

Docker Reference

From first container to production-ready data pipeline environments.

Last reviewed 29 Sep 2026 ·

Prerequisites: [Linux & Bash](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/linux-bash/) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/linux-bash/)

Related: [Airflow](https://sarangambekar1997.github.io/data-engineering-handbook/03-orchestration/airflow-reference/) (https://sarangambekar1997.github.io/data-engineering-handbook/03-orchestration/airflow-reference/) · [Kubernetes](https://sarangambekar1997.github.io/data-engineering-handbook/06-infrastructure/kubernetes-for-de/) (https://sarangambekar1997.github.io/data-engineering-handbook/06-infrastructure/kubernetes-for-de/) · [Terraform](https://sarangambekar1997.github.io/data-engineering-handbook/06-infrastructure/terraform-for-de/) (https://sarangambekar1997.github.io/data-engineering-handbook/06-infrastructure/terraform-for-de/) · [Testing and CI/CD](https://sarangambekar1997.github.io/data-engineering-handbook/06-infrastructure/testing-cicd/) (https://sarangambekar1997.github.io/data-engineering-handbook/06-infrastructure/testing-cicd/) · [Local LLMs](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/local-llms/) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/local-llms/) · [Glossary](https://sarangambekar1997.github.io/data-engineering-handbook/99-reference/glossary/) (https://sarangambekar1997.github.io/data-engineering-handbook/99-reference/glossary/)

Overview

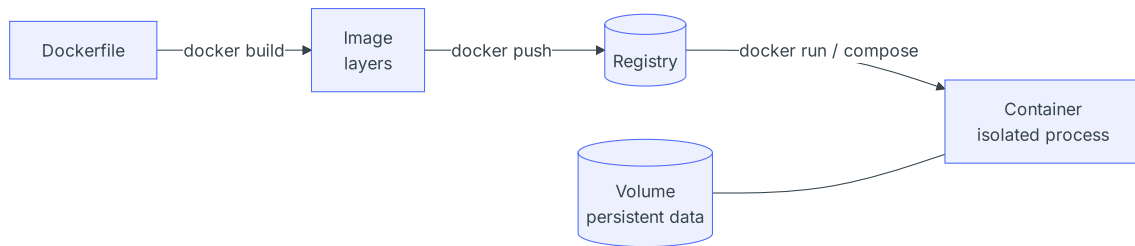
Challenge: A pipeline depends on a specific Python version, pinned libraries, a Java runtime for Spark, and database drivers. Differences between developer machines, CI, and production servers cause failures that are hard to reproduce.

Solution: Docker packages the complete environment, described in a `Dockerfile`, into an immutable **image** that runs as a **container** anywhere — a laptop, CI, Kubernetes, or a managed batch service. The same image runs identically in every environment.

```
Dockerfile —build→ Image (versioned, immutable)
—push→ Registry (ECR / GHCR / Docker Hub)
  recipe                e.g. orders-etl:1.4.2
|
pull + run anywhere

laptop · CI · Kubernetes · Airflow
```

Typical uses in data engineering: running an orchestrator, database, message broker, or Spark locally with Docker Compose; packaging pipeline jobs so the orchestrator can run them in isolation; and building reproducible CI environments for transformation and Spark tests.



On this page

Basics - [What is Docker?](https://sarangambekar1997.github.io/data-engineering-handbook/06-infrastructure/docker-reference/#what-is-docker) (https://sarangambekar1997.github.io/data-engineering-handbook/06-infrastructure/docker-reference/#what-is-docker) - [Core Concepts](https://sarangambekar1997.github.io/data-engineering-handbook/06-infrastructure/docker-reference/#core-concepts) (https://sarangambekar1997.github.io/data-engineering-handbook/06-infrastructure/docker-reference/#core-concepts) - [Essential CLI Commands](https://sarangambekar1997.github.io/data-engineering-handbook/06-infrastructure/docker-reference/#essential-cli-commands) (https://sarangambekar1997.github.io/data-engineering-handbook/06-infrastructure/docker-reference/#essential-cli-commands) - [Dockerfile](https://sarangambekar1997.github.io/data-engineering-handbook/06-infrastructure/docker-reference/#dockerfile) (https://sarangambekar1997.github.io/data-engineering-handbook/06-infrastructure/docker-reference/#dockerfile) - [Images & Layers](https://sarangambekar1997.github.io/data-engineering-handbook/06-infrastructure/docker-reference/#images--layers) (https://sarangambekar1997.github.io/data-engineering-handbook/06-infrastructure/docker-reference/#images--layers)

Intermediate - [Volumes & Bind Mounts](https://sarangambekar1997.github.io/data-engineering-handbook/06-infrastructure/docker-reference/#volumes--bind-mounts) (https://sarangambekar1997.github.io/data-engineering-handbook/06-infrastructure/docker-reference/#volumes--bind-mounts) - [Networking](https://sarangambekar1997.github.io/data-engineering-handbook/06-infrastructure/docker-reference/#networking) (https://sarangambekar1997.github.io/data-engineering-handbook/06-infrastructure/docker-reference/#networking) - [Environment Variables & Secrets](https://sarangambekar1997.github.io/data-engineering-handbook/06-infrastructure/docker-reference/#environment-variables--secrets) (https://sarangambekar1997.github.io/data-engineering-handbook/06-infrastructure/docker-reference/#environment-variables--secrets) - [Docker Compose](https://sarangambekar1997.github.io/data-engineering-handbook/06-infrastructure/docker-reference/#docker-compose) (https://sarangambekar1997.github.io/data-engineering-handbook/06-infrastructure/docker-reference/#docker-compose)

Advanced - [Multi-Stage Builds](https://sarangambekar1997.github.io/data-engineering-handbook/06-infrastructure/docker-reference/#multi-stage-builds) (https://sarangambekar1997.github.io/data-engineering-handbook/06-infrastructure/docker-reference/#multi-stage-builds) - [Docker for DE Pipelines](https://sarangambekar1997.github.io/data-engineering-handbook/06-infrastructure/docker-reference/#docker-for-de-pipelines) (https://sarangambekar1997.github.io/data-engineering-handbook/06-infrastructure/docker-reference/#docker-for-de-pipelines) - [Running Airflow in Docker](https://sarangambekar1997.github.io/data-engineering-handbook/06-infrastructure/docker-reference/#running-airflow-in-docker) (https://sarangambekar1997.github.io/data-engineering-handbook/06-infrastructure/docker-reference/#running-airflow-in-docker) - [Best Practices](https://sarangambekar1997.github.io/data-engineering-handbook/06-infrastructure/docker-reference/#best-practices) (https://sarangambekar1997.github.io/data-engineering-handbook/06-infrastructure/docker-reference/#best-practices)

Reference - [Common Pitfalls](https://sarangambekar1997.github.io/data-engineering-handbook/06-infrastructure/docker-reference/#common-pitfalls) (https://sarangambekar1997.github.io/data-engineering-handbook/06-infrastructure/docker-reference/#common-pitfalls) - [Cheat Sheet](https://sarangambekar1997.github.io/data-engineering-handbook/06-infrastructure/docker-reference/#cheat-sheet) (https://sarangambekar1997.github.io/data-engineering-handbook/06-infrastructure/docker-reference/#cheat-sheet) - [Interview Questions](https://sarangambekar1997.github.io/data-engineering-handbook/06-infrastructure/docker-reference/#interview-questions) (https://sarangambekar1997.github.io/data-engineering-handbook/06-infrastructure/docker-reference/#interview-questions) - [Further Reading](https://sarangambekar1997.github.io/data-engineering-handbook/06-infrastructure/docker-reference/#further-reading) (https://sarangambekar1997.github.io/data-engineering-handbook/06-infrastructure/docker-reference/#further-reading)

What is Docker?

Docker packages an application and all its dependencies (libraries, configs, runtime) into a **container** — an isolated, reproducible environment that runs the same on any machine.

Without Docker:	With Docker:
"Works on my machine" →	Runs identically everywhere
Manual dependency install →	One command: <code>docker run</code>
"Python 3.8 or 3.11?" →	Pinned inside the container
Dev ≠ Prod environment →	Same image everywhere

VM vs Container

Virtual Machine:	Container:
App Libraries Guest OS heavy lightweight Hypervisor Host OS slow	App Libraries Docker Engine Host OS fast
~GB, minutes to start	~MB, seconds to start

Core Concepts

Concept	Definition
Image	A read-only template — the blueprint for a container. Built from a Dockerfile
Container	A running instance of an image — isolated process with its own filesystem
Dockerfile	A text file with instructions to build an image
Registry	A repository for images (Docker Hub, ECR, GCR, GHCR)
Layer	Each Dockerfile instruction creates a cached layer; layers are shared across images

Concept	Definition
Volume	Persistent storage that outlives the container
Bind mount	Mount a host directory into a container
Network	Virtual network connecting containers
docker-compose	Tool to define and run multi-container applications via YAML

Essential CLI Commands

```
# — Images —————
docker pull python:3.11-slim      # download image from registry
docker images                    # list local images
docker rmi python:3.11-slim      # delete image
docker image prune               # delete dangling images

# — Containers —————
docker run python:3.11-slim      # create + start container (foreground)
docker run -d python:3.11-slim sleep infinity # -d = detached (background)
docker run -it python:3.11-slim bash # -it = interactive terminal
docker run --rm python:3.11-slim python -c "print('hello')"
# --rm = auto-delete on exit

docker ps                       # list running containers
docker ps -a                    # list all containers (including stopped)
docker stop <id or name>        # graceful stop (SIGTERM → SIGKILL after timeout)
docker kill <id or name>        # immediate stop (SIGKILL)
docker rm <id or name>          # delete container
docker rm $(docker ps -aq)      # delete all stopped containers
```

```

# — Exec into a running container —————
docker exec -it <container_id> bash      # open bash shell
docker exec <container_id> python manage.py migrate

# — Logs —————
docker logs <container_id>
docker logs -f <container_id>           # follow (tail -f)
docker logs --tail 100 <container_id>

# — Build —————
docker build -t myapp:1.0 .              # build from
./Dockerfile, tag as myapp:1.0
docker build -t myapp:1.0 -f docker/Dockerfile . # custom
Dockerfile path
docker build --no-cache -t myapp:1.0 . # rebuild without
layer cache

# — Push to registry —————
docker tag myapp:1.0 myrepo/myapp:1.0
docker push myrepo/myapp:1.0

# — System cleanup —————
docker system prune                # remove unused images,
containers, networks
docker system prune -a            # also remove unused images
(not just dangling)
docker system df                  # show disk usage

```

Dockerfile

```

# Base image — always pin a specific version tag, never use
:latest
FROM python:3.11-slim

# Metadata
LABEL maintainer="maintainer@example.com"
LABEL version="1.0"

# Set working directory inside the container

```

```

WORKDIR /app

# Copy dependency files first (leverages layer caching)
# If requirements.txt doesn't change, this layer is cached
COPY requirements.txt .

# Install dependencies
RUN pip install --no-cache-dir -r requirements.txt

# Copy application code
COPY src/ ./src/
COPY config/ ./config/

# Environment variables with defaults
ENV PYTHONUNBUFFERED=1 \
    PYTHONDONTWRITEBYTECODE=1 \
    LOG_LEVEL=INFO

# Expose a port (documentation only — doesn't actually
publish)
EXPOSE 8080

# Create a non-root user (security best practice)
RUN useradd -m -u 1000 appuser
USER appuser

# Default command when container starts
CMD ["python", "src/main.py"]

# Or use ENTRYPOINT for fixed command + CMD for default
args
ENTRYPOINT ["python", "src/pipeline.py"]
CMD ["--mode", "daily"]      # can be overridden at runtime

```

RUN vs CMD vs ENTRYPOINT

Instruction	When it runs	Can be overridden?
<code>RUN</code>	At build time — creates a layer	No
<code>CMD</code>	Container start — default command	Yes (<code>docker run myapp custom_command</code>)

Instruction	When it runs	Can be overridden?
<code>ENTRYPOINT</code>	Container start — fixed executable	Hard to override (use <code>--entrypoint</code>)

Images & Layers

```
# Each instruction = a new layer
FROM python:3.11-slim          # Layer 1 (base)
WORKDIR /app                  # Layer 2
COPY requirements.txt .        # Layer 3
RUN pip install -r requirements.txt # Layer 4 ← expensive, cache this
COPY src/ .                    # Layer 5 ← code changes often; goes last
```

Layer caching rule: Docker reuses a cached layer if the instruction AND all previous layers are unchanged. Put frequently-changing steps (COPY source code) AFTER rarely-changing steps (pip install).

```
# Bad order — any code change busts the pip install cache
COPY . .
RUN pip install -r requirements.txt

# Good order — code changes don't invalidate pip cache
COPY requirements.txt .
RUN pip install -r requirements.txt
COPY . .
```

Volumes & Bind Mounts

```
# Named volume — managed by Docker, persists across container restarts
docker run -v mydata:/app/data myapp

# Bind mount — mount a host directory
docker run -v /host/path:/container/path myapp
docker run -v $(pwd)/data:/app/data myapp # current
```

directory

Read-only bind mount

```
docker run -v $(pwd)/config:/app/config:ro myapp
```

List volumes

```
docker volume ls
```

```
docker volume inspect mydata
```

```
docker volume rm mydata
```

When to use each

	Named Volume	Bind Mount
Data managed by Docker?	Yes	No — you control the host path
Dev: hot reload code changes?	No	Yes — edit on host, reflects in container
Production data persistence?	Yes	Depends on host path
Works on all OS?	Yes	Path differences on Windows

Networking

Create a network

```
docker network create my-network
```

Connect containers to a network

```
docker run -d --name postgres --network my-network  
postgres:15
```

```
docker run -d --name airflow --network my-network  
apache/airflow:2.9.0
```

Containers on the same network can reach each other by name

Inside the airflow container: connect to "postgres:5432"

Publish a port — map host port to container port

```
docker run -p 8080:8080 myapp # host:container
```

```
docker run -p 5432:5432 postgres
```

```
# List networks
docker network ls
docker network inspect my-network
```

Environment Variables & Secrets

```
# Pass env vars at runtime
docker run -e DB_HOST=localhost -e DB_PORT=5432 myapp
```

```
# Load from a .env file
docker run --env-file .env myapp
```

```
# .env file format
DB_HOST=localhost
DB_PORT=5432
DB_PASSWORD=secret
```

```
# In Dockerfile – build-time defaults (don't put secrets
here)
ENV DB_PORT=5432
ENV LOG_LEVEL=INFO
```

```
# Never hardcode secrets in a Dockerfile or commit .env
files
# Use Docker secrets (Swarm) or mount secrets at runtime
```

Docker Compose

Docker Compose defines multi-container applications in a single YAML file.

```
# docker-compose.yml (Compose V2 – the old top-level
`version:` key is obsolete and ignored)
services:
  postgres:
    image: postgres:15
    environment:
```

```

    POSTGRES_USER: airflow
    POSTGRES_PASSWORD: airflow
    POSTGRES_DB: airflow
volumes:
  - postgres_data:/var/lib/postgresql/data
ports:
  - "5432:5432"
healthcheck:
  test: ["CMD", "pg_isready", "-U", "airflow"]
  interval: 10s
  retries: 5

redis:
  image: redis:7-alpine
  ports:
    - "6379:6379"

myapp:
  build:
    context: .
    dockerfile: Dockerfile
  image: myapp:latest
  depends_on:
    postgres:
      condition: service_healthy      # wait for
healthcheck to pass
  environment:
    - DB_HOST=postgres                # use service name as
hostname
    - DB_PORT=5432
  env_file:
    - .env                            # load additional vars
from .env
  volumes:
    - ./src:/app/src                  # bind mount for
development
    - ./logs:/app/logs
  ports:
    - "8080:8080"
  restart: unless-stopped            # auto-restart on
failure

```

```
volumes:
  postgres_data:           # named volume
  declaration
```

```
# Docker Compose commands
docker compose up           # start all services
(foreground)
docker compose up -d        # start in background
docker compose up --build   # rebuild images
before starting
docker compose down         # stop and remove
containers
docker compose down -v      # also remove volumes
docker compose logs -f myapp # follow logs for one
service
docker compose exec myapp bash # open shell in
running service
docker compose ps           # list service status
docker compose restart myapp # restart one service
docker compose pull         # pull latest images
```

Multi-Stage Builds

Reduce final image size by building in one stage and copying only the output to a lean final image.

```
# Stage 1: build
FROM python:3.11 AS builder
WORKDIR /build
COPY requirements.txt .
RUN pip install --prefix=/install --no-cache-dir -r
requirements.txt

# Stage 2: runtime — lean final image
FROM python:3.11-slim AS runtime
WORKDIR /app

# Copy only installed packages from builder
COPY --from=builder /install /usr/local
```

```
# Copy application code
COPY src/ ./src/

RUN useradd -m appuser
USER appuser

CMD ["python", "src/main.py"]

# Result: final image doesn't include build tools, pip
cache, or intermediate files
# Typical reduction: 800 MB → 150 MB
```

Docker for DE Pipelines

Packaging a PySpark job

```
# Official Apache Spark image (Python variant)
FROM apache/spark:3.5.3-python3

USER root
WORKDIR /opt/spark/jobs

# Install Python dependencies
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# Copy job files
COPY jobs/ .

USER spark

ENTRYPOINT ["/opt/spark/bin/spark-submit"]
CMD ["--master", "local[*]", "main.py"]
```

```
# Run the job
docker run --rm \
  -e AWS_ACCESS_KEY_ID=$AWS_ACCESS_KEY_ID \
  -e AWS_SECRET_ACCESS_KEY=$AWS_SECRET_ACCESS_KEY \
  -v $(pwd)/output:/output \
```

```
spark-job:1.0 \
--master local[*] \
jobs/orders_etl.py \
--date 2024-03-15
```

Packaging a SQL transformation project

The same pattern applies to any CLI-driven transformation tool; this example uses dbt with its official adapter image.

```
FROM ghcr.io/dbt-labs/dbt-snowflake:1.7.0

WORKDIR /usr/app/dbt
COPY . .

# profiles.yml will be mounted or env vars used at runtime
ENTRYPOINT ["dbt"]
CMD ["run"]
```

```
docker run --rm \
-e SNOWFLAKE_USER=$SNOWFLAKE_USER \
-e SNOWFLAKE_PASSWORD=$SNOWFLAKE_PASSWORD \
-v ~/.dbt:/root/.dbt \
my-dbt-project:latest \
run --target prod --select marts.*
```

Running Airflow in Docker

For real use, start from the official Compose file, which tracks each release (Airflow 3 adds `api-server`, `dag-processor`, and `triggerer` services): `curl -Lf0`

`'https://airflow.apache.org/docs/apache-airflow/stable/docker-compose.yaml'`

The simplified Airflow 2.x stack below shows how the pieces fit together.

```
# docker-compose.airflow.yml – simplified Airflow 2.x stack
x-airflow-common: &airflow-common
  image: apache/airflow:2.9.0
  environment:
    AIRFLOW__CORE__EXECUTOR: LocalExecutor
    AIRFLOW__DATABASE__SQL_ALCHEMY_CONN:
```

```

postgresql+psycpg2://airflow:airflow@postgres/airflow
  AIRFLOW__CORE__FERNET_KEY: ""
  AIRFLOW__CORE__DAGS_ARE_PAUSED_AT_CREATION: "true"
  AIRFLOW__CORE__LOAD_EXAMPLES: "false"
  _PIP_ADDITIONAL_REQUIREMENTS: "apache-airflow-
providers-postgres apache-airflow-providers-amazon"
volumes:
  - ./dags:/opt/airflow/dags
  - ./logs:/opt/airflow/logs
  - ./plugins:/opt/airflow/plugins
depends_on:
  postgres:
    condition: service_healthy

services:
  postgres:
    image: postgres:15
    environment:
      POSTGRES_USER: airflow
      POSTGRES_PASSWORD: airflow
      POSTGRES_DB: airflow
    volumes:
      - postgres_data:/var/lib/postgresql/data
    healthcheck:
      test: ["CMD", "pg_isready", "-U", "airflow"]
      interval: 5s
      retries: 5

  airflow-webserver:
    <<: *airflow-common
    command: webserver
    ports:
      - "8080:8080"

  airflow-scheduler:
    <<: *airflow-common
    command: scheduler

  airflow-init:
    <<: *airflow-common
    command: >
      bash -c "airflow db migrate &&

```

```
        airflow users create --username admin --
password admin
        --firstname Admin --lastname User --role
Admin
        --email admin@example.com"

volumes:
  postgres_data:
```

```
docker compose -f docker-compose.airflow.yml up -d
# Airflow UI: http://localhost:8080 (admin/admin)
```



Best Practices

```
# Recommended: Pin base image versions
FROM python:3.11.7-slim-bookworm # good
FROM python:latest # bad

# Recommended: Use slim or alpine variants
FROM python:3.11-slim # ~50 MB
FROM python:3.11 # ~350 MB
FROM python:3.11-alpine # ~20 MB (but may have glibc
compatibility issues)

# Recommended: One process per container
# Don't run both a web server and a background worker in
one container
# Use separate services in docker-compose instead

# Recommended: Non-root user
RUN useradd -m -u 1000 appuser
USER appuser

# Recommended: .dockerignore — exclude files from build
context
```



```
# .dockerignore
.git
```



```
.env
__pycache__
*.pyc
*.pyo
.pytest_cache
.venv
venv
*.egg-info
dist
build
docs
tests
*.log
```

```
# Recommended: Minimize layers – combine related RUN commands
# Bad
RUN apt-get update
RUN apt-get install -y curl wget
RUN apt-get clean

# Good
RUN apt-get update \
    && apt-get install -y --no-install-recommends curl wget \
    && apt-get clean \
    && rm -rf /var/lib/apt/lists/*

# Recommended: COPY only what's needed – not COPY . .
blindly
COPY requirements.txt .
COPY src/ ./src/
COPY config/ ./config/

# Recommended: Use healthchecks
HEALTHCHECK --interval=30s --timeout=10s --retries=3 \
    CMD curl -f http://localhost:8080/health || exit 1
```

Common Pitfalls

Pitfall	Symptom	Fix
<code>FROM python:latest</code> or unpinned <code>pip install</code>	An image rebuilt next month behaves differently or breaks	Pin the base image tag (ideally the digest) and dependency versions
<code>COPY . .</code> before installing dependencies	Every code change reinstalls all packages — slow builds	Copy the requirements/lock file, install, <i>then</i> copy the code
No <code>.dockerignore</code>	Huge build context; <code>.git</code> , <code>.env</code> , and data files end up inside the image	A <code>.dockerignore</code> excluding VCS, secrets, venvs, data, and caches
Secrets in <code>ENV</code> , <code>ARG</code> , or copied files	Anyone who can pull the image can read the secrets (<code>docker history</code> shows them)	Inject at runtime (env vars from a secrets manager, mounted files); <code>RUN --mount=type=secret</code> for build-time secrets
Running as root	A container escape or bug has root privileges	Create and switch to a non-root <code>USER</code>
Writing important data to the container filesystem	Data disappears when the container is removed	Volumes for local state; object storage/databases for pipeline outputs
<code>localhost</code> inside a container to reach another container	"Connection refused"	Use the Compose service name (<code>postgres:5432</code>); <code>host.docker.internal</code> to reach the host
<code>depends_on</code> without a healthcheck	App starts before the database is ready and crashes	<code>depends_on: {db: {condition: service_healthy}}</code> plus a healthcheck
Building on Apple Silicon, running on x86 servers	<code>exec format error</code> in production	<code>docker buildx build --platform linux/amd64</code> (or multi-arch builds)
Images several GB in size	Slow pulls, slow pod startup, bigger attack surface	Slim base images, multi-stage builds, clean package caches in the same <code>RUN</code>
Docker Desktop's VM running out of memory with Spark/Airflow	Containers get OOM-killed with no clear error	Raise the Docker memory limit; set container memory limits explicitly

Cheat Sheet

Task	Command
Build and tag	<code>docker build -t app:1.0 .</code>
Build for another CPU architecture	<code>docker buildx build --platform linux/amd64 -t app:1.0 .</code>
Run once and clean up	<code>docker run --rm app:1.0 --date 2024-03-15</code>
Interactive shell in a new container	<code>docker run --rm -it --entrypoint bash app:1.0</code>
Shell in a running container	<code>docker exec -it <name> bash</code>
Env vars / env file	<code>-e KEY=val · --env-file .env</code>
Mount the current directory	<code>-v "\$(pwd)":/app</code>
Publish a port	<code>-p 8080:8080 (host:container)</code>
Logs	<code>docker logs -f --tail 100 <name></code>
Resource usage	<code>docker stats</code>
Inspect image layers	<code>docker history app:1.0</code>
Disk usage / cleanup	<code>docker system df · docker system prune</code>
Compose: start / rebuild / stop	<code>docker compose up -d · up --build · down (-v also deletes volumes)</code>
Compose: logs / shell	<code>docker compose logs -f svc · docker compose exec svc bash</code>
Push to ECR	<code>aws ecr get-login-password docker login --username AWS --password-stdin <acct>.dkr.ecr.<region>.amazonaws.com → docker push</code>

Dockerfile template for a Python job



```
FROM python:3.12-slim
ENV PYTHONUNBUFFERED=1 PYTHONDONTWRITEBYTECODE=1
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY src/ ./src/
RUN useradd -m -u 1000 app
USER app
ENTRYPOINT ["python", "-m", "src.pipeline"]
CMD ["--help"]
```

Image vs container: image = the class · container = an instance · volume = its persistent disk · network = how containers find each other by name

Interview Questions

Q: What is the difference between an image and a container? A: An image is an immutable, layered filesystem plus metadata (the default command, environment, and exposed ports) — the packaged application. A container is a running instance of an image with its own writable layer, process namespace, and network. You can run many containers from one image; when a container is removed, its writable layer (and any data written there) goes with it.

Q: How does Docker layer caching work and how do you take advantage of it? A: Each Dockerfile instruction produces a layer, cached by the instruction and its inputs (for `COPY`, the file contents). On rebuild, Docker reuses cached layers until the first changed step, then rebuilds everything after it. So order instructions from least to most frequently changing: base image, system packages, dependency manifest + install, then application code. That way a code change only rebuilds the last layer.

Q: How is a container different from a virtual machine? A: A VM virtualizes hardware and runs a full guest OS on a hypervisor — strong isolation but heavy (GBs, minutes to boot). Containers share the host's kernel and isolate processes with namespaces and cgroups — lightweight (MBs, starts in seconds) with weaker isolation. That's why containers are the standard unit for packaging pipeline jobs and services.

Q: What are multi-stage builds and why use them? A: A Dockerfile with several `FROM` stages, where later stages copy only the artifacts they need from earlier ones. You compile or install dependencies in a full "builder" image with compilers and headers, then copy the results into a slim runtime image. The final image is smaller, faster to pull, and has fewer vulnerabilities because build tools aren't shipped.

Q: How do you handle secrets with Docker? A: Never bake them into the image — anything in `ENV`, `ARG`, or a copied file stays in the layers. Inject them at runtime: environment variables populated by the orchestrator from a secrets manager, or files mounted from Kubernetes secrets or Docker secrets. For secrets needed during the build (a private package index, for example), use BuildKit's `RUN --mount=type=secret`, which isn't persisted in any layer.

Q: How would you use Docker in a data pipeline? A: Package each job (a Spark job, SQL transformation project, or Python extractor) as a versioned image built in CI and pushed to a registry. The orchestrator runs it with a specific tag — Airflow's `KubernetesPodOperator` or `DockerOperator`, ECS/Batch, or Kubernetes Jobs — passing parameters like the run date as arguments and credentials from a secrets manager. Every run is reproducible, dependencies don't conflict between jobs, and rollbacks are just "run the previous tag".

Further Reading

- [Docker documentation](https://docs.docker.com/) (<https://docs.docker.com/>)
- [Dockerfile best practices](https://docs.docker.com/build/building/best-practices/) (<https://docs.docker.com/build/building/best-practices/>)
- [Docker Compose file reference](https://docs.docker.com/reference/compose-file/) (<https://docs.docker.com/reference/compose-file/>)
- [Running Airflow in Docker](https://airflow.apache.org/docs/apache-airflow/stable/howto/docker-compose/index.html) (<https://airflow.apache.org/docs/apache-airflow/stable/howto/docker-compose/index.html>)
- [Hadolint](https://github.com/hadolint/hadolint) (<https://github.com/hadolint/hadolint>) — Dockerfile linter
- [Dive](https://github.com/wagoodman/dive) (<https://github.com/wagoodman/dive>) — explore image layers and find wasted space
- [Trivy](https://trivy.dev/) (<https://trivy.dev/>) — scan images for vulnerabilities in CI

Previous: [Cloud Storage](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/cloud-storage/) (<https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/cloud-storage/>) • **Next:** [Kubernetes](https://sarangambekar1997.github.io/data-engineering-handbook/06-infrastructure/kubernetes-for-de/) (<https://sarangambekar1997.github.io/data-engineering-handbook/06-infrastructure/kubernetes-for-de/>) • **Back to:** [Index](https://sarangambekar1997.github.io/data-engineering-handbook/) (<https://sarangambekar1997.github.io/data-engineering-handbook/>)

 September 29, 2026