K8s & OpenShift: Security

2026-07-01

RBAC, Pod Security Admission, NetworkPolicy, and supply-chain controls — the security practices every Kubernetes production workload needs. Part 5 of 7.

Part 4 covered scaling and resilience. This part covers security — not as a checklist to complete before an audit, but as a set of controls that reduce real attack surface and limit the blast radius when something goes wrong.

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 (this post)** - Part 6 — Day-2 operations & GitOps - Part 7 — Compliance (regulatory frameworks, audit logging, log retention)

Security is a default, not a project

The default Kubernetes security posture is permissive. Without deliberate configuration, pods run as root, all traffic flows freely between namespaces, any pod can call the Kubernetes API with broad permissions, and container images arrive from registries without verification. None of these defaults are appropriate for a production cluster.

The controls in this part do not require specialised security expertise to apply — they are standard Kubernetes and OpenShift configuration. What they do require is applying them intentionally, rather than discovering their absence during an incident.

Part 2 already covered part of this at the individual container level — `runAsNonRoot`, `allowPrivilegeEscalation: false`, `capabilities.drop: [ALL]`, and a read-only root filesystem. This part builds on that with the controls that operate above a single container: who can do what against the API (RBAC), what a pod is allowed to do regardless of what its manifest asks for (Pod Security Admission), which pods can talk to which (NetworkPolicy), and whether an image can be trusted before it runs at all (supply chain).

RBAC — the access control layer

Role-Based Access Control (RBAC) governs which identities — users, groups, or service accounts — can perform which actions on which Kubernetes resources. It is the foundational access control mechanism for both humans and workloads.

RBAC lives in its own API group, `rbac.authorization.k8s.io/v1`, separate from the core API and from the resources it governs. Four object kinds do the work, in two pairs:

- **`Role` / `ClusterRole`** define *what* is allowed: a set of verbs (`get`, `list`, `create`, `delete`, and so on) on a set of resources. A `Role` only exists within one namespace. A `ClusterRole` is

not namespaced — it can describe genuinely cluster-wide permissions, or serve as a reusable permission set that gets scoped down later.
- **`RoleBinding` / `ClusterRoleBinding`** define *who* gets those permissions: they attach a `Role` or `ClusterRole` to one or more subjects — a user, a group, or a `ServiceAccount`.

The binding's own kind decides the scope, not the kind it references: a `RoleBinding` only ever grants access within its own namespace, even when it references a `ClusterRole`. A `ClusterRoleBinding` grants access across the whole cluster. That distinction is what makes a single `ClusterRole` reusable both ways — bound per-namespace via a `RoleBinding` wherever it is needed, or bound once cluster-wide via a `ClusterRoleBinding`.

RBAC is purely additive — there is no deny rule. Access is entirely a function of what has been granted; removing unwanted access means removing or narrowing a binding, not adding an exclusion. See the Kubernetes RBAC reference for the full object model.

Roles and ClusterRoles

A `Role` grants permissions within a single namespace. A `ClusterRole` grants permissions cluster-wide, or is used as a template that `RoleBindings` can apply within specific namespaces.

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: pod-reader
  namespace: myapp
rules:
  - apiGroups: [""]
    resources: ["pods", "pods/log"]
    verbs: ["get", "list", "watch"]
```

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: pod-reader-binding
  namespace: myapp
subjects:
  - kind: ServiceAccount
    name: myapp
    namespace: myapp
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

Practical guidelines:

- **Grant at namespace scope, not cluster scope.** Most workloads and most operators need access to their own namespace, not to the whole cluster. Use `Role` + `RoleBinding` by default;

reach for `ClusterRole` only when the use case genuinely requires cluster-wide access (e.g., a monitoring agent that scrapes all namespaces).

- **Never use `cluster-admin` for application workloads.** `cluster-admin` is the root equivalent — it bypasses all RBAC checks. It is appropriate for cluster administrators performing break-glass operations, not for deployments.
- **Audit what is bound to `cluster-admin`.** Helm chart installations and operator deployments sometimes request more permissions than they need. Check with the command below.
- **Prefer read-only service accounts.** Most pods do not need to create or modify Kubernetes resources at all. If a pod does not call the Kubernetes API, its service account needs no RBAC permissions. If it does, grant only the verbs and resources it actually uses.
- **`list` and `watch` on Secrets are as sensitive as `get`.** Both return the full Secret object, contents included — not just metadata. A Role that grants `list` or `watch` on `secrets` without `get` still exposes every value.
- **Watch for the escalation verbs: `escalate`, `bind`, `impersonate`.** These allow a subject to grant itself or others permissions it does not already hold, or to act as a different identity entirely. They are rarely needed outside cluster-administration tooling — audit any `Role` or `ClusterRole` that includes them.

See the Kubernetes RBAC good practices guide for the complete list, including certificate-signing and admission-webhook permissions.

To list every subject bound to `cluster-admin`:

```
kubectl get clusterrolebindings -o json | \
  jq '.items[] | select(.roleRef.name=="cluster-admin") | .subjects'
```

Service accounts for pod identity

Every pod runs with a service account. The `default` service account in each namespace receives no RBAC permissions by default, which is correct — but it does mount a service account token into the pod automatically (a JWT that authenticates to the Kubernetes API). For pods that never call the Kubernetes API, disable this mount entirely:

```
spec:
  automountServiceAccountToken: false
```

For pods that do need API access, create a dedicated service account and grant only the permissions that workload requires. Sharing service accounts across unrelated workloads conflates their permissions and makes least-privilege impossible to achieve.

> **i OpenShift projects and service accounts**
>
> OpenShift projects create three service accounts by default: `default`, `builder`, and `deployer`. The `builder` account has permissions to push to the internal registry; the `deployer` account can manage deployments. Neither should be used for application workloads — create a dedicated service account per workload, just as on vanilla Kubernetes.

A user, group, or service account is not limited to a single binding: the same subject can be the target of several `RoleBindings` and `ClusterRoleBindings` at once — for example a `RoleBinding` granting namespace-scoped access plus a separate `ClusterRoleBinding` granting a narrow cluster-wide permission. Its effective permissions are simply the union of everything bound to it, consistent with RBAC being purely additive.

Pod Security Admission

Pod Security Admission (PSA) is the built-in mechanism — available since Kubernetes 1.25 — for enforcing security standards at the namespace level. It replaced PodSecurityPolicy, which was removed in 1.25.

PSA defines three security levels:

Level	What it enforces
`privileged`	No restrictions — equivalent to no policy
`baseline`	Prevents the most egregious misconfigurations (host namespaces, privileged containers, host ports)
`restricted`	Enforces the full security hardening from Part 2: non-root, read-only filesystem, no privilege escalation, capabilities dropped, seccomp profile set

Each level can be applied in three modes per namespace:

Mode	Behaviour
`enforce`	Pods that violate the policy are rejected at admission
`audit`	Violations are logged but pods are allowed
`warn`	A warning is returned to the client but pods are allowed

Configure PSA with namespace labels:

```yaml
apiVersion: v1
kind: Namespace
metadata:
  name: myapp
  labels:
    pod-security.kubernetes.io/enforce: restricted
    pod-security.kubernetes.io/enforce-version: latest
    pod-security.kubernetes.io/warn: restricted
    pod-security.kubernetes.io/warn-version: latest
```

Start new namespaces at `restricted`. For namespaces with existing workloads that are not yet compliant, use `audit` or `warn` first to identify violations without breaking deployments, then migrate workloads and switch to `enforce`.

The `restricted` level maps directly to what Part 2 recommends for `securityContext`: non-root UID, `allowPrivilegeEscalation: false`, `readOnlyRootFilesystem: true`, `capabilities.drop: [ALL]`, and `seccompProfile: RuntimeDefault`. Workloads that follow Part 2's guidance pass `restricted` without changes.

Checking what is actually enforced

PSA has no API resource of its own — it will not show up anywhere in `kubectl api-resources`. There is no `kind: PodSecurity...` to query. The entire mechanism is a built-in admission plugin that reads three labels off the `Namespace` object; that makes `Namespace` labels the only place to look:

```
# one namespace
kubectl get ns myapp --show-labels

# every namespace, PSA columns only
kubectl get ns -o custom-
columns='NAME:.metadata.name,ENFORCE:.metadata.labels.pod-
security\.kubernetes\.io/enforce,AUDIT:.metadata.labels.pod-
security\.kubernetes\.io/audit,WARN:.metadata.labels.pod-
security\.kubernetes\.io/warn'
```

A namespace with none of the three labels set is not "secure by default" — it means the cluster-wide default applies, which on an unconfigured cluster is `privileged` (no restriction at all). Seeing a blank `ENFORCE` column across every namespace is a signal to act on, not a signal that nothing is enforced yet by design. The cluster-wide default itself lives in the API server's `AdmissionConfiguration` file, which is not a Kubernetes object either — it is not visible via `kubectl` at all, only through access to the control plane host or the managed provider's support channel.

> **i OVH MKS — confirmed empirically: no cluster-wide PSA hardening**
>
> Tested directly against an OVH MKS cluster: a pod requesting `hostNetwork: true`, `hostPID: true`, `privileged: true`, `runAsUser: 0`, and an added `NET_ADMIN` capability — violating nearly every `restricted` rule and most `baseline` rules — was scheduled and started with no warning at all, in a namespace carrying no `pod-security.kubernetes.io/*` labels. Managed control plane does not mean hardened defaults; PSA on OVH MKS behaves exactly like the documented vanilla-Kubernetes default (`privileged`) unless a namespace is labelled explicitly.

> **i OpenShift Security Context Constraints**
>
> OpenShift uses Security Context Constraints (SCCs) instead of PSA. The `restricted-v2` SCC, which applies to new workloads by default, enforces equivalent controls to PSA `restricted` — including non-root, seccomp, and capability drops. OpenShift's admission controller rejects non-compliant pods at deployment time, surfacing the violation as a clear error message.
>
> When a workload needs permissions beyond `restricted-v2` (a storage driver, a monitoring agent), grant the minimum SCC that covers the need — `nonroot` before `anyuid`, `anyuid` before `privileged`. Never grant `privileged` SCC to application workloads.

NetworkPolicy — traffic isolation between workloads

By default, every pod in a Kubernetes cluster can reach every other pod across all namespaces. There are no network boundaries unless you create them. NetworkPolicy objects define which traffic is allowed; anything not explicitly allowed is dropped once a policy applies to a pod.

Anyone used to classic firewall ACLs will expect rule order to matter — the first or last matching rule wins, evaluated top to bottom. NetworkPolicy has no such concept. Like RBAC, it is purely additive: there is no rule order and no explicit deny rule anywhere. A pod that no NetworkPolicy selects still allows all traffic — that part of the default never changes. The moment at least one NetworkPolicy selects a pod, the default flips for that pod: only traffic matching a rule in *any* of the policies selecting it is allowed, and everything else is dropped by omission, not by an explicit deny. This is also why a policy with no rules at all is a complete lock: it selects pods but contributes nothing to the union of allowed traffic, so nothing is permitted until another policy adds an explicit allow rule for those same pods.

Start with default deny

The baseline is a default-deny policy in every production namespace — applied to all pods, blocking all ingress and egress. Add allow rules on top of it.

```
# Block all ingress to all pods in the namespace
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny-ingress
  namespace: myapp
spec:
  podSelector: {}
  policyTypes:
    - Ingress
---
# Block all egress from all pods in the namespace
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny-egress
  namespace: myapp
spec:
  podSelector: {}
  policyTypes:
    - Egress
```

Then add specific allow rules:

```
# Allow ingress to the backend from the frontend namespace only
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
```

```
  name: allow-from-frontend
  namespace: myapp
spec:
  podSelector:
    matchLabels:
      app: backend
  policyTypes:
    - Ingress
  ingress:
    - from:
        - namespaceSelector:
            matchLabels:
              kubernetes.io/metadata.name: frontend
      ports:
        - protocol: TCP
          port: 8080
```

```
# Allow egress to DNS (required for name resolution) and the database
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-egress-dns-and-db
  namespace: myapp
spec:
  podSelector:
    matchLabels:
      app: backend
  policyTypes:
    - Egress
  egress:
    - ports:
        - protocol: UDP
          port: 53
    - to:
        - namespaceSelector:
            matchLabels:
              kubernetes.io/metadata.name: database
      ports:
        - protocol: TCP
          port: 5432
```

⚠ Default-deny egress blocks DNS — add the DNS allow rule

DNS resolution uses UDP port 53. A default-deny egress policy without an explicit DNS allow rule breaks name resolution for all pods in the namespace — `curl example.com` fails, service

discovery breaks, and the application cannot reach anything by hostname. Always add the DNS egress allow rule alongside any default-deny egress policy.

CNI support for NetworkPolicy

NetworkPolicy enforcement depends on the CNI plugin. Calico, Cilium, and OVN-Kubernetes (the default CNI on OpenShift and on many managed providers) enforce NetworkPolicy. Flannel alone does not — it accepts NetworkPolicy objects but ignores them silently. **Canal** is the one common exception worth knowing by name: it pairs Flannel for pod networking with Calico's policy engine (Felix) purely for NetworkPolicy enforcement, without Calico's own IPAM or BGP routing — so a Flannel-based cluster running Canal does enforce NetworkPolicy, unlike plain Flannel.

On managed clusters, check the provider's documentation to confirm NetworkPolicy enforcement — or check directly: `kubectl -n kube-system get pods` and look at the CNI DaemonSet name (`calico-*`, `canal-*`, `cilium-*`, `kube-flannel-*`), or `kubectl get crd | grep -iE 'calico|cilium'` for the CNI's own CRDs. On EKS the default VPC CNI enforces NetworkPolicy since 1.29 when network policy support is explicitly enabled; AKS with the Azure CNI Overlay and Cilium enforces it; GKE's Dataplane V2 uses Cilium. **OVH MKS installs Canal by default** — confirmed both in OVH's own documentation and empirically on a live cluster (`canal-*` DaemonSet pods, `*.crd.projectcalico.org` CRDs) — with Cilium available as an alternative CNI choice on newer Standard-plan clusters. STACKIT SKE supports NetworkPolicy on its Flatcar-Linux-based nodes.

> **i Cilium Network Policies on OpenShift**
>
> OpenShift uses OVN-Kubernetes as its default CNI, which fully supports standard NetworkPolicy objects. OpenShift also provides its own `EgressNetworkPolicy` and `AdminNetworkPolicy` (an upstream Kubernetes SIG-Network API) for cluster-admin-level egress control that cannot be overridden by namespace owners. These are useful for enforcing cluster-wide egress restrictions — for example, blocking all pods from calling cloud metadata endpoints.

Supply chain security

The software supply chain attack surface runs from the base image in a Containerfile to the deployment manifest applied to a production cluster. Three controls address the main risks: image scanning, image signing, and admission policy enforcement.

Image scanning

Image scanning checks container images against known vulnerability databases (CVEs). The result is a list of vulnerabilities in the image's packages and libraries, with severity levels and available fixes.

Trivy is a widely-used open-source scanner with good coverage across OS packages, language runtimes (Go modules, npm, pip, Maven), and infrastructure-as-code files. It integrates with CI pipelines, container registries, and Kubernetes admission controllers.

Integrate scanning into CI at the build step, before the image is pushed to the registry:

```
trivy image --exit-code 1 --severity CRITICAL myapp:1.4.2
```

`--exit-code 1` fails the build on critical findings. Set policy on which severity levels block the pipeline — `CRITICAL` always, `HIGH` depending on context and available fix.

Scanning in CI alone is not sufficient. Images are deployed once and run for weeks or months. A vulnerability discovered after deployment is not caught by a scan that only ran at build time. Registry-level continuous scanning (available in ECR, ACR, Artifact Registry, Harbor, and others) rescans stored images as new CVE data arrives and surfaces new findings without a new build.

Git-hosting platforms increasingly bundle scanning directly into the pipeline: GitLab's Container Scanning CI/CD template runs a vulnerability scan automatically and reports findings in the merge request and the project's Security Dashboard; GitHub Container Registry integrates with Docker Scout to surface CVE data for images pushed there. These cover images your team builds and pushes yourself.

They do not cover images you did not build. For a base image or an application image pulled directly from a third-party vendor's registry, you inherit whatever scanning — or lack of it — the vendor applies, and get no visibility into new CVEs yourself. The standard pattern is to **mirror the vendor's image into your own registry** — a scheduled sync or a pull-through registry proxy — and run it through the same scanning and signing pipeline as internally built images. This also protects against the vendor's registry becoming unavailable or a tag being silently overwritten upstream without notice.

Image signing with Cosign

Cosign (part of the Sigstore project) attaches a cryptographic signature to a container image in the registry. The signature proves that the image was produced by a specific build pipeline — not modified after the fact by a compromised registry account or a supply chain attack.

Sign after push in CI:

```
cosign sign --key cosign.key myapp:1.4.2@sha256:<digest>
```

Verify before deploy (or enforce via admission controller):

```
cosign verify --key cosign.pub myapp:1.4.2
```

Keyless signing (using OIDC identity from the CI provider, without managing a key pair) is the current direction for Sigstore — the signature is anchored to the CI job's identity rather than a stored key.

Admission policy enforcement

Image scanning and signing are only valuable if unsigned or unscanned images cannot be deployed. An admission controller enforces this at the cluster level — pods referencing non-compliant images are rejected before they schedule.

Two options are widely used:

- **Kyverno** — a Kubernetes-native policy engine that uses YAML policies and integrates directly with cluster resources. Policies can verify Cosign signatures, require image digests, enforce label conventions, and mutate resources to apply defaults.
- **OPA Gatekeeper** — uses Rego policies (a purpose-built policy language) and a constraint framework. More expressive for complex policies, higher learning curve.

A Kyverno policy that requires images to come from a specific registry and be pinned to a digest:

```
apiVersion: kyverno.io/v1
kind: ClusterPolicy
metadata:
  name: require-digest-and-registry
spec:
  validationFailureAction: Enforce
  rules:
    - name: check-registry-and-digest
      match:
        any:
          - resources:
              kinds: ["Pod"]
      validate:
        message: "Images must be from registry.example.com and pinned to
a digest."
        pattern:
          spec:
            containers:
              - image: "registry.example.com/*@sha256:*"
```

i OVH MKS & STACKIT SKE — enforcing signature verification

Neither platform has a built-in image signing framework like OpenShift's. Enforcement works the same way as on any upstream Kubernetes cluster: install Kyverno or OPA Gatekeeper and apply a policy that verifies the Cosign signature before a pod is admitted. Admission webhooks are a standard Kubernetes API feature, not something either managed control plane restricts — OVH even publishes its own guide for running Kyverno on MKS. STACKIT SKE has no equivalent published guide at the time of writing, but as an upstream-conformant cluster there is no known restriction on registering admission webhooks there either.

i OpenShift Image Streams and signing

OpenShift's Image Streams track image versions and can trigger deployments on new pushes. OpenShift also ships with a built-in image signing framework and supports Cosign verification via the `image.config.openshift.io/cluster` object, where trusted registries and signature stores are configured cluster-wide. On ROSA and ARO, signature policies are inherited from the Red Hat build of OpenShift and enforced for core platform images by default.

Scanning, signing, and admission enforcement cover the chain up to the moment a container starts. None of them tell you anything about what a container actually does once it is running — that is a separate, ongoing concern, covered as part of Day-2 operations in Part 6.

Secrets management

Kubernetes Secrets store sensitive data as base64-encoded values in etcd. **Base64 is encoding, not encryption** — anyone with `kubectl get secret` access to a namespace can read the contents.

Two controls address this:

Enable etcd encryption at rest. Most managed Kubernetes providers offer this as a cluster configuration option (EKS, AKS, GKE). It encrypts Secret data before writing it to etcd, so that access to the etcd data files does not expose secret values. Where it is available, enable it — it is typically a checkbox or a single flag in the cluster configuration.

> ⚠ **OVH MKS & STACKIT SKE — no customer-facing etcd encryption toggle**
>
> Unlike EKS, AKS, and GKE, neither platform exposes etcd encryption at rest as a cluster configuration option. OVH publishes a guide for encrypting etcd with OVHcloud KMS, but it configures the `kube-apiserver` process directly — installing a binary on the apiserver host, setting `--encryption-provider-config`, mounting a Unix socket into the apiserver — none of which a customer can do on a managed control plane. That guide is written for self-hosted clusters running on OVH infrastructure, not for MKS itself. No equivalent customer-facing option is documented for STACKIT SKE either. On both platforms, the second control below — keeping secrets out of Kubernetes Secrets entirely — is the control actually available to you.

Store secrets outside the cluster. External Secrets Operator (ESO) is the standard approach: secrets live in a dedicated secrets manager (HashiCorp Vault, OpenBao, AWS Secrets Manager, Azure Key Vault, GCP Secret Manager) and ESO synchronises them into Kubernetes Secrets on a schedule. The Kubernetes Secret is a local cache — the source of truth is the external store, where access is audited, versioned, and rotated centrally. OpenBao is the Linux-Foundation-hosted fork of Vault, created after HashiCorp relicensed Vault under BSL — worth considering where an OSI-approved open-source license is a requirement.

```yaml
apiVersion: external-secrets.io/v1beta1
kind: ExternalSecret
metadata:
  name: myapp-db-password
  namespace: myapp
spec:
  refreshInterval: 1h
  secretStoreRef:
    name: vault-backend
    kind: ClusterSecretStore
  target:
```

```
    name: myapp-db-password
  data:
    - secretKey: password
      remoteRef:
        key: myapp/db
        property: password
```

SOPS (CNCF Sandbox project, originally from Mozilla) is a simpler alternative for GitOps workflows: it encrypts the values inside a YAML/JSON/ENV file in place, while leaving the structure readable, so the encrypted file can be committed to git. Decryption keys can come from AWS KMS, GCP KMS, Azure Key Vault, age, or PGP — no cluster-specific keypair to manage. Flux and ArgoCD both have native SOPS decryption support, so no separate controller needs to run in the cluster. Producing the encrypted file always starts from the plaintext value — the plaintext has to exist somewhere outside the cluster (a developer's machine, a CI job) at encryption time. ESO has no equivalent step; the plaintext never has to leave the external secrets manager in the first place. The tradeoff versus ESO is that SOPS has no built-in rotation and no integration with an existing secrets manager as the source of truth — the encrypted file in git *is* the source of truth, decrypted on demand.

Regardless of the approach: never commit plaintext secrets to git, never log secret values, and never pass secrets as environment variables that applications might expose via `/env` endpoints or debug output. Mount secrets as files where possible — the application reads the file, and the secret value is not visible in the pod's environment.

> **i OVH MKS & STACKIT SKE — no native pod identity system**
>
> AWS IRSA, GCP Workload Identity, and Azure Managed Identity all allow a pod to assume a cloud IAM identity without storing credentials — the cloud control plane injects a short-lived token into the pod based on its service account annotation. Neither OVH MKS nor STACKIT SKE has a native equivalent.
>
> The practical consequence: workloads that need to access OVH Object Storage, STACKIT S3-compatible storage, or any external cloud API must authenticate with static credentials. A common approach is ESO backed by HashiCorp Vault or the provider's secret store, with rotation on a defined schedule and NetworkPolicy restricting which pods can reach the external API endpoint.
>
> For workloads that require short-lived, automatically rotated credentials without a cloud-managed identity system, **SPIFFE/SPIRE** provides a workload identity layer that runs on any Kubernetes cluster. The SPIRE agent issues short-lived X.509 SVIDs to workloads based on their pod identity, which services can verify without static secrets. SPIRE is a CNCF graduated project and is provider-agnostic — it works identically on OVH MKS, STACKIT SKE, or any other cluster.

Next: Part 6 covers Day-2 operations and GitOps — declarative delivery, drift detection, observability, cost control, and upgrades.

Partly used sources — specific references for this part:

- Using RBAC Authorization — kubernetes.io
- RBAC Good Practices — kubernetes.io
- How to Enforce Policy Management on OVHcloud Managed Kubernetes with Kyverno — OVHcloud
- How to Encrypt Kubernetes ETCD with OVHcloud KMS — OVHcloud
- 17 container security best practices for 2026 — Sysdig (2026)
- Kubernetes security 101: 10 best practices and fundamentals — Sysdig (2026)
- Kubernetes Best Practices for Safer, More Reliable Clusters — Komodor (2026)
- Kubernetes Best Practices I Wish I Had Known Before — Pulumi (2025, updated 2026)
- kubernetes-best-practices — Diego Lima, Apache-2.0