K8s & OpenShift: Building Workloads Right

2026-07-01

Container image hygiene, health probes, graceful shutdown, and securityContext — what every workload running in Kubernetes has to get right. Part 2 of 7.

Part 1 covered the business case and organisational framing. This part goes inside the pod: image hygiene, security context, health probes, graceful shutdown, and the application design choices that determine whether a workload actually fits the platform.

Series navigation: - **Full guide:** The Kubernetes & OpenShift Best Practices Guide (2026 Edition) - Part 1 — The Big Picture - **Part 2 — Building workloads right (this post)** - Part 3 — Resource management deep dive - Part 4 — Scaling & resilience - Part 5 — Security - Part 6 — Day-2 operations & GitOps - Part 7 — Compliance (regulatory frameworks, audit logging, log retention)

The contract between your application and the platform

Kubernetes has an implicit contract with every workload it runs. The platform will start your container, route traffic to it, restart it when it crashes, and reschedule it when a node goes down. In return, it expects your application to behave in specific ways: to report its own health, to handle termination signals gracefully, and not to assume that local state survives a restart.

Applications that honour this contract are resilient by default. Applications that ignore it work fine until the first rolling update, node eviction, or scaling event — at which point the platform cannot help them.

What you actually deploy — Deployment, StatefulSet, and friends

Everything in this part talks about "the pod" — the running container and its runtime behaviour. In production, a bare `Pod` object is almost never created directly. Instead, a controller object wraps a pod template and manages the pods created from it. Which controller you pick depends on the workload:

- **Deployment** — the default choice for stateless workloads. Manages a ReplicaSet, supports rolling updates, and is what most of this series assumes unless stated otherwise.
- **StatefulSet** — for workloads that need a stable identity, stable storage, or an ordered rollout — databases, message brokers, anything where "pod 0" and "pod 1" are not interchangeable.
- **DaemonSet** — runs exactly one pod per matching node, without a replica count. Log shippers, CNI agents, and node-level monitoring exporters use this pattern.
- **Job / CronJob** — run-to-completion workloads, either once or on a schedule, instead of a long-running service.

All four wrap the same thing: a pod template under `spec.template.spec`. That is where `securityContext`, probes, `resources`, and everything else covered in Parts 2 through 5 actually lives:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp
spec:
  replicas: 2
  selector:
    matchLabels:
      app: myapp
  template:              # the pod template
    metadata:
      labels:
        app: myapp
    spec:                # everything else in this series lives here
      containers:
        - name: myapp
          image: registry.example.com/myapp:1.4.2
```

The YAML snippets throughout this series show just that inner fragment, so they apply unchanged regardless of which controller kind wraps them. How these manifests get packaged — a Helm chart, a Kustomize overlay — and delivered to the cluster is a separate concern, covered in Part 6.

Container image hygiene

Choose your base image carefully

Alpine is tempting: minimal size, fast pulls. The catch is that Alpine uses musl libc instead of glibc, which could raise problems that tend to surface only in production. The `TZ` environment variable does not work for timezone configuration without explicitly installing `tzdata`. Some native libraries behave differently, and certain Java, Python, and Go packages assume glibc. Ubuntu and Red Hat Universal Base Image (UBI) use glibc and have proven reliable across a wide range of workloads over time.

> **i UBI in the OpenShift context**
>
> Red Hat UBI is the base image Red Hat supports and continuously scans for CVEs as part of the OpenShift subscription. Teams running on OpenShift or ROSA/ARO who want supported, scanned base images will generally reach for UBI rather than upstream Debian or Ubuntu.

Shell-less base images — distroless, scratch — shrink the attack surface to an absolute minimum: no shell, no package manager, nothing beyond the runtime and your binary. From a security standpoint this is appealing. The operational cost is that debugging a running container becomes very difficult without a shell. `kubectl exec` drops you into nothing. Teams with mature observability

— structured logs, distributed traces, rich metrics — can afford this tradeoff. Teams that still rely on `kubectl exec` to diagnose live issues will find shell-less images a significant operational burden.

Two approaches work well in practice:

Option 1 — include tools directly in the production image. Use a supported runtime image based on Ubuntu or UBI and install at minimum the following in the same layer as the application:

- `curl` — for testing connectivity and HTTP endpoints from inside the pod
- `ps` — for inspecting running processes
- `dig` — for diagnosing DNS resolution, which is one of the most common causes of pod connectivity problems in Kubernetes

Option 2 — maintain a parallel `-debug` image. Keep the production image minimal and build a second image tagged `-debug` that extends the same production image and adds only the debugging tools. Both images are built from the same Containerfile using multi-stage builds, so the application layer is always identical — no drift between what runs in production and what you debug with.

```
# Shared application build
FROM golang:1.23 AS builder
WORKDIR /app
COPY . .
RUN CGO_ENABLED=0 go build -o server .

# Production image — minimal
FROM ubuntu:24.04 AS production
COPY --from=builder /app/server /server
USER 1001
ENTRYPOINT ["/server"]

# Debug image — same base, adds tooling
FROM production AS debug
USER root
RUN apt-get update && apt-get install -y --no-install-recommends \
    curl dnsutils procps \
    && rm -rf /var/lib/apt/lists/*
USER 1001
```

Tag and push both in CI:

```
myapp:1.4.2        ← production
myapp:1.4.2-debug  ← identical app, debug tools included
```

When a production issue needs interactive investigation, temporarily swap the image tag in the deployment or use `kubectl debug` with the `-debug` variant. After the session the original image tag goes back in — no permanent change to the production workload.

Pin exact versions — never `latest`

`latest` is not a version tag. It is an alias that resolves to "whatever the registry had when I last pulled", which changes without notice. In production this means non-reproducible deployments and silent breakage when an upstream maintainer releases an incompatible change.

```
# unpinned — avoid in production
image: nginx:latest

# pinned to patch version — minimum acceptable
image: nginx:1.27.0

# pinned to digest — fully reproducible, no moving target
image:
nginx:1.27.0@sha256:a484819eb60211f5299034ac80f6a681b06f89e65866ce91f356ed7c72af059c
```

Your image scanning pipeline (covered in Part 5) can flag pinned versions that have known CVEs and prompt an intentional upgrade decision rather than a silent one.

Use multi-stage builds

Multi-stage builds separate the build environment from the runtime environment. The final image contains no compiler, no package manager, and no build tools — only what the process needs at runtime.

```
# Build stage — fat image with build tooling
FROM golang:1.23 AS builder
WORKDIR /app
COPY . .
RUN CGO_ENABLED=0 go build -o server .

# Runtime stage — minimal image, only the binary
FROM ubuntu:24.04
RUN apt-get update && apt-get install -y --no-install-recommends \
    curl \
    dnsutils \
    procps \
    && rm -rf /var/lib/apt/lists/*
COPY --from=builder /app/server /server
USER 1001
ENTRYPOINT ["/server"]
```

Do not run as a specific named user

Do not bake a specific named user into the image (e.g., creating and switching to `appuser`). Instead, leave user specification to the container runtime via `securityContext.runAsUser`. This gives the platform the flexibility to enforce the runtime identity it requires — which matters especially in OpenShift, where the platform assigns a random UID from a namespace-specific range. An image that insists on a specific named user will fail under OpenShift's default SCC.

securityContext — harden by default

The default Kubernetes security posture for a pod is permissive: root user, all Linux capabilities retained, writable filesystem, privilege escalation allowed. None of these defaults are appropriate for production workloads.

Apply this baseline at the container level:

```yaml
securityContext:
  runAsNonRoot: true
  runAsUser: 1001
  runAsGroup: 1001
  allowPrivilegeEscalation: false
  readOnlyRootFilesystem: true
  capabilities:
    drop:
      - ALL
```

And at the pod level:

```yaml
spec:
  securityContext:
    runAsNonRoot: true
    seccompProfile:
      type: RuntimeDefault
```

A few details worth knowing:

- **`runAsNonRoot: true`** causes Kubernetes to reject the pod at admission if the image's declared user is root — a useful early failure that surfaces image misconfigurations before they reach production.
- **`readOnlyRootFilesystem: true`** is a strong default. If your application needs a writable path (temp files, PID files, caches), mount an `emptyDir` volume for that specific path rather than making the whole filesystem writable.
- **`capabilities.drop: [ALL]`** removes all Linux capabilities. Add back only what the process genuinely needs — `NET_BIND_SERVICE` if it binds to ports below 1024, for example. Most application servers do not need any capabilities.

> **i OpenShift enforces this by default**
>
> OpenShift's Security Context Constraints (SCCs) enforce non-root and drop-all-capabilities on most workloads without any configuration. Several items on this checklist are simply the OpenShift baseline. The `restricted-v2` SCC, which applies to new workloads by default, also sets `seccompProfile: RuntimeDefault`. Pods that do not comply with the applied SCC are rejected at admission — which surfaces misconfigurations early, at the cost of requiring SCC-aware image design from the start.

Health probes — the three kinds

Kubernetes uses probes to decide what to do with a pod. Getting them wrong is one of the most common sources of production incidents: missing probes lead to traffic being routed to pods that are not ready; overly aggressive liveness probes cause restart cascades under load.

There are three distinct probe types, each answering a different question.

Startup probe — "has it finished initialising?"

The startup probe runs only during pod initialisation, and only until it succeeds once. While it is running, liveness and readiness probes are suspended. This is the right tool for slow-starting applications — JVMs loading large classpaths, services running database migrations on startup.

```
startupProbe:
  httpGet:
    path: /health/startup
    port: 8080
  failureThreshold: 30
  periodSeconds: 10
```

This gives the application up to 300 seconds to start (`periodSeconds × failureThreshold` = `10 × 30` = 300). Without a startup probe, you set `initialDelaySeconds` on the liveness probe instead — a static delay that either wastes time on fast starts or kills slow-starting pods before they finish.

Liveness probe — "is it still alive?"

The liveness probe **runs continuously** after startup, at every `periodSeconds` interval. If it fails `failureThreshold` times in a row, Kubernetes kills the container and restarts it. Use it only to detect states the application cannot recover from on its own: deadlocks, corrupted state, hung threads.

```
livenessProbe:
  httpGet:
    path: /health/live
    port: 8080
  periodSeconds: 10
  failureThreshold: 3
```

The liveness check should be minimal: is the process up and responding? It must not check downstream dependencies. If a database is temporarily unreachable, you want the pod to stay running and return errors — not restart in a loop that accomplishes nothing and churns the connection pool.

Readiness probe — "can it serve traffic right now?"

The readiness probe controls whether a pod **receives traffic** from the Service. A failing readiness probe removes the pod from the endpoint list — traffic stops going to it — **but the pod is not restarted**. Use it to signal temporary unavailability: cache warming, connection pool saturation, a dependency temporarily unreachable.

```yaml
readinessProbe:
  httpGet:
    path: /health/ready
    port: 8080
  periodSeconds: 5
  failureThreshold: 3
```

Unlike liveness, readiness checks may include dependency checks. “I cannot reach the database” is a valid reason to stop receiving traffic.

Keep all probes lightweight

Probe endpoints must return quickly — under 100 ms is a reasonable target — and must not trigger expensive operations. A probe that queries a database or calls an external service will become unreliable under load, exactly when reliable probes matter most.

Java probe endpoints

Spring Boot Actuator exposes `/actuator/health/liveness` and `/actuator/health/readiness` out of the box when `management.health.probes.enabled=true`. Tomcat can use the `HealthCheckValve` to expose a lightweight endpoint without additional application code. Both are designed to be used directly as probe targets.

⚠ Long initialisation times — increase timeouts or the pod will never come up

Applications with slow startup — JVMs loading large classpaths, services running schema migrations, or anything that takes more than a few seconds to become ready — will be killed by Kubernetes mid-initialisation if the probe timeouts are left at their defaults. The result is a restart loop: the pod is killed, restarted, killed again, and never reaches a running state.

Use a startup probe with a `failureThreshold` and `periodSeconds` that together cover the worst-case startup time, and leave `initialDelaySeconds` off the liveness probe entirely:

```yaml
startupProbe:
  httpGet:
    path: /health/startup
    port: 8080
  failureThreshold: 60   # 60 × 10s = 10 minutes maximum startup
window
  periodSeconds: 10

livenessProbe:
  httpGet:
    path: /health/live
    port: 8080
  periodSeconds: 10
```

```
  failureThreshold: 3
  # no initialDelaySeconds needed — startup probe protects the window
```

Measure the actual startup time under load (not on your laptop) and add a safety margin before setting `failureThreshold`.

Graceful shutdown — the SIGTERM contract

When Kubernetes terminates a pod — during a rolling update, node drain, or scale-down — it sends `SIGTERM` to PID 1 in the container. The application has `terminationGracePeriodSeconds` (default: 30 seconds) to:

1. Stop accepting new connections
2. Finish in-flight requests
3. Flush write buffers, close database connections, release locks, and exit

If the process is still running after the grace period, Kubernetes sends `SIGKILL` — immediate termination, no cleanup.

The PID 1 signal forwarding problem

When a container starts with a shell as the entrypoint, the shell becomes PID 1. Kubernetes sends `SIGTERM` to PID 1. Most shells do not forward signals to child processes, so the application never receives the signal and is killed by `SIGKILL` after 30 seconds on every single rolling update.

The fix is to make the application process PID 1 by using the exec form of `CMD` or `ENTRYPOINT`:

```
# Shell form — shell is PID 1, does not forward signals
CMD ["sh", "-c", "java -jar app.jar"]

# Exec form — application is PID 1, receives SIGTERM directly
CMD ["java", "-jar", "app.jar"]
```

If a wrapper script is genuinely needed, use `exec` as the final statement to replace the shell process with the application:

```
#!/bin/sh
# environment setup, config rendering, etc.
exec java -jar app.jar
```

For containers that run more than one process — a main application plus a sidecar, a log shipper, or a controller — a dedicated init daemon is a cleaner option than hand-wiring signal forwarding yourself. vigil-rs, `tini`, and `dumb-init` are the minimal options: they run as PID 1, reap zombie processes, and forward signals to their child. They are the right choice when you have a single main process that needs a safe init wrapper.

For multi-process containers with health checks, restart policies, and per-service stop signals, vigil-rs is an option worth knowing about. It is a Rust PID 1 / container init daemon with native zombie reaping, per-service YAML configuration, independent restart policies, HTTP/TCP/exec health checks, and a JSON REST API over a Unix socket. It handles the signal forwarding problem

correctly and also gives each supervised process its own configurable stop signal and kill delay — so `SIGUSR1` for a graceful drain and 30 seconds before `SIGKILL` is a one-line config entry, not a wrapper script:

```
services:
  myapp:
    command: /usr/local/bin/myapp
    startup: enabled
    stop-signal: SIGUSR1
    kill-delay: 30s
    on-failure: restart
```

preStop hook

There is a race between Kubernetes sending `SIGTERM` and the kube-proxy or cloud load balancer finishing the removal of the pod's IP from the endpoint list. Requests can arrive in the gap. A short `preStop` sleep gives the load balancer time to propagate the endpoint removal before the application starts shutting down:

```
lifecycle:
  preStop:
    exec:
      command: ["/bin/sh", "-c", "sleep 5"]
terminationGracePeriodSeconds: 60
```

Increase `terminationGracePeriodSeconds` proportionally if your application needs more than the default 30 seconds to drain.

Application design for the platform

Beyond the technical configuration, a few design decisions determine how well a workload fits the K8s operating model. These are application-level choices, not platform configuration.

Externalise all state. Kubernetes can reschedule a pod to a different node without warning. Local disk state and in-memory session data are lost when that happens. Session state belongs in an external store. If the application cannot run statelessly, it needs persistent volumes — for example RWO (`ReadWriteOnce`), which can only be mounted by one node at a time, unlike RWX-capable backends such as NFS or CephFS that can be shared across nodes. Needing a persistent volume does not by itself rule out `replicas > 1`; it depends on the access mode and on a supported clustering mechanism existing for that data.

Multiple replicas require cluster-readiness. If you run more than one instance, the application must handle concurrent instances gracefully: no in-process locks that assume a single writer, no write conflicts between pods hitting the same data simultaneously. Stateless HTTP services are naturally cluster-ready. Scheduled jobs, message consumers, and stateful services need explicit thought — and often leader-election or exactly-once semantics.

Log to stdout/stderr. Kubernetes captures stdout and stderr automatically and routes them to whatever log aggregation stack is in place. Writing logs to files inside the container requires a

sidecar shipper, risks filling the container filesystem, and loses logs on pod restart. For existing applications that only write to files, a sidecar log-shipper is a workable solution — for new applications, stdout is the correct default.

Expose a Prometheus metrics endpoint. The `/metrics` endpoint in Prometheus text format is the de-facto observability standard for the Kubernetes ecosystem. Client libraries exist for Java, Go, Python, Node.js, Ruby, and most other runtimes. Exposing metrics from the start costs almost nothing and provides the foundation for alerting, dashboards, and custom-metric autoscaling (HPA can scale on Prometheus metrics via the metrics adapter).

Instrument with OpenTelemetry. Metrics alone do not show why a specific slow request was slow. Adding the OpenTelemetry SDK to emit traces costs little at build time and pays off the first time a request needs to be followed across service boundaries. This series covers the collector setup and trace backends in Part 6.

> **i Planning for more than one replica? Think cluster-first.**
>
> As soon as `replicas > 1` is on the table, the application design needs to be reviewed through a cluster lens — before the second instance is ever started. The key questions:
>
> - **State:** is all mutable state externalised (database, cache, object storage)? Any state kept in memory or on local disk is invisible to the other replicas and lost on restart.
> - **Sessions:** are user sessions stored externally (Redis, DB), or does the load balancer need sticky sessions as a workaround?
> - **Locks and scheduling:** does the application use in-process locking, timers, or scheduled tasks that assume a single instance? These need leader-election or an external coordination mechanism (e.g. a database-backed distributed lock, a Kubernetes `Lease` object).
> - **Writes:** can two instances write to the same data simultaneously without conflicts? If not, a clustering mechanism or a single-writer pattern is required.
>
> A stateless HTTP service typically needs no changes. Everything else — job schedulers, message consumers, stateful services — needs an explicit answer to each of these before scaling beyond one replica.

Next: Part 3 covers resource management — how requests and limits actually work, cgroups v1 versus v2, QoS classes, and what happens when a container hits its memory limit.

Partly used sources — specific references for this part:

- Best practices for running apps in Kubernetes — Part 1 — Palark (2021)
- Best practices for running apps in Kubernetes — Part 2 — Palark (2021)
- Kubernetes Best Practices I Wish I Had Known Before — Pulumi (2025, updated 2026)
- 17 container security best practices for 2026 — Sysdig (2026)
- Kubernetes production readiness checklist — learnkube.com (2026)
- Kubernetes Best Practices for Safer, More Reliable Clusters — Komodor (2026)