The Kubernetes & OpenShift Best Practices Guide (2026 Edition)

2026-07-01

A seven-part field guide to running Kubernetes and OpenShift in production in 2026 — workloads, resources, scaling, security, Day-2 operations, and compliance.

This is the index and reading guide for a seven-part series on running Kubernetes and OpenShift in production, written from the vantage point of 2026 and eight years of hands-on cluster design, Day-2 operations, and production debugging. Each part stands on its own; together they cover a workload's whole life on the platform — from the image it ships in to the audit trail it leaves behind.

The series is deliberately grounded. Every practice is written to apply to vanilla Kubernetes first, with the OpenShift-specific mechanism (Security Context Constraints, the Cluster Version Operator, Routes, the built-in monitoring stack) called out where it differs. Two EU-native managed platforms — OVHcloud Managed Kubernetes Service (MKS) and STACKIT Kubernetes Engine (SKE) — recur throughout as concrete, empirically checked reference points, because "it depends on your provider" is only useful if someone actually names the providers and the differences.

This series is fairly broad, but it is not, and cannot be, complete. Every use case carries its own constraints — a regulatory obligation, a legacy dependency, a team's operational maturity, a platform quirk not covered here — that can turn a general practice into the wrong call for that specific situation. Treat this guide as a starting point, not a checklist to apply unmodified: it is always worth discussing your own use case with people who know it before committing to any of the choices below.

How to read this guide

If you are deciding *whether* to run Kubernetes at all, start with Part 1. If you already run it and want to harden what you have, jump straight to the chapter that matches your gap — the summaries below say what each part decides for you. The parts are ordered to follow a workload outward: what it is, what it consumes, how it scales, how it is secured, how it is operated, and how that all becomes compliance evidence.

- Part 1 — The Big Picture: does a container platform earn its keep, and when
- Part 2 — Building Workloads Right: images, probes, graceful shutdown, securityContext
- Part 3 — Resource Management Deep Dive: requests, limits, QoS, cgroup v2
- Part 4 — Scaling & Resilience: HPA, VPA, PDBs, taints, rolling updates
- Part 5 — Security: RBAC, Pod Security Admission, NetworkPolicy, supply chain, secrets
- Part 6 — Day-2 Operations & GitOps: GitOps, observability, runtime security, FinOps, upgrades

- Part 7 — Compliance: NIS2, DORA, PCI-DSS, HIPAA, CRA, audit logging, retention

Part 1 — The Big Picture

Audience: decision makers and platform leads.

Kubernetes in 2026 is boring infrastructure, and that is the point. The interesting question is no longer "should we adopt containers?" but "do we run this well enough to justify the operational surface it adds?" Part 1 frames the decisions that happen before anyone writes a YAML manifest: when a container platform pays for itself, when a simpler runtime would serve better, and where OpenShift's opinionated defaults change the calculation. It is the one deliberately non-technical part — aimed at the person who has to defend the platform choice, not just operate it.

Key takeaways:

- Kubernetes is a means, not an outcome; the decision is about operational maturity, not technology fashion.
- OpenShift trades flexibility for a hardened, supported baseline — valuable when compliance and support contracts matter, overhead when they do not.
- The practices in the rest of the series apply regardless of distribution — Rancher, Tanzu, or vanilla — even though the series is written against Kubernetes and OpenShift specifically.

Read Part 1 — The Big Picture →

Part 2 — Building Workloads Right

Audience: application developers.

Kubernetes has an implicit contract with every workload: it will start, route to, restart, and reschedule your container, and in return it expects the application to report its own health, handle termination signals, and not assume local state survives. Part 2 is what goes inside the pod — choosing a base image, pinning versions, running as non-root with a locked-down `securityContext`, wiring liveness/readiness/startup probes so the platform can actually help, and handling `SIGTERM` so a rolling update does not drop traffic. It also sets the object-hierarchy foundation the rest of the series builds on: you deploy a Deployment, StatefulSet, DaemonSet, or Job — the pod template under `spec.template.spec` is where everything else lives.

Key takeaways:

- Pin exact image versions; treat a debug variant as a separate tag built from the same base, not a shell added to production.
- The three probe types answer three different questions — conflating them causes restart loops and dropped traffic.
- A container that ignores `SIGTERM` gets `SIGKILL` after the grace period; graceful shutdown is a correctness requirement, not a nicety.
- Harden at the container level by default: non-root UID, `allowPrivilegeEscalation: false`, read-only root filesystem, drop all capabilities.

Read Part 2 — Building Workloads Right →

Part 3 — Resource Management Deep Dive

Audience: application developers and platform engineers.

Requests and limits are the two numbers that decide where a pod lands and what happens when it misbehaves — and they do very different jobs. Part 3 explains how the scheduler uses requests for placement, how limits are enforced through the cgroup hierarchy (CPU is throttled, memory is OOM-killed), how the three Quality-of-Service classes fall out of the request/limit combination, and how `ResourceQuota` and `LimitRange` set boundaries at the namespace level. It also covers the container-awareness traps — the JVM and cgroup v2 — that make "why did my pod get OOM-killed at half its limit?" a recurring incident.

Key takeaways:

- Set CPU requests always: the scheduler needs them for placement and the HPA divides usage by the request to compute utilisation.
- Memory limits are enforced by the kernel with an OOM kill; CPU limits only throttle — treat the two incompressible-vs-compressible resources differently.
- `LimitRange` defaults apply per container, not per pod — a multi-container pod can silently multiply its footprint or lose its QoS class.
- Neither OVH MKS nor STACKIT SKE exposes the kubelet flags (`cpuManagerPolicy` and friends) that finer-grained resource pinning would need.

Read Part 3 — Resource Management Deep Dive →

Part 4 — Scaling & Resilience

Audience: platform engineers and SRE.

Scaling and staying available are the same problem viewed from two angles: adding capacity when load rises, and not losing capacity when the platform reshapes itself underneath you. Part 4 covers the Horizontal and Vertical Pod Autoscalers, the Cluster Autoscaler (managed provider-side on both MKS and SKE, not a customer CRD), PodDisruptionBudgets, and the placement primitives — topology spread, anti-affinity, taints and tolerations — that decide how a scaled workload is distributed. It also walks the full pod-termination sequence and the rolling-update mechanics that determine whether a deploy is invisible or an outage.

Key takeaways:

- `minReplicas: 1` is not a resilient default; a second replica only helps if the application is genuinely cluster-ready.
- A PDB with zero allowed disruptions blocks node drains — and on STACKIT SKE that block is force-broken after a two-hour hard limit with `terminationGracePeriod=0`.
- Topology spread, anti-affinity, and taints/tolerations solve overlapping problems; the series includes a "which one do I need" comparison.
- For a Deployment bound to an RWO volume, `strategy: Recreate` is the safe rollout choice, not surge-based rolling updates.

Read Part 4 — Scaling & Resilience →

Part 5 — Security

Audience: security engineers and platform engineers.

The default Kubernetes security posture is permissive: pods run as root, traffic flows freely, any service account can reach the API, and images arrive unverified. Part 5 treats security as a set of defaults to set intentionally rather than a project to schedule — RBAC (additive, no deny rule), Pod Security Admission (namespace labels only, `privileged` by default), NetworkPolicy (additive union, no rule order), the supply chain (scanning, Cosign signing, admission enforcement via Kyverno/OPA), and secrets kept out of the cluster with the External Secrets Operator or encrypted in git with SOPS.

Key takeaways:

- RBAC and NetworkPolicy are both purely additive — you remove access by narrowing a grant, not by adding a deny.
- PSA has no API resource of its own: absence of namespace labels means the cluster default (`privileged`) applies — verify, do not assume.
- Neither OVH MKS nor STACKIT SKE ships cluster-wide PSA hardening or a customer-facing etcd-encryption toggle — confirmed empirically.
- `list` and `watch` on Secrets leak values just like `get`; audit the escalation verbs `escalate`, `bind`, `impersonate`.

Read Part 5 — Security →

Part 6 — Day-2 Operations & GitOps

Audience: operations and SRE teams.

Day-1 is getting a cluster to work; Day-2 is keeping it healthy after everyone's attention has moved on — and Day-2 is where clusters quietly accumulate debt. Part 6 covers the operational loop: GitOps delivery with Argo CD or Flux (the cluster as code, with drift detection), the three observability pillars and how applications must correlate them for real end-to-end debugging, runtime security monitoring with Falco, FinOps cost attribution, and the upgrade cadence that keeps a cluster inside its support window.

Key takeaways:

- GitOps gives you the audit trail, the recovery runbook, and drift prevention in one model — but exclude fields like HPA replica counts from auto-remediation.
- Metrics, logs, and traces only add up to debugging if the app propagates trace context and stamps `trace_id` into structured logs.
- Falco's modern eBPF probe runs on both MKS and SKE; the real blocker on small node pools is CPU headroom, not kernel or admission compatibility.
- STACKIT SKE auto-updates Kubernetes and the node OS; OVH MKS needs explicit node-pool rotation — plan the upgrade path around which one you run.

Read Part 6 — Day-2 Operations & GitOps →

Part 7 — Compliance

Audience: security and compliance officers.

The most useful framing for compliance in Kubernetes is not "what extra controls do we need" but "which controls are already in place, and can we prove it?" Part 7 maps the practices from Parts 2–6 onto five regulatory frameworks — NIS2, DORA, PCI-DSS, HIPAA, and the Cyber Resilience Act — and covers the two things compliance adds on top of security: audit logging that produces evidence, and log retention that keeps it available. It closes with the platform-specific reality of audit-log coverage and cost on MKS and SKE.

Key takeaways:

- Compliance is mostly the security and operations work from earlier parts, viewed through a regulatory lens and backed by evidence.
- Retention obligations differ sharply — DORA five years, HIPAA six, PCI-DSS twelve months with three immediately queryable; CRA sets no log-retention period but a ten-year technical-documentation/SBOM duration.
- OVH MKS produces real object-level `kube-apiserver` audit logs; STACKIT SKE's Kubernetes-API audit integration was still not generally available as of mid-2026.
- Storing and querying those logs is a separately-billed service on both platforms — audit retention is not bundled into the base cluster price.

Read Part 7 — Compliance →

The through-line

Three themes run across all seven parts, and noticing them makes the series cohere:

Day-1 versus Day-2. Almost every failure mode in the series is invisible on Day-1 and expensive on Day-2 — a missing PDB, an unset request, an image with no debug path, an absent audit log. The practices are cheap to apply up front and hard to retrofit into a running cluster.

The platform actually matters. "It depends on your provider" is made concrete throughout with OVH MKS, STACKIT SKE, and OpenShift. The differences are not cosmetic: managed Cluster Autoscaler versus a CRD, a two-hour drain limit, no customer etcd-encryption toggle, object-level audit logs on one platform and not the other. The vanilla-Kubernetes practice is the baseline; the provider determines which knobs you actually get to turn.

Security becomes compliance. The RBAC, Pod Security Admission, NetworkPolicy, image signing, and audit logging from Parts 5 and 6 are not separate from Part 7 — they *are* the technical implementation of what NIS2, DORA, PCI-DSS, HIPAA, and the CRA require. Compliance work adds the mapping and the evidence, not a second set of controls.

Related deep-dives

Several topics touched here have their own dedicated series on this blog:

- Choosing a log backend for the retention requirements in Part 7 — the Elasticsearch vs. OpenSearch vs. Loki vs. Quickwit vs. ClickHouse comparison and the managed log archiving series.

- A full observability stack on OVH MKS in practice — SigNoz on OVH infrastructure.
- Running GPU inference workloads on the same managed platform — LLM inference on OVH.

Where to start

If you have no strong prior, read the parts in order — they build on each other. If you are here to fix something specific, use the summaries above to jump to the chapter that names your problem, and follow the cross-links back into the others as you go. The whole series assumes production intent: every practice is there because its absence eventually turns into an incident.

Start with Part 1 — The Big Picture →