

Container Lifecycle Management & Deployments Interview Guide

Platform, SRE & DevOps interview guide

Questions	103 questions · spoken answers included
Online (always latest)	interviewhub.eknathalabs.com/container_lifecycle_interview_guide.html
Printed	2026-10-21

Free to use and share for personal study. Answers are written for interview practice, so check command flags and versions against the official documentation before running anything in production.

Spoken answer · ~84 s

A container is an isolated process, or group of processes, running on the host's kernel. Isolation comes from Linux namespaces, which give each container its own view of process IDs, network, mounts, hostnames, and users, and cgroups, which limit resources like CPU and memory. A virtual machine, by contrast, runs a full guest operating system on virtualised hardware managed by a hypervisor. The differences follow from that. Containers share the host kernel, so they start in milliseconds, use megabytes rather than gigabytes, and you can pack many more onto a host. VMs have stronger isolation because each has its own kernel, so a kernel exploit in one guest doesn't directly expose the others, and they can run different operating systems. Containers must match the host kernel type, so Linux containers need a Linux kernel. Containers package an application with its dependencies in an image, giving consistent behaviour from laptop to production, which supports microservices and fast CI/CD. VMs suit workloads needing strong isolation, different kernels, or legacy applications. They're often combined: containers running inside VMs in the cloud. Newer runtimes like gVisor and Kata Containers add stronger isolation to containers. Choose based on isolation needs, density, startup speed, and operational model.

Likely follow-ups

- What kernel features make containers possible?
- When would you prefer a VM over a container?

Container vs VM

Virtual machine	Container
Own guest kernel	Shares host kernel
Heavier, strong isolation	Light, fast start

A container is a lightweight, isolated process that shares the host OS kernel but has its own filesystem, network stack, and process namespace. A VM virtualizes hardware and runs a full guest OS kernel.

KEY DIFFERENCES

VM: Hypervisor → Guest OS kernel → Application (heavy, slow boot)

Container: Host kernel → Namespaces + cgroups → Application (lightweight, fast)

VM size: GBs (full OS). Container size: MBs (app + deps only)

VM boot: 30-60 seconds. Container start: milliseconds

Isolation: VM = hardware-level. Container = kernel-level (namespaces)

Containers use Linux namespaces (pid, net, mnt, uts, ipc, user) for isolation and cgroups for resource limits. The shared kernel means containers are less isolated than VMs — a kernel exploit can potentially escape a container. VMs provide stronger isolation but at the cost of resource overhead. In practice: run containers on VMs in cloud environments — you get both the performance of containers and the isolation of VMs.

Interview tip: Mention that containers are not magic — they are simply Linux processes with namespace and cgroup constraints applied. `docker run` is ultimately a `fork()` + `clone()` with specific flags. This depth shows you understand the kernel primitives underneath.

 **Start here (official docs)**

- [Docker documentation](#)

Q2 Explain the Docker architecture — daemon, client, registry, and image layers.

Docker fundamentals

 Spoken answer · ~83 s

Docker uses a client-server architecture. The Docker client, the `docker` command, sends requests via a REST API to the Docker daemon, `dockerd`, which does the work: building images, running containers, managing networks and volumes. The client and daemon can be on the same machine or remote. Under the hood, `dockerd` delegates container lifecycle to `containerd`, which in turn uses `runc`, the low-level OCI runtime, to create the actual container process. A registry, such as Docker Hub, Artifact Registry, or ECR, stores and distributes images; `docker pull` downloads layers and `docker push` uploads them. An image is a read-only template made of ordered layers; each layer is a filesystem diff produced by a Dockerfile instruction, identified by a content hash, so layers are cached and shared between images, saving space and bandwidth. When a container starts, Docker adds a thin writable layer on top of the image layers, using a union or overlay filesystem; changes made at runtime go into that layer and disappear when the container is removed unless stored in volumes. Images are referenced by repository, tag, and an immutable digest. Understanding the layers explains build caching, image size, and why instruction order in a Dockerfile matters.

Likely follow-ups

- What role does `containerd` play?
- Why are digests safer than tags?

COMPONENTS

Docker CLI (client) → sends API requests over Unix socket or TCP

`dockerd` (daemon) → manages images, containers, networks, volumes

`containerd` → container runtime (manages container lifecycle)

`runc` → OCI runtime (actually creates the container)

Registry → stores and distributes images (Docker Hub, GCR, ECR)

docker info # daemon configuration

docker system df # disk usage by images, containers, volumes

docker system prune -a # remove all unused resources

Docker is a layered architecture. The CLI sends requests to dockerd via REST API. dockerd delegates to containerd for container lifecycle. containerd uses runc (or another OCI runtime) to actually create the container. Image layers are content-addressed (SHA256 digest) and shared across containers using copy-on-write. This layering means two containers based on the same ubuntu image share those base layers in storage.

Kubernetes context: Kubernetes deprecated dockershim in 1.24 — it now uses containerd or CRI-O directly via the CRI (Container Runtime Interface). Docker images still work because they follow the OCI image spec. The change only affects the runtime, not images or Dockerfiles.

Official docs

- [Docker: Storage drivers \(overlayfs\)](#)
- [Docker documentation](#)

Q3 What is the difference between docker run, docker start, and docker exec?

Docker fundamentals

🗣 Spoken answer · ~85 s

These three commands do different things in the container lifecycle. `docker run` creates a new container from an image and starts it in one step: it pulls the image if missing, creates the writable layer, sets up networking, applies options like ports, volumes, environment variables, and limits, and runs the specified command. Each run creates a new container. `docker start` restarts an existing stopped container, using its original configuration; it doesn't create anything new, and the writable layer's data is still there. `docker exec` runs an additional command inside an already running container, for example `docker exec -it name sh` to open a shell for debugging; it launches a new process in the container's namespaces and doesn't restart anything, and the process ends when the command finishes without affecting the main one. Related: `docker create` builds a container without starting it; `docker stop` sends `SIGTERM` and then `SIGKILL` after a timeout; `docker attach` connects to the main process's streams. A common mistake is using `docker run` repeatedly and accumulating stopped containers; use `--rm` for throwaway containers. Another is expecting `exec` changes to persist after the container is recreated; they won't. In production, containers are created by orchestrators, and `exec` is for diagnostics only.

Likely follow-ups

- Why does a `SIGTERM` matter for `docker stop`?
- What is the difference between `attach` and `exec`?

```
docker run nginx:latest          # create NEW container from image + start it
```

```
docker start <container-id>      # start an EXISTING stopped container
```

```
docker exec -it <id> /bin/sh     # run command in RUNNING container
```

```
docker attach <id>               # attach to PID 1 stdin/stdout of running container
```

```
docker run -d --name web nginx   # detached (background) with name
```

```
docker run --rm nginx ls /       # run + auto-remove on exit
```

```
docker run -it ubuntu bash       # interactive + TTY (for debugging)
```

`docker run` = `docker create` + `docker start`. It pulls the image if not cached, creates a new container layer, and starts the process. `docker exec` enters a running container by joining its namespaces — it does not start a new container. `docker attach` connects your terminal to PID 1 — sending `Ctrl+C` will stop the container. Use `exec` for debugging, not `attach`, to avoid accidentally stopping production containers.

docker attach danger: Attaching to a production container and pressing `Ctrl+C` sends `SIGINT` to PID 1, which may stop the container. Always use `docker exec -it` for interactive debugging of running

containers.

Official docs

- [Docker documentation](#)

Q4 What do `docker ps`, `docker inspect`, and `docker stats` reveal — when do you use each?

Docker fundamentals

Spoken answer · ~85 s

These three commands give different views of containers. `docker ps` lists containers, running by default and all with `-a`, showing ID, image, command, status, ports, and names; I use it first to see what is running, what has exited, and whether a container is restarting repeatedly. `docker inspect` returns detailed low-level JSON configuration and state: environment variables, mounts, network settings including IP addresses, restart policy, health check status, exit code, OOMKilled flag, log path, and image digest. I use it to answer why something is configured the way it is, or why it exited; with a format flag, you can extract single fields. `docker stats` shows a live stream of resource usage per container: CPU percentage, memory usage against limit, network and block I/O, and process count. I use it to find noisy neighbours, memory growth approaching a limit, or CPU saturation. Together: `ps` to find the container, `inspect` to understand configuration and exit reason, `stats` to understand runtime resource behaviour. Alongside these, `docker logs` gives output, `docker top` lists processes, and `docker events` streams daemon events, helpful for seeing kills and restarts. In production orchestrators, equivalents are `kubectl get`, `describe`, and `top`, plus metrics systems for history, since `stats` only shows now.

Likely follow-ups

- Which field in `inspect` tells you if a container was OOM killed?
- What does `docker events` help with?

```
docker ps # running containers: ID, image, status, ports
```

```
docker ps -a # all containers including stopped
```

```
docker ps --format "table {{.Names}} {{.Status}} {{.Ports}}"
```

```
docker inspect <container> # full JSON config: mounts, env, network, PID
```

```
docker inspect -f "{{.State.Pid}}" <container> # extract specific field
```

```
docker inspect -f "{{.NetworkSettings.IPAddress}}" <container>
```

```
docker stats # live CPU, memory, net I/O, block I/O per container
```

```
docker stats --no-stream # snapshot (not live)
```

```
docker top <container>          # processes inside container (from host perspective)
```

`docker inspect` is the most powerful diagnostic tool — it shows the full container configuration as JSON: environment variables, mounted volumes, network settings, restart policy, health check config, and the container PID on the host. Use `-f` with Go template syntax to extract specific fields in scripts. `docker stats` shows real-time resource consumption — critical for detecting containers approaching their limits before an OOM kill.

Production use: `docker inspect -f "{{.State.Pid}}"` lets you get the container PID to enter its namespaces with `nsenter`, run `strace`, or examine `/proc` — invaluable when a container lacks debugging tools (distroless images).

Official docs

- [Docker documentation](#)

Q5 How does docker networking work — what are the default network drivers?

Docker fundamentals

Spoken answer · ~83 s

Docker networking is built from drivers that determine how containers connect. The default bridge driver creates a private virtual network on the host, `docker0`: containers get IP addresses from a subnet, can reach each other by IP, and reach outside through NAT; ports must be published to be reachable from outside. On user-defined bridge networks, containers additionally get automatic DNS resolution by name, better isolation, and the ability to attach and detach at runtime, so you should use those rather than the default bridge. The host driver removes network isolation, so the container shares the host's network stack: no port mapping, best performance, less isolation. The none driver gives only a loopback interface, for fully isolated containers. The overlay driver spans multiple Docker hosts, used by Swarm, via VXLAN encapsulation. Macvlan assigns containers their own MAC and IP on the physical network, so they appear as separate devices, useful for legacy integration. Under the hood, Linux network namespaces, veth pairs, bridges, iptables rules for NAT and port publishing, and an embedded DNS server at 127.0.0.11 do the work. In Kubernetes, CNI plugins replace this model with a flat pod network. For debugging, `docker network inspect` shows connected containers and subnets.

Likely follow-ups

- Why prefer a user-defined bridge over the default one?
- When would you use host networking?

```
docker network ls                # list networks
```

```
docker network inspect bridge    # inspect default bridge network
```

```
docker network create mynet      # create user-defined bridge
```

```
docker run --network mynet nginx # attach container to network
```

```
docker run --network host nginx  # share host network namespace
```

```
docker run --network none nginx  # no network (fully isolated)
```

Network drivers: bridge (default — virtual switch, NAT for external access, containers communicate by IP), host (container shares host network namespace — best performance, no port mapping needed, no isolation), overlay (multi-host networking for Docker Swarm), macvlan (assign MAC address directly, appears as physical device), none (no networking). User-defined bridge networks enable DNS resolution by container name — the default bridge network does not support this.

User-defined bridge vs default: Always create user-defined bridge networks. They provide automatic DNS resolution (containers reach each other by name), better network isolation, and dynamic connect/disconnect. The default bridge network requires `--link` (deprecated) for DNS and is considered legacy.

Official docs

- [Docker: Networking](#)
- [Docker documentation](#)

Spoken answer · ~89 s

When a container exits immediately, it usually means the main process finished or crashed, since a container lives only as long as its PID 1 process. My approach: first `docker ps -a` to see the exit code, and `docker logs` for output. Exit code 0 means the process completed normally, for instance a command that is not long-running or a daemon that forked to the background and left the foreground; 1 or another non-zero code indicates an application error; 125 to 127 point to docker or command problems such as a missing executable or permission issues; 137 means SIGKILL, often out of memory, which `docker inspect` confirms with `OOMKilled`; 139 is a segmentation fault; 143 is SIGTERM. Then `docker inspect` to check the entrypoint, command, environment variables, and mounts. Common causes: missing environment variables or config files, a wrong CMD or ENTRYPOINT, shell script without execute permission or with Windows line endings, a dependency like a database not ready, port conflicts, and the process daemonising. To investigate interactively, override the entrypoint: `docker run -it --entrypoint sh image`, then run the command manually. Compare with a working environment. Fix by running the service in the foreground, handling configuration properly, adding retries for dependencies, and logging to stdout. Add health checks and a sensible restart policy.

Likely follow-ups

- What does exit code 137 indicate?
- Why must the main process stay in the foreground?

```
docker run myimage          # exits immediately
```

```
docker ps -a                # show stopped container
```

```
docker logs <container-id>  # see output/errors before exit
```

```
docker inspect <id> --format "{{.State.ExitCode}}" # exit code
```

```
docker inspect <id> --format "{{.State.Error}}"    # error message
```

```
# Debug: override entrypoint to get a shell
```

```
docker run -it --entrypoint /bin/sh myimage
```

```
# Or run with sh -c to see what CMD does:
```

```
docker run -it myimage sh -c "echo starting; exec myapp; echo done"
```

A container exits when PID 1 exits. If your ENTRYPOINT or CMD exits immediately (wrong command, missing binary, startup error), the container stops. Common causes: wrong CMD (typo in binary name — container gets exec error), application crashes on startup (missing env var, DB not reachable), foreground process not

running (supervisord/init not used for multi-process), or script runs and exits cleanly. Check docker logs first, then override the entrypoint to drop to a shell for interactive investigation.

Shell vs exec form: CMD ["app"] (exec form) runs app as PID 1. CMD "app" (shell form) runs /bin/sh -c "app" as PID 1, and app is a child — it won't receive SIGTERM. Always use exec form in Dockerfiles for production containers so your app handles signals correctly.



Official docs

- [Docker documentation](#)

Q7 What are Docker image layers — how does the copy-on-write filesystem work?

Images & Dockerfile

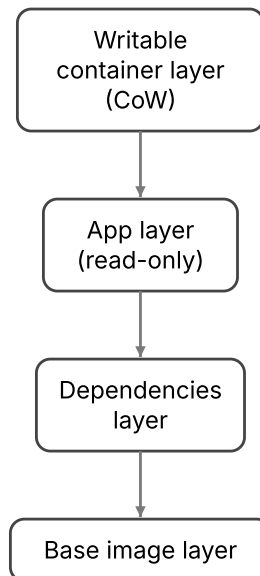
🔊 Spoken answer · ~82 s

A Docker image is a stack of read-only layers, each representing the filesystem changes made by a Dockerfile instruction such as COPY, RUN, or ADD. Layers are content-addressed by hash, so identical layers are stored once on disk and downloaded once, even if many images share them, and the build cache reuses a layer if the instruction and its inputs haven't changed. When a container starts, Docker's storage driver, usually overlay2, presents the layers as a single merged filesystem and adds a thin writable layer on top. Copy-on-write governs changes: reading a file reads it from the highest layer where it exists; modifying a file that lives in a lower, read-only layer first copies the entire file up into the writable layer, then changes the copy; deleting a file creates a whiteout marker that hides it rather than actually removing it from lower layers. Consequences: files deleted in a later layer still take space in the image, so combine commands and clean up in the same RUN instruction; writing large files in containers is slow at first and bloats the writable layer, so use volumes for heavy I/O; and the writable layer vanishes with the container.

Likely follow-ups

- Why does deleting a file in a later layer not shrink the image?
- How does layer ordering affect build speed?

🚧 Image layers



```
docker image history myimage    # show all layers and sizes
```

```
docker image inspect myimage    # full image metadata including layers
```

```
docker diff <container>        # changes made in container layer vs image
```

```
dive myimage                    # interactive layer explorer (third-party tool)
```

Docker images are built as a stack of read-only layers. Each Dockerfile instruction (FROM, RUN, COPY, ADD) creates a new layer. Layers are content-addressed by SHA256 and shared across images — if 10 images use the same ubuntu:22.04 base, that layer is stored once. When a container starts, a thin read-write layer is added on top. Copy-on-write: when a container modifies a file from a lower layer, that file is copied to the writable layer first — the original layer is unchanged. This means reading unmodified files from lower layers is fast; modifying large files adds I/O overhead.

Layer caching in CI: Docker build cache reuses layers if the instruction and all previous layers haven't changed. Order Dockerfile instructions from least to most frequently changing: base image → system deps → app deps (package.json/requirements.txt) → app code. This maximizes cache hits and speeds up builds significantly.

📖 Official docs

- [Docker: Storage drivers \(overlayfs\)](#)
- [Docker documentation](#)

Q8 How do you write an optimized production Dockerfile — what are the best practices?

Images & Dockerfile

 Spoken answer · ~88 s

An optimised production Dockerfile is small, fast to build, secure, and reproducible. Practices: choose a minimal, trusted base image, such as slim, Alpine where compatible, or distroless, and pin its version or digest rather than latest. Use multi-stage builds so compilers and build tools stay out of the final image. Order instructions for cache efficiency: copy dependency manifests and install dependencies first, then copy the source. Combine related RUN commands and clean package caches in the same layer. Use a .dockerignore to keep the build context small and avoid including secrets. Run as a non-root user with a dedicated UID, drop unnecessary capabilities at runtime, and consider a read-only root filesystem. Never bake secrets into images or build args; use BuildKit secret mounts. Use COPY rather than ADD, and set a clear ENTRYPOINT and CMD in exec form, so signals reach the process. Add a HEALTHCHECK where appropriate, set labels for metadata, and define EXPOSE and a sensible WORKDIR. Install only production dependencies, with pinned versions for reproducibility. Scan images for vulnerabilities and lint with hadolint. Build with BuildKit and cache mounts. Keep one main process per container, log to stdout, and tag images with the commit SHA. Smaller images mean faster pulls, a smaller attack surface, and quicker deployments.

Likely follow-ups

- Why use exec form for ENTRYPOINT?
- How do BuildKit secret mounts work?

BEST PRACTICES

```
# 1. Use specific base image tags (not :latest)
```

```
FROM node:20.11-alpine3.19      # pinned version + minimal base
```

```
# 2. Multi-stage build (separate build from runtime)
```

```
FROM node:20-alpine AS builder
```

```
WORKDIR /app
```

```
COPY package*.json ./           # copy deps manifest first (cache layer)
```

```
RUN npm ci --only=production
```

```
COPY . .
```

```
RUN npm run build
```

```
FROM node:20-alpine AS runtime  # fresh minimal runtime stage
```

```
RUN adduser -D -u 1001 appuser # non-root user
```

```
WORKDIR /app
```

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

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

```
USER appuser # drop privileges
```

```
EXPOSE 3000
```

```
ENTRYPOINT ["node", "dist/server.js"] # exec form (not shell form)
```

Key practices: pin base image versions (avoid supply chain surprises), use Alpine or distroless for minimal attack surface and image size, multi-stage builds to exclude build tools from production image, copy dependency manifests before source code to maximize layer cache, run as non-root user, use exec form ENTRYPOINT, add .dockerignore to exclude .git, node_modules, test files.

Distroless images: Google's distroless images (gcr.io/distroless/nodejs) contain only the runtime and your app — no shell, no package manager, no utilities. Attack surface is minimal. Downside: no shell for debugging. Use a debug variant (distroless:debug) that includes busybox for troubleshooting.



Official docs

- [Docker: Dockerfile best practices](#)

Spoken answer · ~88 s

A multi-stage build uses multiple FROM statements in one Dockerfile, where each stage is a separate image context, and you copy only the needed artefacts from earlier stages into the final one. Typically, the first stage uses a full SDK image with compilers and dev dependencies to build the application, and the last stage starts from a minimal runtime image, such as slim or distroless, and copies just the compiled binary or built assets with COPY --from=build. It solves several problems. Image size: build tools, source code, and intermediate files are left behind, sometimes shrinking images from over a gigabyte to tens of megabytes. Security: fewer packages mean a smaller attack surface, and no compilers or shells for an attacker to use, and secrets used in build stages don't end up in the final image if handled correctly. Simplicity: one Dockerfile replaces the old builder-pattern with separate Dockerfiles and scripts. Cache efficiency: stages build in parallel under BuildKit, and only the stages needed for the target are built. You can also name stages and use --target to build a test or debug stage from the same file, for example running unit tests in a test stage in CI. It also helps reproducibility, since the build environment is defined in code.

Likely follow-ups

- How does --target help in CI?
- What gets left out of the final image?

```
# Without multi-stage: production image includes compiler, build tools, test deps
```

```
FROM golang:1.22
```

```
RUN apt-get install -y gcc make # build tools in prod image
```

```
COPY . .
```

```
RUN go build -o app .
```

```
# Result: 800MB image with Go compiler, gcc, etc.
```

```
# With multi-stage: only the binary goes to production
```

```
FROM golang:1.22-alpine AS builder
```

```
WORKDIR /app
```

```
COPY go.mod go.sum ./
```

```
RUN go mod download
```

```
COPY . .
```

```
RUN CGO_ENABLED=0 go build -o app .
```

```
FROM scratch AS runtime          # empty base (ultimate minimal)
```

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

```
ENTRYPOINT ["/app"]
```

```
# Result: ~10MB image with only the static binary
```

Multi-stage builds solve: large image sizes (build tools not in prod), security (compilers are attack surface), and separation of concerns (build environment vs runtime). Each FROM starts a new stage. COPY --from=<stage> copies artifacts between stages. Only the final stage is the production image. You can target intermediate stages: docker build --target builder for debugging.

FROM scratch: The scratch base is an empty filesystem — no shell, no libc, no utilities. Only works for statically compiled binaries (Go with CGO_ENABLED=0, Rust). Produces the smallest possible image and the smallest attack surface. Extremely popular for microservices written in Go.

Official docs

- [Docker: Multi-stage builds](#)

Q10 How do you scan container images for vulnerabilities — what tools and workflows exist?

Images & Dockerfile

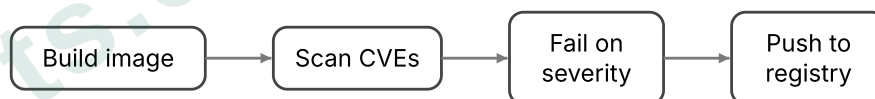
🗣 Spoken answer · ~87 s

Image scanning compares the packages and libraries inside an image against vulnerability databases, such as the NVD and distribution advisories, and reports known CVEs with severity. Tools include Trivy, Gype, Snyk, Docker Scout, Clair, and cloud-native scanners in Artifact Registry, ECR, and Defender for Containers. Beyond OS packages they scan language dependencies, misconfigurations in Dockerfiles, secrets, and licences, and can generate SBOMs. A sound workflow has several points. Developer: scan locally or in the IDE for fast feedback. CI: scan the freshly built image on every pull request or build, and fail the pipeline above a severity threshold, for example critical or high with an available fix; ignoring unfixable findings reduces noise, and a reviewed ignore list with expiry handles exceptions. Registry: scan on push and rescan continuously, because new CVEs are published daily against images already built. Admission: block deployment of images that violate policy, using an admission controller or Binary Authorization. Runtime: monitor running workloads and prioritise by exploitability and exposure, since not every CVE is reachable. Remediate by updating base images, rebuilding frequently, using minimal images to reduce findings, and pinning and updating dependencies automatically with tools like Dependabot. Track metrics such as time to remediate. Scanning is necessary but not sufficient on its own.

Likely follow-ups

- Why rescan images already in the registry?
- How do you cut scanner noise?

🚩 Image scanning



```
trivy image myimage:latest          # scan image for CVEs
```

```
trivy image --severity HIGH,CRITICAL myimage:latest
```

```
trivy fs .                          # scan filesystem/Dockerfile
```

```
trivy image --format sarif -o results.sarif myimage # CI integration
```

```
gype myimage:latest                # alternative scanner (Anchore)
```

```
docker scout cves myimage          # Docker's built-in scanner
```

```
# GCP Artifact Registry: automatic scanning on push
```

```
gcloud artifacts docker images list-vulnerabilities <image-url>
```

Vulnerability scanning layers: CI pipeline (scan before push — fail build on CRITICAL CVEs), registry (GCR/Artifact Registry auto-scan on push), runtime (Falco for runtime anomaly detection). Trivy is the most popular open-source scanner — fast, covers OS packages, language ecosystems (npm, pip, go, cargo), and IaC. Fix vulnerabilities by rebuilding with updated base images — most CVEs come from OS packages in the base layer, not application code.

Supply chain security: Pin base image digests (not tags) in production: FROM

ubuntu@sha256:abc123... A tag like ubuntu:22.04 can be updated by the publisher. A digest is immutable — you know exactly what you're building on. Combine with GCP Binary Authorization to enforce that only scanned, attested images run in production.

 **Official docs**

- [Trivy](#)

Q11 What is .dockerignore — why is it critical for build performance and security?

Images & Dockerfile

 **Spoken answer** · ~87 s

A .dockerignore file lists paths excluded from the build context, which is the set of files the Docker client sends to the daemon at the start of docker build, and is what COPY and ADD can reference. It works like .gitignore. It matters for three reasons. Performance: without it, a large context, such as node_modules, .git, build outputs, logs, and test data, is uploaded on every build, slowing builds considerably. It also affects the cache: COPY . . invalidates the layer whenever any file in the context changes, including irrelevant ones like logs or editor files, so excluding them improves cache hits. Security: it keeps secrets and sensitive files, like .env files, private keys, cloud credentials, and .git history, from being copied into the image, where anyone who can pull it could extract them, since layers persist even if you delete the file later. Image size and reproducibility: unneeded files, like local dependency folders built for a different operating system, don't pollute the image. Typical entries: .git, node_modules, *.log, .env, Dockerfile itself, docker-compose files, test folders, IDE settings, and build directories. A good approach is to start by excluding everything and then allowing only what's needed. Review the context size in the build output to check it's effective.

Likely follow-ups

- How does .dockerignore improve cache hits?
- What sensitive files should it always exclude?

```
cat .dockerignore
```

```
.git
```

```
node_modules
```

```
*.test.js
```

```
.env
```

```
.env.*
```

```
coverage/
```

```
*.log
```

```
Dockerfile*
```

```
docker-compose*
```

```
README.md
```

```
# Check what gets sent to daemon:
```

```
docker build --no-cache . 2>&1 | head -5 # shows build context size
```

When you run `docker build`, Docker sends the entire build context (the directory) to the daemon first. Without `.dockerignore`, this includes `.git` (potentially GBs of history), `node_modules` (hundreds of MB), test files, and importantly `.env` files with secrets. Problems: slow builds (transferring large context over socket), accidentally baking secrets into images (if `COPY . .` is used), and unnecessary layer size. `.dockerignore` is the Docker equivalent of `.gitignore` — always include it.

Secrets in images: Never copy `.env` files, AWS credentials, or SSH keys in `COPY . .`. Even if you delete them in a later layer, they remain in the previous layer and are extractable with `docker history`. Use build-time secrets (`docker build --secret`) or environment variables at runtime, never bake secrets into image layers.

Official docs

- [Docker: Dockerfile best practices](#)

Q12 **What is the OCI (Open Container Initiative) — what are the OCI image and runtime specs?**

Container runtime

 Spoken answer · ~83 s

The Open Container Initiative is a Linux Foundation project that defines open standards for container formats and runtimes so tools from different vendors interoperate. It was formed in 2015 when Docker donated its image format and runc. There are three specifications. The Image Specification defines how an image is structured: a manifest listing the layers and a config, layers as compressed tar archives of filesystem changes identified by digest, an optional index for multi-architecture images, and the config containing environment, entrypoint, and history. The Runtime Specification defines how to run a container from an unpacked filesystem bundle: a config.json describing the process, mounts, namespaces, cgroups, capabilities, and hooks, and the lifecycle operations of create, start, kill, and delete. runc is the reference implementation, with alternatives like crun, gVisor's runsc, and Kata. The Distribution Specification standardises the registry API for pushing and pulling images and artefacts. Benefits: portability, since an image built with Docker, Buildah, or Kaniko runs on containerd, CRI-O, or Podman; no vendor lock-in; a stable base for Kubernetes, which talks to runtimes through the CRI; and the ability to swap runtimes for stronger isolation. OCI artefacts also let registries store non-image content such as signatures and SBOMs.

Likely follow-ups

- What is the difference between the image and runtime specs?
- What does a multi-architecture index do?

OCI Image Spec: defines image format (layers, manifest, config) — all OCI images are compatible

OCI Runtime Spec: defines how to run a container (create, start, kill, delete lifecycle)

runc: reference OCI runtime implementation (used by containerd, Docker)

containerd: high-level runtime that manages images, snapshots, and delegates to runc

CRI-O: lightweight CRI implementation for Kubernetes (uses runc)

gVisor (runsc): sandboxed OCI runtime (GCP uses for GKE Sandbox)

Kata Containers: VM-based OCI runtime (hardware isolation, same API)

The OCI standardizes what a container image looks like and how it runs — any OCI-compliant image works with any OCI-compliant runtime. This is why Docker images work on Kubernetes with containerd, and why you can swap runtimes without changing images. Kubernetes uses the CRI (Container Runtime Interface) to talk to runtimes — containerd and CRI-O implement CRI. gVisor intercepts syscalls in user space, providing stronger isolation than standard containers — used for untrusted workloads in GKE Sandbox.

Runtime selection in GKE: Standard GKE nodes use containerd + runc. GKE Sandbox uses containerd + gVisor (runsc). Enable per node pool. Use Sandbox for multi-tenant or untrusted code (user-submitted jobs, CI runners). The tradeoff is performance overhead (~10-20% slower) vs stronger security isolation.

Official docs

- [OCI Runtime Specification](#)
- [OCI Image Specification](#)

Q13 How does containerd manage containers — how does it differ from Docker?

Container runtime

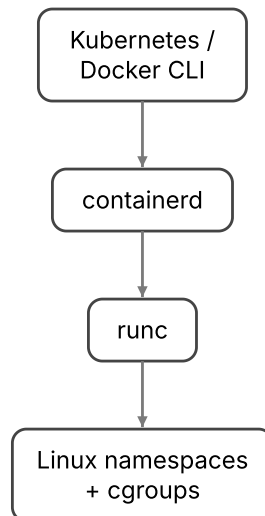
Spoken answer · ~87 s

containerd is a high-level container runtime that manages the complete container lifecycle on a host: pulling and storing images, managing snapshots and storage, creating and supervising containers, and network attachment through plugins, while delegating the actual process creation to an OCI runtime like runc via shims. It was extracted from Docker and donated to the CNCF, where it graduated. The difference from Docker: Docker is a full platform, with the CLI, build tooling, Compose, a daemon with an API, networking and volume management, and developer conveniences, and it uses containerd underneath. containerd is a lean daemon with a gRPC API focused on running containers and is designed to be embedded by other systems; it ships a basic CLI, ctr, and a more user-friendly nerdctl. Kubernetes talks to containerd through the Container Runtime Interface via its built-in CRI plugin, which is why Kubernetes removed dockershim in version 1.24: nodes can use containerd directly without the Docker daemon, with less overhead and fewer moving parts. Practical implications: on Kubernetes nodes, you debug with crictl instead of docker commands; images built with Docker still run unchanged because both follow the OCI specs; and docker-in-docker socket mounts in CI no longer work on those nodes. CRI-O is the other common Kubernetes-focused runtime.

Likely follow-ups

- Why did Kubernetes remove dockershim?
- What replaces docker commands when debugging nodes?

Runtime stack



```
ctr images pull docker.io/library/nginx:latest # containerd CLI
```

```
ctr run docker.io/library/nginx:latest nginx # run with containerd
```

```
ctr containers list # list containers
```

```
ctr tasks list # list running tasks (processes)
```

On a Kubernetes node (use crictl):

```
crictl ps # list running containers
```

```
crictl logs <container-id> # container logs
```

```
crictl inspect <container-id> # container details
```

```
crictl pods # list pods
```

```
crictl exec -it <id> sh # exec into container
```

containerd is a lower-level daemon focused on container lifecycle: pulling images, managing snapshots (layer storage), creating and running containers via runc. It does not include Docker's higher-level features (build, Compose, Swarm). Docker uses containerd internally as its runtime. On Kubernetes nodes, you interact with containerd via crictl (CRI CLI) — not docker commands, which won't work unless Docker is also installed. Use crictl for node-level debugging when kubectl exec is not enough.

crictl vs kubectl exec: kubectl exec runs through the API server → kubelet → CRI. crictl talks directly to containerd. Use crictl when the API server is down, when debugging node-level issues, or when a container is stuck and kubectl exec hangs. It's the emergency debugging tool for Kubernetes nodes.

Official docs

- [Kubernetes: Container runtimes](#)
- [Docker documentation](#)
- [containerd documentation](#)

Q14 What are container resource limits — how do CPU and memory limits work internally?

Container runtime

Spoken answer · ~81 s

Container resource limits are enforced by Linux cgroups. For memory, you set a limit, and the kernel tracks the container's memory use, including page cache; if usage exceeds the limit and memory can't be reclaimed, the kernel's OOM killer terminates a process in the container, usually showing exit code 137 and an OOMKilled flag. There's no throttling for memory, only killing, so limits must be set with headroom. Swap can be limited or disabled. For CPU, limits work differently: the CFS bandwidth controller gives the container a quota of CPU time per period, typically 100 milliseconds, so a limit of 0.5 CPU allows 50 ms per period. If the container uses its quota early, its threads are throttled until the next period, regardless of idle CPU on the host, which causes latency spikes. CPU shares, or requests in Kubernetes, are relative weights that apply only under contention and guarantee a proportional share. Other limits include the number of processes via the pids cgroup, which protects against fork bombs, and block I/O weights. In Docker use `--memory`, `--cpus`, and `--cpu-shares`; in Kubernetes, requests and limits in the pod spec determine scheduling and quality of service class.

Likely follow-ups

- Why is exceeding a memory limit different from a CPU limit?
- What do CPU shares mean under no contention?

```
docker run --memory="512m" --memory-swap="512m" nginx # memory limit + disable swap
```

```
docker run --cpus="1.5" nginx # 1.5 CPU cores
```

```
docker run --cpu-shares=512 nginx # relative weight (default 1024)
```

```
docker run --pids-limit=100 nginx # max processes inside container
```

```
# Check limits set on container:
```

```
cat /sys/fs/cgroup/memory/docker/<id>/memory.limit_in_bytes
```

```
cat /sys/fs/cgroup/cpu/docker/<id>/cpu.cfs_quota_us
```

```
docker stats --no-stream <container> # current usage vs limits
```

Limits are enforced by cgroups in the kernel. Memory limit: container is OOM-killed when RSS exceeds the limit (Exit Code 137). CPU limit: implemented as CFS (Completely Fair Scheduler) quota — `--cpus=1.5` sets `cpu.cfs_quota_us=150000` per 100ms period. CPU throttling is silent: the container runs slowly but isn't killed. Set `--memory-swap` equal to `--memory` to disable swap — otherwise the container can swap and performance degrades significantly rather than failing fast.

Container memory vs host memory: `/proc/meminfo` inside a container shows the HOST's total memory, not the container's limit. Applications that read `/proc/meminfo` (JVM, many Go apps) may configure themselves based on the wrong value. Fix: use JVM flags (`-XX:MaxRAMPercentage`), or cgroup-aware tooling. Many modern runtimes have fixed this but old JVM versions (pre-Java 8u191) do not.

Official docs

- [Kubernetes: Resource management for Pods & containers](#)

Q15 What is a container health check — how do you configure and debug it? Container runtime

Spoken answer · ~88 s

A container health check is a command or probe that tells the platform whether the application inside is actually working, not just whether the process is running. In Docker, the `HEALTHCHECK` instruction or run flags specify a command, an interval, timeout, start period, and retries; the container's status moves between starting, healthy, and unhealthy, visible in `docker ps` and `inspect`. Docker itself only reports the status, though Swarm acts on it. In Kubernetes, you configure probes in the pod spec instead: liveness, readiness, and startup, using HTTP GET, TCP socket, exec commands, or gRPC. A good health endpoint is cheap, fast, and reflects the app's ability to serve; for readiness it may check critical dependencies, but a liveness check should not depend on external services, otherwise a database outage restarts every pod. To debug failing checks: look at the probe definition, event messages in `kubectl describe`, and logs; run the same command or `curl` manually inside the container, since the failure may be a wrong port, path, or auth requirement; check timing, because slow startup needs a startup probe or a longer initial delay, and aggressive timeouts cause false failures under load; check resource starvation and CPU throttling; and ensure the endpoint binds to the pod IP, not just localhost.

Likely follow-ups

- Why shouldn't liveness probes check the database?
- When do you need a startup probe?

```
# In Dockerfile:
```

```
HEALTHCHECK --interval=30s --timeout=5s --start-period=10s --retries=3 \
```

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

```
# At docker run:
```

```
docker run --health-cmd="curl -f http://localhost/health" \
```

```
--health-interval=30s --health-timeout=5s nginx
```

```
# Check health status:
```

```
docker inspect --format "{{.State.Health.Status}}" <container>
```

```
docker inspect --format "{{json .State.Health}}" <container> | jq
```

```
docker ps # shows (healthy) / (unhealthy) / (health: starting)
```

Health checks run a command inside the container periodically. States: starting (within start-period — health failures ignored), healthy (last N retries passed), unhealthy (N consecutive failures). Docker alone does not restart unhealthy containers — it only marks them. Docker Swarm and Kubernetes do react to health state. In Kubernetes, use readiness and liveness probes (separate concepts) instead of Dockerfile HEALTHCHECK — they are more flexible and the kubelet acts on them.

Health check endpoint design: The /health endpoint should check actual application health (DB connection, dependency availability), not just that the process is running. Return 200 if healthy, 503 if not. Keep it fast (<1s) — a slow health check that times out causes cascading failures in load balancers.

Official docs

- [Microsoft: Cloud design pattern – Health Endpoint Monitoring](#)
- [Docker documentation](#)

Q16 How does container-to-container communication work — with and without an orchestrator?

Container networking

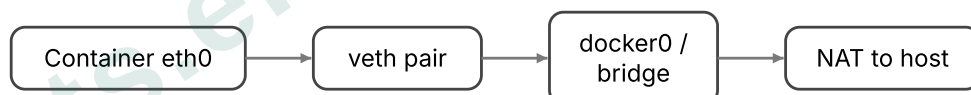
 Spoken answer · ~84 s

Without an orchestrator, containers on a single Docker host communicate through Docker networks. On the default bridge, containers can reach each other by IP address; on a user-defined bridge network, Docker's embedded DNS lets them resolve each other by container name or alias, which is how Compose services find one another. Containers on different networks are isolated unless attached to both. To reach a container from outside, you publish ports on the host. Across hosts, you need overlay networking, as in Swarm, or you route over the host network and published ports, or an external service discovery mechanism. With an orchestrator like Kubernetes, every pod gets its own IP, and the network model requires all pods to communicate with each other without NAT. Because pod IPs are ephemeral, applications talk to Services, which provide a stable virtual IP and DNS name, like `my-service.namespace.svc.cluster.local`, load balanced across ready pods by kube-proxy or eBPF dataplanes. CoreDNS handles name resolution. Ingress or Gateway resources expose services externally, and network policies control which pods may talk. A service mesh adds mTLS, retries, and observability. The key shift is moving from IP-based or name-based single-host links to service discovery with stable names, health-based routing, and policy-controlled traffic.

Likely follow-ups

- Why do pods talk to Services rather than pod IPs?
- How does name resolution work on a user-defined bridge?

Container networking



```
# Docker: same network = communicate by container name
```

```
docker network create appnet
```

```
docker run -d --name db --network appnet postgres
```

```
docker run -d --name api --network appnet myapi
```

```
# api container can reach db as: postgres://db:5432
```

```
# Kubernetes: pods communicate by pod IP or Service DNS
```

```
curl http://db-service.default.svc.cluster.local:5432
```

```
# Within same pod: use localhost (containers share network namespace)
```

```
curl http://localhost:8080 # sidecar talking to main container
```

In Docker: containers on the same user-defined bridge network resolve each other by name via embedded DNS (127.0.0.11). Containers on different networks cannot communicate without explicit routing. In Kubernetes: every pod gets a unique IP, and all pods can reach each other by IP without NAT (flat network model). Services provide stable DNS names and load balancing across pod IPs. Containers in the same pod share a network namespace — they communicate via localhost and must use different ports.

Pod network namespace sharing: All containers in a Kubernetes pod share the same network interface, IP address, and port space. This enables the sidecar pattern (Envoy proxy in Istio intercepts all traffic via localhost). It also means port conflicts between containers in the same pod cause startup failures.



Official docs

- [Docker documentation](#)

Q17 What is port mapping in Docker — how does it work under the hood?

Container networking

Spoken answer · ~86 s

Port mapping, or publishing, makes a container port reachable from outside the host: `docker run -p 8080:80` maps host port 8080 to container port 80. Inside, the container's process listens on its own port in its own network namespace, which isn't visible from outside. Docker implements the mapping with iptables, or nftables, rules: a DNAT rule in the nat table's DOCKER chain rewrites traffic arriving at the host's port 8080 to the container's IP and port 80, and a MASQUERADE rule handles outgoing traffic from containers so replies appear to come from the host. For connections from the host itself via localhost, and for IPv6 or userland handling in some cases, Docker may also run a docker-proxy process that forwards traffic. Traffic flows through the bridge to a veth pair that connects the container namespace. Practical points: publishing with just `-p` binds to all host interfaces, 0.0.0.0, which can expose services unintentionally and also bypass host firewalls such as ufw because Docker edits iptables directly, so bind to 127.0.0.1 when only local access is needed; two containers can't publish the same host port; EXPOSE in a Dockerfile only documents and doesn't publish; with host networking there is no mapping; and in Kubernetes, Services and NodePorts replace this.

Likely follow-ups

- Why can Docker bypass a host firewall like ufw?
- What does EXPOSE actually do?

```
docker run -p 8080:80 nginx # host:8080 → container:80
```

```
docker run -p 127.0.0.1:8080:80 nginx # bind to localhost only
```

```
docker run -P nginx # map all EXPOSE ports to random host ports
```

```
docker port <container>          # show port mappings
```

```
# Underlying iptables rules:
```

```
iptables -t nat -L DOCKER -n      # DNAT rules added by Docker
```

Port mapping uses iptables DNAT rules. Docker adds a rule: when a packet arrives on host port 8080, rewrite the destination to the container IP:80. docker-proxy also handles the loopback case (localhost traffic). The container IP is assigned from the bridge network's subnet (typically 172.17.0.0/16 for the default bridge). Without -p, container ports are not reachable from outside the Docker network. The EXPOSE instruction in Dockerfile is documentation only — it does not publish ports; -p at runtime does.

Security: 0.0.0.0 binding: By default, -p 8080:80 binds to all interfaces (0.0.0.0:8080), making the service reachable from any network interface including public ones. Use -p 127.0.0.1:8080:80 to bind to localhost only for development services. In production, control access via firewall rules or a reverse proxy, not by relying on port mapping.

Official docs

- [Docker: Networking](#)
- [Docker documentation](#)

Q18 **How do you debug container networking issues — what tools and approaches work inside containers?**

Container networking

 Spoken answer · ~84 s

To debug container networking, I work from symptoms to layers. First clarify the failure: can't resolve names, can't connect, timeouts, or intermittent drops, and between which endpoints. Check configuration from outside: `docker inspect` for IP address, network, and ports; `docker network inspect` to see which containers share the network; and published ports with `docker port`. Then test from inside the container. Many production images lack tools, so use a debug approach: `exec` into the container if it has a shell, or attach a troubleshooting container sharing the network namespace, such as `nicolaka/netshoot` with `--network container:name`, or in Kubernetes use `kubectrl debug` with an ephemeral container. Inside, check DNS with `nslookup` or `dig` and the `resolv.conf`; connectivity with `curl`, `nc`, or `telnet` to the target port; routes with `ip route`; listening sockets with `ss -tlnp`, confirming the application binds to `0.0.0.0` and not `127.0.0.1`; and packet traces with `tcpdump`. On the host, inspect `iptables` rules, bridge state, and firewall or security group rules. Common causes: wrong port or binding address, different networks, DNS misconfiguration, MTU mismatches, network policies, and port conflicts. In Kubernetes also verify Service selectors and endpoints. Narrow the problem step by step: DNS, then routing, then port, then application.

Likely follow-ups

- Why does binding to `127.0.0.1` break connectivity from other containers?
- What is an ephemeral debug container?

```
# Container lacks tools – use a debug sidecar or nsenter from host
```

```
docker run --network container:<target-id> nicolaka/netshoot # network debug image
```

```
# nsenter from host into container network namespace:
```

```
PID=$(docker inspect -f "{{.State.Pid}}" <container>)
```

```
nsenter -t $PID -n -- ss -tlnp # run ss in container's net namespace
```

```
nsenter -t $PID -n -- tcpdump -i eth0 port 80
```

```
# From inside container (if tools available):
```

```
curl -v http://other-service:8080
```

```
nslookup other-service
```

```
cat /etc/resolv.conf # check DNS config
```

```
ip route show # routing table inside container
```

nicolaka/netshoot is an invaluable debug container — it includes curl, dig, nslookup, tcpdump, ip, ss, ping, traceroute, and many more tools. Run it sharing the target container's network namespace (`--network container:ID`) to debug from within that namespace. nsenter from the host lets you run host tools inside any container namespace without restarting anything — critical for debugging minimal/distroless containers.

Kubernetes network debug: `kubectl debug node/<nodename> -it --image=nicolaka/netshoot -- bash` — creates a debug pod on a specific node with access to host namespaces. Or: `kubectl run tmp --rm -i --tty --image=nicolaka/netshoot -- bash` to spin up a debug pod in the cluster for DNS/connectivity testing.

Official docs

- [Docker: Networking](#)

Q19 What are Docker volumes — how do bind mounts, volumes, and tmpfs differ?

Storage & volumes

Spoken answer · ~88 s

Docker offers three ways to give containers storage beyond the writable layer. Volumes are managed by Docker, stored under `/var/lib/docker/volumes`, independent of the container's lifecycle, and the preferred mechanism for persistent data: they can be created, backed up, and shared between containers, support volume drivers for remote storage, work on Linux and Windows, and get initial content from the image. Bind mounts map a specific host path into the container; they're useful in development for live code reloading and for sharing host configuration, but depend on the host's directory structure, can expose sensitive host files, and have permission and ownership pitfalls; they offer higher performance on Linux but can be slow on Docker Desktop for macOS. tmpfs mounts store data in the host's memory only, never written to disk, and vanish when the container stops; they suit sensitive temporary data like secrets or fast scratch space. Choose volumes for databases and application data in single-host deployments, bind mounts for development and config injection, tmpfs for ephemeral sensitive data. Without any of these, data in the writable layer is lost when the container is removed, and writes are slower. Use read-only mounts where possible. In Kubernetes, PersistentVolumes and claims abstract this, while emptyDir and hostPath correspond to tmpfs-like and bind-style usage.

Likely follow-ups

- Why prefer volumes over bind mounts in production?
- What is tmpfs good for?

🚧 Storage types

Volume	Bind mount / tmpfs
Managed by Docker, persistent	Host path / memory
Best for data	Dev use / scratch

Named volume (managed by Docker, persists across container restarts):

```
docker volume create mydata
```

```
docker run -v mydata:/data/db postgres
```

```
docker volume ls; docker volume inspect mydata
```

Bind mount (mount host directory into container):

```
docker run -v /host/path:/container/path nginx
```

```
docker run --mount type=bind,src=/host/path,dst=/container/path nginx
```

tmpfs (RAM-backed, non-persistent):

```
docker run --tmpfs /tmp:size=100m nginx
```

Inspect volume mounts on a container:

```
docker inspect <container> --format "{{json .Mounts}}" | jq
```

Named volumes: managed by Docker, stored in `/var/lib/docker/volumes/`, best for production data, can be backed by volume plugins (NFS, cloud block storage). Bind mounts: map a specific host path into the container, good for development (live code reload), risky in production (host directory can be modified from outside). tmpfs: RAM-backed, not persisted to disk, good for secrets (never written to disk) and performance-sensitive temp data. In Kubernetes: use `PersistentVolumeClaims` for production storage — they abstract the underlying storage backend.

Volume permissions: A common problem with bind mounts is file ownership mismatch — the container runs as UID 1001 but the host directory is owned by UID 0 (root). Files are unreadable or unwritable. Fix: set correct ownership on host directory, or use `:z/:Z` selinux flags on RHEL, or run the container with `--user` to match host UIDs.

Official docs

- [Linux man-pages: mount_namespaces\(7\)](#)
- [Docker: Volumes](#)
- [Docker documentation](#)

Q20 How do you manage persistent data for stateful containers in production?

Storage & volumes

Spoken answer · ~85 s

Persistent data for stateful containers needs storage that outlives the container and can follow it when scheduled elsewhere. On Kubernetes, use PersistentVolumeClaims backed by a StorageClass that dynamically provisions cloud disks through a CSI driver, and StatefulSets with volumeClaimTemplates to give each replica a stable identity and its own volume. Choose the storage type by need: block storage for databases, file storage like NFS or Filestore for shared access, and object storage for large data. Think about access modes, zone affinity, since disks are zonal and pods must schedule in the same zone, and volume expansion. Protect the data: take regular snapshots with VolumeSnapshots, run application-level backups to object storage, test restores, and set reclaim policy to Retain for important volumes. For availability, rely on database replication rather than only storage: run primary and replicas, with operators like CloudNativePG, the Percona operators, or Strimzi managing failover, backups, and upgrades. Often the best choice is a managed database service, such as Cloud SQL or RDS, instead of running databases in containers, trading control for reduced operational burden. Monitor volume usage and I/O latency, set alerts before disks fill, and apply resource limits. Avoid hostPath for production data, document recovery procedures, and rehearse disaster recovery.

Likely follow-ups

- Why is zone affinity important for disks?
- When would you choose a managed database over containers?

```
# Kubernetes: PersistentVolumeClaim binds to PersistentVolume
```

```
kubectl get pvc -n production    # PVC status (Bound/Pending/Lost)
```

```
kubectl describe pvc <name>      # events show provisioning issues
```

```
kubectl get pv                  # available persistent volumes
```

```
# GKE: use StorageClass for dynamic provisioning
```

```
kubectl get storageclass         # available storage classes
```

```
kubectl describe sc standard-rwo # see provisioner and parameters
```

```
# StatefulSet: each pod gets its own PVC
```

```
kubectl get pvc -l app=postgres # one PVC per StatefulSet pod
```

Production stateful data principles: use PersistentVolumes with appropriate storage classes (SSD for databases, standard for logs), use StatefulSets for databases (ordered deployment, stable network identity, per-pod PVCs), implement backup strategies (volume snapshots, logical backups), and design for failure (PVC survives pod deletion — data is safe). StorageClass with volumeBindingMode=WaitForFirstConsumer delays PVC binding until a pod is scheduled, ensuring the volume is provisioned in the same zone as the pod.

Avoid stateful containers where possible: Use managed cloud databases (Cloud SQL, RDS) instead of running databases in containers. Stateful containers require careful orchestration. The complexity of HA databases in Kubernetes (leader election, replication, backup, recovery) often exceeds the benefits. Use containers for stateless services; use managed services for databases.

Official docs

- [Kubernetes: Persistent Volumes](#)
- [Kubernetes: StorageClasses](#)

Q21 What is a rolling deployment — how does Kubernetes implement it and what can go wrong?

Deployment strategies

Spoken answer · ~83 s

A rolling deployment replaces old pod instances with new ones gradually, so the service stays available. In Kubernetes, a Deployment's RollingUpdate strategy creates a new ReplicaSet and scales it up while scaling the old one down, controlled by maxSurge, how many extra pods can exist above the desired count, and maxUnavailable, how many can be missing. A new pod must pass its readiness probe before it receives traffic and before the controller proceeds, and old pods receive SIGTERM with a termination grace period. What can go wrong: no or poor readiness probes, so traffic hits pods that aren't ready, causing errors; a bad version rolling out slowly with failing pods stuck, where progressDeadlineSeconds marks the rollout as failed but doesn't automatically roll back; insufficient capacity or quota so new pods stay pending; strict settings that make the rollout very slow; incompatible versions running simultaneously, such as database schema or API changes, requiring backward compatibility; connection draining issues where in-flight requests are cut because the application ignores SIGTERM or load balancer deregistration lags, fixed with preStop hooks; PodDisruptionBudgets interfering; and long-lived connections not rebalancing. Mitigate with good probes, graceful shutdown, a canary phase, monitoring, and tested rollbacks with kubectl rollout undo.

Likely follow-ups

- What do maxSurge and maxUnavailable control?
- Why does a preStop hook help during rollouts?

```
kubectl set image deployment/myapp myapp=myimage:v2 # trigger rolling update
```

```
kubectl rollout status deployment/myapp # watch progress
```

```
kubectl rollout history deployment/myapp # revision history
```

```
kubectl rollout undo deployment/myapp # rollback to previous
```

```
kubectl rollout undo deployment/myapp --to-revision=3 # rollback to specific
```

```
# Control rolling update speed:
```

```
strategy:
```

```
  type: RollingUpdate
```

```
  rollingUpdate:
```

```
    maxUnavailable: 0 # never reduce below desired replicas
```

```
    maxSurge: 1 # allow 1 extra pod during update
```

Rolling update: Kubernetes gradually replaces old pods with new ones. With `maxUnavailable=0` and `maxSurge=1`, it starts 1 new pod, waits for it to become Ready, then terminates 1 old pod — repeating until all pods are updated. What can go wrong: new pod crashes (`CrashLoopBackOff`) causing rollout to stall (stuck at partial deployment), readiness probe failing causing zero traffic to new pods (rollout appears complete but service is degraded), or a slow rollout leaving both versions running simultaneously causing compatibility issues.

Rollout stall detection: Set `progressDeadlineSeconds` on the Deployment (default 600s). If the rollout doesn't make progress within this time, it marks the deployment as Failed — this triggers alerts. Without this, a stalled rollout can go undetected for hours.

Official docs

- [Kubernetes: Deployment strategies \(rolling update\)](#)

Q22 What is blue-green deployment — how do you implement it in Kubernetes?

Deployment strategies

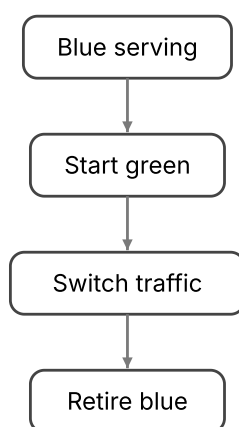
 Spoken answer · ~86 s

Blue-green deployment runs two full environments: blue is the current production, and green is the new version, deployed and tested alongside it with no live traffic. When green is verified, you switch all traffic from blue to green at once, and keep blue available for instant rollback. In Kubernetes, the simplest approach uses two Deployments with a version label, say `app=web` with `version=blue` and `version=green`, and one Service whose selector points to the active version. To cut over, patch the Service selector to `version=green`; rollback means patching it back. You can test green beforehand through a separate preview Service or ingress hostname. Alternatives use ingress or Gateway API routing rules, or Argo Rollouts, which has a built-in blueGreen strategy with active and preview services, optional automated analysis, and a promotion step with scale-down delay for the old version. Benefits: instant cutover and rollback, and full testing of the production-like environment. Costs and challenges: double the resources during the switch; database changes must be backward compatible, since both versions touch the same data; long-lived connections and sessions need handling; and cutover is all or nothing, so unlike a canary it exposes everyone to risk simultaneously. Pair it with smoke tests and monitoring before and after the switch.

Likely follow-ups

- How does the cutover happen in Kubernetes?
- What is the main resource cost of blue-green?

Blue-green



```
# Blue (current v1) Deployment running
```

```
kubectl get deployment myapp-blue # v1 running, service pointing here
```

```
# Deploy green (v2) alongside
```

```
kubectl apply -f deployment-green.yaml # v2 Deployment (version: green)
```

```
kubectl rollout status deployment/myapp-green # wait until ready
```

```
# Switch service selector from blue to green (atomic switch)
```

```
kubectl patch svc myapp -p '{"spec":{"selector":{"version":"green"}}}'
```

```
# Verify, then decommission blue
```

```
kubectl delete deployment myapp-blue
```

```
# Rollback: switch selector back to blue (instant)
```

```
kubectl patch svc myapp -p '{"spec":{"selector":{"version":"blue"}}}'
```

Blue-green runs two identical environments simultaneously. The Service selector points to blue (production). Green (new version) is deployed and tested with zero production traffic. When ready, the Service selector is switched — instant cutover with zero downtime. Rollback is instant (switch selector back). Advantages: instant cutover, instant rollback, full testing of new version before any user traffic. Disadvantages: requires 2x resources during deployment, database migrations must be backward-compatible (both blue and green hit the same DB).

Database migrations: The hardest part of blue-green is schema changes. The new version may need schema changes the old version can't handle. Use the expand-contract pattern: first expand schema (add nullable columns, new tables — both versions work), then cut over, then contract (remove old columns after old version is gone). Never do breaking schema changes in a blue-green deployment.

Official docs

- [Argo Rollouts](#)
- [Martin Fowler: BlueGreenDeployment](#)

Q23 What is canary deployment — how do you implement traffic splitting in Kubernetes?

Deployment strategies

 Spoken answer · ~85 s

A canary deployment releases a new version to a small fraction of traffic first, evaluates its health, and gradually increases exposure, limiting the blast radius. In Kubernetes there are several ways to split traffic. The basic approach uses two Deployments behind one Service with the same label, where the traffic ratio follows the replica ratio, for instance one canary pod to nine stable pods gives about ten percent; it's simple but coarse and tied to pod counts. For precise control, use an ingress controller with canary annotations, such as NGINX's canary-weight or header-based routing, or the Gateway API with weighted backendRefs. A service mesh like Istio or Linkerd offers weighted routing independent of replicas, plus header-based targeting and mTLS. Argo Rollouts and Flagger automate the whole process: they shift weights in steps, such as five, twenty-five, fifty, and one hundred percent, pause between steps, run analysis comparing canary metrics from Prometheus against thresholds for error rate and latency, and abort and roll back automatically if they fail. Requirements: good observability labelled by version, enough traffic for statistical significance, backward-compatible data changes, and consistent routing for users where needed. Canaries are safer than rolling updates for risky changes, at the cost of added complexity.

Likely follow-ups

- Why does replica-ratio splitting lack precision?
- What does automated canary analysis compare?

```
# Basic canary: use replica ratio to split traffic
```

```
kubectl scale deployment myapp-stable --replicas=9 # 90% traffic
```

```
kubectl scale deployment myapp-canary --replicas=1 # 10% traffic
```

```
# Both have same label (app: myapp) so Service routes to both
```

```
# Advanced: Argo Rollouts for metric-based canary
```

```
kubectl argo rollouts set image myapp myapp=myimage:v2
```

```
kubectl argo rollouts get rollout myapp --watch
```

```
kubectl argo rollouts promote myapp # advance to next canary step
```

```
kubectl argo rollouts abort myapp # rollback if metrics fail
```

```
# Istio/service mesh: weight-based traffic splitting
```

```
kubectl apply -f virtual-service-canary.yaml # 10% to v2, 90% to v1
```

Canary deployment routes a small percentage of traffic to the new version, monitors metrics, and gradually increases if successful. The replica-ratio approach is simple but not precise (k8s load balancing isn't perfectly proportional). Argo Rollouts provides: automated canary steps, metric-based promotion (only advance if error rate < 1%), automatic rollback on threshold breach, and pause/resume controls. Istio/Linkerd provide precise traffic splitting at the request level (not replica level).

Canary metrics: Monitor: error rate (5xx responses), P99 latency, business metrics (conversion rate, checkout success). Alert thresholds: if error rate exceeds 1% or P99 latency increases >10% vs baseline, roll back automatically. The power of canary is catching issues before they affect 100% of users.

Official docs

- [Google SRE Workbook: Canarying Releases](#)
- [Argo Rollouts](#)
- [Martin Fowler: CanaryRelease](#)

Q24 What is a recreate deployment strategy — when is it appropriate?

Deployment strategies

Spoken answer · ~88 s

A recreate strategy stops all old instances first and then starts the new version, so there is a period of downtime between them. In Kubernetes, set strategy type Recreate on the Deployment: the old ReplicaSet scales to zero, and only after all old pods terminate does the new ReplicaSet scale up. It is simple and guarantees that two versions never run at the same time. When it's appropriate: applications that can't tolerate two versions running concurrently, for example because of incompatible database schema changes that can't be made backward-compatible, or software that holds an exclusive lock or licence, or uses a single-writer volume such as a ReadWriteOnce disk that cannot attach to two pods on different nodes; non-production and development environments where downtime doesn't matter and saving resources does; batch or scheduled workloads; singleton workers where duplicate processing would be harmful; and low-traffic internal tools with a maintenance window. Drawbacks: the downtime, which lasts as long as shutdown plus startup plus readiness; no gradual validation and rollback also needs a full restart. Mitigation: schedule during low traffic, communicate maintenance, make startup fast, and show a maintenance page. For anything user-facing and critical, prefer rolling, blue-green, or canary strategies, and invest in making changes backward compatible so you don't need Recreate.

Likely follow-ups

- Why would a single-writer volume force Recreate?
- How can you reduce Recreate downtime?

strategy:

```
type: Recreate # in Kubernetes Deployment spec
```

```
# Effect: terminates ALL old pods before starting new ones
```

```
# Causes downtime equal to: pod termination + new pod startup time
```

```
kubectl apply -f deployment-recreate.yaml
```

```
kubectl rollout status deployment/myapp # watch: old pods die, then new start
```

Recreate strategy terminates all existing pods before creating new ones — causing deliberate downtime. Use when: the application cannot run two versions simultaneously (singleton processes, exclusive resource locks, non-backward-compatible DB migrations), stateful applications where having two versions would cause data corruption, or resource-constrained environments where you cannot run old and new simultaneously. Also use for development/staging environments where downtime is acceptable and simplicity is preferred over zero-downtime complexity.

When to choose recreate: If your application uses a distributed lock, holds an exclusive file lock, or connects to a messaging queue that can only have one consumer per partition — running two versions simultaneously would cause conflicts. Recreate ensures clean handoff at the cost of a brief downtime window.

Official docs

- [Kubernetes: Deployments](#)
- [Kubernetes: Deployment strategies \(rolling update\)](#)

 Spoken answer · ~89 s

A rollback strategy is how you return quickly to a known-good version. In Kubernetes, Deployments keep revision history in ReplicaSets, controlled by `revisionHistoryLimit`, so `kubectl rollout undo` returns to the previous revision or to a specific one, and `kubectl rollout history` shows what's available. Helm offers `helm rollback` to a prior release revision. With GitOps tools such as Argo CD, the right rollback is to revert the commit in the configuration repository, so Git remains the truth, and the controller reconciles; using manual rollbacks in-cluster would be reverted by self-heal. Progressive delivery tools like Argo Rollouts and Flagger roll back automatically when analysis fails, and blue-green rollbacks just switch the Service back. In CI/CD, because artefacts are immutable and versioned, the pipeline can redeploy a previous image digest through a one-click job. Feature flags provide the quickest rollback with no deployment. The hard parts are data and dependencies: database migrations should follow expand and contract so the old code still works with the new schema; messages and caches might have a new format; irreversible changes require a forward-fix plan. Make rollback decisions based on defined triggers from monitoring, such as an error rate above a threshold. Practise rollbacks regularly, keep recent artefacts available, document who can trigger it, and measure time to restore.

Likely follow-ups

- Why must rollback in GitOps go through Git?
- What makes database changes a rollback risk?

Kubernetes deployment rollback:

```
kubectl rollout history deployment/myapp          # show revision history
```

```
kubectl rollout history deployment/myapp --revision=3 # inspect revision
```

```
kubectl rollout undo deployment/myapp            # roll back to previous
```

```
kubectl rollout undo deployment/myapp --to-revision=2
```

GitOps rollback (preferred): revert the Git commit

```
git revert HEAD          # create revert commit
```

```
git push origin main     # ArgoCD detects and syncs
```

Image tag rollback:

```
kubectl set image deployment/myapp myapp=myimage:v1.2.3 # explicit previous tag
```

```
# Argo Rollouts: automatic rollback on metric threshold
```

```
kubectl argo rollouts abort myapp # manual abort triggers rollback
```

Rollback strategies: Kubernetes deployment rollback uses stored revision history (revisionHistoryLimit, default 10). This is fast but manual. GitOps rollback reverts the Git commit — ArgoCD detects the change and re-syncs, using Git as the single source of truth. Image tag pinning: always deploy with explicit version tags (myapp:1.2.3) not mutable tags (myapp:latest) — you can roll back to any previous tag. Argo Rollouts automates rollback based on Prometheus metrics — no human required.

:latest tag anti-pattern: Never use :latest in production Deployments. It's mutable (can be overwritten) and makes rollback ambiguous (which "latest" are you rolling back to?). Always use immutable tags: semantic versions (v1.2.3), git commit SHAs, or build numbers. This is table-stakes for production deployments.

Official docs

- [Kubernetes: Deployments](#)
- [AWS Builders' Library: Ensuring rollback safety during deployments](#)

Q26 What is a PodDisruptionBudget — how does it protect availability during deployments?

Deployment strategies

Spoken answer · ~84 s

A PodDisruptionBudget limits how many pods of an application can be voluntarily disrupted at once, protecting availability during planned operations. You define it with a selector for the pods and either minAvailable or maxUnavailable, as a number or percentage. For example, a deployment with five replicas and minAvailable of four means at most one pod can be evicted at a time. It applies to voluntary disruptions that go through the Eviction API: node drains for upgrades, cluster autoscaler scale-downs, and tools like kubectl drain. If an eviction would violate the budget, it's refused, and the drain waits and retries. It doesn't protect against involuntary disruptions such as node crashes or hardware failure, and it doesn't govern rolling updates of Deployments directly, which are controlled by maxSurge and maxUnavailable, though both matter for overall availability. Best practices: define PDBs for critical, multi-replica workloads; avoid settings that block all evictions, such as minAvailable equal to the replica count or a single replica with minAvailable of one, because they stall node upgrades and autoscaling; combine with topology spread constraints and anti-affinity so replicas are on different nodes and zones; and use readiness probes, as only ready pods count as available. Review PDBs before cluster upgrades.

Likely follow-ups

- What disruptions does a PDB not cover?
- Why can a too-strict PDB block upgrades?

```
kubectl apply -f - <<EOF
```

```
apiVersion: policy/v1
```

```
kind: PodDisruptionBudget
```

```
metadata:
```

```
  name: myapp-pdb
```

```
spec:
```

```
  minAvailable: 2          # or maxUnavailable: 1
```

```
  selector:
```

```
    matchLabels:
```

```
      app: myapp
```

```
EOF
```

```
kubectl get pdb          # shows current disruptions allowed
```

```
kubectl describe pdb myapp-pdb # shows allowed disruptions now
```

PDB guarantees a minimum number of pods remain available during voluntary disruptions: node drains, rolling updates, cluster upgrades. With `minAvailable: 2` and 3 replicas, draining a node will not proceed until 2 other replicas are running and ready. PDBs only protect against voluntary disruptions — a node crash (involuntary) bypasses PDB protection. Common mistake: setting `minAvailable` equal to total replicas blocks all drains permanently. Always set `minAvailable` less than total replicas to allow at least 1 pod to be evicted.

PDB for production: Every production Deployment with >1 replica should have a PDB. Without it, a cluster upgrade could drain multiple nodes simultaneously, taking down all replicas. Combine PDB with anti-affinity (spread replicas across nodes) for true high availability — PDB prevents mass eviction, anti-affinity prevents all replicas being on one node.

Official docs

- [Kubernetes: Disruptions & PodDisruptionBudget](#)
- [Kubernetes: Safely drain a node](#)

Spoken answer · ~88 s

Container security risks cluster around the image, the runtime, the host, and the orchestration. Image risks: vulnerable base images and dependencies, malicious or untrusted images from public registries, secrets baked into layers, and bloated images with unnecessary tools. Mitigate by using minimal trusted bases, scanning in CI and registries, pinning digests, signing images, and generating SBOMs. Runtime risks: running as root, privileged containers, excessive Linux capabilities, writable filesystems, and exposed sensitive host paths or the Docker socket; mitigate by running as non-root, dropping all capabilities and adding only what's needed, read-only root filesystems, seccomp and AppArmor or SELinux profiles, no privilege escalation, and Pod Security Standards enforced by admission. Resource risks: no limits allow noisy neighbours and denial of service, so set requests and limits. Network risks: flat networks allow lateral movement, so use network policies, mTLS, and restricted egress. Secrets risks: store them in a secrets manager with least privilege rather than in environment files or images. Orchestrator risks: overly permissive RBAC, exposed API servers or dashboards, and unrestricted service accounts; use least privilege, audit logs, and private control planes. Host risks: kernel vulnerabilities, since containers share the kernel, so patch nodes and consider sandboxed runtimes like gVisor. Finally add runtime detection with tools like Falco, and maintain incident response.

Likely follow-ups

- Why does a shared kernel matter for container security?
- What does dropping capabilities achieve?

TOP RISKS AND MITIGATIONS

1. Running as root → run as non-root user (USER 1001 in Dockerfile)
2. Privileged containers → never use `--privileged`; drop ALL capabilities
3. Secrets in images → use runtime env vars or secret stores (Vault, SM)
4. Writable root filesystem → `readOnlyRootFilesystem: true` in K8s
5. Unscanned images → trivy scan in CI; block CVE-CRITICAL images
6. Unrestricted network → NetworkPolicies to deny by default
7. Over-privileged SA → dedicated SA with minimal RBAC per workload
8. No resource limits → always set CPU and memory limits

Kubernetes securityContext (enforce all the above):

securityContext:

runAsNonRoot: true

runAsUser: 1001

allowPrivilegeEscalation: false

readOnlyRootFilesystem: true

capabilities:

drop: ["ALL"]

Container security is defense-in-depth: secure the image (scanning, minimal base, non-root), secure the runtime (no privileged, dropped capabilities, read-only filesystem), secure the network (NetworkPolicies, mTLS), and secure access (RBAC, workload identity). The Pod Security Standards (restricted, baseline, privileged) provide a framework — enforce restricted in production namespaces.

CKS exam core topic: All of these are tested in CKS. Know how to configure securityContext at both pod and container level, how to enforce PSA labels on namespaces, and how to use Falco rules to detect runtime security violations. The restricted PSS profile covers most of the above automatically.

Official docs

- [Docker: Engine security](#)

Q28 What is a privileged container — why is it dangerous and how do you detect them?

Security

🗣 Spoken answer · ~85 s

A privileged container runs with almost all the host's capabilities and access to all host devices, with the usual security restrictions like seccomp and AppArmor profiles disabled. In Docker it's `--privileged`; in Kubernetes it's `securityContext.privileged true`. It's dangerous because the isolation barrier practically disappears: a process inside can mount host filesystems, load kernel modules, access devices such as disks, manipulate the network stack, and escape to the host, from where it can access other containers, secrets, and credentials. A compromised privileged container is effectively root on the node, and in a cluster that means possible takeover, especially with cloud credentials on the node. Related risks include `hostPID`, `hostNetwork`, `hostPath` mounts, a mounted Docker socket, and added capabilities such as `SYS_ADMIN`. Legitimate uses are rare, such as certain CNI, storage drivers, or monitoring agents, and then only in dedicated namespaces. Detection and prevention: scan manifests in CI with tools such as `kubesecc`, `Checkov`, or `Trivy`; enforce Pod Security Admission at the baseline or restricted level, or policies via Kyverno or OPA Gatekeeper to deny privileged pods; audit running workloads with `kubectl` queries on `securityContext`; use runtime tools such as Falco to alert on privileged container starts; and restrict who can deploy to privileged namespaces with RBAC.

Likely follow-ups

- What practical alternatives exist to privileged mode?
- How do admission policies block privileged pods?

🚩 Privileged container

Normal	Privileged
Dropped capabilities	All host devices and caps
Isolated	Container escape risk

```
# Privileged mode: container has ALL capabilities, can see all devices
```

```
docker run --privileged myimage # almost like running as root on host
```

```
# Detection: find privileged containers in K8s cluster
```

```
kubectl get pods -A -o json | jq '.items[] | select(.spec.containers[].securityContext.privileged==true) | .metadata.name'
```

```
# OPA/Gatekeeper policy to block privileged:
```

```
kubectl get constrainttemplate k8spsprivilegedcontainer
```

```
# Kyverno policy:
```

```
kubectl get clusterpolicy disallow-privileged-containers
```

```
# Check capabilities on running container:
```

```
docker inspect <container> | jq .[].HostConfig.CapAdd
```

A privileged container has all 40+ Linux capabilities — effectively root on the host. It can mount host filesystems, modify iptables, load kernel modules, and escape the container entirely. A privileged container escape is trivial: mount the host filesystem, chroot, and you're on the host. Legitimate uses are rare: node-level agents (CNI plugins, DaemonSet node exporters), kernel module loading. Use specific capabilities (CAP_NET_ADMIN) instead of --privileged whenever possible.

Container escape via privileged: `docker run --privileged --pid=host ubuntu nsenter -t 1 -m -u -i -n -p - - bash` — this gives a root shell on the host from inside a "container." This is why admission controllers blocking privileged containers are non-negotiable in production clusters.

Official docs

- [Kubernetes: Pod Security Standards](#)

Q29 How do you manage secrets in containers — what are the patterns and anti-patterns?

Security

🗣 Spoken answer · ~85 s

Secrets in containers should be kept out of images and source code, delivered at runtime, and limited by least privilege. Good patterns: store secrets in a dedicated manager such as HashiCorp Vault, AWS Secrets Manager, Google Secret Manager, or Azure Key Vault, and deliver them to pods through the External Secrets Operator, the Secrets Store CSI driver, or a Vault agent sidecar. Mount them as files in a memory-backed volume rather than exposing them as environment variables where practical, because environment variables leak through process listings, crash dumps, and child processes. Kubernetes Secrets are only base64 encoded, so enable encryption at rest with a KMS provider, restrict access with RBAC, and avoid broad get and list permissions. Use workload identity so pods authenticate to the manager without static credentials. Prefer short-lived, dynamically generated credentials with automatic rotation. For GitOps use Sealed Secrets or SOPS so encrypted values can be committed. For builds, use BuildKit secret mounts. Anti-patterns: secrets in Dockerfiles, ENV or ARG, which persist in image history; committing .env files; sharing one secret across many services; logging secrets; never rotating them; and passing them on command lines. Scan repositories and images for leaked secrets, audit access, and have a rotation and revocation process.

Likely follow-ups

- Why are environment variables a weak way to deliver secrets?
- What does the External Secrets Operator do?

Anti-patterns:

```
ENV DB_PASSWORD=mysecret      # baked into image – never do this
```

```
COPY .env /app/.env           # .env in image layer – never do this
```

Pattern 1: Kubernetes Secrets (base64 encoded, etcd encrypted)

```
kubectl create secret generic db-creds --from-literal=password=secret
```

```
envFrom: [{secretRef: {name: db-creds}}] # mount as env vars
```

```
volumeMounts: [{name: db-creds, mountPath: /etc/secrets}] # mount as files
```

Pattern 2: External Secrets Operator + GCP Secret Manager

```
kubectl get externalsecret db-password # syncs from GCP SM to K8s Secret
```

Pattern 3: Vault Agent sidecar injection

```
vault.hashicorp.com/agent-inject: "true" # pod annotation
```

```
vault.hashicorp.com/role: "myapp"
```

Secret management hierarchy: env vars in Dockerfile (worst — in image layer) → Kubernetes Secrets without encryption-at-rest (basic, acceptable) → Kubernetes Secrets with etcd encryption (better) → External Secrets Operator syncing from GCP Secret Manager/AWS SSM (production standard) → Vault with agent injection (enterprise standard). Prefer mounted files over environment variables for secrets — env vars appear in process listings, crash dumps, and logs more easily than file-mounted secrets.

External Secrets Operator: ESO syncs secrets from GCP Secret Manager, AWS SSM, Azure Key Vault into Kubernetes Secrets automatically. The secret never needs to be in Git. Rotation is automatic (ESO re-syncs on a schedule). The GCP SA accessing Secret Manager uses Workload Identity — no key files needed anywhere. This is the production-grade pattern for secret management in GKE.

Official docs

- [Microsoft: Anti-patterns \(performance antipatterns\)](#).

Q30 What is container image signing and attestation — how does it fit into supply chain security?

Security

Spoken answer · ~88 s

Image signing creates a cryptographic proof that an image was produced by a trusted party and hasn't been altered. You sign the image digest with a private key, or keyless through short-lived certificates tied to an OIDC identity, as Sigstore Cosign supports, and store the signature in the registry alongside the image. Consumers verify the signature before running. Attestations go further: signed statements about an image, such as build provenance describing who built it, from which source commit and builder; an SBOM listing contents; vulnerability scan results; or test results. Standards include SLSA provenance and in-toto attestations. In supply chain security this addresses tampering, registry compromise, and unauthorised builds: even if an attacker pushes a malicious image, it won't carry a valid signature from your pipeline. A typical flow: CI builds the image, scans it, pushes it, then signs it with Cosign and attaches provenance and an SBOM. At deployment, an admission controller such as Kyverno, Sigstore policy-controller, Ratify, or Binary Authorization verifies the signature and required attestations against policy and rejects anything else. Deploy by digest, protect signing keys in a KMS or use keyless identity, rotate and audit them, and record signing events in a transparency log such as Rekor. Signing proves origin, not safety, so combine with scanning.

Likely follow-ups

- What does keyless signing with Cosign rely on?
- How is an attestation different from a signature?

```
# Sign image with cosign (sigstore):
```

```
cosign sign --key cosign.key myregistry/myapp@sha256:abc123
```

```
cosign verify --key cosign.pub myregistry/myapp:v1.0.0
```

```
# Keyless signing (OIDC-based, no key management):
```

```
cosign sign myregistry/myapp:v1.0.0 # signs with GitHub Actions OIDC identity
```

```
# GCP Binary Authorization attestation:
```

```
gcloud container binauthz attestations sign-and-create \
```

```
--artifact-url=<image-digest-url> \
```

```
--attestor=projects/myproj/attestors/vuln-scan-passed \
```

```
--keyversion=projects/myproj/locations/global/keyRings/binauthz/cryptoKeyVersions/1
```

```
# Enforce: GKE rejects pods with unattested images
```

```
gcloud container binauthz policy import policy.yaml --project=myproj
```

Image signing proves that a specific image digest was produced by a specific identity (CI pipeline).

Attestation adds semantic claims: "this image passed vulnerability scan," "this image was built from commit abc123." Binary Authorization (GCP) enforces at deploy time: only images with required attestations can run. This closes the supply chain loop: source → build → scan → attest → enforce. cosign (sigstore) is the open-source standard for image signing — integrates with GitHub Actions, GitLab CI, and GCP.

SLSA framework: Supply-chain Levels for Software Artifacts. Level 3 (achievable with GCP Cloud Build) provides: signed provenance, ephemeral build environments, and two-party review. This is increasingly required for enterprise and government procurement. Knowing SLSA levels demonstrates Principal-level supply chain security thinking.



Official docs

- [GitHub: Artifact attestations \(build provenance\)](#)
- [SLSA specification](#)
- [Sigstore documentation](#)

Q31 What is a Kubernetes Deployment vs StatefulSet vs DaemonSet — when do you use each?

Orchestration

🗣 Spoken answer · ~87 s

These three controllers fit different workload types. A Deployment manages stateless applications: replicas are interchangeable, pods get random names, can start and stop in any order, share or don't need persistent identity, and scale and roll out easily with rolling updates. Use it for web servers, APIs, and workers. A StatefulSet manages stateful applications that need stable identity: pods get ordinal names like db-0, db-1, with stable network identities through a headless Service, ordered startup, scaling, and termination, and each replica gets its own PersistentVolumeClaim from volumeClaimTemplates that stays attached when the pod is rescheduled. Use it for databases, message brokers, and clustered systems like Kafka, Zookeeper, or Elasticsearch, ideally with operators to manage replication and failover. Updates are ordered, and it needs more care, since storage and quorum matter. A DaemonSet ensures one pod runs on every node, or on a selected subset, and adds pods automatically as nodes join. Use it for node-level agents: log collectors, monitoring exporters, CNI and storage plugins, and security agents. Related controllers: Job and CronJob for run-to-completion tasks. The choice depends on whether the workload needs identity and stable storage, is a singleton per node, or can be freely replicated. Don't use StatefulSet unless you need its guarantees, as it's operationally heavier.

Likely follow-ups

- Why does a StatefulSet need a headless Service?
- Give an example of a DaemonSet workload.

Deployment: stateless apps, any replica is identical, rolling update support

StatefulSet: stateful apps, stable network identity, ordered deployment, per-pod PVCs

DaemonSet: one pod per node (or selected nodes), node-level agents

Job: run-to-completion tasks, batch processing

CronJob: scheduled Jobs (cron-like scheduling)

`kubectl get deploy,sts,ds -A # see all workload types`

Deployment: web servers, APIs, workers — stateless, any pod can handle any request, pods are interchangeable. StatefulSet: databases (PostgreSQL, MySQL, Cassandra), message queues (Kafka, RabbitMQ), any app needing stable identity. StatefulSet pods are named predictably (pod-0, pod-1), deployed in order, and each gets its own PVC. DaemonSet: log shippers (Fluentbit), monitoring agents (node exporter, Datadog agent), CNI plugins, security agents — one per node ensures every node is covered. CronJob for scheduled tasks (backups, report generation, cache warming).

StatefulSet pod deletion: Deleting a StatefulSet pod does NOT delete its PVC. This is intentional — the PVC contains data. When the pod is rescheduled, it reattaches to the same PVC and recovers its data. Never delete PVCs without intentionally decommissioning the stateful workload. To wipe a StatefulSet cleanly: delete the StatefulSet, then manually delete the PVCs.

Official docs

- [Kubernetes: StatefulSets](#)
- [Kubernetes: DaemonSet](#)
- [Kubernetes: Workloads](#)

Q32 How does Kubernetes handle container restarts — what is the restart policy and backoff?

Orchestration

Spoken answer · ~89 s

Kubernetes restarts containers according to the pod's restartPolicy, which applies to all containers in the pod and is handled by the kubelet on the node. Always, the default for Deployments, restarts a container whenever it exits, whatever the exit code. OnFailure restarts only on non-zero exit, and Never doesn't restart; the latter two are used with Jobs and one-off pods. Restarts happen in place, within the same pod and node, and the pod isn't rescheduled. If a container keeps failing, the kubelet applies an exponential backoff delay, starting at ten seconds and doubling, 20, 40, and so on up to a cap of five minutes, and the backoff resets after the container runs successfully for ten minutes. During backoff the pod shows CrashLoopBackOff status. That is a symptom, not a cause: to investigate, use kubectl logs with --previous to see the crashed instance's output, kubectl describe pod for the last state, exit code, reason like OOMKilled or Error, and events; check liveness probe failures, since the kubelet also kills containers failing liveness probes; check config, secrets, image commands, resource limits, and dependencies. The restart count appears in kubectl get pods. A failing node will lead to eviction and rescheduling by the control plane, which is separate from container restarts. Alert on high restart counts.

Likely follow-ups

- What does CrashLoopBackOff actually mean?
- How do you see why the previous container died?

```
restartPolicy: Always    # default for Deployments/StatefulSets – always restart
```

```
restartPolicy: OnFailure # for Jobs – only restart on non-zero exit
```

```
restartPolicy: Never     # for Jobs/debugging – do not restart
```

```
# Exponential backoff: 10s → 20s → 40s → 80s → 160s → 300s (max)
```

```
kubectl describe pod <pod> | grep -E "Restart|Back-off"
```

```
kubectl get pod <pod> -o wide # RESTARTS column
```

```
# Reset backoff: pod stays running for 10 minutes resets backoff counter
```

Kubernetes restarts containers (not pods) when they fail. The restart policy applies at the pod level. Exponential backoff prevents rapid restart storms — after each crash, the wait time doubles up to 5 minutes. CrashLoopBackOff status means the container keeps crashing and the backoff is active. The backoff timer resets if the container runs successfully for 10 minutes. A container stuck in CrashLoopBackOff with a long backoff delay can be forced to restart faster by deleting the pod (the pod will be recreated by the controller and the backoff resets).

CrashLoopBackOff investigation: `kubectl logs --previous <pod> -c <container>` shows logs from the previous (crashed) instance — critical because the current container may not have started yet. Always use `--previous` when the current container is in CrashLoopBackOff, otherwise `kubectl logs` shows nothing from the waiting container.

Official docs

- [Kubernetes: Pod lifecycle & termination](#)

Spoken answer · ~76 s

Init containers are special containers that run to completion, one at a time in sequence, before the app containers in a pod start. If an init container fails, the kubelet retries it according to the restart policy, and the app containers don't start until all succeed. They share the pod's volumes and network, but have their own images, which lets you keep setup tooling out of the application image. Use cases: waiting for dependencies to be ready, such as a database or another service, though applications should still handle retries; running database migrations or schema checks before the app starts, with care about concurrency across replicas; fetching configuration, secrets, or files from remote sources and writing them to a shared emptyDir volume; setting file permissions and ownership on volumes, for instance fixing permissions for non-root containers; registering the pod with a service or generating certificates; and cloning a repository or downloading models. Advantages: separation of concerns, since the main image stays small and doesn't need extra tools or elevated privileges; ordering guarantees; and failures are visible as a pod stuck in Init status.

Likely follow-ups

- Why should init containers be idempotent?
- When is a sidecar better than an init container?

```
initContainers:
```

```
- name: wait-for-db
```

```
  image: busybox
```

```
  command: ['sh', '-c', 'until nc -z db-service 5432; do sleep 2; done']
```

```
- name: run-migrations
```

```
  image: myapp:latest
```

```
  command: ['python', 'manage.py', 'migrate']
```

```
  env: [{name: DATABASE_URL, valueFrom: {secretKeyRef: ...}}]
```

```
# Main container only starts after ALL init containers complete
```

```
kubectl get pod <pod> # shows: Init:0/2, Init:1/2, then Running
```

Init containers run sequentially to completion before any app containers start. Use for: waiting for dependencies (database, message queue — prevents race conditions), running database migrations (before multiple app instances start simultaneously), seeding ConfigMaps/secrets into shared volumes, or setting up

kernel parameters requiring privileges the main container doesn't have. Init containers can have different images and security contexts from the main container — you can run privileged setup steps without making the main container privileged.

Migration pattern: Run migrations in an init container before the app starts. This ensures: migrations run once (not N times for N replicas), the app doesn't start until migrations complete, and migration failures prevent the deployment from proceeding (pod stays in Init state, alerts fire). Much safer than running migrations in the app startup code.



Official docs

- [Kubernetes: Init containers](#)

Q34 What are sidecar containers — what are the common patterns?

Orchestration

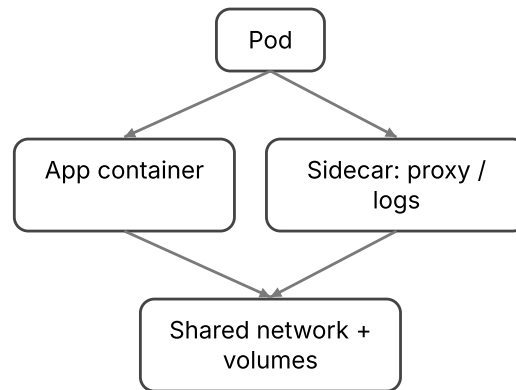
Spoken answer · ~88 s

A sidecar is a helper container that runs alongside the main application container in the same pod, sharing its network namespace, volumes, and lifecycle, to add capabilities without changing the application code. Common patterns: a service mesh proxy, like Envoy in Istio, intercepting traffic to provide mTLS, retries, routing, and telemetry; log shipping, where a sidecar such as Fluent Bit reads log files from a shared volume and forwards them; metrics exporters that translate application metrics into Prometheus format; configuration or secret reloaders that watch for changes and update shared files or signal the app; authentication proxies like OAuth2 Proxy placed in front of an app; the Cloud SQL Auth Proxy providing secure database connections; and the ambassador and adapter patterns, where a sidecar proxies outbound connections or normalises output formats. Benefits: language-independent reuse, separation of concerns, and independent versioning. Costs: extra resource usage per pod, more complexity, and startup and shutdown ordering problems, for example the app starting before the proxy is ready, or a Job never completing because the sidecar keeps running. Kubernetes native sidecars, from version 1.28, are init containers with restartPolicy Always, starting before the app and terminating after it, fixing those issues. Evaluate whether a per-node DaemonSet or ambient mesh model would use fewer resources.

Likely follow-ups

- What problems do native sidecars solve?
- When would you use a DaemonSet instead of a sidecar?

🚧 Sidecar pattern



```
# Log shipping sidecar:
```

```
containers:
```

```
- name: app
```

```
  image: myapp
```

```
  volumeMounts: [{name: logs, mountPath: /var/log/app}]
```

```
- name: log-shipper
```

```
  image: fluent/fluent-bit
```

```
  volumeMounts: [{name: logs, mountPath: /var/log/app}]
```

```
volumes:
```

```
- name: logs
```

```
  emptyDir: {}
```

```
# Service mesh sidecar (Istio Envoy – auto-injected):
```

```
kubectl label namespace default istio-injection=enabled
```

```
kubectl get pod <pod> -o jsonpath='{.spec.containers[*].name}' # shows istio-proxy
```

Common sidecar patterns: log shipping (app writes to shared volume, sidecar ships to ELK/GCS), proxy (Envoy/nginx in front of app for TLS termination, circuit breaking), secret sync (Vault agent fetches and refreshes secrets into a shared volume), adapter (transforms app output format for a monitoring system), and service mesh (transparent mTLS + observability without code changes). Kubernetes 1.29+ introduced native sidecar support — sidecars can now be declared as init containers with `restartPolicy: Always`, ensuring they start before and outlive the main container.

Native sidecar (K8s 1.29+): Traditional sidecars have ordering problems: the main container may exit before the sidecar (Jobs), or sidecars start in undefined order. Native sidecars (initContainers with restartPolicy: Always) start before main containers and stay running even after main exits — perfect for log shippers in Jobs.

 Official docs

- [Kubernetes: Sidecar containers](#)

Q35 How do you configure liveness and readiness probes — what happens when each fails?

Orchestration

 Spoken answer · ~85 s

Probes let the kubelet check container health. A liveness probe asks whether the container is still functioning; if it fails the configured number of times, the kubelet kills the container and restarts it according to the restart policy. Use it to recover from deadlocks and unrecoverable states, and keep it simple and independent of external dependencies. A readiness probe asks whether the container can serve traffic; when it fails, the pod is removed from Service endpoints, so no new requests are routed to it, but it isn't restarted; when it passes again, it rejoins. It also gates rolling updates. A startup probe protects slow-starting applications: until it succeeds, the liveness and readiness probes are disabled, preventing premature restarts. Each probe can use HTTP GET, TCP, exec, or gRPC, with parameters initialDelaySeconds, periodSeconds, timeoutSeconds, successThreshold, and failureThreshold. Pitfalls: a liveness probe that checks a database causes mass restarts during a database outage; too aggressive timeouts under load cause restart storms; identical liveness and readiness endpoints remove the benefit of having both; missing readiness probes send traffic to unready pods. Design the readiness endpoint to reflect the ability to serve, tune the thresholds from observed startup and latency, and watch probe failure events in kubectl describe.

Likely follow-ups

- Why shouldn't liveness depend on downstream services?
- What happens to traffic when readiness fails?

```

livenessProbe:          # restart container if this fails

  httpGet:

    path: /healthz

    port: 8080

    initialDelaySeconds: 30 # wait before first check

    periodSeconds: 10

    failureThreshold: 3     # 3 consecutive failures = restart

  readinessProbe:        # remove from Service endpoints if this fails

    httpGet:

      path: /ready

      port: 8080

      initialDelaySeconds: 5

      periodSeconds: 5

  startupProbe:          # disable liveness/readiness during slow startup

    httpGet: {path: /healthz, port: 8080}

    failureThreshold: 30   # allow 5 minutes for startup (30 * 10s)

    periodSeconds: 10

```

Liveness probe failure: kubelet restarts the container. Use for deadlock detection — if the app is alive but stuck, liveness restarts it. Do not check upstream dependencies in liveness — if the DB is down, you don't want all pods restarting simultaneously. Readiness probe failure: pod is removed from Service endpoints — traffic stops flowing to it. Use for: app still warming up (cache loading), temporarily overloaded, or dependency unavailable. Startup probe: disables liveness and readiness during slow application startup, preventing premature restarts of slow-starting JVM or Python applications.

Liveness probe misconfiguration: A liveness probe that checks an upstream database will cause cascading restarts when the DB has issues — all pods restart simultaneously, making recovery harder. Liveness should only check the application itself (can it process requests?). Readiness should check dependencies (DB reachable? Cache warm?).

Official docs

- [Kubernetes: Liveness, readiness and startup probes](#)

Q36 What is a container registry — how do you manage images in production?

CI/CD & registry

Spoken answer · ~85 s

A container registry is a service that stores and distributes container images and other OCI artefacts. Examples: Docker Hub, GitHub Container Registry, Amazon ECR, Google Artifact Registry, Azure Container Registry, and self-hosted options like Harbor. In production I manage registries with several concerns. Access control: private repositories, least-privilege IAM, separate push rights for CI and pull rights for clusters, using workload identity rather than long-lived credentials. Image hygiene: immutable tags or deploy by digest, a clear tagging scheme using commit SHA and semantic versions, and never relying on latest. Security: vulnerability scanning on push and continuous rescanning, image signing and verification, and blocking deployment of unscanned or unsigned images. Lifecycle: retention and cleanup policies to delete old and untagged images, controlling storage cost, while keeping images for currently deployed and rollback versions. Availability and performance: place registries in the same region as clusters, use replication across regions for resilience and speed, and pull-through caches or mirrors to avoid public registry rate limits and outages. Governance: restrict allowed registries through admission policies, host trusted base images internally, and keep audit logs. Reliability: avoid single points of failure in the deployment path, as an unreachable registry blocks scaling and rescheduling. Monitor storage, pull errors, and costs.

Likely follow-ups

- Why use a pull-through cache?
- What retention policy would you set?

```
# GCP Artifact Registry:
```

```
gcloud artifacts repositories create myrepo --repository-format=docker --location=us-central1
```

```
gcloud auth configure-docker us-central1-docker.pkg.dev
```

```
docker tag myapp us-central1-docker.pkg.dev/myproject/myrepo/myapp:v1.0.0
```

```
docker push us-central1-docker.pkg.dev/myproject/myrepo/myapp:v1.0.0
```

```
# Image management:
```

```
gcloud artifacts docker images list us-central1-docker.pkg.dev/myproject/myrepo
```

```
gcloud artifacts docker images delete <image-url> # cleanup old images
```

```
# Lifecycle policies (auto-delete old images):
```

```
gcloud artifacts repositories set-cleanup-policies myrepo \
```

```
--location=us-central1 --policy=policy.json
```

Registry best practices: use private registries (never rely on Docker Hub rate limits in production), co-locate registry with compute (Artifact Registry in same region as GKE — faster pulls, no egress cost), implement cleanup policies (delete untagged images, keep last N versions per tag), scan images on push (Artifact Registry auto-scans), and use immutable tags (SHA digest pins). Artifact Registry replaces GCR and supports multiple formats (Docker, Helm, npm, pip, Maven).

Image pull performance: Image pull is often the bottleneck in pod startup time, especially for cold starts in new nodes. Solutions: use image streaming (GKE feature — starts container before image fully pulled), keep images small (Alpine/distroless), use regional registries co-located with the cluster, and pre-pull images via DaemonSet or warm-up techniques.

Official docs

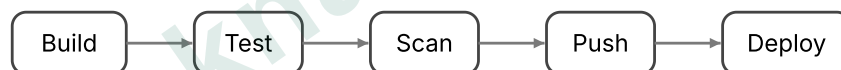
- [Artifact Registry](#)
- [Docker documentation](#)

Spoken answer · ~83 s

A pipeline for containerised applications follows build once, promote the same artefact. On a pull request: lint, run unit tests, build the image to confirm it builds, scan the image and the Dockerfile, and run integration tests, perhaps with Compose or ephemeral environments. On merge: build the image with BuildKit caching, tag with the commit SHA, run the security scan with a gate on critical findings, generate an SBOM, push to the registry using short-lived credentials via OIDC, and sign it with Cosign. Then deploy: update the deployment configuration to reference the image digest, either through a GitOps repository that Argo CD or Flux reconciles, or by running Helm or kubectl from the pipeline. Deploy to a staging environment first, run smoke and end-to-end tests, then promote the same digest to production, with approval gates where required, using a rolling, canary, or blue-green strategy with automated rollback on failed health or metrics. Add admission policies so only signed, scanned images from approved registries run. Keep configuration separate from the image, manage secrets externally, and handle database migrations as a controlled step. Track DORA metrics, cache dependencies for speed, and notify on failures. Everything is defined as code, versioned, and reviewed.

Likely follow-ups

- Why promote the same digest across environments?
- Where does GitOps fit in this pipeline?

Container pipeline**# Stage 1: Continuous Integration**

git push → trigger CI

lint + unit tests (fast feedback, 2-3 min)

docker build (with layer cache)

trivy scan (fail on CRITICAL CVEs)

docker push to registry (tag: branch-sha)

Stage 2: Staging deployment

integration tests against staging

smoke tests (key user flows)

performance baseline check

Stage 3: Production deployment

manual approval gate (or auto for low-risk)

canary deploy (10% traffic)

monitor error rate + latency (5-15 min)

promote to 100% or rollback

tag image as :stable

Pipeline design principles: fast feedback (lint/test before building), immutable artifacts (build once, deploy everywhere — same image to staging and production), explicit version tags (branch-sha for staging, semver for production), separation of CI and CD (CI produces the artifact, CD deploys it — different tools/permissions), and GitOps for CD (ArgoCD watching a manifests repo updated by CI). Never rebuild the image for production — deploy the exact image that passed staging tests.

GitOps CD pattern: CI pipeline (Cloud Build/GitHub Actions) builds and pushes the image, then updates the image tag in the Kubernetes manifests Git repo. ArgoCD detects the commit and deploys. Rollback = revert the manifest commit. Audit trail = Git history. This separates application code (app repo) from deployment config (manifests repo) cleanly.

 **Start here (official docs)**

- [Docker documentation](#)

Q38 **How do you handle environment-specific configuration in containerized apps?**

CI/CD & registry

 Spoken answer · ~83 s

The principle is one immutable image for all environments, with configuration injected at runtime, following the twelve-factor approach. Options in Kubernetes: environment variables sourced from ConfigMaps for non-sensitive settings and from Secrets or an external secrets manager for sensitive ones; configuration files mounted from ConfigMaps as volumes; and command-line arguments. Per-environment values are managed through Helm values files, such as values-prod.yaml, Kustomize overlays, or GitOps repository folders for each environment, so that the differences between dev, staging, and production are explicit, reviewed, and version-controlled. Keep the differences minimal and avoid environment-specific code branches. Use sensible defaults in the application, validate required configuration at startup and fail fast with clear errors, and document each setting. Feature flags can control behaviour changes without redeploying. For dynamic configuration, consider a configuration service, or reloading mounted files, noting that environment variables require a restart and that ConfigMap volume updates propagate eventually, not instantly. Keep secrets separate with appropriate access control and rotation. Avoid baking environment details into images, and avoid using different images per environment since you'd no longer test what you ship. Include configuration in change management: review, test in lower environments, and use drift detection to ensure clusters match the declared state.

Likely follow-ups

- Why avoid building a separate image per environment?
- How do ConfigMap changes reach running pods?

```
# Pattern 1: Environment variables (twelve-factor app)
```

```
docker run -e DATABASE_URL=postgres://... myapp
```

```
kubectl create configmap app-config --from-literal=LOG_LEVEL=info
```

```
envFrom: [{configMapRef: {name: app-config}}]
```

```
# Pattern 2: ConfigMap mounted as file
```

```
kubectl create configmap nginx-config --from-file=nginx.conf
```

```
volumeMounts: [{name: config, mountPath: /etc/nginx/nginx.conf, subPath: nginx.conf}]
```

```
# Pattern 3: Helm values per environment
```

```
helm install myapp ./chart -f values-prod.yaml
```

```
helm upgrade myapp ./chart -f values-staging.yaml
```

```
# Pattern 4: Kustomize overlays
```

```
kustomize build overlays/production | kubectl apply -f -
```

Config management strategies: environment variables (twelve-factor — simple, universal, visible in env), ConfigMaps (Kubernetes-native, version-controlled, hot-reloadable with appropriate tooling), Helm values (template-based, env-specific values files), Kustomize overlays (patch-based, no templating, pure YAML). Never bake environment-specific config into the image — the same image should run in dev, staging, and prod with different config injected at runtime. Config changes should be deployable without rebuilding the image.

ConfigMap hot reload: Kubernetes updates mounted ConfigMap files when the ConfigMap changes (with ~1 minute delay). But env vars from ConfigMaps are NOT updated — the pod must restart. If your app watches config files for changes, you get hot reload free. Otherwise, use tools like Reloader to watch ConfigMaps and trigger rolling restarts automatically.

Official docs

- [Kubernetes: ConfigMaps](#)
- [Kustomize reference](#)

Spoken answer · ~85 s

Helm is the package manager for Kubernetes. A chart is a package of templated Kubernetes manifests plus default values and metadata; installing a chart with specific values creates a release, tracked with revisions, which enables upgrades and rollbacks. Templates use Go templating with values from values.yaml and overrides, which lets one chart serve many environments. Managing charts in production: store charts in version control with a clear structure and semantic versioning of both the chart and the app version; publish them to an OCI registry or chart repository; keep per-environment values files in Git; lint with helm lint and validate rendered output with kubeconform; test with helm template and chart tests; and review diffs before upgrades using the helm diff plugin. Deploy through a GitOps tool like Argo CD or Flux, rather than running helm manually, so releases are declarative and auditable. Pin chart versions for third-party charts and mirror them internally, review upstream changes, and avoid latest. Keep sensitive values out of Git by using external secrets or SOPS. Use library charts or a standard application chart to reduce duplication across services. Be aware of caveats: CRD handling is limited, templates can become complicated, and helm upgrade needs care with immutable fields.

Likely follow-ups

- Why use helm diff before upgrades?
- How do you avoid putting secrets in values files?

```
helm install myapp ./mychart --namespace prod -f values-prod.yaml
```

```
helm upgrade myapp ./mychart --namespace prod -f values-prod.yaml
```

```
helm rollback myapp 1 --namespace prod # rollback to revision 1
```

```
helm history myapp --namespace prod # release history
```

```
helm diff upgrade myapp ./mychart -f values-prod.yaml # preview changes
```

```
helm lint ./mychart # validate chart
```

```
helm template ./mychart -f values-prod.yaml # render without deploying
```

```
helm test myapp --namespace prod # run chart tests
```

```
# List all releases:
```

```
helm list -A # all namespaces
```

Helm is the Kubernetes package manager — charts are templated collections of Kubernetes manifests. Helm manages releases with history, enabling rollback. Production practices: pin chart versions (Chart.yaml dependencies with exact versions), separate values files per environment (values-prod.yaml, values-staging.yaml), use helm diff before upgrade (plugin), never use helm install in CI — always helm upgrade --install (idempotent), and store chart state (helm history) for audit. For GitOps: push rendered Helm templates to the manifests repo, or use ArgoCD's native Helm support.

Helm vs Kustomize: Helm uses Go templates (powerful but complex, easy to create invalid YAML). Kustomize uses strategic merge patches (simpler, pure YAML, no templating). Both are valid. Helm is better for distributing reusable charts (third-party apps). Kustomize is better for organization-specific overlays (customize vendor charts without forking). Many teams use both: Kustomize to patch Helm chart outputs.

Official docs

- [Helm: Chart best practices](#)
- [Helm documentation](#)

Q40 How do you collect and manage logs from containers — what are the patterns?

Observability

Spoken answer · ~87 s

In containers, applications should write logs to stdout and stderr, and the platform handles collection, since container filesystems are ephemeral. The container runtime captures the streams and stores them in files on the node, rotated by size; kubectl logs reads from there. For aggregation, the most common pattern is a node-level agent, deployed as a DaemonSet, such as Fluent Bit, Vector, Promtail, or the OpenTelemetry Collector, that tails the log files on each node, enriches them with Kubernetes metadata like pod, namespace, and labels, and forwards them to a backend such as Loki, Elasticsearch, or a cloud logging service. A second pattern is a sidecar that reads application log files from a shared volume and streams them, used when an application can only write to files, or needs special parsing. A third is the application pushing directly to a logging endpoint, which couples it to the backend and is usually avoided. Good practices: structured JSON logs with consistent fields, trace IDs for correlation, appropriate levels, no sensitive data, and rate-limiting noisy logs. Manage cost and volume with sampling, filtering at the agent, and retention tiers. Set log rotation limits on nodes so disks don't fill, buffer and retry in agents to avoid loss, and monitor the pipeline itself.

Likely follow-ups

- Why log to stdout rather than to files?
- When is a logging sidecar justified?

```
# 12-factor: write to stdout/stderr, not log files
```

```
docker logs <container> -f --tail=100 # Docker log driver reads stdout
```

```
kubectl logs <pod> -c <container> -f # K8s reads from container log file
```

```
kubectl logs <pod> --previous # previous container instance
```

```
kubectl logs -l app=myapp --all-containers # all pods matching label
```

```
# Log aggregation stack options:
```

```
Fluent Bit (DaemonSet) → Cloud Logging / Elasticsearch / Loki
```

```
Fluentd → Elasticsearch → Kibana (EFK stack)
```

```
Promtail → Loki → Grafana (lightweight, label-based)
```

```
# GKE: Cloud Logging built-in (no setup required)
```

```
gcloud logging read "resource.type=k8s_container" --limit=50
```

Container logging principle: containers write to stdout/stderr. The container runtime captures these streams to files on the node (/var/log/containers/). A log aggregation agent (Fluent Bit DaemonSet) reads these files and ships to a central destination. This means: no log files inside containers, no volume mounts for logging, and no log management responsibility for the application. GKE automatically ships container logs to Cloud Logging — zero configuration needed.

Log rotation: Container log files on nodes grow indefinitely without rotation. Configure kubelet's --container-log-max-size (default 10Mi) and --container-log-max-files (default 5). Without this, a verbose container can fill node disk in hours. In GKE, this is managed automatically.



Official docs

- [Cloud Logging](#)

Q41 How do you implement metrics collection for containers — what are the key metrics to monitor?

Observability

 Spoken answer · ~81 s

Container metrics come from several layers. Resource usage comes from cAdvisor, built into the kubelet: CPU usage, memory working set, network and disk I/O per container. Kubernetes state comes from kube-state-metrics: replica counts, pod phases, restarts, and deployment status. Node health comes from node-exporter. Application metrics are exposed by the app itself on a /metrics endpoint using Prometheus client libraries or OpenTelemetry. Collect with Prometheus, using ServiceMonitors and PodMonitors, or a managed equivalent, with Grafana for dashboards and Alertmanager for alerts. Key metrics: CPU usage against requests and limits, and especially CPU throttling; memory working set against limits, with OOM kill counts; restart counts and CrashLoopBackOff; pod readiness and pending pods; network errors and traffic; disk usage, for volumes and ephemeral storage; and for the application, the golden signals of latency, traffic, errors, and saturation, such as queue depth and connection pool usage. Also track deployment status, HPA scaling behaviour, and image pull failures. Alert on symptoms and user impact, and on leading indicators like memory approaching limits or throttling. Control cardinality by avoiding unbounded labels like request IDs. Right-size resources using the historical data, and correlate metrics with logs and traces for diagnosis.

Likely follow-ups

- Which metric reveals CPU throttling?
- Why watch working set rather than total memory?

Key container metrics (Prometheus):

container_cpu_usage_seconds_total # CPU used

container_cpu_cfs_throttled_seconds_total # CPU throttled (limit hit)

container_memory_working_set_bytes # actual memory usage

container_memory_limit_bytes # configured limit

container_oom_events_total # OOM kill count

container_restarts_total # restart count

kube_pod_status_phase # pod phase

Stack:

kube-prometheus-stack (Helm chart) # Prometheus + Grafana + Alertmanager

kubectl top pods -A --sort-by=memory # quick resource check

```
kubectl top nodes
```

```
# node-level resource
```

Key alerts: `container_cpu_cfs_throttled_seconds_total` > 25% of periods (CPU limit too low — container is being slowed down), `container_memory_working_set_bytes` / `container_memory_limit_bytes` > 80% (approaching OOM kill), `container_restarts_total` increasing (CrashLoopBackOff starting), `kube_pod_status_phase{phase="Failed"}` (failed pods), and `container_oom_events_total` (OOM kills happening). Use `working_set_bytes` not `usage_bytes` for memory — `working_set` excludes reclaimable cache and reflects actual memory pressure.

Resource right-sizing: VPA (Vertical Pod Autoscaler) in recommendation mode observes actual usage and suggests request/limit values. Run VPA for 1-2 weeks to collect data, then apply recommendations. Container resource over-provisioning wastes money; under-provisioning causes throttling and OOM kills. VPA recommendations are the data-driven middle ground.

Official docs

- [Kubernetes Autoscaler: Vertical Pod Autoscaler](#)
- [Linux kernel docs: /proc/sys/vm](#)

Q42 How do you debug a container that is running but not responding to requests?

Observability

Spoken answer · ~78 s

A container that's running but not responding is a different problem from a crash: the process is alive but something blocks requests. My steps: check the basics from outside: are the pod and container ready, restarts, and recent events, and does the Service have endpoints? Test the path in layers: from inside the cluster, curl the pod IP directly, then the Service, then the ingress, to find where it breaks. Check the application: logs for errors, hanging requests, or stack traces; the process state; thread dumps or goroutine dumps to find deadlocks; connection pool exhaustion; and blocked calls to dependencies like a database or downstream service with no timeout. Check resources: CPU throttling, memory pressure and GC pauses, file descriptor limits, and disk full. Check networking: whether the app listens on the expected port and address, network policies, DNS, and the liveness and readiness probe results, since a failing readiness removes the pod from traffic. Use debug tools: `kubectl exec`, an ephemeral debug container with `kubectl debug`, `ss`, `netstat`, `strace`, `tcpdump`, and profilers. Check recent changes such as deployments or configuration, and compare with healthy replicas.

Likely follow-ups

- How do you capture evidence before restarting?
- What does a thread dump reveal?

```
# Step 1: Check if process is running and port is open
```

```
docker exec <container> ps aux          # processes running?
```

```
docker exec <container> ss -tlnp        # port listening?
```

```
docker exec <container> curl -v localhost:8080/health
```

```
# Step 2: Check logs for errors
```

```
docker logs <container> --tail=100 --since 5m
```

```
# Step 3: Check resource pressure
```

```
docker stats <container> --no-stream
```

```
# Step 4: Inspect from host if container lacks tools
```

```
PID=$(docker inspect -f "{{.State.Pid}}" <container>)
```

```
strace -p $PID -e trace=network        # syscall trace
```

```
nsenter -t $PID -n -- ss -tlnp        # network from host tools
```

```
# Kubernetes:
```

```
kubectl exec -it <pod> -- curl localhost:8080/health
```

```
kubectl describe pod <pod> | grep -A5 "Readiness|Liveness"
```

Running but not responding means: process is alive (cgroup has it) but not processing requests. Causes: application deadlock (goroutine/thread blocked), port listening on wrong interface (0.0.0.0 vs 127.0.0.1), all worker threads occupied (thread pool exhaustion), dependency unavailable causing every request to block, or CPU throttling causing extreme slowness. Use strace to see which syscall the process is blocking on — read() waiting on network = waiting for upstream. futex() waiting = deadlock on mutex.

Thread pool exhaustion pattern: Service receives traffic, logs show requests arriving, but all requests timeout. The process is alive and port is open. Check connection count: if open connections equal worker thread pool size, all threads are busy (likely blocked on slow DB queries). Fix: increase thread pool, fix slow queries, add circuit breaker.

Official docs

- [Docker documentation](#)

Q43 What is distributed tracing — how do you implement it for containerized microservices?

Observability

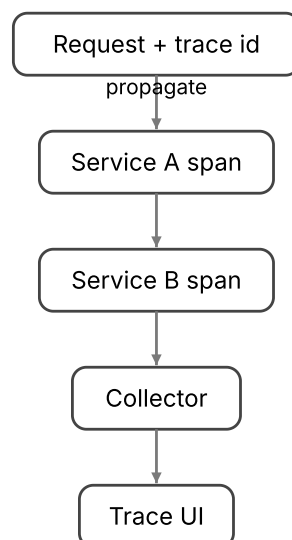
Spoken answer · ~87 s

Distributed tracing follows one request through many services by assigning it a trace ID and recording each unit of work as a span with timing and metadata. In a containerised microservices system, implement it by instrumenting each service with OpenTelemetry SDKs or auto-instrumentation agents; in Kubernetes, the OpenTelemetry Operator can inject them via pod annotations. Services propagate context through headers, the W3C traceparent standard, on every outgoing HTTP or gRPC call and through message headers for queues, so spans link into one trace. A service mesh sidecar can generate spans for network hops but the application must still propagate headers. Spans are exported over OTLP to a collector, deployed as a DaemonSet agent and a gateway, which batches, enriches with Kubernetes metadata, applies sampling, and forwards to a backend like Tempo, Jaeger, or a cloud trace service. Sampling matters at scale: head-based for simplicity, tail-based to keep errors and slow requests. Add useful attributes, such as customer tier, endpoint, and version, but avoid sensitive data. Connect traces to logs by writing the trace ID in each log line, and to metrics using exemplars. Use traces to find latency bottlenecks, failing dependencies, and unexpected call patterns. Roll out incrementally, starting with the critical request paths, and monitor tracing overhead.

Likely follow-ups

- What breaks a trace across services?
- How do exemplars connect metrics to traces?

🚩 Distributed tracing



OpenTelemetry (OTEL) – the standard:

```
pip install opentelemetry-sdk opentelemetry-exporter-otlp
```

Configure OTEL exporter (via env vars):

```
OTEL_EXPORTER_OTLP_ENDPOINT=http://otel-collector:4317
```

```
OTEL_SERVICE_NAME=my-api-service
```

```
OTEL_TRACES_SAMPLER=parentbased_traceidratio
```

```
OTEL_TRACES_SAMPLER_ARG=0.1    # sample 10% of traces
```

```
# OTEL Collector DaemonSet or Deployment routes traces:
```

```
OTEL Collector → Cloud Trace / Jaeger / Tempo / Zipkin
```

```
# Istio: automatic tracing without code changes
```

```
kubectl get meshconfig -n istio-system # tracing config
```

Distributed tracing follows a request through multiple services by propagating trace context (trace ID, span ID) in headers (W3C traceparent standard). Each service creates a span recording its work. The OTEL collector aggregates spans into traces and exports to a backend. Key use: identifying which service in a microservice call chain added latency. Without tracing, a 500ms response from an API could come from any of 10 services it calls. With tracing, you see: API=10ms, Auth=20ms, DB=470ms — instantly pinpoints the bottleneck.

Auto-instrumentation: OTEL provides language-specific auto-instrumentation agents that inject tracing without code changes. For Java: `-javaagent:opentelemetry-javaagent.jar`. For Python: `opentelemetry-instrument python app.py`. These capture HTTP, gRPC, database, and message queue calls automatically. Start with auto-instrumentation, then add custom spans for business logic.

Official docs

- [OpenTelemetry: Signals \(traces, metrics, logs\)](#)

Q44 How do you implement zero-downtime deployments — what are the prerequisites?

Deployment strategies

 Spoken answer · ~84 s

Zero-downtime deployment means users see no errors or interruption while a new version replaces the old. Prerequisites start with the application: run multiple replicas so some always serve; implement correct readiness probes so traffic only reaches ready pods; support graceful shutdown by handling SIGTERM, finishing in-flight requests, and stopping accepting new ones; and keep the termination grace period longer than the longest request. Pods in Kubernetes are removed from endpoints asynchronously, so add a short preStop sleep to let load balancers and proxies drain before the process stops. Use a rolling update with maxUnavailable of zero and sensible maxSurge, or blue-green or canary. Ensure compatibility: the old and new versions run side by side, so APIs and message formats must be backward compatible, and database changes must follow expand and contract. Configure PodDisruptionBudgets and spread replicas across nodes and zones. Ensure capacity and quota for surge pods, and fast image pulls through caching or pre-pulled images. Handle long-lived connections, WebSockets, and sessions by draining or reconnect logic. Add monitoring with automated rollback on error or latency regression, and feature flags to decouple release from deploy. Test the deployment process under load in staging, as many failures are only visible with real traffic.

Likely follow-ups

- Why add a preStop sleep?
- Why must old and new versions be compatible?

Prerequisites checklist:

1. Multiple replicas (minimum 2, recommend 3+ for true zero-downtime)
2. Readiness probe configured and working
3. Graceful shutdown: handle SIGTERM, drain in-flight requests
4. terminationGracePeriodSeconds > request timeout
5. Database migrations are backward-compatible
6. maxUnavailable: 0 in rolling update strategy
7. PodDisruptionBudget configured

Graceful shutdown implementation:

lifecycle:

preStop:

```
exec:
```

```
command: ["/bin/sh", "-c", "sleep 5"] # wait for LB to stop routing
```

```
terminationGracePeriodSeconds: 60
```

Zero-downtime requires: new pods pass readiness before old pods are terminated (maxUnavailable: 0), old pods drain in-flight requests before exit (SIGTERM handling + preStop hook), and the load balancer stops routing to terminating pods before they die. The preStop sleep is critical: when a pod is terminated, Kubernetes simultaneously sends SIGTERM AND removes it from Service endpoints — but the endpoint removal propagates asynchronously (1-2 seconds). Without preStop sleep, the pod receives and starts rejecting traffic for 1-2 seconds before the LB catches up.

The race condition: Pod termination and endpoint removal happen concurrently, not sequentially. Without preStop: SIGTERM → app starts shutting down → LB still sends requests for 1-2 seconds → 502 errors. With preStop sleep(5): SIGTERM → sleep 5s → endpoint removal propagates → app starts shutting down → no requests arrive → clean shutdown. This 5-second sleep eliminates deployment 502 errors.



Official docs

- [Kubernetes: Pod lifecycle & termination](#)
- [Linux man-pages: signal\(7\)](#)

Q45 How do you handle database schema migrations in a containerized deployment pipeline?

Deployment strategies

 Spoken answer · ~89 s

Schema migrations are risky in containerised pipelines because many replicas start concurrently and old and new application versions overlap during rollout. Principles: make changes backward compatible using expand and contract, so the old code works with the new schema; first add nullable columns or new tables, deploy the application, backfill, and remove old structures in a later release. Run migrations as a distinct, single-run step, not at startup of every replica, where racing instances can corrupt or deadlock; use a Kubernetes Job, a Helm pre-upgrade hook, an init container with a lock, or a pipeline stage, using a tool like Flyway, Liquibase, or the framework's migrator, which takes a database lock to ensure single execution. Version migration files in the repository, make them idempotent, and keep them small. Test against production-like data to measure duration and locking, using online techniques for big tables, such as batched backfills and concurrent index builds, with lock timeouts. Use credentials with the required privileges only for the migration job. Take a backup or snapshot before risky changes and define a forward-fix rather than depending on reverse migrations. Make the pipeline stop if the migration fails, and monitor database health during it. Separate deploying code from running destructive changes. Coordinate with rollbacks so the previous version still works.

Likely follow-ups

- Why not run migrations in every replica at startup?
- What does expand and contract mean?

```
# Option 1: Init container (recommended for most cases)
```

```
initContainers:
```

```
- name: run-migrations
```

```
  image: myapp:v2.0.0          # same image as app
```

```
  command: ["python", "manage.py", "migrate"]
```

```
# Option 2: Kubernetes Job (for complex migrations)
```

```
kubectl apply -f migration-job.yaml
```

```
kubectl wait --for=condition=complete job/db-migration --timeout=300s
```

```
kubectl apply -f deployment.yaml # only after migration job succeeds
```

```
# Option 3: Argo Workflows / Tekton pipeline step
```

```
# Expand-contract pattern for zero-downtime schema changes:
```

Deploy v2 schema (additive only: new columns nullable, new tables)

Deploy v2 app (writes to both old and new columns)

Backfill new columns

Deploy v3 app (reads from new columns only)

Deploy v3 schema (remove old columns)

Migration strategy depends on risk: small migrations (adding nullable column, new index) can run in init containers safely. Large migrations (millions of rows backfill, table restructure) should run as a separate Job with monitoring, decoupled from the app deployment. The expand-contract pattern is the gold standard for zero-downtime migrations: never make breaking schema changes in a single deployment. Each schema change must be compatible with both the current and new application version simultaneously.

Migration tools: Flyway and Liquibase for Java, Alembic for Python, golang-migrate for Go. All provide migration versioning, applied migration tracking, and rollback support. Always test migrations on a copy of production data in staging before running in production.

Official docs

- [AWS Builders' Library: Automating safe, hands-off deployments](#)
- [Martin Fowler: Continuous Delivery](#)
- [gh-ost: GitHub's online schema migration](#)

Q46 What is a distroless container image — when and how do you debug them?

Container runtime

🔊 Spoken answer · ~85 s

A distroless image contains only the application and its runtime dependencies, without a package manager, shell, or common utilities. Google's distroless images exist for Java, Python, Node, Go, and static binaries, and Chainguard images follow a similar philosophy. Benefits: a much smaller attack surface, because attackers can't use shells or tools after a compromise; fewer packages mean far fewer CVEs to track; and smaller image size. The trade-off is debugging, since you can't exec into a shell. Approaches: in Kubernetes, use ephemeral debug containers with `kubectl debug`, attaching a tool-rich image such as `busybox` or `netshoot` that shares the target container's process namespace and network, letting you inspect processes, files via `/proc/PID/root`, and network; Docker offers a similar approach by running a debug container with shared namespaces; distroless provides debug variants with a `busybox` shell, usable in non-production or temporarily. Other practices: rely on strong observability, logs, metrics, and traces, so you need to exec less; build a debug stage into a multi-stage Dockerfile and use `--target` for local troubleshooting; copy the pod and run it with a debug image using `kubectl debug --copy-to`. Combine with a read-only filesystem and non-root user. Document the debugging procedure for on-call, and test it before incidents.

Likely follow-ups

- How does `kubectl debug` work with distroless images?
- What do you lose by removing the shell?

🚩 Distroless



```
# Build with distroless runtime stage:
```

```
FROM gcr.io/distroless/java21-debian12 # JRE only, no shell
```

```
FROM gcr.io/distroless/python3-debian12
```

```
FROM gcr.io/distroless/nodejs20-debian12
```

```
FROM gcr.io/distroless/static-debian12 # for Go static binaries
```

```
# Debug a distroless container (no shell available):
```

```
kubectl debug -it <pod> --image=busybox --target=<container> # share process namespace
```

```
kubectl debug -it <pod> --copy-to=debug-pod --set-  
image=myapp=gcr.io/distroless/nodejs:debug
```

```
# Use :debug tag (includes busybox):
```

```
FROM gcr.io/distroless/nodejs20-debian12:debug # has shell for debugging
```

```
# nsenter from node:
```

```
PID=$(crictl inspect <container-id> | jq .info.pid)
```

```
nsenter -t $PID -m -u -i -n -p -- sh
```

Distroless images contain only the language runtime and your application — no shell (/bin/sh), no package manager, no coreutils. Benefits: dramatically smaller attack surface (nothing for an attacker to use post-exploit), smaller image size, passes most security scans. Debugging: `kubectl debug` with `--target` shares the container's process namespace, letting you run a busybox container alongside it. The `:debug` tag variant includes a busybox shell for temporary debugging — switch to `:debug` in staging, not production.

Image size comparison: Ubuntu base: ~70MB. Alpine base: ~7MB. Distroless static: ~2MB. For a Go binary, the final image is the binary size (~5-20MB) plus ~2MB distroless overhead = ~20MB total vs ~80MB with Alpine or ~90MB with Ubuntu. Smaller images = faster pulls, less storage, smaller attack surface — all wins.



Official docs

- [Kubernetes: Debug running Pods \(kubectl debug\)](#)
- [Distroless images](#)

🗣 Spoken answer · ~78 s

Kubernetes network policies control which pods can communicate, by selecting pods and defining allowed ingress and egress rules based on pod labels, namespace labels, and IP blocks plus ports. By default all pods can talk to each other, so policies implement segmentation and least privilege. They require a CNI plugin that enforces them, such as Calico, Cilium, or Antrea. Policies are additive allow rules: once a pod is selected by any policy for a direction, only traffic explicitly allowed is permitted for that direction. A sound approach starts with a default-deny policy per namespace for ingress and egress, then add specific allows: for example, the frontend may reach the API on port 8080, the API may reach the database on 5432, and everything may reach DNS on port 53 in kube-system; allow ingress from the ingress controller namespace and egress to required external services. Use namespace selectors to isolate tenants and environments. Practices: label workloads consistently, manage policies in Git, test with tools such as netshoot or Cilium's connectivity tests, and visualise flows with Hubble or the network policy editor before enforcing. Roll out gradually, observing traffic first.

Likely follow-ups

- Why start with default deny?
- Why must egress to DNS be allowed explicitly?

```
# Default deny all ingress in a namespace:
```

```
apiVersion: networking.k8s.io/v1
```

```
kind: NetworkPolicy
```

```
metadata: {name: default-deny-ingress}
```

```
spec:
```

```
  podSelector: {}    # applies to all pods in namespace
```

```
  policyTypes: [Ingress]
```

```
# Allow: only api pods can reach db pods on port 5432
```

```
spec:
```

```
  podSelector: {matchLabels: {app: db}}
```

```
  ingress:
```

```
    - from: [{podSelector: {matchLabels: {app: api}}}]
```

```
ports: [{protocol: TCP, port: 5432}]
```

```
kubectl get networkpolicy -A # list all policies
```

```
# Test connectivity:
```

```
kubectl run tmp --rm -i --image=busybox -- nc -zv db-service 5432
```

NetworkPolicies implement micro-segmentation inside the Kubernetes cluster — without them, any pod can reach any other pod. Start with a default-deny policy per namespace, then add explicit allow rules. Required allows: pods to CoreDNS (UDP/TCP 53), pods to kube-apiserver (if needed), monitoring agent to pods (scrape port), and inter-service communication. NetworkPolicies are enforced by the CNI plugin — Calico, Cilium, and Weave support them; Flannel (without Canal) does not.

Breaking DNS with NetworkPolicy: A namespace-wide ingress deny without an explicit allow for DNS (CoreDNS on port 53) breaks all DNS resolution in that namespace. Every pod suddenly cannot resolve any hostnames. Always add: allow egress to kube-system namespace pods (CoreDNS) on port 53 UDP/TCP when implementing default-deny egress policies.

Official docs

- [Kubernetes: Network Policies](#)

Q48 **How do you manage container configuration with ConfigMaps and how do updates propagate?**

Orchestration

 Spoken answer · ~87 s

A ConfigMap stores non-sensitive configuration as key-value pairs or whole files, decoupling configuration from the image. Pods consume it as environment variables, command arguments, or mounted files. Update behaviour differs by method. Environment variables from a ConfigMap are set when the container starts, so changes aren't reflected until the pod restarts. Volume-mounted ConfigMaps are updated by the kubelet eventually, typically within a minute or so, as it syncs through symlink swaps, but the application must watch or reload the files to use them; subPath mounts do not receive updates at all. To roll out changes reliably, a common pattern is making the ConfigMap immutable and versioned, or including a hash of its contents in the Deployment's pod template annotations, as Helm does with a checksum annotation, so that any change triggers a rolling restart through the normal deployment process. Tools like Reloader can restart workloads automatically when a ConfigMap changes. Immutable ConfigMaps improve safety and performance, since they prevent accidental edits and reduce API load. Practices: keep ConfigMaps in version control, validate them, separate configuration per environment, avoid putting secrets in them, watch the size limit of one megabyte, and test configuration changes in lower environments. Be careful that a bad ConfigMap change can break all pods at once.

Likely follow-ups

- Why don't subPath mounts update?
- What does a checksum annotation do?

```
# Create ConfigMap:
```

```
kubectl create configmap app-config --from-file=config.yaml --from-literal=LOG_LEVEL=debug
```

```
kubectl get configmap app-config -o yaml
```

```
# Mount as volume (auto-updates when CM changes, ~1min delay):
```

```
volumes: [{name: config, configMap: {name: app-config}}]
```

```
volumeMounts: [{name: config, mountPath: /etc/app}]
```

```
# Use as env vars (does NOT auto-update – pod restart required):
```

```
envFrom: [{configMapRef: {name: app-config}}]
```

```
# Trigger rolling restart when CM changes:
```

```
kubectl rollout restart deployment/myapp
```

```
# Auto-restart on CM change (Reloader controller):
```

```
kubectl annotate deployment myapp reloader.stakater.com/auto=true
```

ConfigMap update propagation: volume-mounted files update automatically (kubelet syncs every ~1 minute, then an additional sync cycle). Environment variables from ConfigMaps are set at pod creation and never update — the pod must restart. For zero-downtime config updates with env vars: use immutable ConfigMaps (create new CM with new name, update deployment to reference it — triggers rolling restart). Reloader controller watches ConfigMaps and automatically triggers rolling restarts when they change — the production-grade solution.

Immutable ConfigMaps: Set `immutable: true` on ConfigMaps that should never change. Benefits: no watch overhead on the API server (massive scale improvement in large clusters), prevents accidental updates. To change config: create a new ConfigMap (v2) and update the Deployment reference. This is the GitOps-compatible approach to config management.

Official docs

- [Kubernetes: ConfigMaps](#)

Q49 What is Docker Compose — how does it relate to production Kubernetes deployments?

Docker fundamentals

 Spoken answer · ~84 s

Docker Compose defines and runs multi-container applications on a single host from a YAML file: services, networks, volumes, environment variables, dependencies, and health checks, started with `docker compose up`. It's excellent for local development, integration tests in CI, demos, and small single-server deployments, because it's simple, readable, and fast to iterate on. Its relationship to Kubernetes: both describe multiple containers declaratively, but Compose targets one machine, while Kubernetes is a distributed orchestrator providing scheduling across nodes, self-healing, rolling updates, autoscaling, service discovery, load balancing, secrets and configuration management, and a rich ecosystem. Compose files don't translate one-to-one; tools such as Kompose can generate Kubernetes manifests as a starting point, but the result needs rework, for instance for probes, resource limits, ingress, and persistent storage. A useful workflow: use Compose for the inner development loop, and Kubernetes manifests, Helm, or Kustomize for deployment, accepting some difference between them; or use tools such as Skaffold, Tilt, or local clusters like kind and minikube to develop against Kubernetes directly, reducing environment drift. For production I'd choose Kubernetes or a managed platform like Cloud Run or ECS for availability and scale, and keep Compose limited to development and testing. Keep the same images across both.

Likely follow-ups

- When is Compose still a reasonable production choice?
- How would you reduce drift between Compose and Kubernetes?

```
docker compose up -d          # start all services in background
```

```
docker compose down          # stop and remove containers
```

```
docker compose logs -f api    # follow specific service logs
```

```
docker compose ps             # status of all services
```

```
docker compose exec api bash  # exec into service container
```

```
docker compose build          # rebuild images
```

```
# Convert Compose to K8s manifests:
```

```
kompose convert -f docker-compose.yaml -o k8s/
```

```
# Compose for local dev, K8s for production:
```

```
docker compose up            # dev: full stack locally
```

```
kubectl apply -k overlays/prod # prod: K8s with production config
```

Docker Compose is a tool for defining and running multi-container applications locally — single yml file defines all services, networks, and volumes. It excels for local development and CI (spin up entire stack for integration tests). It is NOT production-ready: no health-based load balancing, no rolling updates, no self-healing, no multi-node scaling. Use Compose for: local development, integration test environments, simple single-server deployments (small apps without K8s). Use Kubernetes for: production, multi-node, auto-scaling, self-healing. kompose converts Compose files to K8s manifests as a migration starting point.

Dev/prod parity: Use Docker Compose locally with the same images you deploy to production. Compose for dev, K8s for prod. This maintains the 12-factor dev/prod parity principle without requiring a full K8s cluster locally (minikube/kind are alternatives but heavier than Compose for quick iteration).

Official docs

- [Docker Compose](#)
- [Docker documentation](#)

Q50 Walk through a complete methodology for diagnosing a failed container deployment in production.

Deployment strategies

Spoken answer · ~90 s

My methodology for a failed container deployment in production is to stabilise, then diagnose layer by layer. First, check impact and decide on immediate mitigation: if users are affected and a recent deploy is suspected, roll back to the previous version, then investigate calmly in a safe environment. Next, identify the failure stage by looking at the rollout status and the pod state. ImagePullBackOff or ErrImagePull points to image name or tag, registry authentication, or network problems. Pending means scheduling problems: insufficient resources, quota, node selectors, taints, or unbound volumes; check events. CrashLoopBackOff or Error means the container starts then dies: read logs, including --previous, the exit code, and OOMKilled status. Running but not Ready means probe failures or the app not listening correctly. A stalled rollout may be the progress deadline or a PodDisruptionBudget. Use kubectl describe for events, logs, and exec or a debug container for deeper checks. Check what changed: image, configuration, secrets, environment variables, migrations, permissions, and dependencies, comparing with the last good revision. Also check cluster-wide causes like node issues, admission webhooks rejecting pods, and network policies. Then fix, test in staging, and redeploy gradually. Afterwards write a postmortem, and add checks to the pipeline, such as manifest validation, smoke tests, and automated rollback, so the failure is caught earlier.

Likely follow-ups

- What would the status Pending tell you to check?
- When do you roll back versus fix forward?

This is the Principal-level incident response question. Show structure, breadth, and depth — not just commands.

SYSTEMATIC DIAGNOSIS METHODOLOGY

Phase 1 – Scope the failure:

```
kubectl rollout status deployment/myapp # stuck? rolling back?
```

```
kubectl get pods -l app=myapp # pod states
```

```
kubectl get events --sort-by=.lastTimestamp -n <ns> | tail -20
```

Phase 2 – Inspect the failing pod:

```
kubectl describe pod <failing-pod> # Events section is gold
```

```
kubectl logs <failing-pod> --previous # logs from crashed container
```

```
kubectl logs <failing-pod> -c init-container-name # init container logs
```

Phase 3 – Identify the root cause category:

ImagePullBackOff → registry auth, wrong tag, image doesn't exist

CrashLoopBackOff → app crash: check logs, exit code, OOM (137)

Pending → scheduling: resources, node selectors, taints, PVC unbound

OOMKilled (exit 137) → increase memory limit, fix memory leak

Readiness failing → app not ready: check probe endpoint, startup time

Phase 4 – Immediate relief:

```
kubectl rollout undo deployment/myapp # rollback if new version
```

```
kubectl scale deployment/myapp --replicas=5 # scale up if degraded
```

Phase 5 – Fix and verify:

Fix root cause → commit → CI pipeline → re-deploy → verify rollout

Common patterns: ImagePullBackOff with "unauthorized" = Workload Identity not configured or imagePullSecret missing. CrashLoopBackOff with exit 1 = application error (check logs). CrashLoopBackOff with exit 137 = OOM kill (increase memory limit). Pending with "0/3 nodes are available" = insufficient resources (scale node pool) or taint mismatch. Readiness probe failing = app slow to start (add startupProbe) or dependency unavailable (check DB connectivity).

Principal-level deliverable: After every significant deployment failure, produce a postmortem: what failed, why it wasn't caught in staging, what the detection gap was (missing alert/test), and what process change prevents recurrence. Deploy failures that reach production represent gaps in testing or staging environment fidelity — fix the pipeline, not just the immediate issue.

 Official docs

- [Kubernetes: Debug Pods](#)

Q51 **What is the difference between CMD and ENTRYPOINT in a Dockerfile?**

Docker fundamentals

 Spoken answer · ~62 s

ENTRYPOINT sets the program the container runs, and CMD provides default arguments to it, or a default command if no entrypoint exists. When I run the image with extra arguments, they replace CMD but are appended to ENTRYPOINT, so a well-built image behaves like a command-line tool: the entrypoint is the binary and CMD holds sensible defaults. Both should use the exec form, a JSON array, not the shell form. The shell form wraps the process in a shell, which becomes process one and may not forward SIGTERM, so the application will not shut down gracefully and Kubernetes waits until the kill timeout. If I need environment preparation, I write an entrypoint script that ends with exec on the real process, so it replaces the shell. To override the entrypoint for debugging, I use the entrypoint flag on docker run. Knowing this helps with containers that ignore stop signals.

Likely follow-ups

- Why use exec form?
- How do you override the entrypoint for debugging?

ENTRYPOINT defines the executable; CMD supplies default arguments or a default command. Arguments to docker run replace CMD but append to ENTRYPOINT.

```
ENTRYPOINT ["python", "app.py"]
CMD ["--port", "8080"]
```

```
docker run img --port 9090
```

Interview tip: Use exec form so signals reach the process.

 Official docs

- [Docker: Dockerfile reference](#)

Q52 **What is the difference between COPY and ADD, and which should you use?**

Docker fundamentals

 Spoken answer · ~60 s

Both instructions add files to the image, but ADD has extra features: it can fetch a remote URL and automatically extract a local tar archive. Those features make behaviour less obvious, and a remote download inside ADD adds no checksum verification and cannot be cached or cleaned up in the same layer. So best practice is to use COPY for plain file copying, since what it does is explicit, and to use a RUN step with curl, a checksum verification and cleanup for downloads. ADD is acceptable when I deliberately want to unpack a local archive, such as a root filesystem, into the image. I also set ownership with the chown flag to avoid running everything as root, and I order COPY instructions from least to most frequently changing, so that the layer cache stays effective. Clear instructions make images reproducible and easier to review.

Likely follow-ups

- Why avoid ADD for URLs?
- How does COPY ordering affect caching?

COPY copies local files; ADD also extracts local archives and fetches URLs. Prefer COPY for predictability.

```
COPY --chown=app:app src/ /app/src/
```

```
ADD rootfs.tar.gz /
```

Interview tip: Use COPY unless you need tar extraction.

 Official docs

- [Docker: Dockerfile reference](#)

Q53 **How do ARG and ENV differ in a Dockerfile, and what are the secret-leak risks?**

Docker fundamentals

 Spoken answer · ~64 s

ARG defines a variable that is available only while building, and can be set with the build-arg flag. ENV sets an environment variable that is stored in the image configuration and visible to the running container. The security problem is that both are recorded: anyone with the image can run history or inspect and read ARG values used in commands and ENV values. So passing a token as a build argument to download a private package leaks it into the image. The correct approach is BuildKit secret mounts, which expose a secret file to a single RUN step without writing it into any layer, or SSH agent forwarding for private repositories. Runtime secrets should be injected by the orchestrator at start, not baked in. I also remember that ARG values declared before FROM are not available after it unless redeclared, and that changing an ARG busts the cache from the point of use.

Likely follow-ups

- Why is ARG not safe for secrets?
- What is a BuildKit secret mount?

ARG exists only at build time; ENV persists into the image and runtime. Both are visible in image history, so never pass secrets through them.

```
ARG VERSION=1.0
ENV APP_ENV=prod
```

```
docker build --secret id=npm,src=.npmrc .
```

Interview tip: Use BuildKit secret mounts for secrets.

 Start here (official docs)

- [Docker documentation](#)

Q54 **What happens when you run `docker stop`, and why do containers sometimes take ten seconds to stop?**

Docker fundamentals

 Spoken answer · ~63 s

Docker stop sends SIGTERM, or the image's configured stop signal, to the main process of the container, waits for the timeout, which is ten seconds by default, and then sends SIGKILL. If a container always takes the full ten seconds, the application is not handling SIGTERM. Common reasons are the shell form of CMD, where the shell is PID one and does not forward the signal, an entrypoint script that does not use exec, or an application that has no signal handler; as PID one, Linux does not apply default signal actions, so without a handler the signal is ignored. Fixes include using the exec form, adding exec to scripts, implementing graceful shutdown in the application, or using a tiny init such as tini through the init flag, which forwards signals and reaps zombies. In Kubernetes the same applies with the termination grace period, and a preStop hook helps drain connections.

Likely follow-ups

- Why is PID 1 special?
- What does the init flag do?

Docker sends SIGTERM, waits the grace period, then sends SIGKILL. Delays mean the process ignores or never receives SIGTERM.

```
docker stop -t 30 web
```

```
docker inspect --format '{{.Config.StopSignal}}' web
```

Interview tip: PID 1 behaves differently for signals.

 Official docs

- [Docker documentation](#)

Q55 **How do you clean up disk space used by Docker safely on a build host?**

Docker fundamentals

 Spoken answer · ~63 s

I start by looking at usage with `docker system df`, which breaks down images, containers, local volumes and build cache. Then I prune in order of safety. Stopped containers and dangling images can be removed freely. For unused images, I use the `all` option with an `until` filter, so recent images that are probably still in use remain. Build cache can grow very large on CI hosts, so I cap it with `builder prune` and a `keep-storage` limit. Volume pruning is the dangerous one because it can delete data for stopped containers' named volumes, so I avoid it automatically and only do it after checking what is unused. On shared CI runners, I schedule the cleanup and also use ephemeral runners so the problem disappears. I also check the container log size, because unrotated JSON logs are a common hidden cause of full disks, and set `max-size` and `max-file` options.

Likely follow-ups

- Why is volume prune risky?
- What else fills the disk?

Inspect with `docker system df`, prune dangling images, stopped containers, unused networks and build cache with age filters.

```
docker system df
```

```
docker image prune -a --filter until=168h
```

```
docker builder prune --keep-storage 10GB
```

Interview tip: Be careful with volume prune.

 Official docs

- [Docker documentation](#)

Q56 **What is Docker's restart policy and how does it differ from Kubernetes restarts?**

Docker fundamentals

 Spoken answer · ~65 s

Docker restart policies tell the daemon what to do when a container exits. No means never restart; on-failure restarts only on non-zero exit and can have a maximum retry count; always restarts regardless and also on daemon start; unless-stopped is like always but respects a manual stop across daemon restarts. This gives basic self-healing on a single host. It lacks health awareness beyond the process exit, scheduling to other nodes and rescheduling when the host fails. Kubernetes handles this in layers: the pod restartPolicy governs restarts by the kubelet with exponential backoff, which produces CrashLoopBackOff, while controllers like Deployments replace pods and the scheduler places them on healthy nodes. Probes add health awareness. For a single-node setup, unless-stopped plus a health check is reasonable; for anything important, I use an orchestrator. I verify the result in a staging cluster before relying on it in production. That keeps the behaviour predictable for everyone on the team.

Likely follow-ups

- What is the difference between always and unless-stopped?
- What does CrashLoopBackOff mean?

Policies like no, on-failure, always and unless-stopped control daemon-level restarts; Kubernetes uses pod restartPolicy and controllers.

```
docker run --restart unless-stopped nginx
```

```
docker update --restart on-failure:5 web
```

Interview tip: Use on-failure with a cap for crash loops.

 Official docs

- [Kubernetes: Pod lifecycle & termination](#)
- [Docker documentation](#)

Q57 **How do you copy files in and out of containers and when is it appropriate?**

Docker fundamentals

 Spoken answer · ~58 s

Docker cp copies files between a container and the host, whether the container is running or stopped, and kubectl cp does the same for pods. They are useful for debugging, such as extracting a heap dump, a log file or a generated report. They are not a deployment mechanism, because anything changed inside a container's writable layer disappears when it is replaced. If a workflow regularly needs files in or out, the design should change: mount a volume or bind mount for shared data, push artifacts to object storage, or ship logs to a log system. Kubectl cp depends on tar being present in the container, which distroless images lack, so I use ephemeral debug containers or exec with other means in those cases. I also watch for sensitive data in dumps and clean up copied files afterwards.

Likely follow-ups

- Why doesn't kubectl cp work in distroless images?
- What is a better approach for regular file exchange?

docker cp works for debugging; production data should use volumes or object storage, not manual copies.

```
docker cp web:/var/log/app.log ./app.log
```

```
kubectl cp ns/pod:/tmp/dump ./dump
```

Interview tip: Treat container filesystems as disposable.

 Start here (official docs)

- [Docker documentation](#)

Q58 **How do you choose a base image: Alpine, Debian slim, distroless or scratch?**

Images & Dockerfile

 Spoken answer · ~58 s

The base image determines size, security surface and compatibility. Debian slim variants are a safe default: moderately small, glibc-based and widely compatible. Alpine is very small and uses musl libc, which can cause subtle problems or slow builds for software with native extensions, such as some Python wheels, and DNS behaviour differences. Distroless images contain only the runtime and certificates, with no shell or package manager, reducing vulnerabilities and attacker tools, but making debugging harder, which I address with ephemeral debug containers. Scratch is completely empty and fits statically linked binaries, like Go programs. I also consider vendor-maintained minimal images, how quickly security patches arrive, and the ability to pin by digest. Whatever I choose, I rebuild regularly to pick up patches, run as a non-root user, and standardise on a small set of approved bases across the organisation.

Likely follow-ups

- What problems can musl cause?
- How do you debug a distroless container?

Balance size, compatibility, debuggability and attack surface. Alpine uses musl, distroless has no shell, scratch is empty for static binaries.

```
FROM gcr.io/distroless/static:nonroot
```

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

Interview tip: Smaller is not always better if it breaks libc compatibility.

 Official docs

- [Google SRE Workbook: Implementing SLOs](#)
- [Distroless images](#)

 Spoken answer · ~55 s

Reproducible means building the same inputs produces the same image. I pin the base image by digest rather than tag, since tags move. I pin operating system package versions where practical, or accept controlled drift with scheduled rebuilds, and I install language dependencies from lock files with hashes. I avoid commands that fetch whatever is current, such as curl of a latest script without a checksum. Build steps should not embed timestamps or random values, and I set SOURCE_DATE_EPOCH for tools that support normalising dates. The build runs in a clean, controlled environment with a pinned builder version, and I record provenance and an SBOM. Perfect bit-for-bit reproducibility is hard, but even reaching consistent dependency versions greatly improves debugging, security patching and audits, because I can state exactly what ran in production.

Likely follow-ups

- Why pin by digest?
- What does SOURCE_DATE_EPOCH do?

Pin base images by digest, pin package versions, use lock files, avoid latest and timestamps, and build in a controlled environment.

```
FROM python:3.12-slim@sha256:<digest>
```

```
pip install --no-cache-dir -r requirements.lock
```

Interview tip: latest is not a version.

 Start here (official docs)

- [Docker documentation](#)

 Spoken answer · ~58 s

Smaller images pull faster, start faster and have fewer vulnerabilities. The biggest win is a multi-stage build: compile and package in a builder stage with all the tools, then copy only the final artifacts into a slim runtime stage. Then I choose a small base, install packages with no-install-recommends, and remove package caches in the same RUN instruction, because deleting files in a later layer does not shrink earlier ones. A dockerignore file keeps the build context and image free of git history, tests and local files. I avoid installing debugging tools in production images, strip debug symbols when appropriate, and check what takes space with docker history or a tool like dive. I validate that nothing needed at runtime was removed by running tests against the final image, since overly aggressive trimming can break dynamic libraries or certificates.

Likely follow-ups

- Why must cleanup be in the same layer?
- How do you inspect what makes an image large?

Multi-stage builds, slim bases, combined RUN steps with cleanup, dockerignore, and removing build tools and caches.

```
RUN apt-get update && apt-get install -y --no-install-recommends curl && rm -rf /var/lib/apt/lists/*
```

```
docker history --no-trunc img
```

Interview tip: Cleanup must be in the same layer.

 Official docs

- [Docker: Dockerfile best practices](#)

What is an image digest versus a tag, and why does it matter for deployments?

 Spoken answer · ~58 s

A tag such as 1.4 or latest is just a pointer in a registry and can be moved to a different image at any time, by accident or by an attacker. A digest is the SHA-256 hash of the image manifest; it identifies exactly one set of bytes. Deploying by digest ensures that every replica, in every cluster, runs the same image, and that rollbacks restore the exact earlier artifact. It also lets signature and scan results be tied to the specific image. In practice, CI tags images with the commit SHA for readability but records and deploys the digest, and manifests in Git reference it, often updated by automation. Kubernetes with a tag can behave differently across nodes if the tag changed and image pull policy forces a re-pull. Admission policies can also require digests or signatures.

Likely follow-ups

- Why is latest dangerous?
- How do you find an image digest?

Tag vs digest

Tag	Digest
Mutable pointer	Immutable sha256
Can change underneath you	Reproducible deploys

A tag is a movable label; a digest is the immutable content hash. Deploy by digest for exact, verifiable artifacts.

```
docker pull nginx@sha256:...
```

```
docker inspect --format '{{index .RepoDigests 0}}' app:1.4
```

Interview tip: Tags can be overwritten.

Official docs

- [OCI Image Specification](#)

 Spoken answer · ~59 s

By default, processes in a container run as root, and while namespaces limit what that means, a container escape or a mounted host path turns root inside into a much bigger problem. Running as an unprivileged user limits damage from application vulnerabilities, prevents writing to system files and satisfies security baselines. In the Dockerfile I create a dedicated user with a fixed numeric UID and switch with USER, and make sure the directories the application writes to are owned by it. In Kubernetes, I enforce this with runAsNonRoot, runAsUser, a read-only root filesystem with writable emptyDir mounts where necessary, dropping all capabilities, and disabling privilege escalation, and I use admission policies to reject non-compliant pods. Binding to ports below 1024 needs a capability or a higher port with a service mapping, which is a common stumbling block when migrating existing images.

Likely follow-ups

- Why a numeric UID?
- How do you handle privileged ports?

Create a user in the image, set USER, drop capabilities and, in Kubernetes, set runAsNonRoot and a read-only root filesystem.

```
RUN adduser --system --uid 10001 app
USER 10001
```

```
securityContext:
  runAsNonRoot: true
  readOnlyRootFilesystem: true
```

Interview tip: Use a numeric UID.

 Official docs

- [Kubernetes: Security context](#)

Q63 **What are Linux namespaces and cgroups, and how do containers use them?**

Container runtime

 Spoken answer · ~60 s

A container is not a separate machine but a normal Linux process with restricted view and resources. Namespaces provide isolation of what the process can see: PID for its own process tree, network for interfaces and routing, mount for the filesystem view, UTS for hostname, IPC, user for ID mapping and cgroup. Control groups, or cgroups, limit and account for resource usage such as CPU, memory, I/O and number of processes; exceeding a memory limit triggers the OOM killer inside that group. The runtime combines these with a root filesystem from the image layers, restricted capabilities, seccomp filters and security modules like AppArmor or SELinux. Because all containers share the host kernel, a kernel vulnerability affects all, which is why stronger isolation like gVisor or microVMs exists. Understanding this explains behaviour such as seeing host CPU counts and limits applying only through cgroups.

Likely follow-ups

- What does the PID namespace do?
- What happens when a cgroup memory limit is exceeded?

Namespaces isolate what a process sees; cgroups limit what it can use. A container is a process with both applied.

```
lsns
```

```
cat /sys/fs/cgroup/memory.max
```

```
unshare --pid --fork --mount-proc bash
```

Interview tip: No hypervisor is involved.

 Official docs

- [Linux man-pages: namespaces\(7\)](#)
- [Linux man-pages: cgroups\(7\)](#)

 Spoken answer · ~63 s

The stack has layers. At the bottom, runc is an OCI-compliant tool that actually creates the container using namespaces, cgroups and the root filesystem, then exits. Above it, containerd and CRI-O are daemons that pull images, manage snapshots and storage, handle networking hooks and supervise containers by calling runc. Kubernetes talks to them through the Container Runtime Interface, and since the removal of dockershim, clusters use containerd or CRI-O directly. Docker is a full developer experience: CLI, build system, image management and API, running on top of containerd. In a node, I use crictl to inspect CRI containers rather than docker commands. Alternative low-level runtimes like gVisor's runsc or Kata Containers plug in at the same level as runc for stronger isolation, selected in Kubernetes through RuntimeClass. I write the reasoning into the runbook so the on-call engineer understands it. That keeps the behaviour predictable for everyone on the team.

Likely follow-ups

- What did the dockershim removal change?
- What is a RuntimeClass?

runc is the low-level OCI runtime; containerd and CRI-O are higher-level runtimes managing images and lifecycle; Docker adds a developer-friendly engine on top.

```
crictl ps
```

```
ctr -n k8s.io containers list
```

Interview tip: Kubernetes talks CRI, not Docker.

 Official docs

- [Kubernetes: Container runtimes](#)
- [OCI Runtime Specification](#)
- [Docker documentation](#)

Q65 **How do you investigate a container killed by OOM, and how do you set memory limits properly?**

Container runtime

 Spoken answer · ~60 s

An OOM-killed container shows exit code 137 and a reason OOMKilled in its last state. I check the memory usage trend in monitoring before the kill to distinguish a leak, a spike from load or a limit set too low. Kernel messages on the node confirm the cgroup that hit the limit. For runtimes with managed memory, like the JVM, I make sure the heap maximum is set relative to the container limit, leaving room for metaspace, thread stacks and native memory, otherwise the process grows beyond the cgroup limit even though the heap looks fine. I size requests from typical usage and limits from the observed peak plus headroom, and use memory limits equal to requests for critical pods to get predictable behaviour. If it is a leak, I use profilers and heap dumps to fix it instead of raising limits indefinitely.

Likely follow-ups

- What does exit code 137 mean?
- Why can a JVM exceed the container limit?

Check exit code 137 and OOMKilled in status, review memory metrics, then size requests and limits from observed usage with headroom.

```
kubectl describe pod p | grep -A5 'Last State'
```

```
dmesg | grep -i 'killed process'
```

Interview tip: Heap limits must sit below container limits.

 Official docs

- [Kubernetes: Resource management for Pods & containers](#)

Q66 **How does CPU limiting work in containers and why can it cause latency?**

Container runtime

 Spoken answer · ~58 s

Kubernetes CPU requests translate to relative CPU shares used for scheduling and contention, while limits use the CFS bandwidth controller: the container gets a quota of CPU time per hundred-millisecond period. A multi-threaded application can burn its whole quota in a few milliseconds across several cores and then be paused until the next period, even though the node has idle CPU. This throttling shows up as latency spikes, particularly at the tail, and is easy to miss because average CPU utilisation looks low. I detect it using the throttled periods metrics from cAdvisor. Mitigations include raising or removing CPU limits while keeping requests accurate, which many teams do for latency-sensitive services, tuning thread pool sizes to the CPU count, and using the static CPU manager policy for guaranteed workloads. The trade-off of no limit is noisy neighbours, so monitoring matters.

Likely follow-ups

- How do you detect throttling?
- Why do some teams remove CPU limits?

CPU limits use CFS quota per period; hitting the quota throttles the process even when the node has idle CPU.

```
cat /sys/fs/cgroup/cpu.stat
```

```
rate(container_cpu_cfs_throttled_periods_total[5m])
```

Interview tip: Throttling hurts tail latency.

 Official docs

- [Kubernetes: Resource management for Pods & containers](#)

Q67 What is the container lifecycle from image pull to termination in Kubernetes?

Container runtime

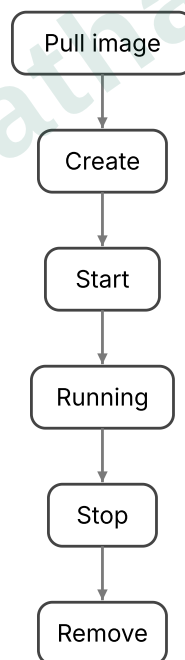
 Spoken answer · ~58 s

After the scheduler assigns a pod to a node, the kubelet asks the runtime to create the pod sandbox and set up networking via the CNI plugin. Volumes are mounted, images are pulled according to pull policy, and init containers run to completion in order. Then the main containers start. Startup, readiness and liveness probes determine when it is considered started, when it receives traffic from services, and when it is restarted. During normal operation the kubelet restarts failed containers according to the restart policy with backoff. On termination, the pod is marked as terminating and removed from service endpoints, the preStop hook runs, SIGTERM is sent, the grace period counts down, and finally SIGKILL if needed. Understanding the phases helps debug states like ContainerCreating, ImagePullBackOff, Init errors and slow shutdowns, since each points to a specific stage.

Likely follow-ups

- What happens to endpoints during termination?
- What is the pod sandbox?

Container lifecycle



Schedule, pull image, create sandbox and network, run init containers, start containers with probes, serve traffic, then preStop and SIGTERM on termination.

```
kubectl get events --sort-by=.lastTimestamp
```

```
kubectl describe pod p
```

Interview tip: Events narrate each phase.

 Official docs

- [Kubernetes: Images \(pull policy, registries\)](#)

Q68 How does Docker's default bridge network provide NAT and DNS, and how do user-defined bridges differ?

Container networking

 Spoken answer · ~64 s

On a single host, Docker creates a bridge device and connects each container with a virtual ethernet pair. Outbound traffic is masquerade NAT through iptables, and published ports use DNAT rules to forward host ports to container IPs. The default bridge does not provide name resolution between containers, so you would need legacy links or IP addresses. A user-defined bridge network provides an embedded DNS server, so containers reach each other by name or alias, and it offers better isolation because only containers on that network can talk to each other. Containers can join multiple networks. I use user-defined networks for compose stacks and local development. Multi-host communication needs overlay networks or an orchestrator's networking. When debugging, I check the bridge, veth interfaces, iptables rules and the container's resolv.conf. I verify the result in a staging cluster before relying on it in production. That keeps the behaviour predictable for everyone on the team.

Likely follow-ups

- Why is DNS missing on the default bridge?
- How are published ports implemented?

The default bridge uses iptables NAT and no name resolution between containers; user-defined bridges add automatic DNS and better isolation.

```
docker network create app-net
```

```
docker run --network app-net --name db postgres
```

```
iptables -t nat -L -n
```

Interview tip: Use user-defined networks.

 Official docs

- [RFC 3022: Traditional NAT](#)
- [Docker: Networking](#)
- [Docker documentation](#)

 Spoken answer · ~59 s

I start inside the container, examining `resolv.conf` for the nameserver, search domains and the `ndots` option. Then I test resolving a cluster service name, an external name and a fully qualified name with a trailing dot, using `nslookup` or `dig`, from a debug container if the image lacks tools. If cluster names fail but external names work, or the reverse, that points to different layers: CoreDNS, upstream resolvers or network policies blocking port 53. I review CoreDNS logs and metrics for errors, timeouts and `SERVFAIL`, and check that its pods are healthy and not overloaded. A classic cause of slowness is the default `ndots` of five, which makes unqualified external names try many search suffixes first, multiplying queries; using fully qualified names or lowering `ndots` helps. Intermittent five-second delays can indicate conntrack races with UDP, mitigated by node local DNS caching.

Likely follow-ups

- What does `ndots` do?
- What is node-local DNS cache for?

Check `resolv.conf`, test lookups with `nslookup` or `dig`, review CoreDNS logs and `ndots` behaviour, and test search domains.

```
cat /etc/resolv.conf
```

```
nslookup kubernetes.default
```

```
kubectl logs -n kube-system -l k8s-app=kube-dns
```

Interview tip: `ndots:5` multiplies queries.

 Official docs

- [Kubernetes: Debugging DNS resolution](#)
- [Kubernetes: DNS for Services and Pods](#)

Q70 **What is the difference between ClusterIP, NodePort, LoadBalancer and Ingress?**

Container networking

 Spoken answer · ~58 s

A ClusterIP service gives a stable virtual IP reachable only inside the cluster and is the default for internal communication. NodePort opens the same port on every node and forwards to the service, useful for testing and as a building block, but awkward for production. LoadBalancer asks the cloud provider to provision an external load balancer pointing at the service, which is simple but creates one balancer per service, costing money. Ingress is an HTTP layer resource: rules map hostnames and paths to services, implemented by a controller such as NGINX or a cloud load balancer, so one entry point serves many services with TLS termination. The newer Gateway API offers a more expressive, role-oriented model. I choose ClusterIP internally, Ingress or Gateway for web traffic, and LoadBalancer for non-HTTP protocols or when a dedicated address is required.

Likely follow-ups

- Why not a LoadBalancer per service?
- What is the Gateway API?

ClusterIP is internal; NodePort exposes on each node; LoadBalancer provisions a cloud LB; Ingress routes HTTP by host and path through a controller.

```
kubectl expose deploy web --port 80 --type ClusterIP
```

```
kubectl get ingress
```

Interview tip: One Ingress can front many services.

 Official docs

- [Kubernetes: Service](#)
- [Kubernetes: Ingress](#)

 Spoken answer · ~57 s

A service IP is virtual: no process listens on it. Kube-proxy on each node watches services and endpoints and programs the kernel to rewrite traffic destined for the service IP to one of the healthy pod IPs. In iptables mode it creates chains of rules with random probability to pick a backend; this works well but updates and lookups slow down with tens of thousands of services because rules are evaluated linearly. IPVS mode uses the kernel's load balancer with hash tables, giving constant-time lookups and more algorithms, such as least connections. Some CNIs, like Cilium, replace kube-proxy with eBPF programs for better performance and visibility. Connection tracking underlies these methods, so conntrack table exhaustion can cause dropped connections on busy nodes. Knowing this helps debug why a service does not route when endpoints are empty.

Likely follow-ups

- Why is IPVS better at scale?
- How can conntrack cause problems?

kube-proxy programs node rules that translate service virtual IPs to pod endpoints; iptables mode is rule-chain based, IPVS uses kernel load balancing with better scale.

```
iptables-save | grep KUBE-SVC
```

```
ipvsadm -Ln
```

Interview tip: Services have no process listening on the VIP.

 Official docs

- [Kubernetes: Service](#)
- [Kubernetes: kube-proxy](#)
- [Netfilter documentation](#)

 Spoken answer · ~61 s

In Compose, each service is reachable by its service name on the project's network through Docker's embedded DNS. In Kubernetes, services get names like `service.namespace.svc.cluster.local`, resolved by CoreDNS, and headless services return pod IPs directly. The pitfall is client-side caching. Some runtimes, like older JVM defaults, cache lookups indefinitely, and connection pools keep long-lived connections, so when pods are replaced or a database fails over, clients keep talking to a dead IP. I configure DNS TTL honouring in the runtime, set connection lifetimes and health checks in pools, and use retries with re-resolution. For stable service IPs the ClusterIP hides pod churn, but for headless services and external databases the client must handle changes. Testing failover deliberately reveals these issues before an incident does. I write the reasoning into the runbook so the on-call engineer understands it. That keeps the behaviour predictable for everyone on the team.

Likely follow-ups

- What is a headless service?
- How do connection pools hide failover problems?

Both give services DNS names; clients that cache resolved IPs forever can miss changes after pods are replaced.

```
curl http://db:5432
```

```
curl http://orders.shop.svc.cluster.local
```

Interview tip: Honour DNS TTL in your client.

 Official docs

- [Docker Compose](#)
- [Docker documentation](#)

Q73 **What is the difference between PersistentVolume, PersistentVolumeClaim and StorageClass?**

Storage & volumes

 Spoken answer · ~59 s

A PersistentVolume represents a piece of storage in the cluster, either created by an administrator or dynamically provisioned. A PersistentVolumeClaim is a request by a workload for storage with a size, access mode and optionally a class; the control plane binds it to a matching volume. A StorageClass describes a type of storage and its provisioner and parameters, such as disk type, replication or encryption, and enables dynamic provisioning: creating a claim automatically creates the underlying disk. Important settings include the volume binding mode, where wait-for-first-consumer delays provisioning until a pod is scheduled so the disk lands in the right zone, the reclaim policy, where delete removes the disk with the claim and retain keeps it, and allowVolumeExpansion. For StatefulSets, claim templates create one claim per replica. I back up important volumes with snapshots because deletion of a claim can destroy data.

Likely follow-ups

- Why use WaitForFirstConsumer?
- What does the reclaim policy control?

A PV is storage; a PVC is a request for it; a StorageClass defines how to dynamically provision PVs.

```
kubectl get pv,pvc,sc
```

```
kubectl describe pvc data-0
```

Interview tip: Reclaim policy decides what happens on delete.

 Official docs

- [Kubernetes: Persistent Volumes](#)
- [Kubernetes: StorageClasses](#)

Q74 **What are the access modes ReadWriteOnce, ReadOnlyMany, ReadWriteMany and ReadWriteOncePod?**

Storage & volumes

 Spoken answer · ~58 s

ReadWriteOnce allows the volume to be mounted read-write by a single node; multiple pods on that node can still use it. ReadOnlyMany allows many nodes to mount it read-only, and ReadWriteMany allows many nodes to read and write, which requires a shared file system such as NFS, Filestore or EFS, because ordinary block disks cannot be attached to many nodes. ReadWriteOncePod restricts to a single pod in the whole cluster. These modes matter when scaling or rolling updates: with a Deployment using an RWO volume, a new pod scheduled to a different node cannot attach until the old one releases it, causing stuck rollouts, so I use the Recreate strategy or a StatefulSet. If an application needs shared storage across replicas, I consider whether object storage or a database would be a better architecture than a shared file system.

Likely follow-ups

- Why do RWO volumes block rolling updates?
- When is object storage better than RWX?

They define how many nodes or pods can mount a volume and with what permission; block disks are usually RWO, shared file systems support RWX.

```
accessModes: ["ReadWriteOnce"]
```

```
kubectl get pvc -o wide
```

Interview tip: RWO limits to one node, not one pod.

 **Start here (official docs)**

- [Docker documentation](#)

 Spoken answer · ~60 s

First I confirm the usage with `df` inside the pod, then find the largest directories and whether growth is logs, temporary files, write-ahead logs or real data. For quick relief I remove safe items such as old logs or compacted data according to the application's guidance, not by randomly deleting database files. For a durable fix I expand the claim: if the storage class allows volume expansion, I patch the requested size, and the controller resizes the disk and then the file system, often online, though some drivers need a pod restart. Shrinking is generally impossible, so I size with headroom. Afterwards I add monitoring on free space percentage and on predicted time to full, set retention and rotation policies, and make sure backups exist before any risky action. For databases, an out-of-disk condition can cause corruption or readonly mode, so urgency is high.

Likely follow-ups

- Can volumes be shrunk?
- What alerts prevent this?

Identify what consumes space, free it safely or expand the PVC online if the class allows, and add alerts and retention.

```
kubectl exec pod -- df -h
```

```
kubectl patch pvc data-0 -p '{"spec":{"resources":{"requests":{"storage":"200Gi"}}}}'
```

Interview tip: Check `allowVolumeExpansion` first.

 Start here (official docs)

- [Docker documentation](#)

 Spoken answer · ~58 s

With CSI snapshot support, a VolumeSnapshot object triggers the storage backend to take a point-in-time copy of a volume, and I can create a new claim from that snapshot to restore or clone. These snapshots are crash-consistent, meaning like a power loss at that moment; databases usually recover from them but for guaranteed consistency I flush and lock tables, use the database's own backup tool, or pre-and-post hooks. Tools such as Velero back up Kubernetes resources together with volume snapshots or file-level copies to object storage, enabling cluster-level disaster recovery and migration. I store backups in a different failure domain, encrypt them, apply retention and, critically, test restores on a schedule, since an untested backup is only a hope. I also define recovery point and recovery time objectives, which decide how frequently to snapshot and how restore is rehearsed.

Likely follow-ups

- What is crash-consistent versus application-consistent?
- What does Velero back up?

CSI snapshots capture a point-in-time copy of a volume; application-consistent backups also need quiescing or native tools.

```
kubectl get volumesnapshot
```

```
velero backup create daily --include-namespaces prod
```

Interview tip: Test restores regularly.

 Official docs

- [Kubernetes: Volume snapshots](#)

Q77 **What are emptyDir and ephemeral volumes, and when should containers use them?**

Storage & volumes

 Spoken answer · ~55 s

An emptyDir volume is created when a pod is assigned to a node and deleted when the pod is removed, so it survives container restarts but not pod replacement. It is shared by all containers in the pod, which makes it handy for scratch space, caches, sidecar file exchange, or providing writable directories when the root filesystem is read-only. The medium can be the node's disk or memory using tmpfs, which is fast but counts toward the container's memory limit. Without a size limit, a runaway process can fill the node's disk and trigger evictions, so I set sizeLimit and monitor ephemeral storage usage. Generic ephemeral volumes allow a per-pod claim from a storage class for larger scratch needs. Anything that must survive pod rescheduling belongs in a persistent volume or external storage.

Likely follow-ups

- What survives a container restart?
- Why set sizeLimit?

emptyDir is scratch space tied to the pod lifetime, optionally memory-backed; use it for caches and temporary files, not persistent data.

```
volumes:  
- name: tmp  
  emptyDir:  
    medium: Memory  
    sizeLimit: 256Mi
```

Interview tip: Set sizeLimit to avoid node pressure.

 Official docs

- [Terraform: Sensitive data in state](#)

 Spoken answer · ~62 s

In a rolling update, the Deployment controller replaces old pods with new ones gradually. Max surge is how many pods above the desired replica count may exist temporarily; max unavailable is how many pods below desired may be unavailable during the update. Setting max unavailable to zero with a surge of one or a percentage means new pods must become ready before old ones are removed, preserving full capacity, at the cost of needing extra resources and a slower rollout. Allowing unavailability speeds things up but reduces capacity. These interact with readiness probes: a pod counts as available only when ready, so a good probe prevents traffic going to a broken version and stalls the rollout, which is the desired failure mode. I also set progress deadline seconds and min ready seconds, so that a bad rollout is flagged and pods must remain stable before the rollout continues.

Likely follow-ups

- Why set maxUnavailable to zero?
- What does minReadySeconds do?

maxSurge sets how many extra pods can be created above desired; maxUnavailable sets how many may be unavailable; together they set rollout speed and capacity.

```
strategy:
  rollingUpdate:
    maxSurge: 25%
    maxUnavailable: 0
```

Interview tip: maxUnavailable 0 keeps full capacity.

 Official docs

- [Kubernetes: Deployments](#)
- [Kubernetes: Deployment strategies \(rolling update\)](#)

Q79 How do you achieve graceful shutdown so rolling updates drop no requests?

Deployment strategies

🗣 Spoken answer · ~56 s

When a pod terminates, removal from service endpoints and the SIGTERM signal happen in parallel, so for a short time traffic may still arrive after the application begins shutting down. To avoid dropped requests, I add a preStop hook that sleeps for a few seconds, giving load balancers and kube-proxy time to stop sending new connections. On SIGTERM, the application stops accepting new work, finishes in-flight requests, closes connections and exits, within the termination grace period which I size larger than the longest request plus the preStop delay. Applications should flip readiness to false at the start of shutdown. For long-lived connections like websockets, clients need reconnection logic. I test this by running a load generator during a rollout and checking for errors. Without these steps, every deployment causes a small burst of 502 errors.

Likely follow-ups

- Why do requests arrive after SIGTERM?
- How do you test graceful shutdown?

🚩 Graceful shutdown



Handle SIGTERM, fail readiness first, delay with preStop to allow endpoint propagation, then drain connections within the grace period.

```
lifecycle:
  preStop:
    exec:
      command: ["sleep", "10"]
terminationGracePeriodSeconds: 45
```

Interview tip: Endpoint removal is asynchronous.

📖 Official docs

- [Kubernetes: Pod lifecycle & termination](#)
- [Kubernetes: Deployments](#)
- [Kubernetes: Deployment strategies \(rolling update\)](#)

Q80 **What are Kubernetes Jobs and CronJobs, and what are their failure modes?**

Deployment strategies

 Spoken answer · ~56 s

A Job runs pods until a specified number complete successfully, retrying failures up to the backoff limit, which suits batch tasks and migrations. A CronJob creates Jobs on a cron schedule. Common failure modes: overlapping runs when a job takes longer than its interval, controlled with the concurrency policy, where forbid skips a run and replace cancels the old one; jobs that retry non-idempotent work and duplicate effects; jobs stuck without an active deadline; missed schedules after controller downtime, governed by the starting deadline; and accumulating completed pods, handled with history limits and `ttlSecondsAfterFinished`. Time zones also matter, as schedules default to the controller's zone unless specified. I design batch work to be idempotent, log clearly, alert on failed or missing runs using metrics, and set resource requests so jobs do not starve online services.

Likely follow-ups

- What does `concurrencyPolicy` control?
- How do you clean up finished jobs?

Jobs run to completion with retries; CronJobs create Jobs on a schedule. Watch concurrency policy, backoff limit, deadlines and missed schedules.

```
kubectl create job --from=cronjob/backup manual-1
```

```
concurrencyPolicy: Forbid
backoffLimit: 3
```

Interview tip: Make job work idempotent.

 Official docs

- [Kubernetes: CronJob](#)

 Spoken answer · ~64 s

The scheduler places pods using their requests, treating them as reserved capacity, while limits are enforced at runtime. Based on these, each pod gets a Quality of Service class. Guaranteed means every container has equal requests and limits for CPU and memory; Burstable means some requests are set but not equal to limits; BestEffort means none are set. Under node memory pressure, the kubelet evicts BestEffort pods first, then Burstable ones exceeding requests, and Guaranteed last, so critical services should have accurate requests and preferably be Guaranteed. Pod priority classes add scheduling preference and preemption. I base values on observed usage, use the vertical pod autoscaler recommendations, and set namespace quotas and limit ranges so teams cannot deploy unbounded pods. Over-requesting wastes money; under-requesting causes noisy-neighbour problems and evictions. I verify the result in a staging cluster before relying on it in production. That keeps the behaviour predictable for everyone on the team.

Likely follow-ups

- What is the eviction order?
- What do LimitRanges do?

Requests drive scheduling; limits cap usage. QoS class Guaranteed, Burstable or BestEffort decides eviction order under node pressure.

```
kubectl get pod p -o jsonpath='{.status.qosClass}'
```

```
kubectl top pods
```

Interview tip: Equal requests and limits give Guaranteed.

 Official docs

- [Kubernetes: Pod Quality of Service classes](#)

Q82 **What is the horizontal pod autoscaler and what can make it behave badly?**

Deployment strategies

 Spoken answer · ~56 s

The horizontal pod autoscaler periodically compares a metric to a target and computes the desired replica count, within minimum and maximum bounds. CPU utilisation is measured relative to the pod's requests, so without realistic requests the target is meaningless. Trouble comes from several places: metrics lag, so scaling reacts late to sudden spikes; flapping when thresholds are too tight, mitigated by stabilisation windows and scaling policies; slow startup times, so new pods are not ready when needed; scaling on CPU for workloads that are bound by queues or latency, where custom or external metrics such as queue length or requests per second are better; and downstream limits, because scaling the web tier may overwhelm a database. I combine HPA with cluster autoscaling for node capacity, set pod disruption budgets, and load test the scaling behaviour.

Likely follow-ups

- Why do requests matter for CPU-based HPA?
- When use custom metrics?

HPA adjusts replicas based on metrics like CPU or custom metrics; problems include missing requests, slow metrics, flapping and dependency limits.

```
kubectl autoscale deploy web --cpu-percent=60 --min=3 --max=20
```

```
kubectl describe hpa web
```

Interview tip: CPU percent is relative to requests.

 Official docs

- [Kubernetes: Horizontal Pod Autoscaling](#)

Q83 **How do you deploy a stateful database in containers, and when should you not?**

Deployment strategies

 Spoken answer · ~64 s

Containers can run databases, but the operational burden is real. A StatefulSet gives stable network identities, ordered rollout and per-replica persistent volumes. I add anti-affinity or topology spread constraints to separate replicas across nodes and zones, resource guarantees, PodDisruptionBudgets, and careful storage class selection for performance and expansion. Backups, point-in-time recovery, failover, upgrades and monitoring are the hard parts, so I prefer a mature operator that encodes them, rather than hand-written manifests. Even so, for critical production data I often recommend a managed database service, because the provider handles patching, replication and recovery, and the team's time goes into the product. Self-hosting in Kubernetes is reasonable for development environments, specialised needs, portability or when the team has strong database operations skills. The decision should consider recovery objectives, team capacity and cost. I write the reasoning into the runbook so the on-call engineer understands it. That keeps the behaviour predictable for everyone on the team.

Likely follow-ups

- What does a StatefulSet guarantee?
- When would you choose a managed database?

Use StatefulSets with persistent volumes, anti-affinity, backups and an operator, but consider managed services for critical data.

```
kubectl get sts,pvc
```

```
topologySpreadConstraints:  
- maxSkew: 1  
  topologyKey: topology.kubernetes.io/zone
```

Interview tip: Operators encode operational knowledge.

 Start here (official docs)

- [Docker documentation](#)

Q84 **What are Linux capabilities and seccomp, and how do you harden a container with them?**

Security

 Spoken answer · ~56 s

Traditional root has all privileges; Linux capabilities divide them into distinct units such as binding low ports, changing file ownership or administering the network. Containers get a default subset, but most applications need none, so I drop all and add back only specific ones, for example `NET_BIND_SERVICE`. Seccomp restricts which system calls the process may make; the runtime default profile blocks dozens of dangerous calls with little compatibility risk, and I enable it with `seccompProfile RuntimeDefault`, since by default Kubernetes may run unconfined. Together with a read-only root filesystem, no privilege escalation, running as non-root, and AppArmor or SELinux profiles, these controls limit what an attacker can do after exploiting the application. Pod Security Standards, enforced by admission, set baseline and restricted profiles that bundle these settings. For high-risk workloads, sandboxed runtimes add further isolation.

Likely follow-ups

- Which capability do you add for port 80?
- What does the restricted Pod Security Standard require?

Capabilities split root privileges into units; seccomp filters system calls. Drop all capabilities, add back only what is needed, and use the runtime default seccomp profile.

```
securityContext:
  capabilities:
    drop: ["ALL"]
  seccompProfile:
    type: RuntimeDefault
```

```
capsh --print
```

Interview tip: Drop ALL, then add.

 Official docs

- [Kubernetes: Security context](#)
- [Kubernetes: Seccomp tutorial](#)
- [Linux man-pages: capabilities\(7\)](#)

Q85 **What are Pod Security Standards and admission controls, and how do you roll them out?**

Security

 Spoken answer · ~63 s

Pod Security Standards define three levels: privileged with no restrictions, baseline that blocks known dangerous settings like privileged containers and host namespaces, and restricted, which follows hardening best practice including non-root, dropped capabilities and seccomp. The built-in Pod Security Admission controller enforces them per namespace using labels, with three modes: enforce rejects violating pods, audit records violations and warn tells the user. To roll out safely, I first label namespaces with audit and warn at the target level, review which workloads violate and fix their manifests, then switch to enforce. For finer or custom rules, such as allowed registries or required labels, I add a policy engine like Kyverno or Gatekeeper. System namespaces may need exemptions. Combining admission controls with CI checks gives early feedback and runtime guarantees. I verify the result in a staging cluster before relying on it in production. That keeps the behaviour predictable for everyone on the team.

Likely follow-ups

- What are the three modes?
- When do you need Kyverno or Gatekeeper too?

Privileged, baseline and restricted profiles enforced per namespace via labels; roll out in audit and warn modes before enforce.

```
kubectl label ns prod pod-security.kubernetes.io/enforce=restricted
```

```
kubectl label ns prod pod-security.kubernetes.io/warn=restricted
```

Interview tip: Audit first, then enforce.

 Official docs

- [Kubernetes: Pod Security Standards](#)
- [Google SRE Book: Handling Overload](#)

 Spoken answer · ~60 s

Detection comes from several sources: runtime security tools such as Falco that alert on suspicious behaviour like unexpected shells, outbound connections or reads of sensitive files; Kubernetes audit logs showing exec calls or role changes; network flow logs revealing unusual destinations; and image or workload anomalies. On suspicion, I isolate the pod, for example by applying a network policy to deny traffic using a quarantine label, rather than deleting it immediately, so memory and filesystem evidence remain. I capture logs, a filesystem snapshot and the pod specification, check which service account and secrets it could access, and rotate those credentials. Then I look for lateral movement and how it got in, such as a vulnerable dependency. Recovery means replacing the workload from a verified clean image, patching the cause and reviewing detections. A rehearsed runbook and least privilege design reduce the blast radius.

Likely follow-ups

- Why isolate instead of delete?
- What does Falco do?

Use runtime detection, audit logs and network telemetry; isolate the pod, preserve evidence, rotate credentials and rebuild from a clean image.

```
kubectl label pod p quarantine=true --overwrite
```

```
kubectl get events -A | grep -i exec
```

Interview tip: Isolate, then investigate.

 Start here (official docs)

- [Docker documentation](#)

Q87 **How do Kubernetes RBAC and service accounts apply to workloads, and what are common mistakes?**

Security

 Spoken answer · ~56 s

RBAC defines Roles, which list allowed verbs on resources within a namespace, and ClusterRoles for cluster-wide or non-namespaced resources, connected to subjects by RoleBindings and ClusterRoleBindings. Pods act through a service account, and a token is mounted by default. Common mistakes are binding cluster-admin to applications or CI for convenience, wildcard verbs and resources, sharing the default service account across workloads, leaving tokens mounted on pods that never call the API, and granting read access to secrets broadly. The remedy is least privilege: one service account per workload, minimal verbs, namespace-scoped roles, automount disabled unless needed, and short-lived bound tokens. I audit with `kubectl auth can-i` and tools that analyse effective permissions, review role bindings regularly, and use impersonation to test before rollout. A compromised pod with broad RBAC can become a cluster compromise.

Likely follow-ups

- Why avoid the default service account?
- How do you test what an identity can do?

Roles grant verbs on resources; bindings attach them to users or service accounts. Avoid cluster-admin, wildcard rules and default service account tokens.

```
kubectl auth can-i list secrets --as=system:serviceaccount:app:web
```

```
automountServiceAccountToken: false
```

Interview tip: Test permissions with `can-i`.

 Official docs

- [Kubernetes: RBAC](#)
- [Kubernetes: Service accounts](#)

 Spoken answer · ~58 s

Pulling arbitrary images from public registries exposes the organisation to malware, outdated software and rate limits. I establish an approved path: a curated internal registry where upstream images are mirrored after scanning, with a process to request new base images. Scanners check for known vulnerabilities, secrets and misconfigurations on push and continuously afterwards, since new CVEs appear daily against old images. Policies set thresholds, for example blocking critical fixable vulnerabilities, with an exception process that has expiry dates. At admission, a policy engine only allows images from approved registries, requires digests or signatures, and rejects latest. Teams build on a small number of hardened, centrally maintained base images that are rebuilt regularly and distributed automatically. This reduces the attack surface, makes patching systematic, and gives security teams one place to answer which workloads contain a given vulnerable library.

Likely follow-ups

- Why scan continuously, not only at build?
- What does an exception process look like?

Use an approved internal registry, mirror and scan upstream images, enforce signatures and allowlists at admission, and rebuild regularly.

```
trivy image --severity HIGH,CRITICAL --ignore-unfixed app:1.4
```

```
cosign verify --key cosign.pub reg/app@sha256:...
```

Interview tip: Approve sources, not just images.

 Start here (official docs)

- [Docker documentation](#)

Q89 **What is the difference between a Deployment, ReplicaSet and Pod, and how does a rollout use them?**

Orchestration

 Spoken answer · ~58 s

A pod is the smallest deployable unit, a group of containers sharing network and storage. A ReplicaSet ensures a specified number of identical pods are running, replacing lost ones. A Deployment adds declarative updates on top: it owns ReplicaSets, one per pod template version. When I change the template, for example the image, the Deployment creates a new ReplicaSet and gradually scales it up while scaling the old one down according to the rolling update strategy. Old ReplicaSets are retained according to the revision history limit, which makes rollback just a matter of scaling a previous one back up. Because the Deployment controller reconciles these, manual edits to ReplicaSets are overwritten. Changes to labels or selectors need care since the selector is immutable. Understanding this chain explains why deleting a pod just creates another and why rollback is quick.

Likely follow-ups

- Why does deleting a pod create another?
- How does rollback use ReplicaSets?

A Deployment manages ReplicaSets, which manage Pods; each rollout creates a new ReplicaSet and scales old ones down.

```
kubectl get rs -l app=web
```

```
kubectl rollout history deploy/web
```

Interview tip: Never edit ReplicaSets directly.

 Official docs

- [Kubernetes: Deployments](#)

Q90 **How does the scheduler decide where to place a pod, and how do affinity, taints and tolerations influence it?**

Orchestration

 Spoken answer · ~56 s

Scheduling has two phases. Filtering removes nodes that cannot run the pod: insufficient requested CPU or memory, unmatched node selectors or affinity, intolerated taints, volume zone mismatches or port conflicts. Scoring ranks the remaining nodes using factors like resource balance, image locality and spreading, and the pod goes to the best. Node affinity and selectors attract pods to nodes with certain labels. Taints on nodes repel pods unless they have a matching toleration, which is used to dedicate nodes, such as GPU or system pools. Pod affinity and anti-affinity place pods near or away from others, and topology spread constraints distribute replicas evenly across zones or nodes, which is better for high availability at scale. When a pod stays pending, the FailedScheduling event explains which filters failed, guiding whether to add capacity or relax constraints.

Likely follow-ups

- What is the difference between a taint and affinity?
- How do spread constraints help availability?

It filters nodes by constraints then scores the rest; node affinity attracts, taints repel, anti-affinity and spread constraints distribute replicas.

```
kubectl describe pod p | grep -A10 Events
```

```
tolerations:
- key: dedicated
  operator: Equal
  value: gpu
  effect: NoSchedule
```

Interview tip: FailedScheduling events state the reason.

 Official docs

- [Kubernetes: Taints and tolerations](#)

 Spoken answer · ~58 s

ConfigMaps hold non-sensitive configuration and Secrets hold sensitive values; both can be exposed to pods as environment variables or mounted files. Secrets are only base64-encoded in the API, and by default stored unencrypted in etcd, so anyone with read access to the object or etcd can read them. To harden, I enable encryption at rest with a KMS provider, restrict RBAC so few identities can read secrets, limit who can exec into pods, and avoid putting secrets in environment variables when files are an option, since env vars can leak through process listings and crash dumps. A stronger pattern is using an external secret manager with an operator or CSI driver that syncs or mounts secrets at runtime, with rotation and audit logs. Never commit secrets to Git, even in encoded form; use sealed or encrypted formats for GitOps.

Likely follow-ups

- Why isn't base64 security?
- What are the benefits of an external secrets operator?

Both inject configuration; Secrets are base64-encoded, not encrypted by default, so enable encryption at rest, restrict RBAC and prefer external secret managers.

```
kubectl create secret generic db --from-literal=password=...
```

```
kubectl get secret db -o jsonpath='{.data.password}' | base64 -d
```

Interview tip: Base64 is encoding, not encryption.

Official docs

- [Kubernetes: Secrets](#)
- [Kubernetes: ConfigMaps](#)

 Spoken answer · ~59 s

Namespaces provide a scope for names and a unit for applying policy. Resource quotas cap the total CPU, memory, storage and object counts a namespace can consume, preventing one team from exhausting the cluster. Limit ranges set default requests and limits for containers that omit them and enforce minimum and maximum sizes, so quotas work smoothly. For real multi-tenancy I add RBAC scoped to the namespace, network policies for default-deny traffic, Pod Security admission, separate service accounts and, where tenants are untrusted, node pools or even separate clusters, because namespaces share the kernel, control plane and some resources. Cost allocation uses labels and namespace-level reporting. Governance such as naming conventions, owner labels and automatic provisioning through templates keeps things consistent. The level of isolation chosen depends on trust: internal teams can share a cluster, while external customers often need stronger separation.

Likely follow-ups

- Why are quotas and limit ranges used together?
- When do you need separate clusters?

Namespaces separate teams; quotas cap aggregate resources; limit ranges set defaults and bounds per container.

```
kubectl create quota team-a --hard=cpu=20,memory=64Gi,pods=100 -n team-a
```

```
kubectl describe limitrange -n team-a
```

Interview tip: Namespaces are not a security boundary alone.

 Official docs

- [Kubernetes: Multi-tenancy](#)
- [Kubernetes: Resource Quotas](#)
- [Kubernetes: LimitRange](#)

 Spoken answer · ~55 s

A custom resource definition extends the Kubernetes API with a new type, such as a Postgres cluster. An operator is a controller that watches those custom resources and continuously reconciles the real world to match, creating pods, configuring replication, taking backups and handling failover, encoding operational knowledge in software. Helm is a package manager: it renders templates with values into manifests and installs them as a release, with upgrade and rollback, but after install it does not react to changes or failures. They complement each other: it is common to install an operator with Helm, and then manage instances with custom resources. I use Helm for packaging and deploying applications, and operators for complex stateful or lifecycle-heavy systems. Writing an operator is a significant maintenance commitment, so I prefer proven community operators.

Likely follow-ups

- Can Helm and operators be used together?
- What does a reconcile loop do?

A CRD adds a new resource type; an operator is a controller that manages it continuously. Helm templates and installs manifests but does not run ongoing logic.

```
kubectl get crd
```

```
helm template web ./chart | kubectl diff -f -
```

Interview tip: Helm installs; operators operate.

 Official docs

- [Kubernetes: Custom Resources](#)
- [Helm: Chart best practices](#)
- [Helm documentation](#)

Spoken answer · ~58 s

After building and testing, the pipeline pushes the image tagged with the commit and records its digest. Then it scans for vulnerabilities and misconfigurations, failing on policy violations. If it passes, the pipeline signs the digest, ideally keyless with the pipeline's identity, and attaches an SBOM and provenance. Promotion between environments or registries copies or references the same digest, so what was tested is exactly what runs, and the signature remains valid. In the cluster, an admission policy verifies signatures and optionally attestations, rejecting anything not produced by the official pipeline. I keep production credentials out of build jobs and use separate registries or repositories per stage with different write permissions. Retention policies clean up old images, but images that are currently deployed or needed for rollback are protected. This creates a traceable chain from commit to running container.

Likely follow-ups

- What does signing prove?
- What do you check at admission?

Scan, sign, promote

Build once, scan, sign the digest, push to a registry, and promote the same digest across environments, verifying at admission.

```
trivy image app@sha256:...
```

```
cosign sign app@sha256:...
```

```
crane copy dev/app@sha256:... prod/app@sha256:...
```

Interview tip: Promote by digest.

Start here (official docs)

- [Docker documentation](#)

 Spoken answer · ~59 s

Integration tests are more realistic with real dependencies such as databases and message brokers than with mocks. In CI I start them in containers, either through Docker Compose with health checks and the wait option, or through Testcontainers, a library that starts containers from inside the test code, picks random ports, and cleans up automatically. The important part is waiting until the service is truly ready, using health checks or log-based waits, not just until the container has started, which is a common source of flaky tests. Each test run uses fresh containers and data, so tests do not depend on previous state, and I tear everything down with volumes removed. Pinning image versions to match production avoids surprises. To save time, containers can be shared per test suite with data reset between tests, and I cache images in the CI environment.

Likely follow-ups

- Why wait for health, not start?
- What does Testcontainers add over Compose?

Start real dependencies in disposable containers, wait for readiness, run tests, and tear down.

```
docker compose up -d --wait
```

```
docker compose down -v
```

Interview tip: Wait for health, not just start.

Start here (official docs)

- [Docker documentation](#)

 Spoken answer · ~64 s

With ARM nodes becoming cost-effective, images often need to support both amd64 and arm64. Buildx builds for multiple platforms and pushes a manifest list, an index that points to the image for each architecture, so clients automatically pull the right one. Cross-building can use QEMU emulation, which is simple but slow for compilation-heavy builds, or native builders on machines of each architecture combined into one build, which is faster. Cross-compilation inside the Dockerfile, using the target platform arguments, is the fastest for languages like Go and Rust. I make sure every dependency and base image supports both architectures and test the arm64 image, not just build it, since subtle differences exist, for example native libraries or performance characteristics. In the cluster, node selectors or the mixed-architecture manifest ensure scheduling works correctly. I write the reasoning into the runbook so the on-call engineer understands it. That keeps the behaviour predictable for everyone on the team.

Likely follow-ups

- What is a manifest list?
- Why prefer native builders?

Use buildx with QEMU emulation or native builders to produce a manifest list so each platform pulls the right image.

```
docker buildx build --platform linux/amd64,linux/arm64 -t reg/app:1.4 --push .
```

```
docker buildx imagetools inspect reg/app:1.4
```

Interview tip: Native builders beat emulation for speed.

 Start here (official docs)

- [Docker documentation](#)

 Spoken answer · ~58 s

I tag every build with the commit SHA, which is unique and traceable, and add human-friendly tags like semantic versions for releases. Tag immutability prevents anyone from overwriting a tag, so version 1.4.2 always means the same image. Moving tags like latest or environment names are avoided for deployments, and manifests use digests. Retention policies prevent unbounded storage growth: delete untagged images after a few days, keep the last N builds of a branch, expire pull request images quickly, and retain release images for the support period. Lifecycle rules must not delete images currently deployed or needed for rollback, so I align them with the deployment history. Access control separates who can push, typically only the pipeline, from who can pull. Vulnerability scanning and replication to other regions are enabled where needed. Good hygiene lowers cost and makes audits simpler.

Likely follow-ups

- Why enable tag immutability?
- What must retention never delete?

Tag immutably with commit SHA and semantic versions, protect release tags, and expire old untagged or unused images.

```
gcloud artifacts repositories set-cleanup-policies repo --policy=policy.json
```

```
aws ecr put-lifecycle-policy --repository-name app --lifecycle-policy-text  
file://policy.json
```

Interview tip: Enable tag immutability.

 Official docs

- [Prometheus: Storage \(TSDB\)](#)

Q98 **How do you deploy the same image to multiple environments with different configuration using Helm or Kustomize?**

CI/CD & registry

 Spoken answer · ~59 s

The principle is one artifact and several configurations. With Helm, a single chart takes a values file for each environment covering replicas, resources, ingress hosts and feature settings, plus the image reference. With Kustomize, a base holds the common manifests and overlays patch environment-specific fields. These files live in Git, so every difference between environments is reviewed and auditable. Promotion means changing the image digest or chart version in the next environment's values via a pull request or an automated bot, and the deployment tool applies it. I keep the differences minimal and explicit to avoid environment-specific surprises, validate rendered output with a linter and policy checks in CI, and run `helm diff` or `kubectl diff` to preview. Secrets come from an external store, not values files. This yields repeatable, reviewable deployments and a clear answer to what is different in production.

Likely follow-ups

- How do you preview changes before applying?
- How do you handle secrets in values?

Keep the chart or base generic and supply per-environment values or overlays; promote only the image reference and config changes via Git.

```
helm upgrade --install web ./chart -f values/prod.yaml --set image.digest=sha256:...
```

```
kubectl apply -k overlays/prod
```

Interview tip: Config differences should be small and visible.

 Official docs

- [Kusto Query Language \(KQL\)](#)
- [Helm documentation](#)
- [Kustomize reference](#)

Q99 **How do you design container logging so logs survive pod restarts and remain searchable?**

Observability

 Spoken answer · ~61 s

The twelve-factor approach is to write logs as an event stream to standard output and error, and let the platform collect them. The container runtime stores them on the node with rotation, and a node-level agent such as Fluent Bit or the Cloud Logging agent tails them, enriches each line with pod, namespace, node and labels, and ships them to a central system like Loki, Elasticsearch or a cloud service. This way logs survive pod restarts and node loss, and developers can search across all replicas. Using structured JSON logs with consistent fields such as level, message, trace identifier and request identifier makes filtering and correlation much easier. I configure rotation limits to avoid filling node disks, set retention and sampling to control cost, and avoid logging secrets and personal data. For apps that insist on writing files, a sidecar can stream those files to stdout.

Likely follow-ups

- Why log to stdout?
- What fields should every log line have?

Write to stdout and stderr in structured form; a node agent ships them to a central store with metadata, retention and rotation.

```
kubectl logs pod --previous
```

```
docker run --log-opt max-size=10m --log-opt max-file=3 img
```

Interview tip: Do not write log files inside containers.

 **Start here (official docs)**

- [Docker documentation](#)

Q100 **Which container and Kubernetes metrics should you alert on, and which only on dashboards?**

Observability

 Spoken answer · ~58 s

I page on symptoms that affect users or will soon: elevated error rate and latency against SLOs, pods not ready for longer than a threshold, crash looping, persistent volume predicted to fill within hours, node not ready, and failing deployments or jobs. Resource saturation alerts, like memory approaching the limit, are useful as tickets or warnings because they indicate future problems but are not always user-facing. Detailed data, such as per-container CPU, network throughput, restart counts and throttling, belongs on dashboards for investigation, not on pagers. Key metric sources include cAdvisor and kubelet for container resources, kube-state-metrics for object state, node exporter for hosts, and application metrics for RED signals. I tune alerts using burn rates and durations to avoid flapping, and include a runbook link and a dashboard with every alert so the responder has immediate context.

Likely follow-ups

- Which alerts should page someone?
- What does kube-state-metrics provide?

Alert on symptoms such as errors, latency and saturation risks; keep resource detail like per-container CPU on dashboards.

```
kube_pod_container_status_restarts_total
```

```
container_memory_working_set_bytes / kube_pod_container_resource_limits
```

Interview tip: Page on user impact.

 Official docs

- [Grafana documentation](#)

Q101 How do you use kubectl debug and ephemeral containers to troubleshoot a running pod?

Observability

 Spoken answer · ~55 s

Production images are often minimal or distroless with no shell, so I cannot exec into them with useful tools. Ephemeral containers solve this: kubectl debug adds a temporary container to the running pod, sharing its network namespace and, with the target flag, its process namespace, so I can see processes, check network connectivity, inspect files through the process root path and run tools like curl, dig, tcpdump or strace. It does not restart the pod or change the original container. I can also create a copy of a pod with a modified command or image to reproduce a crash, and debug a node by starting a privileged pod with the host filesystem mounted. Access is controlled by RBAC and should be audited, since it grants powerful capabilities. I remove debug artifacts afterwards.

Likely follow-ups

- What does the target flag do?
- Why restrict access to debug?

Attach a temporary container with debugging tools to an existing pod's namespaces, without restarting or changing the image.

```
kubectl debug -it pod/web --image=nicolaka/netshoot --target=app
```

```
kubectl debug node/n1 -it --image=busybox
```

Interview tip: Works for distroless images.

Official docs

- [Kubernetes: Debug running Pods \(kubectl debug\)](#)
- [Terraform: Sensitive data in state](#)

Q102 How do you set up readiness, liveness and startup probes correctly to avoid restart storms?

Observability

 Spoken answer · ~58 s

Each probe has a distinct job. The startup probe gives slow-starting applications time to initialise, and until it succeeds, liveness and readiness checks are held off. The readiness probe decides whether the pod receives traffic; failing it removes the pod from service endpoints without restarting it, which is right for temporary overload or dependency problems. The liveness probe restarts the container when it is stuck beyond recovery, such as a deadlock. The classic mistake is making liveness depend on downstream systems like the database: when the database slows down, every pod fails liveness, restarts and the whole service goes down, amplifying the incident. Liveness should check only the process's own health, be cheap, and have generous thresholds and timeouts. Readiness can include critical dependencies. I also tune timeouts to realistic response times under load to avoid false failures.

Likely follow-ups

- Why shouldn't liveness check the database?
- What does a startup probe protect?

Startup probe protects slow starts, readiness controls traffic, liveness restarts only deadlocked processes; do not check dependencies in liveness.

```
startupProbe:
  httpGet: {path: /healthz, port: 8080}
  failureThreshold: 30
  periodSeconds: 5
livenessProbe:
  httpGet: {path: /live, port: 8080}
```

Interview tip: Liveness should be cheap and local.

Official docs

- [Kubernetes: Liveness, readiness and startup probes](#)

Q103 How do you trace a request across containers and correlate traces with logs and metrics?

Observability

 Spoken answer · ~58 s

Each request carries a trace context, standardised as the W3C traceparent header, which every service reads and forwards, so spans from different containers join into one trace. I instrument services with OpenTelemetry, using auto-instrumentation where available and manual spans for important operations, and send data to a collector that batches, samples and exports to a backend. To correlate with logs, the logging library adds the current trace and span identifiers to each log line, so I can jump from a slow trace to the exact logs, and vice versa. For metrics, exemplars attach trace identifiers to histogram samples, linking a latency spike to example traces. Sampling is important for cost: head sampling is simple, tail sampling in the collector keeps slow or error traces. Consistent service naming and resource attributes like pod and version make analysis much easier.

Likely follow-ups

- How do traces and logs get linked?
- Why use tail sampling?

Propagate trace context headers, instrument with OpenTelemetry, put trace IDs in logs, and use exemplars to link metrics to traces.

```
traceparent: 00-<trace-id>-<span-id>-01
```

```
OTEL_EXPORTER_OTLP_ENDPOINT=http://collector:4317
```

Interview tip: Trace ID in every log line.

 **Start here (official docs)**

- [Docker documentation](#)