING

How containers communicate with each other and the outside world

bridge (default)

Private internal network. NAT to host for external access.

macvlan

Container gets MAC address. Appears as physical device.

host

Container uses host network stack directly. No isolation.

none

No networking. Complete isolation. Loopback only.

overlay

Multi-host networking for Swarm. VXLAN encapsulation.

custom

User-defined bridge. Built-in DNS. Network isolation.

Network drivers are pluggable - can write your own (CNI plugins)

Each driver trades off isolation vs performance vs complexity

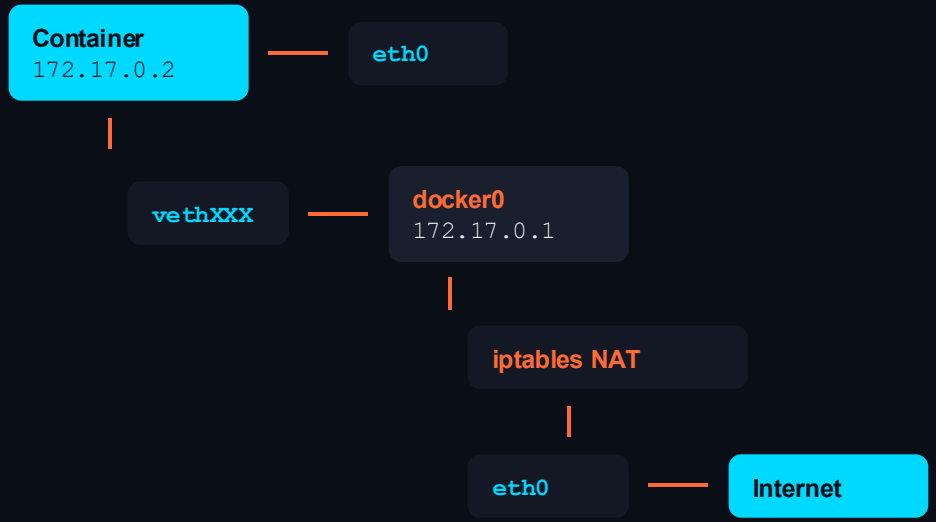
BRIDGE NETWORK INTERNALS

Default Docker networking - veth pairs + Linux bridge + iptables NAT

1. **Docker creates virtual bridge (docker0)**
   Acts like network switch. Default: 172.17.0.0/16 subnet.
2. **Create veth pair for each container**
   vethXXX (host side) ↔ eth0 (container side). Virtual ethernet cable.
3. **Attach vethXXX to docker0 bridge**
   Bridge forwards packets between containers on same network.
4. **Move eth0 into container netns**
   Container only sees eth0 with private IP (172.17.0.2, .3, etc.)
5. **iptables NAT for external traffic**
   MASQUERADE rule: container IPs → host IP for internet access.

`ip link show` • `brctl show` • `iptables -t nat -L` to inspect

PACKET FLOW: CONTAINER TO INTERNET



Source NAT:
172.17.0.2:8080 → 10.0.2.15:12345

PORT PUBLISHING (-p / -P)

Forwarding traffic from host port to container port

SYNTAX:

`-p 8080:80`
Host 8080 → Container 80

`-p 127.0.0.1:8080:80`
Only localhost can access

`-p 80`
Random host port → 80

`-P`
Publish all EXPOSE'd ports

IPTABLES RULES:

DOCKER chain in nat table:

```
DNAT tcp --dport 8080
to 172.17.0.2:80
```

Traffic flow:

1. Packet arrives at host:8080
2. DNAT changes dest IP:port
3. Routes to 172.17.0.2:80
4. Bridge forwards to container

Docker automatically manages iptables rules - don't edit manually

Check rules: iptables -t nat -L -n | grep DOCKER

DNS & SERVICE DISCOVERY

Containers find each other by name, not IP

DEFAULT BRIDGE:

NO automatic DNS

Must use --link (deprecated)
Or hardcode IPs (bad)

USER-DEFINED BRIDGE:

Built-in DNS server

Container name = hostname
Network aliases supported

HOW IT WORKS:

/etc/resolv.conf in container:

```
nameserver 127.0.0.11
```

Points to embedded DNS resolver

Resolution order:

1. Check container name on network
2. Check network aliases
3. Forward to external DNS

```
docker run --network mynet --name api backend:v1
```

Other containers: curl http://api:8080

DEMO 4

NETWORK INSPECTION For Docker

Examine Docker networking primitives in action

Linux

bash

Key Insight

Containers aren't new.

Docker made them accessible.

Kubernetes made them scalable.

OCI made them portable.

RESOURCES

- https://labs.iximiuz.com/skill-paths?category=containers
- Docker networking: https://docs.docker.com/network/
- https://learn.kodekloud.com/user/courses/docker-training-course-for-the-absolute-beginner
- https://learn.kodekloud.com/user/courses/docker-certified-associate-exam-course
- https://www.udemy.com/course/complete-docker-kubernetes
- https://github.com/nicolaka/netshoot
- https://github.com/helayoty/cloud-native-in-arabic

THANK YOU!

@helayoty

Questions? Let's discuss!