K8s & OpenShift: Day-2 Operations & GitOps

2026-07-01

GitOps with Argo CD and Flux, observability, FinOps controls, and cluster upgrade strategies — keeping a Kubernetes cluster healthy after launch. Part 6 of 7.

Part 5 covered the security controls that harden workloads and the cluster. This part covers what keeps a cluster healthy after the initial deployment: GitOps workflows, observability, cost control, and the operational cadence of upgrades.

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

Day-2 is where clusters succeed or quietly degrade

Getting a Kubernetes cluster to a working state is Day-1. What happens after launch — how changes reach the cluster, how failures are detected, whether costs are monitored, whether the cluster is upgraded before it falls out of support — is Day-2. Day-1 gets attention. Day-2 determines whether the platform is a reliable foundation or a liability that accumulates debt silently.

The practices in this part share a common property: their absence is invisible until something goes wrong. A cluster with no GitOps workflow, no observability, and no upgrade process can run fine for months — until a security vulnerability, a misconfigured deploy, or an end-of-life kernel becomes a problem that cannot be fixed quickly.

GitOps — the cluster as code

GitOps is the operational model where the desired state of everything running in the cluster is declared in git, and a reconciliation controller running in the cluster continuously ensures that the actual state matches what git says. What is in git is what is in the cluster. Deviations are either corrected automatically or flagged as alerts.

This model solves three operational problems at once:

- **Auditability.** Every change to the cluster is a git commit with an author, a timestamp, and a diff. The question "what changed and who changed it?" has a precise answer.
- **Recoverability.** If the cluster is lost or corrupted, the desired state in git is the recovery runbook. Pointing a new cluster at the same git repository restores the known state.
- **Drift prevention.** Manual changes made directly via `kubectl apply` or the console are detected and reverted. The cluster cannot silently diverge from what is declared.

Argo CD

Argo CD is a GitOps operator that watches git repositories and applies their contents to the cluster. It uses an `Application` custom resource to map a git repository path to a cluster namespace:

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: myapp
  namespace: argocd
spec:
  project: default
  source:
    repoURL: https://git.example.com/platform/apps.git
    targetRevision: main
    path: apps/myapp/overlays/production
  destination:
    server: https://kubernetes.default.svc
    namespace: myapp
  syncPolicy:
    automated:
      prune: true
      selfHeal: true
    syncOptions:
      - CreateNamespace=true
```

`selfHeal: true` means Argo CD corrects drift automatically — any manual change to a resource managed by this Application is overwritten on the next sync cycle (default: every 3 minutes). `prune: true` removes resources from the cluster when they are deleted from git.

The Argo CD UI provides a real-time view of sync status: `Synced` means the cluster matches git, `OutOfSync` means there is a detected difference and either automatic or manual sync is needed. `Degraded` means the resources exist but are not healthy (e.g., a Deployment with unavailable replicas).

Flux

Flux takes a composable approach using multiple small CRDs rather than a single `Application` object. A `GitRepository` watches a git repo; a `Kustomization` applies a path from that repo; a `HelmRelease` manages a Helm chart.

```yaml
apiVersion: source.toolkit.fluxcd.io/v1
kind: GitRepository
metadata:
  name: platform-apps
  namespace: flux-system
spec:
  interval: 1m
  url: https://git.example.com/platform/apps.git
```

```
  ref:
    branch: main
---
apiVersion: kustomize.toolkit.fluxcd.io/v1
kind: Kustomization
metadata:
  name: myapp
  namespace: flux-system
spec:
  interval: 5m
  path: ./apps/myapp/overlays/production
  prune: true
  sourceRef:
    kind: GitRepository
    name: platform-apps
  targetNamespace: myapp
```

Flux reconciles on the configured interval and applies changes as they appear in git. Unlike Argo CD, Flux has no built-in UI — it is fully CLI-driven (`flux get all`, `flux logs`), which fits GitOps-native teams that prefer observing the system through git rather than a dashboard.

Both Argo CD and Flux are CNCF graduated projects with broad production adoption. The difference is primarily in interface and composition style, not capability.

> **i OpenShift GitOps Operator**
>
> Red Hat's OpenShift GitOps Operator ships Argo CD as a managed component, integrated with OpenShift's authentication (SSO via Dex), the OpenShift console, and the cluster's RBAC. `AppProject` boundaries map cleanly to OpenShift teams and namespaces. On ROSA and ARO, the GitOps Operator is available from the OperatorHub without manual installation.

Structuring the git repository

A GitOps repository needs a structure that supports multiple environments (development, staging, production) without duplicating configuration. Two patterns are common:

Kustomize overlays — a `base/` directory holds the common configuration; `overlays/<env>/` directories hold environment-specific patches (replicas, resource sizes, image tags). Argo CD and Flux both support Kustomize natively. Both Kustomize and Helm package the same Deployment/StatefulSet/DaemonSet manifests introduced in Part 2 — they change how a manifest is templated and delivered, not what kind of controller it describes.

Git branches per environment — `main` is production, `staging` and `dev` branches hold environment-specific state. Promotion is a merge or a cherry-pick. This is simpler to navigate but harder to audit across environments at a point in time.

For most setups, Kustomize overlays are the more maintainable approach: a single branch, clear separation between shared and environment-specific configuration, and changes that diff clearly in PRs.

Drift detection without auto-remediation

`selfHeal: true` in Argo CD (or `prune: true` in Flux) automatically corrects drift — which is desirable for most resources. For some resources, auto-remediation is too aggressive: Kubernetes `HorizontalPodAutoscaler` objects change their `currentReplicas` field dynamically, which would trigger endless sync cycles if not excluded. Similarly, Secret values managed by External Secrets Operator should not be reverted by GitOps.

Argo CD supports `ignoreDifferences` at the Application level to exclude fields that are expected to diverge from the git state:

```
ignoreDifferences:
  - group: apps
    kind: Deployment
    jsonPointers:
      - /spec/replicas
  - group: autoscaling
    kind: HorizontalPodAutoscaler
    jqPathExpressions:
      - .status
```

Observability — the three pillars

Observability is the ability to understand what a system is doing from its external outputs. For a Kubernetes cluster, three signal types are needed: metrics (what happened, in numbers), logs (what happened, in context), and traces (how a request flowed through the system).

Metrics

Metrics are numerical measurements over time: CPU usage, request rate, error rate, queue depth. They are the primary signal for alerting and for dashboards that show cluster health at a glance.

The standard stack for Kubernetes metrics is:

- **Prometheus** — scrapes metrics endpoints and stores time-series data. The Prometheus Operator (now part of the kube-prometheus project) manages Prometheus instances, `ServiceMonitor` / `PodMonitor` CRDs for discovery, and `PrometheusRule` CRDs for alert rules.
- **kube-state-metrics** — generates metrics from Kubernetes object states: `kube_pod_container_status_restarts_total`, `kube_deployment_status_replicas_unavailable`, `kube_horizontalpodautoscaler_status_current_replicas`, and hundreds more.
- **node-exporter** — collects host-level metrics: CPU, memory, disk I/O, network interfaces.
- **Grafana** — visualises Prometheus data. The kube-prometheus project ships pre-built dashboards for cluster overview, node utilisation, pod resource usage, and API server latency.

Key metrics to alert on:

Metric	Signal
`kube_pod_container_status_restarts_total`	CrashLoopBackOff or OOMKill — rate > 1/hour per pod warrants investigation
`container_cpu_throttled_seconds_total`	CPU limit too low — covered in Part 3
`kube_deployment_status_replicas_unavailable`	Deployment not fully available — rollout stalled or probe failing
`kube_node_status_condition`	Node `NotReady` — node problem controller or hardware issue
`etcd_server_leader_changes_seen_total`	etcd instability — leader elections under normal operation should be rare

Instrument application code to expose a `/metrics` endpoint in the Prometheus exposition format — or use the OpenTelemetry SDK, which can push metrics to both Prometheus and OTLP collectors. Application-level metrics (request rate, error rate, latency histograms) are what make the difference between knowing a service is broken and knowing what is broken about it.

i OpenShift Monitoring

OpenShift ships a fully managed monitoring stack based on Prometheus Operator, Alertmanager, and Thanos Querier — running in the `openshift-monitoring` namespace. The read-only Grafana that older releases bundled was deprecated in 4.11 and removed in 4.12; metrics, dashboards, and alerts are now viewed through the console's Observe section (Observe → Dashboards / Metrics) instead of a separate Grafana instance. Exactly which components are present still depends on the OpenShift version and any monitoring-related operators installed. By default the stack monitors the cluster itself. User workload monitoring (scraping application `ServiceMonitor` objects in user namespaces) is enabled cluster-wide via the `cluster-monitoring-config` ConfigMap and requires no separate Prometheus installation.

i OVH MKS & STACKIT SKE — no pre-installed monitoring stack

AKS offers Container Insights as an optional managed addon; GKE ships Cloud Logging and the Ops Agent by default; EKS provides the CloudWatch Agent as a managed addon. OVH MKS and STACKIT SKE ship no monitoring components — the cluster starts empty.

The full Prometheus Operator + Alertmanager + kube-state-metrics + node-exporter + Grafana stack described in this section is not optional on these platforms: it is the mandatory starting point for any production-grade cluster. The same applies to log shipping — Fluent Bit or Vector must be deployed as a DaemonSet from day one.

Plan this into the initial cluster setup. Retrofitting observability into a cluster that has been running without it — with workloads that do not expose `/metrics` endpoints and logs that were never shipped centrally — is significantly harder than onboarding into an observable cluster.

Logs

Structured, centralised logging answers questions that metrics cannot: what was the exact error message, what was the request payload, what happened in what order during an incident.

Two things matter for Kubernetes logs:

Write structured logs. Applications that emit plain-text logs force operators to parse them with fragile regex. Applications that emit JSON or logfmt give any log system — Loki, Elasticsearch, ClickHouse — structured fields it can filter, aggregate, and alert on without post-processing.

Centralise log shipping. The `kubectl logs` command reads from the kubelet's local log buffer, which rotates. Pod restarts lose logs. Long-term storage requires a log shipper (Fluent Bit, Vector, or Fluentd) that reads from container runtimes and forwards to a backend with retention.

The choice of backend — Loki, Elasticsearch/OpenSearch, ClickHouse, Quickwit — depends on retention requirements, query patterns, and operational preference. The log archiving implications for regulatory retention are covered in Part 7; the technical comparison of backends is covered in the managed log archiving series.

> **Fluent Bit vs. Vector for cluster-level shipping**
>
> Fluent Bit is the smaller-footprint option: a C binary of roughly 1 MB that ships as the default DaemonSet agent in many Kubernetes distributions, with filter plugins that cover most structured-log transformations. Vector, written in Rust, carries a larger footprint but is still low-overhead compared to JVM- or Ruby-based agents — its advantage is VRL (Vector Remap Language), which handles far more expressive in-flight transformations than Fluent Bit's filters, plus native sinks across a broader range of backends. For a DaemonSet running one instance per node with straightforward parsing needs, Fluent Bit is the leaner choice; Vector earns its overhead once the pipeline needs non-trivial transformation logic or targets a sink Fluent Bit does not support natively. See the log backend comparison series for the full shipping-agent support matrix across Elasticsearch/OpenSearch, Loki, Quickwit, and ClickHouse.

Traces

Distributed tracing records the end-to-end path of a request across services. A trace consists of spans — each span is a unit of work in one service, with a start time, duration, and context (service name, operation, status code, error).

Tracing answers questions that neither metrics nor logs answer well: why is this request slow (it is spending 600 ms in the payment service waiting for a database query), which service introduced the error, and what is the shape of the call graph for a given endpoint.

The standard for instrumenting Kubernetes workloads is **OpenTelemetry (OTel)**. The OTel SDK integrates with most languages (Go, Java, Python, Node.js, Rust) and emits traces (and metrics and logs) over the OTLP protocol to an OTel Collector running as a DaemonSet or Deployment in the cluster. The collector fans out to a tracing backend:

- **Tempo** (Grafana) — low-cost object-storage-backed trace store; queries from Grafana

- **Jaeger** — mature open-source tracing backend with its own UI
- **OTLP-compatible commercial backends** — Honeycomb, Lightstep, Datadog, Dynatrace

Automatic instrumentation (via the OpenTelemetry Operator and language-specific auto-instrumentation) can inject the OTel agent into pods without code changes — useful for getting initial trace coverage quickly. Manual SDK instrumentation gives finer control over span names and attributes.

These three signals only add up to useful end-to-end debugging if applications wire them together. That requires three things from application code, not just from the cluster's observability stack: propagate trace context (the W3C `traceparent` header) across every outbound call, so a single request produces one connected trace instead of disconnected fragments per service; emit structured logs that include the active `trace_id` and `span_id`, so a log line can be pivoted straight to the trace it belongs to; and label metrics with the same identifiers used in logs and traces (service name, namespace, version), so a metric spike can be pivoted to the traces and logs from the same window. Without this correlation, metrics, logs, and traces stay three separate systems that each answer their own question in isolation, and an incident still means manually cross-referencing timestamps across three UIs.

Runtime security monitoring

Part 5 covered scanning, signing, and admission enforcement — controls that decide what is allowed to start. None of them observe what a container actually does once it is running, which matters because a container can pass every one of those checks and still misbehave later: a compromised dependency pulled in at build time, a legitimate process exploited after deployment, or a credential used from an unexpected place. Runtime security monitoring is the ongoing, Day-2 counterpart to Part 5's pre-deployment gates.

Falco, a CNCF graduated project, is the de-facto standard. It runs as a DaemonSet and uses eBPF (or a kernel module, on older kernels) to observe syscalls from every container on the node with negligible overhead, evaluating them against a rule set in real time. Common default rules detect things like:

- A shell spawned inside a container that never runs one interactively
- An outbound connection to an unexpected destination
- A write to a sensitive path (`/etc/shadow`, the container runtime socket)
- Privilege escalation inside a running container (e.g., an unexpected `setuid` call)

Falco alerts are events, not metrics — route them into the same log pipeline covered earlier in this part (Fluent Bit to the log backend) or to a dedicated channel (Slack, a SIEM) for the security team to triage. Treat a Falco alert the same way as any other production incident signal: it needs an owner and a response process, not just a dashboard nobody watches.

> **i OVH MKS & STACKIT SKE — Falco**
>
> Falco's modern eBPF probe (the default driver since Falco 0.36) needs a kernel ≥ 5.8 with BTF exposed — no kernel module compilation required. Both platforms clear that bar comfortably:

- **OVH MKS** worker nodes run Ubuntu 22.04 LTS, kernel 5.15+ (up to 6.5+ on the HWE stack), with BTF enabled by default since Ubuntu's 5.4 kernels.
- **STACKIT SKE** nodes run Flatcar Linux (kernel 6.6.x as of this writing), which explicitly enables `CONFIG_DEBUG_INFO_BTF`, `CONFIG_IKHEADERS`, and `CONFIG_BPF_LSM` in its kernel build — specifically so eBPF tooling works without kernel headers or a compiler on an immutable OS. Flatcar's immutability would block Falco's legacy kernel-module driver, but that is no longer the default.

Neither platform enforces cluster-wide Pod Security restrictions beyond the Kubernetes default (confirmed empirically for OVH MKS in Part 5; STACKIT SKE follows the same namespace-label-based model with no stricter platform default found). Falco's DaemonSet needs `hostPID`, `hostNetwork`, and privileged capabilities — that runs unmodified on both, unless a restrictive `pod-security.kubernetes.io/enforce` label has been applied to its own namespace.

Kernel and admission compatibility do not guarantee the pods actually get scheduled, though. Falco's default Helm chart requests 100m CPU per node for the main container; on a small node pool that is already running a full observability and service-mesh stack (an OTel/SigNoz pipeline, Istio Ambient's `istiod`/`ztunnel`/CNI, the CNI DaemonSet itself), that headroom may not exist. Check `kubectl describe node` for allocated-vs-allocatable CPU before assuming Falco will schedule — a node already at 95%+ requested CPU needs either a bigger node pool or a lower resource request on the Falco chart, not a compatibility fix.

i OpenShift: Red Hat Advanced Cluster Security

Red Hat Advanced Cluster Security for Kubernetes (RHACS, built on the open-source StackRox project) covers the same build-deploy-runtime lifecycle as Falco but as an integrated OpenShift product. Its Collector component uses eBPF for runtime process, network, and filesystem monitoring per node, feeding into the same Central console used for image scanning and admission policy. Falco also runs on OpenShift and is a valid alternative; RHACS is the platform-native option with tighter console integration.

FinOps — keeping costs proportional to value

Kubernetes makes it easy to run workloads without thinking about their cost. The FinOps practices in this section connect resource consumption back to spend, without requiring a dedicated FinOps team.

Right-sizing with VPA recommendations

VPA in `Off` mode (covered in Part 4) collects consumption data and produces right-sizing recommendations without making changes. This is the most actionable input for reducing waste: the VPA recommendation shows what the workload actually uses, and the difference between that and the configured request is idle capacity being paid for.

Read recommendations across all workloads in a namespace:

```
kubectl -n myapp get vpa -o json | \
  jq '.items[] | {
    name: .metadata.name,
    cpu_request: .spec.resourcePolicy,
recommended_cpu: .status.recommendation.containerRecommendations[].target.cpu,
recommended_mem: .status.recommendation.containerRecommendations[].target.memory
  }'
```

A consistent gap between configured requests and VPA recommendations across a namespace is a signal that requests were set conservatively and have not been revisited.

Node pools for mixed workload profiles

Not all workloads have the same cost-availability trade-off. Batch jobs, CI runners, and non-critical background workers can tolerate interruption. Frontend APIs and databases cannot.

Most managed Kubernetes providers support multiple node pools, which allows different pools to use different instance types and pricing models:

- **On-demand nodes** — standard pricing, guaranteed availability — for latency-sensitive and stateful workloads
- **Spot / preemptible nodes** — 60–90% less expensive, but may be reclaimed with 30–120 seconds notice — for fault-tolerant batch, stateless workers, and development namespaces

Schedule fault-tolerant workloads onto spot nodes using `nodeSelector` or a node `taint` and `toleration`. Ensure those workloads have PDBs and are stateless — a spot interruption is a voluntary eviction from Kubernetes' perspective and triggers the same drain behaviour as a node upgrade.

Cost visibility per team and workload

Without labels, Kubernetes costs are opaque: the cloud bill shows node hours, but not which team or application consumed them. A label convention applied consistently across all workloads enables cost attribution.

A minimal label set for cost visibility:

```
labels:
  app.kubernetes.io/name: myapp
  app.kubernetes.io/component: backend
  cost-center: engineering-platform
  environment: production
```

Tools like **OpenCost** (CNCF incubating) and **Kubecost** read these labels and allocate cluster costs per namespace, workload, or label value — producing a per-team or per-application cost breakdown from the same cluster. Provider-native cost views (EKS Cost Insights, AKS Cost Analysis, GKE Cost Allocation) provide similar breakdowns using resource labels when they are consistently applied.

Idle and over-provisioned workload detection

Workloads with resource requests significantly above their actual consumption represent idle capacity. A namespace where every Deployment requests 2 vCPU but peak consumption is 200m is paying for 10× the capacity actually needed.

The detection query in Prometheus:

```
# CPU idle ratio per container (lower = more idle)
container_cpu_usage_seconds_total
  /
kube_pod_container_resource_requests{resource="cpu"}
```

A ratio below 0.1 (10% of requested CPU consumed) across a sustained period is a signal to investigate. Combine with VPA recommendations for a prioritised right-sizing backlog.

Cluster upgrades

Kubernetes releases three minor versions per year. Each minor version is supported for approximately 14 months from release. Running end-of-life Kubernetes means running without security patches and on APIs that the ecosystem has moved past. Upgrades are not optional; the question is only whether they are planned or reactive.

Version skew and the upgrade order

The Kubernetes version skew policy defines the safe distance between control plane and node versions. The control plane (API server, controller manager, scheduler) must always be upgraded before the nodes (kubelet). Since Kubernetes 1.28 the kubelet may be at most three minor versions behind the API server (it was two on clusters older than 1.25):

- API server at 1.31 ⟶ kubelets can run 1.28, 1.29, 1.30, or 1.31
- API server at 1.32 ⟶ kubelets can run 1.29, 1.30, 1.31, or 1.32

On managed clusters (EKS, AKS, GKE, OVH MKS, STACKIT SKE), the provider upgrades the control plane first, then offers node pool upgrades. The node pool upgrade can be deferred to a maintenance window. Do not let the gap between control plane and node version exceed three minor versions.

Node pool rotation — the zero-downtime upgrade pattern

For managed clusters, the safest upgrade pattern for nodes is pool rotation rather than in-place drain:

1. **Create a new node pool** with the target version and desired node size.
2. **Cordon the old pool** — `kubectl cordon <node>` on every old node, or set the pool to `NoSchedule` taint — so no new pods schedule onto it.
3. **Drain the old pool** — `kubectl drain --ignore-daemonsets --delete-emptydir-data <node>` for each node. Kubernetes honours PDBs during drain: a drain that would violate a PDB waits.
4. **Verify workload health** on the new pool before proceeding.
5. **Delete the old pool** once all nodes are drained and new pods are running correctly.

This pattern works because the new pool comes up before the old pool drains — capacity is maintained throughout, and the upgrade is reversible until the old pool is deleted.

⚠ Check for deprecated APIs before upgrading

Each Kubernetes minor version deprecates and eventually removes API versions. A Deployment using `apps/v1` is fine; a manifest still using a removed beta API fails at apply time after the upgrade. Check for deprecated API usage before upgrading with **Pluto** (`pluto detect-files -d` against your GitOps repository) or `kubectl convert`. Providers also document removed APIs in their upgrade guides — read them before triggering a minor version upgrade.

i OpenShift cluster upgrades via CVO

OpenShift upgrades are managed by the Cluster Version Operator (CVO), which pulls update payloads from the Red Hat release service. Upgrades are triggered via the console (Administration → Cluster Settings → Update) or via `oc adm upgrade`. Three channels control which versions are offered: `stable` (production-ready), `fast` (newer, less soak time), and `candidate` (pre-release). CVO upgrades are sequential — you cannot skip minor versions.

CVO drains and replaces nodes in sequence using machine config pools. A PDB that cannot be satisfied will pause the upgrade; OpenShift surfaces this via the `MachineConfigPool` status and emits events. Checking `oc get mcp` and `oc get clusteroperators` is the standard diagnostic during a stalled upgrade.

i OVH MKS & STACKIT SKE — upgrade paths differ between providers

OVH MKS: Control-plane upgrades are triggered via the OVHcloud console or API — select the target version and confirm. For worker nodes, the node pool rotation pattern (create new pool → cordon old pool → drain → verify → delete) is the standard path. Node upgrades require explicit action; OVH MKS does not automatically upgrade running worker nodes.

STACKIT SKE: Includes auto-update functionality for both Kubernetes versions and node operating system (Flatcar Linux) — clusters can be kept up to date automatically. SKE also ships automatic repair functions that detect and remediate node-level issues without operator intervention. This reduces the manual upgrade overhead compared to OVH MKS.

The "a drain that would violate a PDB waits" behaviour described above does not hold unconditionally on SKE: during a rolling node update, SKE drains each node individually and waits at most **2 hours** for a PDB to allow it. If the PDB still blocks the drain after that window, SKE force-deletes the remaining pods with `terminationGracePeriod=0` and proceeds. A PDB with `Allowed disruptions: 0` reduces disruption risk but does not guarantee the node will never be drained out from under those pods — plan for the forced-termination case, not just the PDB-respected case.

On both platforms there is no CVO-style channel selection. Run pre-upgrade checks (Pluto, deprecated API detection) before any minor version upgrade regardless of whether the update is manual or automatic.

Testing upgrades before they hit production

Staging clusters that mirror production workloads are the standard mechanism for catching upgrade incompatibilities before they affect users. The staging cluster runs the target version; CI deploys the same manifests to staging that will go to production. Failures in staging are cheap; failures in production during an upgrade window are not.

If a dedicated staging cluster is not available, the provider's upgrade canary approach — upgrade a subset of node pool capacity first, verify workload health, then upgrade the remainder — provides a partial equivalent.

Automated checks worth running before any minor version upgrade:

```
# Check for deprecated API usage in the GitOps repository
pluto detect-files -d ./gitops-repo --target-versions k8s=v1.32

# Check current cluster for deprecated API calls (server-side)
kubectl get --raw /metrics | grep apiserver_requested_deprecated_apis

# Verify all Deployments have at least 2 replicas and a PDB
kubectl get deploy -A -o json | \
  jq '.items[] | select(.spec.replicas < 2) | "\(.metadata.namespace)/
\(.metadata.name)"'
```

Putting it together — the operational loop

The practices in this part form a continuous loop rather than a one-time configuration:

- **GitOps** provides the delivery mechanism and the audit trail for every change.
- **Observability** provides the signal that something has changed or broken.
- **FinOps** closes the feedback loop between resource consumption and cost, informing the right-sizing decisions made in Part 3 and the autoscaling decisions made in Part 4.
- **Upgrades** keep the platform within the supported version window and introduce improvements in the Kubernetes scheduler, storage, and networking layers.

None of these are one-time tasks. A GitOps workflow needs to be maintained as the repository grows. Observability coverage grows as new workloads are onboarded. Cost reviews should happen on a regular cadence — quarterly at minimum. Upgrades follow the version release calendar. The question for Day-2 operations is not "is this done?" but "does this loop run reliably?"

Next: Part 7 covers compliance — mapping K8s controls to NIS2, DORA, PCI-DSS, and HIPAA, audit logging as evidence, and log retention requirements.

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)
- Kubernetes production readiness checklist — learnkube.com (2026)

- Kubernetes Best Practices for Safer, More Reliable Clusters — Komodor (2026)
- Architecture Best Practices for Azure Container Apps — Microsoft Azure Well-Architected Framework (2026)
- Falco — CNCF
- Red Hat Advanced Cluster Security for Kubernetes — Red Hat
- kubernetes-best-practices — Diego Lima, Apache-2.0