

Docker, CI/CD & Deployment interview questions

26 questions · 8 easy · 17 medium · 1 hard. Open one to see the answer, a code example and the likely follow-up, then mark it as known or to review.

1

What is the difference between a container and a virtual machine?

Easy



#basics #containers

A **virtual machine** emulates a whole computer: a hypervisor runs a full guest OS with its own kernel on top of the host. It is heavy (GBs), boots in tens of seconds, and gives strong isolation.

A **container** is just an isolated process that **shares the host kernel**. Linux **namespaces** give it its own view of processes, network and files; **cgroups** limit its CPU and memory. Images are MBs and containers start in milliseconds.

- Containers: dense, fast, portable, ideal for packaging apps
- VMs: stronger security boundary, can run a different OS kernel (Windows on Linux)

In practice containers usually run **inside** cloud VMs.

Likely follow-up: How can Docker run Linux containers on a Mac?

2

What is the difference between a Docker image and a container?

Easy



#basics #images

An **image** is a read-only template: a stack of filesystem layers plus metadata (default command, env, exposed ports). You build it from a Dockerfile and push it to a registry.

A **container** is a **running instance of an image**. Docker adds a thin writable layer on top, so many containers can share one image. Anything written inside the container is lost when it is removed, unless it goes to a volume.

Think of it like a class (image) and objects (containers).

Example

```
docker build -t myapp:1.0 .      # Dockerfile -> image
docker images                    # list images
docker run -d --name web -p 8080:3000 myapp:1.0  # image -> container
docker ps                       # running containers
docker exec -it web sh          # shell inside it
docker rm -f web                # container gone, image stays
```

3

Walk through the common Dockerfile instructions.

Easy



#dockerfile

- **FROM**: the base image, always first
- **WORKDIR**: sets (and creates) the working directory
- **COPY** / **ADD**: put files into the image
- **RUN**: executes a command **at build time** and saves the result as a new layer
- **ENV** / **ARG**: runtime env vars vs build-time variables
- **EXPOSE**: documents the port (it does not publish it)
- **USER**: which user later instructions and the container run as
- **CMD** / **ENTRYPOINT**: what runs **when the container starts**

Only **RUN**, **COPY** and **ADD** create filesystem layers; the rest add metadata.

Example

```
FROM node:20-alpine
WORKDIR /app
COPY package*.json ./
RUN npm ci --omit=dev
COPY . .
ENV NODE_ENV=production
EXPOSE 3000
USER node
CMD ["node", "server.js"]
```

Likely follow-up: Why copy **package.json** before the rest of the source?

4

CMD vs **ENTRYPOINT** ?

Medium 🔥🔥 #dockerfile

Both define what runs when the container starts.

- **ENTRYPOINT** is the **fixed executable**. Arguments to `docker run` are appended to it.
- **CMD** gives **default arguments** (or a default command). Anything after the image name in `docker run` replaces it.

Used together, **ENTRYPOINT** is the program and **CMD** is its **overridable default args**.

Prefer the **exec form** (`["node", "app.js"]`) over the shell form (`node app.js`): with the shell form your process runs under `/bin/sh`, is not PID 1, and does not receive **SIGTERM**, so graceful shutdown breaks.

Example

```
FROM alpine:3.20
ENTRYPOINT ["ping", "-c", "3"]
CMD ["localhost"]

# docker run img          -> ping -c 3 localhost
# docker run img google.com -> ping -c 3 google.com
# docker run --entrypoint sh img  -> override the entrypoint
```

5

COPY vs **ADD** ?

Easy 🔥🔥 #dockerfile

COPY simply copies files or folders from the build context into the image.

ADD does the same plus two extras: it **auto-extracts** local tar archives and can **download URLs**.

Best practice is to **use COPY unless you need tar extraction**. The magic of **ADD** is surprising, and downloading with **ADD** cannot be cached well or verified; use `RUN curl ... && sha256sum -c` instead.

6

How do image layers and the build cache work? How should you order instructions?

Medium



#dockerfile #performance

Each `RUN`, `COPY` and `ADD` produces a **layer**. On rebuild Docker reuses a cached layer if the instruction and its inputs (file checksums for `COPY`) are unchanged. Once one layer changes, **every layer after it is rebuilt**.

So order from **least to most frequently changing**:

- base image and system packages first
- copy only the dependency manifest, install dependencies
- copy the source code last

Then editing source code skips the slow `npm install`. Also combine related commands in one `RUN` and clean up in the same step (`apt-get install ... && rm -rf /var/lib/apt/lists/*`), because files deleted in a later layer still take space in the earlier one.

Example

```
# Bad: any code change re-runs npm ci
COPY . .
RUN npm ci

# Good: npm ci is cached until package files change
COPY package.json package-lock.json ./
RUN npm ci
COPY . .
```

7

What is a multi-stage build and why use it?

Medium



#dockerfile #image size

A multi-stage Dockerfile has several `FROM` stages. You build in a fat stage (compilers, dev dependencies, source) and then `COPY --from=build` only the output into a small runtime stage.

Benefits:

- **much smaller final image** (often 1 GB down to under 150 MB)

- smaller attack surface: no compilers, no dev dependencies, no source
- one Dockerfile for build and run, no separate build scripts

Stages can also be targeted (`docker build --target test`) to run tests in CI.

Example

```
FROM node:20-alpine AS build
WORKDIR /app
COPY package*.json ./
RUN npm ci
COPY . .
RUN npm run build && npm prune --omit=dev

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

8

How do you reduce the size of a Docker image?

Medium 🔥🔥 #image size #performance

- **Smaller base:** `alpine`, `-slim` or **distroless** images instead of full `node` or `ubuntu`
- **Multi-stage builds:** ship only the build output and production dependencies
- `.dockerignore`: keep `node_modules`, `.git`, logs and test data out of the context
- **Clean up in the same RUN:** package caches deleted in a later layer still count
- install only production dependencies (`npm ci --omit=dev`)
- for Go or Rust, a static binary on `scratch`

Inspect with `docker history <image>` or the `dive` tool to see **which layer is heavy**.
Smaller images pull faster, start faster and have fewer CVEs.

🔗 Likely follow-up: What trade-offs come with Alpine (musl) images?

9

Volumes vs bind mounts? How do you persist data?

Easy



#storage

A container's writable layer disappears with the container, so data that must survive goes outside it.

- **Named volume** (`-v pgdata:/var/lib/postgresql/data`): managed by Docker, stored in Docker's area, portable, best for **production data like databases**
- **Bind mount** (`-v $(pwd):/app`): maps a specific host path; great in development for live code reload, but depends on host layout and permissions
- **tmpfs**: in memory only, for secrets or scratch files

Volumes outlive containers; remove them explicitly with `docker volume rm`.

Example

```
docker volume create pgdata
docker run -d -v pgdata:/var/lib/postgresql/data postgres:16 # volume
docker run -it -v "$(pwd)":/app -w /app node:20 npm run dev # bind
mount
docker volume ls
```

10

How does Docker networking work? What does `-p 8080:3000` do?

Medium



#networking

- **bridge** (default): containers get private IPs on a virtual network on the host. On a **user-defined bridge** they can reach each other **by container name** via Docker's built-in DNS.
- **host**: the container shares the host's network stack (no isolation, no port mapping)
- **none**: no networking
- **overlay**: spans multiple hosts (Swarm)

`-p 8080:3000` publishes a port: traffic to **host** port 8080 is forwarded to **container** port 3000. `EXPOSE` alone does not do this.

A common bug: the app listens on `127.0.0.1` inside the container, so mapped traffic cannot reach it. Bind to `0.0.0.0`.

? Likely follow-up: Inside a container, why does `localhost:5432` not reach the database container?

11

What is Docker Compose and when do you use it?

Easy



#compose

Compose defines a **multi-container app** in one YAML file: services, images or build contexts, ports, env, volumes and networks. `docker compose up` starts everything on a shared network where services reach each other by **service name** (`db:5432`).

It is ideal for local development and CI integration tests. For production across many machines you would use an orchestrator like Kubernetes or ECS.

Note `depends_on` only orders startup; add `condition: service_healthy` with a health check to wait until the dependency is actually ready.

Example

```
services:
  api:
    build: .
    ports: ["3000:3000"]
    environment:
      DATABASE_URL: postgres://app:secret@db:5432/app
    depends_on:
      db: { condition: service_healthy }
  db:
    image: postgres:16
    environment: { POSTGRES_USER: app, POSTGRES_PASSWORD: secret }
    volumes: [pgdata:/var/lib/postgresql/data]
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U app"]
      interval: 5s
volumes:
  pgdata:
```

12

How do you handle environment variables and secrets in containers?

Medium



#security #config

Config goes in **environment variables** at runtime (`-e`, `--env-file`, Compose `environment`, Kubernetes ConfigMaps), so the **same image runs in every environment**.

Secrets need more care:

- never `COPY` a `.env` into the image or put secrets in `ENV` / `ARG`: they stay in the layers and `docker history`
- inject at runtime from a secret store (AWS Secrets Manager, Vault, Kubernetes Secrets, GitHub Actions secrets)
- for build-time secrets (a private npm token) use BuildKit `RUN --mount=type=secret`
- prefer mounted secret files over env vars when possible, since env vars leak into logs and crash dumps

Example

```
# syntax=docker/dockerfile:1
FROM node:20-alpine
WORKDIR /app
COPY package*.json ./
# token is available only during this RUN, never stored in a layer
RUN --mount=type=secret,id=npmmc,target=/root/.npmmc npm ci
# build: docker build --secret id=npmmc,src=$HOME/.npmmc .
```

13

What is `.dockerignore` for?

Easy



#dockerfile #security

When you run `docker build .`, the whole directory (the **build context**) is sent to the Docker daemon. `.dockerignore` excludes files from it, like `.gitignore`.

Why it matters:

- **faster builds**: no sending hundreds of MB of `node_modules` or `.git`
- **better caching**: `COPY . .` is not invalidated by irrelevant files changing
- **security**: keeps `.env` files and keys out of the image
- **correctness**: the host's `node_modules` (built for macOS) does not overwrite the Linux one

Example

```
# .dockerignore
node_modules
.git
.env*
dist
coverage
*.log
Dockerfile
```

14

What are health checks, and what is the difference between liveness and readiness?

Medium



#reliability #kubernetes

A **health check** is a command or HTTP call that tells the platform whether the app is working, not just whether the process exists.

- **Liveness:** "is it stuck?" If it fails, the container is **restarted**.
- **Readiness:** "can it take traffic now?" If it fails, it is **removed from the load balancer** but not killed (warming up, DB down).
- **Startup** (Kubernetes): gives slow apps time before liveness kicks in.

Keep liveness cheap and local; do not make it fail when a downstream dependency is down, or you get restart storms. In Docker use **HEALTHCHECK**; in Kubernetes

livenessProbe / **readinessProbe**.

Example

```
HEALTHCHECK --interval=30s --timeout=3s --retries=3 \
  CMD wget -qO- http://localhost:3000/health || exit 1
```

15

Why should containers run as a non-root user?

Medium



#security

By default the process inside a container runs as **root (UID 0)**. Containers share the host kernel, so if an attacker exploits your app and then a kernel or runtime bug, **root in the**

container can become root on the host. Root also lets them modify app files and install tools.

Hardening:

- add `USER node` (or create a user) in the Dockerfile
- read-only root filesystem (`--read-only`), drop Linux capabilities (`--cap-drop ALL`)
- `runAsNonRoot: true` in the Kubernetes security context
- scan images (Trivy, Docker Scout) and pin base image versions

Example

```
FROM node:20-alpine
RUN addgroup -S app && adduser -S app -G app
WORKDIR /app
COPY --chown=app:app . .
USER app
CMD ["node", "server.js"]
```

16

What is Kubernetes? Explain pod, deployment and service.

Medium



#kubernetes #orchestration

Kubernetes is a **container orchestrator**: you declare the desired state in YAML and it keeps reality matching it across a cluster (scheduling, restarts, scaling, rolling updates, service discovery).

- **Pod**: the smallest unit; one or more containers sharing a network IP and volumes. Pods are disposable.
- **Deployment**: manages a ReplicaSet to keep N identical pods running, and does rolling updates and rollbacks.
- **Service**: a stable virtual IP and DNS name that load-balances to the matching pods (pods come and go, their IPs change). Types: `ClusterIP`, `NodePort`, `LoadBalancer`.
- **Ingress**: HTTP routing from outside (host/path rules, TLS).

Also: ConfigMap/Secret for config, HPA for autoscaling.

Example

```
apiVersion: apps/v1
kind: Deployment
metadata: { name: api }
spec:
  replicas: 3
  selector: { matchLabels: { app: api } }
  template:
    metadata: { labels: { app: api } }
    spec:
      containers:
        - name: api
          image: registry.example.com/api:1.4.2
          ports: [{ containerPort: 3000 }]
---
apiVersion: v1
kind: Service
metadata: { name: api }
spec:
  selector: { app: api }
  ports: [{ port: 80, targetPort: 3000 }]
```

🔗 Likely follow-up: What happens step by step when a pod crashes?

17

CI vs continuous delivery vs continuous deployment?

Easy 🔥🔥 #ci/cd

- **Continuous Integration:** every push or PR is automatically built and tested (lint, unit tests, build). Developers merge small changes often, so problems surface early.
- **Continuous Delivery:** every change that passes the pipeline is **always in a releasable state** and deployed to staging; production release is a **manual** button.
- **Continuous Deployment:** no manual step; every green change goes straight to production.

Continuous deployment needs strong tests, feature flags, monitoring and quick rollback.

Describe a typical CI/CD pipeline, for example with GitHub Actions.

Medium



#ci/cd #github actions

A typical pipeline on push to `main`:

1. **Checkout** and set up the runtime, with dependency caching
2. **Lint, type-check, unit tests** (fast feedback first)
3. **Build** the Docker image, tagged with the commit SHA
4. **Scan** it and **push** to a registry
5. **Deploy** to staging, run smoke/E2E tests
6. **Promote** to production (manual approval or automatic), then monitor

Key points: secrets come from the CI secret store, `tag images with the git SHA` (not just `latest`) so you know exactly what runs and can roll back, and PRs run only the test stages.

Example

```
name: ci
on: { push: { branches: [main] }, pull_request: {} }
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-node@v4
        with: { node-version: 20, cache: npm }
      - run: npm ci
      - run: npm run lint && npm test
      - if: github.ref == 'refs/heads/main'
        run: |
          echo "${{ secrets.REGISTRY_TOKEN }}" | docker login ghcr.io -u
            ${{ github.actor }} --password-stdin
          docker build -t ghcr.io/acme/api:${{ github.sha }} .
          docker push ghcr.io/acme/api:${{ github.sha }}
```

19

Blue-green vs canary vs rolling deployments?

Medium



#deployment

- **Rolling:** replace instances a few at a time. No extra infrastructure, but old and new versions run side by side for a while, and rollback is another rollout. Kubernetes default.
- **Blue-green:** run a full new environment (green) next to the live one (blue), test it, then **switch all traffic at once** at the load balancer. Instant rollback (switch back), but double the resources during the release.
- **Canary:** send a small share of traffic (1%, 5%, 25%...) to the new version, watch error rates and latency, then increase. Limits the blast radius; needs good metrics and traffic splitting.

All of them require backward-compatible DB changes, because two versions may be live at once.

🔗 Likely follow-up: How do you handle a database migration during a blue-green deploy?

20

A bad release reached production. How do you roll back?

Medium



#deployment #reliability

First **stop the bleeding**, then debug.

- **Redeploy the previous immutable image** (tagged by git SHA): `kubectl rollout undo deployment/api`, or switch the blue-green router back
- If the bug is behind a **feature flag**, just turn it off: fastest of all
- Revert the commit so the pipeline stays the source of truth

What makes rollback safe:

- **immutable, versioned artifacts** (never rebuild the old version)
- **backward-compatible migrations** (expand then contract), so old code still works with the new schema
- automated rollback when health checks or error-rate alerts fire during a canary

Afterwards write a blameless postmortem.

21

What is a reverse proxy? How is Nginx used in deployments?

Medium



#nginx #networking

A **forward proxy** acts for clients; a **reverse proxy** sits **in front of your servers** and receives requests on their behalf.

Nginx as a reverse proxy typically does:

- **TLS termination** (HTTPS) so apps speak plain HTTP inside
- **load balancing** across app instances (round robin, least connections, IP hash)
- serving static files and caching, gzip compression
- routing by host or path (`/api` to the API, `/` to the frontend)
- rate limiting, request size limits, hiding internal topology

In cloud setups the same role is played by an AWS ALB, Traefik or a Kubernetes Ingress controller.

Example

```
# nginx.conf (excerpt)
upstream api { least_conn; server api1:3000; server api2:3000; }
server {
    listen 443 ssl;
    server_name example.com;
    location /api/ { proxy_pass http://api; proxy_set_header Host $host; }
    location /      { root /usr/share/nginx/html; try_files $uri
/index.html; }
}
```

22

What is the 12-factor app methodology?

Medium



#architecture #best practices

A set of principles for building cloud-friendly services. The ones interviewers care about most:

- **Config in the environment**, not in code; the same build runs everywhere
- **Stateless processes**: no sessions or files on local disk; state lives in backing services (DB, Redis, S3), so you can **scale horizontally and kill any instance**
- **Backing services as attached resources** swapped by a URL

- **Build, release, run** are separate stages; releases are immutable
- **Port binding**: the app serves HTTP itself
- **Disposability**: fast startup, graceful shutdown on **SIGTERM**
- **Dev/prod parity, logs as event streams** to stdout

23

How do you do logging and monitoring for containerised apps?

Medium



#observability

Logs: write to **stdout/stderr** in **structured JSON** (level, timestamp, request ID). The runtime collects them and a shipper (Fluent Bit, Vector) sends them to a central store (ELK, Loki, CloudWatch, Datadog). Never write log files inside the container.

Metrics: expose counters and histograms (**/metrics** for Prometheus), graph in Grafana. Watch the **golden signals**: latency, traffic, errors, saturation.

Traces: OpenTelemetry to follow one request across services.

Alerting: alert on symptoms users feel (error rate, p99 latency, SLO burn), not on every CPU spike, and link alerts to runbooks.

24

VMs vs containers vs serverless: how do you choose? Where do S3 and a CDN fit?

Medium



#cloud #serverless

- **VMs (EC2)**: full control of the OS; you patch and scale them. Good for legacy or special workloads.
- **Containers (ECS, Kubernetes, Cloud Run)**: portable, consistent, efficient; the default for most web services.
- **Serverless functions (Lambda)**: no servers, **pay per request**, scale to zero. Great for spiky or event-driven work; watch for cold starts, time limits and vendor lock-in.

Object storage (S3) holds files: uploads, backups, static site builds. It is cheap and durable, but not a file system or a database.

A **CDN** (CloudFront, Cloudflare) caches static assets and pages at edge locations near users, cutting latency and origin load. A common setup: static frontend in S3 behind a CDN, API in containers.

How do you achieve zero-downtime deployments?

Hard



#deployment #reliability

Users should never see errors while versions change:

- run **multiple instances** behind a load balancer and use rolling, blue-green or canary deploys
- **readiness probes**: new instances get traffic only when ready
- **graceful shutdown**: on `SIGTERM`, stop accepting new requests, **finish in-flight requests**, close DB connections, then exit; the balancer drains connections first
- **backward-compatible DB migrations** with expand and contract: add the new column, deploy code that writes both, backfill, then remove the old column in a later release
- keep APIs backward compatible, since old and new clients overlap
- stateless servers, so no sessions are lost

Example

```
// Graceful shutdown pattern (simulated server)
const inFlight = new Set()
let accepting = true
function handle(id, ms) {
  if (!accepting) return console.log('rejected', id)
  const p = new Promise((r) => setTimeout(r, ms)).then(() => {
    inFlight.delete(p); console.log('done', id) })
  inFlight.add(p)
}
async function onSigterm() {
  accepting = false // readiness fails, LB stops sending traffic
  console.log('draining', inFlight.size, 'requests')
  await Promise.all(inFlight) // finish in-flight work
  console.log('closed, exiting')
}
handle('a', 50); handle('b', 100)
onSigterm()
handle('c', 10)
```

? **Likely follow-up:** How do you rename a column used by the running code without downtime?

26

A container keeps restarting (CrashLoopBackOff). How do you debug it?

Medium



#debugging #kubernetes

Work from cheapest signal to deepest:

- `docker logs <c>` / `kubectl logs pod --previous`: the logs of the crashed run usually say it all
- `kubectl describe pod`: events, exit code, OOMKilled, failed probes, image pull errors
- **exit codes:** `137` = killed (often out of memory), `1` = app error, `127` = command not found (wrong `CMD` or missing binary)
- check env vars and secrets are present, and dependencies (DB) are reachable
- a liveness probe that is too strict or too early also causes restarts
- run the image interactively with a shell: `docker run -it --entrypoint sh img`