Docker & Container Optimization (Production-Grade)

This module explains **what Docker is doing under the hood**, *why* each optimization matters, and *how* to apply it in real production systems. The goal is not just to write Dockerfiles, but to **engineer secure, fast, and minimal containers**.

---

1. Background: How Docker Really Works

1.1 Containers vs Virtual Machines

Feature	Virtual Machine	Container
OS	Full guest OS	Shares host kernel
Startup time	Minutes	Seconds
Resource usage	Heavy	Lightweight
Isolation	Strong	Process-level

Containers: - Run as **processes on the host kernel** - Isolation via **namespaces** (PID, NET, FS) - Resource control via **cgroups**

Key implication:

> Anything you add to an image directly affects **startup time, attack surface, and memory usage**.

---

2. Docker Images & Layers (Core Concept)

2.1 What Is a Docker Image?

A Docker image is: - A **stack of immutable layers** - Each instruction in Dockerfile → new layer

Example:

```
FROM python:3.12-slim   ← Layer 1
COPY . /app             ← Layer 2
RUN pip install ...     ← Layer 3
```

If layer 3 changes, Docker **rebuilds only from that layer downward**.

2.2 Why Layer Caching Matters

Good caching: - Faster builds - Faster CI/CD - Less bandwidth usage

Bad caching: - Every change triggers full rebuild

Rule:

> Stable instructions go **first**, frequently-changing ones go **last**.

---

3. Multi-Stage Builds (Most Important Optimization)

3.1 The Problem

Traditional Dockerfiles mix: - Build tools (gcc, node, pip) - Runtime binaries

Result: - Huge images - Larger attack surface

3.2 Multi-Stage Build Concept

Split the image into stages: 1. **Builder stage** → compile / install dependencies 2. **Runtime stage** → copy only what is needed

---

3.3 Example: Python Multi-Stage Dockerfile

```
# ===== Builder Stage =====
FROM python:3.12 AS builder
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt --prefix=/install

# ===== Runtime Stage =====
FROM python:3.12-slim
WORKDIR /app
COPY --from=builder /install /usr/local
COPY . .
CMD ["python", "app.py"]
```

✔️ Removes compilers & build cache from final image

3.4 Example: Node.js Multi-Stage Build

```
FROM node:20 AS builder
WORKDIR /app
COPY package*.json ./
RUN npm ci
COPY . .
RUN npm run build

FROM node:20-alpine
WORKDIR /app
COPY --from=builder /app/dist ./dist
COPY --from=builder /app/node_modules ./node_modules
CMD ["node", "dist/index.js"]
```

---

4. Base Image Selection (Critical Security Decision)

4.1 Common Base Images

Image	Size	Security	Use Case
ubuntu	~70MB	Medium	Debug-heavy
slim	~30MB	Good	General apps
alpine	~5MB	Good	Small services
distroless	~2MB	Excellent	Production
scratch	0MB	Maximum	Static binaries

4.2 Distroless Example

```
FROM gcr.io/distroless/python3
COPY app.py /
CMD ["/app.py"]
```

Benefits: - No shell - No package manager - Fewer CVEs

---

5. Instruction Ordering & Layer Optimization

Bad Dockerfile

```
COPY . .
RUN pip install -r requirements.txt
```

Every code change → reinstall dependencies ❌

Optimized Dockerfile

```
COPY requirements.txt .
RUN pip install -r requirements.txt
COPY . .
```

✔️ Dependencies cached

---

6. RUN Command Optimization

Each RUN creates a layer:

Bad

```
RUN apt update
RUN apt install -y curl
RUN rm -rf /var/lib/apt/lists/*
```

Good

```
RUN apt update &&
    apt install -y curl &&
    rm -rf /var/lib/apt/lists/*
```

✔️ Fewer layers ✔️ Smaller image

---

7. .dockerignore (Often Forgotten)

Why It Matters

Without `.dockerignore`, Docker sends **entire context** to daemon.

Example `.dockerignore`

```
.git
node_modules
__pycache__
.env
venv
*.log
```

✔️ Faster builds ✔️ Smaller images ✔️ Fewer secrets leaked

---

8. Security Best Practices for Containers

8.1 Run as Non-Root

```
RUN useradd -m appuser
USER appuser
```

8.2 Pin Image Versions

```
FROM python:3.12.1-slim
```

Avoid:

```
FROM python:latest
```

8.3 Vulnerability Scanning

Tools: - Trivy - Docker Scout - Grype

Example:

```
trivy image myapp:latest
```

Goal:

> Zero **CRITICAL** vulnerabilities

---

9. Image Size Optimization (Before / After)

Example Results

Stage	Image Size
Single-stage	900 MB
Multi-stage + slim	180 MB
Distroless	65 MB

✔️ Checklist: Size optimization documented

---

10. Pushing Images to Registry

Docker Hub

```
docker tag myapp user/myapp:1.0
docker push user/myapp:1.0
```

Private Registries

- AWS ECR
- GCP Artifact Registry
- GitHub Container Registry

---

11. Real Production Mindset

What senior engineers look for: - Minimal images - Deterministic builds - Zero critical CVEs - Fast CI pipelines - Clear separation of build vs runtime

---

Daily Completion Checklist

- ☑️ Multi-stage Dockerfiles created
- ☑️ Image size optimized (before/after recorded)
- ☑️ Security scan completed (0 critical CVEs)
- ☑️ Images pushed to registry

Key Interview Insight

> Docker is not about "it works on my machine". Docker is about **repeatability, security, and efficiency**.

If you want next: - Kubernetes basics (Pods, Deployments) - Docker + CI/CD pipelines - Container security deep dive - Hands-on optimization challenge